Хотя в .NET управление памятью осуществляется автоматически, понимание того, как именно это делается, сулит немало преимуществ. Вы сможете писать более качественные программы, эффективно взаимодействующие с памятью.

Книга, проверенная командой разработки .NET из «Майкрософт», содержит 25 сценариев поиска и устранения неисправностей, призванных помочь в диагностике сложных проблем при работе с памятью. Приводится также ряд полезных рекомендаций по написанию кода, учитывающих особенности управления памятью и позволяющих избежать типичных ошибок.

В книге представлены:

- теоретические основы автоматического управления памятью;
- глубокое погружение во все аспекты управления памятью в .NET, в т. ч. подробное описание реализации сборщика мусора (GC);
- практические советы по разработке реальных программ;
- правила использования инструментов, относящихся к управлению памятью в .NET;
- эффективные методы работы с памятью, включая типы Span и Memory.

Книга адресована разработчикам программного обеспечения для платформы .NET, архитекторам и специалистам по производительности.

Конрад Кокоса – опытный проектировщик и разработчик ПО, независимый консультант, блогер, сооснователь сайта Dotnetos.org. Главная область его интересов – технологии корпорации «Майкрософт». Он программирует уже больше десяти лет, занимаясь решением проблем производительности и архитектурными головоломками в мире .NET, проектирует приложения и повышает их быстродействие.

Интернет-магазин: www.dmkpress.com

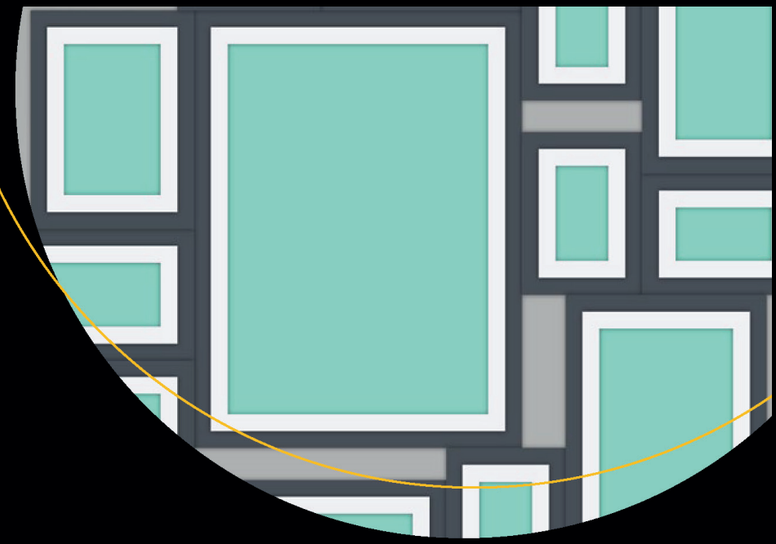
Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

Apress
www.apress.com

ISBN 978-5-97060-800-5

Управление памятью в .NET для профессионалов



Управление памятью в .NET для профессионалов

Написание более качественного, производительного и масштабируемого кода

Конрад Кокоса

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

DOT NEXT

DOT NET .RU

Конрад Кокоса

Управление памятью в .NET для профессионалов

Pro .NET Memory Management

For Better Code, Performance, and Scalability

Konrad Kokosa

Apress®

Управление памятью в .NET для профессионалов

Написание более качественного, производительного и масштабируемого кода

Конрад Кокоса

ДМК Пресс
издательство
Москва, 2020

УДК 004.438.NET
ББК 32.973.26-018.2
К55

Кокоса К.
К55 Управление памятью в .NET для профессионалов. – М.: ДМК Пресс, 2020. – 800 с.: ил.

ISBN 978-5-97060-800-5

Хотя в .NET управление памятью осуществляется автоматически, понимание того, как именно это делается, сулит немало преимуществ. Вы сможете писать более качественные программы, эффективно взаимодействующие с памятью. Книга содержит 25 сценариев поиска и устранения неисправностей, призванных помочь в диагностике сложных проблем при работе с памятью. Приводится также ряд полезных рекомендаций по написанию кода, учитывающих особенности управления памятью и позволяющих избежать типичных ошибок.

Книга адресована разработчикам программного обеспечения для платформы .NET, архитекторам и специалистам по производительности.

УДК 004.438.NET
ББК 32.973.26-018.2

Original English language edition printed on acid-free paper. Copyright © 2018 by Konrad Kokosa. Russian language edition copyright © 2020 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-4842-4026-7 (англ.)
ISBN 978-5-97060-800-5 (рус.)

Copyright © 2018 by Konrad Kokosa
© Оформление, издание, перевод, ДМК Пресс, 2020

Моей любимой жене Юстине,
без которой в моей жизни не случилось бы
ничего сколько-нибудь важного

Содержание

Об авторе ... 15

О технических рецензентах ... 15

Благодарности ... 16

Предисловие ... 18

Введение ... 19

От издательства ... 25

Глава 1. Основные понятия ... 26

Терминология, относящаяся к памяти ... 27
- Статическое выделение ... 33
- Регистровая машина ... 34
- Стек ... 35
- Стековая машина ... 40
- Указатель ... 43
- Куча ... 45

Ручное управление памятью ... 47

Автоматическое управление памятью ... 52
- Распределитель, модификатор и сборщик ... 54

Подсчет ссылок ... 58

Отслеживающий сборщик ... 63
- Этап пометки ... 63
- Этап сборки ... 67

Немного истории ... 71

Резюме ... 73
- Правило 1: учиться, учиться и учиться ... 74

Глава 2. Низкоуровневое управление памятью ... 75

Оборудование ... 76
- Память ... 81
- Центральный процессор ... 84

Операционная система ... 99
- Виртуальная память ... 100
- Большие страницы ... 104
- Фрагментация виртуальной памяти ... 105
- Общая структура памяти ... 105
- Управление памятью в Windows ... 107
- Организация памяти в Windows ... 112

Управление памятью в Linux ... 114
Организация памяти в Linux ... 116
Зависимость от операционной системы ... 117
NUMA и группы процессоров ... 118
Резюме ... 120
Правило 2: избегайте произвольного доступа, отдавайте предпочтение последовательному ... 120
Правило 3: улучшайте пространственную и временную локальность данных ... 121
Правило 4: пользуйтесь продвинутыми средствами ... 121

Глава 3. Измерения памяти ... 123

Измеряйте как можно раньше ... 124
Накладные расходы и вмешательство ... 125
Выборка и трассировка ... 126
Дерево вызовов ... 126
Графы объектов ... 127
Статистика ... 129
Задержка и пропускная способность ... 132
Дампы памяти, трассировка, динамическая отладка ... 133
Среда Windows ... 134
Краткий обзор ... 134
VMMap ... 135
Счетчики производительности ... 136
Трассировка событий для Windows ... 142
Windows Performance Toolkit ... 152
PerfView ... 162
ProcDump и DebugDiag ... 171
WinDbg ... 171
Дизассемблеры и декомпиляторы ... 174
BenchmarkDotNet ... 174
Коммерческие инструменты ... 176
Среда Linux ... 186
Краткий обзор ... 186
Perfcollect ... 187
Trace Compass ... 189
Дампы памяти ... 198
Резюме ... 199
Правило 5: измеряйте GC как можно раньше ... 201

Глава 4. Фундаментальные основы .NET ... 202

Версии .NET ... 202
Детали внутреннего устройства .NET ... 205
Разбираем пример программы ... 208
Сборки и домены приложений ... 213
Забираемые сборки ... 215
Области памяти процесса ... 216
Сценарий 4.1. Сколько места в памяти занимает моя программа? ... 220
Сценарий 4.2. Моя программа потребляет все больше и больше памяти ... 222
Сценарий 4.3. Моя программа потребляет все больше и больше памяти ... 225
Сценарий 4.4. Моя программа потребляет все больше и больше памяти ... 227
Система типов ... 230

Категории типов ... 231
Хранение типов ... 232
Типы значений ... 233
Ссылочные типы ... 241
Строки ... 246
Интернирование строк ... 252
Сценарий 4.5. Моя программа потребляет слишком много памяти ... 257
Упаковка и распаковка ... 259
Передача по ссылке ... 264
Передача по ссылке экземпляра типа значений ... 264
Передача по ссылке экземпляра ссылочного типа ... 265
Локальность типов данных ... 266
Статические данные ... 269
Статические поля ... 269
Внутреннее устройство статических данных ... 270
Резюме ... 273
Структуры ... 274
Классы ... 274

Глава 5. Разделение памяти на части ... 277

Стратегии разделения памяти ... 278
Разделение по размеру ... 279
Куча малых объектов ... 280
Куча больших объектов ... 281
Разделение по времени жизни ... 284
Сценарий 5.1. Как чувствует себя моя программа? Динамика размеров поколений ... 290
Запомненные наборы (Remembered sets) ... 292
Таблицы карт (Card tables) ... 298
Связки карт ... 303
Физическое разделение ... 306
Сценарий 5.2. Утечка памяти в nopCommerce? ... 311
Сценарий 5.3. Растранжиривание кучи больших объектов? ... 319
Анатомия сегментов и кучи ... 321
Повторное использование сегментов ... 324
Резюме ... 326
Правило 11: следите за размерами поколений ... 326
Правило 12: избегайте лишних ссылок в куче ... 327
Правило 13: наблюдайте за использованием сегментов ... 328

Глава 6. Выделение памяти ... 329

Введение в распределение памяти ... 329
Выделение памяти сдвигом указателя ... 330
Выделение памяти из списка свободных блоков ... 337
Создание нового объекта ... 341
Выделение памяти в куче малых объектов ... 343
Выделение памяти в куче больших объектов ... 347
Балансировка кучи ... 351
Исключение OutOfMemoryException ... 353
Сценарий 6.1. Нехватка памяти ... 355
Выделение памяти в стеке ... 356
Избегание выделения памяти ... 358

Явное выделение памяти для ссылочных типов ... 360
Скрытое выделение памяти ... 381
Скрытое выделение памяти в библиотеках ... 389
Сценарий 6.2. Исследование выделения памяти ... 393
Сценарий 6.3. Функции Azure ... 396
Резюме ... 397
Правило 14: избегайте выделения памяти в куче на критических с точки зрения производительности участках программы ... 397
Правило 15: избегайте дорогостоящего выделения памяти в LOH ... 398
Правило 16: по возможности выделяйте память в стеке ... 398

Глава 7. Сборка мусора – введение ... 400

Общее описание ... 400
Пример процесса сборки мусора ... 402
Шаги процесса сборки мусора ... 408
Сценарий 7.1. Анализ использования GC ... 408
Профилирование GC ... 412
Данные для настройки производительности сборки мусора ... 414
Статические данные ... 414
Динамические данные ... 417
Сценарий 7.2. Демонстрация бюджета выделения ... 419
Инициаторы сборки мусора ... 428
Запуск по причине выделения памяти ... 429
Явный запуск ... 430
Сценарий 7.3. Анализ явных вызовов GC ... 433
Запуск по причине нехватки памяти у системы ... 439
Запуск по различным внутренним причинам ... 439
Приостановка движка выполнения ... 440
Сценарий 7.4. Анализ времени приостановки GC ... 442
Выбор поколения для сборки ... 444
Сценарий 7.5. Анализ выбираемых поколений ... 447
Резюме ... 448

Глава 8. Сборка мусора – этап пометки ... 449

Обход и пометка объектов ... 449
Корни – локальные переменные ... 450
Хранилище локальных переменных ... 451
Корни на стеке ... 452
Лексическая область видимости ... 452
Живые стековые корни и лексическая область видимости ... 453
Живые стековые корни с ранней сборкой корней ... 455
Информация для GC (GC Info) ... 461
Закрепленные локальные переменные ... 465
Просмотр стековых корней ... 468
Корни финализации ... 468
Внутренние корни GC ... 469
Корни – описатели GC ... 470
Анализ утечек памяти ... 476
Сценарий 8.1. Утечка памяти в nopCommerce? ... 478
Сценарий 8.2. Нахождение самых популярных корней ... 482
Резюме ... 484

Глава 9. Сборка мусора – этап планирования ... 485
Куча малых объектов ... 486
Заполненные и пустые блоки ... 486
Сценарий 9.1. Дамп памяти с поврежденными структурами ... 491
Таблица кирпичей ... 492
Закрепление ... 494
Сценарий 9.2. Исследование закрепления ... 499
Границы поколений ... 504
Оставление ... 504
Куча больших объектов ... 509
Заполненные и пустые блоки ... 509
Принятие решения об уплотнении ... 511
Резюме ... 512

Глава 10. Сборка мусора – очистка и уплотнение ... 513
Этап очистки ... 513
Куча малых объектов ... 513
Куча больших объектов ... 514
Этап уплотнения ... 515
Куча малых объектов ... 515
Куча больших объектов ... 519
Сценарий 10.1. Фрагментация кучи больших объектов ... 520
Резюме ... 528
Правило 17: следите за приостановкой среды выполнения ... 529
Правило 18: избегайте кризиса среднего возраста ... 529
Правило 19: избегайте фрагментации старого поколения и LOH ... 530
Правило 20: избегайте явной сборки мусора ... 531
Правило 21: избегайте утечек памяти ... 531
Правило 22: избегайте закрепления ... 532

Глава 11. Варианты сборки мусора ... 533
Обзор режимов ... 533
Режим рабочей станции и серверный режим ... 533
Неконкурентный и конкурентный режим ... 535
Конфигурирование режимов ... 536
.NET Framework ... 537
.NET Core ... 537
Приостановка и накладные расходы GC ... 538
Описание режимов ... 540
Неконкурентный режим рабочей станции ... 541
Конкурентный режим рабочей станции (до версии 4.0) ... 542
Фоновый режим рабочей станции ... 544
Неконкурентный серверный режим ... 552
Фоновый серверный режим ... 554
Режимы задержки ... 556
Пакетный режим ... 556
Интерактивный режим ... 557
Режим низкой задержки ... 557
Режим длительной низкой задержки ... 558
Регион без сборки мусора (No GC Region) ... 559
Цели оптимизации задержки ... 562

Выбор варианта GC ... 562
Сценарий 8.1. Проверка параметров GC ... 563
Сценарий 8.2. Измерение и тестирование производительности различных режимов GC ... 566
Резюме ... 573
Правило 23: выбирайте режим GC обдуманно ... 573
Правило 24: помните о режимах задержки ... 574

Глава 12. Время жизни объекта ... 575

Жизненные циклы объекта и ресурса ... 575
Финализация ... 577
Введение ... 577
Проблема ранней сборки корней ... 582
Критические финализаторы ... 585
Внутреннее устройство финализации ... 586
Сценарий 12.1. Утечка памяти из-за финализации ... 593
Воскрешение ... 599
Уничтожаемые объекты ... 603
Безопасные описатели ... 609
Слабые ссылки ... 614
Кеширование ... 618
Паттерн слабых событий ... 620
Сценарий 9.2. Утечка памяти из-за событий ... 626
Резюме ... 629
Правило 25: избегайте финализаторов ... 629
Правило 26: отдавайте предпочтение явной очистке ... 630

Глава 13. Разное ... 632

Зависимые описатели ... 632
Локальная память потока ... 638
Статические поля потока ... 638
Слоты данных потока ... 641
Внутреннее устройство локальной памяти потока ... 642
Сценарии использования ... 649
Управляемые указатели ... 650
Ссылочные локальные переменные ... 651
Возвращаемые ссылочные значения ... 652
Постоянные ссылочные переменные и in-параметры ... 654
Внутреннее устройство ссылочных типов ... 658
Управляемые указатели в C# – ссылочные переменные ... 669
И снова о структурах ... 675
Постоянные структуры ... 676
Ссылочные структуры (byref-подобные типы) ... 677
Буферы фиксированного размера ... 679
Размещение объектов и структур в памяти ... 683
Ограничение unmanaged ... 694
Непреобразуемые типы ... 698
Резюме ... 700

Глава 14. Продвинутые приемы ... 701

Span<T> и Memory<T> ... 701
Span<T> ... 702

Memory<T> ... 716
IMemoryOwner<T> ... 719
Внутреннее устройство Memory<T> ... 723
Рекомендации по работе с Span<T> и Memory<T> ... 725
Класс Unsafe ... 726
Внутреннее устройство Unsafe ... 730
Проектирование, ориентированное на данные ... 731
Тактическое проектирование ... 732
Стратегическое проектирование ... 736
Еще немного о будущем ... 745
Ссылочные типы, допускающие null ... 746
Конвейеры ... 751
Резюме ... 757

Глава 15. Интерфейсы прикладного программирования (API) ... 759
GC API ... 759
Сведения и статистические данные о сборке мусора ... 760
Уведомления GC ... 768
Контроль потребления неуправляемой памяти ... 770
Явная сборка мусора ... 770
Области без GC ... 770
Управление финализацией ... 770
Потребление памяти ... 771
Внутренние вызовы в классе GC ... 772
Размещение CLR ... 773
ClrMD ... 782
Библиотека TraceEvent ... 787
Пользовательский сборщик мусора ... 790
Резюме ... 793

Предметный указатель ... 795

С далекого 2002 года и до 2016 года .NET Framework оставался продуктом с закрытым исходным кодом. С появлением .NET Core разработчики смогли узнать, как платформа работает, какие технические решения были использованы в тех или иных местах. Одна из самых сложных подсистем .NET – это, вне всякого сомнения, сборщик мусора. Это тот элемент платформы, с которым так или иначе сталкивается каждый разработчик. При этом про сборщик мусора было известно довольно мало. Но и то немногое, что было известно, позволяло его успешно использовать. С другой стороны, механизмы управления памятью в .NET являются источником огромного количества мифов и недопониманий. Теперь, когда мы можем изучить исходный код, есть возможность развеять все мифы и понять, как устроена сборка мусора в .NET.

И вот вы держите в руках уникальную книгу. На сотнях страниц автор последовательно излагает всю информацию, необходимую для понимания работы с памятью на платформе .NET. Это книга не только про сборку мусора. Автор уделяет достаточно внимания и аппаратному уровню, и практическим аспектам программирования на платформе .NET, связанным с использованием памяти. Однако основной объем книги посвящен подробному разбору всех тонкостей работы современного сборщика мусора в .NET. Казалось бы, внутреннее устройство сборщика мусора не особенно полезно при разработке бизнес-приложений, однако это совсем не так. Такая сложная система, как сборщик мусора, не обходится без интересных инженерных решений, изучая которые, можно существенно расширить свои знания о платформе .NET и программировании в целом. Кроме того, каждая глава сопровождается полезными практическими выводами, рекомендациями и примерами использования инструментов отладки и профилирования. Книга будет полезна всем, кто интересуется внутренним устройством .NET, а также тем, кто стремится улучшить свои приложения, сделать их оптимальнее. С ростом популярности облачных вычислений вопросы оптимизации (и в конечном счете экономии денег) будут все более актуальны.

Российское сообщество .NET-разработчиков DotNet.Ru с удовольствием трудилось над этой книгой, чтобы достичь наивысшего качества перевода и позволить читателям погрузиться в мир сборки мусора и работы с памятью. Желаем приятного и полезного чтения!

Над переводом работали представители сообщества DotNet.Ru:
Игорь Лабутин
Ирина Ананьева
Максим Шошин
Елизавета Голенок
Евгений Биккинин
Ренат Тазиев
Анатолий Кулаков

DOT
NET
.RU

Каста авторов книг неизменно пользуется особым вниманием со стороны участников на технических конференциях. Доклады этих людей отличаются глубокой проработкой и широкими взглядами. Ведь за плечами у них рукописи, на подготовку которых уходят годы. Это позволяет не только насладиться качественными презентациями, но и провести несколько часов вместе с автором над обсуждением интересных идей.

Типичным представителем прекрасного докладчика, глубокого специалиста, внимательного автора и является Конрад Кокоса. Благодаря невероятной тяге к исследованиям в столь узкой и сложной области мы получили шанс насладиться этим фундаментальным трудом. Десятилетия разработчики воспринимали сборщик мусора как волшебный ящик. Что порождало немало мифов и заблуждений. Конрад стал первым автором, который не только смог понять тонкости работы этого сложнейшего компонента, но и замечательно систематизировал полученные знания, снабдив наработки великолепными схемами.

Эта книга – незаменимое пособие для разработчиков, интересующихся производительностью программ, а также архитектурой сложных, нестандартных систем. Заложенные здесь знания будут еще долгие годы давать пищу для экспериментов и материал для новых докладов.

Анатолий Кулаков,
член программного комитета конференции DotNext

Об авторе

Конрад Кокоса – опытный проектировщик и разработчик программного обеспечения, интересующийся прежде всего технологиями корпорации Майкрософт, но с любопытством поглядывающий и по сторонам. Он программирует уже больше десяти лет, занимаясь решением проблем производительности и архитектурными головоломками в мире .NET, проектирует и повышает быстродействие приложений. Является независимым консультантом, ведет блог на сайте http://tooslowexception.com, выступает с докладами на встречах по интересам и на конференциях, фанатеет от Твиттера (@konradkokosa). Он также отдается своей страсти к преподаванию в области .NET, особенно в части повышения производительности приложений, хорошего стиля кодирования и диагностики. Основатель варшавской группы по производительности веб-приложений. Имеет звание Microsoft MVP в категории «Visual Studio и средства разработки». Сооснователь сайта Dotnetos.org, созданного тремя любителями .NET, организующими туры и конференции, посвященные производительности в .NET.

О технических рецензентах

Дамьен Фоггон – разработчик, писатель и технический рецензент, работающий в области передовых технологий, внес вклад более чем в 50 книг по .NET, C#, Visual Basic и ASP.NET. Сооснователь базирующейся в Ньюкасле группы пользователей NEBytes (адрес в интернете http://www.nebytes.net), сертифицированный профессионал Майкрософт во многих номинациях, начиная с .NET 2.0. Ведет блог по адресу http://blog.fasm.co.uk.

Маони Стивенс – архитектор и главный разработчик сборщика мусора в .NET, работает в Майкрософт. Ведет блог по адресу https://blogs.msdn.microsoft.com/maoni/.

Благодарности

Во-первых, хочу очень, очень сильно поблагодарить свою жену. Без ее поддержки эта книга никогда не появилась бы на свет. Начиная работу над книгой, я даже представить не мог, каким количеством совместно проведенных часов придется пожертвовать. Спасибо тебе за терпение, поддержку и ободрение, которые ты дарила мне на протяжении всего этого времени!

Во-вторых, я хочу выразить благодарность Маони Стивенс за развернутые, точные и бесценные замечания к первым вариантам рукописи. Без тени сомнения могу утверждать, что благодаря ей книга стала лучше. А то, что ведущий разработчик сборки мусора в .NET помогала при написании этой книги, – само по себе награда для меня! Большое спасибо и другим членам команды .NET, принимавшим участие в рецензировании некоторых частей книги, привлечь которых удалось при помощи Маони. Перечисляю их в соответствии с количеством вложенного труда: Стивен Тоуб (Stephen Toub), Джаред Парсонс (Jared Parsons), Ли Калвер (Lee Culver), Джош Фри (Josh Free), Омар Тофик (Omar Tawfik). Спасибо также Марку Пробсту (Mark Probst) из компании Xamarin, который отредактировал замечания об исполняющей среде Mono. Особая благодарность Патрику Дассуду (Patrick Dussud), «отцу .NET GC», за то, что он нашел время отрецензировать исторический обзор создания CLR.

В-третьих, я признателен Дамьену Фоггону, техническому редактору от издательства Apress, который вложил столько труда в редактирование всех глав. Его бесценный опыт авторской и издательской деятельности помог сделать изложение более понятным и последовательным. Не раз и не два я поражался точности комментариев и предложений Дамьена!

Очевидно, что я благодарен всему коллективу издательства Apress, без которого книга вообще не вышла бы в свет. Отдельное спасибо Лауре Берендсон (редактор-консультант), Нэнси Чен (редактор-координатор) и Джоан Маррей (старший редактор), которые поддерживали меня и терпели бесконечные переносы сроков. В течение какого-то периода даже само упоминание даты сдачи окончательного варианта в наших разговорах было под запретом! Я также благодарен Гвенан Спиринг, с которой начинал работать над книгой, но не смог закончить, потому что она ушла из Apress.

Я очень благодарен сообществу .NET в Польше и во всем мире за идеи, которые черпал из многочисленных презентаций, статей и постов, за ободрение и поддержку и бесконечные вопросы о том, как продвигается книга. Особенно я признателен следующим лицам (перечислены в алфавитном порядке): Мачей Анисерович (Maciej Aniserowicz), Аркадиуш Бенедикт (Arkadiusz Benedykt), Себастьян Гебский (Sebastian Gębski), Михал Гжегоржевский (Michał Grzegorzewski), Якуб Гутковский (Jakub Gutkowski), Павел Климчик (Paweł Klimczyk), Шимон Кулец (Szymon Kulec), Павел Лукашик (Paweł Łukasik), Алисия Мусяу (Alicja Musiał), Лукаш Ольбромский (Łukasz Olbromski), Лукаш Пыржик (Łukasz Pyrzyk), Бартек Сокыл (Bartek Sokył), Себастьян Солница (Sebastian Solnica), Павел Срочиньский (Paweł Sroczyński), Ярек Стадницкий (Jarek Stadnicki), Пётр Стапп (Piotr Stapp),

Михал Сливонь (Michał Śliwoń), Шимон Варда (Szymon Warda) и Артур Винченчак (Artur Wincenciak), все обладатели звания MVP, и многие другие. Искренне прошу прощения у тех, кого не упомянул, огромное спасибо всем, кто в этом нуждается. Перечислить всех просто невозможно. Все вы вдохновляли и поддерживали меня.

Хочу также поблагодарить всех опытных авторов, которые нашли время, чтобы поделиться советом о том, как писать книги, в том числе Тэда Ньюэрда (Ted Neward) (http://blogs.tedneward.com/) и Джона Скита (Jon Skeet) (https://codeblog.jonskeet.uk), хотя готов побиться об заклад, что они сами этих разговоров не помнят! Анджей Кшивда (Andrzej Krzywda) (http://andrzejonsoftware.blogspot.com) и Гынваэль Кольдвинд (Gynvael Coldwind) (https://gynvael.coldwind.pl) также дали мне много советов по написанию и публикации книги.

Далее я благодарю создателей всех тех замечательных инструментов и библиотек, которыми пользовался при написании этой книги: Андрея Щелкина, автора SharpLab (https://sharplab.io), Андрея Акиньшина, автора BenchmarkDotNet (https://benchmarkdotnet.org), и Адама Ситника, отвечающего за ее сопровождение, Сергея Теплякова, автора ObjectLayoutInspector (https://github.com/SergeyTeplyakov/ObjectLayoutInspector), 0xd4d, анонимного создателя dnSpy (https://github.com/0xd4d/dnSpy), Сашу Голдштейна, автора множества полезных инструментов (https://github.com/goldshtn), а также создателей таких великолепных программ, как PerfView и WinDbg (и их расширений, относящихся к .NET).

Я очень признателен своему бывшему работодателю Банку Миллениум, который оказывал мне помощь и поддержку, когда я только приступал к написанию книги. Наши пути разошлись, но я всегда буду помнить, что именно там я начал писать, вести блог и выступать с докладами. Большое спасибо всем моим тогдашним коллегам за ободрение и вопросы «как продвигается книга?» – это здорово мотивирует.

Еще раз спасибо всем анонимным пользователям Твиттера, которые откликались на мои обзоры книг и подсказывали, что интересно, полезно и ценно для нашей семьи .NET, а что – не очень.

И наконец, общее спасибо всей моей семье и друзьям, которым я не мог уделять достаточно внимания, пока был занят книгой.

Предисловие

Когда я вошла в команду разработчиков общеязыковой среды выполнения CLR (исполняющей среды .NET) – тому уже больше десяти лет, – я даже не думала, что компонент под названием «сборщик мусора» (Garbage Collector – GC) станет темой, над которой я буду размышлять большую часть времени, когда не сплю. Среди тех, с кем я работала, был и Патрик Дассуд – архитектор и разработчик CLR GC с момента его создания. Понаблюдав за моей работой на протяжении нескольких месяцев, он передал мне факел, и я стала вторым лицом, отвечающим только за разработку GC для CLR.

Так началось мое путешествие в мир GC. Вскоре я открыла для себя очарование мира сборки мусора – я была поражена сложностью и обширностью возникающих задач и полюбила находить для них эффективные решения. Поскольку количество различных сценариев использования CLR росло, как и число пользователей, а память – один из самых важных аспектов производительности, постоянно возникали новые проблемы в части управления памятью. Когда я начинала работать, редкостью было встретить кучу GC размером хотя бы 200 МБ, сегодня и 20 ГБ не считается чем-то из ряда вон выходящим. Некоторые из самых больших рабочих нагрузок в мире работают на CLR. Как лучше управлять памятью в этих условиях – без сомнения, животрепещущая проблема.

В 2015-м мы раскрыли исходный код CoreCLR. Когда об этом было объявлено, сообщество спрашивало, будет ли код GC исключен из репозитория CoreCLR, – справедливый вопрос, поскольку в нашем GC было много инновационных механизмов и политик. Ответом было решительное «нет», в репозитории находится тот самый код, который используется в CLR. Безусловно, это привлекло некоторых любознательных личностей. Год спустя я с восторгом узнала, что один из наших пользователей планирует написать книгу, посвященную исключительно нашему GC. Когда технологический евангелист из нашего польского отделения спросил, смогу ли я отредактировать книгу Конрада, конечно, я согласилась!

Получая главы от Конрада, я поняла, что он изучил наш код GC со всей тщательностью. Я была поражена детальностью изложения. Конечно, вы можете самостоятельно собрать CoreCLR и пройтись по коду GC в пошаговом режиме. Но эта книга определенно упростит вам задачу. А поскольку важной частью читательской аудитории являются пользователи GC, Конрад включил много материала, который поможет лучше понять поведение GC и способы кодирования, повышающие эффективность его использования. Кроме того, в начале книги имеется основополагающая информация о памяти, а ближе к концу – обсуждение характера использования памяти в различных библиотеках. Я полагаю, что Конрад нашел идеальный баланс между введением в GC, описанием его внутренних механизмов и применения.

Если вы пользуетесь .NET и думаете об эффективном использовании памяти или просто интересуетесь тем, как устроен .NET GC, то эта книга для вас. Надеюсь, что вы получите от чтения не меньшее удовольствие, чем я от редактирования.

Маони Стивенс,
июль 2018

Введение

В информатике память была всегда – перфокарты, магнитные ленты и, наконец, современные высокотехнологичные микросхемы динамических ОЗУ, DRAM. И так будет всегда – быть может, в форме голографических чипов из научно-фантастических романов или еще более удивительных вещей, которые мы даже представить себе сейчас не можем. И конечно, тому есть основательные причины. Хорошо известно определение компьютерных программ как объединения алгоритмов и структур данных. Мне эта формулировка очень нравится. Наверное, все хотя бы раз слышали о книге Никласа Вирта «Алгоритмы + структуры данных = программы» (Prentice Hall, 1976), где она впервые была введена в обращение.

С момента оформления программной инженерии как дисциплины вопрос управления памятью считался приоритетным. Уже конструкторы самых первых вычислительных машин вынуждены были задумываться о памяти для алгоритмов (программного кода) и структур данных (программных данных). Всегда уделялось большое внимание тому, как эти данные загружаются и где сохраняются для дальнейшего использования.

В этом отношении программная инженерия и управление памятью всегда были тесно связаны – так же, как программная инженерия и алгоритмы. И я полагаю, так всегда и будет. Память – ограниченный ресурс, и всегда таковым останется. Поэтому в той или иной степени память будет занимать умы будущих разработчиков. Если ресурс ограничен, то всегда возможны ошибки или неправильное использование, приводящие к его истощению. Память – не исключение из этого правила.

Но при всем при том в управлении памятью есть один постоянно меняющийся аспект – объем. Первые разработчики, а правильнее было бы называть их инженерами, знали о каждом бите в своих программах. В то время в их распоряжении было всего несколько килобайтов памяти. Каждое десятилетие эта величина росла, и сегодня мы живем в эпоху гигабайтов, а в дверь уже стучатся терабайты и петабайты. Вместе с увеличением объема уменьшается время доступа, что позволяет обработать все эти данные за разумное время. Но хотя можно сказать, что память стала быстрой, непритязательные алгоритмы управления памятью, которые пытаются обработать гигабайты данных безо всяких оптимизаций и сложных настроек, обречены на провал. Связано это прежде всего с тем, что время доступа к памяти снижается медленнее, чем вычислительная мощность процессоров, которые с этой памятью работают. Необходимо принимать специальные меры, чтобы доступ к памяти не стал узким местом, ограничивающим мощность современных процессоров.

Это не только придает управлению памятью первостепенную важность, но и делает его по-настоящему чарующей частью информатики. А автоматическое управление памятью – вещь еще более интересная. Оно отнюдь не сводится к пожеланию «освободим-ка неиспользуемые объекты». Что, как и когда – эти простые аспекты управления памятью делают его непрерывным процессом совершенствования старых и изобретения новых алгоритмов. Бесчисленные научные

статьи и диссертации посвящены вопросу о том, как организовать автоматическое управление памятью оптимальным образом. На таких мероприятиях, как Международный симпозиум по управлению памятью (International Symposium on Memory Management – ISMM), подводится итог достижениям в этой области за год: сборке мусора, динамическому выделению, взаимодействиям с исполняющей средой, компиляторами и операционными системами. А затем академические исследования превращаются в коммерческие и открытые продукты, которыми мы пользуемся каждый день.

.NET – идеальный пример управляемой среды, в которой все эти сложности глубоко скрыты и представлены разработчикам в виде приятной и готовой к использованию платформы. Действительно, мы пользуемся всем этим, не подозревая о глубинной сложности, что само по себе является крупным достижением .NET. Однако чем сильнее наша программа нуждается в высокой производительности, тем меньше шансов обойтись без знания о том, как и почему все работает внутри .NET. А что касается меня лично, так мне просто интересно, как устроены вещи, которыми мы пользуемся ежедневно!

Я написал эту книгу такой, какой хотел бы прочитать ее много лет назад, когда только начинал путешествие в мир производительности и диагностики .NET. То есть она начинается не с типичного введения в организацию кучи или стека и не с описания нескольких поколений. Вместо этого я решил начать с самых основ управления памятью. Иными словами, я старался писать так, чтобы вы почувствовали, сколь интересна эта тема, а не ограничиваться констатациями типа «вот это сборщик мусора .NET, и делает он то-то и то-то». Информация не только о том, что, но и о том, как, а главное – почему, поможет вам по-настоящему понять, что происходит за кулисами управления памятью в .NET. А тогда все, что вы будете читать на эту тему в будущем, станет яснее. Я пытался вооружить вас знаниями не только о .NET, особенно в первых двух главах. Это способствует более глубокому проникновению в тему, что часто оказывается полезно в применении к другим задачам программной инженерии (благодаря лучшему пониманию алгоритмов, структур данных и просто добротной инженерной работы).

Я хотел написать книгу, так чтобы ее было приятно читать любому разработчику для .NET. Каким бы ни был ваш предшествующий опыт, что-то интересное для себя вы здесь найдете. Мы начинаем с азов, но младшие программисты довольно быстро получат возможность глубже заглянуть в устройство .NET. Более опытным программистам будут интересны различные детали реализации. И кроме того, каждый, вне зависимости от опытности, получит пользу от изучения практических примеров кода и диагностики проблем.

Таким образом, знания, полученные из этой книги, должны помочь вам писать более качественный код, в котором связанные с памятью средства используются не слепо, а с полным пониманием дела. Это также повысит производительность и масштабируемость приложений – чем сильнее код ориентирован на правильную работу с памятью, тем меньше шансов на возникновение узких мест и неоптимальное использование ресурсов. Я надеюсь, что прочтение книги оправдает подзаголовок «Написание более качественного, производительного и масштабируемого кода».

Я также надеюсь, что все это сделает книгу более общей и долговечной, чем простое описание текущего состояния .NET Framework и его внутреннего устрой-

ства. Как бы ни развивалась платформа .NET в будущем, я верю, что большая часть изложенного в этой книге материала, наверное, сохранит актуальность еще долго. Даже если какие-то детали реализации изменятся, почерпнутые из этой книги знания позволят вам без труда понять их. Просто потому, что основополагающие принципы меняются не так быстро. Желаю вам приятного путешествия по огромному и увлекательному миру автоматического управления памятью!

Вместе с тем я хотел бы подчеркнуть, что некоторые вещи представлены в книге недостаточно полно. Тема управления памятью, хотя и кажется очень узкой и специальной, на удивление широка. И хотя я затронул много вопросов, иногда они из-за недостатка места изложены не так подробно, как мне хотелось бы. Но даже при таких ограничениях книга заняла больше 1000 страниц! К числу опущенных тем относятся, например, исчерпывающие ссылки на другие управляемые среды (Java, Python, Ruby и т. п.). Приношу также извинения поклонникам F# за недостаточное количество ссылок на материалы по этому языку. Просто мне не хватило страниц для серьезного описания, а публиковать что-то менее полное не хотелось. Я хотел бы также уделить больше внимания среде Linux, но на момент написания книги эта тема была еще совсем новой, а инструментов не хватало, поэтому я привел лишь краткие предложения в главе 3 (и по той же причине полностью проигнорировал мир macOs). Понятно, что я вынужден был опустить большую часть вопросов производительности в .NET, не связанных напрямую с памятью, например многопоточность.

И еще – хотя я приложил все усилия, чтобы показать практические применения обсуждаемой теории и технических приемов, не всегда возможно сделать это без примеров. Практических применений попросту слишком много. Поэтому я ожидаю, что читатель будет работать с книгой внимательно, всесторонне осмысливать каждую тему и применять полученные знания в повседневной работе. Поймите, как что-то устроено, – и тогда сможете это использовать!

Особенно это относится к так называемым сценариям. Прошу иметь в виду, что все включенные в книгу сценарии приведены в иллюстративных целях. Их код сведен к абсолютному минимуму, чтобы было проще продемонстрировать глубинную причину какой-то одной проблемы. У наблюдаемого неправильного поведения могут быть и другие причины (например, есть много способов заметить утечку управляемой памяти). Сценарии были подготовлены так, чтобы помочь в иллюстрации подобных проблем на одном примере, поскольку, очевидно, невозможно рассмотреть все мыслимые причины в одной книге. К тому же в реальной ситуации расследование может быть осложнено кучей отвлекающих данных и ложными путями поиска. Зачастую не существует единственного способа решения описанных задач, и тем не менее есть много способов найти истинную причину проблемы в процессе анализа. Поэтому поиск причин ошибки становится гибридом чисто инженерной задачи с искусством, подкрепленным интуицией. Обратите внимание, что сценарии иногда ссылаются друг на друга, чтобы не повторять снова и снова одни и те же шаги, рисунки и описания.

Я специально воздерживался от упоминания различных случаев и источников проблем, характерных для конкретных технологий. Они просто … ну, слишком характерны для конкретных технологий. Если бы я писал эту книгу 10 лет назад, то, наверное, вынужден был бы перечислить типичные сценарии утечки памяти в ASP.NET WebForms и WinForms. Несколько лет назад? ASP.NET MVC, WPF, WCF,

WF… А теперь? ASP.NET Core, EF Core, Azure Functions, что еще? Надеюсь, вы поняли, о чем я. Такие знания слишком быстро устаревают. Книга, под завязку набитая примерами утечек памяти в WCF, сегодня вряд ли кому-то будет интересна. Я очень люблю слова: «Дай человеку рыбу – и он будет сыт один день. Дай человеку удочку – и он будет сыт всю жизнь». Поэтому все знания, изложенные в этой книге, все сценарии служат одной цели – научить человека ловить рыбу. Все проблемы, вне зависимости от технологии, можно диагностировать единообразно, если вы обладаете знаниями и понимаете, как их применять.

Все это также предъявляет большие требования к читателю, поскольку подчас текст изобилует деталями и, быть может, немного перегружен информацией. Но, несмотря ни на что, я советую читать медленно и вдумчиво, не поддаваясь искушению что-то просмотреть по диагонали. Так, чтобы извлечь из книги максимум пользы, следует внимательно изучать код и рисунки (а не просто бросить на них мимолетный взгляд, считая, что все очевидно и можно без ущерба пропустить).

Мы живем в замечательное время, когда исходный код среды выполнения CoreCLR открыт. Это дает качественно новые возможности для изучения и понимания CLR. Не осталось места ни догадкам, ни тайнам. Все есть в коде, все можно прочитать и понять. Поэтому мои исследования внутренних механизмов управления памятью основаны на коде GC из CoreCLR (общем с .NET Framework). Я потратил бесчисленные дни и недели, изучая огромный объем отлично сделанной инженерной работы. Я нахожу ее великолепной и полагаю, что найдутся и другие люди, которые захотят изучить знаменитый файл gc.cpp, насчитывающий десятки тысяч строк кода. Но кривая обучения будет очень крутой. Чтобы помочь вам, я часто оставляю подсказки – с чего начать изучение кода CoreCLR, относящегося к описываемым темам. Ничто не мешает вам еще глубже изучить вопрос, начав с предложенных мной мест в gc.cpp!

Прочитав эту книгу, вы научитесь:

- писать для .NET код с учетом моментов, относящихся к производительности и памяти. Все примеры в книге написаны на C#, но, разобравшись с ними и овладев описанным инструментарием, вы сможете применить полученные знания также к коду на F# и VB.NET;
- диагностировать типичные проблемы, связанные с управлением памятью в .NET. Поскольку большая часть методов основана на данных ETW/LTTng и расширении SOS, они равно применимы к Windows и Linux (правда, в Windows набор инструментов гораздо богаче);
- понимать, как CLR работает в части управления памятью. Я уделил много внимания объяснению не только того, как все устроено, но и почему устроено именно так;
- читать и понимать массу интересных обсуждений, посвященных C# и исполняющей среде CLR на GitHub, и даже принимать в них участие;
- читать код GC в CoreCLR (особенно файл gc.cpp), понимая достаточно, чтобы заниматься дальнейшими исследованиями;
- читать и понимать публикации о сборке мусора и управлении памятью в других средах, например Java, Python и Go.

Теперь конкретно о содержании книги. Глава 1 представляет собой очень общее теоретическое введение в управление памятью, почти без ссылок на особенности .NET. Глава 2 – общее введение в управление памятью на уровне аппаратных

средств и операционной системы. Обе главы можно рассматривать как важное, но все-таки факультативное введение. Они дают полезный, более широкий взгляд на тему и пригодятся в остальной книге. Хотя я, конечно, настоятельно рекомендую прочитать их, можете обойтись без этого, если очень спешите или интересуетесь только сугубо практическими, относящимися к .NET вопросами. Замечание для хорошо осведомленных читателей – даже если вы думаете, что темы, изложенные в первых двух главах, вам хорошо знакомы, пожалуйста, прочитайте их. Я старался включить не только очевидную информацию, но и кое-что такое, что может показаться вам интересным.

Глава 3 целиком посвящена измерениям и разнообразным инструментам (некоторые из которых используются в книге очень часто). Текст состоит в основном из перечня инструментов и информации о том, как с ними работать. Если вас больше интересует теоретическая часть книги, можете бегло пролистать эту главу. С другой стороны, если вы планируете интенсивно использовать полученные знания для диагностики проблем, то, наверное, будете часто возвращаться к ней.

В главе 4 мы впервые начнем предметный разговор о .NET, хотя пока еще общего характера – с целью понять некоторые относящиеся к теме внутренние механизмы, в т. ч. систему типов (включая различия между значимыми и ссылочными типами), интернирование строк и статические данные. Если вы очень торопитесь, то имеет смысл начать чтение отсюда. В главе 5 рассматривается первый по-настоящему относящийся к памяти вопрос – как организована память в приложениях для .NET, включая понятия кучи малых и больших объектов, а также сегментов. В главе 6 обсуждение внутренних механизмов памяти продолжится, речь пойдет исключительно о выделении памяти. Удивительно, что такая длинная глава может быть посвящена столь простому, с теоретической точки зрения, вопросу. Важной и большой частью главы является описание различных источников выделения – а точнее того, как их избегать.

Главы с 7 по 10 содержат описание основных механизмов работы GC в .NET, с практическими примерами и выводами, вытекающими из полученных знаний. Чтобы не вываливать на бедного читателя всю информацию одновременно, в этих главах описывается самый простой вариант GC – неконкурентный, для рабочей станции (Workstation Non-Concurrent). А вот глава 11 посвящена описанию всех остальных вариантов с подробными соображениями о том, какой выбрать. Глава 12 завершает часть книги, посвященную GC, описанием трех важных механизмов: финализации, уничтожаемых объектов и слабых ссылок.

Последние три главы составляют «продвинутую» часть книги в том смысле, что в них обсуждается, как работают механизмы, выходящие за пределы базового управления памятью в .NET. Так, в главе 13 рассматривается вопрос об управляемых указателях и далее о структурах (включая недавно добавленные ссылочные структуры). В главе 14 много внимания уделено типам и техническим приемам, которые в последнее время приобретают все более широкую популярность, в частности `Span<T>` и `Memory<T>`. Имеется также раздел, посвященный не столь известной теме проектирования, ориентированного на данные, и несколько слов сказано об ожидаемых новшествах в C# (ссылочные типы, допускающие и не допускающие null, и конвейеры (pipelines)). В главе 15, последней, описываются различные способы управления и мониторинга работы GC из кода программы, включая API класса GC, размещение CLR и библиотеку ClrMD.

Большая часть приведенных в книге листингов доступна в сопроводительном репозитории на GitHub по адресу https://github.com/Apress/pro-.net-memory. Он организован по главам, и для большинства глав включено два решения: одно для проведенных тестов, а второе содержит прочие листинги. Обратите внимание, что хотя для включенных в книгу проектов имеются листинги, часто доступный вам код не ограничивается ими. Если вы хотите поэкспериментировать с конкретным листингом, проще всего найти его по номеру. Но я также рекомендую познакомиться с проектами по различным темам для лучшего понимания предмета.

Не так уж много есть важных соглашений, о которых я хотел бы здесь упомянуть. Самое существенное – различать две главные концепции, встречающиеся далее в этой книге:

- сборка мусора (GC) – понятный в общих чертах процесс освобождения более не нужной памяти;
- сборщик мусора (GC) – конкретный механизм реализации сборки мусора, очевидно, в контексте .NET[1].

Эта книга в значительной мере автономная, ссылок на другие материалы и книги в ней немного. Но понятно, что в мире существует великое множество знаний, и мне бы надо было очень часто ссылаться на различные источники. Вместо этого я перечислю книги, которые, на мой взгляд, могут служить дополнительными источниками информации:

- Sasha Goldshtein, Dima Zurbalev, Ido Flatow «Pro .NET Performance» (Apress, 2012);
- Jeffrey Richter «CLR via C#» (Microsoft Press, 2012);
- Ben Watson «Writing High-Performance .NET Code» (Ben Watson, 2014);
- Mario Hewardt «Advanced .NET Debugging» (Addison-Wesley Professional, 2009);
- Serge Lidin «.NET IL Assembler» (Microsoft Press, 2012);
- David Stutz «Shared Source CLI Essentials» (O'Reilly Media, 2003);
- «Book Of The Runtime», открытая документация, разрабатываемая параллельно самой исполняющей среде, доступна по адресу https://github.com/dotnet/coreclr/blob/master/Documentation/botr/README.md.

Огромный объем информации доступен также в различных блогах и статьях. Но я не буду загромождать их перечнем страницы этой книги, а переадресую на замечательный репозиторий https://github.com/adamsitnik/awesome-dot-net-performance, поддерживаемый Адамом Ситником.

[1] В оригинале употребляются термины GC и «the GC». Передать это различие в русском языке невозможно, но обычно из контекста понятно, о чем идет речь. – *Прим. перев.*

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Скачивание исходного кода

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Apress очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Глава 1

Основные понятия

Начнем с простого, но очень важного вопроса. Когда следует задумываться об управлении памятью в .NET, если оно полностью автоматизировано? И надо ли задумываться вообще? Вы, наверное, понимаете, что раз уж я написал такую книгу, то настоятельно рекомендую помнить о памяти в любой ситуации. Это просто вопрос нашего профессионализма. Следствие того, как мы строим свою работу. Пытаемся ли мы сделать наилучшим образом или просто сделать? Если мы думаем о качестве своей работы, то недостаточно, чтобы произведение нашего труда просто работало. Важно еще, и как оно работает. Оптимально ли оно использует процессор и память? Удобно ли оно для сопровождения и тестирования, открыто ли для расширения и закрыто для модификации? Отвечает ли наш код принципам SOLID? Полагаю, что такие вопросы отличают начинающего от опытного программиста. Первый думает в основном о том, чтобы код работал, и не печется о вышеупомянутых нефункциональных аспектах своей деятельности. У второго достаточно «ментальной вычислительной мощности», чтобы задумываться о качестве работы. Думается мне, что все хотели бы быть такими. Но, конечно, это не тривиально. Писать элегантный код, который не содержит ошибок и удовлетворяет всем нефункциональным требованиям, действительно трудно.

Но верно ли, что только из стремления к мастерству стоит приобретать глубокие знания об управлении памятью в .NET? Повреждения памяти, ведущие к исключению `AccessViolationException`, случаются крайне редко[1]. Неконтролируемый рост потребления памяти – вроде бы тоже нечастое событие. Так есть ли повод для беспокойства? Поскольку среда выполнения .NET тщательно реализована Майкрософт, то, к счастью, нам не приходится много думать о памяти. С другой стороны, при анализе проблем с производительностью крупных приложений на базе .NET проблемы потребления памяти всегда стояли на одном из первых мест. Станет ли проблемой утечка памяти после нескольких дней непрерывной работы? В интернете можно найти забавный мем об утечке памяти в программном обеспечении некой боевой ракеты, которую не стали устранять, потому что ракета раньше поразит цель, чем кончится память. Является ли наша система такой ракетой одноразового использования? Можем ли мы сказать, чревато автоматизированное управление памятью в нашем приложении серьезными накладными

---

[1] `AccessViolationException` и другие повреждения памяти часто возбуждаются механизмом автоматического управления памятью не потому, что он является причиной, а потому, что это самый крупный компонент среды, относящийся к памяти. Поэтому именно у него наибольшие шансы обнаружить противоречивое состояние памяти.

расходами или нет? Быть может, мы могли бы обойтись всего двумя серверами вместо десяти? И кстати, память не бесплатна даже во времена бессерверных облачных вычислений. Один из примеров – служба Azure Functions, которая тарифицируется на основе показателя «гигабайт-секунды» (GB-s). Он вычисляется путем умножения среднего объема памяти в гигабайтах на время в секундах, необходимое для выполнения конкретной функции. Потребление памяти самым непосредственным образом отражается на сумме счета.

Так или иначе, мы приходим к выводу, что понятия не имеем, с чего начинать поиск причины и что измерять. И вот тогда-то приходит осознание того, что было бы полезно разбираться во внутренних механизмах работы приложения и исполняющей его среды.

Чтобы глубже понять, как устроено управление памятью в .NET, лучше начать с азов. Не имеет значения, новичок вы или крутейший программист. Я рекомендую вместе со мной познакомиться с теоретическим введением, составляющим содержание этой главы. Тогда у нас будет общий уровень знаний и понимания, необходимый для чтения книги. Чтобы не скатываться в сухую теорию, я иногда буду давать отсылки к конкретным технологиям. Заодно у нас будет шанс кое-что узнать об эволюции программного обеспечения. Этот экскурс хорошо ложится на развитие концепций, относящихся к управлению памятью. Мы также отметим несколько любопытных фактов, которые, надеюсь, будут вам интересны. Знание истории неизменно является одним из лучших способов шире взглянуть на вопрос.

Но не пугайтесь – это не книга об истории. Я не буду приводить биографии всех ученых, занимавшихся разработкой алгоритмов сборки мусора, начиная с 1950 года. Знания древней истории тоже не понадобятся. Тем не менее я льщу себя надеждой, что вам будет интересно узнать, как эта дисциплина развивалась и где мы сейчас находимся. Это позволит нам сравнить принятый в .NET подход со многими другими языками и исполняющими средами, о которых вы, вероятно, наслышаны.

Терминология, относящаяся к памяти

В самом начале будет полезно привести ряд очень важных определений, без которых трудно представить себе обсуждение вопроса о памяти.

- *Бит* – самая маленькая единица информации в компьютерных технологиях. Представляет два возможных состояния, которые обычно интерпретируются как числовые значения 0 и 1 или логические значения true и false. В главе 2 мы вкратце опишем, как в современных компьютерах хранятся одиночные биты. Для представления больших чисел необходимо использовать комбинации битов, кодирующие двоичное значение числа, как будет описано ниже. Если требуется выразить размер данных в битах, употребляется буква b.
- *Двоичное число* – целое число, представленное в виде последовательности битов. Каждый последующий бит описывает вклад соответствующей степени 2 в сумму данного значения. Например, для представления числа 5 можно использовать три последовательных бита со значениями 1, 0 и 1,

поскольку 1×1 + 0×2 + 1×4 равно 5. С помощью *n* бит можно представить натуральные числа до 2^*n* – 1 включительно. Часто один дополнительный бит используют для представления знака, чтобы различать положительные и отрицательные значения. Есть также другие, более сложные способы двоичного кодирования числовых значений, особенно чисел с плавающей точкой.

- *Двоичный код* – последовательность битов может представлять не только числовые значения, но и другие данные, например символы текста. Каждому символу сопоставляется своя последовательность битов. Самой простой кодировкой, бывшей самой популярной на протяжении многих лет, является ASCII, которая использовала 7-битовый двоичный код для представления букв и других символов. Есть и другие важные двоичные коды, например *коды операций*, кодирующие команды, выполняемые компьютером.
- *Байт* – исторически так называлась последовательность битов для двоичного кодирования одного символа текста. Чаще всего встречаются 8-битовые байты, хотя размер зависит от архитектуры компьютера и может отличаться для разных архитектур. Из-за такой неоднозначности существует более точный термин – *октет*, означающий блок данных размером точно 8 бит. Но все же 8-битовый байт – это стандарт де-факто, и в таком качестве он используется для задания размера данных. В настоящее время практически невозможно встретить архитектуру, в которой размер байта был бы другим. Если требуется выразить размер данных в байтах, употребляется буква В.

При задании размера данных мы используем префиксы, определяющие порядок величины. Тут постоянно возникает путаница, поэтому пояснения будут нелишними. Такие повсеместно распространенные префиксы, как кило-, мега- и гига-, обозначают степень 1000. *Кило-* – это 1000 (обозначается буквой k), *мега-* – 1 миллион (буква М) и т. д. Но есть и другой, не менее популярный подход – выражать порядок величины степенями 1024. В таком случае употребляют префиксы *киби-* – 1024 (обозначается Ki), *меби-* – 1024*1024 (обозначается Mi), *гиби-* (Gi) – 1024*1024*1024 и т. д. Это вносит неразбериху. Говоря «1 гигабайт», человек может иметь в виду 1 миллиард байт (1 ГБ) или 1024^3 байт (1 ГиБ) – в зависимости от контекста. На практике редко кому интересна точная интерпретация этих префиксов. Емкость модулей памяти для компьютеров практически всегда выражается в гигабайтах (ГБ), хотя на самом деле имеются в виду гибибайты (ГиБ), а в спецификации емкости жестких дисков дело обстоит ровно наоборот. Даже в стандарте JEDEC 100B.01 «Термины, определения и буквенные символы, относящиеся к микрокомпьютерам, микропроцессорам и интегральным схемам памяти» общепринятые буквы K, M и G определяются как умножение на 1024, и их употребление явно не осуждается. В таких ситуациях нам остается только прибегнуть к здравому смыслу и интерпретировать префиксы в зависимости от контекста.

Мы все уже привыкли к употреблению таких терминов, как RAM (ОЗУ[1]) или постоянная память, для обозначения памяти, установленной в компьютере. Даже

---

[1] На самом деле RAM (Random Access Memory) переводится как «запоминающее устройство с произвольной выборкой (ЗУПВ)», но эта аббревиатура уже давно уступила месту ОЗУ (оперативное запоминающее устройство), хотя это и не совсем правильно. – *Прим. перев.*

умные часы теперь оснащаются ОЗУ емкостью 8 ГиБ. Легко забыть, что в первых компьютерах такой роскоши не было. Можно даже сказать, что они вообще не оснащались памятью. Беглый обзор истории развития компьютеров позволит по-другому взглянуть на саму память. Начнем с самого начала.

Следует иметь в виду, что вопрос о том, какое устройство можно назвать «самым первым компьютером», весьма спорный. И не менее трудно назвать имя единственного «изобретателя компьютера». Проблема в определении того, что такое «компьютер». Поэтому не будем вдаваться в бесконечные споры о том, что и кто был первым, а просто посмотрим, что предлагали программистам прежние машины, хотя до появления самого слова *программист* должно было пройти еще много лет. Поначалу они назывались *кодировщиками*, или *операторами*.

Следует подчеркнуть, что вычислительные машины, которые можно было бы назвать первыми компьютерами, были не электронными, а электромеханическими. Поэтому работали они очень медленно и, несмотря на внушительный размер, предлагали совсем немного. Первый такой программируемый электромеханический компьютер был спроектирован в Германии Конрадом Цузе и назван Z3. Он весил тонну! Операция сложения занимала одну секунду, а операция умножения – целых три секунды! Машина состояла из 2000 электромеханических реле и содержала арифметическое устройство, способное только складывать, вычитать, умножать, делить и извлекать квадратный корень. Арифметическое устройство включало два регистра памяти размером 22 бита, используемых для вычислений. Предлагалось также 64 ячейки памяти общего назначения тоже длиной 22 бита. Сегодня мы бы сказали, что машина была оснащена 176 байтами внутренней памяти для данных!

Данные вводились с помощью специальной клавиатуры, а программа считывалась в процессе вычислений с перфорированной целлулоидной пленки. Возможность хранить программу во внутренней памяти компьютера была реализована спустя несколько лет, мы еще дойдем до этого, но уже Цузе эта идея была известна. Однако в данной книге нам важнее вопрос о доступе к памяти в Z3. Для программирования Z3 в нашем распоряжении было всего девять команд! Одна из них позволяла загрузить значение из одной из 64 ячеек памяти в регистр арифметического устройства, другая – сохранить значения обратно в ячейку. И это все, что можно назвать «управлением памятью» в этом первом компьютере. Хотя Z3 во многих отношениях опередил свое время, по политическим причинам и из-за Второй мировой войны его влияние на развитие компьютеров оказалось пренебрежимо малым. Цузе разрабатывал свою линейку компьютеров еще много лет после войны, а последняя версия под номером Z22 была построена в 1955 году.

Во время войны и спустя короткое время после нее основными центрами развития информатики были США и Великобритания. Одним из первых компьютеров, построенных в США, был Harvard Mark I, созданный совместно компанией IBM и Гарвардским университетом. Он получил название «автоматический вычислитель, управляемый последовательностями» (Automatic Sequence Controlled Calculator). Как и Z3, он был электромеханическим. Размеры его были огромны – 17 м в длину, более 2,5 м в высоту и чуть меньше метра в глубину. А весил он 5 т! Его называют самой большой вычислительной машиной всех времен. Строили его несколько лет, а первые программы были выполнены в конце войны, в 1944 году. Он обслуживал интересы ВМС, а также Джона фон Неймана, когда тот рабо-

тал над первой атомной бомбой в рамках Манхэттенского проекта. Несмотря на свои размеры, он предлагал только 72 ячейки памяти для чисел с 23 десятичными разрядами со знаком. Ячейка называлась *аккумулятором* и представляла собой место в памяти, где хранились промежуточные результаты арифметических и логических вычислений. В современных терминах мы бы сказали, что эта 5-тонная машина предоставляла доступ к 72 ячейкам памяти длиной 78 бит (для представления 23-разрядного десятичного числа со знаком нужно 78 бит), т. е. имела 702 байта памяти! Программы представляли собой последовательности математических вычислений над этими 72 ячейками памяти. То были языки программирования первого поколения (1GL), или машинные языки, когда программы хранились на перфоленте, которая физически подавалась машине по мере необходимости, или набирались с помощью переключателей на передней панели. Машина могла выполнять только три операции сложения или вычитания в секунду. Операция умножения занимала 20 секунд, а вычисление sin(*x*) – минуту! Как и в Z3, никакого управления памятью не было – можно было только читать или записывать одно значение в вышеупомянутые ячейки памяти.

Для нас интересно, что с этого компьютера пошел термин «гарвардская архитектура» (см. рис. 1.1), согласно которой память для программ и данных была физически разделена. Данные обрабатываются некоторым электронным или электромеханическим устройством (центральным процессором). Такое устройство часто отвечает также за управление устройствами ввода-вывода: устройствами считывания перфокарт, клавиатурами или устройствами отображения. Хотя в компьютерах Z3 и Mark I эта архитектура применялась из-за ее простоты, она еще не забыта и в наши дни. В главе 2 мы увидим, что практически во всех современных компьютерах используется *модифицированная гарвардская архитектура*. И мы даже увидим, как она влияет на программы, которые мы пишем каждый день.

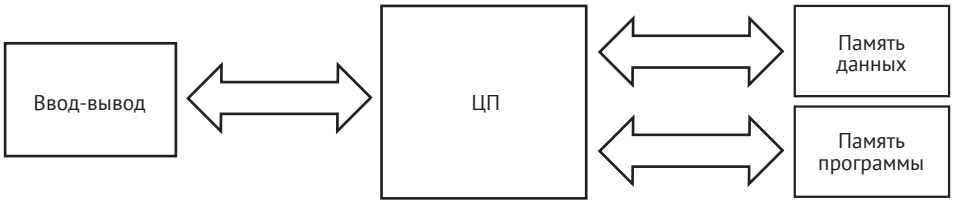


Рис. 1.1 ❖ Гарвардская архитектура

Гораздо более широко известный компьютер ЭНИАК, законченный в 1946 году, был уже полностью электронным устройством на электровакуумных лампах. Скорость выполнения математических операций была в несколько тысяч раз выше, чем у Mark I. Но с точки зрения памяти он выглядел столь же непривлекательно. Предлагалось только двадцать 10-разрядных аккумуляторов со знаком, а внутренней памяти для хранения программ не было. Если говорить по-простому, то во время Второй мировой войны важно было как можно быстрее построить машину для военных целей, а на всякие изыски внимания не обращали.

Но такие ученые, как Конрад Цузе, Алан Тьюринг и Джон фон Нейман, исследовали идею использования внутренней памяти компьютера для хранения про-

граммы вместе с данными. Это позволило бы существенно упростить программирование (а особенно перепрограммирование) по сравнению с кодированием с помощью перфокарт или механических переключателей. В 1945 году Джон фон Нейман опубликовал оказавшую большое влияние статью «Первый проект отчета о EDVAC» (First Draft of a Report on the EDVAC), в которой описал архитектуру, получившую название *архитектура фон Неймана*. Следует отметить, что заслуга принадлежит не только фон Нейману, поскольку он опирался на труды других ученых своего времени.

Архитектура фон Неймана, показанная на рис. 1.2, является упрощенной гарвардской архитектурой, где для хранения программы и данных используется одно запоминающее устройство. Безусловно, она покажется вам похожей на архитектуру современного компьютера – и не без причины. На верхнем уровне именно так сегодня конструируются компьютеры, только гарвардская архитектура и архитектура фон Неймана слились в модифицированную гарвардскую архитектуру.

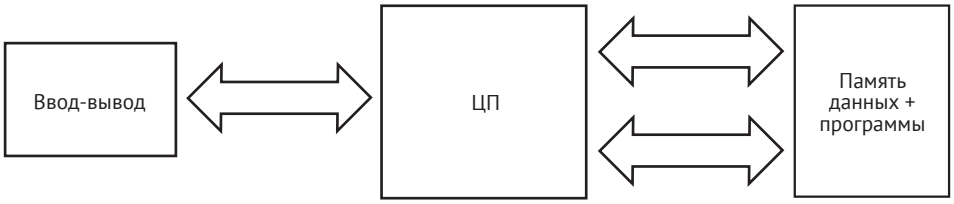


Рис. 1.2 ❖ Архитектура фон Неймана

Манчестерская малая экспериментальная машина (Small-Scale Experimental Machine – SSEM, прозванная также «Baby»), построенная в 1948 году, и Кембриджская EDSAC, построенная в 1949 году, стали первыми в мире компьютерами с хранением данных и команд программы в одном месте, т. е. организованными в соответствии с архитектурой фон Неймана. «Baby» был гораздо более современным и инновационным, поскольку в нем впервые было применено запоминающее устройство нового типа – трубки Вильямса, основанные на электронно-лучевых трубках (ЭЛТ). Трубки Вильямса можно считать первыми запоминающими устройствами с произвольным доступом (ЗУПВ, англ. RAM). В машине SSEM было 32 ячейки памяти по 32 бита каждая. Таким образом, можно сказать, что первый компьютер с ЗУПВ располагал аж 128 байтами памяти! Какое же путешествие нам предстоит – от 128 байтов в 1949 году до 16 гигабайтов в типичном компьютере 2018 года. Тем не менее трубки Вильямса стали стандартом на рубеже 1940-х и 1950-х годов, когда было построено много других компьютеров.

И теперь мы подошли к моменту, когда можем описать все основные понятия компьютерной архитектуры. Они изображены на рис. 1.3.

- *Память*, или *запоминающее устройство*, отвечает за хранение данных и самой программы. Техническая реализация памяти со временем сильно менялась – сначала были перфокарты, потом магнитные ленты и электронно-лучевые трубки, а сейчас транзисторы. Запоминающие устройства можно далее подразделить на две категории:
  - *запоминающее устройство с произвольным доступом* (*ЗУПВ*, англ. *RAM*) – время доступа к данным не зависит от их положения в памя-

ти. Как мы увидим в главе 2, современные запоминающие устройства удовлетворяют этому требованию лишь приближенно, в силу технических причин;
  - *неоднородная память* – противоположность ЗУПВ, время доступа к памяти зависит от положения на физическом носителе. Очевидно, к этой категории относятся перфокарты, магнитные ленты, классические жесткие диски, CD- и DVD-диски и прочие устройства, нуждающиеся в позиционировании (например, повороте) для приведения в нужное для доступа положение.
- *Адрес* представляет конкретное место в памяти. Обычно выражается в байтах, потому что байт – минимальная адресуемая единица на многих платформах.
- *Арифметико-логическое устройство* (*АЛУ*) отвечает за выполнение таких операций, как сложение и вычитание. Это основа компьютера, именно здесь выполняется основная работа. Современные компьютеры включают несколько АЛУ, что дает возможность распараллелить вычисления.
- *Устройство управления* (*УУ*) декодирует команды программы (коды операций), прочитанные из памяти. Исходя из внутреннего описания команды, понимает, какую операцию – арифметическую или логическую – нужно выполнить и над какими данными.
- *Регистр* – часть памяти, к которой АЛУ и УУ (общее название – *исполнительные устройства*) могут обратиться очень быстро. Обычно являются частью АЛУ или УУ. Вышеупомянутые аккумуляторы – упрощенные регистры специального назначения. Время доступа к регистрам очень мало, никакие данные не могут располагаться ближе к исполнительным устройствам, чем регистры.
- *Слово* – базовая единица данных фиксированного размера в конструкции конкретного компьютера. Встречается в описании различных конструктивных элементов, например: размер большинства регистров, максимальный адрес или длина максимального блока данных, передаваемого за одну операцию. Его величина чаще всего выражается в битах и называется *размер слова* или *длина слова*. Современные компьютеры по большей части 32- или 64-разрядные, длина слова составляет соответственно 32 или 64 бит, и таков же размер регистров и т. п.

Архитектура фон Неймана, воплощенная в машинах SSEM и EDSAC, привела к появлению термина *компьютер с хранимой программой*, который сегодня кажется очевидным, хотя в начале компьютерной эры это было далеко не так. При такой конструкции код подлежащей выполнению программы хранится в памяти, и к нему можно обращаться, как к обычным данным, в т. ч. модифицировать его и замещать кодом новой программы.

В устройстве управления имеется дополнительный регистр – *указатель команд* (instruction pointer – IP), или *счетчик команд* (program counter – PC), который указывает на текущую выполняемую команду. Нормальное выполнение программы сводится просто к увеличению адреса, хранящегося в счетчике команд, так чтобы он указывал на следующую команду. А для организации цикла или перехода нужно записать в счетчик команд другой адрес, описывающий, куда программа должна перейти.

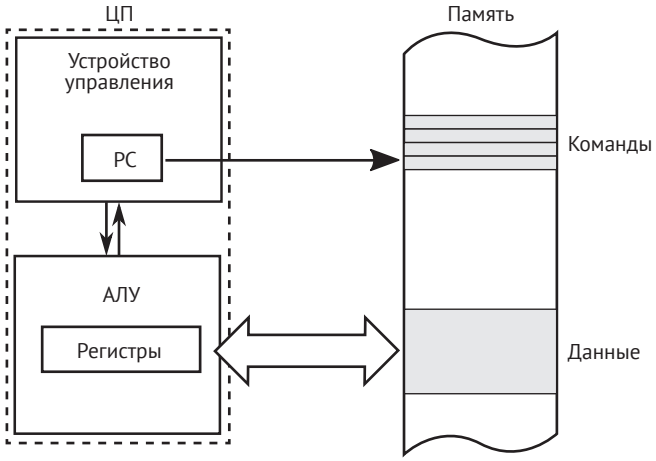


Рис. 1.3 ❖ Компьютер с хранимой программой – память + указатель команд

Первые компьютеры программировались в двоичном коде, который непосредственно описывал выполняемые команды. Но с ростом сложности программ такое решение стало чересчур обременительным. Был создан новый язык программирования (второго поколения – 2GL), позволявший описывать код более простыми средствами – в виде так называемого *ассемблерного кода*. Это текстовое и очень краткое описание команд, выполняемых процессором. Тем не менее этот язык оказался гораздо удобнее прямого двоичного кодирования. Затем были разработаны языки еще более высокого уровня (3GL), в частности хорошо известные C, C ++ и Pascal.

Нам интересно, что программы на любом из этих языков необходимо было преобразовать из текстовой формы в двоичную, а затем поместить в память компьютера. Процесс такого преобразования называется *компиляцией*, а выполняющая его программа – *компилятором*. В случае ассемблерного кода говорят об *ассемблировании* и соответственно об *ассемблере*. В конечном итоге получается программа в двоичном формате, которую можно выполнить позже, – последовательность кодов операций и их аргументов (операндов).

Вооружившись этими базовыми знаниями, мы можем начать путешествие в мир управления памятью.

Статическое выделение

Большинство первых языков программирования допускали только *статическое выделение памяти* – объем и точное расположение памяти должны быть известны на этапе компиляции, еще до начала выполнения программы. Если размер фиксирован и заранее известен, то управление памятью тривиально. Все основные «древние» языки программирования, начиная с машинного или ассемблерного кода, и до первых версий языков FORTRAN и ALGOL располагали лишь такими ограниченными возможностями. У этого способа много недостатков. Статическое выделение ведет к неэффективному использованию памяти – если мы заранее не

знаем, сколько данных предстоит обработать, то откуда же знать, сколько выделить памяти? В результате программы получаются недостаточно гибкими. В общем случае для обработки большего объема данных программу придется перекомпилировать.

В первых компьютерах выделение памяти было статическим, потому что все используемые области памяти (аккумулятор, регистры и ячейки ЗУПВ-памяти) задавались в процессе кодирования программы. Поэтому объявленные «переменные» существовали на протяжении всего времени работы программы. Сегодня статическое выделение в этом смысле используется при создании статических глобальных переменных и подобных им данных, которые хранятся в специальном сегменте данных. Позже мы увидим, где именно они хранятся в программах для .NET.

Регистровая машина

До сих пор мы видели примеры машин, в которых для вычислений в арифметико-логическом устройстве (АЛУ) использовались регистры (в частном случае аккумуляторы). Машины с такой конструкцией называются *регистровыми*, поскольку при выполнении программы на таком компьютере мы фактически производим вычисления с регистрами. Чтобы выполнить сложение, деление или еще какую-то операцию, нужно сначала загрузить данные из памяти в подходящие регистры. Затем выполняются команды, чтобы вызвать соответствующие операции над данными, и напоследок результат записывается из регистра в память.

Пусть требуется написать программу для вычисления выражения `s=x+(2*y)+z` на компьютере с двумя регистрами, `A` и `B`. Предположим также, что `s`, `x`, `y` и `z` – адреса в памяти, по которым хранятся некоторые значения. Еще предположим, что имеется некоторый низкоуровневый псевдоассемблер, в котором определены команды `Load`, `Add`, `Multiply` и т. п. Для такой гипотетической машины можно написать следующую программу.

Листинг 1.1 ❖ Псевдокод программы, реализующей вычисление `s=x+(2*y)+z` на простой регистровой машине с двумя регистрами. В комментариях показано состояние регистров после выполнения каждой команды

```
Load      A, y          // A = y
Multiply  A, 2          // A = A * 2 = 2 * y
Load      B, x          // B = x
Add       A, B          // A = A + B = x + 2 * y
Load      B, z          // B = z
Add       A, B          // A = A + B = x + 2 * y + z
Store     s, A          // s = A
```

Если этот код напоминает x86 или еще какой-то известный вам язык ассемблера, то это не случайно! Дело в том, что большинство современных компьютеров – сложные регистровые машины. В частности, таковыми являются все процессоры компаний Intel и AMD, которые используются в наших компьютерах. При написании ассемблерного кода для архитектур x86 или x64 мы работаем с регистрами общего назначения: `eax`, `ebx`, `ecx` и т. д. Существует, конечно, много других команд, другие специальные регистры и прочее. Но идея сохраняется.

ПРИМЕЧАНИЕ Можно ли представить себе машину с набором команд, которая позволяла бы выполнять команды прямо над операндами в памяти без загрузки в регистры? На нашем языке псевдоассемблера она выглядела бы гораздо более краткой и высокоуровневой, поскольку отпадает нужда в дополнительных командах загрузки из памяти в регистры и сохранения регистров в памяти:

```
Multiply  s, y, 2    // s = 2 * y
Add       s, x       // s = s + x = 2 * y + x
Add       s, z       // s = s + z = 2 * y + x + z
```

Да, такие машины существовали, например IBM System/360, но сегодня я не знаю ни об одном компьютере подобного рода, используемом для решения производственных задач.

Стек

Концептуально стеком называется структура данных, работающая по принципу «последним пришел – первым ушел» (last in, first out – LIFO). Она допускает две основные операции: добавление данных на вершину стека (*push – заталкивание*) и возврат данных, находящихся на вершине (*pop – выталкивание*). См. рис. 1.4.

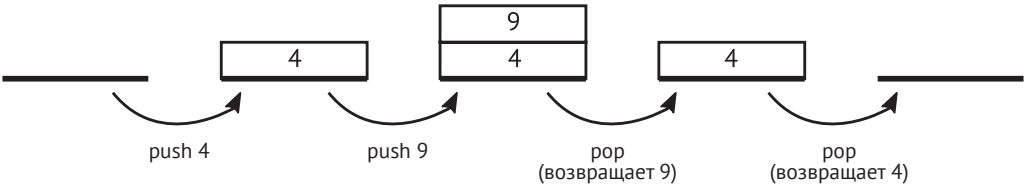


Рис. 1.4 ❖ Операции заталкивания в стек и выталкивания из стека. Рисунок концептуальный и не имеет отношения ни к какой конкретной модели памяти и ее реализации

С самого начала стек был тесно связан с программированием компьютеров, в основном из-за концепции подпрограммы. На современной платформе .NET понятия «стек» и «стек вызовов» используются повсеместно, поэтому посмотрим, с чего все начиналось. Первоначальная семантика стека как структуры данных никуда не делась (например, в .NET есть реализующая ее коллекция `Stack<T>`), но давайте поговорим о том, как она эволюционировала в более общее понятие, связанное с организацией памяти компьютера.

Самые первые компьютеры, о которых мы говорили выше, допускали только последовательное выполнение программы, когда команды считывались одна за другой с перфокарт или перфоленты. Но, конечно, идея о том, чтобы сделать некоторые части программы (*подпрограммы*) повторно используемыми, была очень соблазнительной. Для того чтобы можно было вызывать часть программы, код должен быть адресуемым, поскольку нужно же как-то указать, какую именно часть мы хотим вызвать. Впервые подход к этой проблеме был предложен знаменитой Грейс Хоппер в системе A-0, считающейся первым компилятором. Она записывала набор различных программ на ленту и сопоставляла каждой порядковый номер, который позволял компьютеру найти ее. Таким образом, «программа» состояла из последовательности номеров – индексов программ – и их параметров.

Хотя это действительно вызов подпрограмм, но, очевидно, крайне ограниченный. Программа могла вызывать подпрограммы только поочередно, вложенные вызовы не допускались.

Для вложенных вызовов нужен несколько более сложный подход, потому что компьютер должен где-то запоминать место, с которого следует продолжить вычисление (адрес возврата) после выполнения указанной подпрограммы. Сохранение адреса возврата в одном из аккумуляторов впервые было реализовано Дэвидом Уилером (David Wheeler) в машине EDSAC (этот метод получил название «переход Уилера»). Но его упрощенный подход не допускал *рекурсивных* вызовов, т. е. вызовов подпрограммы из нее же.

Первое упоминание о стеке в том виде, в котором мы знаем его сегодня в контексте компьютерной архитектуры, встречается в отчете Алана Тьюринга об Автоматической вычислительной машине (Automatic Computer Engine – ACE), написанном в начале 1940-х годов. Там описывается концепция машины по типу фон Неймана, т. е. по сути дела компьютера с хранимой программой. Помимо массы других деталей реализации, были описаны две команды – BURY и UNBURY, – работающие с оперативной памятью и сумматорами.

- При вызове подпрограммы (BURY) адрес текущей выполняемой команды плюс единица, указывающий на следующую команду (адрес возврата), сохраняется в памяти. А значение в другой рабочей области памяти, играющее роль указателя стека, увеличивается на 1.
- При возврате из подпрограммы (UNBURY) выполняется противоположное действие.

Это было первой реализацией стека в терминах LIFO-списка адресов возврата из подпрограмм. Именно это решение до сих пор применяется в современных компьютерах; оно, конечно, значительно эволюционировало, но идея сохранилась.

Стек – очень важная часть управления памятью, потому что в программах для .NET данные в нем размещаются сплошь и рядом. Изучим внимательнее стек и его применение для вызова функций. В качестве примера возьмем программу в листинге 1.2, написанную на псевдокоде, напоминающем язык C. В ней имеются два вызова функций: `main` вызывает функцию `fun1` (передавая два аргумента `a` и `b`), в которой объявлены две локальные переменные `x` и `y`. Затем `fun1` в какой-то момент вызывает функцию `fun2` (передавая один аргумент `n`), в которой объявлена одна локальная переменная `z`.

Листинг 1.2 ❖ Псевдокод программы, вызывающей одну функцию из другой

```
void main()
{
   ...
   fun1(2, 3);
   ...
}

int fun1(int a, int b)
{
   int x, y;
   ...
```

```
   fun2(a+b);
}

int fun2(int n)
{
   int z;
   ...
}
```

Изобразим непрерывную область памяти, предназначенную для хранения стека, так что адреса ячеек увеличиваются снизу вверх (левая часть рис. 1.5a), а также другую область памяти, в которой размещена программа (правая часть рис. 1.5a), организованную аналогично. Поскольку код функций не обязан располагаться в смежных участках, блоки, соответствующие `main`, `fun1` и `fun2`, отделены друг от друга. Выполнение программы из листинга 1.2 можно описать следующим образом:

1. Непосредственно перед вызовом `fun1` из `main` (рис. 1.5a). Поскольку программа работает, какой-то стек уже создан (серая часть области стека на рис. 1.5a). В указателе стека (SP) хранится адрес текущей границы стека. Счетчик программы (PC) указывает куда-то внутрь функции `main` (мы обозначили этот адрес `A1`), прямо перед командой вызова `fun1`.

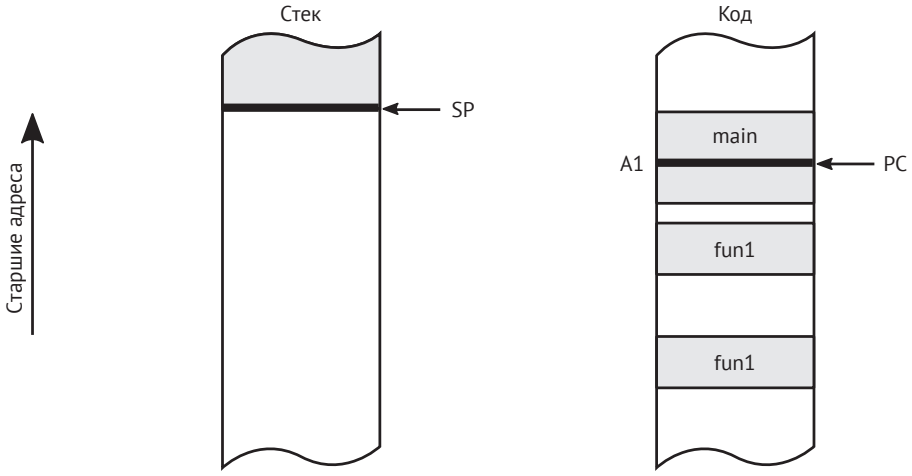


Рис. 1.5a ❖ Стек и память программы – в момент перед вызовом функции `fun1` в листинге 1.2

2. После вызова `fun1` из `main` (рис. 1.5b). Стек растет за счет того, что SP сдвигается, чтобы вместить необходимую информацию. В дополнительной области размещаются:
   - аргументы – все аргументы функции можно сохранить в стеке. В нашем примере туда помещаются аргументы `a` и `b`;
   - адрес возврата – чтобы иметь возможность продолжить выполнение `main` после завершения `fun1`, в стек помещается адрес команды, следующей за вызовом функции. Мы обозначили его `A1+1` (указатель на команду, следующую за командой по адресу `A1`);

– локальные переменные – место для всех локальных переменных также выделяется в стеке. В нашем примере это переменные x и y.

Такая структура, создаваемая в стеке при вызове подпрограммы, называется *записью активации*. В типичной реализации указатель стека уменьшается на соответствующую величину и указывает на область, где может начаться следующая запись активации. Именно поэтому часто говорят, что стек растет вниз.

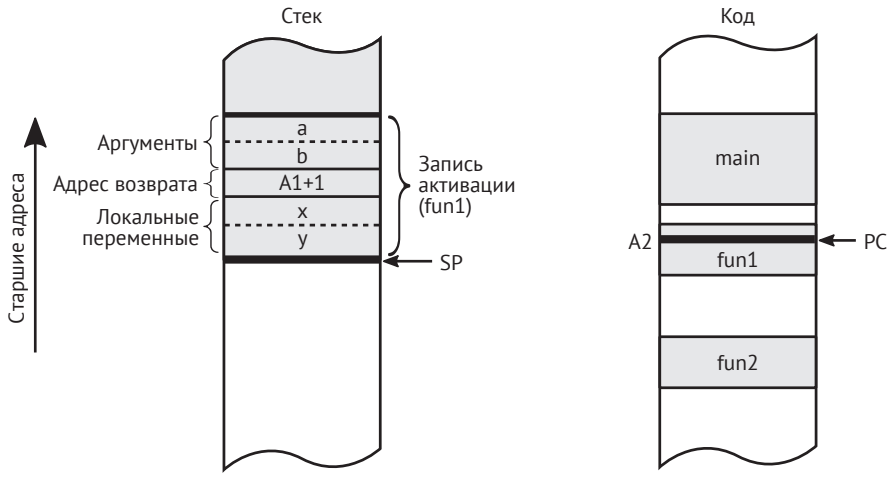


Рис. 1.5b ❖ Стек и память программы – в момент после вызова функции `fun1` в листинге 1.2

3. После вызова `fun2` из `fun1` (рис. 1.5c). Повторяется та же схема создания записи активации. На этот раз она содержит область памяти для аргумента `n`, адреса возврата `A2+1` и локальной переменной `z`.

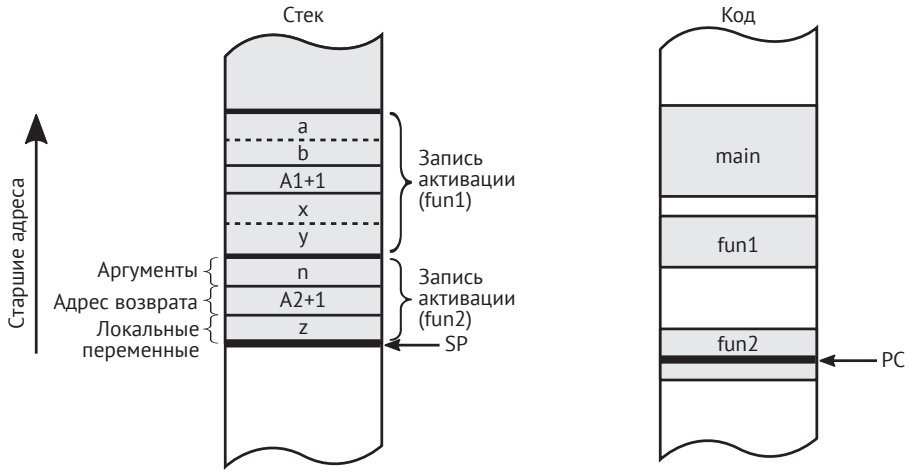


Рис. 1.5c ❖ Стек и память программы – в момент после вызова функции `fun2` из `fun1`

Запись активации называют также более общим термином *кадр стека*, которым обозначают произвольные структурированные данные, сохраненные в стеке для определенных целей.

Как видим, вложенные вызовы подпрограмм просто повторяют схему создания одной записи активации на вызов. Чем больше уровень вложенности, тем больше будет в стеке записей активации. Конечно, бесконечная вложенность невозможна, т. к. это потребовало бы памяти для бесконечного числа записей активации[1]. Если вам когда-нибудь доводилось столкнуться с исключением `StackOverflowException`, то это как раз тот случай – вы сделали так много вложенных вызовов, что память, отведенная для стека, закончилась.

Имейте в виду, что представленный здесь механизм чисто демонстрационный и очень общий. Конкретная реализация может зависеть от архитектуры и операционной системы. В последующих главах мы подробно рассмотрим, как записи активации и стек вообще используются в .NET.

По завершении подпрограммы ее запись активации отбрасывается, для чего достаточно увеличить указатель стека на размер текущей записи активации, а сохраненный адрес возврата копируется в PC, в результате чего выполнение вызывающей функции продолжается. Иными словами, содержимое кадра стека (локальные переменные, параметры) уже не нужно, поэтому увеличения указателя стека достаточно для «освобождения» использованной памяти. Эта область будет попросту перезаписана при следующем использовании стека (рис. 1.6).

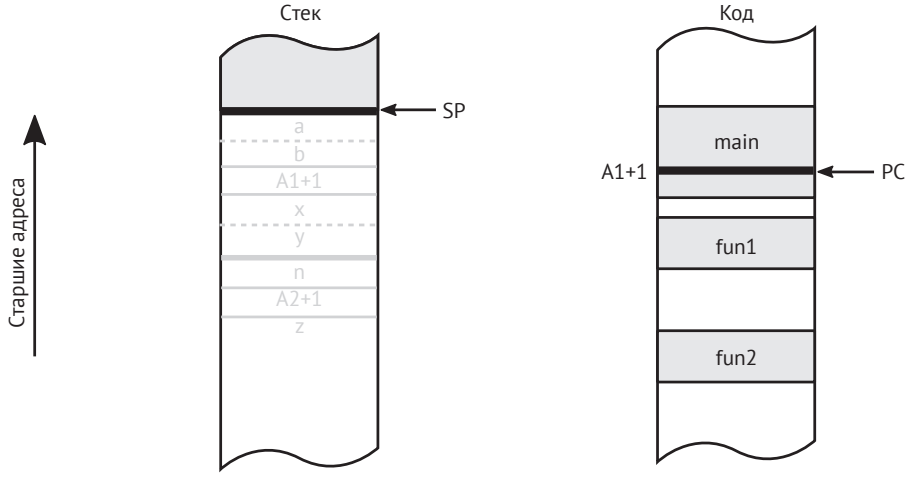


Рис. 1.6 ❖ Стек и память программы – после возврата из функции `fun1` обе записи активации отбрасываются

Что касается реализации: SP и PC обычно хранятся в специальных регистрах. Пока что размеры самого адреса, областей памяти и регистров нам не очень важны.

---

[1] Есть одно интересное исключение – хвостовые вызовы, которое мы за недостатком места здесь не описываем.

В современных компьютерах стек поддержан как на уровне оборудования (специальные регистры для указателей стека), так и на программном уровне (реализованная операционной системой абстракция потока с областью памяти, используемой в роли стека).

Кстати говоря, с точки зрения аппаратной архитектуры, возможно много разных реализаций стека. Стек можно хранить в выделенном блоке памяти внутри ЦП или в выделенной микросхеме. Можно также использовать для этой цели часть общей памяти компьютера. Последний вариант применяется в большинстве современных архитектур – под стек отводится участок памяти процесса фиксированного размера. Возможны даже архитектуры с несколькими стеками. Например, стек адресов возврата может быть отделен от стека параметров и локальных переменных. Это может дать выигрыш в производительности, поскольку доступ к двум стекам может производиться одновременно, что, в свою очередь, открывает возможность для дополнительной оптимизации конвейера ЦП и других низкоуровневых механизмов. Но так или иначе, в современных персональных компьютерах стек – просто часть оперативной памяти.

FORTRAN можно считать первым высокоуровневым языком программирования общего назначения, получившим широкое распространение. Но начиная с 1954 года, когда он был определен, допускалось только статическое выделение памяти. Размер любого массива должен был быть известен на этапе компиляции, а выделение памяти производилось лишь в стеке. ALGOL стал еще одним очень важным языком, он положил начало великому множеству других языков (в частности, C/C++, Pascal, Basic, а через Simula и Smalltalk – всем современным объектно-ориентированным языкам типа Python или Ruby). В ALGOL 60 было только выделение памяти в стеке – наряду с динамическими массивами (размер которых задавался в переменной). Алан Перлис, выдающийся член группы, создавшей ALGOL, писал:

> *Algol 60 невозможно было бы разумно реализовать без понятия стека. Хотя стеки существовали и раньше, лишь после появления Algol 60 они стали занимать центральное место в конструкции процессоров.*

Языки на основе ALGOL и FORTRAN использовались в основном научным сообществом, но одновременно развивалось другое направление, ориентированное на языки программирования для бизнеса: A-0, FLOW-MATIC, COMTRANS и, наконец, широко известный COBOL (Common Business Language). Всем им недоставало явного управления памятью, а работали они в основном с примитивными типами данных: числами и строками.

Стековая машина

Прежде чем двигаться дальше, задержимся еще немного на теме, связанной со стеком, – *стековых машинах*. В отличие от регистровой машины, в стековой машине все команды оперируют специальным *стеком выражений* (или *стеком вычислений*). Имейте в виду, что этот стек не обязан совпадать со стеком, о котором мы говорили до сих пор. Поэтому в такой машине вполне могли бы сосуществовать стек общего назначения и дополнительный *стек выражений*. Регистров может вообще не быть. По умолчанию в такой машине команды получают аргументы

с вершины стека выражений – столько, сколько им необходимо. Результат также записывается в вершину стека. В таком случае говорят о *чисто стековой машине* в противоположность нечистым реализациям, в которых операциям разрешено получать значения не только с вершины стека, но и с более глубоких уровней.

Как именно операция обращается к стеку выражений? Например, гипотетическая команда Multiply (без аргументов) извлекает два значения с вершины стека выражений, перемножает их и помещает результат обратно в стек выражений (рис. 1.7).

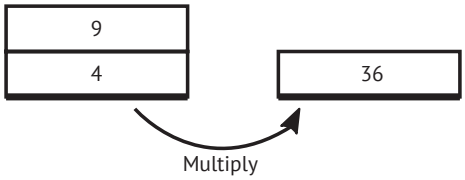


Рис. 1.7 ❖ Гипотетическая команда Multiply в стековой машине – извлекает два элемента из стека и помещает назад результат их умножения

Вернемся к выражению s=x+(2*y)+z из примера для регистровой машины и перепишем его для стековой машины.

Листинг 1.3 ❖ Псевдокод вычисления выражения s=x+(2*y)+z стековой машиной. В комментариях показано состояние стека вычислений

```
            // пустой стек
Push 2      // [2] - в стеке один элемент со значением 2
Push y      // [2][y] - в стеке два элемента: 2 и y
Multiply    // [2*y]
Push x      // [2*y][x]
Add         // [2*y+x]
Push z      // [2*y+x][z]
Add         // [2*y+x+z]
Pop l       // [] (с побочным эффектом - записью значения в l)
```

Эта идея приводит к очень ясному коду. Опишем основные преимущества.

- Не возникает вопроса, как и где хранить временные значения: в регистрах, в стеке или в основной памяти. Концептуально это проще, чем пытаться найти оптимальное решение о месте размещения. А значит, и реализация упрощается.
- Коды операций можно сделать короче, поскольку многие команды вообще не имеют операндов или имеют всего один операнд. Это позволяет более эффективно кодировать команды, а значит, двоичный код будет занимать меньше места в памяти, пусть даже общее число команд больше, чем в регистровой машине (из-за большего числа команд загрузки и сохранения).

На заре развития компьютеров, когда память была очень дорогой и ограниченной, это было существенное преимущество. Оно остается важным и сегодня, когда речь идет о скачиваемом коде для смартфонов или веб-приложений. Кроме того, плотное двоичное кодирование команд улучшает использование кеша процессора.

Но, несмотря на все преимущества, идея стековой машины редко воплощалась в оборудовании. Заметное исключение – машины компании Burroughs, например B5000, в которых имелась аппаратная реализация стека. Сегодня, наверное, нет ни одной широко распространенной машины, которую можно было бы назвать стековой, если не считать сопроцессора x87 для вычислений с плавающей точкой (часть совместимых с x86 процессоров), который проектировался как стековая машина и ради обратной совместимости до сих пор так и программируется.

Тогда зачем же вообще упоминать эти машины? Поскольку такая архитектура очень удобна для проектирования платформенно-независимых виртуальных машин или исполняющих движков. Виртуальная машина Java компании Sun и среда выполнения .NET – идеальные примеры стековых машин. Они исполняются такими хорошо известными регистровыми машинами, как архитектуры x86 или ARM, но это не отменяет тот факт, что они реализуют логику стековой машины. Мы убедимся в этом, когда в главе 4 будем описывать промежуточный язык .NET – Intermediate Language (IL). Почему среда выполнения .NET и JVM (виртуальная машина Java) спроектированы таким образом? Как всегда, причин несколько – инженерного и исторического характера. Код для стековой машины более высокого уровня и лучше абстрагирует особенности реального оборудования. И среду выполнения Майкрософт, и Sun JVM можно было написать как регистровые машины, но как решить, сколько регистров необходимо? Если все они виртуальные, то оптимальный ответ – бесконечно много. Тогда необходимо как-то управлять ими и организовать повторное использование. И как бы выглядела оптимальная абстрактная регистровая машина?

Даже если оставить такие проблемы в стороне и поручить кому-то еще (в данном случае среде выполнения Java или .NET) платформенно-зависимую оптимизацию, то регистровые или стековые механизмы вычислений все равно пришлось бы отобразить на конкретную регистровую архитектуру. Но стековые машины концептуально проще. Виртуальная стековая машина (не исполняемая реальной аппаратной стековой машиной) может обеспечить приемлемую платформенную независимость, порождая при этом высокопроизводительный код. А в совокупности с вышеупомянутой повышенной плотностью кода это дает хорошую платформу, которая может работать на самых разных устройствах. Наверное, именно по этой причине Sun решила пойти по данному пути, когда проектировала Java для небольших устройств, например телевизионных приставок. Майкрософт при проектировании .NET также выбрала этот путь. Концепция стековой машины элегантна, проста и работает. Поэтому реализация виртуальной машины – более приятная инженерная задача!

С другой стороны, регистровые виртуальные машины ближе к конструкции реального оборудования, на котором работают. Это расширяет спектр возможных оптимизаций. Сторонники данного подхода говорят, что так можно добиться гораздо более высокой производительности, особенно для интерпретирующих сред выполнения. У интерпретатора гораздо меньше времени на то, чтобы заниматься сложными оптимизациями, поэтому чем ближе интерпретируемый код к машинному, тем лучше. Кроме того, оперирование самым часто используемым набором регистров улучшает локальность ссылок в кеше[1].

---

[1] Примечание: о важности характера доступа к памяти с точки зрения использования кеша мы будем говорить в главе 2.

Как всегда, принимая решение, приходится идти на компромиссы. Спор между сторонниками обоих подходов долгий и неразрешенный. Но факт остается фактом – среда выполнения .NET реализована в виде стековой машины, хотя и не совсем чистой (мы отметим это в главе 4). Мы также увидим, как стек вычислений отображается на реальное оборудование, состоящее из регистров и памяти.

> **ПРИМЕЧАНИЕ** Все ли виртуальные машины и среды выполнения являются стековыми машинами? Вовсе нет! Например, Dalvik, виртуальная машина в Google Android до версии 4.4, – регистровая реализация JVM. Это был интерпретатор промежуточного «байт-кода Dalvik». Позже JIT (компиляция «на лету», объясняется в главе 4) была включена в преемник Dalvik – среду выполнения Android (ART). Есть и другие примеры: BEAM – виртуальная машина для Erlang/Elixir, Chakra – среда выполнения JavaScript в IE9, Parrot (виртуальная машина в Perl) и Lua VM (виртуальная машина для языка Lua). Никто не скажет, что такие машины не пользуются популярностью.

Указатель

До сих пор мы рассматривали только два вида памяти: статически выделенную и выделенную в стеке (как часть кадра стека). Концепция *указателя* очень общая, она появилась уже в самом начале компьютерной эры – вспомните, например, уже упоминавшиеся указатель команд (счетчик программы) и указатель стека. Специальные регистры, служащие для адресации памяти, например *индексные регистры*, тоже можно считать указателями[1].

В 1965 году компания IBM предложила язык PL/I, предназначенный как для научных, так и для коммерческих приложений. Хотя его цели так и не были в полной меры достигнуты, он является важной исторической вехой, поскольку в нем впервые была введена концепция указателей и выделения памяти. На самом деле Гарольд Лоусон, один из разработчиков языка PL/I, в 2000 году был удостоен премии IEEE «за изобретение переменной указателя и включение этой концепции в язык PL/I, что впервые дало возможность гибко реализовать связные списки в высокоуровневом языке общего назначения». Именно эта задача и послужила стимулом для изобретения указателей – обработка списков и других более-менее сложных структур данных. Концепция указателя затем была использована при разработке языка C, преемника языка B (и его предшественников BCPL и CPL). Лишь в версии FORTRAN 90, последовавшей за FORTRAN 77 и утвержденной в 1991 году, появились динамическое выделение памяти (посредством подпрограмм allocate и deallocate), атрибут `POINTER`, присваивание указателю и выражение `NULLIFY`.

Указатели – это переменные, в которых хранится адрес местоположения в памяти. Проще говоря, указатель позволяет ссылаться на другие места в памяти по адресу. Размер указателя связан с вышеупомянутой длиной слова и зависит от архитектуры компьютера. Поэтому в настоящее время мы работаем с 32- или 64-разрядными указателями. Поскольку это всего лишь небольшой участок па-

---

[1] Если говорить об адресации памяти, то важным усовершенствованием был индексный регистр, впервые появившийся в машине Manchester Mark 1, пришедшей на смену Baby. Индексный регистр позволяет адресовать память косвенно путем сложения его с другим регистром. Поэтому для действий с непрерывными участками памяти, например массивами, требуется меньше команд.

мяти, его можно поместить в стек (например, в качестве локальной переменной или аргумента функции) или в регистр ЦП. На рис. 1.8 изображена типичная ситуация, когда одна из локальных переменных (хранящихся в записи активации функции) – указатель на другую область памяти по адресу `Addr`.

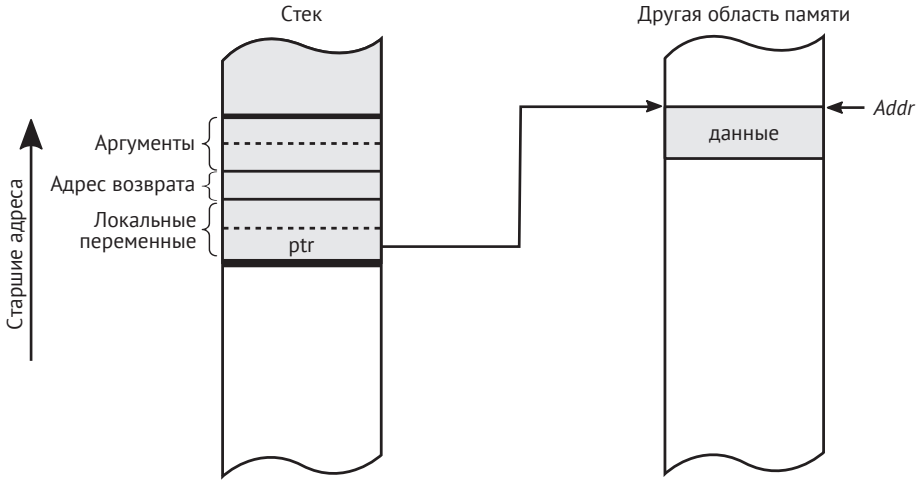


Рис. 1.8 ❖ Локальная переменная функции, являющаяся указателем `ptr` на область памяти по адресу `Addr`

Простая идея указателей позволяет создавать сложные структуры данных, в т. ч. связные списки и деревья, поскольку структуры данных в памяти могут ссылаться друг на друга, образуя еще более сложные структуры (рис. 1.9).

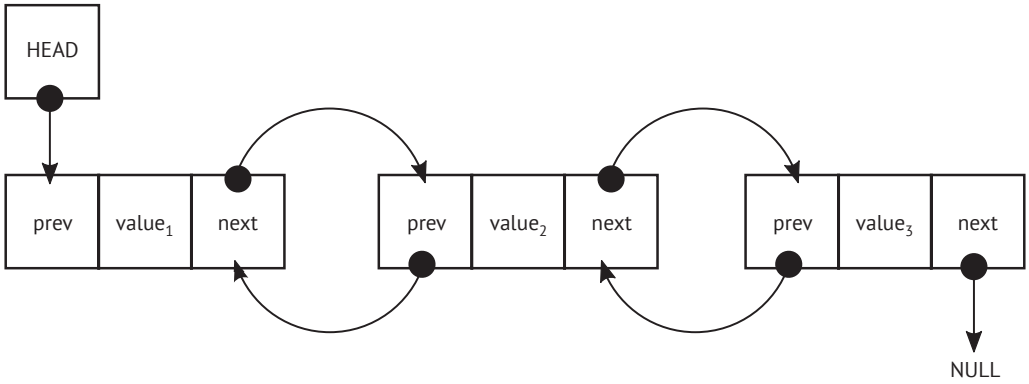


Рис. 1.9 ❖ Указатели, используемые для построения двусвязного списка, в котором каждый элемент указывает на предыдущий и последующий элементы

Кроме того, для указателей можно определить так называемую *адресную арифметику*. Например, оператор инкремента увеличивает значение указателя на длину объекта, на который он указывает, а не на один байт, как можно было бы ожидать.

В языках высокого уровня, в частности Java и C#, указатели нередко недоступны или их использование должно быть разрешено явно, в результате чего код перестает быть безопасным. Почему это так, станет ясно, когда мы будем говорить о ручном управлении памятью ниже в этой главе.

Куча

Вот, наконец, мы и добрались до самого важного понятия в контексте управления памятью в .NET. *Куча* (менее известно название *свободное хранилище*) – это область памяти, используемая для динамически выделенных объектов. Термин «свободное хранилище» лучше, потому что не предполагает никакой внутренней структуры, а только назначение. На самом деле можно с полным правом задать вопрос, что общего между структурой данных *куча* (heap) и собственно кучей. Ответ – ничего. Если стек имеет четкую организацию (структура данных с дисциплиной обслуживания LIFO), то куча больше похожа на черный ящик, у которого можно попросить память, не интересуясь, откуда он ее берет. Поэтому «пул» или «свободное хранилище» лучше отражали бы ее назначение. Возможно, название «куча» происходит от традиционного в английском языке значения «хаотическое нагромождение» – как противоположности упорядоченному стеку. Исторически выделение памяти из кучи было введено в языке ALGOL 68, который так и не получил широкого признания. Однако само название, вероятно, идет оттуда. Но факт остается фактом – точно о происхождении термина теперь никто не знает.

Куча – это механизм, позволяющий получить непрерывный блок памяти указанного размера. Эта операция называется *динамическим выделением* (или *распределением*) *памяти*, поскольку ни размер, ни фактическое местоположение области памяти неизвестно на этапе компиляции. Так как местоположение заранее неизвестно, обращаться к динамически выделенной памяти можно только по указателю. Следовательно, понятия указателя и кучи неразрывно связаны.

Адрес, возвращенный некоторой функцией «дай мне X байт памяти», должен быть запомнен в указателе для последующего обращения к выделенному блоку памяти. Его можно сохранить в стеке (рис. 1.10), в самой куче или где-то еще:

```
PTR ptr = allocate(10);
```

Операция, обратная выделению, называется *освобождением*, она возвращает указанный блок памяти в пул для будущего использования. Как именно из кучи выделяется область указанного размера – деталь реализации. Существует много таких «распределителей», и о некоторых мы скоро узнаем.

Если выделяется и освобождается много блоков, то можно столкнуться с ситуацией, когда блока памяти нужного размера нет, хотя всего в куче памяти достаточно. Это называется *фрагментацией* кучи и может стать причиной крайне неэффективного использования памяти. На рис. 1.11 показано, как выглядит отсутствие непрерывного блока памяти для объекта X. Распределители применяют различные стратегии оптимального управления пространством, чтобы избежать фрагментации (или воспользоваться ей во благо).

Отметим также, что в одном процессе может быть как одна, так и несколько куч – это тоже деталь реализации (мы вернемся к этому вопросу, когда будем глубже обсуждать .NET).

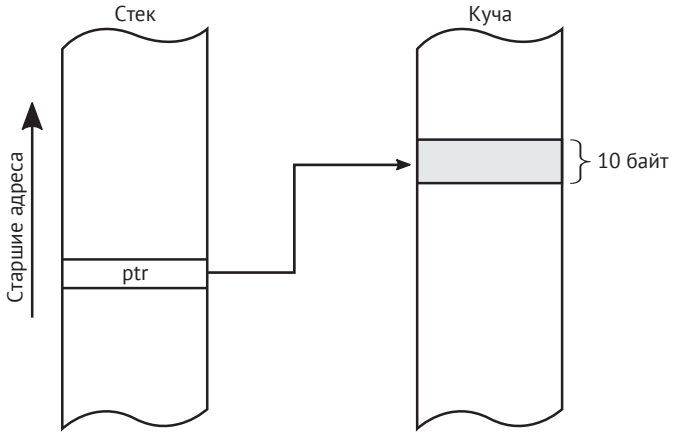


Рис. 1.10 ❖ В стеке хранится указатель `ptr` на область в куче размером 10 байт

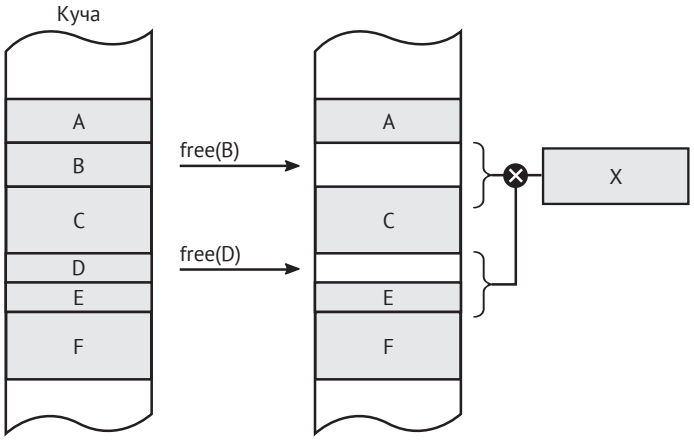


Рис. 1.11 ❖ Фрагментация – после удаления объектов B и D для нового объекта X не хватает памяти, хотя всего памяти достаточно

В табл. 1.1 приведена краткая сводка различий между стеком и кучей.

Помимо этих различий, стек и куча чаще всего располагаются в противоположных концах адресного пространства процесса. Мы вернемся к детальному рассмотрению размещения стека и кучи в адресном пространстве процесса, когда будем обсуждать низкоуровневое управление памятью в главе 2. Однако помните, что это всего лишь деталь реализации. Благодаря абстракциям значимых и ссылочных типов (см. главу 4) нам безразлично, где именно они создаются. Теперь перейдем к сравнению ручного и автоматического управления памятью. В «Аннотированном руководстве по языку C++» Эллис и Страуструп пишут:

> *Программисты на C считают, что управление памятью – слишком важная вещь, чтобы оставлять ее компьютеру. Программисты на Lisp считают, что управление памятью – слишком важная вещь, чтобы оставлять ее пользователю.*

Таблица 1.1. Сравнение стека и кучи

Свойство	Стек	Куча
Время жизни	Область видимости локальных переменных (помещаются при входе, извлекаются при выходе)	Явное (в результате выделения и, возможно, освобождения)
Область видимости	Локальная (поточно[1])	Глобальная (любой, кто имеет доступ к указателю)
Доступ	Локальная переменная, аргументы функции	Указатель
Время доступа	Быстро (с большой вероятностью находится в кеше процессора)	Медленнее (может даже временно выгружаться на диск)
Выделение	Перемещение указателя стека	Возможны различные стратегии
Время выделения	Очень быстро (сдвиг указателя стека вниз)	Медленнее (зависит от стратегии выделения)
Освобождение	Перемещение указателя стека	Возможны различные стратегии
Использование	Параметры подпрограмм, локальные переменные, записи активации, данные небольшого известного на этапе компиляции размера (массивы)	Все, что угодно
Емкость	Ограничена (обычно несколько мегабайтов на поток)	Ограничена только размером жесткого диска
Переменный размер	Нет	Да[2]
Фрагментация	Нет	Вероятна
Основные угрозы	Переполнение стека	Утечка памяти (забыли освободить выделенную память), фрагментация

Ручное управление памятью

Все, о чем мы говорили до сих пор, можно назвать «ручным управлением памятью». Это означает, в частности, что разработчик отвечает за явное выделение памяти и должен освободить ее, когда она больше не нужна. Это самая настоящая ручная работа. Как ручная передача в большинстве европейских автомобилей. Я сам из Европы, и мы привыкли переключать передачи вручную. Мы должны подумать, пора ли уже переключать или надо подождать несколько секунд, пока двигатель наберет нужные обороты. Тут есть одно большое преимущество – мы полностью контролируем машину. Мы сами отвечаем за то, оптимально работает двигатель или нет. А поскольку люди гораздо лучше адаптируются к изменяющимся условиям, хороший водитель справляется лучше, чем автоматическая коробка. Но, конечно, есть и большой недостаток. Вместо того чтобы сосредоточиться на главной цели – как добраться из точки A в точку B, мы должны еще думать о переключении передач – сотни, тысяч раз в течение долгой поездки. Это

[1] Это не совсем так, потому что указатель на переменную в стеке можно передать в другой поток. Однако это, безусловно, аномальное использование.

[2] В силу динамической природы кучи существуют функции, позволяющие изменить размер указанного блока памяти (перераспределить память).

и время отнимает, и утомляет. Я знаю людей, которым это по душе и которые считают, что отдавать управление автоматической коробке было бы скучно. Я даже могу согласиться с ними. Но все равно мне нравится эта автомобильная метафора в применении к управлению памятью.

Явное выделение и освобождение памяти сродни ручной передаче. Вместо того чтобы сосредоточиться на главной цели – а это, наверное, какая-то бизнес-задача, решаемая нашей программой, – мы должны думать также о том, как управлять памятью. Это отвлекает наше ценное внимание от главной цели. Вместо того чтобы размышлять об алгоритмах, бизнес-логике и предметной области, мы вынуждены думать о том, когда и сколько памяти нам нужно. И надолго ли? И кто будет отвечать за ее освобождение? Это что, бизнес-логика? Нет, конечно. Но вопрос о том, хорошо это или плохо, – совсем другая история.

Хорошо известный язык C был создан Дэннисом Ричи в начале 1970-х годов и стал одним из самых распространенных в мире языков программирования. История того, как C эволюционировал от ALGOL через промежуточные языки CPL, BCPL и B, интересна сама по себе, но в нашем контексте важно, что наряду с Pascal (прямой потомок ALGOL) они были в свое время двумя самыми популярными языками, допускающими явное управление памятью. Что касается C, то я могу без сомнения сказать, что компилятор для него имеется на любой когда-либо существовавшей аппаратной платформе. Не буду удивлен, если и на борту кораблей пришельцев тоже есть компилятор C (возможно, применяемый для реализации стека TCP/IP – еще один пример повсеместно распространенного стандарта). Влияние этого языка на другие языки программирования огромно, переоценить его невозможно. Давайте сделаем короткую паузу и взглянем на него в контексте управления памятью. Это позволит сформулировать некоторые характеристики ручного управления.

Рассмотрим простую программу, написанную на C.

Листинг 1.4 ❖ Пример программы на C, демонстрирующий ручное управление памятью

```
#include <stdio.h>

void printReport(int* data)
{
   printf("Отчет: %d\n", *data);
}

int main(void) {
   int *ptr;
   ptr = (int*)malloc(sizeof(int));
   if (ptr == 0)
   {
      printf("ОШИБКА: недостаточно памяти\n");
      return 1;
   }
   *ptr = 25;
   printReport(ptr);
   free(ptr);
   ptr = NULL;
   return 0;
}
```

Конечно, пример несколько утрированный, но он позволяет продемонстрировать суть проблемы. Легко заметить, что у этой тривиальной программы одна простая цель: напечатать «отчет». Для простоты отчет состоит из единственного целого числа, но можно представить себе сложную структуру, содержащую указатели на другие структуры данных. Эту простую задачу затмевает «церемониальный код», который не занимается ничем, кроме работы с памятью. Это и есть ручное управление памятью во всей красе.

Помимо написания бизнес-логики, разработчик в этом примере должен:

- выделить нужное количество памяти для данных с помощью функции `malloc`;
- привести общий указатель (типа `void*`) к указателю на нужный тип (`int*`), чтобы показать, что он указывает на числовое значение (в данном случае типа `int`);
- запомнить указатель на выделенную область памяти в локальной переменной `ptr`;
- проверить, было ли выделение памяти успешным (в случае неудачи возвращенный адрес будет равен 0);
- разыменовать указатель (получить доступ к памяти по этому адресу), чтобы сохранить данные (числовое значение 25);
- передать указатель в функцию `printReport`, которая разыменует его для собственных целей;
- освободить выделенную память, когда необходимость в ней отпадет, вызвав функцию `free`;
- не забыть присвоить указателю специальное значение `NULL` (так мы говорим, что указатель ни на что не указывает, в действительности это значение соответствует 0[1]).

Как видим, управляя памятью вручную, мы многое должны держать в голове. А если любое из описанных выше действий выполнено неправильно или пропущено, то могут воспоследовать серьезные проблемы. Посмотрим, какие именно.

- Мы должны точно знать, сколько памяти необходимо. В нашем примере все просто – `sizeof(int)` байт, но что, если структура гораздо более сложная, да еще вложенная? Легко представить ситуацию, когда из-за мелкой ошибки в вычислениях размера мы выделим меньше памяти, чем необходимо. Впоследствии, когда мы захотим записать в такую область памяти или прочитать из нее, дело, вероятно, закончится ошибкой *Segmentation Fault* (ошибка сегментации) вследствие попытки доступа к памяти, которую мы не выделяли или выделили для других целей. С другой стороны, из-за такой же ошибки можно выделить слишком много памяти, что ведет к ее неэффективному использованию.
- Приведение типов всегда чревато ошибками и может стать причиной трудно диагностируемых дефектов, если случайно допустить несоответствие типов. Это означает, что мы пытаемся интерпретировать указатель одного типа как указатель совершенно другого типа, что может привести к ошибке доступа к памяти.

[1] Детали реализации значения NULL в .NET будут объяснены в главе 10.

- Запомнить адрес нетрудно. Но что, если мы забудем это сделать? Мы выделили память, но не можем ее освободить – адреса-то нет! Это прямой путь к утечке памяти, поскольку размер неосвобождаемой памяти будет только расти. А может быть и так, что указатель хранится в некой сложной структуре, а не просто в локальной переменной. Что, если мы забудем указатель на сложный граф объектов, потому что освободили содержащую его структуру?
- Одна проверка успешности выделения запрошенной памяти не обременительна. Но делать это сотни раз в каждой функции, безусловно, надоедает. И возможно, мы решим опустить эти проверки, что может привести к неопределенному поведению во многих местах приложения, поскольку мы пытаемся обратиться к памяти, которая не была выделена.
- Разыменование указателей всегда опасно. Никто не знает, куда они указывают. Находится ли по этому адресу действительный объект или он уже освобожден? Да и правилен ли сам указатель? Действительно ли он указывает на адрес в доступной пользователю памяти? Полный контроль над указателями в языках, подобных C, обязательно сопряжен с такими трудностями. А также влечет за собой серьезные проблемы с безопасностью – только на программиста возлагается ответственность не раскрывать данные за пределами области, доступной в соответствии с текущей моделью и типом памяти.
- Передача указателя между функциями и потоками лишь умножает вышеуказанные проблемы в многопоточной среде.
- Мы должны помнить о необходимости освободить выделенную память. Если опустить этот шаг, то будет иметь место утечка памяти. В таком простом примере, как выше, забыть о вызове функции `free`, конечно, трудно. Но в более сложном коде, когда владелец структуры данных не столь очевиден и указатели на структуру передаются туда-сюда, ситуация становится куда более сложной. И есть еще одна опасность – никто не помешает нам освободить память, которая уже была освобождена ранее. А это тоже неопределенное поведение и вероятная причина ошибки сегментации.
- И наконец, мы должны обнулить наш указатель (присвоить ему значение NULL, 0 и т. п.). В противном случае останется *висячий указатель*, что рано или поздно приведет к ошибке сегментации или иному неопределенному поведению, поскольку кто-нибудь попробует разыменовать его, считая, что он все еще представляет действительные данные.

Как видим, с точки зрения разработчика, явное выделение и освобождение памяти могут стать очень обременительным делом. Это очень мощный механизм, у которого, безусловно, есть свои применения. Если требуется максимальная производительность и разработчик на 100 % уверен, что контролирует происходящее, то такой подход может быть полезен.

Но «кому многое дано, с того много и спросится», меч этот обоюдоострый. Поэтому по мере развития программной инженерии языки становились все более и более изощренными, стремясь избавить разработчика от груза этих проблем.

Язык C++, прямой преемник C, в этом отношении не сильно изменился. Однако стоит уделить некоторое время C++ , поскольку он очень популярен и в нем впервые появились другие широко распространенные идеи. Как все мы знаем, в этом

языке управление памятью осуществляется вручную. Прямая трансляция предыдущего примера на C++ могла бы выглядеть так.

Листинг 1.5 ❖ Пример программы на C++, демонстрирующий ручное управление памятью

```
#include <iostream>

void printReport(int* data)
{
   std::cout << "Отчет: " << *data << "\n";
}

int main()
{
   try
   {
      int* ptr;
      ptr = new int();
      *ptr = 25;
      printReport(ptr);
      delete ptr;
      ptr = 0;
      return 0;
   }
   catch (std::bad_alloc& ba)
   {
      std::cout << "ОШИБКА: недостаточно памяти\n";
      return 1;
   }
}
```

В контексте нашего рассмотрения мы можем отметить значительные улучшения.

- Оператор `new` сам выделит достаточно памяти, поскольку благодаря поддержке со стороны компилятора (которому известен размер типа) знает, сколько памяти необходимо.
- Не нужно приводить полученный указатель к требуемому типу. Это снимает некоторые из упомянутых выше опасений по поводу типобезопасности.
- Обработка ошибок также стала лучше: мы не обязаны вручную проверять успешность выделения, потому что в случае проблемы будет возбуждено исключение.

И тем не менее в этом примере много церемониального кода. И появилась еще одна причина для беспокойства. Что, если функция `printReport()` возбудит исключение? Без надлежащей обработки ошибок оператор `delete` не будет выполнен – налицо утечка памяти. Исправить этот код легко, но в более сложных приложениях, где владелец данных (кто и на каком уровне должен освобождать память) не очевиден, ситуация может оказаться не столь тривиальной.

Все описанные в этой главе проблемы становятся еще серьезнее в многопоточной среде, когда указатели могут разделяться между несколькими единицами выполнения. В этих случаях необходима тщательная синхронизация, чтобы не допустить смешения некорректных данных. Например, что, если один поток про-

веряет действительность указателя (отличие от NULL), а сразу после этого другой поток освобождает память, на которую тот указывает? Это может привести к нерегулярным ошибкам, которые очень трудно искать. В мире явного управления памятью разработчик сам должен обеспечить подходящий механизм синхронизации, предотвращающий такие ситуации.

В примере из листинга 1.5 сознательно не используются современные паттерны работы с памятью в C++. Следовало бы использовать какой-то вариант идиомы RAII (получение ресурса есть инициализация), когда ресурс (например, память) представляется локальной переменной, принадлежащей типу, который обеспечивает тот или иной вид владения памятью. Подобный пример приведен ниже в листинге 1.10. И хотя, как мы увидим, такие паттерны помогают решить часть проблем, они принципиально ничего не меняют в общих рассуждениях о ручном и автоматическом управлении памятью.

Автоматическое управление памятью

Чтобы преодолеть проблемы ручного управления памятью и предоставить программисту более приятный способ работы с ней, были предложены различные подходы к автоматическому управлению памятью. Интересно, что уже второй старейший высокоуровневый язык программирования, LISP, предложенный в 1958 году (всего через несколько лет после FORTRAN), мог многое предложить в этой области, поскольку в преимущественно функциональном языке, опирающемся на обработку списков, вручную управлять памятью было бы крайне неудобно. В функциональной парадигме программирования программу рассматривают как вычисление комбинированных функций и всячески избегают модификации данных и побочных эффектов. А выделение и освобождение памяти – без сомнения, операции, изменяющие данные, и с очевидными побочными эффектами. Такая работа с памятью в функциональном коде испортила бы его императивными запахами, тогда как LISP проектировался как в высшей степени декларативный язык. Создатель языка LISP говорил: «если бы мы были вынуждены явно стирать списки, то все стало бы безумно уродливым». Поэтому нужно было разработать что-то более изощренное. В самых первых версиях LISP существовала встроенная функция eralist (стереть список), но после реализации автоматического управления памятью она была исключена.

Вообще, LISP был в высшей степени новаторским языком, благодаря его дизайну в информатику вошло много важных идей, одной из которых и было автоматическое управление памятью. Кстати, Джон Маккарти, автор термина «искусственный интеллект» и изобретатель LISP, является также отцом первых алгоритмов сборки мусора. Многие высказанные тогда идеи по-прежнему живут и используются в современных языках. Можно с уверенностью сказать, что автоматическое управление памятью зародилось в LISP'е. В первой статье, написанной Маккарти в 1958 году, был изложен алгоритм пометки и очистки (Mark and Sweep), который мы подробно изучим в последующих главах, потому что он все еще используется в среде .NET и многих других местах.

Благодаря своей выразительности и лаконичности LISP позволяет представить наш пример в очень простой форме.

Листинг 1.6 ❖ Пример программы на LISP, демонстрирующий автоматизированное управление памятью

```
(defun printReport(data)
   (write-line (format nil "Report: ~a" data))
)

(prog
   ((ptr 25))
   (printReport ptr)
)
```

Благодаря автоматическому управлению памятью все лишние детали ушли из кода, и стало отчетливо видно высокоуровневое описание цели программы – напечатать «отчет».

Приведем интересную историю из статьи Джона Маккарти о дизайне LISP, «Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I». Он кратко описал этот механизм, но назвал его просто «reclamation» (восстановление, рекультивация, регенерация). Позже, в аннотации к этой части он писал:

> *Мы уже тогда называли этот процесс «сборкой мусора», но, думаю, я побоялся использовать данный термин в статье, опасаясь, что дамы из редакторского отдела Исследовательской лаборатории по электронике не пропустят ее.*

Но, если не считать названия, идея уже присутствовала в статье и только ждала реализации. В настоящее время термины автоматическое управление памятью и *сборка мусора* употребляются как синонимы. Мы можем определить это как механизм, снимающий с программиста ответственность за ручное управление памятью и гарантирующий, что созданные объекты автоматически уничтожаются (с возвратом памяти системе), когда необходимость в них отпадает.

Один из главных посылов этой книги – тот факт, что даже полностью автоматическое управление памятью не избавляет от всех проблем. В качестве неформального подтверждения стоит привести забавную историю, относящуюся к первой реализации сборки мусора в LISP. В своей книге «История языков программирования» Маккарти вспоминает, что в ходе первой публичной демонстрации LISP на одном из симпозиумов по промышленной связи в MIT из-за мелкого недосмотра Flexowriter (тогдашняя электрическая пишущая машинка) начал печатать одну за другой страницы, содержащие сообщение об ошибке, которое начиналось словами[1]:

> *THE GARBAGE COLLECTOR HAS BEEN CALLED. SOME INTERESTING STATISTICS ARE AS FOLLOWS*

Презентацию пришлось отменить, а в аудитории стоял хохот. Никто, кроме Джона, не знал, что причиной было неправильное использование сборщика мусора. И хотя это была не алгоритмическая, а человеческая ошибка, можно все-таки сказать, что сборщики мусора причиняли неприятности с самого начала!

---

[1] Вызван сборщик мусора. Далее следует представляющая интерес статистика. – *Прим. перев.*

Распределитель, модификатор и сборщик

Модификатор и другие термины, с которыми мы познакомимся в этой главе, являются важными терминами в академических исследованиях по автоматическому управлению памятью. Благодаря строгим определениям мы сможем недвусмысленно распознать их в научных и технических статьях. Например, можно говорить о «накладных расходах модификатора» в конкретном алгоритме. При рассмотрении различных алгоритмов сборки мусора часто обсуждают влияние сборщика на модификатор и наоборот. Познакомимся с этими терминами поближе.

Модификатор

Среди немногих основных понятий, связанных с управлением памятью, самым главным и весьма важным является абстракция *модификатора* (Mutator). В простейшем варианте модификатор можно определить как сущность, отвечающую за выполнение кода приложения. Название связано с тем, что модификатор модифицирует (изменяет) состояние памяти – объекты выделяются или модифицируются, а ссылки между ними изменяются. Иными словами, модификатор стоит за всеми изменениями в приложении, имеющими отношение к памяти. Это название (в числе прочих) предложил Эдсгер Дейкстра в статье 1978 года «On-the-Fly Garbage Collection: An Exercise in Cooperation», где можно найти подробное изложение вопроса. Попутно отметим, что предложение Дейкстра, высказанное в этой довольно старой статье, все еще применяется, например, в языке Go, созданном в 2015 году, и дает неплохие результаты.

Мне нравится абстракция модификатора, поскольку она позволяет элегантно и четко классифицировать сущности в конкретном фреймворке или среде выполнения. Модификатором можно назвать все, что имеет возможность модифицировать память – путем изменения существующих объектов или создания новых. Хотя это не совсем строго, мы можем расширить данное определение, включив в него все, что может читать память (поскольку чтение – важнейшая для выполнения программы операция). Это приводит к важному наблюдению. Полнофункциональный модификатор должен предоставлять работающему приложению три вида операций:

- `New(amount)` – выделить память указанного размера, которая затем будет использована для создания нового объекта. Заметим, что на этом уровне абстракции мы не рассматриваем информацию о типе объекта, которая может предоставляться или не предоставляться средой выполнения. Мы просто выделяем запрошенное количество памяти;
- `Write(address, value)` – записать указанное значение по указанному адресу. Здесь мы также абстрагируемся от того, куда записываем: в поле объекта (в объектно-ориентированном программировании), в глобальную переменную или еще куда-то;
- `Read(address)` – прочитать значение, хранящееся по указанному адресу.

В простейшем мире, где нет никаких алгоритмов сборки мусора, эти три операции реализуются тривиально (псевдокод на C-подобном языке приведен в листинге 1.7).

Листинг 1.7 ❖ Реализация трех главных методов модификатора без автоматического управления памятью

```
Mutator.New(amount)
{
   return Allocator.Allocate(amount);
}

Mutator.Write(address, value)
{
   *address = value;
}

Mutator.Read(address) : value
{
   return *address;
}
```

Но в мире автоматической сборки мусора эти три операции – места, где модификатор кооперируется со сборщиком мусора (*сборщиком*) и механизмом выделения памяти (*распределителем*). Как выглядит эта кооперация и насколько она усложняет показанные выше реализации – один из самых важных вопросов проектирования. Самое распространенное улучшение, которое встретится в этой книге, – это добавление *барьера*, будь то *барьер чтения* или *барьер записи*. Это способ включить дополнительную операцию до или после определенных операций. Барьеры позволяют синхронизироваться (прямо или косвенно, синхронно или асинхронно) с механизмом сборки мусора, чтобы сообщить информацию о выполнении программы и использовании памяти. Методы в листинге 1.7 – это точки, к которым может подключиться любой сборщик мусора. Мы опишем наиболее распространенные вариации в последующих главах, когда будем рассматривать различные алгоритмы сборки мусора.

В повседневной жизни самая часто встречающаяся реализация абстракции модификатора – всем известный *поток*. Он точно отвечает определению: это программная единица, которая выполняет код и изменяет объекты и графы ссылок между ними. Нам это интуитивно понятно, потому что в подавляющем большинстве популярных сред выполнения используется именно эта реализация. В числе прочего функционала поток посредством некоего дополнительного слоя взаимодействует с операционной системой, делая возможными операции `New`, `Write` и `Read`.

Модификаторы необязательно реализовывать как потоки операционной системы. Популярный пример – экосистема языка Erlang с его процессами; они управляются как сверхлегкие сопрограммы, живущие в самой среде выполнения. Их можно считать так называемыми «зелеными потоками», но в терминологии виртуальной машины Erlang лучше называть их «зелеными процессами», поскольку среда выполнения обеспечивает гораздо более сильное разделение, чем между потокоподобными сущностями. Таким образом, управление этими сущностями осуществляется на уровне среды выполнения, а не операционной системы. Другая распространенная реализация модификатора основана на *волокнах* (*fibers*), облегченных единицах выполнения, реализованных и в Linux, и в Windows.

Распределитель

Модификатор должен использовать операцию `New`, которую мы обсудили в предыдущем разделе. Если задуматься о ее внутреннем устройстве, то рано или поздно мы придем еще к одному важному понятию – *распределителю*. Эта сущность отвечает за динамическое выделение и освобождение памяти. Мы уже отмечали, что в древних языках типа ALGOL и FORTRAN распределителя не было, как не было и никакого динамического выделения памяти.

У распределителя две основные операции:

- `Allocator.Allocate(amount)` – выделить указанное количество памяти. Очевидно, сюда можно добавить методы выделения памяти для объекта указанного типа, если распределителю доступна информация о типе. Как мы видели, эта операция используется внутри `Mutator.New`;
- `Allocator.Deallocate(address)` – освободить память по указанному адресу, сделав ее доступной для выделения в будущем. Отметим, что в случае автоматического управления памятью этот метод внутренний, недоступный модификатору (а потому пользовательский код не может вызывать его явно).

Идея может показаться совсем простой, если не сказать тривиальной. Но, как мы увидим, на поверку она совсем не так примитивна. При проектировании распределителя надо учесть массу различных аспектов. И как всегда, повсюду компромиссы, в основном между производительностью, сложностью реализации (а значит, удобством сопровождения) и прочим. Мы подробно рассмотрим два самых популярных вида распределителей: *последовательный* и *список свободных*. Но, поскольку это деталь реализации, уместнее будет рассказать об этом в конкретном контексте .NET в главе 4.

Сборщик

Поскольку мы определили модификатор как сущность, отвечающую за выполнение кода приложения, то можем по аналогии определить сборщик как сущность, которая выполняет код сборки мусора (автоматического освобождения памяти). Иными словами, мы можем рассматривать сборщик либо как программный код, либо как исполняющий его поток, либо как то и другое сразу. Все зависит от контекста.

Откуда сборщик знает, какие объекты больше не нужны и могут быть освобождены? Это неразрешимая задача, потому что для ее решения пришлось бы заглянуть в будущее – собирается ли кто-нибудь использовать объект? Все зависит от кода, который еще только будет выполнен, а это, в свою очередь, может зависеть от таких неизвестных факторов, как действия пользователя, внешние данные и т. д. Идеальный сборщик должен был бы знать *жизнеспособность* объекта – живыми называются объекты, которые понадобятся в будущем. Противоположное понятие – *мертвые* (или *мусорные*) *объекты*, т. е. такие, которые уже не понадобятся и могут быть уничтожены. Таким образом, понятно, что сборщик обычно называют *сборщиком мусора* (Garbage Collector), или кратко – GC.

Отметим интересное следствие кооперации модификатора, распределителя и сборщика. Вспомните, что метод `Allocator.Deallocate` не доступен открыто, т. е. модификатор не может явно освободить полученную память. Модификатор может только запрашивать все новую и новую память, как будто она поступает

из неисчерпаемого источника. По существу, это означает, что механизм сборки мусора моделирует компьютер с бесконечной памятью. Как эта модель работает и насколько она эффективна – деталь реализации.

Можно представить себе специальный сборщик мусора, который вообще не освобождает выделенную память. Он называется *нулевым*, или *пустым*, *сборщиком мусора*. Он правильно работал бы только на компьютерах с бесконечным объемом памяти, каковых, к сожалению, пока не существует. Но и пустой сборщик мусора не совсем бесполезен. Его можно, например, использовать в программах, работающих очень недолго, когда неограниченный рост памяти приемлем. Быть может, такие сборщики мусора станут популярными в мире бессерверных одиночных короткоживущих функций. Пример пустого сборщика мусора для .NET приведен в главе 15.

Поскольку узнать жизнеспособность объекта невозможно[1], сборщик опирается на менее строгое свойство объекта – *достижимость* хотя бы одним модификатором. Объект называется *достижимым*, если существует последовательность ссылок (начинающаяся в памяти, доступной хотя бы одному модификатору) между объектами, которая приводит к данному объекту (рис. 1.12). Очевидно, что достижимость не означает, что объект жизнеспособен, но это наилучшее возможное приближение. Если объект не достижим ни одним модификатором, то его невозможно использовать, т. е. он мертв и может быть безопасно очищен. Обратное не всегда верно. Достижимый объект может вечно оставаться достижимым (его удерживает какой-то сложный граф ссылок), но выполнение устроено так, что к нему никогда не будет обращений, поэтому он все равно что мертв. На самом деле именно в пограничной области между жизнеспособностью и достижимостью происходит большинство утечек управляемой памяти.

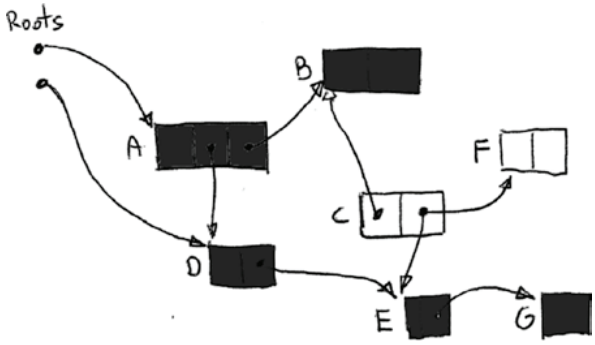


Рис. 1.12 ❖ Достижимость – объекты C и F недостижимы, потому что к ним нет пути от корней (адресов в памяти модификатора)

Когда говорят о достижимости, начальные точки в области памяти модификатора называются *корнями*. Что они собой представляют в действительности, зависит от реализации модификатора. Но в наиболее распространенных случаях,

[1] В главе 4 мы будем обсуждать анализ локальности – метод определения истинной жизнеспособности указателей, по крайней мере в некоторых специальных случаях.

когда модификатор – это просто поток (представленный потоком операционной системы), корнями могут быть:

- локальные переменные и аргументы подпрограмм, размещенные в стеке или в регистрах;
- статически выделенные объекты (например, глобальные переменные), размещенные в куче;
- другие внутренние структуры данных, хранящиеся в самом сборщике.

Зная о трех основных структурных элементах – модификаторе, распределителе и сборщике, – мы можем перейти к знакомству с разнообразными подходами к автоматическому управлению памятью. И хотя есть искушение представить полный список с детальным описанием каждого подхода, книга слишком мала для этого. Вместо этого мы изучим некоторые из наиболее популярных подходов, встречающихся в современных языках.

Подсчет ссылок

Один из двух самых популярных методов автоматического управления памятью называется *подсчетом ссылок*. Идея очень проста. Нужно подсчитывать количество ссылок на каждый объект. В каждом объекте хранится свой *счетчик ссылок*. Когда объект присваивается переменной или полю, количество ссылок на него увеличивается. Одновременно счетчик ссылок в объекте, на который раньше указывала эта переменная, уменьшается.

Жизнеспособность объекта при таком подходе оценивается количеством ссылающихся на него объектов. Если счетчик ссылок обращается в ноль, то на объект никто не ссылается, поэтому его можно освободить. Но что, если счетчик не равен нулю? Это ничего не говорит о жизнеспособности, а означает лишь, что кто-то хранит ссылку на объект, хотя, быть может, и не использует его. Поэтому подсчет ссылок – еще один не особенно строгий способ судить о гипотетической жизнеспособности объекта.

Вернемся к нашему тривиальному примеру модификатора в листинге 1.7. В случае подсчета ссылок его можно было бы написать, как показано ниже.

Листинг 1.8 ❖ Псевдокод, описывающий простой алгоритм подсчета ссылок

```
Mutator.New(amount)
{
   obj = Allocator.Allocate(amount);
   obj.counter = 0;
   return obj;
}

Mutator.Write(address, value)
{
   if (address != NULL)
   ReferenceCountingCollector.DecreaseCounter(address);
   *address = value;
   if (value != NULL)
      value.counter++;
}
```

```
ReferenceCountingCollector.DecreaseCounter(address)
{
   *address.counter--;
   if (*address.counter == 0)
   Allocator.Deallocate(address)
}
```

Поведение подсчета ссылок иллюстрируется простой программой, показанной на рис. 1.13 и в листинге 1.9. Три строки кода переписаны в терминах методов модификатора, чтобы показать, как изменяются ссылки.

Листинг 1.9 ❖ Псевдокод, иллюстрирующий подсчет ссылок

```
o1 = new SomeObject();
o2 = new SomeObject();
o2 = o1;

// транслируется в:

addr1 = Mutator.New(SizeOf(SomeObject))  // addr1.counter = 0
Mutator.Write(&o1, addr1)                // addr1.counter = 1
addr2 = Mutator.New(SizeOf(SomeObject))  // addr2.counter = 0
Mutator.Write(&o2, addr2)                // addr2.counter = 1
Mutator.Write(&o2, &o1)                  // addr1.counter = 0; addr2.counter = 2
```

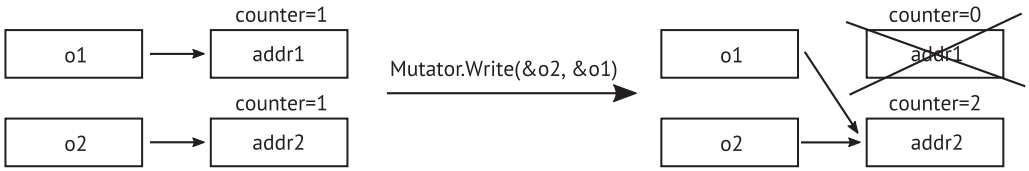


Рис. 1.13 ❖ Иллюстрация подсчета ссылок программе из листинга 1.9

Как видно из листинга 1.9, в операцию `Mutator.Write` добавлены большие накладные расходы. Она должна проверять и изменять счетчик и выполнять освобождение, если тот обращается в ноль. В многозадачной среде (когда параллельно работает несколько модификаторов) эта задача существенно усложняется. Чтобы операции были потокобезопасными, необходимо включить синхронизацию со свойственными ей накладными расходами. Операция `Mutator.Write` встречается очень часто (при любом присваивании), поэтому любые накладные расходы в ней негативно сказываются на времени выполнения программы в целом. К тому же, с точки зрения реализации, не вполне понятно, где хранить счетчики ссылок на объект. Это может быть какая-то выделенная область памяти или некий заголовок, хранящийся настолько близко к объекту, насколько возможно. Но в любом случае каждое присваивание порождает лишнюю запись в память, что крайне нежелательно. Помимо всего прочего, это может привести к неэффективному использованию кеша ЦП, но об этой теме мы будем говорить в следующей главе.

Возвращаясь к вышеупомянутому свойству достижимости, можно сказать, что подсчет ссылок аппроксимирует жизнеспособность по локальным ссылкам, но не пытается отслеживать глобальное состояние графа ссылающихся друг на друга

объектов. В частности, если не принять дополнительных мер, то этот алгоритм будет введен в заблуждение циклическими ссылками, каковые встречаются в таких популярных структурах данных, как двусвязные списки (рис. 1.14). В таком случае счетчик ссылок никогда не обратится в ноль, потому что элементы списка со значениями `value1` и `value2` указывают друг на друга.

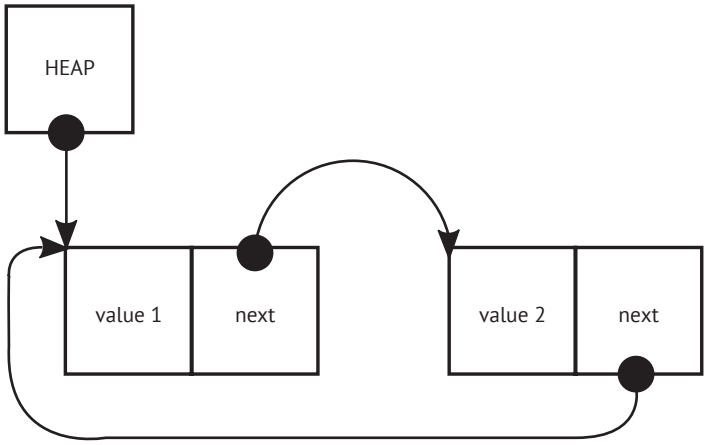


Рис. 1.14 ❖ Проблема подсчета ссылок в случае с циклическими ссылками

Однако создание циклических ссылок можно затруднить на уровне языка, что обнадеживает. В таком случае алгоритм подсчета ссылок можно использовать, не заботясь об утечках памяти, вызванных этой проблемой.

Большим преимуществом и причиной популярности подсчета ссылок является тот факт, что он не требует поддержки со стороны среды выполнения. Его можно реализовать во внешней библиотеке в виде дополнительного механизма для некоторых специальных типов. Следовательно, изначальные методы `Mutator.New` и `Mutator.Write` можно не трогать, а просто ввести высокоуровневые аналоги, например классы с подходящим образом перегруженными операторами и конструкторами. Именно так обстоит дело в большинстве популярных реализаций C++.

Были введены так называемые *умные (интеллектуальные) указатели*, более сложно управляющие жизнью объектов, на которые они указывают. С точки зрения реализации, умные указатели в C++ – это просто шаблоны классов, которые ведут себя как обычные указатели благодаря перегрузке операторов. В C++ есть два вида таких указателей:

- `unique_ptr` реализует семантику монопольного владения (указатель является единственным владельцем объекта, который уничтожается, как только `unique_ptr` покидает область видимости или ему присваивается другой объект);
- `shared_ptr` реализует семантику подсчета ссылок.

С помощью умных указателей код на C++ из листинга 1.5 можно переписать следующим образом:

Листинг 1.10 ❖ Программа на C++, демонстрирующая автоматизированное управление памятью с помощью умных указателей

```
#include <iostream>
#include <memory>

void printReport(std::shared_ptr<int> data)
{
   std::cout << "Отчет: " << *data << "\n";
}

int main()
{
   try
   {
      std::shared_ptr<int> ptr(new int());
      *ptr = 25;
      printReport(ptr);
      return 0;
   }
   catch (std::bad_alloc& ba)
   {
      std::cout << "ОШИБКА: недостаточно памяти\n";
      return 1;
   }
}
```

Если бы мы вызвали метод `data.use_count()` из функции `printReport`, то он вернул бы значение 2, потому что внутри этой функции два разных разделяемых указателя (shared pointers) указывают на один и тот же объект. С другой стороны, после выхода из блока `try` счетчик ссылок будет равен 0, т. к. не осталось ни одного умного указателя на наш объект.

> Отметим, что код в листинге 1.10 не отвечает хорошему стилю программирования на C++. Если требуется только прочитать данные, то лучше передавать умный указатель на них по константной ссылке (`const&`), а не по значению, но тогда счетчик ссылок не увеличился бы, так что пример был бы бесполезен с педагогической точки зрения.

Этот код представляет собой значительное усовершенствование, поскольку:

- нам не нужно вручную уничтожать объект оператором `delete`;
- упрощается обработка исключений, т. к. если функция `printReport()` возбуждает исключение, то умный указатель выйдет из области видимости блока `try` (и всех вложенных в него), поэтому будет автоматически уничтожен. Это следствие вышеупомянутой идиомы *RAII* (*получение ресурса есть инициализация*) – время жизни объекта зависит от области видимости указателя на представляющую его переменную.

Разделяемые и уникальные указатели можно использовать также в полях классов, что делает их весьма мощным и полезным средством.

Проблема в том, что в C++ умные указатели реализованы на уровне библиотеки, а не самого языка. В других библиотеках имеются свои реализации, и иногда взаимодействие между ними затруднительно. В библиотеке Qt имеется класс `QtSharedPointer`, в wxWidgets – `wxSharedPtr<T>` и т. д. Без поддержки со стороны языка

и компилятора это попросту неизбежно. Именно поэтому автоматическое управление памятью так важно в компонентной среде[1] программирования, каковой является .NET. При проектировании .NET передача ответственности за управление памятью от разработчика самой среде выполнения была одним из принципиальных решений. Единый механизм создания, управления и освобождения объектов означает, что все компоненты будут использовать его одним и тем же способом и между компонентами не будет никакой другой связи, кроме самой среды выполнения.

Раз уж речь зашла о C++, интересно отметить, что Бьёрн допустил в стандарте более продвинутый сборщик мусора – он не запрещен, но пока не реализован. Более того, благодаря гибкости C++ сборку мусора можно реализовать в виде дополнительной библиотеки – Memory Pool System или сборщика Бема–Демерса–Вейзера. Мы поговорим об этом чуть ниже.

В других языках умные указатели (включающие подсчет ссылок) могут быть встроены непосредственно, именно так обстоит дело в Rust – современном низкоуровневом языке программирования, созданном в корпорации Mozilla. Он гарантирует безопасность данных на уровне компиляции за счет встраивания концепции умного указателя (на самом деле есть несколько их разновидностей) в сам язык. В языке реализована строгая семантика владения и принцип RAII, что позволяет во время компиляции проверять отсутствие таких нарушений, как разыменование висячего указателя. Другое известное применение подсчета ссылок – встроенный механизм автоматического подсчета ссылок в языке Swift.

Ниже перечислены достоинства и недостатки подсчета ссылок.

Достоинства:

- детерминированный момент освобождения – мы знаем, что освобождение произойдет, когда счетчик ссылок на объект обратится в ноль. Поэтому память будет возвращена системе, как только в ней отпадет необходимость;
- меньшее потребление памяти – поскольку память освобождается сразу, как только объект перестает использоваться, устраняется перерасход памяти из-за того, что она остается занятой в ожидании сборки мусора;
- можно реализовать без поддержки со стороны среды выполнения.

Недостатки:

- наивная реализация типа указанной в листинге 1.8 приводит к очень большим накладным расходам модификатора;
- многопоточные операции со счетчиками ссылок требуют продуманной синхронизации, которая может повлечь дополнительные издержки;
- без дополнительных мер невозможно освободить объекты, связанные циклическими ссылками.

Существуют усовершенствования наивных алгоритмов подсчета ссылок, например *отложенный подсчет ссылок* и *объединенный подсчет ссылок*, которые решают некоторые из этих проблем ценой утраты тех или иных преимуществ (в основном жертвуя немедленным возвратом памяти). Но их описание выходит за рамки этой книги.

---

[1] Когда имеются небольшие взаимозаменяемые зависимости.

Отслеживающий сборщик

Определить достижимость объекта трудно, потому что это глобальный атрибут объекта (он зависит от графа объектов всей программы), а простой явный вызов освобождения объекта – вещь очень локальная. В этом локальном контексте мы ничего не знаем о глобальном – есть ли еще какие-нибудь объекты, пользующиеся данным объектом? Подсчет ссылок – попытка решить эту проблему, снабдив локальный контекст дополнительной информацией – количеством ссылок на объект. Но, очевидно, это может приводить к проблемам при наличии циклических ссылок и, как мы уже видели, имеет ряд других недостатков.

Отслеживающий сборщик мусора опирается на знание глобального контекста времени жизни объекта и может принять более обоснованное решение о том, пора ли уже удалить объект (вернуть память). На самом деле этот подход настолько популярен, что, говоря о сборщике мусора, почти наверняка имеют в виду именно отслеживающий сборщик. Он встречается в .NET, различных реализациях JVM и других средах выполнения.

Основная идея отслеживающего сборщика мусора заключается в том, чтобы определить истинную достижимость объекта, начав с корней модификатора и рекурсивно обойдя весь граф объектов, созданных программой. Очевидно, что это нетривиальная задача, т. к. память процесса может занимать несколько гигабайтов, а проследить все ссылки между объектами при таком объеме данных трудно, особенно если принять во внимание, что модификаторы все время работают и ссылки изменяются. Работа типичного отслеживающего сборщика мусора состоит из двух шагов:

- *пометка* – на этом шаге сборщик определяет достижимость объектов в памяти и решает, какие из них можно убрать в мусор;
- *сборка* – на этом шаге сборщик возвращает системе память, занятую недостижимыми объектами.

Этот простой двухшаговый алгоритм можно расширить, именно так и сделано в .NET, где алгоритм можно описать как пометка–планирование–очистка–уплотнение. Как это устроено, мы подробно обсудим в следующих главах. А пока рассмотрим шаги пометки и сборки в общем виде, поскольку даже это позволяет поднять интересные вопросы.

Этап пометки

На этапе пометки сборщик определяет, какие из находящихся в памяти объектов следует убрать, для чего вычисляет их достижимость. Начав с корней модификатора, сборщик обходит весь граф объектов и помечает те, которые посетил. Объекты, которые остались непомеченными в конце этого этапа, недостижимы. Пометка устраняет проблему циклических ссылок. Если во время обхода графа мы наткнемся на ранее посещенный объект, то просто продолжим обход, как ни в чем не бывало, поскольку этот объект уже помечен.

На рис. 1.15 показано несколько начальных шагов этого алгоритма. Начав с корней, мы движемся внутрь графа объектов по ссылкам между объектами. Как производится обход – в глубину или в ширину, – деталь реализации; на рис. 1.15 показан обход в глубину. Объект может находиться в одном из трех состояний:

- еще не посещен, изображен белым квадратиком;
- помещен в очередь подлежащих посещению, изображен серым квадратиком;
- уже посещен (помечен как достижимый), изображен черным квадратиком.

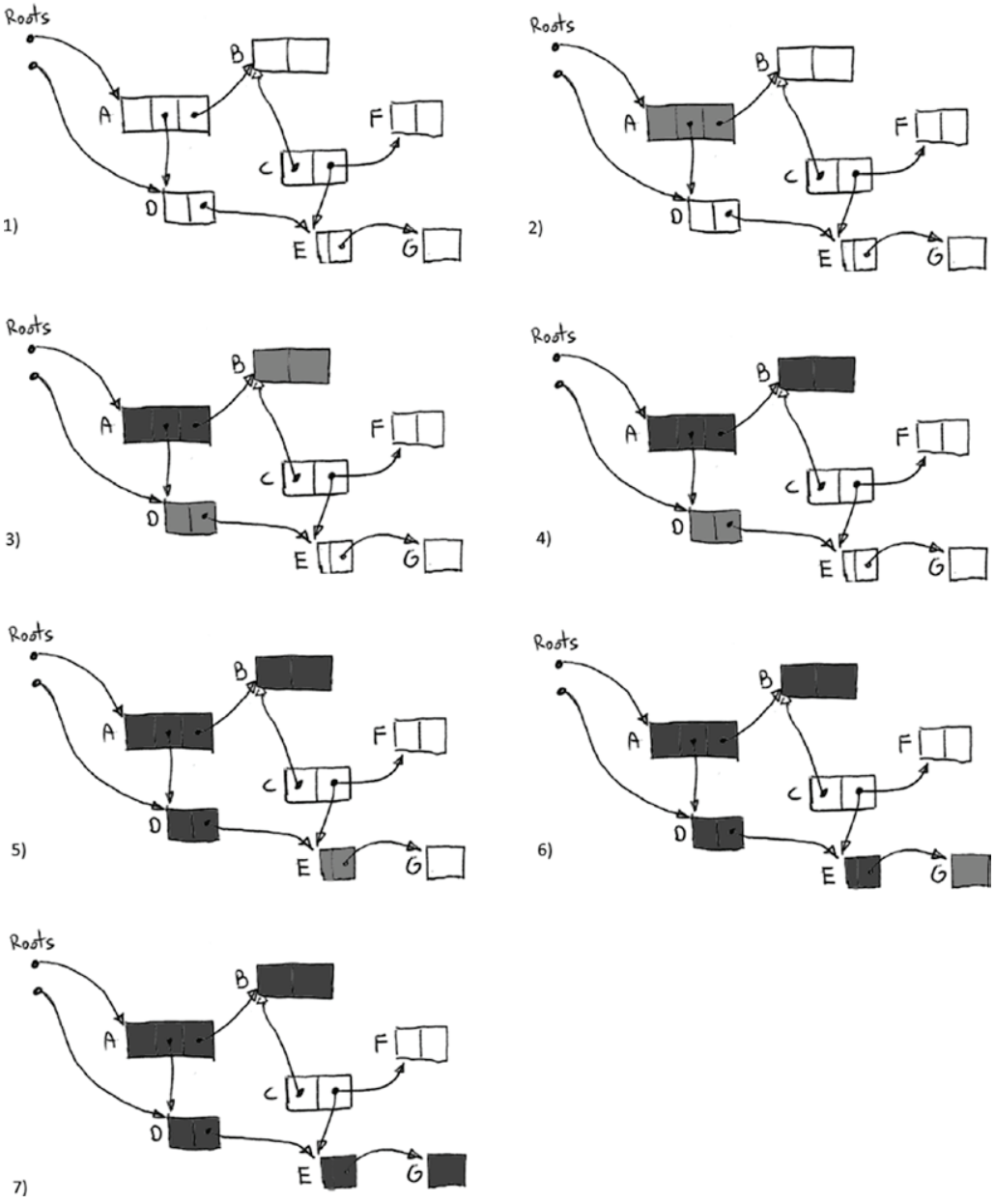


Рис. 1.15 ❖ Несколько начальных шагов фазы пометки

Показанные на рис. 1.15 начальные шаги можно описать словами следующим образом (каждый шаг соответствует одной части рисунка):

1. В начальный момент ни один объект еще не посещен.
2. Объект A запланирован к посещению как первый корень.
3. Поскольку из объекта A ведут указатели (хранящиеся в его полях) на объекты B и D, то они тоже ставятся в очередь на посещение. Сам объект A на этом шаге помечается как достижимый.
4. Посещается следующий объект из очереди ожидающих посещения – объект B. Поскольку из него не выходит ни одной ссылки, он просто помечается как достижимый.
5. Посещается следующий объект из очереди ожидающих посещения – объект D. Он содержит одну ссылку на объект E, который помещается в очередь. Сам объект D помечается как достижимый.
6. Объект G, на который ведет ссылка из объекта E, помещается в очередь на посещение. Сам объект E помечается как достижимый.
7. Посещается последний объект из очереди ожидающих посещения – G. Из него не выходит ссылок, поэтому он просто помечается как достижимый. Больше объектов, требующих посещения, не осталось, поэтому мы выяснили, что объекты C и F недостижимы (мертвы).

Понятно, что обходить такой граф во время нормальной работы модификатора трудно, т. к. он постоянно изменяется как результат работы программы – создаются новые объекты и переменные, полям объектов присваиваются значения и т. д. Поэтому в некоторых реализациях сборщика мусора все модификаторы просто приостанавливаются на время работы фазы пометки. Это позволяет обойти граф безопасно и согласованно. Конечно, как только потоки возобновят выполнение, знания сборщика о графе объектов тут же устаревают. Но это не касается недостижимых объектов – коль скоро объект один раз оказался недостижимым, он уже никогда не станет достижимым. Однако существует немало реализаций сборщика мусора, в которых этап пометки конкурентный, то есть может выполняться одновременно с кодом модификатора. Так устроены популярные алгоритмы CMS в JVM (Concurrent Mark Sweep – конкурентная пометка и очистка), G1 в JVM и сборка мусора в самой среде .NET. Как именно работает конкурентная пометка в .NET, будет подробно описано в главе 11.

На этапе пометки есть одна неочевидная проблема. Для отслеживания достижимости сборщик должен знать о корнях и о том, где в памяти находятся ссылки на другие объекты. Это тривиально, если среда выполнения поддерживает такую информацию. Но есть и обходные пути решения этой проблемы.

Консервативный сборщик мусора

Этот тип сборщика можно считать решением на крайний случай. Его стоит использовать, когда ни среда выполнения, ни компилятор не поддерживают сборку напрямую, т. е. не предоставляют точную информацию о типе (размещение объекта в памяти), и сборщик не получает никакой поддержки от модификатора при работе с указателями. Когда так называемый *консервативный сборщик* хочет узнать, какие объекты достижимы, он просматривает весь стек, области статических данных и регистры. Поскольку без помощи извне он не знает, что является указателем, а что нет, он просто пытается угадать. Для этого сборщик проверяет

несколько условий (какие именно, зависит от реализации), но самое главное из них такое: верно ли, что данное слово, интерпретируемое как адрес (указатель), соответствует действительной, управляемой распределителем области кучи? Если да, то сборщик на всякий случай, *консервативно* (отсюда и название), предполагает, что это в самом деле указатель. И обращается с ним как со ссылкой, по которой нужно проследовать в процессе описанного выше обхода графа на этапе пометки.

Конечно, догадка сборщика может оказаться неверной, поскольку случайные комбинации битов могут выглядеть как указатель на действительный адрес. Тогда мусорная память не будет возвращена системе. Это не слишком частая проблема, потому что числовые значения в памяти обычно малы (счетчики, финансовые данные, индексы), поэтому неприятности могут возникнуть при работе с плотными двоичными данными: растровыми изображениями, числами с плавающей точкой или блоками IP-адресов[1]. Существуют хитроумные улучшения алгоритма, помогающие преодолеть эту проблему, но здесь мы их рассматривать не будем. Кроме того, консервативный подход означает, что объекты нельзя перемещать в памяти. В самом деле, это потребовало бы изменения указателей на перемещенный объект, что, очевидно, невозможно, коль скоро мы не уверены, что нечто, похожее на указатель, действительно таковым является.

Так кому же может понадобиться такой сборщик? Его главное достоинство – возможность работать без поддержки со стороны среды выполнения. По сути дела, он просто просматривает память, так что в поддержке среды выполнения (прослеживании ссылок) не нуждается. Поэтому он удобен, например, при разработке новой среды выполнения, когда полная информация о типах для GC еще отсутствует. Это позволяет заниматься другими частями системы, не останавливая процесс разработки. Майкрософт применяла такую тактику при разработке некоторых версий своей среды выполнения[2].

Однако консервативный сборщик нуждается в поддержке распределителя, чтобы решить проблемы, связанные с неизвестным размещением объекта в памяти. Например, можно организовать выделение памяти таким образом, что объекты одинакового размера будут сгруппированы в сегменты. Консервативный просмотр таких участков возможен, потому что границы объекта определяются путем простого умножения на размер объекта в сегменте.

Во многих языках распределитель можно заменять на уровне языка (библиотеки), что способствует популярности реализации консервативной сборки мусора в виде библиотеки. Одна из самых распространенных реализаций для C и C++, безразличная к API, – *сборщик мусора Бема–Демерса–Вейзера* (или просто *GC Бема*).

Он использовался, например, в Mono (реализация CLR с открытым исходным кодом) вплоть до версии 2.8 (2010 год), а затем был заменен сборщиком мусора *SGen*, в котором реализован смешанный подход – стек и регистры по-прежнему

---

[1] В сборщике мусора Бема и в других консервативных GC разрешается при выделении блока памяти задавать специальный флаг (в случае сборщика Бема – `GC_MALLOC_ATOMIC`), который говорит сборщику, что этот блок не содержит указателей и просматривать его не нужно. В таком блоке можно хранить плотные двоичные данные, например растровые изображения.

[2] Интересно, что .NET содержит реализацию консервативного сборщика мусора, хотя по умолчанию она не включена.

просматриваются консервативно, а просмотр кучи поддерживается информацией о типе во время выполнения.

Коротко перечислим основные свойства консервативного сборщика мусора.

Достоинства:

- подходит для сред, изначально не поддерживающих сборку мусора – например, на ранних этапах разработки среды выполнения и в неуправляемых языках.

Недостатки:

- неточность – все, что случайно выглядит как действительный указатель, препятствует освобождению памяти, хотя это нечастая ситуация, которую можно предотвратить улучшением алгоритма и заданием дополнительных флагов;
- при наивном подходе объекты нельзя перемещать (уплотнять), поскольку сборщик не уверен, что в действительности является указателем (и не может взять и изменить значение только на основании догадки).

Точный сборщик мусора

В так называемом точном сборщике мусора все значительно проще, потому что компилятор и (или) среда выполнения предоставляют сборщику полную информацию о размещении объекта в памяти. Среда может также поддерживать обход стека (перечисление всех корневых объектов в стеке). В таком случае не остается места догадкам. Начав с точно известных корней, сборщик просто просматривает память объект за объектом. Зная, где в памяти находится указатель на начало объекта (или располагая указателем внутрь объекта плюс информацией о том, как такую ссылку интерпретировать), сборщик может просто рекурсивно обойти граф объекта, следуя по таким ссылкам.

В .NET используется точный сборщик мусора, поэтому в следующих главах мы гораздо подробнее узнаем, как он устроен. Собственно, главы с 7 по 10 целиком посвящены этому вопросу.

Этап сборки

После того как отслеживающий сборщик мусора нашел достижимые объекты, он может вернуть системе память, занятую всеми остальными, мертвыми объектами. Этап сборки можно спроектировать по-разному, поскольку у него много аспектов. В одном коротком разделе невозможно описать все возможные комбинации и варианты, но можно и нужно выделить два главных подхода, лежащих в основе различных реализаций.

Очистка

В этом подходе мертвые объекты просто помечаются как свободное пространство, которое можно повторно использовать позже. Эта операция может быть очень быстрой, поскольку (в простейшей реализации) нужно изменить всего один бит в блоке памяти. Такая ситуация показана на рис. 1.16, где неиспользуемые объекты C и F (см. рис. 1.15) становятся доступной памятью после их пометки свободными.

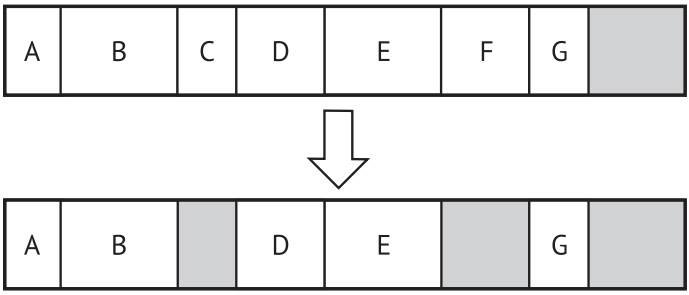


Рис. 1.16 ❖ Сборка очисткой – наивная реализация

Впоследствии в наивной реализации выделения вся память просматривается в поисках свободного участка, размер которого не меньше размера создаваемого объекта.

Но в нетривиальной реализации необходима структура данных, в которой хранится информация о свободных блоках памяти, чтобы их можно было находить быстрее. Обычно для этого используется *список свободных блоков* (рис. 1.17). Более того, эти списки свободных блоков должны быть достаточно умными, чтобы объединять соседние блоки памяти. Возможна и дальнейшая оптимизация – вести несколько списков свободных блоков для блоков памяти разных размеров. Существуют также различные способы просмотра таких списков. Два самых популярных: *лучшее соответствие* и *первое соответствие*. Если используется метод первого соответствия, то просмотр прекращается, как только найден свободный блок подходящего размера. Метод лучшего соответствия подразумевает полный просмотр всего списка свободных блоков с целью найти лучший блок для запрошенного размера. Первый метод быстрее, но может увеличивать фрагментацию, со вторым все наоборот.

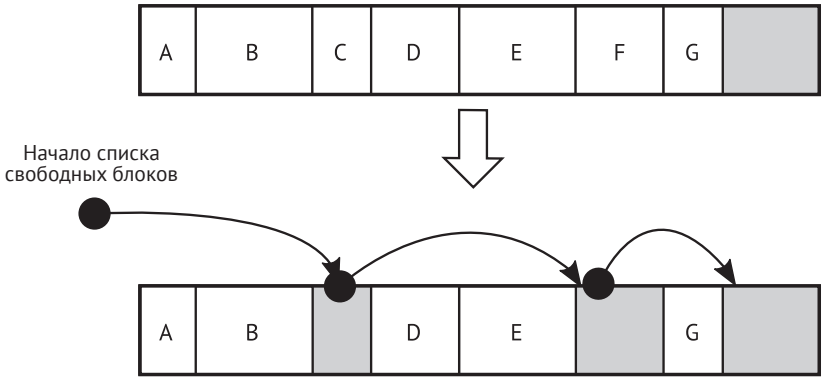


Рис. 1.17 ❖ Сборка очисткой – реализация списка свободных блоков

Хотя подход на основе очистки довольно быстрый, у него есть крупный недостаток – в конечном итоге он приводит к более или менее серьезной фрагментации. По мере создания и уничтожения объектов в куче образуется все больше

мелких или больших свободных участков. Это может привести к ситуации, когда для создания нового объекта памяти достаточно, но вся она распределена между мелкими блоками, а единого сплошного участка нужного размера нет. Мы видели такую ситуацию на рис. 1.11, когда описывали выделение из кучи вообще.

Уплотнение

При таком подходе фрагментация устраняется ценой снижения производительности, потому что требуется перемещать объекты в памяти. Объекты перемещаются таким образом, чтобы убрать пропуски, оставшиеся на месте удаленных объектов. И тут можно, в свою очередь, выделить два разных подхода.

Более простое, с точки зрения реализации, *уплотнение копированием* заключается в том, что все живые (достижимые) объекты копируются в другую область памяти при каждой сборке мусора (рис. 1.18). Уплотнение сводится к простой последовательности операций копирования одного объекта за другим, причем мертвые объекты пропускаются. Очевидно, что это вызывает интенсивное использование памяти (memory traffic), поскольку все живые объекты постоянно перемещаются туда-сюда. К тому же предъявляются повышенные требования к объему памяти, т. к. на время перемещения нужно в два раза больше памяти, чем при нормальной работе.

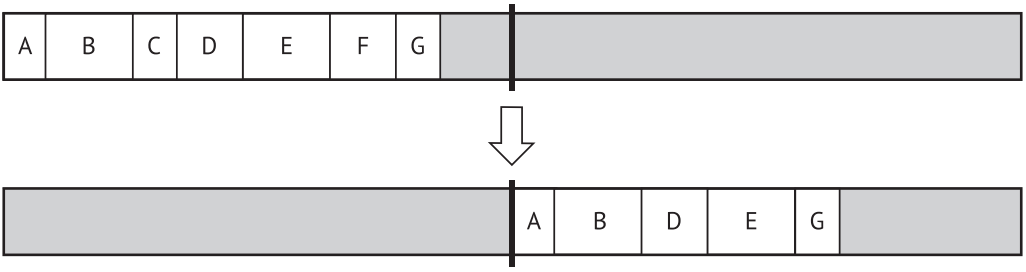


Рис. 1.18 ❖ Сборка уплотнением – реализация путем копирования

Может сложиться впечатление, что из-за этих недостатков алгоритм не имеет практической ценности. Однако его можно использовать эффективно. Нужно только применять его к некоторым небольшим областям памяти, а не ко всей памяти процесса в целом. Именно так делается в некоторых реализациях JVM.

Есть и более сложный алгоритм – *уплотнение на месте*. Объекты сдвигаются друг к другу, так чтобы устранить промежутки между ними (рис. 1.19). Это интуитивно самое естественное решение, так мы поступаем, когда двигаем детали в конструкторе Lego. С точки зрения реализации, это нетривиально, но возможно. Главный вопрос – как перемещать объекты, не затирая один другим и не прибегая к временному буферу?

Как мы увидим в главе 9, в .NET применяется именно этот подход с очень интересной структурой данных для оптимизации, там мы и дадим ответ на этот вопрос.

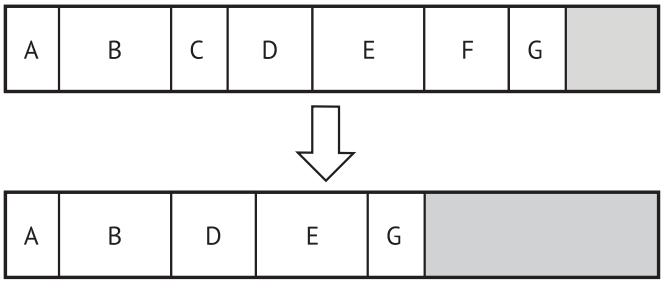


Рис. 1.19 ❖ Сборка уплотнением – реализация на месте

Сравнение сборщиков мусора

Можно задать вопрос: какой сборщик мусора лучше? HotSpot Java 1.8 или .NET 4.6? А быть может, лучше сборщики мусора в Python или Ruby? И что вообще означает «лучше» для GC? Первое и самое важное правило сравнения алгоритмов сборки мусора – всякое сравнение с самого начала необъективно. Дело в том, что GC очень трудно выделить и сравнивать сами по себе. Они настолько переплетены со средой выполнения, что тестировать их по отдельности практически невозможно. Поэтому провести по-настоящему честное сравнение затруднительно. Если бы мы захотели сравнить производительность разных GC, то стали бы измерять такие показатели, как пропускная способность, задержка, время приостановки (в главе 3 мы объясним, что все это такое). Но все они берутся в контексте среды выполнения в целом, а не только GC. Механизмы фреймворка или среды выполнения (к примеру, паттерны выделения, внутренний пул объектов, дополнительная компиляция и прочие скрытые вещи) могут привносить такие накладные расходы, что вклад GC в общую производительность окажется пренебрежимо мал. Кроме того, для любого GC существуют многочисленные настройки, позволяющие ему показывать лучшую производительность на определенных рабочих нагрузках. Одни можно оптимизировать для уменьшения времени реакции в интерактивной среде, другие – для обработки очень больших наборов данных. Третьи могут динамически подстраивать свои характеристики под текущую рабочую нагрузку. Более того, разные GC могут вести себя по-разному в зависимости от конфигурации оборудования (оптимизированы для конкретной архитектуры процессора, количества процессорных ядер или архитектуры памяти).

Конечно, можно сравнивать GC по использованным алгоритмам и предоставляемой функциональности. Есть и еще много способов классификации сборщиков мусора. Как мы уже видели, CG может быть консервативным (Mono до версии 2.8), или точным (.NET), или даже смешанным (Mono 2.8+). В одних реализована сборка очисткой, в других – уплотнением, а в третьих – обоими методами. Еще одно важное различие – как GC делит память на области. В главе 5 мы увидим, как можно разбить кучу на меньшие части. Подсчет ссылок может использоваться иногда или вообще не использоваться. Как реализован распределитель? Является ли GC параллельным или конкурентным (глава 11)? При таком изобилии функциональных различий очень трудно сказать, какая комбинация «лучше». Остановимся на том, что идеального во всех смыслах решения не существует.

Приведем краткий перечень достоинств и недостатков отслеживающего сборщика мусора.

Достоинства:

- полная прозрачность с точки зрения разработчика – память просто представляется бесконечной, и не нужно думать об освобождении памяти, занятой уже не используемыми объектами;

- отсутствуют проблемы из-за циклических ссылок;
- не сильно обременяет модификаторы дополнительными накладными расходами.

Недостатки:

- более сложная реализация;
- недетерминированное освобождение объектов – они освобождаются спустя некоторое время, после того как стали недостижимыми;
- вся работа прекращается (stop the world), когда выполняется этап пометки, – но только в случае неконкурентной реализации;
- повышенный расход памяти – поскольку объекты освобождаются не сразу, потребление памяти может возрастать (мусор продолжает занимать память в течение некоторого времени).

Отслеживающий GC популярен в различных средах выполнения в основном благодаря первому преимуществу.

Немного истории

Заложив основательный теоретический фундамент, бросим беглый взгляд на историю автоматического управления памятью в контексте различных языков программирования.

LISP – один из самых старых до сих пор живущих языков, за время существования которого сменилось множество диалектов, наиболее популярные из которых – Common LISP и Scheme. Но сейчас, без сомнения, шире всего распространен диалект Clojure, который компилируется, среди прочих, для виртуальной машины Java, общеязыковой среды выполнения (.NET) и JavaScript. Это делает его очень гибким и мощным, и, конечно, в этом воплощении LISP встречается со сборкой мусора.

Но не только функциональные языки типа LISP могли похвастаться автоматизированным управлением памятью во времена своего расцвета. Никакой исторический обзор не будет полон без упоминания еще одного языка, оказавшего чрезвычайно сильное влияние, – Simula. Его называют первым по-настоящему объектно-ориентированным языком, в нем впервые появились понятия объектов и классов, наследования, полиморфизма и другие столпы ООП. Все языки, начиная со Smalltalk, а затем через C++ к Java и C#, Python и Ruby, так или иначе черпали идеи из Simula. Что важно, в Simula 67 появилось автоматическое управление памятью, которое поначалу было реализовано с помощью подсчета ссылок и отслеживающего сборщика мусора, а с развитием языка было заменено уплотняющим сборщиком мусора по образцу применяемого в LISP. Благодаря языку Smalltalk сборка мусора приобрела популярность у проектировщиков языков. Возрастание сложности программного обеспечения подталкивало проектировщиков к введению различных по степени изощренности способов освободить программиста от управления памятью.

Популярность веба и начало эры интернета в 1990-х годах заставили индустрию разработки ПО обратиться к программированию на более высоком уровне. Времена беспрекословного доминирования C и C++ миновали. Их способность к управлению системами на низком уровне не имела никакой ценности в контексте веб-программирования и бурного роста серверных приложений. Вместе

с ошеломляюще быстрым развитием интернета возрастала и сложность веб-приложений, и потребность быстрее разрабатывать код.

Невозможно изложить историю автоматического управления памятью, не упомянув языка и платформы Java. Этот язык был задуман компанией Sun как «улучшенный C++», а сборка мусора стала одним из первых и самых фундаментальных требований к новой платформе. Начиная с 1990-х годов, когда проект стартовал под видом внутреннего языка Oak, в нем уже присутствовал механизм пометки и очистки. Первая общедоступная версия Java 1.0a была анонсирована в 1994 году. Взрывной рост популярности Java стал причиной широкой осведомленности о механизмах сборки мусора. Начиная с того времени автоматическое управление памятью стало почти само собой разумеющимся для всех проектировщиков языков высокого уровня.

Уже после рождения Java на свет появились еще два широко распространенных языка: Python и Ruby. Оба были оснащены автоматическим управлением памятью по описанным выше причинам. В Python вплоть до версии 2.0 существовал только подсчет ссылок, но затем были добавлены более сложные способы обращения с циклическими ссылками. Ruby предлагает более простой механизм, основанный на пометке и очистке.

В нашем историческом обзоре нельзя обойти вниманием язык JavaScript, появившийся в те же годы, что и Java. И хотя созвучие с названием Java было больше маркетинговым трюком, чем реальным сходством, JavaScript тоже был задуман как скриптовый язык высокого уровня. В нем не было места ручному управлению памятью. Его целью было манипулирование HTML-содержимым на высоком уровне, не задумываясь о таких вещах, как использование памяти. За это отвечала среда выполнения JavaScript. По мере появления сценариев, в которых программы на JavaScript работали длительное время – одностраничные приложения и серверные программы на node.js, – важность автоматической сборки мусора в движках JavaScript только возрастала. Например, в очень популярном движке V8, который применяется в node.js, используется подход на основе пометки, очистки и уплотнения с дополнительными оптимизациями.

Таким образом, можно отметить, что хотя языки с автоматическим управлением памятью существуют уже 50 лет, настоящий взлет их популярности пришелся на 1990-е годы. И это то место, где можно перейти к истории самой важной и интересной для нас среды – .NET Framework.

Но еще важнее, что в то время Майкрософт уже разработала собственную реализацию JavaScript под названием JScript. JScript играет важную роль в нашей истории, потому что заложил фундамент для решений, использованных при создании .NET. Конечно, больше всего нас интересует вопрос об управлении памятью. На самом деле все началось с языка JScript, написанного четырьмя людьми за несколько выходных. Одним из них был Патрик Дассуд, которого мы можем без всяких сомнений назвать отцом сборщика мусора в .NET. Он написал простой консервативный GC в качестве доказательства концепции.

Прежде чем приступить к работе над CLR, Патрик Дассуд работал над JVM. Да-да, в какой-то момент Майкрософт серьезно подумывала о собственной реализации JVM, вместо того чтобы заняться тем, что мы теперь знаем под названием «среда выполнения .NET». Итак, находясь под воздействием идей JVM и имея в своем багаже реализацию JScript, он написал еще один, по-прежнему консервативный

сборщик мусора. Но команда, которая впоследствии сформировала костяк CLR, быстро поняла, что JVM привносит неудобные ограничения. Во-первых, от новой среды ожидалась всесторонняя поддержка технологии COM и неуправляемого кода. Одной из целей было добиться того, чтобы перекомпиляция программы на C++ с новым флагом /CLR позволила бы запустить ее в новой среде. Кроме того, стандартизация вызывала сложности, и они просто испугались возможных ограничений. В какой-то момент они даже рассматривали возможность выпустить среду выполнения C++ с расширением для сборки мусора.

Позже, посоветовавшись со своим другом (Дэвидом Муном из компании Symbolics, занимавшейся сборщиками мусора на основе поколений), Патрик принял решение написать «лучший из возможных GC» с нуля и реализовал прототип на Common LISP. Почему был выбран этот язык? Потому что он работал на нем много лет и хорошо знал его. Кроме того, у него был опыт работы с лучшими для того времени средствами отладки для LISP. Имея версию на LISP, он затем написал конвертор для транспиляции[1] кода на C++. Так и появился экспериментальный сборщик мусора для экспериментальной же версии JVM. Когда началась работа над CLR, часть этого экспериментального кода была переписана с нуля на C++ и вошла в проект. Так что слухи о том, что код GC для CLR был полностью конвертирован с LISP, – не более чем легенда.

Познакомившись с теоретическими основами и историей, мы можем сформулировать первое из многих правил, которые встретятся в этой книге.

Резюме

В этой главе мы рассмотрели очень широкий круг вопросов. Им вполне можно было бы посвятить несколько книг. Начав с таких базовых понятий, как биты и байты, мы перешли к основным типам компьютерных архитектур – гарвардской и фон Неймана. Мы узнали об основах конструирования компьютеров – регистрах, адресах и словах. Рассмотрев понятия статического и динамического выделения памяти, указателя, стека и кучи, мы добрались до самого главного – идеи автоматического управления памятью, которое еще называют сборкой мусора. Попутно мы поговорили о неудобствах ручного управления памятью и о причинах, по которым его хотелось бы автоматизировать. Фундаментальные для .NET-концепции, в частности отслеживающий сборщик мусора и его этапы – пометка, очистка и уплотнение, мы обсудили лишь в общих чертах. Более подробно они будут рассмотрены в соответствующих главах книги. Мы завершили главу кратким экскурсом в историю и обзором объемлющего контекста, что позволило нам взглянуть на проблему шире.

Полученные сейчас знания дадут нам возможность лучше понять последующие главы. Переходя от главы к главе, мы будем все ближе подбираться к вопросам практической реализации в среде .NET. Но без понимания более широкого контекста, представленного в этой главе, книга была бы неполной. А теперь приглашаю вас к главе 2, где мы перейдем от теоретических основ к конструкции низкоуровневых элементов компьютера и памяти.

---

[1] Транспиляцией называется компиляция исходного кода с одного языка на другой.

Правило 1: учиться, учиться и учиться

Применимость. Самая широкая.

Обоснование. Это самое общее правило в книге, оно применимо в гораздо более широком контексте, чем одно лишь управление памятью. И означает оно, что мы всегда должны быть нацелены на приобретение новых знаний, чтобы оставаться профессионалом. Знание не приходит само по себе. Его нужно заработать. Это утомительный процесс, требующий много времени и труда. Поэтому мы должны постоянно мотивировать себя. Разве эта очевидная истина заслуживает отдельного правила? Я думаю, что да. В повседневной жизни об этом легко забыть. Нам кажется, что решаемые каждый день задачи могут чему-то научить. Ну да, в какой-то степени. Но очевидно, чтобы выйти из зоны комфорта, нужно сделать несколько шагов. Сознательно. А это значит – достать книжку, посмотреть учебное пособие в вебе, прочитать статью. Возможностей много, перечислять их все не имеет смысла. Но это положение настолько фундаментально, что должно занять место в списке правил каждого профессионала. Если мои слова вас не убедили, поинтересуйтесь концепцией Ремесла программиста и манифестом, опубликованным по адресу http://manifesto.softwarecraftsmanship.org. Я также большой поклонник склонности к технике, которую провозглашает гонщица Джеки Стюарт:

> *Чтобы стать водителем гоночного автомобиля, не нужно быть инженером, но нужно иметь склонность к технике.*

Позже эта идея была перенесена в мир ИТ Мартином Томпсоном. Что это означает? Очевидно, чтобы быть гонщиком, необязательно быть механиком. Но без глубокого понимания того, как работает автомобиль, каким правилам механики подчиняется, как работает двигатель, какие силы на него действуют, очень трудно стать хорошим гонщиком. Чтобы гармонично слиться с машиной, нужно «почувствовать» ее. Нужно испытывать склонность к технике. И у нас, программистов, то же самое. Конечно, мы можем просто поразмышлять о таких фреймворках, как .NET или JVM, и на этом остановиться. Но тогда мы уподобимся водителям, выезжающим только по выходным, которые видят в машине лишь руль да педали.

Как применять. Для такого общего правила вряд ли удастся назвать один простой подход. Можете читать книги о том, как работает компьютер или ваш любимый фреймворк. Можете прибегнуть к услугам многочисленных обучающих сервисов. Можете посещать конференции или собрания местных групп пользователей. Можете вести блог и писать на эти темы, поскольку нет лучшего способа научиться, чем начать преподавать. Возможностей много, я даже не буду пытаться перечислить все. Просто помните о девизе «учиться, учиться и учиться» и старайтесь следовать ему в собственной жизни!

Глава 2

Низкоуровневое управление памятью

Чтобы понять, как работает управление памятью, нам нужен более широкий контекст. В предыдущей главе мы рассмотрели теоретические основы этого вопроса. Теперь можно было бы перейти прямо к деталям управления памятью, рассказать, как устроен сборщик мусора и где могут происходить утечки. Но если мы действительно хотим «прочувствовать» эту тему, то стоит потратить немного времени и вспомнить еще об одном ее аспекте. Это позволит лучше уяснить себе различные проектные решения, принятые авторами сборщика мусора в .NET (да и в других управляемых средах выполнения). Создатели этих подсистем живут не в вакууме и должны адаптироваться к условиям существования – ограничениям и механизмам, приводящим в действие оборудование и операционные системы. Вот об этом мы сейчас и поговорим.

Итак, эта глава о механизмах и ограничениях. Конечно, эти темы сами по себе достаточно обширны и заслуживают нескольких книг солидного объема. Мы рассмотрим только некоторые базовые вопросы, более-менее связанные с управлением памятью. Честно говоря, нелегко изложить столь разноплановые предметы, так чтобы не ошеломить читателя и не погрязнуть в несущественных деталях. В то же время я хотел представить материал достаточно подробно, чтобы показать влияние на управление памятью в .NET. Приглашаю к чтению!

Знание, пусть даже поверхностное, всех этих деталей позволит вам подступиться к такой сложной теме, как управление памятью. Даже если при решении повседневных задач наше общение с памятью ограничивается только вызовом оператора `new`, полезно понимать, сколько существует различных механизмов и на каких уровнях. Оборудование, операционная система, компилятор – все они влияют на устройство и работу .NET, хотя и не всегда очевидным образом. Приобретение этих знаний в полной мере отвечает духу склонности к технике, упомянутому в предыдущей главе. Надеюсь, вам тоже понравится узнать о кое-каких из упомянутых ниже фактов.

Впрочем, никто не мешает рассматривать эту главу как факультативную. В ней приведено много теоретической информации, которая, хотя помогает прочувствовать тему управления памятью, не обязательна для понимания изложенного в книге материала. Так что если вы торопитесь или хотите поскорее перейти к внутреннему устройству .NET и к примерам, то можете проглядеть эту главу по диагонали или вообще пропустить (и, надеюсь, вернуться к ней, когда со временем будет не так напряженно).

Оборудование

Как работает современный компьютер? Наверное, любой программист сможет как-то ответить на этот вопрос. Если вы изучали информатику в вузе, то что-то из лекций должно было остаться в голове. Если учились самостоятельно, то, вероятно, что-то где-то читали. И быть может, сумеете извлечь из памяти какие-то факты, например: в компьютере имеется процессор, являющийся главным обрабатывающим устройством – он выполняет программы. Компьютер имеет доступ к оперативной памяти (быстрой) и жестким дискам (медленным). Еще есть графическая карта, очень важная для геймеров (и всяких там графических дизайнеров), которая отвечает за создание изображения на экране. Но такого взгляда с высоты птичьего полета для наших целей недостаточно. Нам нужно погрузиться в предмет глубже. Поэтому позвольте мне представить архитектуру современного компьютера, изображенную на рис. 2.1.

> На рынке современных компьютеров доминируют PC и Mac'и. Представленная мной схематическая диаграмма основана на их архитектуре. При необходимости я буду отмечать нюансы, скажем, наличие процессоров ARM или более сложно устроенных серверных компьютеров.

Ниже перечислены основные компоненты архитектуры компьютера.

- *Процессор* (ЦП, центральный процессор) – главное устройство, отвечающее за выполнение команд. Мы уже встречались с ним в главе 1. В нем находятся такие компоненты, как арифметико-логическое устройство (АЛУ), блок выполнения операций с плавающей точкой (Floating-Point Unit – FPU, математический сопроцессор), регистры и *конвейеры выполнения команд*, – они отвечают за эффективное выполнение команд, составленных из нескольких элементарных операций, выполняемых (по возможности) параллельно.
- *Системная шина* (Front Side Bus – FSB) – шина данных, соединяющая ЦП с северным мостом.
- *Северный мост* – устройство, содержащее в основном контроллер доступа к памяти, отвечающий за управление взаимодействием между памятью и ЦП.
- *Оперативная память*, *ОЗУ* (англ. RAM) – главная компьютерная память. Когда подается электропитание, в ней хранятся данные и программы, поэтому ее называют также *динамическим ОЗУ* (англ. DRAM), или *энергозависимой памятью*.
- *Шина памяти* – шина данных, соединяющая ОЗУ с северным мостом.
- *Южный мост* – микросхема, отвечающая за все функции ввода-вывода: USB, звук, последовательная передача, системная BIOS, шина ISA, контроллер прерываний и каналы IDE – контроллеры устройств внешней памяти, такие как PATA или SATA.
- *Внешние запоминающие устройства* (ЗУ) – энергонезависимая память, в т. ч. популярные жесткие и SSD-диски.

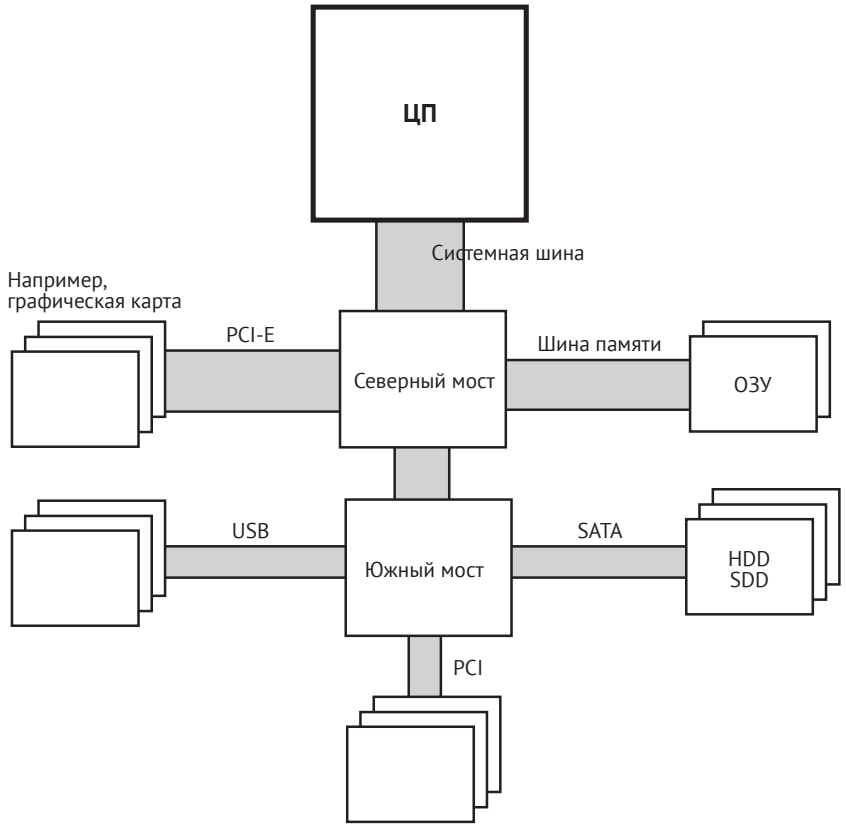


Рис. 2.1 ❖ Архитектура компьютера – ЦП, ОЗУ, северный мост, южный мост и прочее. Ширина шины обозначает количество передаваемых по ней данных (очень приблизительно)

Стоит отметить, что раньше ЦП, северный мост и южный мост были отдельными микросхемами, но теперь они тесно интегрированы. Начиная с микроархитектур Intel Nehalem и AMD Zen, северный мост находится внутри (и тогда часто называется *внеядром* (uncore), или системным агентом). Эта эволюция архитектуры показана на рис. 2.2.

Такая интеграция лучше, потому что размещение контроллера доступа к памяти (внутри северного моста) рядом с исполняющими устройствами ЦП сокращает задержку благодаря уменьшению физических расстояний. Но на рынке все еще представлены процессоры (самые популярные из них принадлежат семейству AMD FX), в которых ЦП, северный мост и южный мост разделены.

Главная проблема любого механизма управления памятью – различное быстродействие процессоров, памяти и внешних запоминающих устройств. Процессор работает гораздо быстрее памяти, поэтому каждый доступ к памяти вносит нежелательные задержки. Если процессор вынужден ждать завершения доступа к памяти (будь то чтение или запись), то говорят, что он *приостановлен*. Чем больше таких приостановок, тем хуже используется ЦП, т. к. вся его мощь впустую расходуется на ожидание.

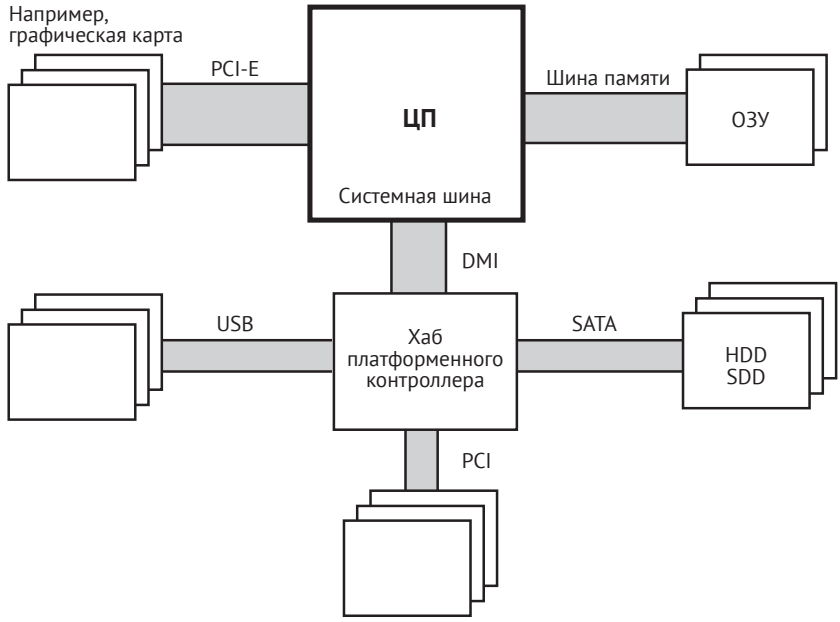


Рис. 2.2 ❖ Современная архитектура – ЦП с северным мостом, ОЗУ, южный мост (который Intel теперь называет хабом платформенного контроллера) и прочее. Ширина шины обозначает количество передаваемых по ней данных (очень приблизительно)

Типичный современный процессор работает на частоте 3 ГГц и выше. А память управляется внутренним генератором тактовых импульсов с частотой всего 200–400 МГц, т. е. медленнее на порядок. Микросхемы ОЗУ, работающие на частоте процессора, стоили бы слишком дорого. Это связано с конструкцией современных ОЗУ – зарядка и разрядка конденсаторов занимает время, и уменьшить его очень трудно.

Вы, наверное, удивились, узнав, что память работает на такой низкой частоте. Ведь в компьютерных магазинах мы покупаем модули памяти, на которых указана частота 1600 или 2400 МГц, что гораздо ближе к частоте ЦП. Откуда же берутся эти цифры? Как мы увидим, эти спецификации – лишь часть гораздо более сложной правды.

Модуль памяти состоит из *внутренних ячеек памяти* (для хранения данных) и дополнительных буферов, которые помогают преодолеть ограничения низкой внутренней тактовой частоты. Применяются и другие трюки (рис. 2.3). По большей части они опираются на кратное увеличение скорости чтения данных.

- Передача данных из внутренней ячейки памяти дважды на протяжении одного такта. Точнее, это происходит по переднему и заднему фронтам сигнала. Отсюда и название самой популярной памяти различных поколений – *Double Data Rate* (DDR, с удвоенной скоростью передачи). Эту технику называют также *двойной накачкой*.
- Применение внутренней буферизации, чтобы за один такт обращения к памяти можно было выполнить несколько операций чтения. Это позволяет

кратно увеличить объем данных, поступающих вовне, по сравнению с тем, что дает внутренняя тактовая частота. Интерфейс памяти DDR2 удваивает внешнюю частоту памяти, а DDR3 и DDR4 учетверяют ее.

Эти приемы сейчас применяются в модулях DDR, пришедших на смену гораздо более простым модулям SDRAM (синхронная DRAM).

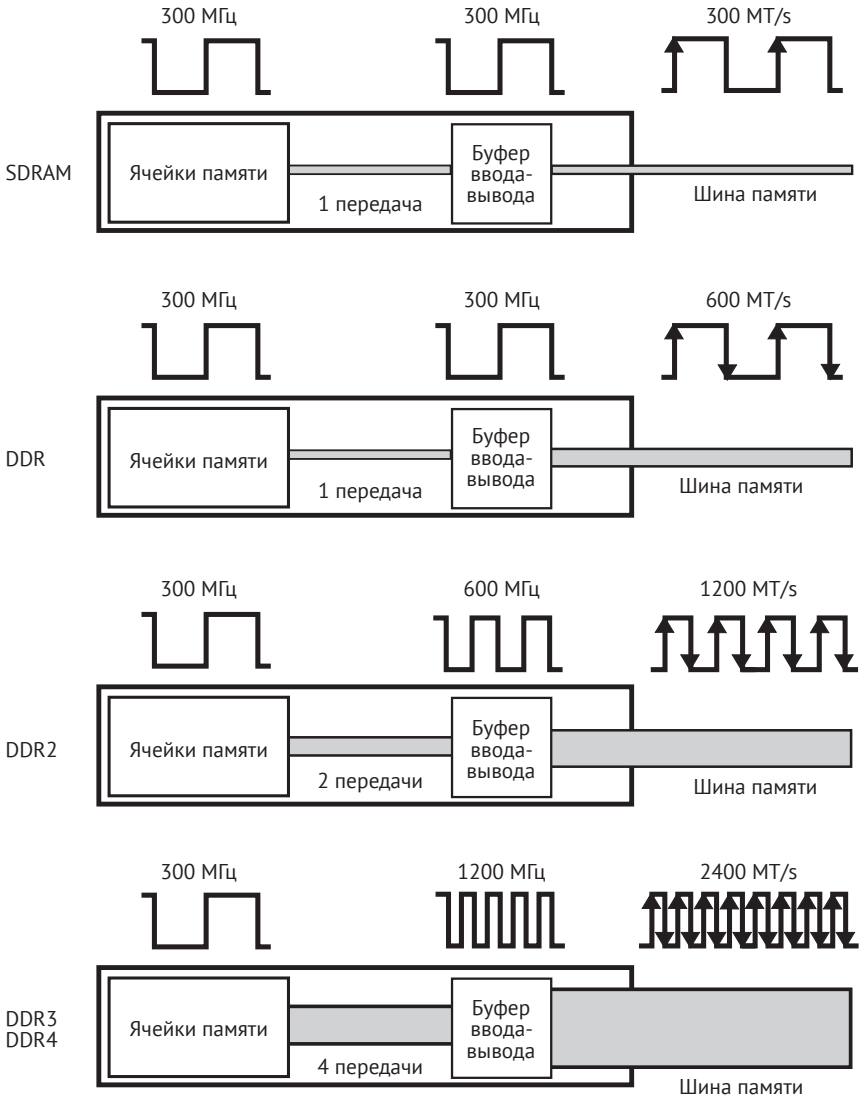


Рис. 2.3 ❖ Внутреннее устройство SDRAM, DDR, DDR2, DDR3, DDR4. Пример модулей памяти с внутренней тактовой частотой 300 МГц. MT/s означает «миллионов передач в секунду»

Рассмотрим типичную микросхему памяти DDR4, например 16 ГБ 2400 МГц (в спецификации это обозначается DDR4-2400, PC4-19200). В этом случае тактовый

генератор внутренней DRAM-памяти выдает частоту 300 МГц. Благодаря внутреннему буферу ввода-вывода частота шины памяти в 4 раза больше и составляет 1200 МГц. Кроме того, поскольку на каждом такте производится две передачи (по обоим фронтам сигнала), получается скорость передачи данных 2400 MT/s (миллионов передач в секунду). Отсюда и появляется число 2400 МГц в спецификации. Проще говоря, благодаря природе двойной накачки в DDR-памяти в качестве быстродействия обычно указывается двойная тактовая частота ввода-вывода, которая, в свою очередь, кратна внутренней тактовой частоте памяти. Включение этого значения в мегагерцах – не более чем маркетинговая уловка. Вторая характеристика – PC4-19200 – связана с теоретически максимальной производительностью такой памяти; 2400 MT/s, умноженное на 8 байт (передается одно слово длиной 64 бита), как раз и дает 19 200 МБ/с.

Рассмотрим в качестве примера архитектуру моего настольного ПК. Он оборудован процессором Intel Core i7-4770K (поколение Haswell), работающим на частоте 3,5 ГГц. Частота системной шины составляет всего 100 МГц. В модуле памяти DDR3-1600 (PC3-12800) внутренняя тактовая частота равна 200 МГц, но благодаря механизму DDR3 тактовая частота шины ввода-вывода составляет 800 МГц. Это показано на рис. 2.4. Все это можно подтвердить, например, с помощью диагностической программы CPU-Z (рис. 2.5).

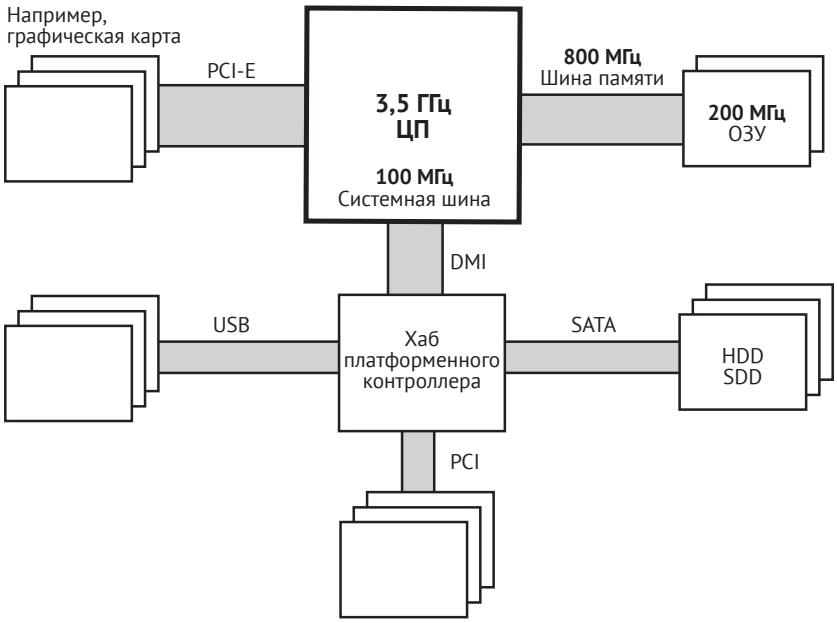


Рис. 2.4 ❖ Современная аппаратная архитектура с типовым тактовым сигналом (Intel Core i7-4770K and DDR3-1600)

Но при всех ухищрениях DDR ЦП все равно работает быстрее памяти, которой пользуется. Чтобы как-то решить эту проблему, аналогичный подход применяется и на других уровнях – перенос части данных в более быстрые (и более дорогие) запоминающие устройства. Это называется *кешированием*.

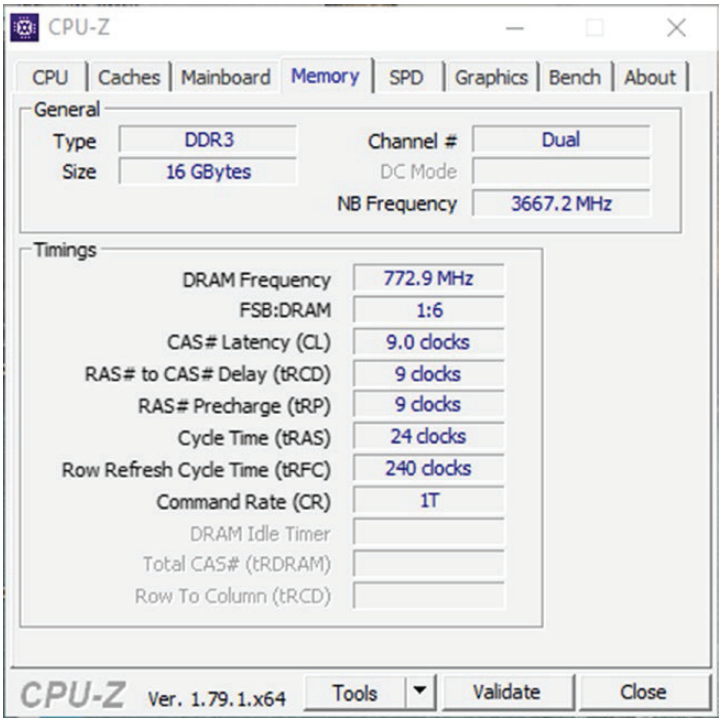


Рис. 2.5 ❖ Снимок экрана CPU-Z – на вкладке **Memory** показаны частоты северного моста (NB) и DRAM, а также отношение частот FSB:DRAM (текущая версия программы показывает неверное значение, должно быть 1:8)

В случае массовых запоминающих устройств, например жестких дисков (HDD), данные обычно кешируются в оперативной памяти (ОЗУ) или на более быстрых, но меньших по объему выделенных устройствах, например небольшом SSD-диске внутри гибридного HDD-диска, на котором хранятся самые востребованные данные. В случае ОЗУ данные кешируются в кеше ЦП, о котором мы скоро поговорим.

Разумеется, существуют и более общие оптимизации памяти, включая усовершенствованную конструкцию оборудования, улучшенные контроллеры доступа к памяти и оптимизацию прямого доступа к памяти (ПДП, англ. DMA) со стороны устройств. Но в этой книге мы не будем касаться ПДП, потому что он напрямую не связан с данными программы, и эти области памяти не находятся в ведении сборщика мусора.

Память

В сколько-нибудь полной книге, посвященной памяти, нельзя хотя бы вскользь не коснуться вопроса о физической конструкции нынешних запоминающих устройств. Возможно, вас удивят приведенные в этом разделе факты. Надеюсь, это поможет лучше понять, почему современные компьютеры имеют именно такую, а не какую-нибудь другую архитектуру.

В настоящее время в персональных компьютерах применяется память двух видов, различающихся по технологии производства, стоимости использования и производительности.

- *Статические оперативные запоминающие устройства* (англ. SRAM) – обеспечивают самый быстрый доступ, но весьма сложны и насчитывают от 4 до 6 транзисторов на каждую ячейку (для хранения одного бита). Данные в них сохраняются; до тех пор, пока подается электропитание, обновление не требуется. Из-за высокого быстродействия они используются преимущественно в процессорных кешах.
- *Динамические оперативные запоминающие устройства* (англ. DRAM) – конструкция ячейки очень проста (гораздо меньше, чем в SRAM) и состоит из одного транзистора и одного конденсатора. Из-за утечки конденсатора ячейка нуждается в периодическом обновлении (на него уходят драгоценные миллисекунды, в течение которых чтение из памяти приостанавливается). Кроме того, сигнал чтения от конденсатора нужно усиливать, что еще больше усложняет конструкцию. Операции чтения и записи занимают время и нелинейны из-за задержек конденсатора (необходимо некоторое время подождать, что получить надежный результат чтения или гарантировать успешную запись).

Посвятим еще несколько слов технологии DRAM, потому что это основа тех планок памяти, которые установлены в DIMM-разъемах наших компьютеров. Как уже сказано, ячейка DRAM, содержащая один бит данных, состоит из транзистора и конденсатора. Такие ячейки группируются в *банки DRAM*. Адрес для доступа к конкретной ячейке подается на *линии адреса*.

Было бы очень сложно и дорого присваивать каждой ячейке в банке DRAM собственный адрес. Например, в случае 32-разрядной адресации понадобился бы *декодер линий адреса* (компонент, отвечающий за выбор конкретной ячейки) шириной 32 бита. От количества линий адреса во многом зависит общая стоимость системы – чем больше линий, тем больше контактов в разъемах и взаимодействий между контроллером доступа к памяти и микросхемами (модулями) памяти. Для компьютеров с 64-разрядным словом сложность и стоимость возросли бы еще сильнее. Поэтому линии адреса используются повторно – как линии выбора строки и линии выбора столбца (рис. 2.6), а формирование полного адреса разбивается на две стадии.

Для одного банка линия адреса (строки) выбирает строку, а линия данных – столбец. Бит, хранящийся в ячейке, читается с помощью следующего процесса:

1. На линии адреса помещается номер строки.
2. На специальную линию помещается сигнал стробирования адреса строки (Row Address Strobe – RAS), определяющий интерпретацию сигнала на линиях адреса.
3. На линии адреса помещается номер столбца.
4. На специальную линию помещается сигнал стробирования адреса столбца (Column Address Strobe – CAS).
5. Прочитать данные – один бит (была адресована конкретная ячейка DRAM).

Модули DRAM, стоящие в наших компьютерах, состоят из большого числа банков DRAM, организованных так, чтобы можно было обращаться к нескольким битам (составляющим одно слово) в одном такте.

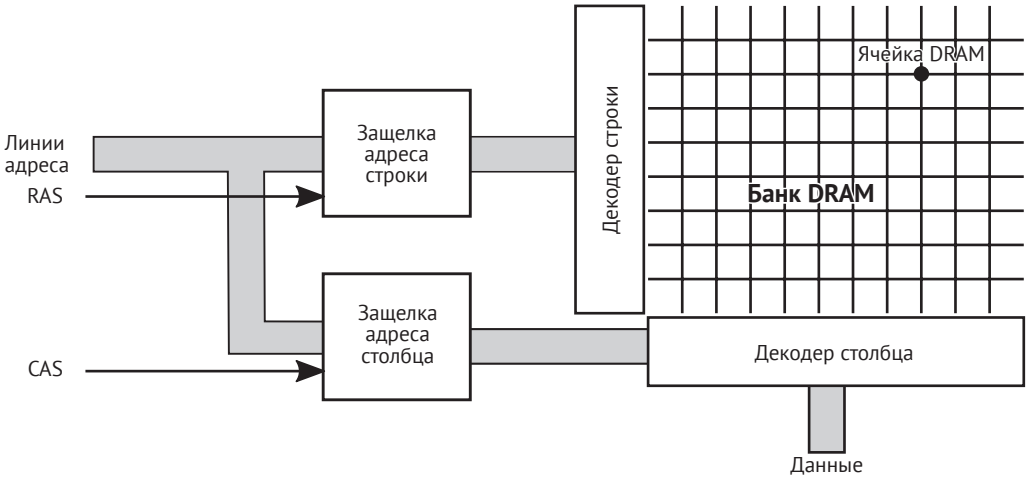


Рис. 2.6 ❖ Пример микросхемы DRAM.
Показаны банк DRAM и самые важные каналы: линии адреса, RAS и CAS

От промежутков времени между отдельными шагами процесса получения одного бита сильно зависит производительность памяти. Эти величины, возможно, вам известны, поскольку они являются важной частью спецификации модуля памяти и, кстати, влияют на его цену. Так что вы, наверное, знаете о временных характеристиках DIMM-модулей, например DDR3 9-9-9-24. Эти характеристики выражены в количестве тактов, необходимых для выполнения соответствующего действия, и означают следующее:

- tCL (задержка CAS) – время между стробированием адреса столбца (CAS) и началом получения ответа (получения данных);
- tRCD (задержка между RAS и CAS) – минимальное время между стробированием адреса строки (RAS) и стробированием адреса столбца (CAS);
- tRP (предзаряд строки) – время, необходимое для *предзаряда* строки перед доступом к ней. Строку нельзя использовать без предварительной подготовки, которая называется предзарядом (precharge);
- tRAS (задержка активности строки) – минимальное время, в течение которого строка должна оставаться активной, чтобы можно было обратиться к находящейся в ней информации. Обычно не меньше суммы всех трех предыдущих величин.

Обратите внимание на важность этих величин. Если интересующие вас строка и столбец уже установлены, то чтение происходит почти мгновенно. Для изменения столбца потребуется tCL тактов. Если нужно изменить строку, то ситуация гораздо хуже. Сначала ее нужно предзарядить (tRP тактов), после чего последуют задержки RAS и CAS (tCL и tRCD).

Все эти времена важны для пользователей компьютера, стремящихся к максимальной производительности. Особенно внимательно к ним относятся геймеры. Нам достаточно знать, что если производительность для вас – высший приоритет, то, покупая модули памяти, нужно брать те, для которых временные характеристики настолько малы, насколько позволяет ваш бюджет.

Однако нас интересует влияние архитектуры DRAM-памяти и ее характеристик на управление памятью. Как мы видели, наиболее значимый фактор – стоимость смены строки, т. е. длительность сигнала RAS и предзаряда. Это одна из многих причин, по которым последовательный доступ к памяти гораздо быстрее произвольного. Чтение данных из одной строки (меняя столбец) гораздо быстрее, чем частая смена строк. Если же характер доступа к памяти полностью случайный, то, скорее всего, мы будем платить за смену строк при каждой операции доступа.

Все вышеизложенное преследует одну цель – объяснить, почему произвольный доступ к памяти так нежелателен. Но, как мы увидим далее, это не единственная причина, по которой полностью случайный доступ – худший из возможных сценариев.

Центральный процессор

Теперь обратимся к теме центрального процессора. Процессор совместим с *архитектурой системы команд* (Instruction Set Architecture – ISA) – она определяет, среди прочего, набор допустимых операций (команд), регистры и их семантику, способы адресации памяти и т. д. В этом смысле ISA – это контракт (интерфейс), заключенный между производителем процессора и пользователями: программы пишутся в соответствии с контрактом. Этот слой мы видим, например, при программировании на языке ассемблера для данной архитектуры. Архитектуры, совместимые с IA-32 (32-разрядный i386, 32-разрядные процессоры Pentium) и AMD64 (большинство современных процессоров, включая Intel Core, AMD FX и Zen и т. д.), относятся к числу наиболее распространенных в экосистеме .NET. Ниже ISA находится реализующая ее *микроархитектура* процессора. Это позволяет совершенствовать микроархитектуру, не затрагивая систему и программное обеспечение, т. е. сохраняя обратную совместимость.

> **ПРИМЕЧАНИЕ** В стандартах 64-разрядной архитектуры наблюдается разнобой, встречаются названия x86-64, EMT64T, Intel 64 и AMD64. Несмотря на присутствие названия компании-производителя и некоторые мелкие отличия, в этой книге мы можем считать, что все это взаимозаменяемые синонимы.

Как было сказано в предыдущей главе, ключевую роль в работе ЦП играют регистры, поскольку все современные компьютеры являются регистровыми машинами. В контексте работы с данными доступ к регистрам является мгновенной операцией в том смысле, что происходит в одном такте процессора и не привносит дополнительных задержек. С точки зрения размещения данных, не существует места, более близкого к процессору, чем регистры. Конечно, в регистрах хранятся данные, необходимые для текущей команды, поэтому их нельзя считать памятью общего назначения. Вообще говоря, в процессоре больше регистров, чем явствует из ISA. Это открывает возможность для различных видов оптимизации (например, *переименования регистров*). Однако это детали реализации микроархитектуры, не влияющие на управление памятью.

Процессорный кеш

Как мы уже отмечали, для сглаживания различий в производительности ЦП и ОЗУ используется промежуточный компонент, в котором хранятся самые часто ис-

пользуемые и востребованные данные, – процессорный кеш. В очень общем виде он показан на рис. 2.7.

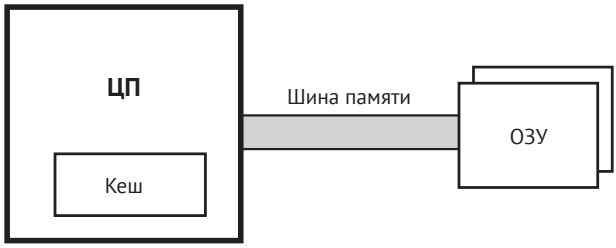


Рис. 2.7 ❖ ЦП и связь между кешем и оперативной памятью

С точки зрения ISA, этот кеш прозрачен. Ни программист, ни операционная система не обязаны знать о его существовании. И управлять им они не должны. В идеальном мире ответственность за правильное использование и управление кешем нес бы только ЦП.

Поскольку в качестве кеша мы хотели бы видеть максимально быструю память, для этой цели используются вышеупомянутые микросхемы SRAM. Из-за стоимости и размерных характеристик (они занимают драгоценное место внутри процессора), присущих этой технологии, их объем, очевидно, не может быть таким же большим, как для основной памяти. Но в зависимости от стоимости они могут работать с такой же скоростью, как ЦП, или всего на один-два порядка медленнее.

Попадание и непопадание в кеш

Идея кеша очень проста. Когда для выполнения команды процессор должен обратиться к памяти (для чтения или записи), он сначала смотрит, присутствуют ли необходимые ему данные в кеше. Если да, то все отлично. Нам повезло – доступ к памяти оказался очень быстрым, такая ситуация называется *попаданием в кеш*. Если данных в кеше нет (*непопадание в кеш*, или *промах кеша*), то они помещаются туда после чтения из ОЗУ – операции, гораздо более медленной. *Коэффициенты попадания* и *непопадания в кеш* – очень важные индикаторы, показывающие, насколько эффективно используется кеш.

Локальность данных

Но почему вообще такой кеш полезен? Идея кеша основана на очень важной концепции – *локальности данных*. Различают два вида локальности.

- *Временная локальность* – если мы обращались к некоторому участку памяти, то с большой вероятностью обратимся к нему и в ближайшем будущем. Поэтому использование кеша вполне оправдано – мы читаем данные сейчас и, вероятно, воспользуемся ими еще не раз впоследствии. Почему временная локальность имеет место? Интуитивно это понятно. Мы редко используем данные только один раз. Обычно структуры данных записываются в переменные, а эти переменные используются неоднократно. Это могут быть разного рода счетчики, прочитанные из файлов временные данные, и т. д.

- *Пространственная локальность* – если мы обратились к некоторой области памяти, то, скорее всего, обратимся к данным в ближайшей окрестности. Этот вид локальности может сослужить нам службу, если мы будем помещать в кеш чуть больше данных, чем необходимо в этот момент. Например, если нам понадобилось несколько байтов из памяти, то прочтем и поместим в кеш их и еще десяток. И это тоже интуитивно понятно. Редко бывает так, что мы используем небольшой изолированный участок памяти. Скоро мы увидим, что стек и куча организованы в виде сегментов, так что работающие потоки в общем случае обращаются к схожим областям памяти. Локальные переменные и структуры данных также, вообще говоря, размещаются рядом.

Заметим, что если указанные условия выполняются, то кеш дает полезный эффект. Но это обоюдоострое оружие. Если программа нарушает принцип локальности данных, то кеш оказывается лишним грузом. Мы еще поговорим об этом ниже в этой главе.

Реализация кеша

Добавим еще, что коль скоро поддерживается совместимость с моделью памяти, описанной в ISA, то детали реализации кеша несущественны. Он просто должен присутствовать для ускорения доступа к памяти – и на этом всё. Однако тут мы наблюдаем идеальное проявление *закона протекающих абстракций*, сформулированного Джоэлем Спольски (Joel Spolsky):

Все нетривиальные абстракции в какой-то степени протекают.

Это означает, что абстракция, которая теоретически должна скрывать детали реализации, при некоторых обстоятельствах раскрывает их. И обычно это происходит непредвиденным или нежелательным образом. Как это выглядит в случае кеша, скоро станет ясно, но пока позвольте мне углубиться в одну деталь реализации.

Самый важный и оказывающий наибольшее влияние факт заключается в том, что между ОЗУ и кешем данные перемещаются блоками, которые называются *строками кеша*. Размер строки кеша фиксирован и в подавляющем большинстве современных компьютеров составляет 64 байта. Запомните, это очень важно – ни прочитать из памяти, ни записать в нее невозможно меньше данных, чем длина строки кеша, 64 байта. Даже если вы захотите прочитать всего один бит, будет прочитан целый блок, насчитывающий 64 байта. При такой конструкции обеспечивается более эффективный последовательный доступ к DRAM (вспомните описанные выше предзаряд строк и задержки RAS).

Как уже было сказано, передача данных между процессором и DRAM производится блоками по 64 бита (8 байт), поэтому для заполнения строки кеша необходимо восемь передач. Для этого нужна уйма тактов процессора, и, чтобы как-то сгладить проблему, применяются различные приемы. Один из них – *сначала критическое слово и ранний рестарт*. Тогда чтение строки кеша осуществляется не слово за словом по порядку, а сначала читается самое нужное слово. Представьте худший случай, когда это 8-байтовое слово находится в конце строки кеша, так что для получения доступа к нему придется ждать завершения семи предшествующих передач. Так вот, чтобы этого не было, самое важное слово читается первым. Команды, ожидающие этого слова, могут продолжить выполнение, а остаток строки кеша будет заполнен асинхронно.

ПРИМЕЧАНИЕ Как выглядит типичный паттерн доступа к памяти? Если кто-то хочет прочитать данные из памяти, то в кеше создается соответствующая строка и в нее читается 64 байта. Если кто-то хочет записать данные в память, то первый шаг точно такой же – заполняется строка кеша, если ее не было раньше. И модифицируются именно эти кешированные данные. Далее возможны две стратегии.

При *сквозной записи* только что модифицированные в кеше данные сразу же сохраняются в основной памяти. Этот подход просто реализовать, но он создает большую нагрузку на шину памяти.

При *отложенной записи* модифицированная строка помечается флагом. Когда в кеше не останется места для новых данных, этот помеченный блок будет записан в основную память и удален из кеша. Процессор может выгружать такие блоки периодически, когда сочтет необходимым (например, во время простоя).

Есть еще один вид оптимизации, который называется *комбинированной записью*. Он гарантирует, что данная строка кеша из данной области памяти записывается целиком (а не отдельными словами). Здесь снова используется тот факт, что последовательный доступ к памяти быстрее.

Из-за строк кеша все данные, хранящиеся в памяти, выравниваются по границе, кратной 64 байтам. Поэтому в худшем случае для чтения двух соседних байтов придется заполнить две строки кеша, т. е. прочитать целых 128 байт. Они попадут в кеш, но если больше никаких данных из этой области памяти не нужно, то время будет потрачено зря. Это показано на рис. 2.8, где мы хотим прочитать всего два байта по адресу A. К сожалению, адрес A расположен всего на один байт раньше конца участка, заканчивающегося на границе строки кеша, поэтому придется прочитать две строки кеша.

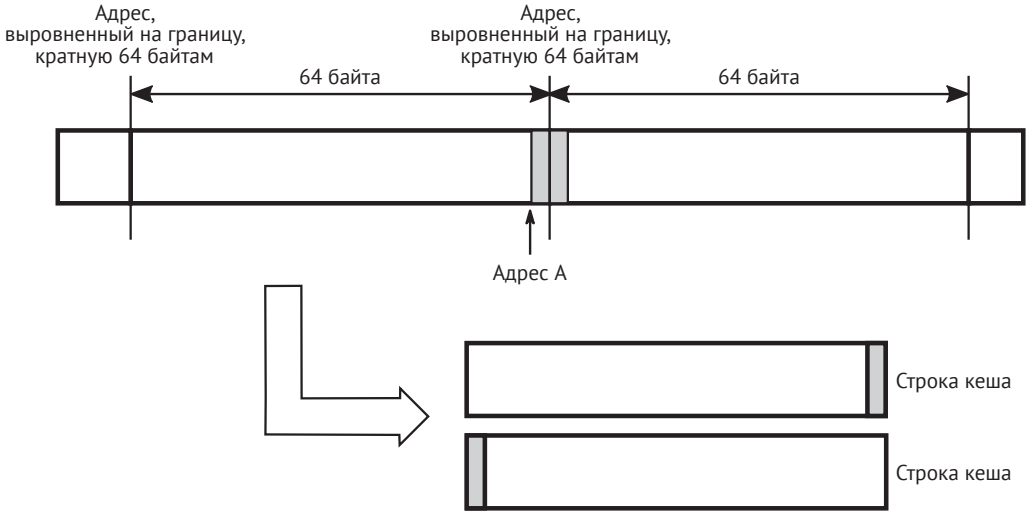


Рис. 2.8 ❖ Для доступа к двум соседним байтам требуется заполнить две строки кеша, потому они очень неудачно расположены

Прекрасно. Это, конечно, только вершина айсберга, но все равно у вас может возникнуть вопрос, к чему нам эти детали аппаратной реализации. Так ли это важно в комфортном мире управляемого кода? Пойдем дальше и выясним.

Плата за непоследовательный доступ к памяти иллюстрируется на примере кода из листинга 2.1 и его результатов в табл. 2.1. Эта программа обращается к одному и тому же двумерному массиву двумя способами: по строкам и по столбцам. Результаты приведены для трех разных сред: ПК (Intel Core i7-4770K 3.5 ГГц), ноутбук (Intel Core i7-4712MQ 2.3 ГГц) и плата Raspberry Pi 2 (ARM Cortex-A7 0.9 ГГц).

Листинг 2.1 ❖ Доступ к массиву по строкам и столбцам (массив целых чисел 5000×5000)

```
int[,] tab = new int[n, m];
for (int i = 0; i < n; ++i)
{
   for (int j = 0; j < m; ++j)
   {
      tab[i, j] = 1;
   }
}
```

```
int[,] tab = new int[n, m];
for (int i = 0; i < n; ++i)
{
   for (int j = 0; j < m; ++j)
   {
      tab[j, i] = 1;
   }
}
```

Таблица 2.1. Результаты доступа к массивам по строкам и по столбцам (n, m = 5000)

Доступ	ПК	Ноутбук	Raspberry Pi 2
По строкам	52 мс	127 мс	918 мс
По столбцам	401 мс	413 мс	2001 мс

Этот пример наглядно демонстрирует, насколько сильно непоследовательный доступ к данным может снизить производительность. Второй вариант программы читает данные по столбцам. В результате мы каждый раз должны изменять активную строку ячеек DRAM. Хуже того, мы очень плохо используем кеш, поскольку для чтения всего одного байта загружаем целую строку кеша, а потом сразу переходим к чтению из далеко расположенного адреса, так что приходится заполнять другую строку кеша. Разница в производительности может быть шестикратной! Процессор очень часто приостанавливается и ждет завершения доступа к памяти.

На рис. 2.9 показано различие между доступом к элементам по строкам и по столбцам для небольшого массива, содержащего числа от 1 до 40 (предполагается, что в одной строке кеша помещается четыре таких числа). Для иллюстрации будем считать, что доступ к массиву производится процессором, буфер которого вмещает только четыре строки кеша[1]. Читая память по строкам (рис. 2.9 слева), мы по существу читаем последовательные целые числа из соседних областей, округленных до размера строки кеша.

[1] В настоящем ЦП «буфер» строк кеша – это весь процессорный кеш, который в типичном случае вмещает сотни или тысячи строк кеша длиной 64 байта.

- Чтобы прочитать первые четыре элемента (1, 2, 3, 4), мы заполняем первую строку кеша и используем все элементы из нее.
- Чтобы прочитать следующие четыре элемента (5, 6, 7, 8), мы заполняем вторую строку кеша и снова используем все элементы из нее.
- Чтобы прочитать следующие четыре элемента (9, 10, 11, 12), мы заполняем третью строку кеша. Этот процесс продолжается, пока не будет прочитан весь массив (и ни одну строку кеша не приходится читать дважды).

В правой части рис. 2.9 показан второй способ доступа, когда мы читаем по одному целому числу из каждой строки кеша и сразу переходим к следующему.

- Чтобы прочитать первые четыре элемента, мы заполняем четыре строки кеша, но используем лишь один элемент из каждой строки (1 из первой строки, 9 из второй и т. д.).
- Чтобы прочитать следующий элемент (33), одну из заполненных строк кеша необходимо вытолкнуть, поскольку буфер полон. Скорее всего, это будет строка, которая использовалась давно (т. е. содержащая элементы 1, 2, 3, 4), и ее заместит строка, содержащая элементы 33, 34, 35, 36.
- Чтобы прочитать следующий элемент (2), снова выталкивается наименее используемая строка, и ЦП вынужден перезагрузить первую строку (содержащую элементы 1, 2, 3, 4), которая только недавно была выгружена.
- Все это повторяется многократно, при этом каждая строка кеша будет прочитана четыре раза.

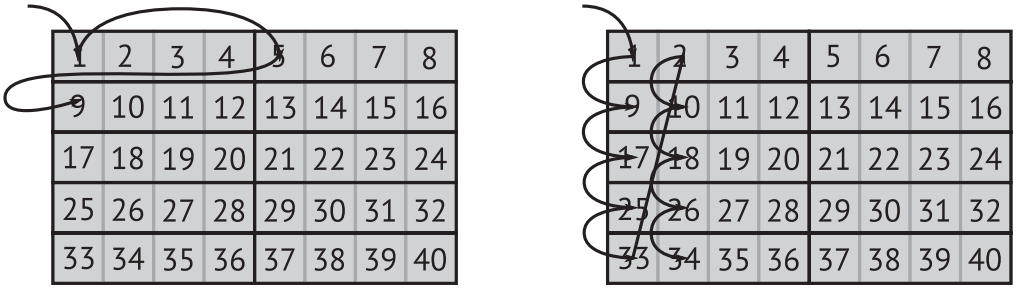


Рис. 2.9 ❖ Доступ к массиву по строкам и столбцам.
Стрелками показаны элементы, доступ к которым вызывает выгрузку строки кеша (для первых 10 элементов)

Понятно, что в настоящем ЦП буфер вмещает более четырех строк кеша и что строка кеша рассчитана более, чем на 4 целых числа, так что рис. 2.9 – всего лишь упрощение для иллюстрации. Но точно такая же проблема возникает и в реальном мире, о чем свидетельствует табл. 2.1.

Как видим, среда выполнения .NET со всеми передовыми методами управления памятью не способна скрыть эти детали реализации ЦП, которые вылезают самым неприятным образом. Неудачный порядок доступа к памяти сильно снижает производительность нашей программы. И мало утешения в том, что аналогичный тест на Java или C/C ++ даст столь же печальные результаты.

Выравнивание данных

У доступа к памяти есть еще один важный аспект. Большинство процессорных архитектур спроектировано с расчетом на то, что данные в памяти правильно выровнены. Это означает, что начальный адрес данных кратен некоторому количеству байтов. Для каждого типа определено свое выравнивание, а выравнивание структуры данных зависит от выравнивания ее полей. Обращение к невыровненным данным может оказаться в несколько раз медленнее, чем к выровненным. Обязанность следить за тем, чтобы этого не было, возлагается на компилятор и проектировщика структур данных. В случае CLR размещением структур данных в памяти занимается в основном сама среда выполнения. Поэтому в коде сборщика мусора можно найти много кода, относящегося к выравниванию. В главе 13 мы увидим, как выглядит размещение объекта в памяти и как им можно управлять с учетом выравнивания данных.

Некешируемый доступ

До сих пор мы говорили, что в большинстве архитектур ЦП доступ к памяти возможен только через кеш. Все данные, прочитанные или записанные процессором в DRAM, хранятся в кеше. Предположим, что требуется инициализировать очень большой массив, но использоваться он будет еще не скоро. Из того, что нам известно, можно сделать вывод, что такая инициализация вызовет интенсивный трафик в памяти. Массив будет записываться блоками, размер которых равен длине строки кеша. Более того, каждая операция записи выполняется в три этапа: прочитать строку в кеш, изменить содержимое кеша и записать строку кеша назад в оперативную память. Мы будем заполнять строки кеша только для того, чтобы отправить их в память. Мало того что это само по себе неоптимально, так еще и отбирает кеш у других программ.

Такого трафика можно избежать, воспользовавшись набором команд *некешируемого доступа*: `MOVNTI`, `MOVNTQ`, `MOVNTDQ` и т. д. Они позволяют обойти кеширование при записи в память. Для их использования из C/C++ имеются функции семейства `_mm_stream_*`, так что писать на ассемблере не придется. Например, функция `_mm_stream_si128` выполняет команду `MOVNTDQ`, которая записывает в память одно четверное слово (4 слова по 4 байта каждое). В листинге 2.2 приведен пример быстрой инициализации массива с использованием этой техники.

Листинг 2.2 ❖ Пример низкоуровневого API некешируемой записи в C++

```
#include <emmintrin.h>

void setbytes(char *p, int c)
{
   __m128i i = _mm_set_epi8(c, c, c, c, c, c, c, c, c, c, c, c, c, c, c, c);
   // инициализировать 16 8-битовых целых чисел со знаком
   _mm_stream_si128((__m128i *)&p[0], i);
   _mm_stream_si128((__m128i *)&p[16], i);
   _mm_stream_si128((__m128i *)&p[32], i);
   _mm_stream_si128((__m128i *)&p[48], i);
}
```

Зачем мы вообще об этом говорим? В настоящее время .NET не поддерживает некешируемый доступ, хотя есть планы использовать его в некоторых частях самой среды выполнения. И есть идеи, как предоставить разработчикам возможность давать среде выполнения указания об использовании некешируемого доступа (в листинге 2.3 показано, как это могло бы выглядеть).

Листинг 2.3 ❖ Возможная реализация будущей функциональности – указание среде выполнения использовать некешируемый доступ при сохранении данных

```
public int[] Sum ( int[] op1, int[] op2 )
{
   var result = new int[op1.Length];
   Contract.Assume( Performance.NonTemporal(result) );
   result[i] = op1[i] + op2[i]
}
```

А еще до того, как это будет реализовано на уровне JIT, кто-то, возможно, решит вызвать `_mm_stream_si128` из C# с помощью механизма P/Invoke в особо критичном с точки зрения быстродействия коде. Но, конечно, предварительно нужно хорошенько подумать.

> **ПРИМЕЧАНИЕ** Отметим, что имеется также команда некешируемой загрузки `MOVNTDQA`, доступная с помощью функции `_mm_stream_load_si128`.

Упреждающая выборка

Локальность данных – замечательная возможность, которой механизм кеширования пользуется автоматически, если программист явно не вмешается. Но есть еще один механизм, направленный на улучшение использования кеша. Речь идет о загрузке в кеш данных, которые с большой вероятностью понадобятся в будущем, – упреждающей выборке. Она может работать в двух режимах.

- Аппаратный – когда ЦП замечает несколько непопаданий в кеш, следующих друг за другом с определенной закономерностью. Большинство процессоров могут отслеживать от 8 до 16 паттернов доступа к памяти (в соответствии с типичным многопоточным или многопроцессным режимом работы). Примечание: хотя мы пока не говорили о страницах памяти, имейте в виду, что упреждающая выборка возможна только *в пределах страницы*.
- Программный – путем явного вызова из программы командой `PREFETCHT0`, доступной с помощью функции C/C++ `_mm_prefetch()`.

Упреждающая выборка, как и все механизмы кеширования, – палка о двух концах. Если мы хорошо понимаем паттерны доступа к памяти в своей программе, то упреждающая выборка может заметно ускорить работу. С другой стороны, очень трудно с уверенностью утверждать, что паттерны доступа понятны, учитывая, что код работает не изолированно, а в окружении других потоков в той же программе, потоков других программ и потоков операционной системы. В коде .NET есть обращение к команде `PREFETCHT0`, но поскольку соответствующая константа препроцессора `PREFETCH` не определена, упреждающая выборка не используется (см. листинг 2.4).

Листинг 2.4 ❖ Относящийся к упреждающей выборке код .NET – по умолчанию он не активирован

```
//#define PREFETCH
#ifdef PREFETCH
__declspec(naked) void __fastcall Prefetch(void* addr)
{
   __asm {
      PREFETCHT0 [ECX]
      ret
   };
}
#else //PREFETCH
inline void Prefetch (void* addr)
{
   UNREFERENCED_PARAMETER(addr);
}
#endif //PREFETCH
```

Обращения к упреждающей выборке встречаются во многих местах сборщика мусора в CLR. Но затем использование `PREFETCHT0` было отключено, вероятно, по вышеизложенным причинам. Среда выполнения – очень общий код, и трудно представить, что в ней найдется хотя бы один кусок, в котором упреждающая выборка будет оправдана при всех обстоятельствах. Поэтому решение не использовать ее безопаснее.

Упреждающая выборка и кешированный доступ, очевидно, играют против нас, если мы не постараемся правильно разместить данные в памяти. Например, если бы мы спроектировали алгоритм сборки мусора так, что какие-то очень маленькие диагностические данные длиной 1 байт были случайно разбросаны по всей памяти, то операция по сбору всей этой информации очень дорого обошлась бы в смысле кеширования. Нам пришлось бы заполнять кеш строками только для того, чтобы прочитать один байт. А, как было сказано, упреждающая выборка может сделать еще хуже – «если вы собираетесь прочитать эти 64 байта, то давайте-ка прочтем в два раза больше – вдруг да понадобится».

Алгоритмы, которые интенсивно работают с памятью (а сборка мусора по сути своей относится к таковым), должны принимать во внимание такие особенности ЦП. Память – это не равнина, на которой можно безнаказанно подбирать одиночные байты или биты то тут, то там!

Иерархический кеш

Но вернемся к нашей архитектуре. В силу требований к производительности, с одной стороны, и стремления оптимизировать затраты – с другой, конструкторы ЦП со временем пришли к более сложному *иерархическому кешу*. Идея проста. Вместо одного кеша создадим несколько, с разными размерами и быстродействием. Тогда у нас будет очень маленький и очень быстрый кеш первого уровня (он называется L1), затем *кеш второго уровня* (L2) побольше и помедленнее и, наконец, *кеш третьего уровня* (L3). В современных архитектурах иерархия ограничена тремя уровнями. Она показана на рис. 2.10. Иногда встречаются процессоры, оснащенные кешем L4, но это немного другой вид памяти, предназначенный в основном для интегрированных графических карт, находящихся внутри ЦП.

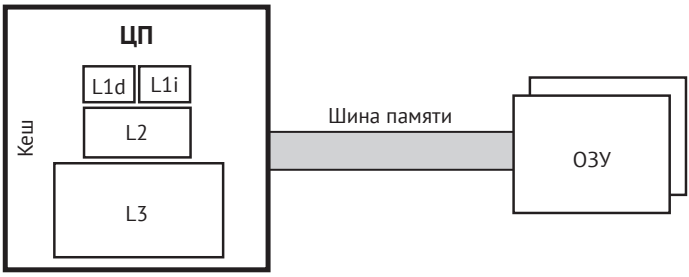


Рис. 2.10 ❖ ЦП с иерархическим кешем – кеш первого уровня, разделенный на кеш команд (L1i) и кеш данных (L1d), и кеши второго (L2) и третьего (L3) уровней. ЦП соединен с DRAM шиной памяти

Кеш первого уровня разделен на два блока: для данных (обозначен L1d) и для команд (L1i). Команды, которые процессор читает из памяти и выполняет, – тоже данные, только особым образом интерпретируемые. На следующих уровнях данные и команды не различаются, что должно напомнить нам об архитектуре фон Неймана, упомянутой в главе 1. Но практика показывает, что на самом нижнем уровне выгоднее обрабатывать данные и команды по-разному, т. е. реализовать подход, свойственный гарвардской архитектуре. Поэтому говорят, что современные компьютеры имеют *модифицированную гарвардскую архитектуру*. Это решение хорошо работает, потому что области памяти для хранения кода и данных программы в значительной степени независимы, но только на самом нижнем уровне.

Коль скоро существует три основных уровня кеша, возникает естественный вопрос: каковы различия в быстродействии и размере между ними и основной памятью? На нижних уровнях кеша – L1 и даже L2 – обращение к памяти занимает меньше тактов процессора, чем выполнение конвейера (если только не нужно ждать вычисления точного адреса, что также является дорогостоящей операцией). И как же выглядят временные характеристики?

Я пишу эту главу на ноутбуке с процессором Intel Core i7-4712MQ (поколение Haswell), работающем на частоте 2,30 ГГц. В предположении, что время одного такта на этом ноутбуке составляет примерно 0,4 нс (~1/2,30 ГГц), и пользуясь спецификацией Haswell i7, мы можем вычислить задержку при доступе к различным уровням памяти (см. табл. 2.2).

Таблица 2.2. Задержка при доступе к различным частям памяти

Операция	**Задержка**
Кеш L1	< 2,0 нс
Кеш L2	4,8 нс
Кеш L3	14,4 нс
Основная память	71,4 нс
Жесткий диск	150 000 нс

Как видим, имеет прямой смысл побороться за оптимальное использование кеша. Если нужные процессору данные имеются в кеше L3, то задержка может оказаться в 5 раз меньше, чем при доступе к ОЗУ. А при нахождении в кеше L1 –

более чем в 30 раз. Вот почему характер использования кеша так важен для общей производительности. Сколько данных помещается в кеш? Все зависит от модели ЦП, но спецификация моего i7-4770K довольно объективно отражает рыночные стандарты. Размер кеша L1 составляет 64 КиБ (32 КиБ для команд и столько же для данных), а размер кеша L2 – 256 КиБ. Кеш L3 всегда гораздо больше – 8 МиБ.

Оказывают ли эти характеристики влияние на жизнь разработчика, особенно в управляемом мире .NET? Рассмотрим простой пример, иллюстрирующий задержку доступа к данным в зависимости от объема обрабатываемой памяти. Программа в листинге 2.5 выполняет последовательное (и потому наиболее оптимальное) чтение. Поскольку размер структуры данных равен 64 байтам, чтение производится блоками по 64 байта, и каждый раз нужно загружать новую строку кеша. На рис. 2.11 показано среднее время доступа в расчете на один элемент массива `tab` в зависимости от того, сколько всего памяти занимает массив.

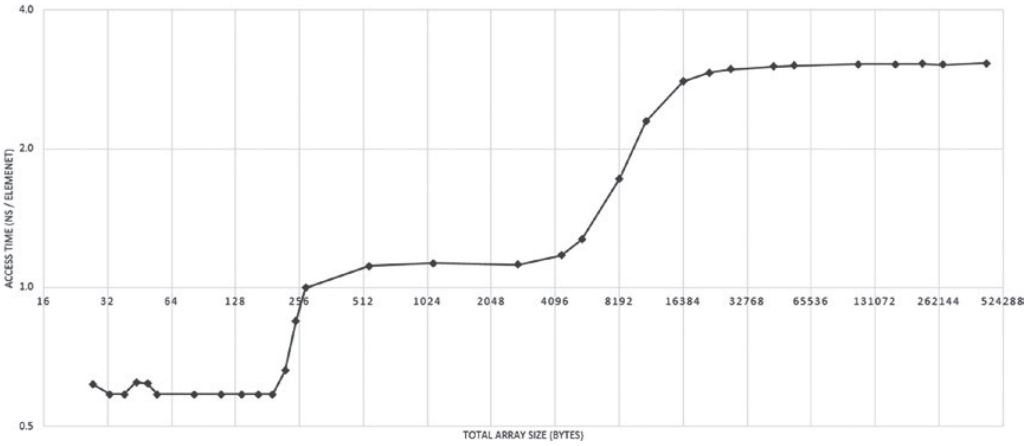


Рис. 2.11 ❖ Зависимость времени доступа от размера данных – последовательное чтение, архитектура Intel. Масштаб по обеим осям – логарифмический

Время доступа резко возрастает, когда размер данных начинает превышать размер кеша каждого уровня. Поскольку тесты производительности выполнялись на процессоре i7-4770K, отчетливые точки ухудшения находятся в районе 256 КиБ и 8192 КиБ, что соответствует размерам кешей L2 и L3. Как видим, работа с данными небольшого размера может оказаться в несколько раз быстрее, чем с данными, не помещающимися в кеш L3.

Листинг 2.5 ❖ Последовательное чтение в соседние строки кеша

```
public struct OneLineStruct
{
   public long data1;
   public long data2;
   public long data3;
   public long data4;
   public long data5;
   public long data6;
```

```
   public long data7;
   public long data8;
}

public static long OneLineStructSequentialReadPattern(OneLineStruct[] tab)
{
   long sum = 0;
   int n = tab.Length;
   for (int i = 0; i < n; ++i)
   {
      unchecked { sum += tab[i].data1; }
   }
   return sum;
}
```

> **ПРИМЕЧАНИЕ** С кешами связан еще один интересный, но не столь важный вопрос – *стратегия вытеснения*. Речь идет о том, как освободить место для новых данных, отсутствующих в кеше данного уровня. Существует два подхода, которые иногда зависят от уровня:
>
> – *исключающий кеш*: данные присутствуют только на одном уровне кеша. Этот метод чаще всего применяется в процессорах AMD;
> – *включающий кеш*: каждая строка кеша более высокого уровня (например, L1d) присутствует также в кеше более низкого уровня (например, L2).
>
> Хотя это и любопытно, к нашим рассуждениям об управлении памятью отношения не имеет. Надо полагать, производители процессоров делают все возможное, чтобы реализация этих механизмов была максимально эффективной.

Иерархический кеш в многоядерных процессорах

Однако наш обзор конструкций компьютеров еще не закончен. В большинстве современных процессоров имеется несколько ядер. Если не вдаваться в подробности, то *ядро* – это отдельный упрощенный процессор, который может выполнять код независимо от других ядер. В прошлом каждое ядро выполняло ровно один поток. Поэтому четырехъядерный процессор мог исполнять четыре потока одновременно. Теперь практически все процессоры поддерживают *механизм одновременной многопоточности* (simultaneous multithreading mechanism – SMT), который позволяет одновременно выполнять два потока в одном ядре. В процессорах Intel это называется гипертредингом, а компания AMD добавила полную поддержку SMT в микроархитектуру Zen. Распределение кешей между отдельными ядрами четырехъядерного процессора показано на рис. 2.12.

Как видно, в каждом ядре имеется свой кеш первого и второго уровней, а кеш третьего уровня – общий для всех ядер. Как именно ядра соединены с кешем L3 – деталь реализации. Например, в самых современных процессорах Intel имеется двунаправленная исключительно быстрая 32-байтовая шина, к которой подключены также интегрированный GPU и системный агент. Отметим, что для SMT-процессоров два потока, исполняемых одним ядром, разделяют кеши L1 и L2, поэтому их размеры фактически надо поделить пополам, если только не приняты специальные меры, чтобы оба потока по возможности работали с одними и теми же данными. Очевидно, для этого требуется, чтобы операционная система осознанно назначала потокам ядра в зависимости от паттернов доступа к данным.

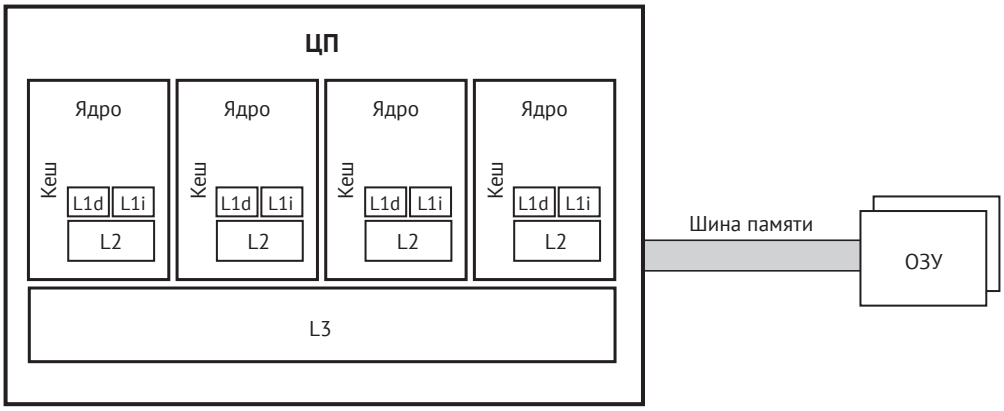


Рис. 2.12 ❖ Процессор с несколькими ядрами.
У каждого ядра имеется свой кеш первого уровня, разделенный на кеши команд (L1i) и данных (L1d), и кеш второго уровня (L2). Кеш третьего уровня (L3) общий для всех ядер. ЦП соединен с DRAM шиной памяти

Поскольку разные потоки могут работать на разных процессорах или ядрах, возникает проблема согласованности кешированных данных. У каждого ядра свой кеш первого и второго уровней, а общим является только кеш третьего уровня. Поэтому приходится вводить сложный механизм *когерентности кешей*, который описывает, как согласованность хранимых данных поддерживается с помощью *протокола когерентности кешей* – способа уведомления об изменении данных различными ядрами. Одно из основных состояний означает, что данные в локальном кеше модифицированы (для уведомления об этом служит *флаг модификации*). Информация о таком изменении должна быть распространена и обновлена по мере необходимости.

Существует много обобщений и продвинутых вариантов протоколов когерентности кешей, призванных обеспечить эффективность операций, в частности очень популярный *протокол MESI*. В его названии отражены четыре состояния, в которых может находиться строка кеша: изменена (Modified), монопольная (Exclusive), разделяемая (Shared) и недействительная (Invalid). Но все равно с этими протоколами связаны большие накладные расходы в части доступа к памяти, поэтому они снижают общую производительность системы. Интуитивно понятно, что необходимость в постоянных взаимных обновлениях кешей разных ядер может стать причиной заметных издержек. Наш код должен по возможности минимизировать доступ из разных ядер к адресам памяти в одинаковых строках кеша. В частности, это означает, что нужно вообще избегать межпотокового взаимодействия или, по крайней мере, тщательно продумывать, какие данные используются потоками совместно и как именно.

> **ПРИМЕЧАНИЕ** Вышеупомянутые команды некешируемого доступа обходят обычные правила когерентности кешей, поэтому использовать их следует в паре со специальной командой `sfence`, чтобы результаты были видны другим ядрам.

Но опять же – чем эта информация полезна в такой высокоуровневой среде, как .NET? Разве сборщик мусора со всеми своими знаниями и внутренними механиз-

мами не прячет такие детали реализации оборудования глубоко внутри? Ответ на этот вопрос дает следующий пример.

В листинге 2.6 показан многопоточный код, который может одновременно исполняться `threadsCount` потоками, обращающимися к одному и тому же массиву `sharedData`. Каждый поток просто увеличивает на 1 один элемент массива и (теоретически) никак не влияет на другие потоки. В нашем примере есть два важных параметра, показывающих, где эти элементы находятся внутри общего массива: расстояние от начала массива (`gap`) и расстояние между самими элементами (`offset`). Если выполнить этот код с параметром `threadsCount=4` на четырехъядерной машине, то, скорее всего, каждому потоку будет назначено свое физическое ядро.

Листинг 2.6 ❖ Возможность ложного разделения между потоками

```
const int offset = 1;
const int gap = 0;
public static int[] sharedData = new int[4 * offset + gap * offset];
public static long DoFalseSharingTest(int threadsCount, int size = 100_000_000)
{
   Thread[] workers = new Thread[threadsCount];
   for (int i = 0; i < threadsCount; ++i)
   {
      workers[i] = new Thread(new ParameterizedThreadStart(idx =>
      {
         int index = (int)idx + gap;
         for (int j = 0; j < size; ++j)
         {
            sharedData[index * offset] = sharedData[index * offset] + 1;
         }
      }));
   }
   for (int i = 0; i < threadsCount; ++i)
      workers[i].Start(i);
   for (int i = 0; i < threadsCount; ++i)
      workers[i].Join();
   return 0;
}
```

Таблица 2.3. Результаты выполнения кода из листинга 2.6, показывающие ложное влияние разделения на время обработки

Вариант	ПК	Ноутбук	Raspberry Pi 2
#1 (offset=1, gap=0)	5,0 с	6,7 с	29,0 с
#2 (offset=16, gap=0)	2,4 с	2,6 с	13,8 с
#3 (offset=16, gap=16)	0,7 с	0,8 с	12,1 с

В табл. 2.3 мы видим существенные различия в производительности между разными комбинациями `gap` и `offset`. Когда массив используется интуитивно понятным и простым способом, `gap = 0` и `offset = 1`. Соответствующий порядок доступа потоков показан на рис. 2.13а. Но при этом возникают очень большие накладные расходы на поддержание когерентности кешей. В каждом потоке (ядре) имеется собственная копия одной и той же области памяти (в строке его кеша), поэтому

каждая операция инкремента вынужденно делает недействительными локальные копии в других потоках. Это заставляет ядра постоянно инвалидировать свои кеши.

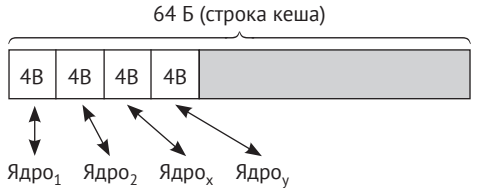


Рис. 2.13a ❖ Вариант 1 со смещением 1 без отступа – каждый доступ из потока модифицирует одну и ту же строку кеша

Очевидное решение проблемы – разнести элементы, к которым обращаются потоки, по разным строкам кеша. Для этого проще всего создать гораздо больший массив и использовать только каждый шестнадцатый элемент в нем (16 × 4 байта, размер одного `Int32`, как раз равно 64 байта). Этот вариант, когда `offset` равно 16, а `gap` по-прежнему равно 0, показан на рис. 2.13b. Из табл. 2.3 видно, что производительность заметно повысилась, но можно сделать еще больше.

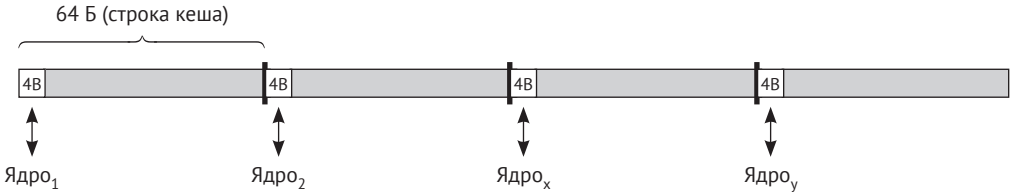


Рис. 2.13b ❖ Вариант 2 со смещением 16 без отступа – каждый доступ из потока модифицирует свою строку кеша

Здесь по-прежнему инвалидируется одна строка кеша, но это с первого взгляда не очевидно. Проблема получила название *ложное разделение* – неудачный паттерн доступа к данным, при котором теоретически не модифицированные разделяемые данные находятся в строке кеша, изменяемой каким-то другим потоком, что приводит к ее постоянной инвалидации. В следующей главе мы узнаем, что с каждым типом в .NET связан дополнительный заголовок, находящийся в его начале. В частности, это справедливо и для массивов. В случае массивов в начале объекта хранится важная информация – размер массива. При этом оператор доступа по индексу проверяет, не вышел ли индекс за пределы массива. Это означает, что при каждом доступе к любому элементу массива мы обращаемся в начало объекта массива, чтобы проверить его длину. Поэтому первое ядро разделяет начало объекта с остальными ядрами, что приводит к постоянной инвалидации соответствующих строк кеша. Чтобы исправить ситуацию, нужно сдвинуть элементы на длину одной строки кеша. Эта версия, в которой `offset = 16` и `gap` тоже равно 16, показана на рис. 2.13c.

В этом случае у каждого ядра имеется собственная локальная копия первой строки кеша, которая только читается. А модифицирует оно опять же собственные

строки кеша, содержащие данные. Протокол когерентности кешей не привносит никаких накладных расходов. Из табл. 2.3 видно, что это изменение ускоряет работу в 7 раз по сравнению с версией с ложным разделением!

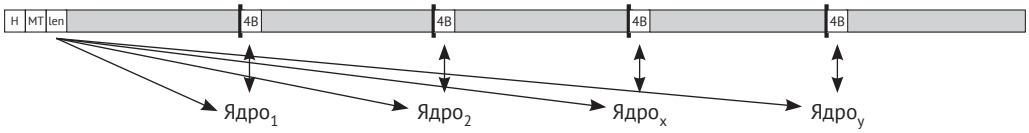


Рис. 2.13c ❖ Вариант 3 со смещением 16 и отступом 16 – каждый поток модифицирует свою строку кеша и читает общую строку кеша, в которой находится заголовок массива

В других архитектурах иногда отказываются от последовательной согласованности, реализованной в x86, что упрощает конструкцию, но усложняет программирование (необходимы явные барьеры в памяти). Пример такой архитектуры – процессор PowerPC, которым до 2006 года оснащались компьютеры Apple.

Мы потратили много времени, чтобы разобраться с кешированием данных. Но много страниц назад мы упомянули о кеше команд (L1i). Здесь мы не рассматриваем его по нескольким причинам. Во-первых, он создает гораздо меньше проблем. Компиляторы вполне могут позаботиться о надлежащей подготовке кода, а процессоры не менее хорошо справляются с угадыванием паттернов доступа к коду. В результате этот кеш работает хорошо – компилятор и сама природа исполнения программы обеспечивают приемлемую временную и пространственную локальность, которой может воспользоваться ЦП[1]. Кроме того, управление кешем команд не относится к управлению памятью в .NET, которое распространяется только на управление данными. Единственное очевидное указание в этом отношении – стремление сгенерировать минимальный код. Поскольку код кешируется вместе с данными на уровнях выше L1, он потребляет эти ценные ресурсы. Но воплотить этот совет на практике трудно – компилятор делает все возможное для оптимизации, а длина кода определяется главным образом его назначением.

Операционная система

До сих пор мы говорили об уровнях системы, очень близких к оборудованию. В самом начале я обещал также рассмотреть операционную систему. Сейчас для этого наступил подходящий момент. На самом деле проектировщики операционной системы должны очень серьезно относиться к изложенным выше фактам, которым мы уделили лишь беглое внимание. И, как мы скоро увидим, это лишь часть более широкой картины.

Архитектура оборудования и операционная система налагают ограничения на объем физической памяти – от 2 ГБ до 24 ТБ. А типичное устройство потребитель-

[1] Однако даже в .NET стоит проектировать вызовы методов, памятуя о возможности непопадания в кеш L1i. В основном это сводится к тому, чтобы избегать большого числа виртуальных вызовов и отдавать предпочтение повторным вызовам одного и того же метода для большого набора данных. Пример мы увидим в главе 10.

ского класса сегодня оснащается памятью размера от 4 до 8 ГБ. Если бы программа должна была работать с физической памятью напрямую, ей пришлось бы управлять всеми выделяемыми и удаляемыми блоками памяти. Мало того что логика такого управления сложна, она еще и повторялась бы в каждой программе. К тому же, с точки зрения низкоуровневого программирования, работа с памятью при таком подходе была бы очень громоздкой. Каждой программе пришлось бы запоминать, какие области памяти она использует, чтобы программы не мешали друг другу. Распределители должны были бы взаимодействовать с такими механизмами управления областями, чтобы правильно учитывать созданные и удаленные объекты. Что уж говорить про безопасность – без промежуточного уровня любая программа могла бы обращаться не только к своим областям памяти!

Виртуальная память

Поэтому была введена очень удобная абстракция – *виртуальная память*. Она перемещает логику управления памятью в операционную систему, которая предоставляет так называемое *виртуальное адресное пространство*. В частности, это означает, что каждый процесс думает, будто в системе работает только он один и вся память находится в его распоряжении. И даже больше, чем вся. Поскольку адресное пространство виртуальное, оно может быть больше физической памяти. Это позволяет расширить физическую DRAM-память за счет внешней, например расположенной на жестких дисках.

> **ПРИМЕЧАНИЕ** Существуют ли операционные системы без виртуальной памяти? Для устройств потребительского класса – нет. Но имеются специализированные, как правило, очень небольшие операционные системы и каркасы, ориентированные на встраиваемые системы. Один из примеров – ядро (micro)Clinux.

Здесь-то и вступает в игру *диспетчер памяти операционной системы*. У него две основные обязанности.

- Отображение виртуального адресного пространства на физическую память. В 32-разрядных машинах виртуальное адресное пространство 32-разрядное, а в 64-разрядных – 64-разрядное (в настоящее время используются только младшие 48 бит, но это все равно позволяет адресовать 128 ТБ данных и в то же время упрощает архитектуру и избавляет от ненужных накладных расходов).
- Перемещение некоторых участков памяти из ОЗУ на жесткие диски, когда они не используются, и обратно, когда в них вновь возникла необходимость. Очевидно, поскольку общий объем использованной памяти может быть больше, чем размер физической памяти, иногда приходится выгружать некоторые участки на более медленные носители, например жесткие диски. Место, где хранятся такие данные, называется *страничным файлом*, или *файлом подкачки*.

У диспетчера памяти ОС есть еще две дополнительные обязанности: управлять файлами, проецируемыми на память, и механизмом копирования при записи. Но здесь мы их касаться не будем, т. к. к нашим целям они отношения не имеют.

С выгрузкой части данных из ОЗУ во временное хранилище, очевидно, ассоциированы значительные потери производительности. В различных опе-

рационных системах этот процесс по историческим причинам называется то *подкачкой* (swapping), то *страничным обменом* (paging). В Windows имеется специальный *страничный файл*, в котором хранятся выгруженные из памяти данные, отсюда и термин «страничный обмен». В Linux такие данные хранятся в специальном *разделе подкачки*, отсюда и термин «подкачка» в Unix-подобных системах.

Виртуальная память реализована на уровне ЦП (с помощью *устройства управления памятью*, англ. MMU) и используется в кооперации с ОС. Виртуальная память организована в виде набора *страниц*. Было бы непрактично отображать виртуальную память на физическую побайтно, вместо этого отображаются целые страницы (непрерывные блоки памяти). С точки зрения операционной системы, страница является основной структурной единицей управления памятью. На рис. 2.14 схематически изображена виртуальная и физическая память.

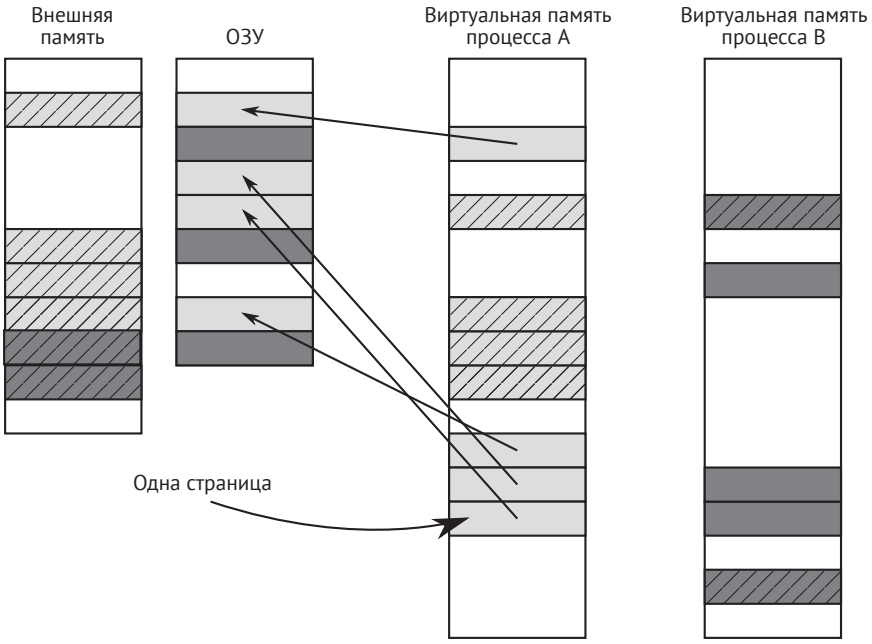


Рис. 2.14 ❖ Отображение виртуальных страниц на физические. Каждый процесс (A обозначен светло-серым, а B – темно-серым цветом) видит свое собственное виртуальное адресное пространство, но физически часть их страниц хранится в ОЗУ (страницы, закрашенные сплошным цветом), а часть выгружена на диск (заштрихованные страницы)

Кроме того, для каждого процесса ОС хранит *таблицу страниц*, которая позволяет отобразить виртуальный адрес в физический. Запись таблицы страниц содержит физический начальный адрес страницы и различные метаданные, например права доступа. В старые добрые времена отображение было одноуровневым, т. е. адрес состоял из *селектора страницы* и *смещения* от начала страницы – это показано на рис. 2.15.

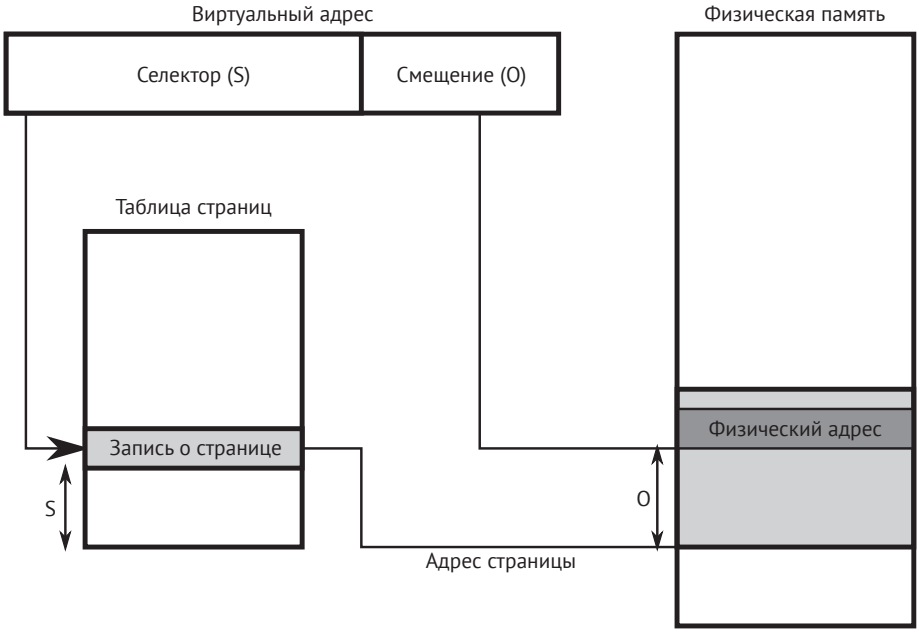


Рис. 2.15 ❖ Одноуровневая таблица страниц – виртуальный адрес состоит из селектора (S), который выбирает одну запись из таблицы страниц, и смещения (O) от начала этой страницы

Но у одноуровневой таблицы страницы есть существенный недостаток: либо страницы должны быть чрезмерно велики, либо таблица страниц получается очень большой. Большая страница составляет проблему, поскольку это приводило бы к растранжириванию ресурсов. При выделении памяти операционная система обязана производить выравнивание на границу страницы, поэтому даже для данных небольшого размера пришлось бы выделять целую большую страницу. С другой стороны, большая таблица страниц – тоже проблема, поскольку она хранится в основной памяти и создается для каждого процесса, так что это тоже непроизводительная трата ресурсов. Рассмотрим простое сопоставление размера страницы с размером таблицы для 32- и 64-разрядной машин (табл. 2.4).

Таблица 2.4. Размеры одноуровневой таблицы страниц на разных машинах

Размер страницы	Размер смещения	32-разрядная		64-разрядная	
		Размер селектора	Размер таблицы страниц	Размер селектора	Размер таблицы страниц
4 КВ	12 бит	20 бит	4 МБ	36 бит	512 ГБ
4 МБ	22 бит	10 бит	4 КБ	26 бит	512 МБ

Примечание: размер смещения должен быть достаточно большим, чтобы можно было адресовать любой байт на странице. Размер селектора равен длине адреса минус размер смещения. Размер таблицы страниц равен 2^селектор * длина адреса.

Такие огромные таблицы страниц на 64-разрядной машине нереализуемы. Если размер страницы равен 4 КБ, то в каждом процессе нужно зарезервировать 512 ГБ для таблицы страниц, что, очевидно, немыслимо. С другой стороны, при размере страницы 4 МБ слишком велики накладные расходы. Да и таблица размером 512 МБ все равно слишком велика.

Кроме того, процессы не потребляют всю доступную виртуальную память. Используемая память разбита на логические блоки (стек, куча, двоичный код и т. д.), поэтому таблицы получаются разреженными, с большими дырами, так что хранить всю таблицу – только зря растрачивать ресурсы.

В настоящее время применяются многоуровневые индексы. Это позволяет уплотнить разреженную таблицу страниц и одновременно оставить небольшой размер страницы. Сейчас в большинстве архитектур (включая x86, x64 и ARM) размер страницы равен 4 КБ, а таблица страниц четырехуровневая (рис. 2.16).

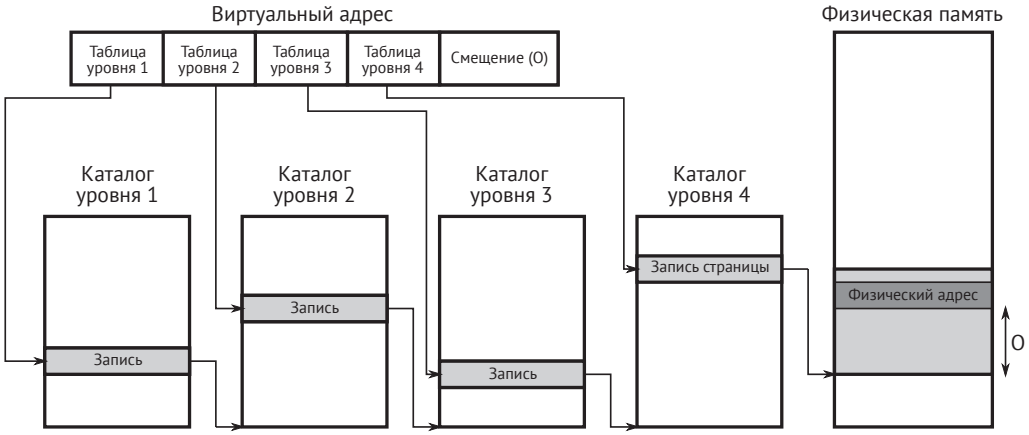


Рис. 2.16 ❖ Четырехуровневая таблица страниц и страница размером 4 КБ – три уровня селектора страниц позволяют представить гораздо больше разреженных данных

Для трансляции виртуального адреса в физический необходим *проход по таблице страниц*.

- Селектор уровня 1 выбирает запись в таблице уровня 1, которая указывает на одну из страниц в таблице уровня 2.
- Селектор уровня 2 выбирает запись на странице таблицы уровня 2, которая указывает на одну из страниц в таблице уровня 3.
- Селектор уровня 3 выбирает запись на странице таблицы уровня 3, которая указывает на одну из страниц в таблице уровня 4.
- Наконец, селектор уровня 4 выбирает запись на странице таблицы уровня 4, которая указывает на страницу в физической памяти.
- Смещение определяет конкретный адрес на выбранной странице.

Для такой трансляции необходимо обойти дерево, но, как мы сказали, таблица страниц хранится в основной памяти, как и все остальные данные. Это значит, что ее можно было бы кешировать в кешах L1, L2, L3. Но все равно накладные расходы были бы огромны, если бы для каждой трансляции адреса (а эта операция производится очень часто) приходилось бы обращаться к этим данным (даже если они

хранятся в кеше L1). Поэтому был введен *буфер ассоциативной трансляции* (Translation Look-Aside Buffer – TLB), который кеширует сам процесс трансляции. Идея проста – TLB работает как отображение, в котором селектор является ключом, а адрес физической страницы – значением. TLB конструируются исключительно быстрыми, поэтому они почти не привносят накладных расходов. Кроме того, они многоуровневые, как и вся структура таблицы страниц. Результатом непопадания в кеш TLB (если виртуальный адрес еще не был транслирован в физический) является полный проход по таблице страниц, который обходится дорого.

> **ИНТЕРЕСНОЕ ЗАМЕЧАНИЕ** Как и для любого кеша, упреждающая выборка в TLB нетривиальна – если бы процессор инициировал ее сам (например, в результате предсказания перехода), то мог бы быть выполнен ненужный проход по таблице страниц (поскольку предсказание может оказаться неверным). Поэтому упреждающая выборка инициируется программно.

Существуют ли оптимизации TLB, относящиеся к разработке ПО? Цель тут может быть только одна: уменьшить общее количество страниц, чтобы избежать большого количества непопаданий в TLB. Заодно это позволило бы уменьшить размер таблицы страниц и, стало быть, увеличить шансы на то, что результаты трансляции будут дольше оставаться в TLB. Однако на уровне .NET мы никак не можем повлиять на управление страницами.

> **ИНТЕРЕСНОЕ ЗАМЕЧАНИЕ** Обычно кеш L1 оперирует виртуальными адресами, потому что стоимость трансляции в физический адрес была бы намного больше стоимости самого доступа к быстрому кешу. Это означает, что когда страница изменяется, все строки кеша или хотя бы их часть должны стать недействительными. Таким образом, частая смена страниц негативно отражается и на производительности кешей.

Большие страницы

Из предыдущего обсуждения видно, что трансляция виртуальных адресов может стоить дорого, и хорошо было бы по возможности избегать ее. Основной подход к решению этой проблемы – использовать страницы большого размера. Тогда понадобилось бы меньше трансляций, поскольку на одной странице умещалось бы больше адресов. Но, как было сказано, большой размер страницы – напрасная трата ресурсов. Однако решение есть – так называемые *большие* (или *гигантские*) *страницы*. При наличии аппаратной поддержки мы можем создать большой непрерывный блок физической памяти, состоящий из многих последовательных страниц обычного размера. Такие страницы, как правило, на два или три порядка больше нормальных. Они могут быть полезны в ситуациях, когда программе необходим произвольный доступ к гигабайтам данных. Примером могут служить серверы баз данных. Операционная система Windows также отображает образ ядра и данные на большие страницы. Большие страницы невыгружаемые (их нельзя выгрузить в страничный файл) и поддерживаются как в Windows, так и в Linux. К сожалению, из-за фрагментации очень трудно выделить память для большой страницы, поскольку может не найтись подходящего непрерывного диапазона физической памяти.

В настоящее время большие страницы в .NET не используются, потому что в большинстве сценариев среда выполнения, напротив, хочет, чтобы страницы

были меньше. Однако же их использование стоит в списке будущих задач для .NET GC, правда, пока график разработки не определен. Мы тоже можем попробовать включить большие страницы в проект нашего собственного хоста CLR, описанного в главе 15.

Фрагментация виртуальной памяти

Как всегда, выделение и освобождение памяти сопровождается угрозой фрагментации. Мы упоминали это явление в главе 1 при обсуждении кучи. В случае виртуальной памяти это означает, что операционная система не может выделить непрерывный блок памяти требуемого размера, поскольку все свободные участки слишком малы, хотя их суммарный размер может значительно превосходить запрошенный.

Эта проблема особенно серьезна для 32-разрядных приложений, поскольку виртуальное адресное пространство может оказаться слишком маленьким для сегодняшних потребностей. Фрагментация может быть особенно сильной, когда процесс выделяет большие блоки памяти и работает долго: точно такая ситуация может возникнуть, например, в веб-приложениях для 32-разрядной версии .NET (размещаемых в IIS). Для предотвращения фрагментации процесс должен правильно управлять памятью (а для .NET таким процессом является сама CLR). Мы займемся такими деталями, когда будем описывать алгоритмы сборки мусора в главах 7–10, поскольку для их рассмотрения нужно лучше понимать саму платформу .NET.

Общая структура памяти

Разобравшись с основами организации памяти, мы теперь можем подняться на более высокий уровень. Первый вопрос: как устроена память программы? При описании типичной структуры памяти программы часто можно встретить картинку, показанную на рис. 2.17. Здесь дана структура виртуальной памяти программы, написанной на C или C++, и именно поэтому она нам интересна. В следующих главах мы узнаем, что CLR написана на C++, поэтому управляемые программы работают в похожей среде.

Легко видеть, что виртуальное адресное пространство разделено на две области:

- *пространство ядра* – область старших адресов занята самой операционной системой. Эта область называется пространством ядра, потому что ей владеет ядро, и только ядру разрешен доступ к ней;
- *пользовательское пространство* – область младших адресов назначена процессу. К ней имеет доступ пользовательский процесс.

Конечно, нас больше интересует пользовательское пространство, потому что именно в этой области размещается программа для .NET. Благодаря механизму виртуальной памяти каждый процесс видит память таким образом, как будто он единственный процесс в системе.

На схемах организации памяти младшие адреса обычно располагаются внизу (начиная с адреса 0), а старшие – вверху. Помните обсуждение стека и кучи в главе 1? По принятому соглашению, стек располагается в старших адресах,

а куча – под ним. Стек растет вниз, куча – вверх. Может возникнуть опасение, что стек рано или поздно столкнется с кучей, но на практике этого не бывает хотя бы в силу ограничений, наложенных на размер стека.

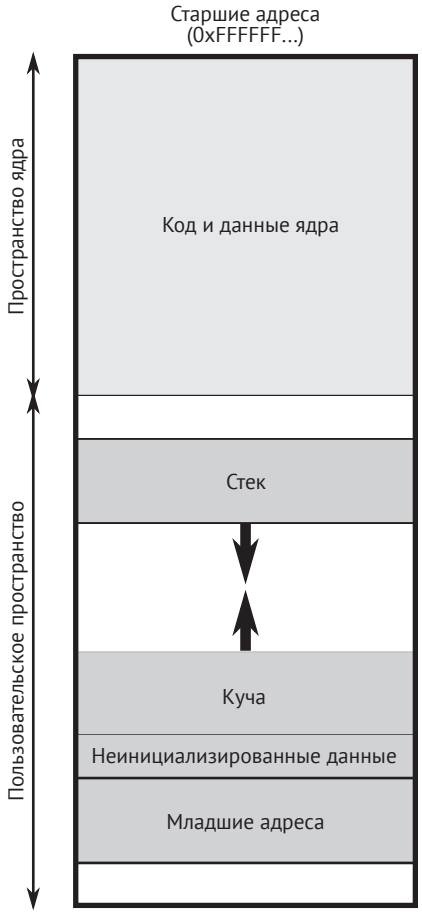


Рис. 2.17 ❖ Типичная структура памяти процесса

Перечислим остальные сегменты памяти на рис. 2.17.

- Сегмент *data* содержит инициализированные и неинициализированные глобальные и статические переменные.
- Сегмент *text* содержит исполняемый код приложения и строковые литералы. Он назван так в силу исторических причин, потому что раньше, по определению, содержал данные, доступные только для чтения.

Такая схема в принципе полезна для реализации структуры памяти в общем случае. Но, как мы скоро увидим, действительность сложнее. И лучше всего описать ее в контексте двух основных для .NET операционных систем: Windows и Linux.

Управление памятью в Windows

ОС Microsoft Windows – без сомнения, самое популярное окружение для платформы .NET. Поэтому разговор об управлении памятью в контексте операционной системы лучше всего начать с Windows.

Максимальный размер виртуального адресного пространства зависит от версии системы. Эти ограничения сведены в табл. 2.5.

Таблица 2.5. Ограничения на размер виртуального адресного пространства в Windows (пользовательское/ядро)

Тип процесса	Windows (32-разрядная)	Windows 8/Server 2012	Windows 8.1+/Server 2012+
32-разрядный	2/2 ГБ	2/2 ГБ	2/2 ГБ
32-разрядный (*)	3/1 ГБ	4 ГБ/8 ТБ	4 ГБ/128 ТБ
64-разрядный	–	8/8 ТБ	128/128 ТБ

* С флагом информированности о больших адресах (он же переключатель /3GB).

> **ПРИМЕЧАНИЕ** Существует также механизм Address Windowing Extensions (AWE), который позволяет выделить больше физической памяти, чем здесь указано, а затем отобразить только часть ее на виртуальное адресное пространство с помощью «окна AWE». Он особенно полезен в 32-разрядном окружении, чтобы преодолеть ограничение 2 или 3 ГБ на процесс. Но к нам это не относится, потому что CLR этим механизмом не пользуется.

Ограничения на размер виртуальной памяти, выделяемой одному процессу, стали сильно мешать в конце эры 32-разрядных систем. Ограничение 2 ГБ (или 3 ГБ в расширенном режиме) может препятствовать работе крупных корпоративных приложений. Классический пример – веб-приложение на платформе ASP.NET, размещенное в IIS на 32-разрядном компьютере с ОС Windows Server. Когда приложение упиралось в этот предел, не оставалось никакого другого выхода, кроме как перезагрузить его целиком. Это вынуждало прибегать к горизонтальному масштабированию крупных веб-систем, создавая несколько экземпляров сервера, каждый из которых обрабатывал меньше запросов и, стало быть, потреблял меньше памяти. Сегодня миром правят 64-разрядные системы и ограничения на размер виртуальной памяти перестали быть проблемой. Нам еще только предстоит дожить до времен, когда стандартной программе нужно будет десятки терабайтов памяти. Кстати, отметим, что программе, скомпилированной для 32-разрядной системы, выделяется не более 4 ГБ виртуальной памяти, даже если она работает в 64-разрядной ОС Windows Server.

Подсистема управления памятью в Windows представлена двумя основными уровнями:

- *Virtual API* – низкоуровневый API, работающий с отдельными страницами. Примерами являются функции `VirtualAlloc` и `VirtualFree`, о которых вы, возможно, слышали;
- *Heap API* – высокоуровневый API, предоставляющий распределитель (вспомните главу 1) для выделения блоков, меньших одной страницы. На этом уровне находятся, в частности, функции `HeapAlloc` и `HeapFree`.

Heap API (включающий диспетчер кучи) обычно используется в реализации управления памятью исполняющей средой C/C++. Вы наверняка знаете об операторах `new` и `delete` в C++ и функциях `malloc` и `free` из стандартной библиотеки C. Поскольку в CLR имеется собственная реализация распределителя для создания объектов .NET (мы познакомимся с ней в главе 6), она пользуется преимущественно Virtual API. Если не вдаваться в детали, то CLR запрашивает у операционной системы дополнительные страницы, а дальше уже сама выделяет память для объектов из этих страниц. Heap API также используется в CLR для создания небольших внутренних структур данных.

В Windows важно понимать различные категории памяти, ассоциированной с процессом. Это не так тривиально, как может показаться. Но без этого нам будет трудно разобраться в одном из самых важных вопросов – сколько памяти в действительности потребляет наблюдаемый процесс?

Чтобы ответить на этот вопрос, понадобятся дополнительные знания об управлении страницами в Windows. Страница может находиться в одном из четырех состояний:

- *свободна* – еще не передана никакому процессу и самой системе;
- *передана* (*частная*) – передана какому-то процессу. Такие страницы еще называют частными, потому что они могут использоваться только этим процессом. Когда процесс в первый раз обращается к переданной ему странице, та инициализируется нулями. Переданные страницы могут выгружаться на диск и загружаться обратно;
- *зарезервирована* – зарезервирована для процесса. Под резервированием понимается получение непрерывного диапазона виртуальных адресов без фактического выделения памяти. Это позволяет заранее зарезервировать память, а затем передавать ее по частям по мере необходимости. Физическая память при этом не потребляется, а лишь подготавливаются некоторые внутренние структуры данных. Программа может также сразу зарезервировать и передать память, если знает, какого размера блок ей нужен в данный момент;
- *разделяемая* – зарезервирована для какого-то процесса, но может совместно использоваться несколькими процессами. Обычно это относится к двоичным образам и спроецированным на память файлам системных библиотек (DLL) и ресурсов (шрифтов, переводов сообщений).

Частные страницы могут быть *заблокированными* – это запрещает перемещать их в страничный файл до явной разблокировки или завершения приложения. Блокировка полезна в критических с точки зрения производительности участках программы. Пример использования блокировки страниц в пользовательском хосте CLR будет показан в главе 15.

Зарезервированными и переданными страницами процесс управляет с помощью вызовов функций `VirtualAlloc`/`VirtualFree` и `VirtualLock`/`VirtualUnlock`. Отметим еще, что попытка доступа к свободной или зарезервированной памяти влечет за собой исключение нарушения доступа, потому что эту память еще нельзя отобразить на физическую.

> **ПРИМЕЧАНИЕ** Зачем понадобилось придумывать такой двухступенчатый процесс получения памяти? Как уже отмечалось, по многим причинам предпочтительнее по-

следовательный доступ к памяти. Если адресное пространство состоит из непрерывной последовательности страниц, то предотвращается фрагментация и, значит, оптимально используется TLB, без прохода по таблице страниц. И конечно, непрерывная память лучше с точки зрения использования кеша. Поэтому имеет смысл заранее зарезервировать немного побольше памяти, даже если сейчас она не нужна.

Теперь, зная о состояниях страниц, мы можем рассмотреть, какие области выделяются в памяти процесса в Windows (на рис. 2.18 эти области частично пересекаются):

- *рабочий набор* – это часть виртуального адресного пространства, которая в настоящий момент находится в физической памяти. Его можно разбить на следующие подкатегории:
  - *частный рабочий набор* – состоит из переданных (частных) страниц в физической памяти;
  - *разделяемый рабочий набор* – состоит из всех разделяемых страниц (не важно, действительно они используются совместно несколькими процессами или нет);
  - *совместно используемый рабочий набор* – состоит из всех разделяемых страниц, которые действительно используются совместно несколькими процессами;
- *байты исключительного пользования* – все переданные (частные) страницы в физической и выгруженной памяти;
- *байты виртуальной памяти* – переданная (частная) и зарезервированная память;
- *байты файла подкачки* – часть байтов виртуальной памяти, которая находится в страничном файле.

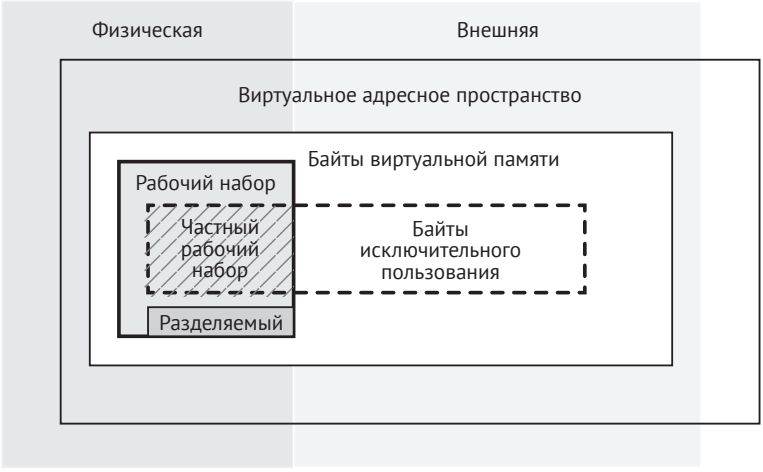


Рис. 2.18 ❖ Соотношения между различными категориями памяти процесса в Windows

Сложновато, да? Пожалуй, теперь мы понимаем, что ответ на вопрос «сколько памяти в действительности занимает процесс .NET» не столь очевиден. Какая из

этих величин нас интересует? Предполагается, что самая важная величина – частный рабочий набор, потому что она отражает фактическое влияние процесса на потребление физической ОЗУ – самой важной. Как следить за этими величинами, мы расскажем в следующей главе. Мы также разберемся, что именно отображает диспетчер задач в столбце «Память».

В силу внутренних соображений Windows при резервировании участка памяти для процесса соблюдает следующее ограничение – адрес начала участка и его размер должны быть кратны размеру системной страницы (обычно 4 КБ) и так называемой *гранулярности выделения памяти* (обычно 64 КБ). На деле это означает, что любой зарезервированный участок начинается по адресу, кратному 64 КБ, и его размер также кратен 64 КБ. Если мы захотим выделить меньше памяти, то остаток будет недоступен для использования. Поэтому важно правильно выравнивать и задавать размеры блоков памяти, чтобы память не расходовалась зря.

Проиллюстрируем на примере. Для этого напишем простой код, показанный на рис. 2.7. Он выделяет страницы виртуальной памяти, начиная с заданного адреса `baseAddress` размера `blockSize` (в байтах). Функция `VirtualAlloc` возвращает действительный адрес выделенной страницы `ptr`.

Листинг 2.7 ❖ Код выделения страниц с помощью Virtual API демонстрирует подводные камни, связанные с гранулярностью выделения памяти

```
IntPtr ptr = DllImports.VirtualAlloc(new IntPtr(baseAddress),
                                     new IntPtr(blockSize),
                                     DllImports.AllocationType.Reserve,
                                     DllImports.MemoryProtection.
                                     ReadWrite);
```

На рис. 2.19 мы видим результат работы этого кода в различных случаях. На рис. 2.19a показана одна еще не используемая страница, начинающаяся по адресу `0x9B0000`. На рис. 2.19b показана типичная, интуитивно понятная ситуация – зарезервировано 64 КБ памяти (размер одной страницы) по правильно выровненному адресу. В результате мы получаем эти 64 КБ по тому адресу, который указали (`ptr` будет равен `0x9B0000`). На рис. 2.19c показана очень похожая ситуация. Если резервируется 4 КБ по подходящему адресу, то выделяется целый блок размером 64 КБ, но его остаток (60 КБ) помечается как неиспользуемый. Эта память для нас потеряна – нет никакого способа воспользоваться ей. Такую ситуацию можно обнаружить с помощью программы VMMap, которую мы опишем в следующей главе.

На рис. 2.19d показана ситуация, когда размер блока не кратен размеру страницы – он округляется до ближайшего кратного. Так что, даже захоти мы выделить 6 КБ, все равно получим 8 КБ. Оставшиеся 56 КБ снова остались неиспользуемыми.

Похожая ситуация показана на рис. 2.19e, где базовый адрес сдвинут на 17 КБ (`0x9B4400`) и мы запросили 4 КБ. Теоретически нужно всего две страницы. Но `VirtualAlloc` все равно выделяет полные 64 КБ, начиная с адреса, кратного единице гранулярности (`0x9B0000`), а не с того, который мы указали.

С учетом всего вышесказанного, худшее, что можно сделать, – зарезервировать память близко к концу единицы гранулярности, как на рис. 2.19f. Здесь мы просим всего 8 КБ, а получаем два блока по 64 КБ, и почти половина этой памяти непригодна для использования.

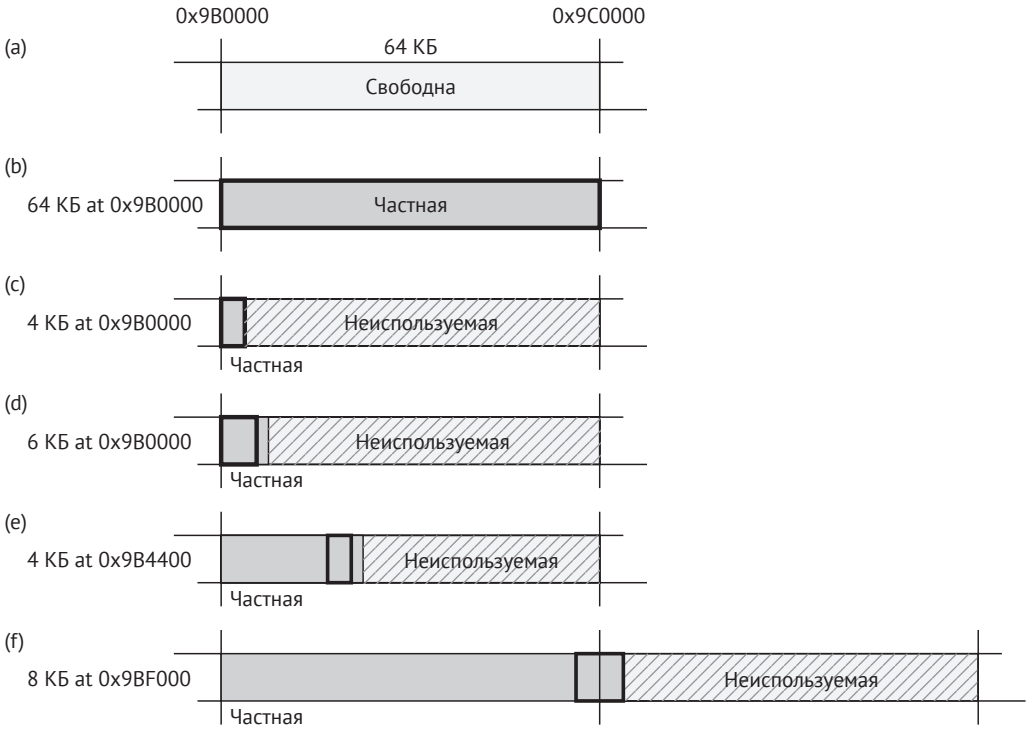


Рис. 2.19 ❖ Сверху вниз: (a) одна свободная страница до каких-либо действий; (b) зарезервировать 64 КБ, начиная с адреса 0x9B0000 (кратен 64 КБ); (c) зарезервировать 4 КБ (одну страницу), начиная с адреса 0x9B0000 (кратен 64 КБ); (d) зарезервировать 6 КБ (больше одной страницы), начиная с адреса 0x9B0000 (кратен 64 КБ); (e) зарезервировать 4 КБ (одну страницу), начиная с адреса, не выровненного на 2 КБ (0x9B0800), (f) зарезервировать 8 КБ (две страницы), начиная с сильно не выровненного адреса (0x9AF000)

Все это было написано для того, чтобы показать, как важно правильное выравнивание страницы. Хотя мы не каждый день управляем памятью на уровне Virtual API, эти знания помогут нам понять заботу о выравнивании в коде CLR. И конечно, вы с благодарностью вспомните об этом, когда в будущем понадобится написать такой низкоуровневый код.

> Внимательный читатель может поинтересоваться, почему гранулярность выделения памяти составляет 64 КБ, если размер страницы равен 4 КБ. На этот вопрос ответил Раймонд Чен (Raymond Chen) из компании Майкрософт, в статье 2003 года «Why is address space allocation granularity 64K?» (https://blogs.msdn.microsoft.com/oldnewthing/20031008-00/?p=42223). Как обычно в таких случаях, ответ очень интересен. Такая гранулярность объясняется в основном историческими причинами. Ядро всего семейства современных операционных систем корнями уходит в раннее ядро Windows NT. Оно поддерживало разные платформы, включая архитектуру DEC Alpha. И данное ограничение понадобилось исключительно для адаптации к этой платформе. А поскольку на других платформах оно не мешало, то было решено, что выгоды от общего базового кода ядра перевешивают неудобства сопровождения специального кода для одной платформы. Подробно о причинах, почему для этой платформы нужно именно такое значение, можете прочитать в статье.

Организация памяти в Windows

Теперь заглянем поглубже в процесс, работающий в Windows и исполняющий .NET-приложение. Процесс содержит одну кучу по умолчанию (в основном для внутренних функций Windows) и сколько угодно дополнительных куч (создаваемых с помощью Heap API). Примером может служить куча, создаваемая средой выполнения Microsoft C, из нее выделяют память вышеупомянутые операторы и функции C/C++. Существует три основных типа куч:

- *нормальная куча (NT)* – используется обычными приложениями (не написанными для универсальной платформы Windows – UWP). Предоставляет базовую функциональность управления блоками памяти;
- *куча с низкой фрагментацией* – дополнительный слой поверх функциональности нормальной кучи, который обеспечивает выделение памяти блоками нескольких предопределенных размеров. Это предотвращает фрагментацию из-за выделения небольших блоков и благодаря внутренним оптимизациям немного ускоряет доступ;
- *куча сегментов* – используется в приложениях для универсальной платформы Windows и предлагает более изощренные распределители (включая распределитель с низкой фрагментацией, аналогичный вышеупомянутому).

Говоря о структуре памяти процесса общего вида, мы отмечали, что виртуальное адресное пространство разделено на две части: старшие адреса заняты ядром, а младшие – пользовательской программой. Это показано на рис. 2.20 (32-разрядная архитектура слева, 64-разрядная справа). На 32-разрядных машинах пользовательское пространство в зависимости от флага больших адресов занимает младшие 2 или 3 ГБ. На современных 64-разрядных ЦП, поддерживающих 48-разрядную адресацию, пространство ядра и пользовательское пространство занимают по 128 ТБ (8 ТБ в предыдущих версиях – Windows 8 и Server 2012).

В некотором приближении можно сказать, что типичная структура пользовательского пространства .NET-приложения в Windows такова:

- вышеупомянутая куча по умолчанию;
- большинство образов (exe, dll) располагаются по старшим адресам;
- стеки потоков (рассмотренные в предыдущей главе) в основном располагаются по относительно младшим адресам, но могут находиться в любом месте. У каждого потока процесса имеется собственный стек. Это относится и к потокам CLR, которые пользуются платформенными потоками операционной системы;
- кучи GC, управляемые CLR, для хранения создаваемых нами объектов .NET (это обычные страницы Windows, полученные с помощью Virtual API);
- различные частные кучи CLR, управляемые CLR для ее внутренних целей. Мы подробно рассмотрим их в следующих главах;
- конечно, существует много свободного места – гигабайтные и терабайтные блоки (в зависимости от архитектуры) где-то в середине виртуального адресного пространства.

Начальный размер стека потока в Windows (зарезервированная и сразу же переданная память) берется из заголовка исполняемого файла (EXE-файла), но может быть также задан в аргументах методов типа `CreateThread` при создании потоков вручную с помощью Windows API.

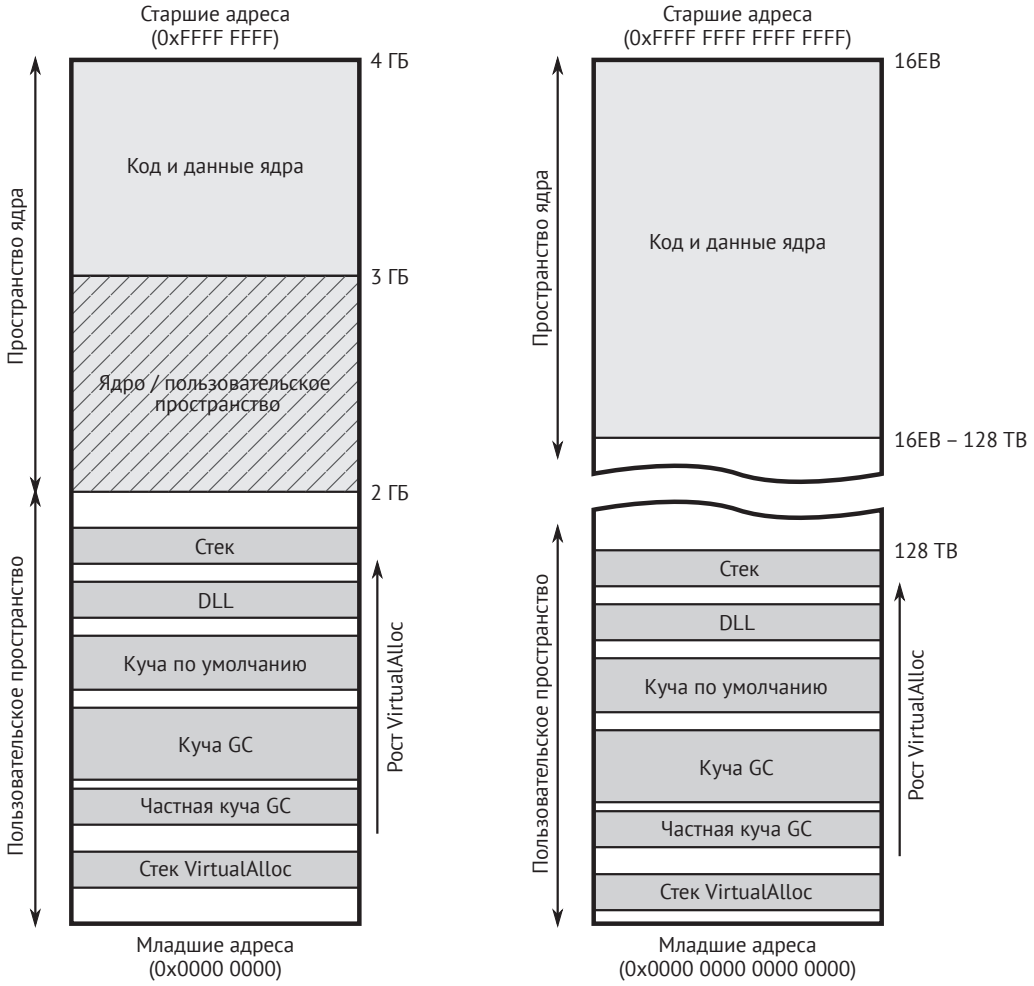


Рис. 2.20 ❖ Структура виртуальной памяти процесса Windows, исполняющего управляемый код .NET, для архитектуры x86/ARM (32-разрядной) и x64 (64-разрядной)

Способ, которым среда выполнения .NET вычисляет размер стека по умолчанию, весьма сложен. По умолчанию размер равен 1 МБ для типичного 32-разрядного приложения и 4 МБ для типичного 64-разрядного приложения. Размер данных в стеке довольно мал, а стек вызовов неглубокий (сотни вложенных вызовов считаются необычным явлением). Поэтому 1 МБ – разумное значение по умолчанию.

Если вы встретите исключение `StackOverflowException`, значит, уперлись в этот барьер. Но даже тогда это, скорее всего, ошибка, вызванная бесконечной рекурсией, которую, конечно, не исправишь увеличением размера стека. Если все-таки программа по какой-то причине должна хранить много данных в стеке, то можно модифицировать заголовок двоичного исполняемого файла. Исполняемый файл .NET интерпретируется как обычный исполняемый файл, поэтому операционная

система увидит это изменение. В главе 4 мы увидим, как увеличить предельный размер стека.

> Из соображений безопасности был введен механизм *рандомизации структуры адресного пространства* (Address Space Layout Randomization – ASLR), из-за которого структура на рис. 2.20 является всего лишь схемой, поскольку все компоненты (образы, кучи, стеки) могут располагаться в любом месте адресного пространства, чтобы злоумышленник не смог сделать никаких предположений об организации памяти.

Надеюсь, что такого взгляда с высоты птичьего полета хватит, чтобы лучше понять место памяти CLR в контексте всей экосистемы Windows. Мы вернемся к этой теме еще раз, когда будем подробно описывать структуру процесса CLR.

Управление памятью в Linux

Еще недавно глава, посвященная Linux, в книге о .NET могла бы послужить разве что для справки по проекту Mono. Но времена меняются. С пришествием среды .NET Core эта платформа пришла и в системы, не имеющие отношения к Windows. Более того, можно предвидеть растущую популярность .NET-приложений на не-Windows машинах. Мы уделим много внимания CoreCLR – реализации среды выполнения .NET Core. Но поскольку Linux становится все более популярной альтернативой, нам придется немного поговорить об этой системе. Так как в Linux используются те же аппаратные технологии – страничная организация, MMU и TLB, многое из рассмотренного выше остается в силе. Поэтому мы сосредоточимся лишь на интересующих нас различиях. Поскольку все большему числу людей придется разбираться с этой новой средой .NET, полагаю, будет весьма уместно хотя бы немного понимать и основы Linux.

В популярных и наиболее распространенных дистрибутивах ОС Linux также используется концепция виртуальной памяти. Ограничения на уровне процесса также похожи и сведены в табл. 2.6.

Таблица 2.6. Ограничения на размер виртуального адресного пространства в Linux (пользовательское/ядро)

Тип процесса	Linux 32-разрядная	Linux 64-разрядная
32-разрядный процесс	3/1, 2/2, 1/3 ГБ	–
64-разрядный процесс	–	128/128 ТБ*

* Каноническая 48-разрядная адресация.

Как и в Windows, основным структурным элементом в Linux является страница, которая обычно имеет размер 4 КБ. Страница может находиться в одном из трех состояний:

- *свободна* – не назначена ни системе, ни какому-либо процессу;
- *выделена* – назначена процессу;
- *разделяемая* – зарезервирована для процесса, но может использоваться также другими процессами. Обычно такие страницы выделяются двоичным образам и проецируемым на память файлам, например системным библиотекам и ресурсам.

Подобная картина потребления памяти процессом проще и понятнее, чем в Windows. Как видим, в отличие от Windows, отсутствует состояние резервирования страницы, хотя неявно оно существует. Для этого в Linux имеется механизм *ленивого выделения* памяти. Память, полученная от системы, считается выделенной, но физические ресурсы с ней еще не связаны (т. е. это аналог резервирования в Windows). Фактической передачи ресурсов (потребления физической памяти) не произойдет, пока кто-то не обратится к этому участку памяти. Если требуется заранее подготовить такие страницы в ситуациях, требующих максимальной производительности, то нужно «коснуться» их, например прочитать хотя бы один байт из страницы.

Зная возможные состояния страниц, мы можем рассмотреть, как организована память процесса в Linux. В этом вопросе имеется путаница. В различных инструментах Linux применяется несколько различающаяся терминология. Использование памяти процессом можно измерять по-разному:

- *виртуальная* (в некоторых инструментах обозначается `vsz`) – общий размер виртуального адресного пространства, зарезервированного для процесса. В популярной программе `top` эта величина показана в столбце `VIRT`;
- *резидентная* (*размер резидентного набора*, англ. *Resident Set Size*, RSS) – размер страниц, находящихся в данный момент в физической памяти. Некоторые резидентные страницы могут совместно использоваться другими процессами (как прочитанные из файла, так и анонимные). Это соответствует «рабочему набору» в Windows. Программа `top` показывает эту величину в столбце `RES`. Ее можно разбить на две части:
  - *частные резидентные страницы* – все анонимные резидентные страницы, зарезервированные для процесса (их число отражается в счетчике ядра `MM_ANONPAGES`). Это неполный аналог «частного рабочего набора» в Windows;
  - *разделяемые резидентные страницы* – прочитанные из файла (отражаются в счетчике ядра `MM_FILEPAGES`) и анонимные резидентные страницы процесса. Соответствует «разделяемому рабочему набору». Программа `top` показывает эту величину в столбце `SHR`;
- *частные* – все частные страницы процесса. Программа `top` показывает эту величину в столбце `DATA`. Отметим, что эта величина относится к зарезервированной памяти и ничего не говорит о том, какая часть этого набора уже испытала обращение и стала резидентной. Соответствует «байтам исключительного пользования» в Windows;
- *выгруженная* – часть виртуальной памяти, находящаяся в файле подкачки.

На рис. 2.21 графически изображены соотношения между этими величинами в виде пересекающихся множеств.

Довольно сложно. Как и в Windows, ответ на вопрос, на что расходуется память в нашем .NET-процессе, не тривиален. Самое разумное – смотреть на «частные резидентные страницы», потому что эта величина показывает фактическое потребление ценного ресурса ОЗУ процессом.

> Если в Windows гранулярность выделения памяти равна 64 КБ, то в Linux это просто размер страницы, который чаще всего равен 4 КБ.

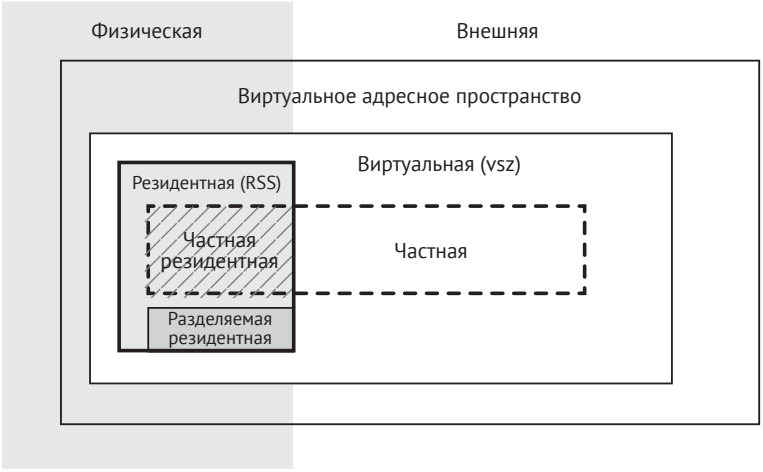


Рис. 2.21 ❖ Соотношение между различными категориями памяти процесса в Linux

Организация памяти в Linux

Память процесса в Linux организована почти так же, как в Windows. В 32-разрядной версии под пользовательское пространство отведено 3 ГБ, а под пространство ядра – 1 ГБ. Эту границу можно сдвинуть, задав параметр `CONFIG_PAGE_OFFSET` на этапе сборки ядра. В 64-разрядной версии граница проходит по тому же адресу, что в Windows (см. рис. 2.22).

Как и в Windows, система предоставляет API для работы со страницами памяти. Он включает следующие функции:

- `mmap` – прямое манипулирование страницами (включая проекции файлов, разделяемых и обычных, и анонимное отображение, не связанное ни с каким файлом, а используемое для хранения данных программы);
- `brk/sbrk` – ближайший аналог функции `VirtualAlloc`. Позволяет установить или увеличить «границу программы», т. е. по существу увеличить размер кучи.

В хорошо известных распределителях C/C++ используется `mmap` или `brk` в зависимости от размера выделяемого блока. Пороговое значение можно задать с помощью функции `mallopt` с параметром `M_MMAP_THRESHOLD`. Ниже мы увидим, что CoreCLR выбирает `mmap` для анонимных частных страниц.

Между Linux и Windows имеется важное различие в части работы со стеком потока. Поскольку двухшагового резервирования памяти не существует, стек расширяется по мере необходимости. Система не резервирует заранее соответствующие страницы памяти. А раз новые страницы создаются по запросу, стек потока уже не является непрерывной областью.

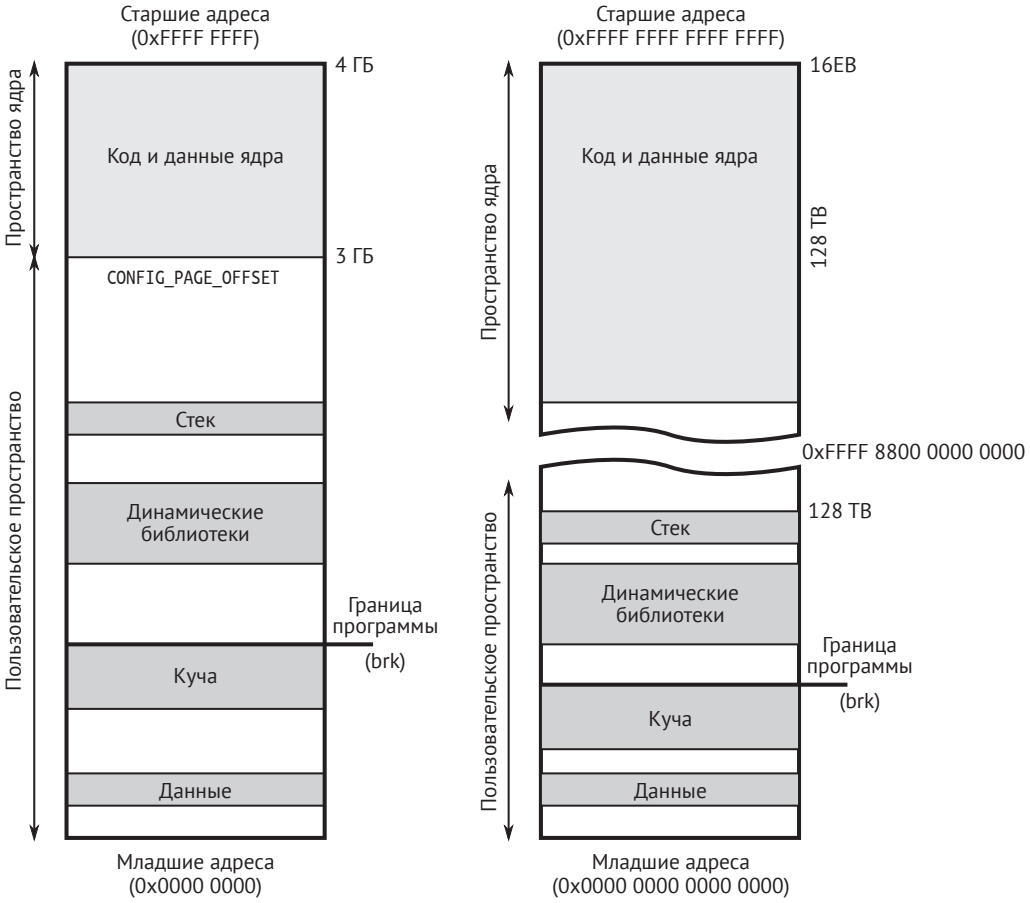


Рис. 2.22 ❖ Структура виртуальной памяти процесса Linux для архитектуры x86/ARM (32-разрядноjt) и x64 (64-разрядной)

Зависимость от операционной системы

Существуют ли такие различия в управлении памятью, которые следует принимать во внимание в кросс-платформенной версии сборщика мусора, включенной в CoreCLR? Вообще говоря, код GC очень слабо зависит от платформы, но очевидно, что где-то он все же должен делать системные вызовы. Подсистема управления памятью в обеих ОС работает похоже – она основана на виртуальной памяти, страничном обмене, и механизмы выделения памяти аналогичны. Хотя вызываемые системные API, конечно, разные, концептуальных различий в коде нет, за исключением двух ситуаций, которые я опишу ниже.

О первом различии мы уже упоминали. В Linux нет двухшаговой процедуры выделения памяти. В Windows можно предварительно использовать системный вызов для резервирования большого блока памяти. При этом инициализируются нужные структуры данных, но физическая память не выделяется. И лишь когда возникнет необходимость, мы выполним второй шаг – передачу программе фи-

зической памяти. Поскольку в Linux такого механизма нет, память просто выделяется, без всякого «резервирования». Однако необходим системный API для имитации такой двухшаговой схемы. Для этого применяется популярный трюк. В Linux «резервирование» имитируется путем выделения памяти в режиме доступа `PROT_NONE`, который фактически не дает никакого доступа к памяти. Однако в такой зарезервированной области мы можем еще раз выделить конкретные подобласти с нормальными правами доступа, имитируя тем самым «передачу» памяти.

Второе отличие связано с механизмом *наблюдения за записью в память*. В последующих главах мы увидим, что сборщику мусора необходимо отслеживать, какие области памяти (страницы) были модифицированы. Для этого в Windows имеется удобный API. При выделении страницы можно задать флаг `MEM_WRITE_WATCH`. Затем системный вызов `GetWriteWatch` позволяет получить список модифицированных страниц. При работе над CoreCLR обнаружилось, что в Linux нет надежного механизма, предлагающего подобный API. Поэтому соответствующую логику пришлось перенести в барьер записи (этот механизм подробно объясняется в главе 5), который поддерживается средой выполнения без помощи со стороны операционной системы.

NUMA и группы процессоров

Есть еще один важный фрагмент головоломки управления памятью, который стоит упомянуть в контексте оборудования и операционной системы. Под *симметричной многопроцессорностью* (SMP) понимается, что компьютер оснащен несколькими идентичными процессорами, подключенными к общей основной памяти. Они управляются одной операционной системой, которая может считать процессоры одинаковыми или разными. Как мы знаем, у каждого ЦП имеются свои кеши L1 и L2. Иными словами, у каждого ЦП есть выделенная локальная память, доступ к которой гораздо быстрее, чем к другим областям. Потоки и программы, работающие на разных ЦП, вероятно, будут сообща использовать какие-то данные, а это далеко не оптимально, потому что разделение данных посредством межсоединений процессоров привносит значительные задержки. Здесь-то и вступает в игру *архитектура с неравномерным доступом к памяти* (non-uniform memory architecture – NUMA). Это означает, что не вся общая память одинакова с точки зрения производительности. Программное обеспечение (в основном операционная система, но иногда и сама программа) должно поддерживать NUMA, если хочет предпочесть локальную память более удаленной. Такая конфигурация показана на рис. 2.23.

Дополнительные накладные расходы при доступе к нелокальной памяти называются *NUMA-фактором*. Поскольку прямые попарные соединения процессоров обошлись бы очень дорого, то обычно каждый ЦП соединен с двумя или тремя другими. При плохом сценарии для доступа к удаленной памяти нужно будет выполнить несколько переходов. Чем больше процессоров, тем значительнее влияние NUMA-фактора в случае, когда используется не только локальная память. Существуют также системы, в которых применяется смешанный подход: у группы процессоров память общая, а для разных групп доступ к памяти неоднородный с большим NUMA-фактором. На самом деле именно такой подход наиболее рас-

пространен в системах с поддержкой NUMA. Процессоры группируются в меньшие системы, называемые *NUMA-узлами*. У каждого NUMA-узла свои процессоры и память, для которой NUMA-фактор невелик благодаря аппаратной организации. Конечно, между NUMA-узлами есть соединения, но передача данных по ним влечет бóльшие накладные расходы.

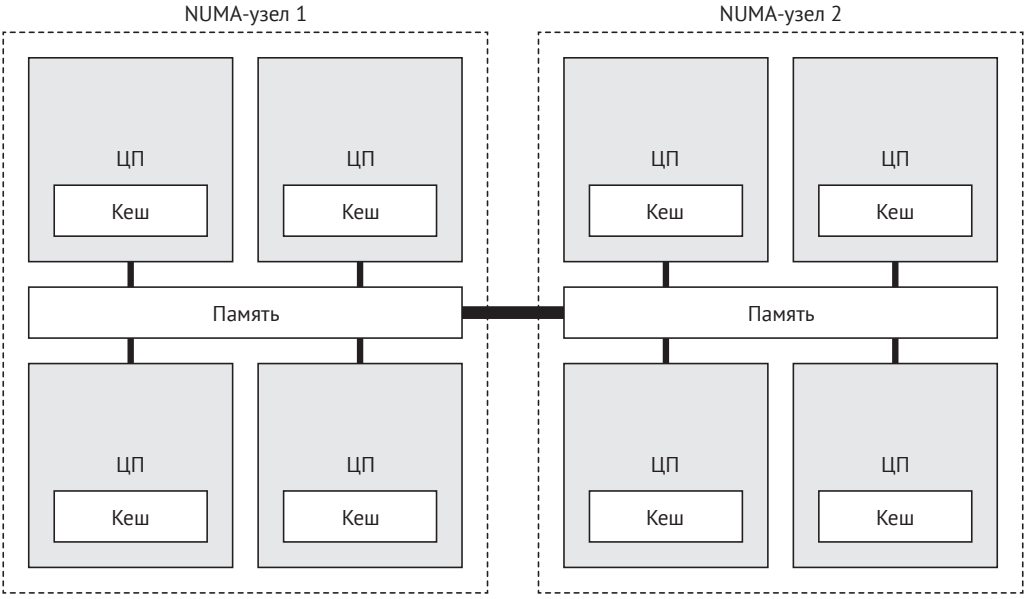


Рис. 2.23 ❖ Простая NUMA-конфигурация, состоящая из восьми процессоров, сгруппированных в два NUMA-узла

Основное требование к поддерживающим NUMA операционной системе и программе – использовать память в DRAM, локальной для NUMA-узла, содержащего исполняющий процесс ЦП. Но это может привести к несбалансированному состоянию, если одни процессы потребляют больше памяти, чем другие. В Linux поддержкой NUMA можно управлять на уровне процесса – должен ли он ограничиваться только локальной памятью (хорошо для небольших процессов) или может попытаться распределить ее более широко (для больших и гигантских процессов). В Windows поддержку NUMA необходимо учитывать на этапе разработки программы.

Возникает вопрос: поддерживает ли .NET CLR архитектуру NUMA? Простой ответ – да! Поддержку NUMA теоретически можно было бы выключить с помощью параметра `GCNumaAware` в разделе `runtime` конфигурационного файла, но в настоящее время этот параметр снаружи не виден.

Однако есть два других важных параметра приложений, относящихся к *группам процессоров* (см. листинг 2.8). В системах Windows, где логических процессоров больше 64, они объединяются в такие группы.

Мы можем включить поддержку NUMA для групп ЦП в среде выполнения .NET на машине с Windows. В средах, насчитывающих более 64 логических процессоров, это, очевидно, важно.

Листинг 2.8 ❖ Задание поддержки NUMA для групп процессоров в среде выполнения .NET

```
<configuration>
   <runtime>
      <Thread_UseAllCpuGroups enabled="true"/>
      <GCCpuGroup enabled="true"/>
      <gcServer enabled="true"/>
   </runtime>
</configuration>
```

Параметр `GCCpuGroup` говорит, должен ли сборщик мусора поддерживать группы ЦП путем создания внутренних потоков GC для всех имеющихся групп и должен ли он принимать во внимание существующие ядра при создании куч и управлении ими.

Параметр `Thread_UseAllCpuGroups` говорит, должна ли CLR распределять нормальные управляемые потоки (исполняющие наш код) между всеми группами ЦП. Оба параметра следует включать одновременно с параметром `gcServer`.

Резюме

В этой главе мы прошли длинный путь. Мы вкратце описали наиболее важные аппаратные и системные механизмы управления памятью. Надеюсь, что эти знания вкупе с теоретическим введением в первой главе открыли перед вами более широкий контекст – тот, в котором следует рассматривать управление памятью .NET. Надеюсь также, что вы стали с уважением относиться к сложности темы, если раньше думали иначе. Да, все, о чем мы говорили, является лишь фундаментом сборщика мусора в .NET! В каждой последующей главе мы будем все дальше отходить от уровня оборудования и общих теоретических положений. И все глубже забираться в среду .NET.

Правило 2: избегайте произвольного доступа, отдавайте предпочтение последовательному

Применимость: в основном к низкоуровневому коду, требующему максимальной производительности.

Обоснование: в силу внутренних механизмов на многих уровнях, включая ОЗУ и конструкцию процессорных кешей, последовательный доступ, безусловно, оптимальнее. Для доступа к DRAM требуется больше тактов процессора, чем для доступа к кешу. Процессор загружает данные 64-байтовыми блоками, называемыми строками кеша. Любой доступ к памяти, включающий менее 64 байтов, – пустая трата дорогостоящих ресурсов. Более того, в случае произвольного доступа маловероятно, что механизм упреждающей выборки в кеш даст эффект, поскольку у процессора не будет шансов предсказать закономерности доступа. Важно, что произвольный доступ необязательно полностью случайный, это лишь означает, что доступ не является упорядоченным, т. е. в нем не прослеживается никакая закономерность (по крайней мере, обнаруживаемая процессором).

Как применять: очевидно, что противоположностью произвольному доступу является доступ последовательный, поэтому старайтесь всегда работать именно так. При обработке большого объема данных стоит организовать их в виде массивов, расположенных в памяти непрерывно. Обход дважды связного списка – типичный пример неструктурированного доступа. Этот аспект доступа к памяти мы будем более подробно изучать в главе 3 при описании проектирования, ориентированного на данные.

Правило 3: улучшайте пространственную и временную локальность данных

Применимость: в основном к низкоуровневому коду, требующему максимальной производительности.

Обоснование: пространственная и временная локальность – столпы кеша. Если они присутствуют, то кеш используется эффективно и способствует повышению производительности. Наоборот, если мы препятствуем временной и пространственной локальности, то производительность резко снижается.

Как применять: проектируйте структуры данных с учетом локальности, стремясь максимально повысить степень повторного использования по времени. В приведенных выше примерах мы видели, что произвольный доступ к данным, находящимся в разных местах, крайне невыгоден с точки зрения производительности и может замедлять работу в несколько раз. Иногда, в тех частях программы, где производительность критична, приходится вносить интуитивно неочевидные изменения, которые будут описаны в главе 13. А иногда требуется лишь делать структуры данных не слишком большими, выделять для них память заранее и использовать повторно.

Правило 4: пользуйтесь продвинутыми средствами

Применимость. Код очень низкого уровня, требующий очень высокой производительности.

Обоснование. Среда выполнения .NET написана в максимально общем виде, поскольку должна правильно функционировать в самых разных ситуациях. Но при написании приложения требования нам точно известны. Может возникнуть необходимость, чтобы некоторые фрагменты кода работали с памятью очень быстро. Тогда стоит рассмотреть продвинутые механизмы, зависящие от операционной системы. Такая потребность возникает, наверное, не более чем у 0,0001 % разработчиков для .NET. Если вы пишете какую-нибудь библиотеку, активно работающую с памятью, например сериализатор, буферизованную систему обмена сообщениями или какой-нибудь сверхбыстрый обработчик событий, то, возможно, вам пригодятся некоторые из рассмотренных низкоуровневых системных API (например, некешируемый доступ к памяти).

Как применять. Потребуется писать по-настоящему непростой код. Им будет трудно управлять, и, возможно, никто не захочет сопровождать его. Кроме вас. И поскольку используется низкоуровневый API операционной системы, могут возникнуть проблемы при переходе на новую версию ОС. Крайне маловероятно,

что такое низкоуровневое управление памятью вообще стоит свеч, поскольку написание кода требует особой тщательности. А если допустить ошибку, то вместо повышения производительности можно получить обратный эффект.

Читайте эту книгу внимательно. А потом внимательно читайте книги о внутреннем устройстве конкретной операционной системы. И только потом можете попробовать применить продвинутые механизмы, такие как большие страницы, некешируемые операции и другие, упомянутые в этой главе.

Глава 3

Измерения памяти

Быть может, вам покажется странным встретить главу с таким названием чуть ли не в самом начале книги. Мы еще ничего толком не сказали об управлении памятью в .NET, а уже собрались говорить о связанных с ней инструментах? Но это обдуманное решение. Во-первых, с помощью описанных здесь инструментов я часто буду иллюстрировать обсуждаемые концепции. Во-вторых, хотя я и старался сделать книгу сбалансированной, у нее есть вполне практическое назначение. При обсуждении различных вопросов мы будем касаться реальных проблем и приводить примеры. Представленные в этой главе инструменты помогут нам найти и диагностировать проблемы. И коль скоро мы не собираемся ограничиваться теоретическим обсуждением устройства сборщика мусора, инструменты являются неотъемлемым приложением к теории.

Не зная, какие инструменты использовать, мы оказываемся в неудобном положении. Мы не знаем, есть ли у нашего процесса проблемы с памятью. Мы не знаем, связано ли большое потребление памяти или высокая загрузка процессора с управлением памятью в .NET. Мы не знаем, в чем причина наблюдаемого нежелательного поведения. Теперь что касается самих инструментов. Универсальной таблетки не существует. Иногда лучше взять один инструмент, а иногда – другой. Чтобы чувствовать себя комфортно, лучше изучить, как пользоваться всеми. По крайней мере, если мы хотим стать специалистами в этой области.

Описываемые инструменты различаются по степени сложности. С одной стороны, есть такие низкоуровневые средства, как WinDbg. С его помощью можно провести очень глубокий анализ. Знание десятков волшебных команд и умение применять их в нужном порядке позволит выяснить многое. С другой стороны, мы обнаружим коммерческие продукты, которые могут похвастаться удобным пользовательским интерфейсом. Здесь все легко и приятно, и многие ответы можно получить быстро. Даже еще не задав вопроса. Но эти инструменты дают только то, что было предусмотрено их авторами, а возможности настройки зачастую крайне ограничены. Между этими крайностями расположено много инструментов, предлагающих компромисс между гибкостью и простотой использования. По моему опыту, этих, скажем так, высокоуровневых коммерческих программ почти всегда достаточно. Но ключевое слово тут – «почти». Иногда мы сталкиваемся с проблемой, которую не удается решить с помощью предлагаемых ими аналитических средств. Иными словами, если подходить к вопросу серьезно, то рано или поздно придется залезть в двигатель и испачкаться в смазке.

Вы, наверное, удивитесь тому, что среди представленных инструментов нет почти ничего для статического анализа кода. Почти все инструменты основаны на

анализе во время выполнения программы. А все потому, что реальность сложна. Код может вести себя по-разному в зависимости от особенностей использования. Даже самый неэффективно написанный код управления памятью не окажет негативного влияния на процесс, если он выполняется раз в час. Статический анализ кода может помочь, но может и навредить, если побудит вас уделить избыточное внимание несущественным частям кода.

Производительность – вещь более сложная, чем функциональность или качество кода, поскольку мы зачастую не знаем, как «могло бы быть» и «как должно быть». Есть инструменты, позволяющие обнаружить превышение некоторых порогов. Но даже в этом случае без владения предметом мы не знаем, применимы ли эти пороги к нашему приложению при данных условиях. Поэтому хотя эта глава очень важна, она будет не особенно полезна без информации, полученной из остальных глав.

Способы измерения поведения .NET-программ принципиально зависят от операционной системы. Поэтому глава состоит из двух частей: для Windows и для Linux. Поскольку .NET не очень популярна на компьютерах с macOS, инструменты для этой платформы не описываются.

Важно отметить, что в этой главе описаны различные инструменты и основы работы с ними. А способы использования и интерпретации результатов будут представлены далее в этой книге. Мы еще не обладаем достаточными знаниями о сборщике мусора, чтобы начать использовать эти инструменты для решения конкретных проблем. Относитесь к этой главе как к полному перечню инструментов, которые можно и нужно применять. Рекомендую попробовать каждый по ходу чтения, хоть чуть-чуть. Это позволит свести с ними практическое знакомство и понять, что есть что, – в последующих главах будет полезно. Конечно, с некоторыми инструментами вы уже можете быть знакомы. Тогда пропускайте их описание, особенно те части, где речь идет о базовых приемах использования.

Отметим еще, что этой главе свойственна проблема яйца или курицы – невозможно продемонстрировать многие связанные с GC аспекты на практике без описанных здесь инструментов, но сами инструменты требуют хорошего понимания этих аспектов. Чтобы не загромождать книгу фрагментами описания, разбросанными то тут, то там, мы представим здесь основные сведения о работе с инструментами, даже если для этого понадобится сослаться на какие-то вещи, относящиеся к GC. Так что не волнуйтесь, если что-то покажется непонятным. Я ожидаю, что время от времени вы будете возвращаться к этой главе, когда начнете применять описанные инструменты в реальной работе, уже вооруженные знаниями, почерпнутыми из этой книги.

Измеряйте как можно раньше

Когда задаешь экспертам, разработчикам библиотек или просто профессионалам, немало повидавшим на своем веку, вопрос: «Что нужно прежде всего делать, когда вы начинаете думать о производительности?» – все отвечают одинаково: *измерять как можно раньше*. Каждый, наверное, слышал о том, что преждевременная оптимизация – корень всех зол. Во-первых, просто не имеет смысла тратить часы

или дни на оптимизацию кода, которая дает пренебрежимо малый эффект и не окупается ни экономией аппаратных ресурсов, ни уменьшением времени работы приложения. Хуже того, это только увеличивает стоимость разработки. И быть может, усложняет и делает менее понятным код. Правильнее делать ровно наоборот – вместо того чтобы раньше времени бросаться оптимизировать, необходимо провести измерения и понять, есть ли вообще необходимость в оптимизации. А поскольку это книга об управлении памятью в .NET, мы приходим к очередному общему правилу (я опишу его в конце главы) – измерять GC как можно раньше.

У каждого измерения может быть погрешность – иногда больше, иногда меньше. Кроме того, само измерение может влиять на наблюдаемый процесс. Мы знаем об этом из физики, а с измерениями параметров процесса дело обстоит точно так же. Поэтому ответ на вопрос «как измерять» может быть как очень простым (если мы не вдаемся в детали), так и очень сложным (если нас интересует точность). Разные инструменты дают разную точность, и мы немного поговорим об этом. Однако статистические рассуждения на тему погрешности измерений выходят за рамки этой книги. Просто имейте в виду, что при измерении всегда имеет место некоторая неточность.

Но все же я хочу упомянуть здесь несколько важных понятий и заблуждений, относящихся к измерениям. Мы встретимся с ними в этой и других главах. А самое главное – встретимся в повседневной работе.

Накладные расходы и вмешательство

При оценке различных инструментов измерения важно помнить о двух моментах.

- *Накладные расходы* – трудно найти измерительный инструмент, использование которого не замедляло бы работу приложения или не приводило бы к потреблению дополнительных ресурсов. Поэтому следует говорить о накладных расходах, которые обычно выражаются в процентах. У некоторых инструментов накладные расходы едва заметны, всего несколько процентов. То есть, например, время реакции веб-приложения будет на несколько процентов больше. Или на несколько процентов уменьшится скорость анимации в настольном приложении. Инструменты со столь низкими накладными расходами можно использовать даже во время промышленной эксплуатации. С другой стороны, есть инструменты, которые при подключении замедляют приложение на несколько порядков. Правда, в качестве компенсации они предоставляют очень подробную информацию. Однако из-за таких больших накладных расходов использовать их можно только во время разработки или на рабочей станции с одним пользователем.
- *Вмешательство* – речь идет о том, в какой мере инструмент влияет на функциональность приложения. Требует ли он перезапуска приложения? Нужны ли дополнительные права? Следует ли установить дополнительные расширения? Идеальный невмешивающийся инструмент можно включать и выключать во время работы приложения, и никто этого не заметит. С другой стороны, максимально вмешивающийся инструмент требует перекомпиляции и повторного развертывания приложения в данной среде.

Выборка и трассировка

Еще одна характеристика инструмента – способ сбора диагностической информации. Есть два основных подхода.

- *Трассировка* – в этом случае диагностические данные собираются в момент возникновения конкретных событий (отсюда другое название подхода – *событийно-ориентированный*). Примером может служить сохранение данных при открытии или закрытии файла, в момент щелчка мышью или в начале процесса сборки мусора. Неоспоримое достоинство такого решения – точность данных, поскольку они получены в момент события, и мы можем протоколировать все события данного типа. Но если события происходят часто, то накладные расходы будут очень велики. Поэтому этот механизм не используется для таких частых и низкоуровневых событий, как вызов функции и возврат из функции. Впрочем, если мы можем позволить себе любые накладные расходы, например на локальном компьютере разработчика, то почему бы и нет?
- *Выборка* – при таком подходе мы соглашаемся на потерю точности и собираем диагностические данные лишь время от времени (отсюда другое название подхода – *периодический*). Мы лишь стараемся получить состояние приложения, и чем реже это делаем, тем менее точны результаты измерений. Типичный пример – периодическое, например один раз в миллисекунду, сохранение стеков вызова функций на всех процессорах. Это позволит статистически оценить, какие функции выполняются дольше всего. При этом мы, конечно, теряем информацию о функциях, которые работают меньше миллисекунды.

Дерево вызовов

Один из самых распространенных способов визуализировать поведение приложения – построить *дерево вызовов*. Каждый узел такого дерева представляет одну функцию, а дочерние узлы представляют вызванные из нее функции. С каждой функцией связан результат некоего измерения, обычно полное время выполнения. Очень часто с каждой функцией (каждым узлом) ассоциируют два показателя:

- *собственный, исключая дочерние вызовы* (exclusive) – измеряется только значение для самой функции. Если речь идет о времени выполнения, то это время, потраченное в этой и только в этой функции;
- *полный, включая дочерние вызовы* (inclusive) – измеряется значение для самой функции и сумма измерений для всех потомков. В случае времени выполнения это будет время, потраченное в самой функции, функциях, вызванных из нее, функциях, вызванных из вызванных, и так далее рекурсивно.

Кроме того, часто определяется процентная доля этого показателя относительно всего исследуемого диапазона. В этом случае употребляются названия *полный%* (*inclusive%*) и *собственный% (exclusive%)*. На рис. 3.1 показаны результаты гипотетического профилировщика.

Мы видим, что в функции `main` потрачено 100 % полного времени работы программы, что составило 3 секунды. Поскольку функция `main` вызывает все остальные функции, это ожидаемый результат. Но лишь 22 % этого времени потрачено в самой функции `main`, остальное – в вызываемых из нее функциях. Например, 78 % времени потрачено в функции `SomeClass.Method1`, а 66.7 % этого времени – в другой функции `SomeClass.HelperMethod`. Пробежавшись по дереву вызовов, мы очень быстро найдем самые медленные компоненты приложения.

Отметим, что такие деревья обычно представляют агрегированные данные. В нашем примере агрегируются все вызовы каждой функции. Так, функция `main` вызывалась только один раз, тогда как `HelperMethod` – две тысячи раз (что объясняет, почему агрегированное полное время настолько велико). Поэтому анализ такого дерева обнаруживает как долго работающие методы, так и методы, которые, возможно, работают быстро, но вызываются много раз.

Method name	Inclusive [ms]	Inclusive [%]	Exclusive [ms]	Exclusive [%]	Exclusive Counter
− main	3000	100.0	660	22%	1
− SomeClass.Method1	2340	78.0	50	1.7%	3
+ SomeClass.HelperMethod	2000	66.7	200	6.7%	2000
+ OtherClass.MethodA	360	12.0	10	3.3%	20
+ OtherClass.MethodB	120	4.0	10	3.3%	21

Рис. 3.1 ❖ Пример дерева вызовов, показывающего информацию о производительности

Та же идея применима для визуализации использования памяти, только в этом случае каждый узел представляет конкретный тип объектов. Дочерними узлами будут другие типы, экземпляры которых данный объект содержит непосредственно или на которые он ссылается. Поверьте мне, анализируя производительность или потребление памяти своим приложением, вы часто будете использовать такие визуализации.

Графы объектов

Если говорить о памяти, то мы часто используем граф, представляющий связи между объектами в памяти, он называется *графом объектов*, или *графом ссылок*. Пример такого графа показан на рис. 1.12 в первой главе и на рис. 3.2 ниже. Мы видим множество объектов, некоторые из которых ссылаются на другие, и только один корневой объект. Вообще говоря, такие графы для программы обычного размера могут быть очень велики, и визуализировать их нелегко, поэтому, как правило, мы анализируем только ту или иную часть. Графы можно использовать для показа как агрегированной информации (сколько экземпляров данного типа ссылаются на другие типы), так и информации о конкретном экземпляре (на какие другие объекты ссылаются данные объекты).

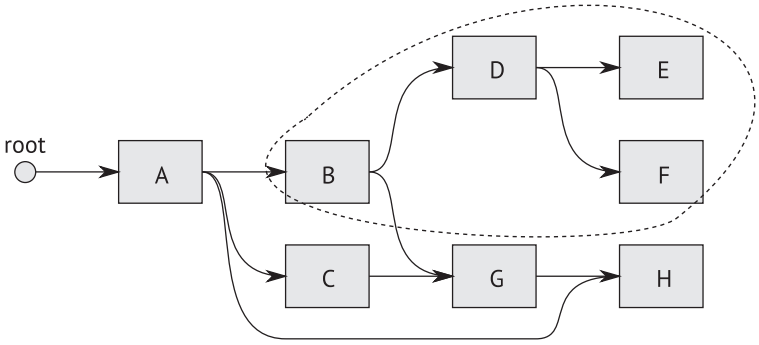


Рис. 3.2 ❖ Пример графа объектов.
Собственный подграф объекта B дополнительно обведен пунктирной линией

Граф объектов открывает доступ к трем важным понятиям, встречающимся в различных инструментах.

- *Кратчайший путь к корню* – это кратчайший путь по ссылкам от выбранного объекта до какого-то корня. Граф может быть сложным, в нем может существовать несколько путей между корнем (или даже несколькими корнями) и объектом, но, очевидно, среди них есть кратчайший. На рис. 3.2 кратчайшим путем к корню от объекта H является путь root-A-H. Существуют и более длинные пути: root-A-C-G-H и root-A-B-G-H. Кратчайший путь к корню важен, потому что обычно он показывает основные и самые сильные связи между объектами и, следовательно, хорошо объясняет главную причину, по которой объект нельзя считать недостижимым (а значит, нельзя и удалить). Другие пути часто являются побочным эффектом более сложных зависимостей. Однако иногда кратчайший путь к корню вводит в заблуждение, поскольку создан какими-то (быть может, временными) вспомогательными ссылками, например из кеша. Такая ситуация, по всей видимости, имеет место на рис. 3.2, где объект A, вероятно, хранит ссылку на объект H для удобства (например, для кеширования), тогда как истинный владелец H – один из объектов B, C или G.
- *Подграф зависимостей* – этот подграф содержит сам выбранный объект и все объекты, на которые он прямо или косвенно ссылается. Так, на рис. 3.2 подграф зависимостей объекта B содержит B, а также объекты D, E, F, G и H.
- *Собственный подграф* (retained subgraph) – подграф, содержащий объекты, которые могут быть удалены, если удалить выбранный объект. Поскольку граф зависимостей может быть сложным, вовсе необязательно, что удаление объекта повлечет за собой удаление всех зависимых от него объектов, поскольку на них могут ссылаться и другие объекты. Собственный подграф объекта B на рис. 3.2 содержит объекты B, D, E и F.

Наряду с этими понятиями есть понятие размера объекта, и различные инструменты интерпретируют его по-разному.

- *Поверхностный размер* (shallow size) – размер самого объекта (всех его полей, включая ссылочные). Этот размер вычисляется легко.
- *Полный размер* (total size) – сумма поверхностного размера объекта и поверхностных размеров всех объектов, на которые он прямо или косвенно ссылается. Иными словами, это сумма поверхностных размеров всех объектов в подграфе зависимостей. Его тоже легко вычислить, потому что нужно

лишь найти подграф зависимостей объекта и просуммировать поверхностные размеры всех входящих в него объектов.

- *Собственный размер* (retained size) – сумма поверхностных размеров всех объектов в собственном подграфе. Иными словами, это объем памяти, который освободится после удаления данного объекта. Чем больше ссылок на объекты в графе зависимостей, тем меньше собственный размер по сравнению с полным. Его посчитать труднее всего, потому что необходим сложный анализ всего графа объектов.

Когда инструмент показывает размер объекта, стоить задаться вопросом, какой из вышеуказанных «размеров» имеется в виду.

Статистика

Всякий раз, как мы тем или иным способом агрегируем результаты измерений, мы пользуемся статистическими инструментами. Если делать это необдуманно, есть риск прийти к ошибочным выводам. Например, самый распространенный способ агрегирования данных – вычисление *среднего*, интерпретируемого как «типичное значение». Но у среднего есть два крупных недостатка: оно необязательно совпадает с каким-то конкретным значением выборки (кто-нибудь видел 2,43 ребенка в средней семье?) и может скрыть истинную природу распределения данных (как мы скоро убедимся). Проблемы этой и других простых метрик, например дисперсии, убедительно иллюстрирует *квартет Энскомба* (см. рис. 3.3, взятый из «Википедии»). Иногда статистические характеристики совершенно разных наборов данных одинаковы.

Достоинство и причина популярности среднего – в его интуитивной понятности и простоте вычисления даже без необходимости хранить отдельные образцы: получив очередной пример, мы прибавляем его к сумме и делим на общее число примеров в выборке. Для применения других методов агрегирования необходимо хранить все образцы, а это может заметно увеличить нагрузку на инструмент.

Какие еще методы агрегирования следует использовать? Перечислим самые распространенные.

- *Медиана* – такое значение, что половина образцов больше него, а половина меньше. Она дает лучшее представление о типичном значении, поскольку более устойчива к выбросам. Кроме того, она совпадает с одним из реальных образцов, а не вычислена искусственно.
- *Перцентиль* – такое значение, что заданная процентная доля выборки меньше него. Например, 95-й перцентиль – это значение, меньше которого 95 % образцов. Он прекрасно характеризует интересующие нас данные, не принимая в расчет выбивающиеся из общего ряда результаты измерения. Я настоятельно рекомендую измерять перцентили при использовании инструментов. Перцентили часто имеют прямое отношение к предметной области. Например, мы хотим, чтобы в 90 % случаев время отклика приложения было не больше 1 секунды, а в 99 % случаев – не больше 4 секунд. Для контроля нужно измерять 90-й и 99-й перцентили времени отклика.
- *Гистограмма* – графическое представление распределения образцов. Показывает, сколько образцов попадает в каждый диапазон значений. Это лучшее из возможных измерений, потому что описывает распределение данных целиком.

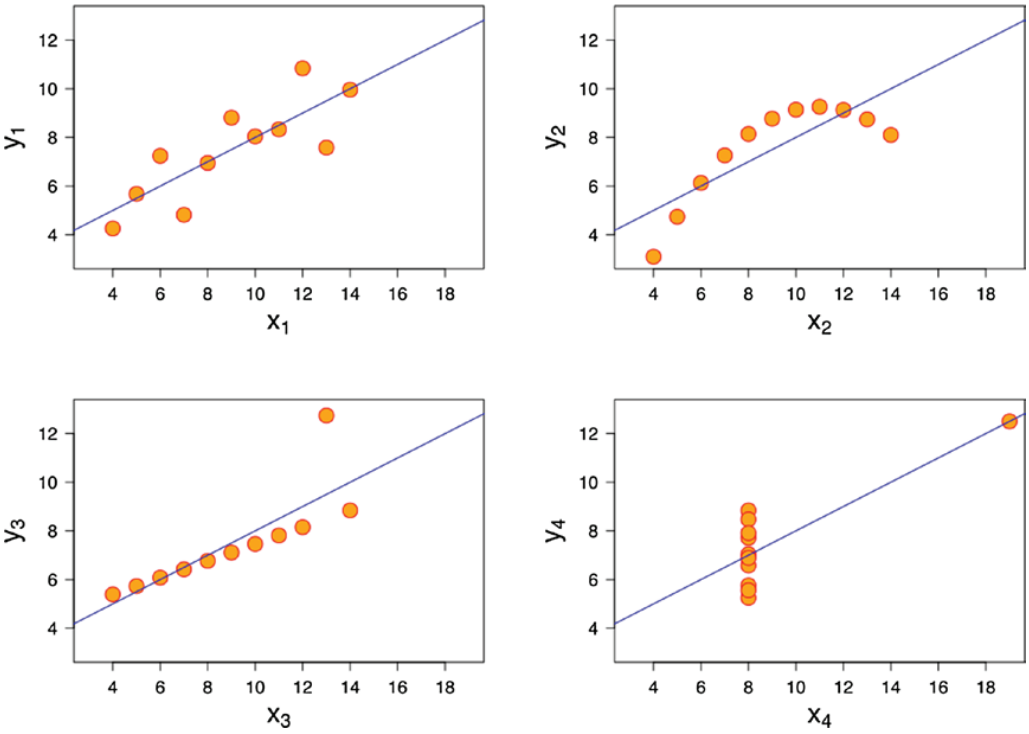


Рис. 3.3 ❖ Квартет Энскомба – четыре набора данных с одинаковыми средними и дисперсиями по осям x и y. Источник: «Википедия»

Все эти метрики показаны на рис. 3.4 – гистограмме распределения времени отклика. Мы видим, сколько значений (выраженных в миллисекундах) попадает в каждый диапазон. Из гистограммы наглядно видно, что по большей части время отклика попадает в диапазон 110 ± 5 мс, и чем сильнее время отклика отличается от этих значений, тем реже оно встречается. Более того, можно сказать, что:

- среднее время отклика равно 104,3 мс;
- 10 % всех значений времени отклика меньше 60 мс (10-й перцентиль);
- медиана равна 100 мс;
- 90 % всех значений времени отклика меньше 150 мс (90-й перцентиль).

Распределение, показанное на рис. 3.4, очень похоже на *нормальное*, которое часто называют также *колоколообразной кривой* за его характерную форму. Результаты многих измерений распределены таким образом, что делает интерпретацию перцентилей (и даже среднего) вполне естественной.

Однако следует помнить и о возможности *бимодальных* (и вообще – *многомодальных*) распределений, для которых среднее и даже медиана и перцентили не имеют особого смысла (рис. 3.5). Очевидно, что в данном случае существует два типа откликов (на самом деле два разных нормальных распределения), поэтому их совместное агрегирование только сбивает с толку. Правильнее было бы сказать, что существует две категории откликов с медианами 40 и 150 мс (и, наверное, выяснить, откуда вообще взялось такое бимодальное распределение времени отклика).

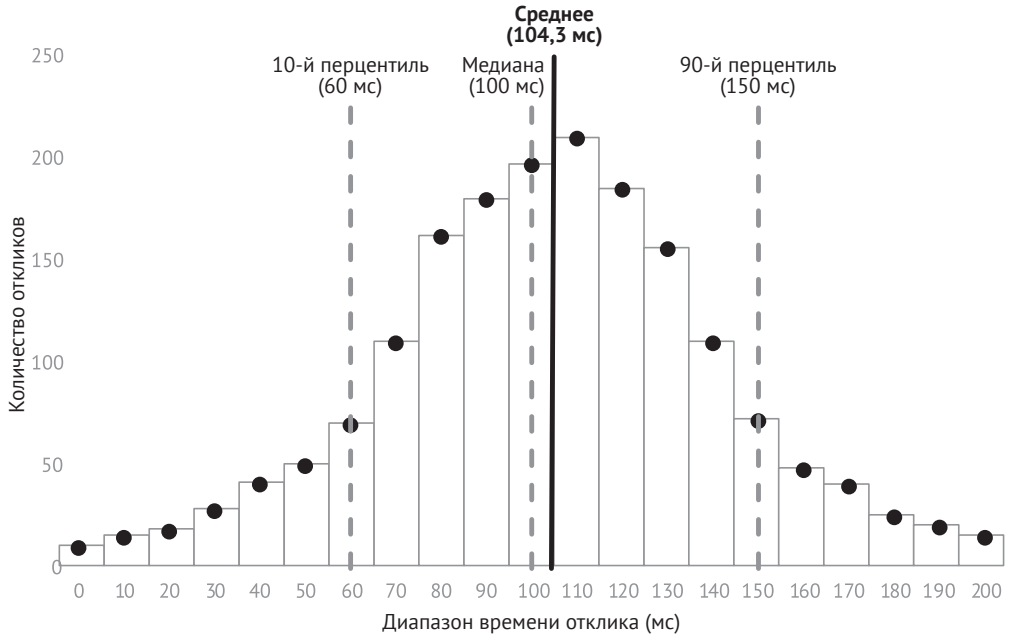


Рис. 3.4 ❖ Пример гистограммы, а также значений медианы, 10-го и 90-го перцентилей. Распределение данных нормальное

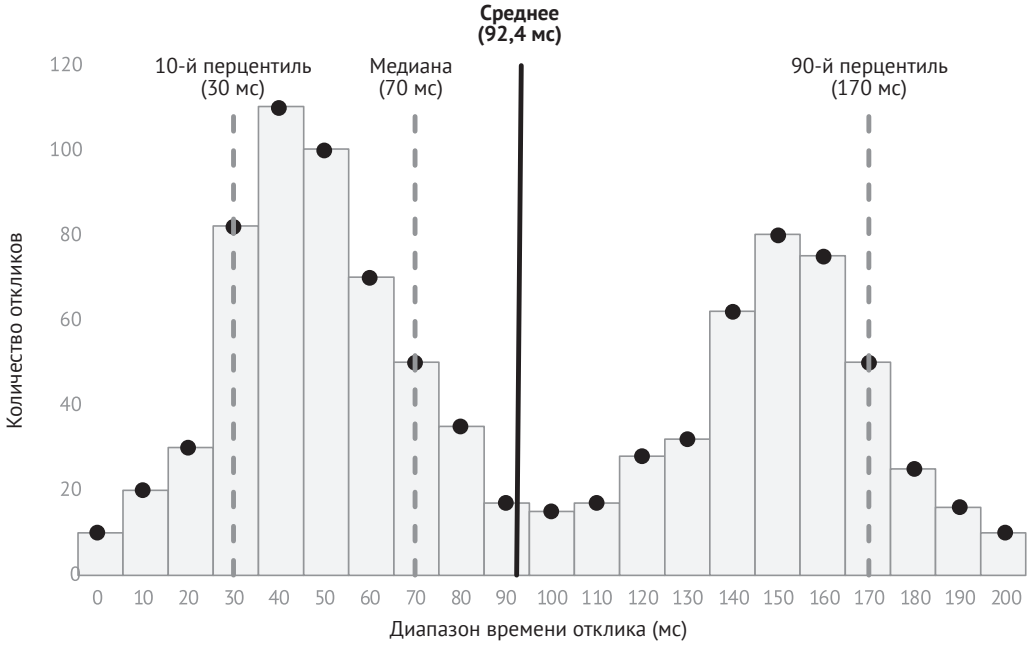


Рис. 3.5 ❖ Пример гистограммы, а также значений медианы, 10-го и 90-го перцентилей. Распределение данных бимодальное

По счастью, многомодальное распределение легко определяется по гистограмме визуально, поэтому так важно собирать подобные данные при измерении чего-либо (или, по крайней мере, получать автоматическое уведомление о том, что обнаружено многомодальное распределение).

Чем больше метрик, помимо среднего, предлагает инструмент, тем лучше. К сожалению, подавляющее большинство инструментов по-прежнему вычисляют только среднее (и лишь немногие показывают еще и гистограммы). Делать какие-либо выводы следует с большой осторожностью. И лучше всего пользоваться инструментом, который дает распределение результатов в виде перцентилей или гистограммы.

Задержка и пропускная способность

Два понятия, вынесенных в заголовок этого раздела, очень важны в контексте анализа производительности и оптимизации. К сожалению, их часто неправильно понимают и интерпретируют. Нередко думают, что одно является следствием другого и что они зависят друг от друга. Поэтому скажем несколько слов в пояснение. Начнем с простых определений.

- *Задержка* – время, необходимое для выполнения данного действия. Измеряется в единицах времени – днях, часах, миллисекундах и т. д.
- *Пропускная способность* – количество действий, выполняемых в единицу времени. Измеряется в действиях (или еще каких-то дискретных элементах) в единицу времени – количество байтов в секунду, количество итераций в миллисекунду или количество книг в год.

Эти показатели связаны между собой простым законом Литтла:

загруженность = задержка * пропускная способность,

где под *загруженностью* понимается количество действий на протяжении времени, равного задержке. Важно, что это уравнение применимо к стабильной системе, в которой нет неестественных очередей или динамической адаптации к изменению нагрузки (например, во время запуска или остановки системы).

Эти два понятия особенно часто встречаются в контексте компьютерных сетей, но нам полезнее контекст веб-приложений. Время обработки одного запроса пользователя определяет задержку, а количество запросов в единицу времени – пропускную способность. Загруженность равна количеству запросов в системе в течение рассматриваемого промежутка времени.

Разумеется, при уменьшении задержки (например, в результате установки более мощного процессора) процесс может обрабатывать больше запросов в единицу времени, поэтому возрастает и пропускная способность. С другой стороны, пропускную способность можно сделать больше, увеличив количество параллельно обрабатываемых запросов (например, установив больше процессорных ядер и т. д.) без изменения задержки (см. рис. 3.6). Вообще, в компьютерных науках считается, что проще увеличить пропускную способность (за счет того или иного вида распараллеливания), чем уменьшить задержку (поставив более производительное оборудование или используя более сложные алгоритмы).

Конечно, невозможно увеличивать пропускную способность до бесконечности. Зачастую по достижении некоторого порога дальнейшее увеличение пропускной

способности негативно отражается на задержке, т. к. эти показатели не совсем независимы. Дополнительные затраты на синхронизацию могут съесть весь выигрыш от повышения пропускной способности.

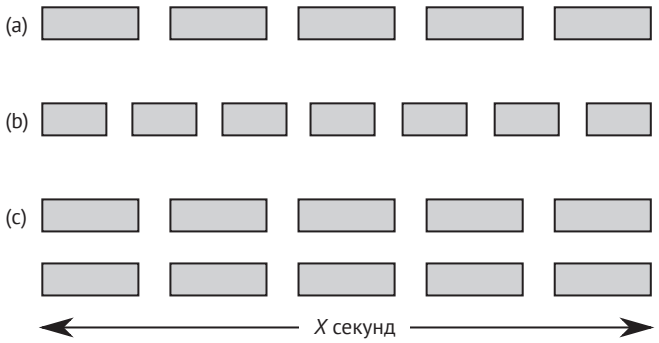


Рис. 3.6 ❖ Связь пропускной способности с задержкой:
(a) базовая задержка позволяет обработать 5 запросов за *X* секунд;
(b) уменьшив задержку, мы сможем обрабатывать 7 запросов за *X* секунд;
(c) удвоив степень параллелизма, мы увеличили пропускную способность до 10 запросов за *X* секунд, не меняя задержку

Имеет место также широко известный закон Амдала, утверждающий, что потенциальное уменьшение задержки ограничено последовательной частью программы (которую невозможно распараллелить). Так что если, например, 90 % программы поддается распараллеливанию, то все равно остается 10 %, с которыми ничего не сделаешь. В таком случае невозможно ускорить работу больше, чем в 10 раз[1].

Дампы памяти, трассировка, динамическая отладка

Для анализа состояния приложения имеется несколько стандартных подходов, различающихся по степени вмешательства.

- *Мониторинг* – обычно имеется в виду невмешивающийся мониторинг приложения и использование порождаемой им диагностической информации (полученной как путем прослеживания, так и выборки). Иногда мониторинг требует большего вмешательства (например, перезагрузки приложения), но все равно позволяет наблюдать за ним в процессе работы, даже в производственной среде.
- *Дамп памяти* – сохранение состояния памяти процесса в данный момент. Чаще всего состояние всей памяти записывается в файл, который затем анализируется различными средствами на другой машине. Такой дамп может занимать несколько гигабайтов, но при наличии навыков позволяет полу-

[1] Заметим, что этот результат относится к приложению в целом, библиотекам, среде выполнения и другим компонентам, а не только к нашему коду. Поэтому если даже в веб-приложении ASP.NET все запросы удастся распараллелить, все равно могут остаться последовательные части, например управление сессиями, части самой среды .NET и веб-сервера, где запущено приложение, а также отдельные участки сборщика мусора.

чить очень подробную информацию о состоянии приложения. С другой стороны, это лишь снимок состояния процесса в данный момент, а без динамики изменения иногда трудно прийти к определенным выводам. Поэтому часто делают два или более дампов памяти и сравнивают их друг с другом. Степень вмешательства при создании дампа памяти зависит от обстоятельств. Обычно процесс приостанавливается на некоторое время. Важное применение дампов памяти – их автоматическое выполнение после сбоя приложения, что позволяет впоследствии изучить его причины (это называется *послеаварийным анализом*). Поэтому иногда встречается термин *аварийный дамп* как частный случай дампа памяти. На практике выражения «аварийный дамп» и «дамп памяти» употребляются в инструментах как синонимы.

- *Динамическая отладка* – наиболее вмешивающийся способ, который заключается в подключении отладчика к процессу и пошаговом выполнении приложения. Он применяется редко, поскольку двух предыдущих способов обычно хватает. Динамическая отладка полностью останавливает приложение, поэтому возможна только во время разработки. Но благодаря развитым средствам мониторинга и диагностики динамическую отладку практически не применяют для решения проблем управления памятью.

Среда Windows

Начнем знакомство с инструментами с той платформы, на которой .NET появилась на свет и существует уже почти 15 лет. Для Windows существует богатый выбор инструментов различной степени развитости. Мы начнем с изучения низкоуровневых инструментов, бесплатных и встроенных в систему, и посвятим им большую часть времени, поскольку будем часто использовать в книге. Но для полноты картины в конце приведем обзор коммерческих программ.

Краткий обзор

Инфраструктура мониторинга и трассировки в Windows весьма зрелая, в т. ч. и для среды .NET. Имеется два основных компонента: *счетчики производительности*, которые дают временной ряд результатов измерений, и событийно-ориентированный механизм *трассировки событий для Windows* (Event Tracing for Windows – ETW). Эти два инструмента почти полностью покрывают все потребности в мониторинге и диагностике. Существует также механизм управления и администрирования Windows (Windows Management Instrumentation), но мы им пользоваться не будем (поскольку, как следует из самого названия, он больше предназначен для управления и администрирования).

При разработке .NET было очевидно, какие диагностические механизмы использовать. И зрелый .NET Framework, и его многоплатформенный вариант .NET Core поддерживают счетчики производительности и ETW. Точнее:

- .NET-приложение может пользоваться классом `EventSource` (из пространства имен `System.Diagnostics.Tracing`), чтобы порождать ETW-события, и, разумеется, к его услугам любые библиотеки, которые протоколируют информацию напрямую в файлы и иные «стоки»;

- .NET Framework порождает счетчики производительности и ETW-события;
- API операционной системы и ядро также порождают счетчики производительности и ETW-события.

Теперь мы подробнее рассмотрим оба этих механизма и их применение в различных инструментах.

VMMap

Этот замечательный инструмент, являющийся частью набора Microsoft Sysinternals, позволяет анализировать память процесса с точки зрения операционной системы. В последующих главах мы воспользуемся им, чтобы показать, как .NET-приложение потребляет память, памятуя о ее организации, описанной в главе 2 (страницы, которые могут быть зарезервированы или переданы для различных целей).

Это автономная программа, не требующая установки, ее можно скачать с сайта https://docs.microsoft.com/en-us/sysinternals/downloads/vmmap. После запуска мы выбираем интересующий нас процесс и сразу видим, как в нем используется память (рис. 3.7). VMMap распознает страницы, применяемые управляемой кучей .NET, а также страницы, предназначенные для стека и загруженных библиотек.

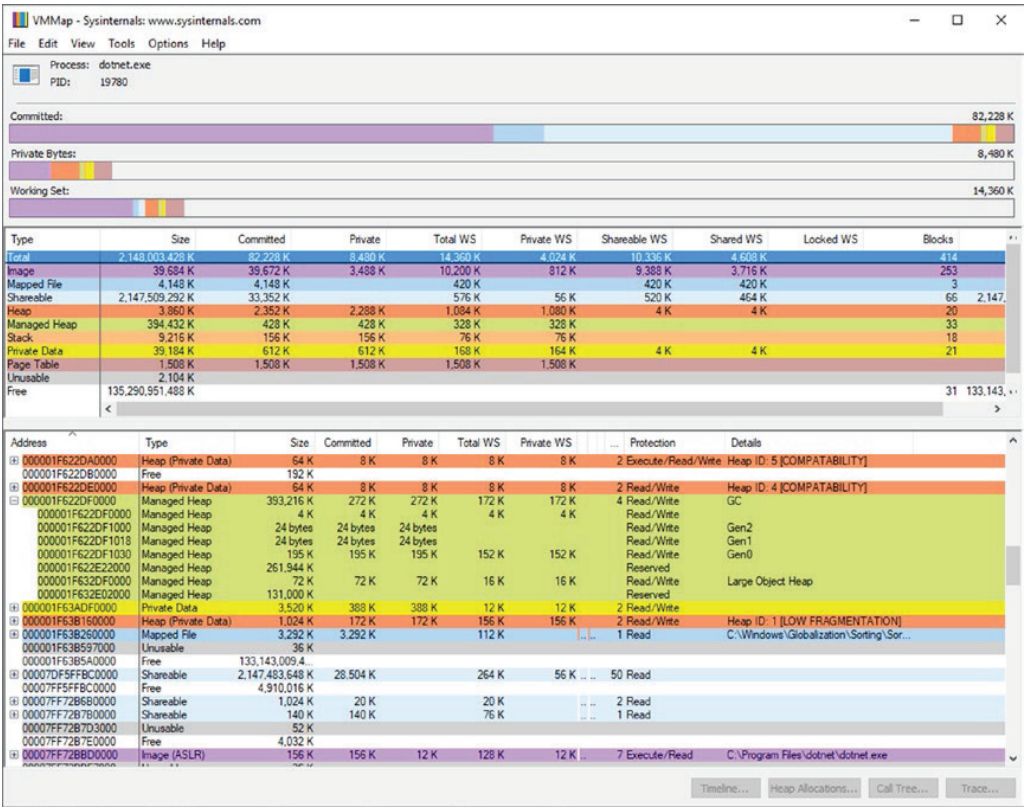


Рис. 3.7 ❖ Пример окна VMMap для простого .NET-приложения (обратите внимание, что правильно определены страницы из управляемых куч)

Счетчики производительности

Один из самых часто используемых инструментов для мониторинга практически всех аспектов работы Windows – *счетчики производительности*. Процессы используют этот механизм для вывода диагностической информации в форме числовых временных рядов. Его огромное достоинство состоит в том, что он вообще не вмешивается в работу приложения и накладные расходы практически незаметны. А недостаток – малая точность; отсчеты генерируются раз в секунду, чего не всегда достаточно.

Существует множество категорий счетчиков, поэтому мы можем получить очень полное представление о работе системы. Общая архитектура счетчиков производительности показана на рис. 3.8.

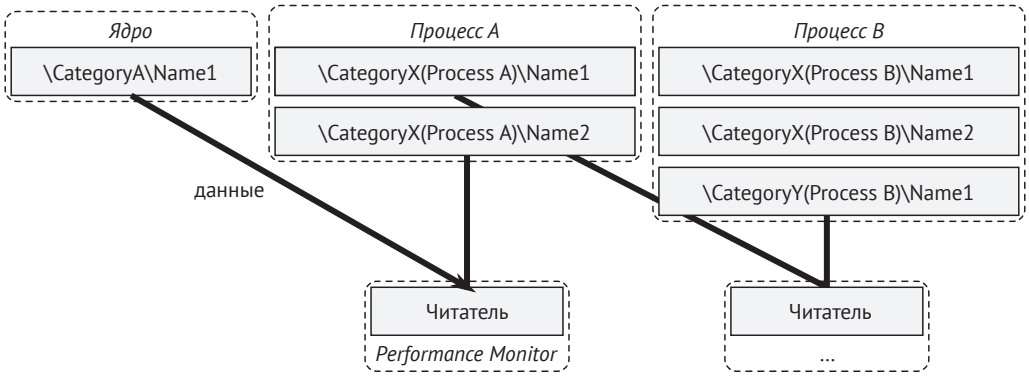


Рис. 3.8 ❖ Архитектура счетчиков производительности

Любой процесс может опубликовать данные в каком-то счетчике производительности, и каждый счетчик могут обновлять несколько процессов. Этот механизм работает в пользовательском пространстве, а не в ядре.

У каждого счетчика производительности имеется несколько важных атрибутов:

- категория – определяет область, к которой относится счетчик;
- имя – однозначно идентифицирует счетчик в данной категории;
- имя экземпляра – в системе может существовать несколько экземпляров одного счетчика. Самый известный пример экземпляра – отдельный процесс.

Уникальный идентификатор счетчика производительности записывается в виде `"\<Категория>(<Экземпляр>)\<Имя>"`. Например, счетчик, сообщающий об использовании ЦП процессом блокнота (`notepad.exe`), обозначается `"\Process(notepad.exe)\% Processor Time"`.

Какие выборочные данные можно получить таким способом? Упомяну лишь немногие, чтобы дать представление о богатстве информации:

- как распределено использование ЦП между ядром и программами (`Processor/% Privileged Time`, `Processor/% User Time`);
- как потребляют ЦП отдельные процессы (`Process/% Processor Time`);
- сколько памяти потребляют отдельные процессы и как именно (`Process/Working Set`, `Process/Working Set - Private`);

- как используется жесткий диск (Process/IO Read Bytes/sec, Process/IO Write Bytes/sec, Process/Page Faults/sec);
- существует ли очередь операций чтения-записи на диск (PhysicalDisk/Current Disk Queue Length);
- сколько исключений генерирует .NET-приложение (.NET CLR Exceptions/# of Exceps Thrown/sec).

Конечно, нас больше всего интересует категория .NET CLR Memory (Память CLR .NET), в которой имеются следующие счетчики (правописание и расстановка заглавных букв сохранены)[1]:

- # Bytes in all Heaps (Байт во всех кучах);
- # GC Handles (Указателей GC);
- # Gen 0 Collections, # Gen 1 Collections, # Gen 2 Collections (Сборов «мусора» для поколения 0, Сборов «мусора» для поколения 1, Сборов «мусора» для поколения 2);
- # Induced GC (Принудительных GC);
- # of Pinned Objects (Закрепленных объектов);
- # of Sink Blocks in use (Используется блоков синхронизации);
- # Total committed Bytes, # Total reserved Bytes (Всего зарезервировано байт, Всего зафиксировано байт);
- % Time in GC (% времени в GC);
- Allocated Bytes/sec (Выделено байт/сек);
- Finalization Survivors (Объектов, оставшихся после сборки мусора);
- Gen 0 heap size, Gen 1 heap size, Gen 2 heap size, Large Object Heap Size (Размер кучи поколения 0, Размер кучи поколения 1, Размер кучи поколения 2, Размер кучи для массивных объектов);
- Gen 0 Promoted Bytes/Sec, Gen 1 Promoted Bytes/Sec (Наследуемых из поколения 0 байт/сек, Наследуемых из поколения 1 байт/сек);
- Process ID (Идентификатор процесса);
- Promoted Finalization-Memory from Gen 0 (Ожидающая выполнения операции Finalize память из поколения 0);
- Promoted Memory from Gen 0, Promoted Memory from Gen 1 (Память, унаследованная из поколения 0, Память, унаследованная из поколения 1).

> **ПРИМЕЧАНИЕ** Названия этих счетчиков производительности (как и счетчиков из других категорий .NET CLR) переведены на язык операционной системы, поэтому на вашем компьютере или сервере они могут называться по-другому. Это ОЧЕНЬ раздражает, потому что в переводах на многие языки их названия звучат, мягко говоря, странно. По этой и по многим другим причинам я рекомендую выбирать английский в качестве языка Windows по умолчанию.

Если вы хоть немного знакомы со сборщиком мусора, то, наверное, догадались о назначении большинства счетчиков. Мы обязательно рассмотрим их ниже. Но уже сейчас скажем, что это полный набор данных, позволяющий очень глубоко проанализировать состояние приложения.

---

[1] Перевод названий счетчиков дан по актуальной программе «Системный монитор» в Windows 10. Он не всегда удачен, не всегда правилен и не всегда совпадает с терминологией, принятой в переводе этой книги. – *Прим. перев.*

Вычисление счетчиков синхронизировано с жизненным циклом сборки мусора. В частности, большая часть измерений производится в начале или в конце сборки мусора. В этом смысле счетчики производительности могут дать очень ценную и точную информацию. Однако следует сделать несколько замечаний.

- Польза счетчиков производительности определяется тем, как часто их читает используемый инструмент. Если он делает выборку достаточно часто (скажем, раз в секунду), то данные будут точны. В противном случае данные будут ошибочными и сбивающими с толку. Например, если чтение производится настолько неудачно, что всегда совпадает с полной сборкой мусора (она потребляет больше всего ресурсов), то мы получим ложное представление о том, какую долю времени занимает GC. Иными словами, при использовании счетчиков производительности следует внимательно смотреть, как осуществляется выборка данных. Лучше всего делать выборку данных как можно чаще.
- Счетчики производительности обновляются только в момент возникновения определенных событий (в основном, как уже было сказано, в начале и в конце сборки мусора), а затем не меняются. Это может ввести в заблуждение. Предположим, к примеру, что в нашем процессе недавно произошла полная сборка мусора, во время которой у счетчика `% время` сборщика мусора было значение 50 %. Начиная с этого момента он так и будет показывать 50 %, даже если наблюдаемый процесс ничего не делает. До начала следующей сборки мусора его значение не изменится. Иными словами, нас должны больше интересовать изменения счетчиков, а не текущие значения. То, что мы видим, – это лишь последнее по времени значение.

Начиная с версии .NET 4.0 Майкрософт предпочитает использовать данные ETW (см. следующий раздел), а не счетчики производительности. Но работать со счетчиками производительности гораздо проще, чем с ETW, отсюда и большая популярность этого механизма. В главе 5 мы подробно рассмотрим различия между измерениями счетчиков производительности и ETW.

У счетчиков производительности может быть много потребителей. Многочисленные средства мониторинга пользуются ими, поскольку это не слишком затратный способ получить большой объем информации. Но один из самых простых и часто применяемых инструментов – встроенный *системный монитор Windows*. Запустите его командой `perfmon.exe` или найдите в меню **Пуск**.

Затем выберите в дереве слева пункт **Performance ➤ Monitoring Tools ➤ Performance Monitor** (Производительность ➤ Средства наблюдения ➤ Системный монитор). В контекстном меню появляющегося графика выберите команду **Add Counters...** (Добавить счетчики) (рис. 3.9).

В диалоговом окне выберите интересующую категорию (в нашем случае `Память CLR .NET`) и конкретные счетчики и их экземпляры (рис. 3.10).

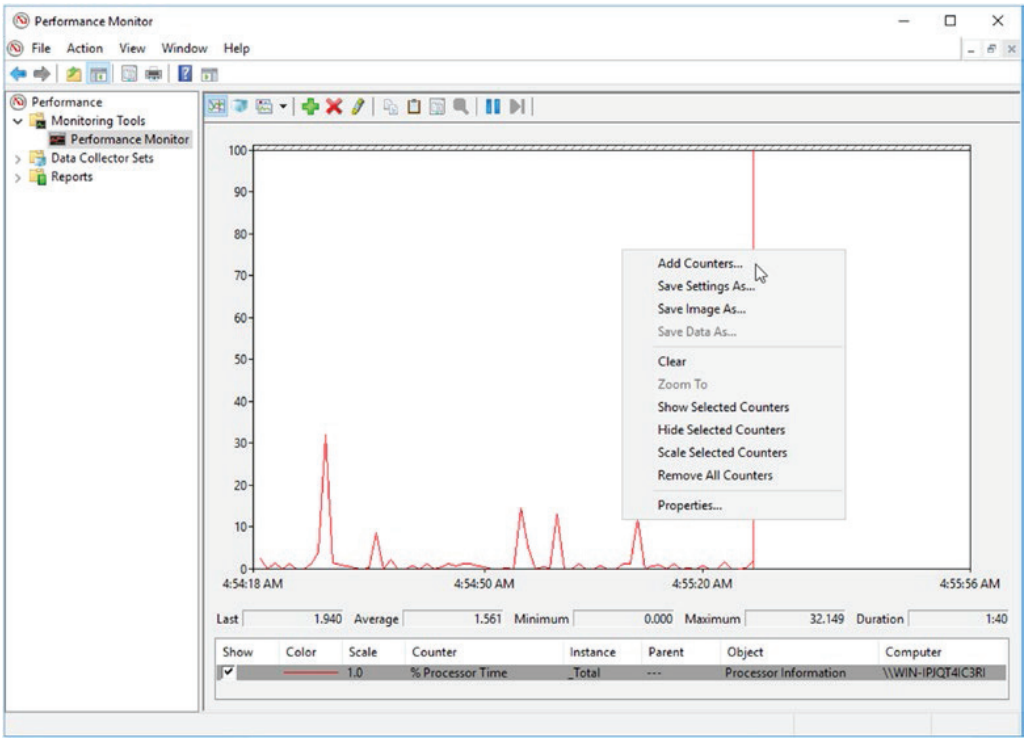


Рис. 3.9 ❖ Системный монитор – общий вид с контекстным меню

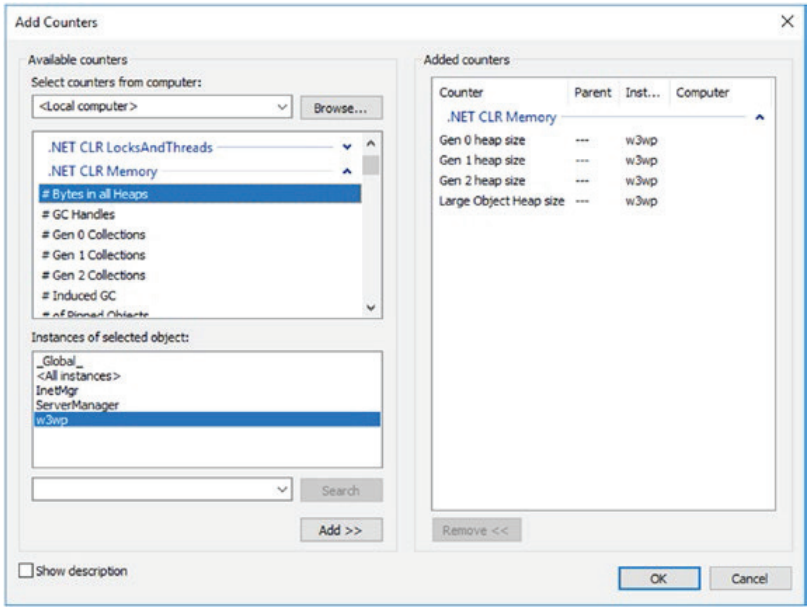


Рис. 3.10 ❖ Системный монитор – диалоговое окно **Add Counters**

После добавления счетчиков имеет смысл адаптировать графики к своим потребностям, а именно задать:

- масштабирование каждого графика (вкладка **Data** [Данные], параметр **Scale** [Масштаб]);
- частоту и количество отсчетов (вкладка **General** [Общие], параметры **Sample every** [Съем показаний каждые] и **Duration** [Длительность]);
- вертикальный масштаб графика (вкладка **Graph** [График], параметры **Vertical scale Minimum** [Диапазон значений вертикальной шкалы Минимум] и **Maximum** [Максимум]);
- способ прокрутки графика (вкладка **Graph** [График], параметр **Scroll style** [Тип прокрутки]).

Задав эти параметры (и, возможно, еще толщину и цвет линии, соответствующей каждому временному ряду), мы можем настроить график для краткосрочного анализа или для наблюдения суточных тенденций. То и другое показано на рис. 3.11 и 3.12.

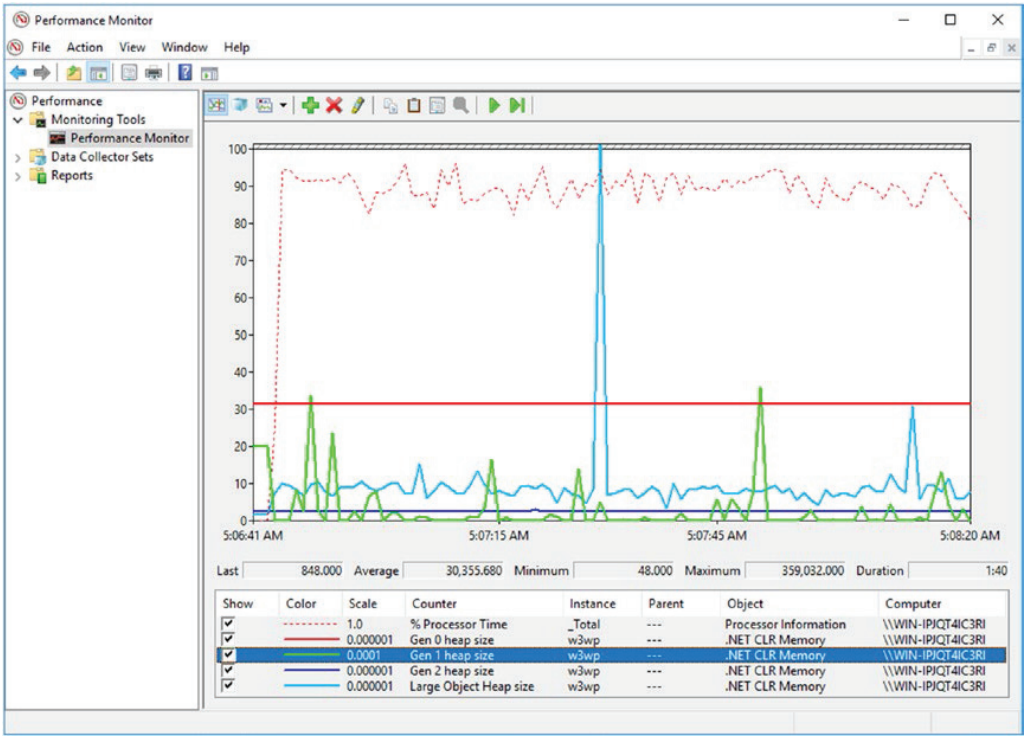


Рис. 3.11 ❖ Системный монитор – краткосрочный анализ (100 секунд), видны размеры поколений GC

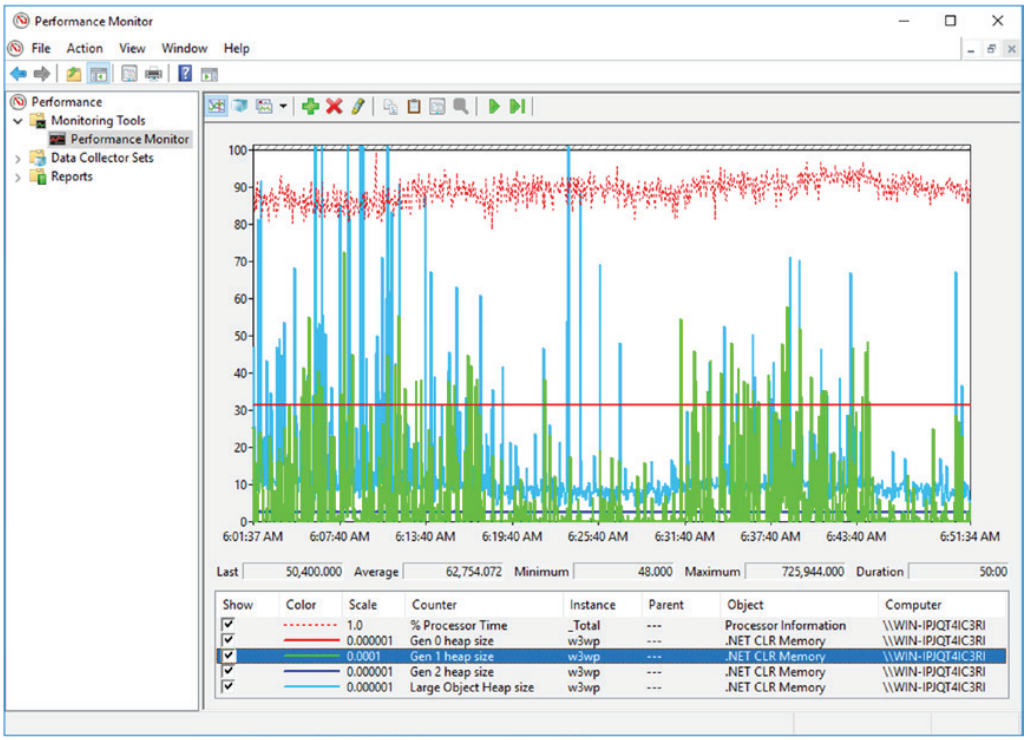


Рис. 3.12 ❖ Системный монитор – долгосрочный анализ (50 минут), видны размеры поколений GC

У механизма счетчиков производительности есть одна раздражающая особенность, с которой придется научиться жить. Я уже говорил, что у каждого процесса, публикующего счетчики с одним именем, имеется уникальное имя экземпляра. Оно соответствует имени процесса. Например, с веб-приложением, размещенным в IIS, будет связан счетчик `\Память CLR .NET (w3wp)\Байт во всех кучах` (поскольку процесс *пула приложений* называется `w3wp.exe`). Но если несколько работающих на сервере приложений размещены в разных пулах, то связанные с ними экземпляры счетчиков будут нумероваться последовательно: `w3wp`, `w3wp#1`, `w3wp#2` и т. д. И как нам выяснить, какой экземпляр какому пулу соответствует? Нам поможет счетчик `Память CLR .NET/Идентификатор процесса`. Благодаря ему мы можем узнать PID процесса каждого экземпляра. Но будьте осторожны! Тут-то и начинаются неприятности – соответствие между процессом и экземпляром счетчика производительности может меняться со временем! Например, если какой-то пул приложений остановлен (например, из-за неактивности), то его счетчик перейдет к одному из оставшихся процессов (см. табл. 3.1).

Таблица 3.1. Динамическое переименование экземпляров пула приложений

До остановки процесса с PID 11200	После остановки процесса с PID 11200
экземпляр w3wp представляет PID 11200	экземпляр w3wp представляет PID 8710
экземпляр w3wp#1 имеет PID 8710	экземпляр w3wp#1 представляет PID 10410
экземпляр w3wp#2 имеет PID 10410	

Это очень раздражает, особенно если вы хотите создать, например, автоматический механизм для наблюдения за конкретными пулами приложений. Тогда важно гарантировать, чтобы таких вещей, как автоматическая остановка пула приложений, вообще не было. С аналогичным механизмом мы имеем дело и тогда, когда в IIS включена возможность перезапуска пула приложений посредством перекрытия. Тогда в какой-то момент у нас будет два экземпляра одного счетчика, поэтому вызывающее проблемы переназначение неизбежно.

Из-за этого неочевидного соответствия наиболее распространенный сценарий ручного наблюдения за размещенными в IIS приложениями таков: проверить текущий PID интересующего нас пула приложения и поискать экземпляр `w3wp`, который имеет соответствующий счетчик `Память CLR .NET/Идентификатор процесса`. Затем добавить счетчики этого конкретного экземпляра.

Вот, собственно, и все, что можно сказать о системном мониторе. Есть много других программ, пользующихся счетчиками производительности, но мы на этом остановимся. Мы будем пользоваться системным монитором для иллюстрации сборки мусора в действии на платформе Windows.

Трассировка событий для Windows

Среди разнообразных диагностических средств, несомненно, одним из самых мощных является механизм *трассировки событий для Windows* (*ETW*). К сожалению, его возможности, похоже, недооценены. Быть может, это связано с тем, что этот механизм развивался постепенно на протяжении нескольких лет и еще не привлек интереса, которого по праву заслуживает. Он появился еще в Windows 2000, и с каждой следующей версией предоставлял все больше и больше возможностей. Его активно разрабатывали в Windows Vista и Windows Server 2003. Так, в Windows 7 появилась возможность записи в журнал стека вызовов при каждом событии (см. https://msdn.microsoft.com/en-us/library/windows/desktop/dd392330).

Сила механизма ETW – в его способности предоставлять огромный объем информации с очень низкими накладными расходами, обычно всего несколько процентов. Поэтому его можно без проблем использовать в промышленной эксплуатации. Его можно включать и выключать, не перезапуская приложение. ETW используется во многих инструментах. Мы, наверное, даже не подозреваем, сколько их на самом деле. Например, этот механизм лежит в основе хорошо известного журнала событий (Event Log) и средства для его просмотра (`eventvwr.exe`), а также монитора ресурсов (`resmon.exe`). Они просто визуализируют события, протоколируемые с помощью ETW. Впрочем, чтобы развеять сомнения, скажу, что счетчики производительности, описанные в предыдущем разделе, не основаны на трассировке событий.

Прежде чем переходить к описанию конкретных инструментов, следует познакомиться с общей архитектурой решения. В механизме ETW имеется несколько понятий, без понимания которых его трудно использовать.

- *Событие ETW* – одно системное событие, которое можно запротоколировать.
- *Сеанс ETW* – главная часть всего механизма. Концептуально, как следует из названия, это продолжающийся сеанс трассировки. Технически же это коллекция системных ресурсов, например буферов в памяти и потоков для записи на диск (рис. 3.13).
- *Поставщик ETW* – любой пользователь или элемент ядра, который может генерировать события. Существует много встроенных в систему поставщиков, они группируются по категориям: сетевые поставщики, процессы и т. д. Сюда входит также среда выполнения .NET и наш код (если мы хотим публиковать свои собственные события ETW). Каждый поставщик имеет глобальный уникальный идентификатор (GUID).
- *Контроллер ETW* – процесс, который отвечает за создание сеанса и подключение его к выбранным поставщикам.
- *Потребитель ETW* – любая программа, которая каким-то образом использует данные событий, например сохраняет их в *журнале трассировки событий* (Event Trace Log – ETL) или выводит в режиме реального времени.

Проектировщики сеанса ETW ставили целью добиться минимальных накладных расходов (рис. 3.13). С точки зрения процесса, это просто быстрое действие, сводящееся к неблокирующей записи в очередь (буфер в памяти), которая обслуживается ядром. И пока приложение продолжает нормально работать, выделенные потоки ядра обрабатывают эти очереди и записывают события по местам назначения, обычно в файл или какой-то другой буфер в памяти (для анализа в режиме реального времени).

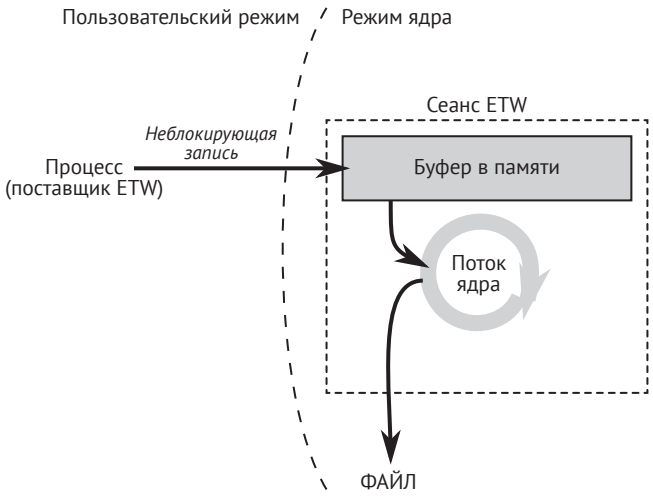


Рис. 3.13 ❖ Трассировка событий для Windows

Концептуально один и тот же поставщик может поставлять информацию нескольким сеансам (рис. 3.14). Наоборот, сеанс может получать информацию от нескольких поставщиков. Характерная особенность ETW – работа на уровне поставщиков, а не процессов. Чтобы получать информацию от одного или нескольких

поставщиков, мы с помощью контроллера создаем новый сеанс, к которому их присоединяем. Как только сеанс начнет работу, все процессы в системе, реализующие функциональность поставщика, начнут доставлять ему события. Поэтому можно сказать, что сеанс собирает события со всей машины, а не от конкретного процесса. Фильтрация данных только от тех процессов, которые нас интересуют, производится на уровне анализа в программе-потребителе.

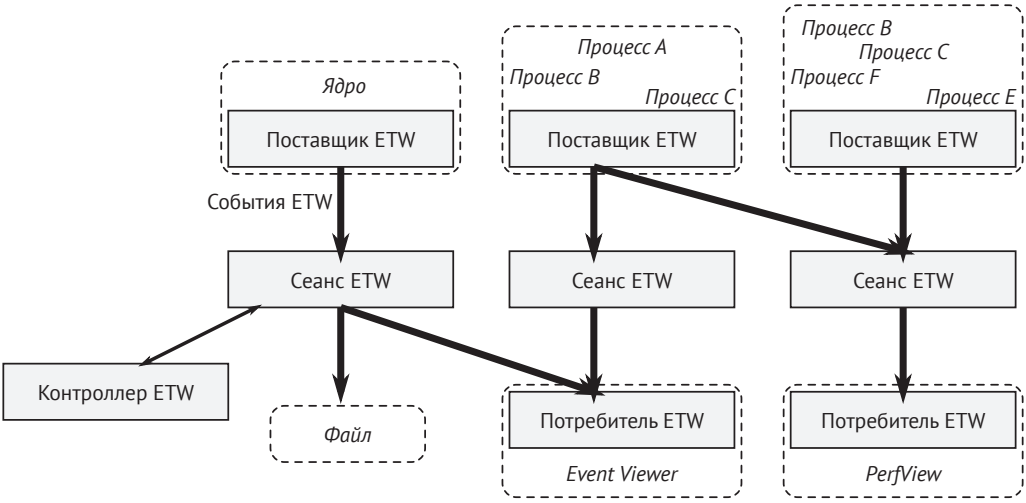


Рис. 3.14 ❖ Структурные элементы трассировки событий для Windows (ETW), иллюстрирующие различные конфигурации. Заметим, что один процесс может выступать в роли нескольких поставщиков ETW, поэтому некоторые процессы встречаются несколько раз

Хранение событий в буферах вне процесса приложения имеет еще одно преимущество – аварийная остановка приложения не приведет к потере диагностических данных. Конечно, когда в журнал записывается много событий, доступ к диску может стать узким местом и нагрузить машину в целом. Однако с такой ситуацией мы сталкиваемся, только когда задаем слишком много интенсивно используемых поставщиков для своего сеанса. Другая опасность – израсходование места на диске, но и тут есть решение. Можно записывать данные в файл в режиме циклического буфера, тогда о переполнении диска можно не волноваться. Данные будут просто размещаться в буфере фиксированного размера и перезаписываться по мере необходимости. Типичный сценарий – запустить сеанс с хранением в циклическом буфере и ждать конкретного события. Затем мы закрываем сеанс и сохраняем данные из буфера в файл.

Начиная с Windows 7 можно сохранять стек вызовов, связанный с событиями ядра и пользователя. В таких специальных событиях полезная нагрузка (помимо самого исходного события) – шестнадцатеричные адреса в кадрах стека, которые декодируются только на этапе анализа. Однако до Windows 8 это можно было делать только для платформенного кода (включая код самой CLR), но не для управляемого. Стек вызовов динамического кода, генерируемого 64-разрядным JIT-компилятором, в этом случае не будет декодироваться (но 32-разрядный код

будет). Эта проблема была исправлена в Windows 8, где подсистему ETW в ядре научили распознавать кадры, созданные 64-разрядным JIT-компилятором, и проходить по ним.

Например, встроенное событие ETW CPU-sampling позволяет отслеживать проблемы с аномально высоким использованием ЦП. К каждому событию (они генерируются каждую миллисекунду) приложены стеки вызовов всех процессов. Это позволяет статистически изучить причину проблемы – узнать, в каких функциях чаще всего находится процессор. Благодаря поддержке поставщиков, находящихся в ОС, можно следить также за проблемами синхронизации (например, взаимоблокировками). Например, эта возможность используется в подключаемом модуле Concurrency Visualizer для Visual Studio.

> При использовании различных диагностических инструментов в Windows часто бывает необходим доступ к файлу отладочных символов (PDB – база данных программы), который позволяет декодировать информацию о методах и функциях в стеке вызовов. Удобнее всего задать адрес общедоступного сервера символов Майкрософт в переменной среды _NT_SYMBOL_PATH:
>
> `srv*C:\Symbols*https://msdl.microsoft.com/download/symbols`
>
> Это даст возможность получать PDB-файлы для операционной системы Windows и библиотек CLR. Кроме того, указывается путь к локальной папке, в которой будут кешироваться загруженные файлы.

Существует специальный сеанс *регистратора ядра NT (NT Kernel Logger)*, которым могут пользоваться только поставщики, работающие в режиме ядра, но не пользователя. Один такой поставщик, например, протоколирует время начала и завершения процесса. Существует пользовательский поставщик Microsoft-Windows-TCPIP, который протоколирует все события от работающего в режиме ядра драйвера tcpip.sys.

Чаще всего вместе с сеансом для поставщиков, работающих в пользовательском режиме, запускается сеанс регистратора ядра NT. Он дает информацию о работающих и завершенных процессах и потоках. На этапе анализа данные объединяются.

Операционная система предоставляет массу интересной информации: об управлении процессами и потоками, о работе сети, об операциях ввода-вывода и т. д. Но для нас интереснее всего тот факт, что CLR тоже является поставщиком ETW, и этот механизм позволяет много узнать о среде выполнения в контексте нашего приложения.

Для нахождения всех связанных с .NET поставщиков в системе можно воспользоваться встроенной утилитой `logman.exe` (листинг 3.1).

Листинг 3.1 ❖ Использование утилиты logman для перечисления всех относящихся к .NET поставщиков ETW

```
> logman query providers | findstr DotNET
Microsoft-Windows-DotNETRuntime          {E13C0D23-CCBC-4E12-931BD9CC2EEE27E4}
Microsoft-Windows-DotNETRuntimeRundown   {A669021C-C450-4609-A035-5AF59AF4DF18}
```

Эта же утилита позволяет узнать, какие поставщики доступны в контексте конкретного процесса. Например, если запросить данные о процессе ASP.NET WebAPI,

размещенном в IIS, то мы получим список, показанный в листинге 3.2 (приведено лишь несколько из многих поставщиков).

Листинг 3.2 ❖ Использование утилиты logman для перечисления всех поставщиков ETW указанного процесса ASP.NET

```
> logman query providers -pid 6228
Provider                        GUID
-------------------------------------------------------------------------------
.NET Common Language Runtime    {E13C0D23-CCBC-4E12-931B-D9CC2EEE27E4}
ASP.NET Events                  {AFF081FE-0247-4275-9C4E-021F3DC1DA35}
IIS: WWW Global                 {D55D3BC9-CBA9-44DF-827E-132D3A4596C2}
IIS: WWW Isapi Extension        {A1C2040E-8840-4C31-BA11-9871031A19EA}
IIS: WWW Server                 {3A2A4E84-4C21-4981-AE10-3FDA0D9B0F83}
Microsoft-Windows-Application   {C651F5F6-1C0D-492E-8AE1-B4EFD7C9D503}
Server-Applications
Microsoft-Windows-Application-  {EEF54E71-0661-422D-9A98-82FD4940B820}
Experience
Microsoft-Windows-              {A669021C-C450-4609-A035-5AF59AF4DF18}
DotNETRuntimeRundown
Microsoft-Windows-IIS           {DE4649C9-15E8-4FEA-9D85-1CDDA520C334}
Microsoft-Windows-IIS-          {DC0B8E51-4863-407A-BC3C-1B479B2978AC}
Configuration
...
```

Если запросить данные о консольном приложении, работающем в среде CoreCLR, то будет выдан несколько иной список поставщиков (листинг 3.3).

Листинг 3.3 ❖ Использование утилиты logman для перечисления всех поставщиков ETW консольного процесса .NET Core

```
> logman query providers -pid 8528
Provider                                GUID
-------------------------------------------------------------------------------
.NET Common Language Runtime            {E13C0D23-CCBC-4E12-931B-D9CC2EEE27E4}
Microsoft-Windows-AsynchronousCausality {19A4C69A-28EB-4D4B-8D94-5F19055A1B5C}
Microsoft-Windows-COM-Perf              {B8D6861B-D20F-4EEC-BBAE-87E0DD80602B}
Microsoft-Windows-Crypto-BCrypt         {C7E089AC-BA2A-11E0-9AF7-68384824019B}
Microsoft-Windows-Crypto-RSAEnh         {152FDB2B-6E9D-4B60-B317-815D5F174C4A}
Microsoft-Windows-DotNETRuntimeRundown  {A669021C-C450-4609-A035-5AF59AF4DF18}
Microsoft-Windows-Networking-Correlation {83ED54F0-4D48-4E45-B16E-726FFD1FA4AF}
Microsoft-Windows-Shell-Core            {30336ED4-E327-447C-9DE0-51B652C86108}
Microsoft-Windows-User-Diagnostic       {305FC87B-002A-5E26-D297-60223012CA9C}
Microsoft-Windows-WinRT-Error           {A86F8471-C31D-4FBC-A035-665D06047B03}
{012616AB-FF6D-4503-A6F0-EFFD0523ACE6}  {012616AB-FF6D-4503-A6F0-EFFD0523ACE6}
{05F95EFE-7F75-49C7-A994-60A55CC09571}  {05F95EFE-7F75-49C7-A994-60A55CC09571}
...
```

Как видим, помимо многих других, в списке присутствуют поставщики, связанные с .NET. У них одинаковые GUID как для WebAPI .NET, так и для консольного приложения CoreCLR. Заметим также, что один и тот же поставщик называется по-разному: `Microsoft-Windows-DotNETRuntime` и `.NET Common Language Runtime`.

У каждого события ETW, порожденного данным поставщиком, есть несколько важных атрибутов:

- `Id` – уникальный идентификатор события;
- `Version` – версия события;
- `Keyword` – это поле является битовой маской, поэтому его можно использовать для соотнесения события с одной или несколькими категориями (ключевыми словами);
- `Level` – уровень протоколирования;
- `Opcode` – обозначает конкретное действие (этап) внутри данного события. Чаще всего используются встроенные значения `Start` и `End`;
- `Task` – используется для группировки событий одного поставщика по функциональности.

Программа `logman` позволяет получить подробные сведения о поставщике. Информация о главном поставщике ETW для .NET приведена в листинге 3.4.

Листинг 3.4 ❖ Получение подробных сведений о поставщике ETW для .NET

```
> logman query providers ".NET Common Language Runtime"
Provider                   GUID
-------------------------------------------------------------------------------
.NET Common Language Runtime {E13C0D23-CCBC-4E12-931BD9CC2EEE27E4}

Value              Keyword                  Description
-------------------------------------------------------------------------------
0x0000000000000001 GCKeyword                GC
0x0000000000000002 GCHandleKeyword          GCHandle
0x0000000000000004 FusionKeyword            Binder
0x0000000000000008 LoaderKeyword            Loader
0x0000000000000010 JitKeyword               Jit
0x0000000000000020 NGenKeyword              NGen
0x0000000000000040 StartEnumerationKeyword  StartEnumeration
0x0000000000000080 EndEnumerationKeyword    StopEnumeration
0x0000000000000400 SecurityKeyword          Security
0x0000000000000800 AppDomainResourceManagementKeyword
                                            AppDomainResourceManagement
0x0000000000001000 JitTracingKeyword        JitTracing
0x0000000000002000 InteropKeyword           Interop
0x0000000000004000 ContentionKeyword        Contention
0x0000000000008000 ExceptionKeyword         Exception
0x0000000000010000 ThreadingKeyword         Threading
0x0000000000020000 JittedMethodILToNativeMapKeyword
                                            JittedMethodILToNativeMap
0x0000000000040000 OverrideAndSuppressNGenEventsKeyword
                                            OverrideAndSuppressNGenEvents
0x0000000000080000 TypeKeyword              Type
0x0000000000100000 GCHeapDumpKeyword        GCHeapDump
0x0000000000200000 GCSampledObjectAllocationHighKeyword
                                            GCSampledObjectAllocationHigh
0x0000000000400000 GCHeapSurvivalAndMovementKeyword
                                            GCHeapSurvivalAndMovement
0x0000000000800000 GCHeapCollectKeyword     GCHeapCollect
0x0000000001000000 GCHeapAndTypeNamesKeyword  GCHeapAndTypeNames
0x0000000002000000 GCSampledObjectAllocationLowKeyword
                                            GCSampledObjectAllocationLow
```

```
0x0000000020000000 PerfTrackKeyword          PerfTrack
0x0000000040000000 StackKeyword              Stack
0x0000000080000000 ThreadTransferKeyword     ThreadTransfer
0x0000000100000000 DebuggerKeyword           Debugger
0x0000000200000000 MonitoringKeyword         Monitoring

Value              Level                     Description
------------------------------------------------------------------------------
0x00               win:LogAlways             Log Always
0x02               win:Error                 Error
0x04               win:Informational         Information
0x05               win:Verbose               Verbose
...
```

Полный список событий, генерируемых поставщиками .NET, можно посмотреть в документации MSDN по адресу https://msdn.microsoft.com/en-us/library/dd264810(v=vs.110).aspx. Однако он не всегда актуален. Поэтому лучше всего обратиться к первоисточнику, каковым является *файл манифеста* поставщика. В этом файле определена строго типизированная информация о событиях, генерируемых данным поставщиком, что позволяет потребителю корректно интерпретировать записанные данные. Для каждой среды выполнения .NET манифесты различны и находятся в разных папках:

- для CoreCLR в папке `.\coreclr\src\vm\ClrEtwAll.man`;
- для .NET Framework версии 4.0 и более поздних версий в папке `c:\Windows\Microsoft.NET\Framework64\v4.0.30319\CLR-ETW.man`;
- для .NET Framework версии 2.0 и более ранних версий файлов манифестов нет, поскольку тогда ETW еще не поддерживалась.

В манифесте содержится полная информация о поставщиках `Microsoft-Windows-DotNETRuntime` и `Microsoft-Windows-DotNETRuntimeRundown`. Его фрагменты приведены в листинге 3.5.

Листинг 3.5 ❖ Фрагменты файла манифеста поставщиков ETW для .NET

```
<instrumentationManifest xmlns="http://schemas.microsoft.com/win/2004/08/events">
  <instrumentation xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:win="http://
manifests.microsoft.com/win/2004/08/windows/events">
   <events xmlns="http://schemas.microsoft.com/win/2004/08/events">
     <!--CLR Runtime Publisher-->
     <provider name="Microsoft-Windows-DotNETRuntime" guid="{e13c0d23-ccbc-
4e12-931b-d9cc2eee27e4}" symbol="MICROSOFT_WINDOWS_
DOTNETRUNTIME_PROVIDER" resourceFileName="%WINDIR%\Microsoft.NET\
Framework64\v4.0.30319\clretwrc.dll" messageFileName="%WINDIR%\
Microsoft.NET\Framework64\v4.0.30319\clretwrc.dll">
      <!--Keywords-->
      <keywords>
       <keyword name="GCKeyword" mask="0x1" message="$(string.
RuntimePublisher.GCKeywordMessage)" symbol="CLR_GC_KEYWORD"/>
       <keyword name="GCHandleKeyword" mask="0x2" message="$(string.
RuntimePublisher.GCHandleKeywordMessage)" symbol="CLR_GCHANDLE_KEYWORD"/>
       ...
      </keywords>
```

```
      <!--Tasks-->
      <tasks>
        <task name="GarbageCollection" symbol="CLR_GC_TASK" value="1"
eventGUID="{044973cd-251f-4dff-a3e9-9d6307286b05}"
message="$(string.RuntimePublisher.GarbageCollectionTaskMessage)">
          <opcodes>
            <!-- These opcode use to be 4 through 9 but we added 128 to
                 them to avoid using the reserved range 0-10 -->
            <opcode name="GCRestartEEEnd" message="$(string.
RuntimePublisher.GCRestartEEEndOpcodeMessage)" symbol="CLR_
GC_RESTARTEEEND_OPCODE" value="132"> </opcode>
            <opcode name="GCHeapStats" message="$(string.
RuntimePublisher.GCHeapStatsOpcodeMessage)" symbol="CLR_GC_
HEAPSTATS_OPCODE" value="133"> </opcode>
            ...
          </opcodes>
        </task>
        <task name="WorkerThreadCreation" symbol="CLR_
WORKERTHREADCREATE_TASK" value="2" eventGUID="{cfc4ba53-fb42-
4757-8b70-5f5d51fee2f4}" message="$(string.RuntimePublisher.
WorkerThreadCreationTaskMessage)">
          <opcodes>
          </opcodes>
        </task>
        ...
      </tasks>
      <!--Maps-->
      <maps>
        <!-- ValueMaps -->
        <valueMap name="GCSegmentTypeMap">
          <map value="0x0" message="$(string.RuntimePublisher.GCSegment.
SmallObjectHeapMapMessage)"/>
          <map value="0x1" message="$(string.RuntimePublisher.GCSegment.
LargeObjectHeapMapMessage)"/>
          <map value="0x2" message="$(string.RuntimePublisher.GCSegment.
ReadOnlyHeapMapMessage)"/>
        </valueMap>
        ...
      </maps>
      <!--Templates-->
      <templates>
        <template tid="GCStart">
          <data name="Count" inType="win:UInt32" outType="xs:unsignedInt"/>
          <data name="Reason" inType="win:UInt32" map="GCReasonMap"/>
          <UserData>
            <GCStart xmlns="myNs">
              <Count> %1 </Count>
              <Reason> %2 </Reason>
            </GCStart>
          </UserData>
        </template>
        ...
```

```
  </templates>
  <events>
    <!-- CLR GC events, value reserved from 0 to 39 and 200 to 239 -->
    <!-- Note the opcode's for GC events do include 0 to 9 for
         backward compatibility, even though they don't mean what those
         predefined opcodes are supposed to mean -->
    <event value="1" version="0" level="win:Informational"
      template="GCStart" keywords="GCKeyword" opcode="win:Start"
      task="GarbageCollection" symbol="GCStart" message="$(string.
      RuntimePublisher.GCStartEventMessage)"/>
    <event value="1" version="1" level="win:Informational"
      template="GCStart_V1" keywords="GCKeyword" opcode="win:Start"
      task="GarbageCollection" symbol="GCStart_V1" message="$(string.
      RuntimePublisher.GCStart_V1EventMessage)"/>
    ...
  </events>
</provider>
```

Как видим, для желающих использовать ETW в контексте .NET манифест является кладезем знаний. Бросим беглый взгляд на события, генерируемые обоими поставщиками. Мы будем возвращаться к ним в следующих главах, так что у вас будет возможность хорошенько познакомиться с каждым. А сейчас уделим внимание только самым интересным. Это позволит оценить, насколько глубока информация, получаемая от механизма ETW.

> Если смотреть только на сами генерируемые события, то возникают интересные вопросы. Например, что это за сегмент `ReadOnlyHeapMapMessage` типа `GCSegmentTypeMap`? Мы ответим на этот вопрос в главе 5.

Нас больше всего интересует поставщик `Microsoft-Windows-DotNETRuntime`, который предлагает события, сгруппированные в 29 задач (Task) (в терминологии ETW атрибут `task` события определяет его функциональную категорию). Чтобы получить представление о богатстве предоставляемой информации, просто перечислим эти задачи (в скобках указано количество событий в данной задаче): `AppDomainResourceManagement` (5), `CLRAuthenticodeVerification` (4), `CLRILStub` (2), `CLRLoader` (18), `CLRMethod` (25), `CLRPerfTrack` (1), `CLRRuntimeInformation` (1), `CLRStack` (1), `CLRStrongNameVerification` (4), `Contention` (3), `Exception` (3), `ExceptionCatch` (2), `ExceptionFilter` (2), `ExceptionFinally` (2), `GarbageCollection` (58), `IOThreadCreation` (4), `IOThreadRetirement` (4), `Thread` (2), `ThreadPool` (5), `ThreadPoolWorkerThread` (3), `Type` (1).

Как видите, больше всего событий в задаче Garbage Collector – целых 58! На самом деле различных только 44, потому что для некоторых имеется несколько версий. Что же тут находится? Очень интересные вещи! В табл. 3.2 перечислено несколько избранных событий с описанием.

Если учесть, что у каждого события имеется точная временная метка и может присутствовать стек вызовов, то открываются заманчивые перспективы создания мощных средств диагностики. Потому-то события и используются во многих инструментах. Некоторые из них будут описаны ниже.

Не переживайте, если не понимаете описания событий ETW, приведенных в табл. 3.2. Понятно, что для этого нужно некоторое знакомство с GC. В следующих главах мы еще вернемся ко многим событиям ETW (в т. ч. и к перечисленным в табл. 3.2).

Таблица 3.2. Пример событий ETW, относящихся к сборке мусора

Событие	Данные
GCStart_V2	ClientSequenceNumber(win:UInt64), ClrInstanceID(win:UInt16), Count(win:UInt32), Depth(win:UInt32), Reason(GCReasonMap), Type(GCTypeMap) Информирует о начале сборки мусора, сообщая причину и номер инициировавшего это событие поколения (в поле Depth)
GCEnd_V1	ClrInstanceID(win:UInt16), Count(win:UInt32), Depth(win:UInt32) Информирует об окончании сборки мусора
GCCreateSegment_V1	Address(win:UInt64), ClrInstanceID(win:UInt16), Size(win:UInt64), Type(GCSegmentTypeMap) Информирует о создании нового сегмента памяти, сообщая его размер и тип
GCSuspendEEBegin_V1	ClrInstanceID(win:UInt16), Count(win:UInt32), Reason(GCSuspendEEReasonMap) Информирует о начале процесса приостановки среды выполнения, что необходимо некоторым частям сборщика мусора
GCSuspendEEEnd_V1	ClrInstanceID(win:UInt16) Информирует об окончании процесса приостановки среды выполнения. Начиная с этого момента большинство потоков приостановлены
GCAllocationTick_V3	Address(win:Pointer), AllocationAmount(win:UInt32), AllocationAmount64(win:UInt64), AllocationKind(GCAllocationKindMap), ClrInstanceID(win:UInt16), HeapIndex(win:UInt32), TypeID(win:Pointer), TypeName(win:UnicodeString) Очень интересное событие периодической выборки (генерируется после выделения каждых 100 КБ). Информирует о статистике выделения памяти
GCHeapStats_V1	ClrInstanceID(win:UInt16), FinalizationPromotedCou nt(win:UInt64), FinalizationPromotedSize(win:UInt64), GCHandleCount(win:UInt32), GenerationSize0(win:UInt64), GenerationSize1(win:UInt64), GenerationSize2(win:UInt64), GenerationSize3(win:UInt64), PinnedObjectCount(win:UInt32), SinkBlockCount(win:UInt32), TotalPromotedSize0(win:UInt64), TotalPromotedSize1(win:UInt64), TotalPromotedSize2(win:UInt64), TotalPromotedSize3(win:UInt64) Еще одно очень интересное событие – содержит много информации о статистике кучи вообще, включая размеры поколений

Сеанс регистратора ядра NT также предоставляет немало ценной информации, включая такие события, как Windows Kernel\ProcessStart, Windows Kernel\ProcessEnd – когда начинается и завершается процесс, Windows Kernel\ImageLoad – когда загружается динамическая библиотека, Windows Kernel\TcpIpRecv – когда принимаются пакеты TCP/IP, Windows Kernel\ThreadCSwitch – когда поток получает и теряет доступ к ЦП. Есть, конечно, и много других, но перечислять их здесь не имеет особого смысла. За дополнительной информацией обратитесь к документации по трассировочному сеансу регистратора ядра NT на сайте MSDN.

Windows Performance Toolkit

Windows Performance Toolkit (WPT) – набор диагностических средств для Windows. Нас в первую очередь интересуют их возможности по сбору и анализу данных ETW. До Windows 8 основным инструментом для этой цели была довольно неудобная программа xperf. Она и до сих пор входит в набор файлов, устанавливаемых в составе WPT. Раньше она использовалась для подготовки и запуска сеансов ETW и их последующего анализа. Таким образом, в терминологии ETW она являлась одновременно контроллером и потребителем. Упоминания о ней часто можно встретить в старых статьях и блогах, посвященных ETW. Благодаря своей выдающейся гибкости этот инструмент все еще иногда применяется для управления сеансами ETW из командной строки. Но в Windows 8 в Windows Performance Toolkit появилось два новых инструмента:

- Windows Performance Recorder – играет роль контроллера ETW;
- Windows Performance Analyzer – играет роль потребителя ETW.

Эти программы, входящие в Windows Performance Toolkit, сейчас наиболее употребимы. Мы вкратце рассмотрим, как с ними работать.

> **ПРИМЕЧАНИЕ** Windows Performance Toolkit можно установить двумя способами – в составе одного из двух больших пакетов: Windows Assessment and Deployment Kit или Windows SDK.

Windows Performance Recorder

С точки зрения пользователя, Windows Performance Recorder – простое диалоговое окно, работающее в роли контроллера ETW (рис. 3.15). Какие события протоколировать и от каких поставщиков, задается в профилях. На рис. 3.15 показано много встроенных профилей, устанавливаемых вместе с программой.

Для настройки доступно два основных параметра:

- уровень детализации записываемых данных. Нас больше всего интересует уровень Verbose, при котором, помимо времени возникновения события, включается дополнительная диагностическая информация;
- режим протоколирования – чаще всего мы будем использовать режим Memory, когда события записываются во временный циклический буфер в памяти. Тем самым гарантируется, что размер буфера никогда не будет превышен и что создание очень больших файлов или буферов в памяти не окажет заметного влияния на работу операционной системы и других приложений.

Что точно включено в профиль, пользовательский интерфейс не показывает. Но это можно увидеть в версии программы для командной строки. Для получения списка встроенных профилей, отображаемого в графическом интерфейсе, выполните команду с флагом `profiles` (листинг 3.6).

Листинг 3.6 ❖ Получение имен всех профилей в командной версии `wpr`

```
> wpr -profiles
```

Затем с помощью команды `profiledetails` можно запросить подробные сведения о конкретном профиле. Это позволяет, например, узнать, какие поставщики и ключевые слова включены в профиле .NET Activity (листинг 3.7).

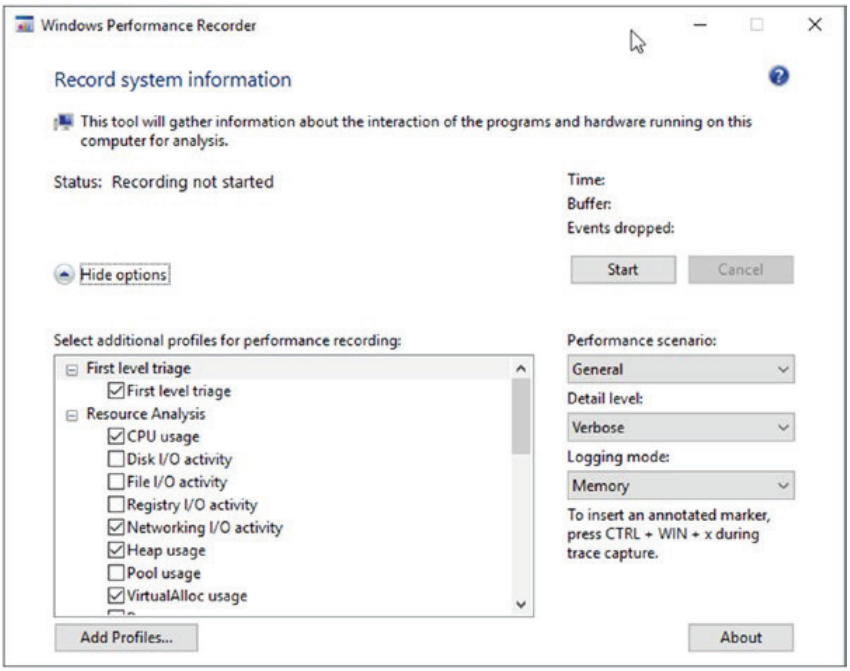


Рис. 3.15 ❖ Диалоговое окно Windows Performance Recorder

Листинг 3.7 ❖ Получение конфигурации заданного профиля в командной версии `wpr` (некоторые поставщики, для которых указан только Guid, для краткости опущены)

```
> wpr -profiledetails DotNet
System Keywords: CSwitch, DiskIO, DiskIOInit, HardFaults, Loader,
MemoryInfo, MemoryInfoWS, NetworkTrace, ProcessCounter, ProcessThread,
SampledProfile
System Stacks: CSwitch, DiskFlushInit, DiskReadInit, DiskWriteInit,
FileCreate, FileRead, FileWrite, ImageLoad, ImageUnload, ProcessCreate,
SampledProfile, ReadyThread
Providers
...
Microsoft-Windows-DotNETRuntime: 0x4007ccbd: 0x05
Microsoft-Windows-IIS: : 0xffI
```

Для среды выполнения .NET выбираемая поставщиком маска ключевых слов соответствует значению `0x4007ccbd`. Чтобы узнать, какие ключевые слова ей соответствуют, можно воспользоваться значениями в листинге 3.4. Легко видеть, что выбраны не все возможные ключевые слова (в т. ч. не включены некоторые слова, относящиеся к сборщику мусора).

Есть также встроенные профили Windows Heap и VirtualAllocations. Чтобы получить полную картину во время анализа работы CLR, имеет смысл выбрать все три профиля.

Кнопка **Add profile** (Добавить профиль) позволяет вручную определить новый профиль. Это единственный способ подключиться только к интересующим нас

поставщикам и точно настроить набор ключевых слов. Пример профиля «Pro .NET Memory Management with stacks» имеется в сопроводительном репозитории на GitHub (файл NetMemoryManagement.wprp). В нем разрешены все события .NET вместе с записью стеков вызова (но имейте в виду, что в такой конфигурации трассировка замедляет работу .NET-приложений, в основном из-за сбора стеков).

Windows Performance Analyzer

Windows Performance Analyzer – это мощный потребитель ETW: в нем можно производить очень сложные виды анализа. В то же время это один из основных инструментов для удобной визуализации данных ETW. Первая встреча с ним может немного ошеломить. Интерфейс универсальный, и как его адаптировать, решать пользователю. Поэтому приступить к работе трудновато, и с первого взгляда не оценить всю мощь инструмента.

Полное описание работы с интерфейсом Windows Performance Analyzer выходит за рамки этой книги. Возможностей так много, что для этого потребовалась бы отдельная небольшая книга. Мы же сосредоточимся на некоторых сценариях, наиболее полезных для наших целей. Для примера будет использоваться программа нагрузочного тестирования с открытым исходным кодом SuperBenchmarker, написанная для .NET 4.5 и доступная на GitHub по адресу https://github.com/aliostad/SuperBenchmarker. В процессе тестирования она генерирует систематическую нагрузку на целевое веб-приложение, поэтому хорошо подходит для экспериментов. К книге прилагается файл `WPA-Tutorial.zip`, содержащий пример сценария `WPA-Tutorial.ETL`, записанного в ходе тестирования с такими параметрами:

```
.\sb.exe -u http://localhost/LeakWebApi/values/concatenated/100 -c 10 -n 100000
-y 100
```

Это означает, что всего производится 100 000 вызовов в 10 параллельных потоках с промежутком между вызовами 100 миллисекунд. Приложение LeakWebApi – очень простой проект типа ASP.NET MVC Web API, размещенный в IIS. В силу самой природы ETW записываются также события от многих других процессов, но нас будут интересовать только два: сам `sb.exe` и процесс `w3wp.exe`, в котором размещено приложение. Файл был создан программой Windows Performance Recorder с включенными профилями `CPU usage`, `Heap usage`, `VirtualAlloc usage` и добавленным нами профилем «.NET Memory Management with stacks». Если вы хотите самостоятельно проделать описанные ниже упражнения, распакуйте файл WPA-Tutorial.zip в какой-нибудь каталог.

Теперь проработаем несколько возможных сценариев работы с Windows Performance Analyzer. Не забывайте о выдающейся гибкости этого инструмента. Если при выполнении описанных ниже упражнений какие-то результаты будут выглядеть иначе, чем на снимках экрана, то проверьте настройки просмотра, в особенности видимость и порядок столбцов.

ОТКРЫТИЕ ФАЙЛА И НАСТРОЙКА После запуска программы появится пустое окно с вкладкой **Getting Started** (Приступая к работе). Откройте записанный файл, выбрав из меню команду **File ➤ Open...** (Файл ➤Открыть).

После открытия файла слева появляется панель Graph Explorer, на которой представлено несколько групп графиков – в зависимости от того, какие данные записывались. Для нашего файла `WPA-Tutorial` должно быть пять групп:

- System Activity – данные общего характера о работе системы, процессах и потоках. Существует также очень важная диаграмма Generic Events, которую мы рассмотрим ниже;
- Computation – данные, относящиеся к работе ЦП;
- Storage – данные, относящиеся к дискам, в т. ч. такие специфические, как использованные смещения на дисках;
- Memory – данные, относящиеся к памяти;
- Power – данные, относящиеся к энергопотреблению, в т. ч. частота и состояния процессоров.

Рядом с названием каждой группы есть кнопка раскрытия, которая позволяет перемещаться по графикам, входящим в группу. Любой из видимых графиков можно перенести на вкладку **Analysis**, дважды щелкнув по нему или перетащив мышью. К графику можно добавить много различных данных, которые будут помещены друг под другом. Все элементы, добавленные на вкладку **Analysis**, синхронизированы (равно как и сам Graph Explorer). Так, например, если изменить масштаб временной шкалы для одного из них, то изменение отразится и на всех остальных. То же самое справедливо для любого рода фильтрации или акцентирования исследуемых данных.

Теперь создадим первое представление, которое поможет нам изучить основы работы с программой на практике. На панели **Graph Explorer** раскройте группу **System Activity**. Перетащите в рабочую область график **Processes** (или дважды щелкните по нему мышью). Он появится на вкладке **Analysis**. Затем раскройте группу **Computation** и дважды щелкните по графику **CPU Usage (Sampled)**. Он должен появиться ниже ранее добавленного. Ожидаемый результат показан на рис. 3.16.

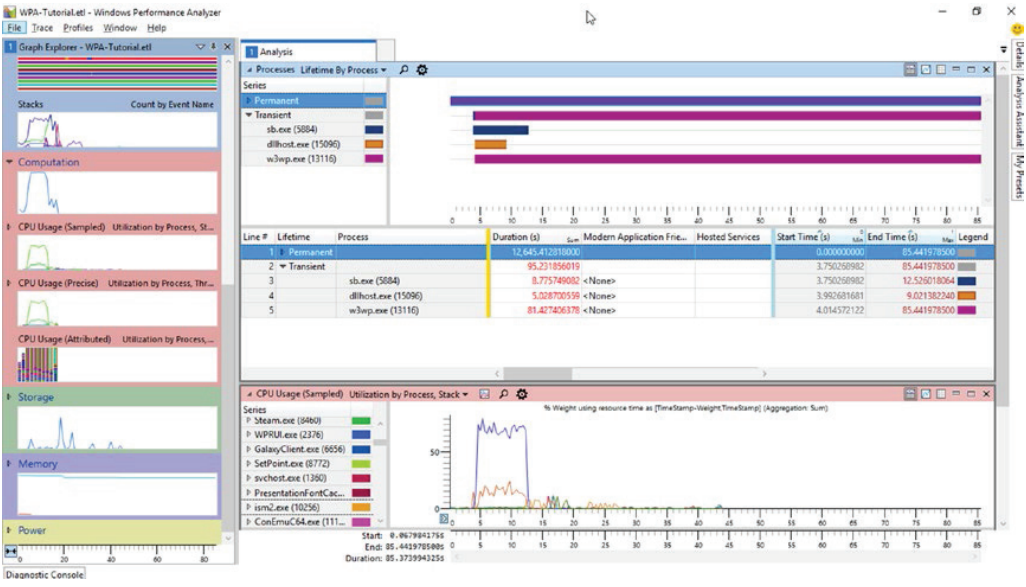


Рис. 3.16 ❖ Пример представления, содержащего панели **Processes** и **CPU Usage**

Довольно быстро мы обнаружим, что со многими элементами ассоциированы всплывающие подсказки, содержащие дополнительную информацию. На панели **Processes** показано, какие процессы работали во время записи. Легко найти блок, соответствующий процессу `sb.exe`. Щелкните по нему левой кнопкой мыши. Временной диапазон этого процесса автоматически будет подсвечен на всех остальных графиках. Это очень полезно для навигации и перекрестных ссылок одних данных на другие.

Иногда данные удобнее анализировать в графической, а иногда в табличной форме. Поэтому в правом верхнем углу любой панели имеются три кнопки: показать только график, показать только таблицу, показать то и другое (по умолчанию выбран третий вариант – **Display graph and table**). Выберите вариант **Display graph only** (Показать только график) на обеих панелях.

Находясь в **Graph Explorer**, добавьте панель **Stacks** из группы **System Activity** и установите в ней режим **Display table only** (Показать только таблицу). Эта панель стеков содержит сгруппированную информацию обо всех собранных стеках вызова.

Теперь мы можем внимательнее изучить процесс `w3wp.exe`. Сначала выберите на графике временной диапазон, соответствующий нагрузочному тестированию, для чего щелкните правой кнопкой по блоку `sb.exe` на панели **Processes** и выберите из меню команду **Zoom** (Увеличить). Выбрав диапазон, мы можем отфильтровать данные, оставив только интересующий нас процесс веб-приложения. Выберите процесс `w3wp.exe` из списка на панели **Stacks** и в его контекстном меню установите режим **Filter to selection** (Фильтрация по выделенному). Затем раскройте (на панели **Stacks**) `w3wp.exe` в столбце **Process**, **Thread: CSwitch** в столбце **Event Name**, **CLR** в столбце **Stack Tag** и **[Root]** под **Stack (Frame Tags) for JIT**. Раскрыв несколько узлов, расположенных под элементом **[Root]**, вы, вероятно, заметите, что информация о вызванных функциях отсутствует (см. рис. 3.17). Для большинства из них показано только имя модуля и вопросительный знак. Это из-за того, что отсутствуют отладочные символы (PDB-файлы). Давайте настроим их.

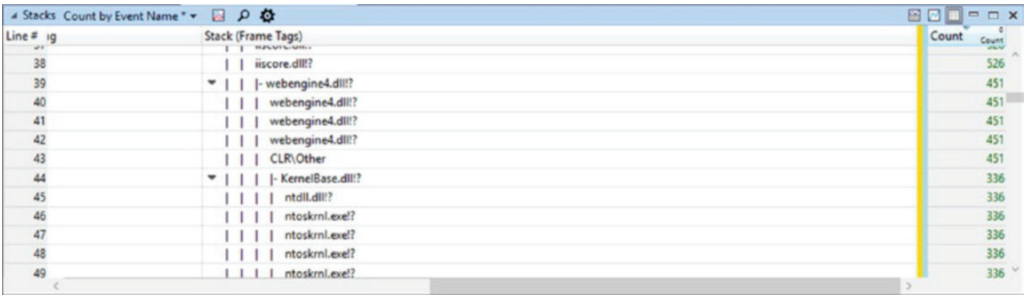


Рис. 3.17 ❖ Из-за отсутствия символов информация в трассе стека неполна

Для настройки символов, используемых в Windows Performance Analyzer, выберите команду **Trace ➤ Configure Symbol Paths** (Трасса ➤ Настройка путей к символам). На этой панели мы зададим каталоги, в которых программа будет искать PDB-файлы. Лучше указать по меньшей мере два источника:

- если переменная среды `_NT_SYMBOL_PATH` установлена, как было рекомендовано в предыдущем разделе, то ее значение будет здесь учтено;
- путь к файлам символов вашего приложения (он прилагается вместе с файлом `WPA-Tutorial.etl`).

На вкладке **Symcache** того же окна укажите каталог, в котором будут храниться локальные копии подготовленных файлов символов. Закончив настройку, закройте окно **Configure Symbols**. Выберите из меню команду **Trace ➤ Load symbols** (Трасса ➤ Загрузка символов). Появится сообщение Loading symbols (Загружаются символы). Скачивание и загрузка символов в программу (даже если они уже закешированы) занимают несколько минут, так что придется подождать.

По завершении этой операции информация в стеке вызовов станет полной. В этом можно убедиться, воспользовавшись функцией **Quick search** (Быстрый поиск) на панели **Stacks** (ей соответствует значок лупы). Наберите «LeakWebApi», чтобы найти все вызовы из нашего тестового приложения (рис. 3.18).

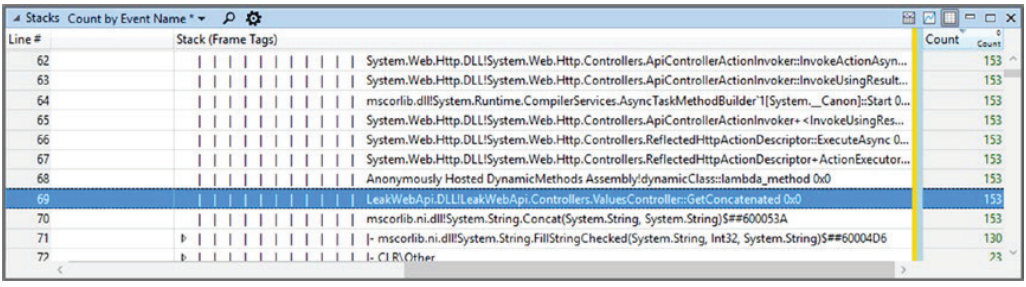


Рис. 3.18 ❖ Полная информация о стеке после загрузки символов

ОБОБЩЕННЫЕ СОБЫТИЯ Многие события интерпретируются в WPA специальным образом, поэтому и созданы такие специализированные панели, как **Processes** или **CPU Usage**. Но, конечно, было бы невозможно подготовить подобные представления для всех событий, записываемых ETW. Поэтому существует панель **Generic Events** (Обобщенные события), на которой показаны все зарегистрированные события. Добавьте ее в наше представление, выбрав из группы **System Activity**. По умолчанию мы увидим все события, сгруппированные по процессам. Можно отфильтровать их, оставив только те, что поступили от процесса `sb.exe`. Для этого выберите команду **Filter to selection** из контекстного меню этого процесса[1]. Раскрыв строку **Microsoft-Windows-DotNETRuntime** в столбце **Provider Name**, а затем задачу **Garbage Collection task** и код операции **win:Start**, мы получим представление, показанное на рис. 3.19 (после увеличения интересующего нас временного диапазона). Заметьте, что для получения такого представления столбцы должны быть расположены в определенном порядке: сначала **Process**, затем **Provider Name**, **Task Name** и, наконец, **Opcode Name**.

Мы настроили представление, так чтобы на экране был виден процесс `sb.exe` (второй столбец), поставщик Microsoft-Windows-DotNETRuntime (третий столбец) и задача GarbageCollection (четвертый столбец). Видно, что в выделенном фраг-

[1] Если вы не видите столбца **Process**, добавьте его и сделайте вторым по порядку на панели **Generic Events**.

менте, продолжительность которого чуть меньше 0,5 с, произошло два события GarbageCollection/Start.

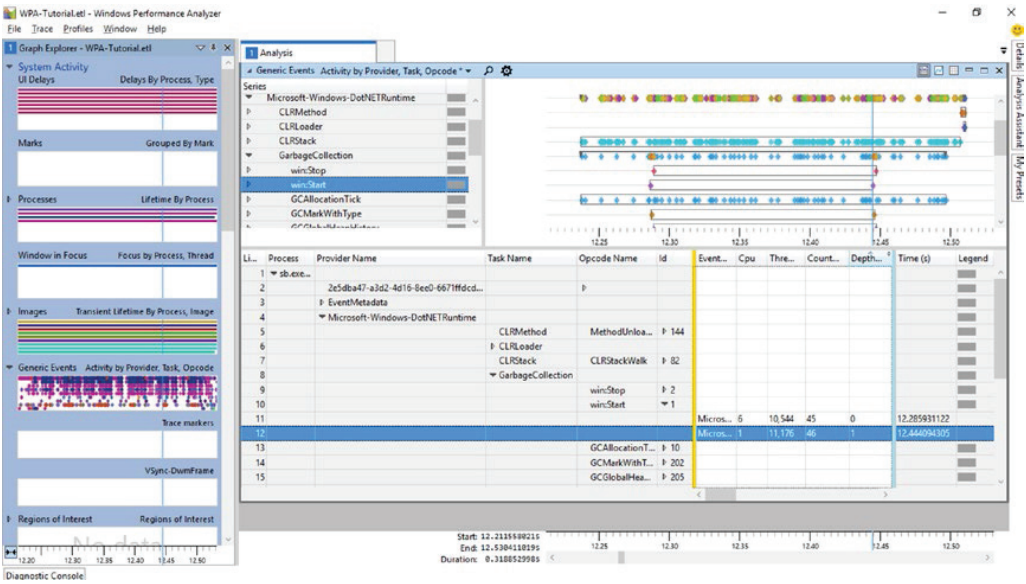


Рис. 3.19 ❖ Представление обобщенных событий для процесса sb.exe и событий, относящихся к поставщику Microsoft-Windows-DotNETRuntime

Мы даже можем посмотреть данные, ассоциированные с этими событиями. Для этого нужно раскрыть группу (в нашем случае раскрыть последний сгруппированный элемент в столбце Id) и прокрутить представление, чтобы стали видны столбцы после желтой линии. Пример такого представления для событий GCStart и GCEnd показан на рис. 3.20.

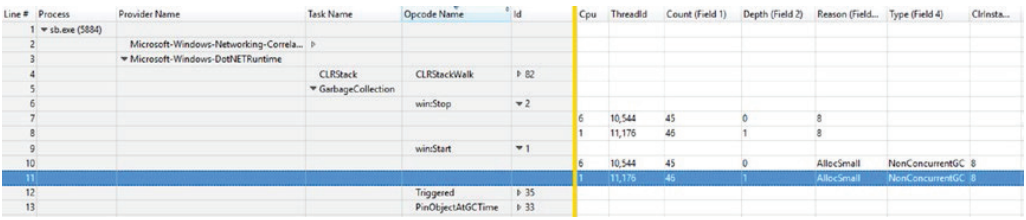


Рис. 3.20 ❖ События начала и конца сборки мусора, видимые в табличном представлении обобщенных событий

Настройка представления путем задания видимости и порядка столбцов, а также желаемой группировки элементов – это и есть основная задача при работе с Windows Performance Analyzer. К счастью, программа в этом отношении весьма гибкая.

Настройку Windows Performance Analyzer можно еще немного продолжить, чтобы упростить анализ. Большую пользу может оказать создание собственных *областей интереса*, *меток стеков* и *профилей*.

ОБЛАСТИ ИНТЕРЕСА Это способ определить области, которые по какой-то причине представляют для нас особый интерес. Границы этих областей определяются заданными событиями – открывающим и закрывающим. Это идеальный механизм для иллюстрации продолжительности сборки мусора, где начальное событие – это `win:Start` (с `Id 1`), а конечное – `win:Stop` (с `Id 2`). Области определяются в отдельном файле, который затем загружается в программу командой **Trace ➤ Trace Properties** (Трасса ➤ Свойства трассы). В появившейся вкладке мы загружаем файлы областей с помощью кнопки **Add...** в секции **Regions of Interest Definitions** (Определение областей интереса). После этого панель **Regions of Interests** становится доступна в разделе **Graph Explorer**.

Такие файлы нужно создавать самостоятельно или искать в интернете. Можно также использовать те, что были подготовлены для этой книги (они есть в сопроводительном репозитории на GitHub): `roi_dotnetfinalization.xml` и `roi_dotnetgc.xml`. Они содержат определения областей в терминах начального конечного события (см. листинг 3.8).

Листинг 3.8 ❖ Пример файла с определением области интереса

```
<Region Guid="{4fbb5999-8f4e-4900-9482-000000000001}"
        Name="DotNETRuntime-GarbageCollection-GC"
        FriendlyName="Garbage Collection">
  <Start>
    <Event Provider="{E13C0D23-CCBC-4E12-931B-D9CC2EEE27E4}" Id="1" Version="2" />
  </Start>
  <Stop>
    <Event Provider="{E13C0D23-CCBC-4E12-931B-D9CC2EEE27E4}" Id="2" Version="1" />
  </Stop>
  <Match>
    <Event TID="true" PID="true" >
    </Event>
    <Parent PID="true" />
  </Match>
  <Naming>
    <PayloadBased NameField="ClrInstanceID" />
  </Naming>
</Region>
```

Как видите, чтобы определить области, необходимо знать, какие события генерируются поставщиком и как они объединяются в пары.

Что касается событий сборщика мусора, то мы можем выделить следующие области:

- сборка мусора (события `GCStart` и `GCEnd`);
- приостановка среды выполнения (события `GCSuspendEEBegin` и `GCSuspendEEEnd`);
- возобновление среды выполнения (события `GCRestartEEBegin` и `GCRestartEEEnd`);
- финализация (события `GCFinalizersBegin` и `GCFinalizersEnd`).

Это позволяет визуализировать и собирать статистику (количество событий и их продолжительность), как показано на рис. 3.21. Обратите внимание, что для создания такого представления нужно задать подходящее увеличение и раскрыть нужные группы элементов в левом списке (он называется Series).

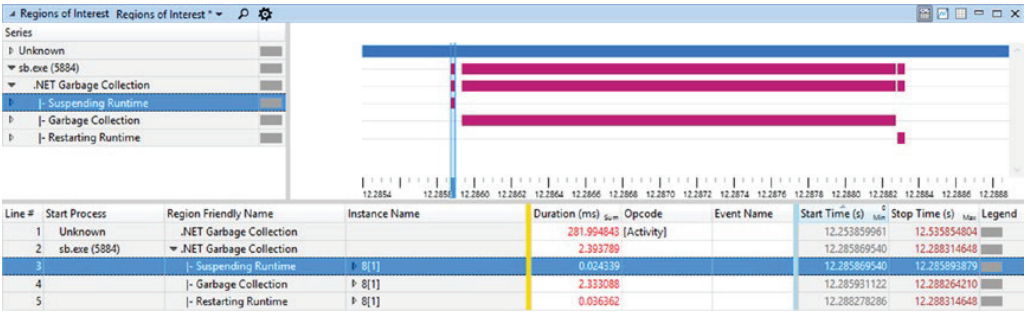


Рис. 3.21 ❖ Представление цикла сборки мусора с помощью пользовательской области интересов

ПЛАМЕННЫЕ ДИАГРАММЫ Анализировать производительность можно и с помощью уже описанных механизмов, в частности группируя вызовы на панели Stacks. Но есть еще один удобный механизм – так называемые *пламенные диаграммы* (flame chart). Представление «Flame by Process, Stack» панели CPU Usage (Sampled) находится в группе Computation. Оно есть в нашем примере ETL-файла, и я рекомендую его использовать. Чтобы получить представление, показанное на рис. 3.22, нужно выполнить следующие действия:

- находясь в табличной части панели **CPU Usage**, выберите из контекстного меню команду **Find in Column...** (Найти в столбце...) и поищите текст LeakWebApi. Если символы загружены, то поиск должен привести к методу `GetConcatenated` нашего контроллера WebAPI;
- выберите его родительский метод (это должен быть метод `lambda_Method`) и выполните команду **Filter To Selection** из его контекстного меню. Она должна сфокусировать представление на одном вызове метода.



Рис. 3.22 ❖ Пример пламенной диаграммы

На пламенной диаграмме очень наглядно представлены последовательности вызовов, но к ней надо привыкнуть. Каждый видимый блок представляет вызовы одной функции. Блоки, расположенные друг над другом, соответствуют вызовам одной функции из другой. Следовательно, диаграмма растет вверх. Чем выше функция, тем глубже стек вызовов. Ширина блока пропорциональна времени, проведенному в конкретной функции (включая вызванные из нее). Таким образом, мы можем быстро понять, какие функции выполняются дольше всего.

> Например, на рис. 3.22 видно, что большую часть времени метод WebAPI `GetConcatened` проводит, вызывая метод `System.String.Concat`, а тот, в свою очередь, тратит время в основном на вызовы `SVR::gc_heap::fire_etw_allocation_event`. Это очевидное доказательство того, что подключение сеанса ETW к нашему приложению обходится отнюдь не даром. Это объясняется тем, что мы требуем запись стека вызовов при каждом событии CLR, в чем можно убедиться, рассмотрев, какие методы вызываются из `fire_etw_allocation_event`. Большая часть времени уходит на вызов `clr.dll!ETW::SamplingLog::GetCurrentThreadsCallStack`. В общем, получение стека вызовов при каждом событии выделения памяти – а они происходят часто – не слишком удачная идея. Однако для учебных целей это вполне оправдано.

МЕТКИ СТЕКОВ Как мы видели, при протоколировании событий ETW можно сохранить также и стек вызовов в момент возникновения события. Windows Performance Analyzer позволяет просматривать эту информацию в столбце Stack. Но для более широкого анализа одного стека вызовов недостаточно, большую ценность представляет агрегированная информация. Один из механизмов агрегирования – *метки стеков*. Они позволяют сгруппировать вызываемые методы в соответствии с заданным шаблоном. Все события, отвечающие ему, будут помечены заданной меткой стека.

Метки стеков по умолчанию находятся в файле `C:\Program Files (x86)\Windows Kits\10\Windows Performance Toolkit\Catalog\default.stacktags`, в т. ч. и метки, относящиеся к CLR и конкретно GC. Поэтому в столбце Stack Tag мы видим стеки, сгруппированные в узлы CLR и GC (а не просто перечисление всех методов подряд).

ПОЛЬЗОВАТЕЛЬСКИЕ ГРАФИКИ Начиная с версии Windows Performance Toolkit для Windows 10 появилась возможность строить собственные графики по данным о событиях. Иными словами, мы можем нарисовать график, в котором по оси Y откладывается одно из полей выбранного события. А по оси X автоматически откладывается время события. Единственное требование – значения выбранного поля должны быть целыми числами.

К сожалению, нас это ограничение не устраивает. Подавляющее большинство интересующих нас событий, относящихся к сборщику мусора, представлены в шестнадцатеричном формате. Это относится к разного рода размерам, использованию памяти и т. д. и т. п. Так что в данный момент этот механизм не особенно полезен, и мы им пользоваться не будем.

ПРОФИЛИ Поскольку настройка всех панелей может отнимать много времени, Windows Performance Analyzer предлагает возможность сохранить текущие представления, используя профили. Для этого выберите из меню команду **Profiles ➤ Export…** (Профили ➤ Экспорт). А для загрузки профиля выберите команду **Profiles ➤ Apply** (Профили ➤ Применить). Помимо собственно настроек представле-

ния (в т. ч. порядка и формата столбцов), в профиле может храниться файл, определяющий области интереса.

PerfView

Программа Windows Performance Toolkit изначально предназначалась для разработчиков Windows и драйверов. Благодаря заложенным в ней широким возможностям настройки мы можем адаптировать ее к среде .NET, как было продемонстрировано в предыдущем разделе. Но есть еще один основанный на ETW инструмент, который с самого начала проектировался для анализа проблем производительности .NET, – PerfView. Его создателем и покровителем является Вэнс Моррисон, архитектор, отвечающий за производительноть среды исполнения .NET, и этот инструмент используется командой .NET для оптимизации производительности самого .NET Framework и управляемого кода вообще. Поэтому он интересен и нам. И кстати, недавно все разработчики, заинтересованные в производительности и внутреннем устройстве CLR, с радостью узнали, что исходный код PerfView теперь полностью открыт и выложен на GitHub.

В терминологии ETW программа PerfView является одновременно контроллером и потребителем (предоставляет развитые аналитические средства). Она в минимальной степени вмешивается в работу приложений и не требует установки, т. к. состоит из единственного исполняемого файла – `perfview.exe`. Это позволяет без труда использовать ее на любом компьютере, в т. ч. на промышленных серверах. Начать работу с PerfView можно двумя способами:

- скачать ZIP-файл по адресу https://www.microsoft.com/en-us/download/details.aspx?id=28567, распаковать его и запускать, когда будет нужно;
- самостоятельно скомпилировать программу из исходного кода, доступного на сайте GitHub: https://github.com/Microsoft/perfview.

Кстати, заметим, что этим инструментом можно управлять из командной строки и из PowerShell, что открывает возможность автоматизации и особенно полезно для анализа системы в условиях промышленной эксплуатации (если доступ к ней ограничен, то подготовленную команду нужно передать системному администратору, который запустит программу).

Несмотря на простоту подготовительных процедур, первая встреча с программой может отпугнуть кого угодно. Она заслуживает звания самого мощного и вместе с тем самого пугающего (на первый взгляд) инструмента. Интерфейс не назовешь ни красивым, ни интуитивно понятным, и даже не ясно, с чего начинать. К счастью, имеется очень подробная справка. Для каждого параметра и каждого элемента GUI имеется ссылка на документацию. Ниже описано несколько простых сценариев использования, но я рекомендую почаще заглядывать в справку. Там вы найдете развернутое описание всех рассматриваемых здесь тем. Поверьте мне, каждая минута, потраченная на изучение этого инструмента, стоит того.

> **ПРИМЕЧАНИЕ** Большая часть функциональности PerfView, связанной с ETW, основана на библиотеке `TraceEvent`. Мы кратко рассмотрим ее возможности в главе 15. Хотя PerfView преимущественно базируется на ETW, в ней имеется встроенный профилировщик `ETWCLrProfiler` (на основе CLR Profiling API), который позволяет перехватывать вызовы методов .NET (чтобы воспользоваться им, включите флажок **.NET Call** в диалоговом окне **Collect**).

> В качестве облегченного инструмента для анализа ETW рекомендую также программу etrace, написанную Сашей Гольдштейном и доступную по адресу https://github.com/goldshtn/etrace. Она позволяет управлять сеансами ETW из командной строки, задавая различные фильтры.

Если в основу Windows Performance Analyzer положена концепция графиков, то PerfView предпочитает табличное представление. Практически все, что показывает эта программа, представлено в форме таблиц. Иногда это сбивает с толку, потому что одинаково анализируется все: потребление памяти, стеки вызовов и т. д.

После запуска программы появляется окно с подробной справкой. В этот момент можно выполнить три главных действия:

- начать сбор данных ETW с помощью команды **Collect ➤ Collect**;
- начать анализ данных, введя путь к каталогу в поле под меню и выбрав интересующий вас ETL-файл;
- получить дамп памяти, выполнив команду **Memory ➤ Take Heap Snapshot** (Память ➤ Сделать снимок кучи).

Как и при работе с другими инструментами, необходимо задать пути к символам, для чего предназначена команда меню **File ➤ Set Symbol Path** (Файл ➤ Задать путь к символам). Рекомендуется задавать три пути:

- адрес общедоступного сервера символов Майкрософт – тот, что указан в переменной среды _NT_SYMBOL_PATH;
- путь к каталогу с символами образа NGEN рядом с открытым ETL-файлом, хотя это необязательно, т. к. PerfView может пересоздать их автоматически;
- путь к файлам символов нашего приложения.

Сбор данных

Поскольку PerfView – контроллер ETW, он позволяет управлять сеансом трассировки ETW. После выбора команды **Collect** появляется диалоговое окно с большим количеством параметров (рис. 3.23).

Среди доступных параметров многие относятся к .NET. Потратим немного времени на их объяснение, хотя все это есть в справке. Самые интересные для нас параметры находятся в секции **Advanced Options** (Дополнительные параметры):

- .NET – включает получение событий по умолчанию от поставщиков .NET;
- .NET Stress – включает события поставщиков .NET, относящиеся к нагрузочному тестированию самой среды выполнения. Это редкие события, которые в основном используются разработчиками CLR;
- GC Collect Only – отключает все остальные поставщики, оставляя только поставщика .NET событий, связанных с процессом GC. В этом режиме накладные расходы очень малы, что позволяет собирать основную диагностическую информацию, относящуюся к сборке мусора, в течение длительного времени;
- GC Only – аналогичен предыдущему, но дополнительно включается выборочный сбор данных о выделении памяти из кучи GC (после выделения каждых 100 КБ памяти для объектов);
- .NET Alloc – включает сбор событий со стеком вызовов при каждом выделении памяти из кучи GC. Это очень накладный режим, который может замедлять работу программы в несколько раз. И на рис. 3.21 мы даже видели, как выглядят эти накладные расходы;

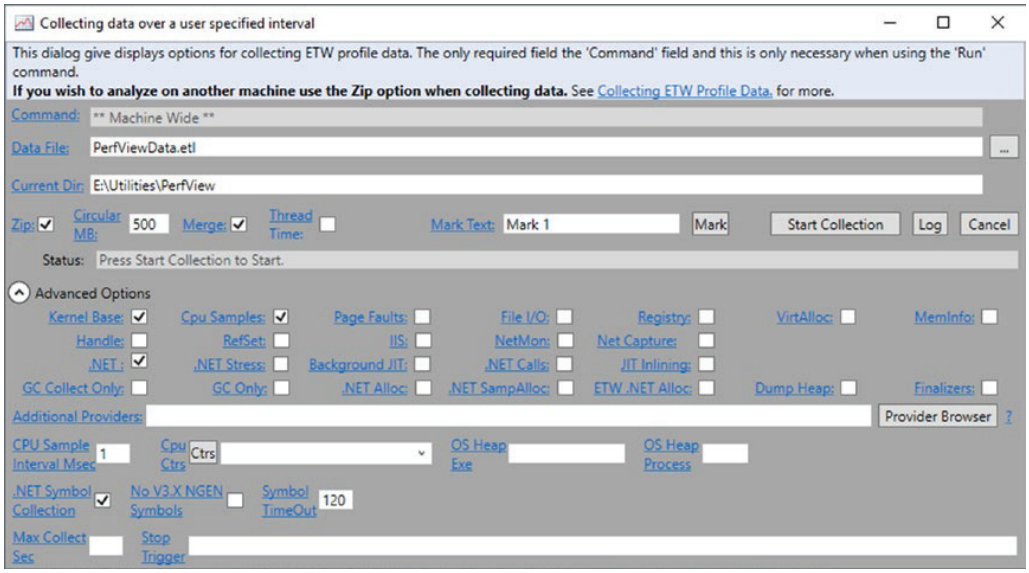


Рис. 3.23 ❖ Диалоговое окно сбора данных с раскрытой секцией Advanced

- .NET SampAlloc – разрешает генерировать события после выделения каждых 10 КБ памяти из кучи GC. Этот режим основан не на встроенных событиях ETW, а на использовании CLR Profiler API, с помощью которого библиотека ETWClrProfiler внедряется в процессы;
- ETW .NET Alloc – разрешает генерировать события о выборочном выделении, но вместо внедрения библиотеки профилировщика используется ключевое слово `GCSampledObjectAllocationHigh`, добавленное в версии .NET 4.5.3;
- Finalizers – разрешает события, относящиеся к процессу финализации внутри GC;
- Additional providers – это поле позволяет задать дополнительные поставщики. Его также можно использовать для точной настройки поставщиков, которые в любом случае включены. Например, чтобы включить сохранение стеков для исключений CLR, можно ввести `Microsoft-Windows-DotNETRuntime:ExceptionKeyword:Always:@StacksEnabled= true`. Для этого поля в справке имеется развернутое описание;
- CPU Ctrs – этот параметр позволяет включить низкоуровневые счетчики, относящиеся к работе ЦП, например счетчики неправильных предсказаний переходов или непопадания в кеш. Имейте в виду, что для доступа к этим событиям необходимо отключить виртуализацию средствами Hyper-V.

Примечание: помимо параметров для .NET, полезно знать о следующих общих параметрах:

- Zip – упаковка файлов в архив, чтобы их было проще передать на другой компьютер для последующего анализа;
- Merge – объединение всех файлов в один, но без создания отдельного ZIP-файла.

На эти два параметра можно не обращать внимания, если вы не планируете никому посылать собранные данные. Однако очень важно включать флажок **Merge**, если вы планируете выполнять анализ не на той машине, где собирались данные. Этот режим включает подготовительные действия для разрешения символов, поэтому если флажок не включен, то собранные данные окажутся бесполезны на другом компьютере.

> Очень популярно включение сбора данных ETW в режиме командной строки PerfView. Так, можно попросить группу технической поддержки собрать данные в производственной среде, сообщив им, какую команду выполнить. Например, следующая команда запускает облегченный сеанс для записи событий, относящихся к GC: `perfview /GCCollectOnly /nogui /accepteula /NoV2Rundown /NoNGENRundown /NoRundown /merge:true /zip:true collect`. В командной строке можно задать также условия остановки сеанса, например если сборка мусора продолжалась дольше заданного числа миллисекунд. Для получения дополнительных сведений наберите `perfview -?`.

Анализ данных

С помощью PerfView мы можем открывать ETL-файлы, записанные как им самим, так и любым другим инструментом, поддерживающим ETW. Открыв ETL-файл, мы увидим окно, показанное на рис. 3.24. Слева находятся все подготовленные данные для анализа – в зависимости от того, какие поставщики и события были выбраны на этапе записи сеанса.

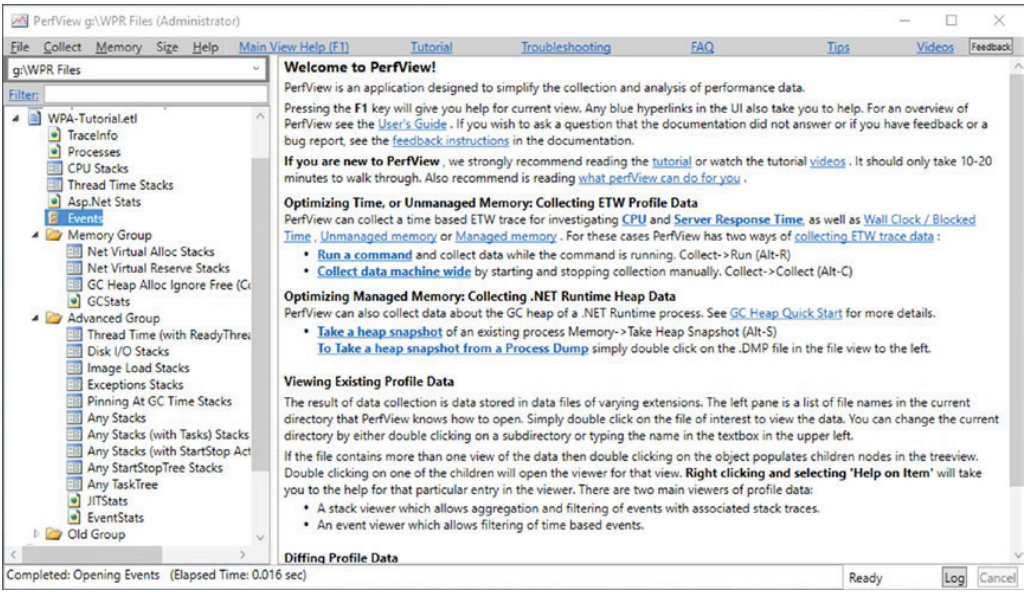


Рис. 3.24 ❖ Пример ETL-файла, открытого в PerfView

Одно из самых простых представлений – панель **Generic Events**, на которой показаны все записанные события. Если открыть ее и ввести GC в поле **Filter**, то мы увидим все события DotNetRuntime, относящиеся к сборке мусора (рис. 3.25):

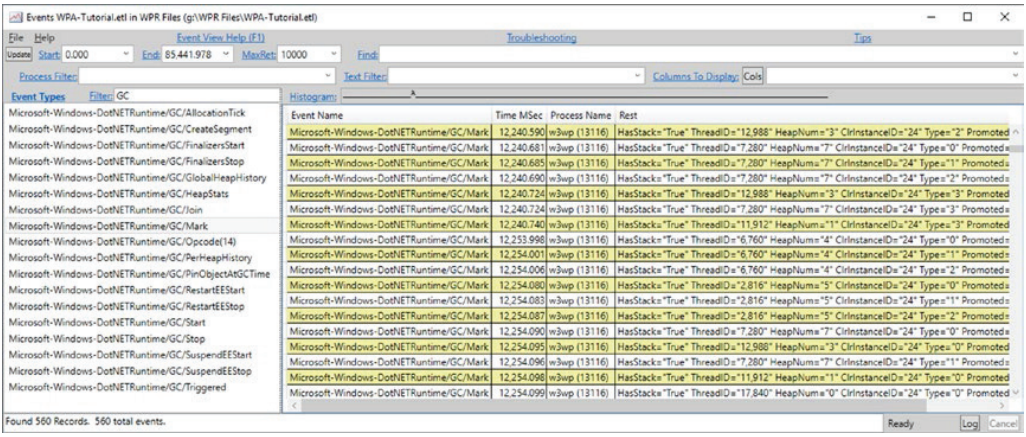


Рис. 3.25 ❖ PerfView – события, относящиеся к сборке мусора, показаны на панели **Generic Events**

Как видим, помимо стандартных столбцов, связанных с событием, есть еще столбец **Rest**, содержащий все сведения о событии. Можно также выбрать конкретные данные из событий, нажав кнопку **Cols** (Столбцы). Например, чтобы отфильтровать все события, кроме `Microsoft-Windows-DotNETRuntime/GC/HeapStats`, введите часть этого имени (например, GC/HeapStats) в поле **Filter**. Тогда нажатие кнопки **Cols** приведет к выбору всех полей GenerationSize. Дополнительно введите в поле **Process Filter** часть имени интересующего вас процесса, однозначно идентифицирующую его. Например, нам хотелось бы создать таблицу статистических данных о GC (рис. 3.26), которую можно было бы импортировать в Excel и визуализировать.

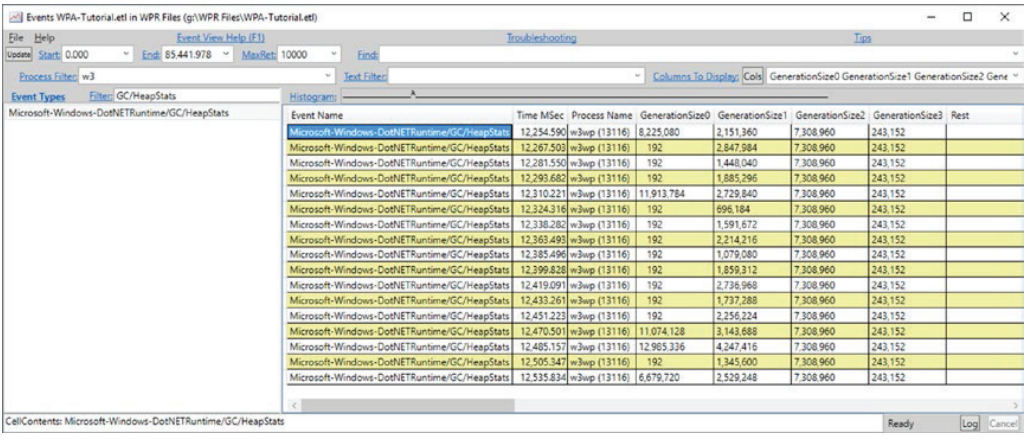


Рис. 3.26 ❖ PerfView – настроенное представление событий, относящихся в GC

Однако просмотр и анализ отдельных событий ETW – дело утомительное. С точки зрения анализа памяти .NET, самым важным, несомненно, является представление GCStats в группе **Memory** в главном окне. Это представление включает

полную агрегированную информацию о поведении сборщика мусора, в т. ч. статистику выполненных сборок (рис. 3.27). Мы будем часто возвращаться к нему в этой книге.

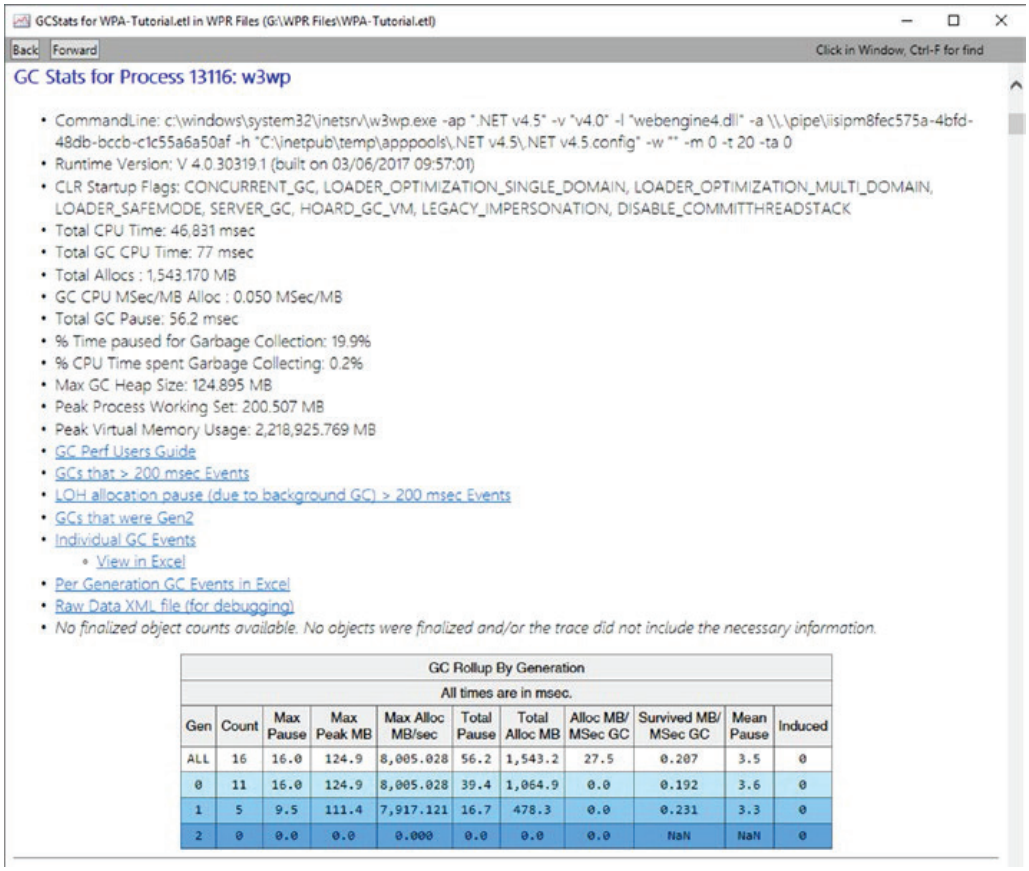


Рис. 3.27 ❖ PerfView – представление GCStats

Дополнительно в столбце **Rest** на рис. 3.25 видно, что у некоторых событий есть атрибут `HasTrack = "True"`. Если вы хотите увидеть стек вызовов такого события, выберите команду **Open Any Stacks** (Открыть все стеки) из его контекстного меню (но внимание: это следует делать в контексте столбца **Time MSec**). В результате откроется еще одно очень популярное представление PerfView – дерево вызовов (рис. 3.28).

Напомним, что если имя функции не распознано, выберите команду **Lookup Symbols** (Поиск символов) из контекстного меню. Она должна инициировать чтение подходящих символов.

Есть и много других чрезвычайно полезных представлений. Мы неоднократно будем пользоваться ими. Но я призываю вас прямо сейчас оглядеться и изучить такие представления, как CPU Stacks, уже упомянутое GC Stats и Asp.Net Stats.

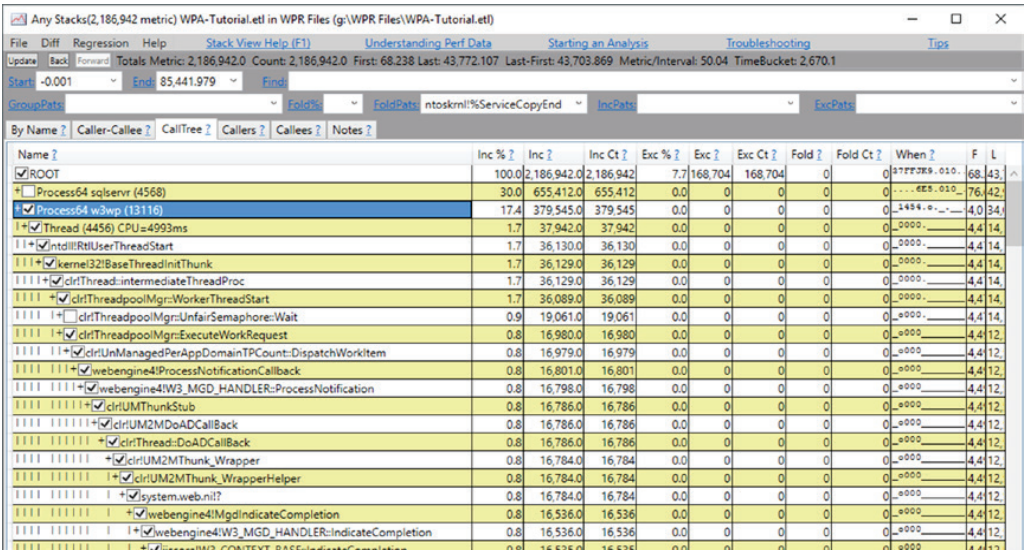


Рис. 3.28 ❖ PerfView – представление Any Stacks

Снимки памяти

Выбрав из меню команду **Take Heap Snapshot** (Сделать снимок кучи), вы увидите окно сбора данных о памяти **Collecting Memory Data**. Имеет смысл сразу же воспользоваться полем **Filter** и найти интересующие процессы. Выбрав процесс, нажмите кнопку **Dump GC Heap** (Выгрузить кучу GC) – до получения результата (рис. 3.29) может пройти несколько секунд или несколько десятков секунд.

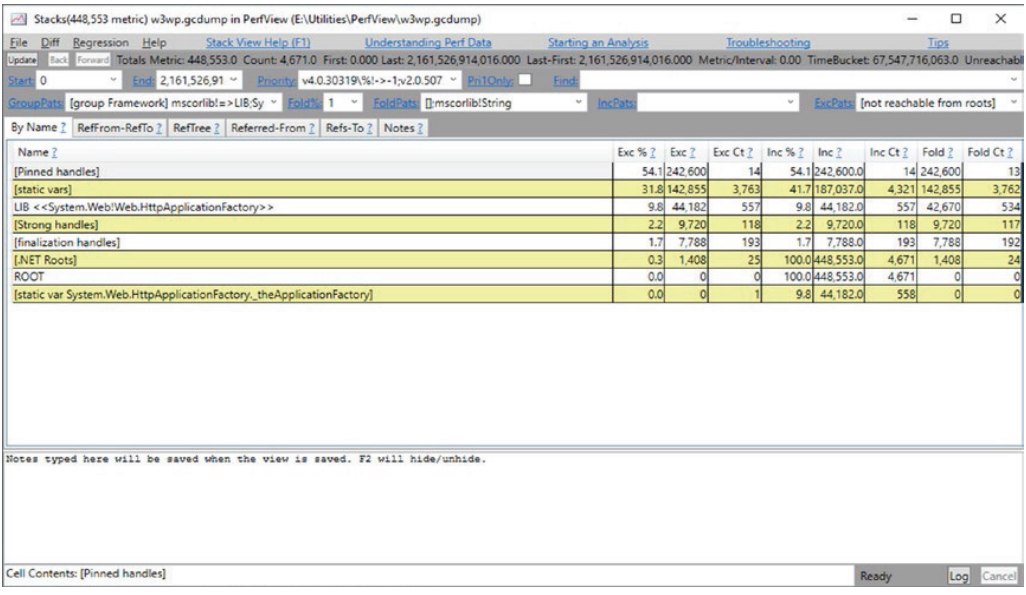


Рис. 3.29 ❖ PerfView – представление снимка памяти

ПРИМЕЧАНИЕ Снимок памяти – не типичный дамп памяти: он содержит не всю память процесса. Это представление о состоянии процесса, в котором хранится предварительно обработанный граф объектов, но отсутствует содержимое объектов и игнорируются все неуправляемые области памяти.

В открывающемся окне показана уже знакомая таблица, но теперь в ней находится не дерево вызовов, а дерево ссылок, узлами которого являются типы объектов или категории типов. Например, на открытой по умолчанию вкладке **By Name** (По имени) показан сводный перечень всех типов, встречающихся в дампе памяти. Можно исследовать любую строку дальше, выбрав команду **Memory ➤ View Objects** (Память ➤Просмотр объектов) из контекстного меню (или нажав **Alt+O**). Сделав это для строки **[static vars]**, мы увидим список всех статических переменных в снимке памяти (рис. 3.30).

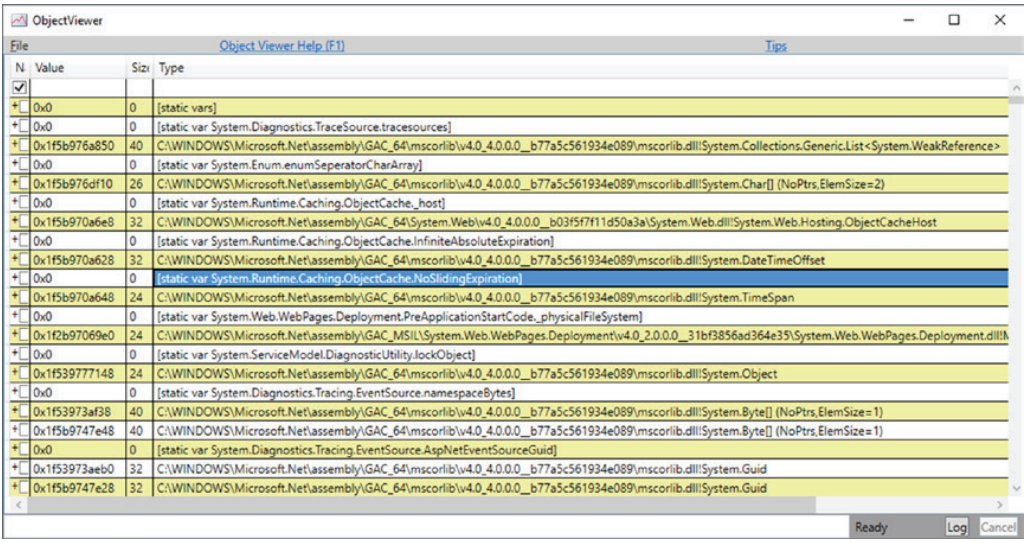


Рис. 3.30 ❖ PerfView – все статические переменные в снимке памяти

Здесь строки идут парами: в одной показана объявленная статическая переменная, в другой – присвоенный ей объект. Если раскрыть этот объект, то можно будет изучить всех его потомков (поля).

У снимков памяти есть еще одна важная функция – сравнение. Это позволяет отслеживать тенденции в программе, например быстро выявлять причину утечки памяти. Чтобы сравнить два снимка (созданных, как описано выше), откройте оба и из меню **Diff** (Различие) выберите способ сравнения со вторым файлом. Мы увидим списки различий (Diff Stack), в которых данные отображаются так же, как в одном снимке, но с тем важным отличием, что значения в столбцах показывают различия между двумя файлами (рис. 3.31).

Заметим, что в диалоговом окне **Collecting Memory Data** имеется флажок **Freeze** (Заморозить), по умолчанию выключенный. Он позволяет приостановить весь процесс на время, необходимое для получения снимка кучи. Очевидно, что степень вмешательства при этом максимальна, но данные получаются очень точными. В условиях про-

мышленной эксплуатации вашего приложения этот флажок, конечно, лучше оставить выключенным, но тогда данные получатся не полностью согласованными (поскольку снимок делается, когда приложение продолжает работать).

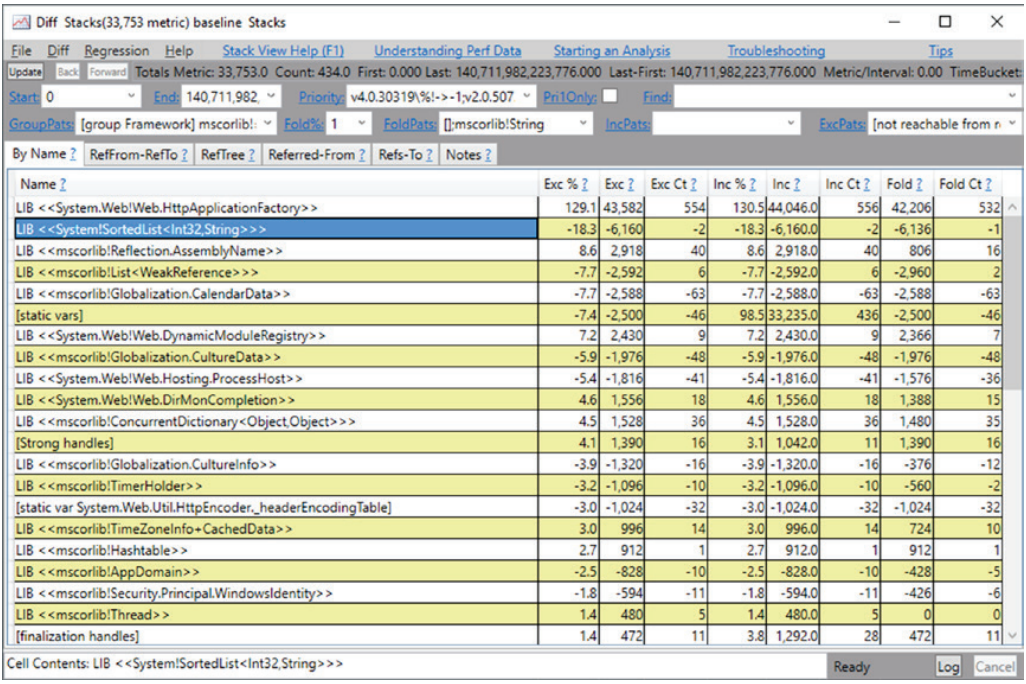


Рис. 3.31 ❖ PerfView – различие двух снимков памяти

Истинная ценность PerfView заключается в низких накладных расходах и возможности анализировать приложение в условиях промышленной эксплуатации. Мы можем использовать ее как для непрерывного мониторинга производительности, так и для поиска ошибок. Она дает колоссальный объем данных, так что большинство проблем, связанных с производительностью или памятью, удастся диагностировать. Единственный недостаток – трудность изучения пользовательского интерфейса и скрытых за многочисленными параметрами возможностей.

Следует очень осторожно подходить к заданию объема требуемой информации. Хотя накладные расходы программы малы, если вы будете собирать слишком большое количество данных, ее использование в промышленной среде станет невозможным. Получение информации от нескольких поставщиков по нескольким ключевым словам не должно стать проблемой. Но, как вы могли убедиться, попытка получить сведения о стеке вызовов для каждого выделяемого из кучи объекта приведет к неприемлемым издержкам. Лучше придерживаться простого принципа – прежде чем собирать некоторый набор данных в промышленной среде, посмотрите, как это повлияет на приложение и систему в целом в тестовой среде.

ProcDump и DebugDiag

Нужда в анализе проблем с памятью чаще всего возникает уже в условиях промышленной эксплуатации. Тогда самое простое – сделать дамп памяти проблемного приложения и спокойно проанализировать его на другой машине. Для получения дампа памяти есть много инструментов. Упомяну лишь два из них, поскольку они покрывают большинство типичных потребностей. Обе программы являются автономными и доступны для скачивания с сайтов Майкрософт:

- ProcDump – https://docs.microsoft.com/en-us/sysinternals/downloads/procdump;
- DebugDiag – https://blogs.msdn.microsoft.com/debugdiag/.

ProcDump – командная программа, позволяющая сделать дамп памяти одной командой:

```
procdump -ma <process_pid>
```

Но у нее есть многочисленные дополнительные параметры, например сделать дамп, когда потребление памяти или процессора либо любой другой счетчик производительности превышает заданный уровень. Есть также возможность снимать дампы памяти периодически и т. д. Полный перечень параметров приведен в справке по программе.

DebugDiag – программа с графическим интерфейсом, которая позволяет делать нечто подобное. Ее функциональность немного шире, в частности она позволяет делать дамп, когда время отклика от заданного HTTP-адреса превышает указанный порог. В состав этой программы входит инструмент DebugDiag Analysis, который служит для автоматической генерации отчетов по сделанным дампам памяти. Это позволяет легко и быстро просмотреть сведения о наиболее очевидных проблемах.

> Можно также рассмотреть замечательную программу Minidumper, написанную Сашей Гольдштейном и доступную по адресу https://github.com/goldshtn/minidumper. Она умеет сохранять минимальный объем памяти, необходимый для анализа памяти .NET (исключая ненужный груз в виде исполняемых файлов, DLL, областей неуправляемой памяти и т. д.). Такой «мини-дамп» анализируется как обычный дамп памяти, но может быть в несколько раз меньше. Особенно полезно это при создании дампов очень больших процессов.

WinDbg

Из всех инструментов, упомянутых в этой главе, WinDbg, без сомнения, самый низкоуровневый. В нем можно делать почти все: запускать отладку .NET-приложений, приложений Windows и даже самого ядра. Универсальность вкупе с толикой негибкости составляет отличительную особенность этого инструмента. Он позволяет заглянуть по-настоящему глубоко – до уровня отдельных битов. Суровая простота его интерфейса дает возможность довольно быстро провести анализ некоторых случаев без лишних деталей вроде красивых рисунков, представляющих результаты различных видов анализа, доступных в других инструментах. Поэтому я иногда предпочитаю воспользоваться WinDbg, чем ждать, пока какой-нибудь другой инструмент обработает данные, как ему удобно.

По счастью, в середине 2017 года появилась полностью обновленная версия WinDbg, в которой пользовательский интерфейс поприятнее и лучше поддается настройке.

Установить WinDbg можно двумя способами – как часть Windows Driver Kit (WDK) или Windows Software Development Kit (старая версия) либо из магазина Windows (новая версия). При установке SDK можно просто отменить все компоненты, кроме средств отладки для Windows (Debugging Tools for the Windows), который включает WinDbg. После установки старой версии программы на диске появляются два варианта: для анализа 32- и 64-разрядных программ. Новая версия, устанавливаемая из магазина Windows, представляет собой универсальную программу (но на момент написания этой книги была доступна только версия для предварительного ознакомления).

Программа WinDbg – великолепное средство для экспериментов, помогающих лучше понять среду выполнения .NET. Мы можем присоединить ее к управляемой программе и отлаживать ее (вместе с самой средой), как привыкли в Visual Studio. Но в повседневной работе WinDbg все же чаще используется для анализа уже имеющегося дампа памяти. Далее мы будем работать с новой версией WinDbg.

> **ПРИМЕЧАНИЕ** На самом деле WinDbg – довольно простая обертка библиотеки DbgEng, представляющей собой платформу отладки в Windows. В контексте анализа .NET ее мощь проистекает из перечисленных ниже расширений, написанных специально для .NET.

После запуска WinDbg появляется окно (рис. 3.32), в котором доступно несколько операций:

- повторно выполнить недавние действия – это особенно полезно для присоединения к одному и тому же процессу (или его запуска);
- запустить процесс или присоединиться к процессу – после щелчка по ссылке **Attach to process** (Присоединиться к процессу) будет показан список всех работающих процессов;
- открыть файл дампа.

Существуют и другие возможности, например: JIT-отладка (пока не поддерживается для управляемого кода), удаленное подключение к другому отладчику и т. д.

По умолчанию WinDbg работает как платформенный отладчик, поэтому не понимает относящихся к .NET структур и понятий. Необходимо использовать расширения, наделяющие WinDbg такими знаниями. Расширений много, назовем лишь самые популярные.

- SOS – простое, но очень мощное расширение, входящее в состав самой среды .NET. Аббревиатура расшифровывается как *Son of the Strike* (сын Страйка). Объясняется это тем, что оно является преемником средства отладки под названием Strike, которое использовалось в ходе разработки .NET Framework.
- SOSEX – это расширение SOS (отсюда и название), которое можно бесплатно скачать со страницы его автора Стива Джонсона: http://www.stevestechspot.com/default.aspx. Оно расширяет функциональность отладки управляемого кода и дампов памяти.
- NetExt (автор – Родни Виана, доступна по адресу https://github.com/rodneyviana/netext) и MEX (Managed-code Debugging Extension, доступна по адресу

https://www.microsoft.com/en-us/download/details.aspx?id=53304) – еще два расширения, которые позволяют делать более сложные вещи, чем два предыдущих.

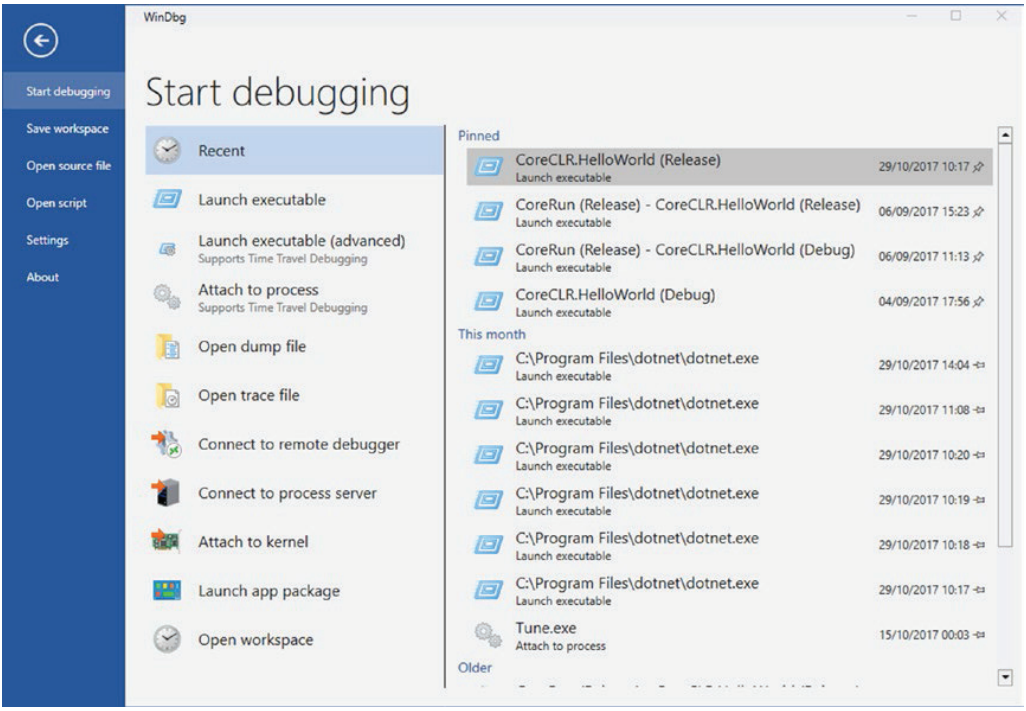


Рис. 3.32 ❖ Главное окно WinDbg

Для загрузки расширения следует выполнить команду `.load <путь к файлу>`, например `.load g:\Tools\Sosex\64bit\sosex.dll`. В случае встроенного в .NET расширения SOS можно также вручную ввести путь к библиотеке `sos.dll` или воспользоваться удобной командой `.loadby`, которая позволяет найти путь, ориентируясь на местоположение второго аргумента. Это значит, что `sos.dll` находится в том же каталоге, что и `clr.dll` (основная библиотека среды выполнения .NET):

```
> .loadby sos clr
```

Чтобы проверить, успешно ли завершилась команда, выполните команду `!sos.help`, которая печатает все имеющиеся в SOS команды. Например, можете поинтересоваться, что делает команда `!threads`. Для загрузки других расширений наберите команду `!load <путь к sosex.dll>` и аналогично для netext и mex. Не забывайте, что нужно загружать версию для x86 или x64 в зависимости от того, какое приложение или дамп памяти вы собираетесь отлаживать. Получить список доступных команд позволят команды `!sosex.help` и `!netext.help`.

Существует еще один полезный инструмент, используемый вместе с WinDbg, – окна дерева команд. Утомительно вводить одни и те же команды снова и снова, поэтому можно создать файл, содержащий структурированный список доступных

команд. Затем с помощью команды `.cmdtree <file>` создаются специализированные окна, из которых команду можно выбрать простым щелчком.

> **ПРИМЕЧАНИЕ** Можно также создать дамп памяти самого ядра операционной системы, подключившись к удаленной машине или проанализировав аварийный дамп. Нам эта возможность не понадобится, просто имейте в виду, насколько мощной является программа WinDbg.

В дополнение к WinDbg можете рассмотреть инструмент msos, созданный Сашей Гольдштейном и доступный по адресу https://github.com/goldshtn/msos. Автор описывает его как «командную среду вроде WinDbg для выполнения команд SOS, не имея самого расширения SOS». Можно считать, что это командная обертка функциональности SOS, работающая без установки WinDbg и поиска расширений. К тому же у нее есть дополнительные возможности, например интерпретация произвольных динамических запросов к объектам в куче и их классам.

Дизассемблеры и декомпиляторы

Хотя это и не связано напрямую с управлением памятью, иногда полезно понимать, что делает некий фрагмент приложения, написанного не вами и существующего только в двоичном виде. Как мы скоро увидим, двоичный код .NET довольно легко разбирается. Существуют инструменты, позволяющие просматривать код других программ в удобной форме. Один из лучших – я буду его постоянно использовать – бесплатная программа с открытым исходным кодом dnSpy, выложенная пользователем 0xd4d на GitHub по адресу https://github.com/0xd4d/dnSpy. Этот инструмент позволяет не только просматривать код, но также отлаживать и модифицировать его. Мы воспользуемся им, чтобы показать код самих стандартных библиотек .NET и программ, скомпилированных для этой платформы.

Имеются и другие популярные инструменты, например ILSpy, JetBrains dotPeek и Redgate .NET Reflector, но dnSpy особенно полезен благодаря возможностям редактирования, и для наших целей его будет вполне достаточно.

BenchmarkDotNet

Часто возникает необходимость измерить производительность некоторых участков программы. Особенно это будет полезно в этой книге, поскольку мы собираемся сравнивать результаты различных методов оптимизации. Было бы идеально, если бы вместе с измерением производительности кода (времени его выполнения) можно было бы измерить и объем необходимой памяти.

Библиотека BenchmarkDotNet умеет делать именно это и многое другое. Она позволяет измерять производительность каждого метода и сравнивать производительность двух методов при различных параметрах. Мы можем тестировать производительность для разных версий .NET, конфигураций JIT-компилятора и GC и т. п.

И что особенно важно, библиотека позволяет нам не наделать ошибок при написании микротестов производительности. В ней имеются тщательно продуманные стадии теста, например прогрев и охлаждение. Для выполнения теста производится много итераций. Все измерения подвергаются статистической обработке. Вычисляются перцентили, и распознается многомодальное распределение данных (в частности, строится упрощенная гистограмма). В общем и целом это мощный и вместе с тем простой в использовании инструмент.

В листинге 3.9 показана подготовка простого теста. По существу, нужно только снабдить атрибутами интересующий нас класс и метод. Как уже отмечалось, мы можем прогонять тест при различных дополнительных параметрах (в данном случае N).

Листинг 3.9 ❖ Пример теста для BenchmarkDotNet

```
[BenchmarkDotNet.Attributes.Jobs.ShortRunJob]
[MemoryDiagnoser]
public class TailCallTest
{
  [Params(5, 10, 20)]
  public int N { get; set; }
  [Benchmark]
  public long FibonacciRecursive()
  {
    return FibonacciRecursiveHelper(N);
  }
  private long FibonacciRecursiveHelper(long n)
  {
    if (n < 3)
       return 1;
    return FibonacciRecursiveHelper(n - 2) + FibonacciRecursiveHelper(n - 1);
  }
}
```

Для выполнения этого теста нужно просто вызвать в нашей программе метод `BenchmarkRunner.Run<TailCallTest>()`. Результат (рис. 3.33) показывает среднее время выполнения каждого метода для двух значений параметров и двух разных JIT-компиляторов, а также разнообразные статистические показатели.

Эту библиотеку можно дополнить различными средствами протоколирования (loggers), анализаторами, средствами диагностики и т. д. Два из них представляют для нас особый интерес. Средство GC and Memory Allocation Diagnoser (MemoryDiagnoser) анализирует, сколько раз запускалась сборка мусора и сколько операций выделения памяти было произведено во время работы теста. Есть также средство Hardware Counters Diagnoser (HardwareCounters), которое доступно только в Windows и может предоставить различные статистические сведения о работе оборудования, например количество непопаданий в кеш.

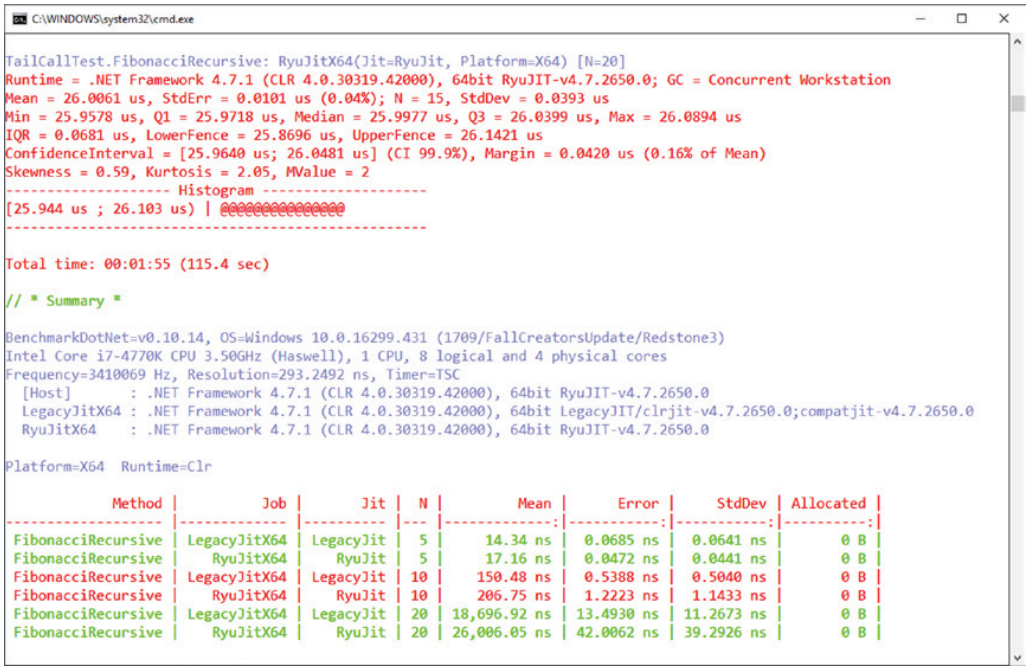


Рис. 3.33 ❖ Результаты теста под управлением BenchmarkDotNet

Коммерческие инструменты

Все рассмотренные выше инструменты бесплатны. И хотя они предлагают весьма неплохие возможности, иногда использовать их неудобно. С другой стороны, коммерческие программы с самого начала пишутся, имея в виду приятный пользовательский интерфейс. Ниже перечислено несколько таких инструментов. Не берусь утверждать, что этот список полон. С момента написания книги до выхода в печать многое может измениться. Я лишь упоминаю те инструменты, которыми пользовался во время работы над книгой и в течение всей своей многолетней практики.

У вас может сложиться другое мнение об этих программах. Рекомендую попробовать каждый из них во время чтения книги и впоследствии. Тогда вы сами можете решить, что для вас больше подходит. Все они очень удобны, особенно в руках специалиста, хорошо владеющего предметом (надеюсь, что после прочтения книги вы войдете в их число).

Нет смысла выделять в этой книге какой-то один инструмент (тогда встал бы вопрос – какой именно?). Вместо этого я предпочел уделить больше внимания бесплатным альтернативам с открытым исходным кодом.

Visual Studio

Трудно представить себе разработчика для .NET, который никогда не пользовался бы Visual Studio. Это по-настоящему мощный и надежный инструмент программирования. Помимо хорошо известной функциональности, он предлагает средства для мониторинга и анализа памяти.

- Открытие файла дампа памяти и анализ использования объектов в нем (рис. 3.34), включая статистику, отдельные объекты и ссылки между ними.
- Динамическое профилирование. Нас, конечно, интересует инструмент Memory Usage (Использование памяти), но имеются также CPU Usage (анализ использования ЦП) и GPU Usage (анализ использования GPU) (рис. 3.35). Этот инструмент дает картину текущего потребления памяти и состояния сборки мусора. В любой момент можно сделать снимок состояния вашего приложения и на его основе получить полную статистику управляемых объектов.

Visual Studio не располагает такими развитыми средствами диагностики, как другие перечисленные ниже программы. Но у нее есть иное несомненное преимущество – с высокой вероятностью эта программа у вас уже есть.

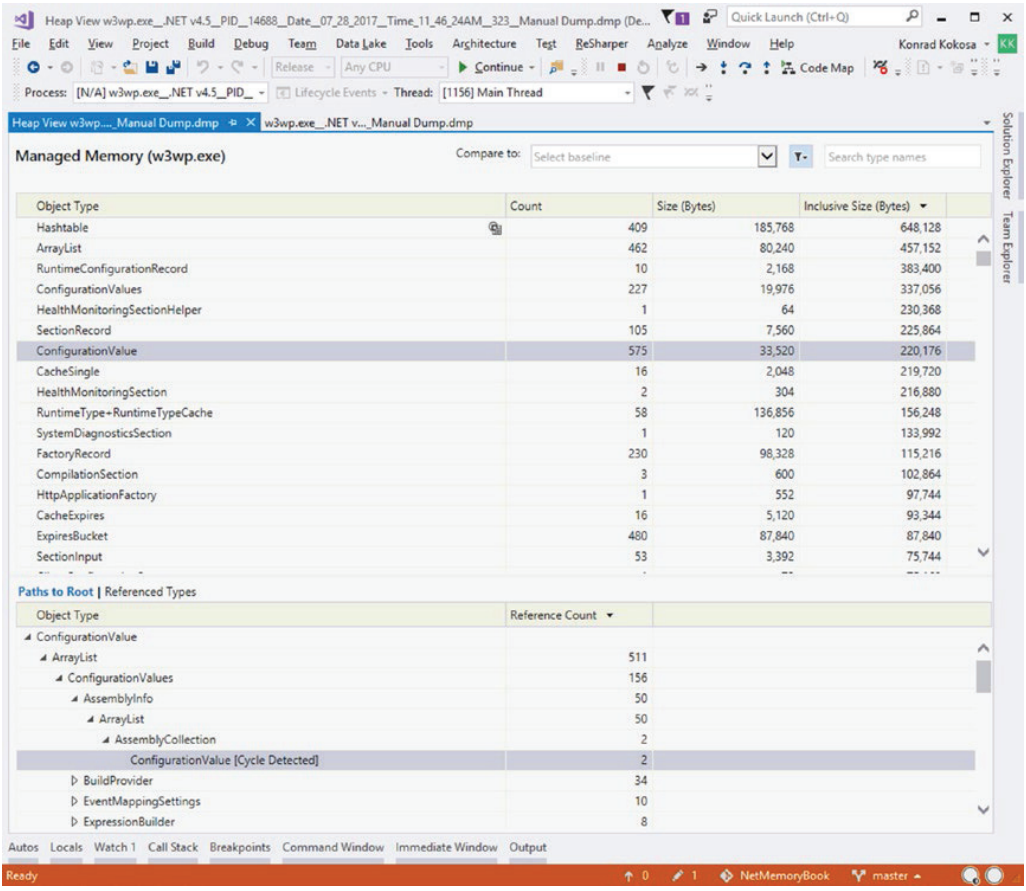


Рис. 3.34 ❖ Просмотр снимка состояния приложения в Visual Studio

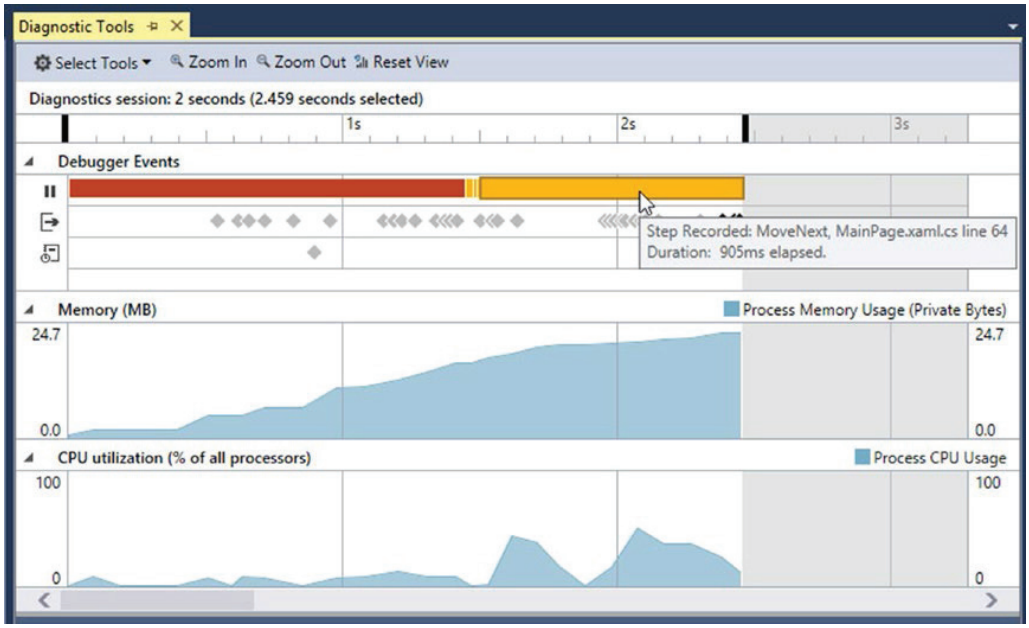


Рис. 3.35 ❖ Представление текущего состояния в Visual Studio

Scitech .NET Memory Profiler

Программа компании Scitech – один из специализированных инструментов для анализа .NET. Она предлагает широкий набор средств для просмотра состояния объектов, включая разделение по поколениям, достижимость объектов и прочее. Ее можно использовать для отображения очень сложных графов ссылок.

В каждом представлении имеются разнообразные фильтры, с помощью которых можно значительно сузить область исследования. Например, всего в два щелчка можно найти все интернированные строки (см. главу 4) в поколении 2. Интерфейс тщательно продуман, так что с программой легко сразу начать работу. Во многих местах приложение уведомляет нас (с помощью значков и всплывающих подсказок) о возможных проблемах, например о том, что слишком много строк-дубликатов, или о количестве закрепленных объектов. В то же время интерфейс нельзя назвать примитивным, он допускает углубленный анализ ситуации (рис. 3.36 и 3.37).

Внутри своей программы мы можем использовать .NET Memory Profiler API для изучения того, как используется память, или для обнаружения утечек. Бесплатная командная утилита NmpCore позволяет запускать диагностические сеансы, в т. ч. в условиях промышленной работы приложения. Впоследствии их можно проанализировать в .NET Memory Profiler.

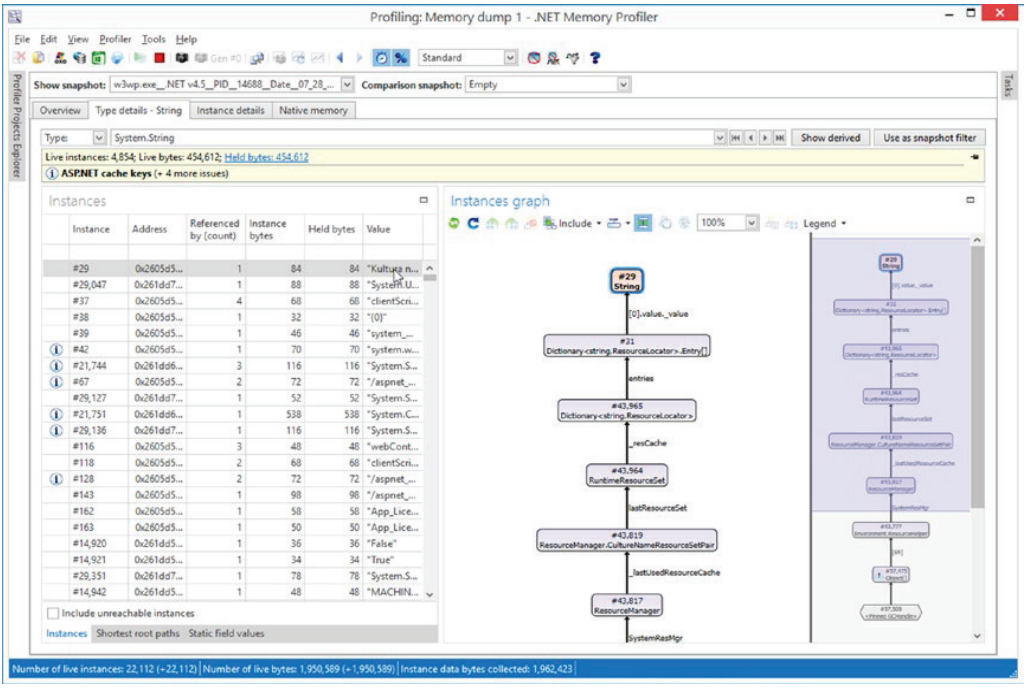


Рис. 3.36 ❖ Снимок состояния памяти с графами ссылок в .NET Memory Profiler

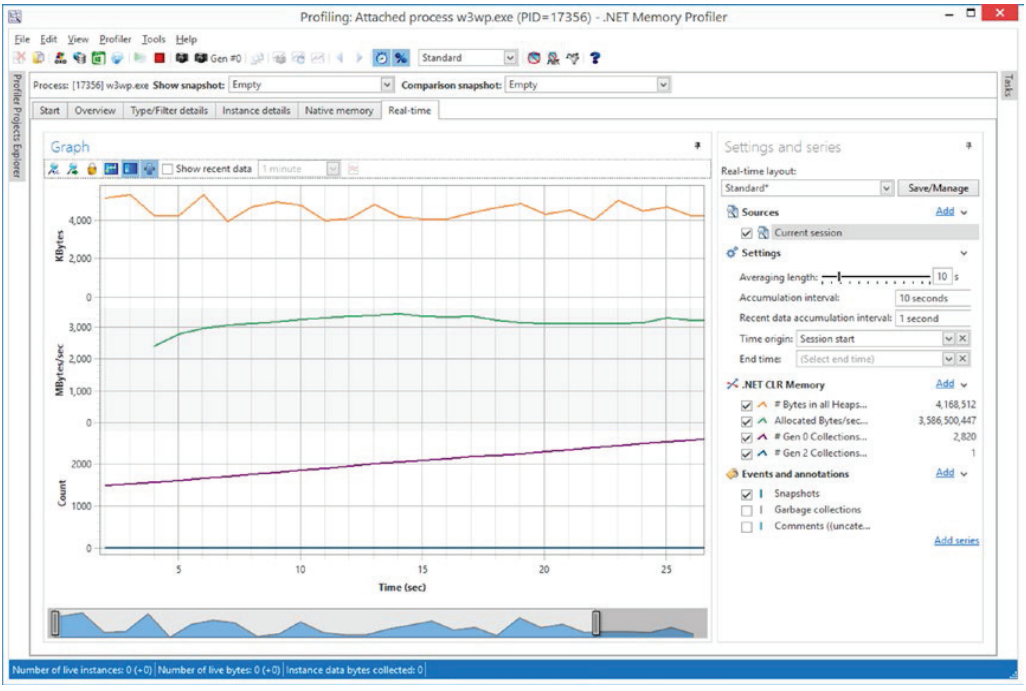


Рис. 3.37 ❖ Представление текущего состояния памяти в .NET Memory Profiler

JetBrains DotMemory

Компания JetBrains известна в мире .NET своим инструментом ReSharper. Но она выпускает также великолепные продукты для профилирования ЦП (dotTrace) и памяти (dotMemory). Нас, конечно, больше интересует вторая. dotMemory предназначена для профилирования работающего приложения, но может также анализировать дампы памяти. Кроме того, есть возможность профилировать приложения на удаленной машине, что может быть полезно, когда анализа в среде разработки недостаточно.

По сравнению с .NET Memory Profiler интерфейс dotMemory проще (что, впрочем, можно рассматривать как преимущество). Многие виды анализа предлагаются на главной странице интерфейса, так что мы получаем ответ, прежде чем успели задать вопрос (рис. 3.38 и 3.39).

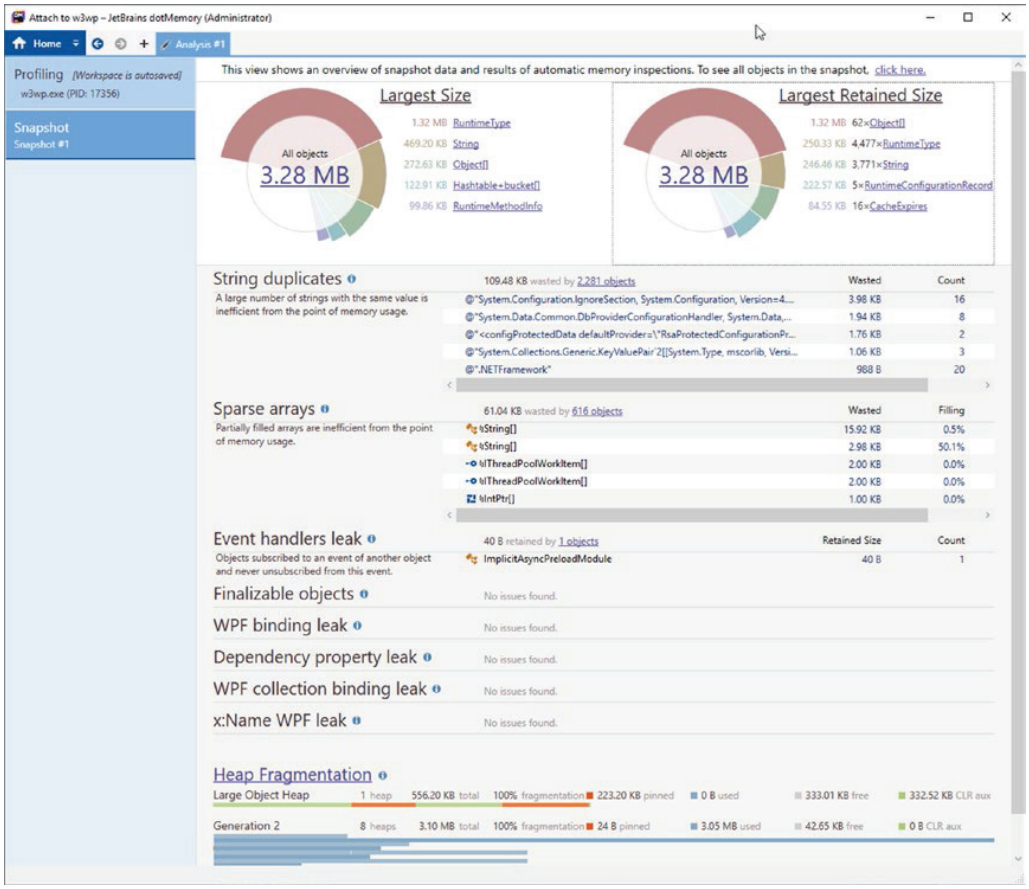


Рис. 3.38 ❖ Краткая сводка состояния памяти в JetBrains DotMemory

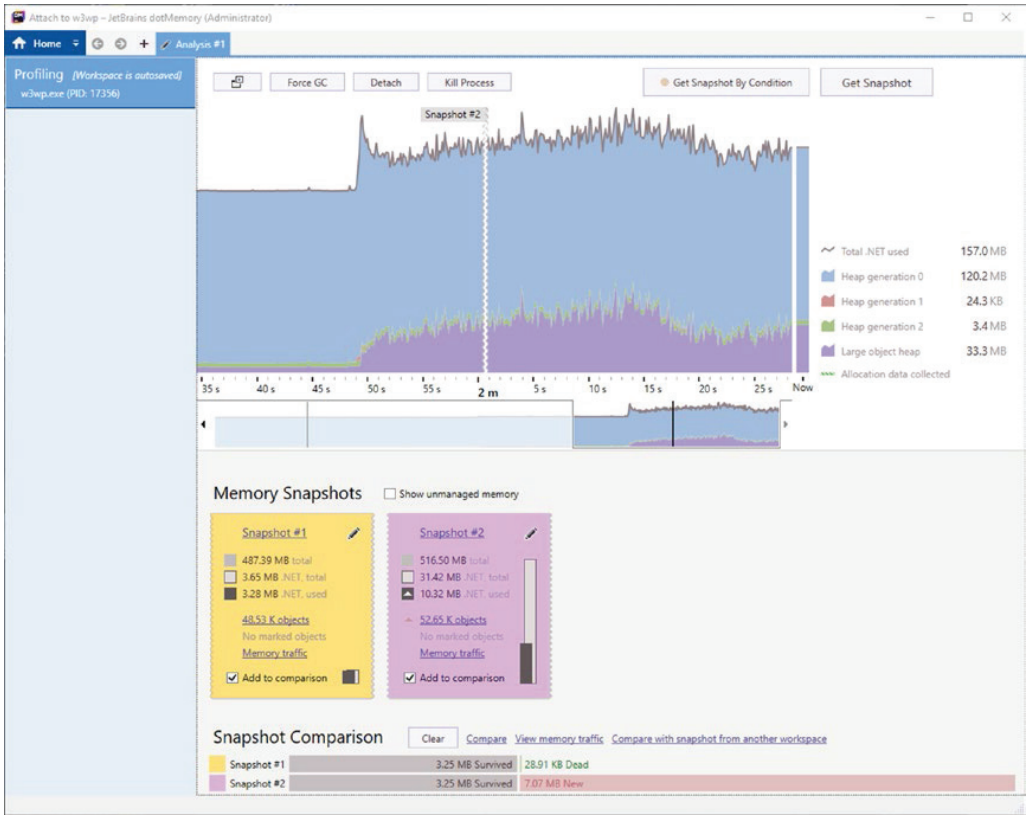


Рис. 3.39 ❖ Представление текущего состояния памяти в JetBrains DotMemory

DotMemory предлагает ряд интересных визуализаций, в т. ч. фрагментации кучи. Кроме того, можно быстро узнать, у каких объектов наибольший полный размер.

Упомянем еще два родственных инструмента. Программа dotMemory Unit позволяет выполнять модульные тесты с учетом потребления памяти. Ее можно включить в Visual Studio как часть библиотеки модульного тестирования или в процесс непрерывной интеграции. Второй инструмент – расширение Heap Allocations Viewer вышеупомянутого расширения ReSharper для Visual Studio. Оно поддерживает статический анализ кода, обнаруживая нежелательные скрытые выделения памяти (подробнее мы поговорим об этом в главе 5).

RedGate ANTS Memory Profiler

Эта программа компании RedGate была одним из первых продуктов подобного типа, с которыми я имел дело. Общее впечатление от работы во многом похоже на инструмент от JetBrains. Она проста в использовании, не перегружена опциями и стремится ответить на вопросы еще до того, как пользователь их задал. Во время работы над книгой в ней можно было выполнять динамическое профилирование, но она не умела анализировать дампы памяти (рис. 3.40 и 3.41).

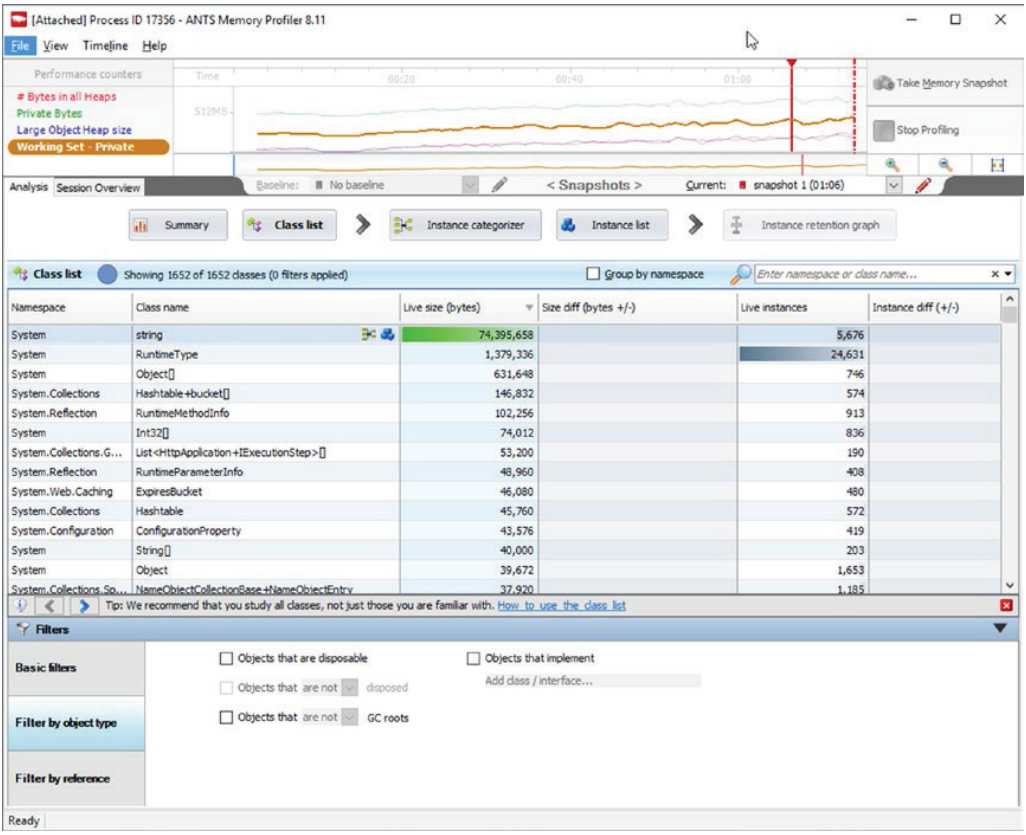


Рис. 3.40 ❖ Краткая сводка состояния памяти в ANTS Memory Profiler

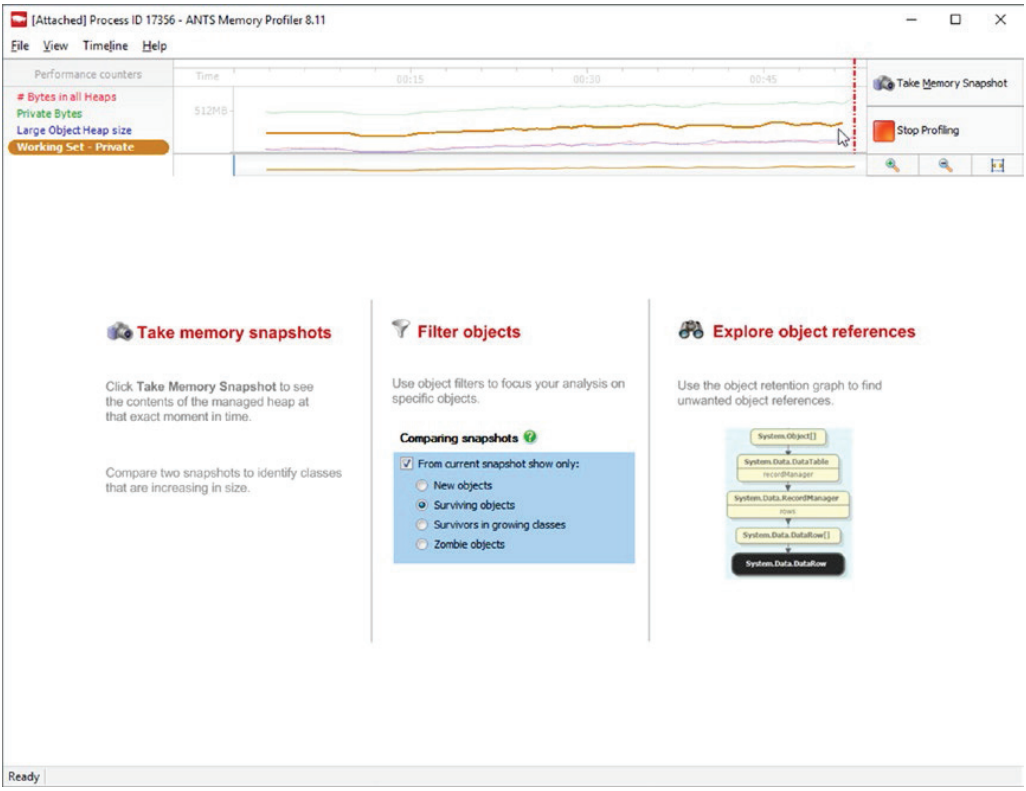


Рис. 3.41 ❖ Представление текущего состояния памяти в ANTS Memory Profiler

Intel VTune Amplifier и AMD CodeAnalyst Performance Analyzer

Эти инструменты умеют выполнять типичные функции профилирования кода и памяти, но в первую очередь предназначены для профилирования кода на аппаратном уровне, поскольку разработаны производителями оборудования. Два инструмента, упомянутых в заголовке, предлагаются компаниями AMD и Intel на платной основе. Они предлагают гораздо более глубокий анализ, далеко выходящий за пределы стандартного профилирования, позволяющий узнать, какие методы работают дольше всего. Можно получать информацию от аппаратных счетчиков, встроенных в оборудование (процессор, графическую карту) и сообщающих о его внутреннем поведении: использовании кеша и памяти, простаивании конвейера и многом другом.

Разработчикам приложений для .NET такие детали обычно неинтересны. Однако они могут оказаться очень кстати во время тонкой настройки приложения, особенно для оптимизации критических путей и коротких циклов, выполняемых миллионы раз.

На самом деле только такие низкоуровневые инструменты способны ясно указать на проблемы типа ложного разделения, упомянутого в главе 2. Взгляните на результаты выборочного анализа программы из листинга 2.6 в главе 2, произведенного в программе Intel VTune Amplifier (рис. 3.42). Ясно видно, что имеет мес-

то какая-то проблема – код очень интенсивно работает с памятью, показано, что имеет место 100 % конкурентных операций доступа (Contested Accesses).

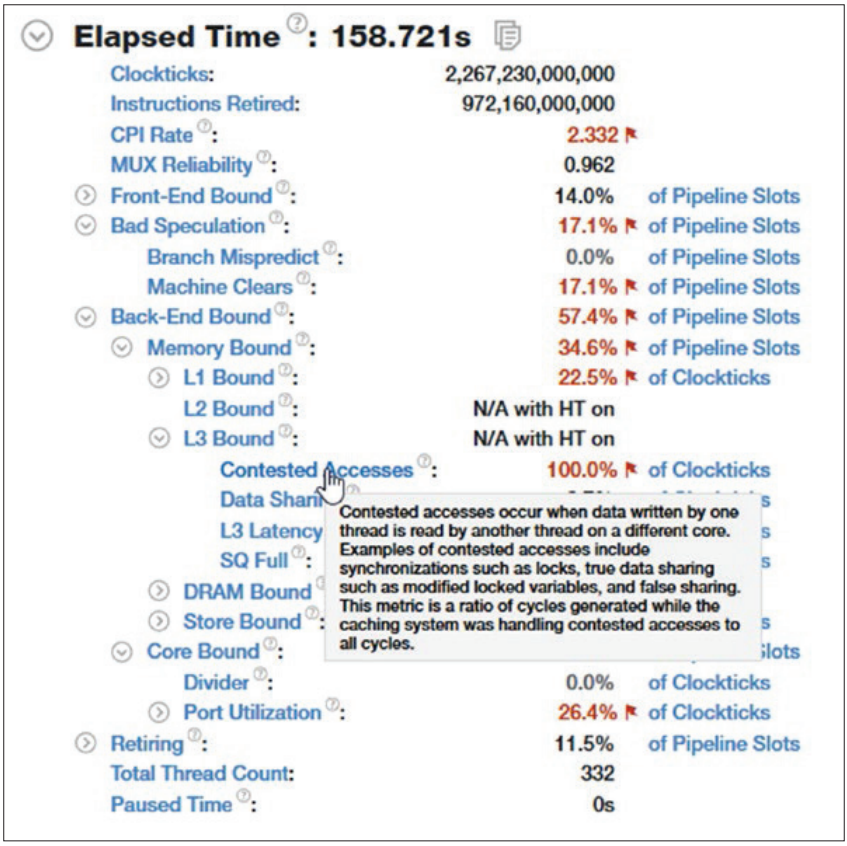


Рис. 3.42 ❖ Результаты выполнения Intel VTune Amplifier – сводный отчет

Поскольку подобные инструменты отслеживают аппаратные счетчики на самом низком уровне, можно даже получить статистику по каждой строке программы и выяснить, в чем истинный корень зла. Для программы из листинга 2.6 такой анализ действительно указывает на источник конкурентного доступа. Понятно, что поскольку за .NET-приложением стоит исполняемый платформенный код (благодаря JIT-компилятору, о котором речь пойдет в главе 4), VTune указывает на конкретные строки ассемблерного кода. Но если мы хорошо понимаем работу JIT-компилятора и ассемблерный код Intel в целом, то сможем сопоставить эти строки со строками нашего кода на языке .NET. Так, в рассматриваемом случае источником проблемы являются две команды (рис. 3.43):

- проверка размера массива (первая выделенная строка);
- доступ к старым данным счетчика (вторая выделенная строка).

Address	Assembly	Clockticks	Locators			
			Back-End Bound			
			Memory Bound			
			L3 Bound			
			Contested Accesses	Data S...	L3 L...	SQ F...
0x7ff8cff94f36	**Block 1:**					
0x7ff8cff94f36	mov rdx, 0x1ffd91f2830	840,000,000	0.0%	0.0%	0.0%	0.0%
0x7ff8cff94f40	mov rdx, qword ptr [rdx]	1,610,000,000	0.0%	0.0%	0.0%	0.0%
0x7ff8cff94f43	mov rcx, rdx	6,930,000,000	0.0%	0.0%	0.0%	0.0%
0x7ff8cff94f46	mov r8d, dword ptr [rdx+0x8]	2,345,000,000	50.0%	1.9%	34.6%	0.0%
0x7ff8cff94f4a	cmp esi, r8d	152,320,000, ...	0.0%	0.0%	0.0%	0.0%
0x7ff8cff94f4d	jnb 0x7ff8cff94f6a					
0x7ff8cff94f4f	**Block 2:**					
0x7ff8cff94f4f	mov edx, dword ptr [rdx+rax*4+0x10]	2,555,000,000	50.0%	2.9%	0.0%	0.0%
0x7ff8cff94f53	inc edx	60,585,000,0 ...	0.0%	0.0%	0.0%	0.0%
0x7ff8cff94f55	mov dword ptr [rcx+rax*4+0x10], edx	5,950,000,000	0.0%	0.0%	0.0%	0.0%
0x7ff8cff94f59	inc edi	4,865,000,000	0.0%	0.0%	0.0%	0.0%
0x7ff8cff94f5b	cmp edi, 0x5f5e100	1,750,000,000	0.0%	0.0%	0.0%	0.0%
0x7ff8cff94f61	jl 0x7ff8cff94f36 <Block 1>	0	0.0%	0.0%	0.0%	0.0%

Рис. 3.43 ❖ Результаты выполнения Intel VTune Amplifier – ассемблерный код

Таким образом, для использования подобных инструментов нужно разбираться в низкоуровневых предметах: оборудовании, среде выполнения .NET и даже языке ассемблера. Стоит также отметить, что имеются версии обоих инструментов как для Windows, так и для Linux.

Dynatrace и AppDynamics

Помимо многочисленных инструментов, предназначенных исключительно для управления памятью в .NET, имеется ряд высокоуровневых программ для мониторинга производительности приложения вообще. Они позволяют составить детальное представление о работе приложения и особенно хорошо подходят для промышленного или предшествующего ему этапа жизненного цикла приложения. Поскольку управление памятью – важный аспект .NET-приложений, в инструментах, поддерживающих эту платформу, есть удобные средства для изучения того, как приложение использует память.

Упомянутые в заголовке инструменты из категории *управления производительностью приложения* (Application Performance Management – APM) от двух ведущих производителей – отличные примеры такого подхода. Непрерывный мониторинг приложений для поиска проблем и оценки их влияния на конечного пользователя даже более ценен, чем самые изощренные средства, работающие только на компьютере разработчика, поскольку они не вступают в контакт с реальностью и не испытывают нагрузки, создаваемой пользователями.

Среда Linux

В идеале все сказанное в предыдущем разделе должно быть повторено и в контексте ОС Linux. Но на самом деле в 2018 году .NET для Linux все еще не отличается зрелостью. Только-только начали появляться первые развертывания на этой платформе. Поэтому и разработка для нее лишь зарождается. А раз так, то существует гигантское различие в уровне знаний и практических приемов по сравнению с Windows. В Windows, как мы видели, имеется много разнообразных инструментов, бесплатных и коммерческих. В Linux выбор очень небогатый. Нет ни стандартных процедур, ни даже по-настоящему опытных специалистов. Мы ступаем по девственной территории.

Краткий обзор

ОС Linux возникла и развивается в результате беспрецедентных усилий бесчисленных разработчиков из сообщества ПО с открытым исходным кодом. В отличие от Windows, она не была изначально спроектирована и реализована одной компанией. Неудивительно, что в некоторых областях заметно отсутствие стандартизации. Одна из них – приложения для мониторинга, которые нас особенно интересуют. Механизмов много, некоторые постепенно теряют популярность, другие только набирают. Поэтому инфраструктура мониторинга в Linux далеко не так однородна, как в Windows.

Не существует общепринятого стандарта трассировки и диагностики, который присутствовал бы во всех дистрибутивах и в ядре системы. При переносе CoreCLR в среду Linux необходимо было решить, какой механизм использовать. Это решение подробно разобрано в документации CoreCLR по адресу https://github.com/dotnet/coreclr/blob/master/Documentation/coding-guidelines/cross-platform-performance-and-eventing.md. Рассматривались и другие механизмы, например SystemTap, DTrace4Linux, FTrace и расширенный фильтр пакетов Extended Berkeley Packet Filter (eBPF).

В настоящее время на разных уровнях используются следующие механизмы:

- .NET-приложение – как и в Windows, мы можем использовать библиотеку EventSource или любую другую, поддерживающую протоколирование в файлы и другие «стоки»;
- среда выполнения .NET Core – генерирует события *LTTng* (*Linux Tracing Toolkit Next Generation*);
- API операционной системы и само ядро – генерирует *события производительности*, perf_events.

В общем, для получения полного представления о работе процесса CoreCLR в Linux следует использовать сочетание двух механизмов:

- perf_events – предоставляет различные данные, собираемые от оборудования и программного обеспечения (включая библиотеки ОС и само ядро). Сюда входят общесистемные измерения: выборочные данные о работе ЦП, о контекстных переключениях, об использовании памяти;
- LTTng – трассировка событий в пользовательском режиме, но с использованием модулей ядра и буферов, находящихся в ядре. Механизм генерирует строго типизированные события и в этом смысле очень похож на Event

Tracing for Windows (ETW). К сожалению, по умолчанию не поддерживается сбор стеков вызовов (программу необходимо перекомпилировать для включения или выключения этой возможности, что для универсального каркаса, каким является CoreCLR, неприемлемо). Используются те же имена событий, что в Windows.

Если механизм perf_events общесистемный, то LTTng можно подключить к отдельным процессам.

В табл. 3.3 перечислены сходства и различия механизмов трассировки в Windows и Linux.

Таблица 3.3. Сравнение механизмов трассировки в Windows и Linux

Аспект	Windows	Linux
Статическая трассировка		
Режим ядра	ETW Kernel Logger	perf_events, BCC
Пользовательский режим	Поставщики ETW Providers, счетчики производительности	LTTng
Определение	Манифест ETW	Определение точки трассировки LTTng
Общесистемный	Да	Нет
Динамическая трассировка		
	Отсутствует	perf_events SystemTrap BCC

Самое заметное отличие – отсутствие механизма динамической трассировки в Windows. Под динамической трассировкой понимается возможность включить или выключить отслеживание вызовов одной функции в работающем приложении.

Perfcollect

Простейший способ получить данные трассировки – воспользоваться официальным bash-скриптом `perfcollect`, а затем проанализировать собранные данные с помощью Perfview в Windows. Но у этого подхода есть несколько недостатков. Главный из них – довольно ограниченные результаты анализа в PerfView; имеется только неформатированный список событий. Второй, не столь удручающий, – необходимость иметь Windows… чтобы анализировать данные, собранные в Linux.

Чтобы приступить к мониторингу своего приложения для .NET Core, следуйте официальным инструкциям по адресу https://github.com/dotnet/coreclr/blob/master/Documentation/project-docs/linux-performance-tracing.md. Это нетрудно. Нужно скачать скрипт `perfcollect` из репозитория CoreCLR на Github по адресу http://aka.ms/perfcollect. Затем выполните команду `sudo ./perfcollect install`, которая установит perf_event и LTTng на ваш компьютер. Чтобы начать сеанс трассировки, нужно экспортировать две переменные окружения (первая разрешает генерацию *карт декодирования*, необходимых, чтобы декодировать символы в записанных трассировках; они будут помещены в файл `/tmp/perf-PID.map`). Требуемые команды показаны в листинге 3.10.

Листинг 3.10 ❖ Установка переменных среды, необходимых для мониторинга CoreCLR

```
> export COMPlus_PerfMapEnabled=1
> export COMPlus_EnableEventLog=1
> sudo ./perfcollect collect sampleTrace [-pid <PID>] [-threadtime]
```

После остановки сеанса будет создан ZIP-файл, содержащий собранные данные. Что же делает скрипт `perfcollect`? Если коротко, то он управляет сеансами и готовит результирующий файл:

- настраивает сеанс LTTng:
  - задает контекст, состоящий из имени процесса procname, идентификатора процесса vpid и идентификатора потока vtid;
  - по умолчанию собирает все события из группы `DotNetRuntime:*` и `DotNetRuntimePrivate:*` (полный список и доступные параметры приведены в самом скрипте);
- запускает сеанс LTTng;
- запускает сеанс perf, в котором собираются отсчеты ЦП с интервалом 1 мс (с частотой 999 Гц);
- подготавливает результирующий ZIP-файл с необходимыми данными:
  - подкаталог `lttngTrace` содержит записанные трассировки LTTng;
  - главный каталог содержит:
    - все файлы `perf.map`, созданные на протяжении сеанса;
    - все файлы символов, сгенерированные для платформенных образов (AOT/NGEN) с помощью программы `crossgen`;
    - все данные о производительности и соответствующие журналы;
  - подкаталог `debuginfo` – содержит отладочную информацию (файлы символов) для всех остальных модулей.

Записав сеанс, мы можем просмотреть его с помощью скрипта `perfcollect` (листинг 3.11).

Листинг 3.11 ❖ Просмотр данных, собранных perfcollect

```
> sudo ./perfcollect view <tracefile>
> sudo ./perfcollect view <tracefile> -viewer lttng
```

Первая команда отображает данные о производительности в виде дерева вызовов, вторая выводит текстовую распечатку всех событий LTTng без какой-либо интерпретации.

Можно, конечно, вручную управлять сеансом LTTng (самостоятельно делать то, что делает скрипт `perfcollect`), чтобы иметь полный контроль над сеансом и записанными событиями (листинг 3.12).

Листинг 3.12 ❖ Ручное управление сеансом LTTng

```
> lttng create sample_trace
> lttng add-context --userspace --type procname // or vpid, vtid
> lttng enable-event --userspace --tracepoint DotNetRuntime:Exception*
> lttng enable-event --userspace --tracepoint DotNetRuntime:GC*
> lttng start
> lttng stop
> lttng destroy
```

Аналогично можно вручную управлять механизмом perf_events для создания сеанса perf (листинг 3.13).

Листинг 3.13 ❖ Ручное управление сеансом perf

```
> perf record -g -F 999 --pid=<PID> -e cpu-clock
```

Эта команда запускает сеанс с записью графа вызовов (флаг -g) с частотой 999 Гц, т. е. с интервалом 1 мс (флаг -F 999).

Trace Compass

На странице руководства для этой программы написано: «Eclipse Trace Compass – приложение с открытым исходным кодом для просмотра и анализа журналов и трассировок любого типа. Его задача – создавать представления, графики, метрики и прочее, чтобы помочь получить полезные сведения из трассировок, так чтобы они были бы в большей степени ориентированы на пользователя и более информативны, чем гигантские текстовые дампы».

Из многих поддерживаемых форматов для нас наиболее важен общий формат *CTF* (*Common Trace Format*), в котором события генерируются механизмом LTTng, используемым в CoreCLR. Trace Compass выглядит как гибрид PerfView и Windows Performance Analyzer – если вы с ними работали, то понимаете, что я хочу сказать. Это мощный инструмент, позволяющий делать очень многое. Но, к сожалению, он, как и обе вышеупомянутые программы, очень труден для изучения. Конфигурационных параметров так много, что не понятно, с чего начинать. Если вам неинтересна диагностика Linux или вы просто не хотите тратить время на чтение довольно подробного описания того, как адаптировать Trace Compass к своим потребностям, то можете пропустить оставшуюся часть этого раздела.

Открытие файла

Получив файл, сформированный `perfcollect`, распакуйте его в какой-нибудь каталог. Интересующие нас данные LTTng находятся в подкаталоге lttngTrace, а точнее в подкаталоге с именем, устроенным по следующему образцу: `lttngTrace\auto-20170801-103533\ust\uid\1000\64-bit`. Чтобы открыть его в Trace Compass, выполните команду **File ➤ Open Trace...** (Файл ➤ Открыть трассировку) и выберите файл метаданных. Показанное на рис. 3.44 представление по умолчанию состоит из двух частей: список всех событий (закладка «64-bit» в нашем примере) и гистограмма распределения событий по времени.

Как видите, на вкладке событий для каждого события имеются дополнительные поля (в т. ч. `context._vpid` и `context._vtid` – идентификаторы процесса и потока, в котором было сгенерировано событие). Для поиска и фильтрации в этом представлении предназначена первая строка. С другой стороны, гистограмма позволяет лишь оценить количество событий в каждом временном интервале и в этом смысле не очень полезна. Мы можем закрыть ее, как и другие вкладки: Control, Control Flow, Resources, Properties и Bookmarks. Останутся только вкладки Project Explorer, Statistics и вкладки трассировки. Но такое представление не особенно интересно, и вот тут-то начинается сложный процесс настройки.

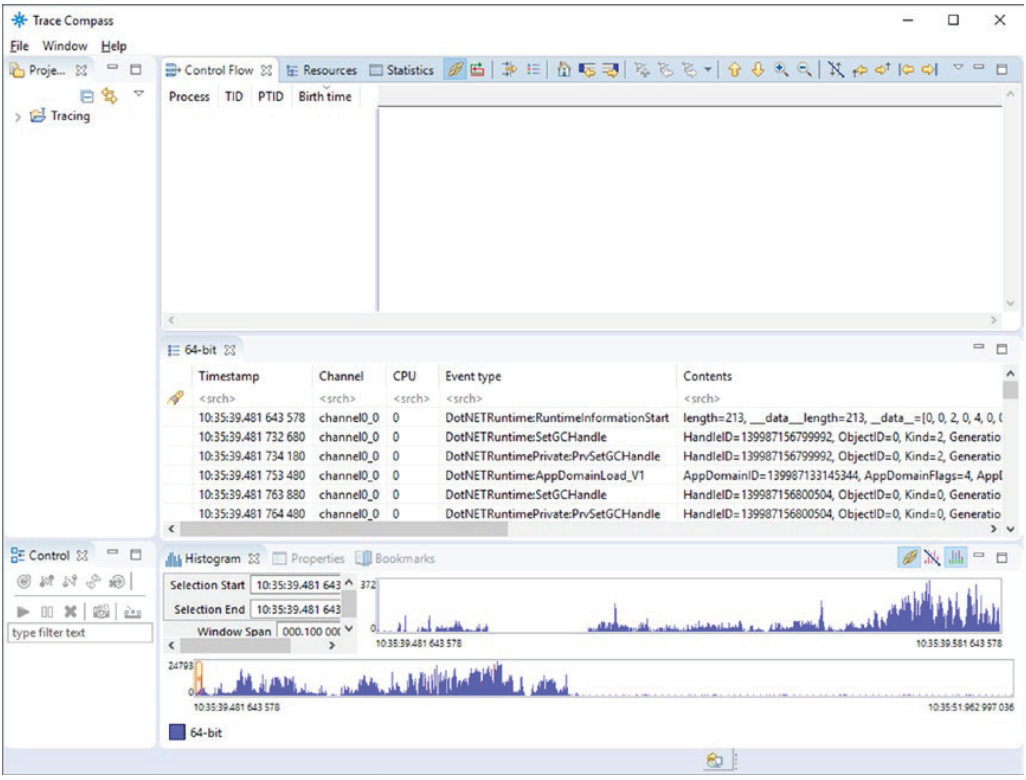


Рис. 3.44 ❖ Eclipse Trace Compass – представление трассировки LTTng по умолчанию

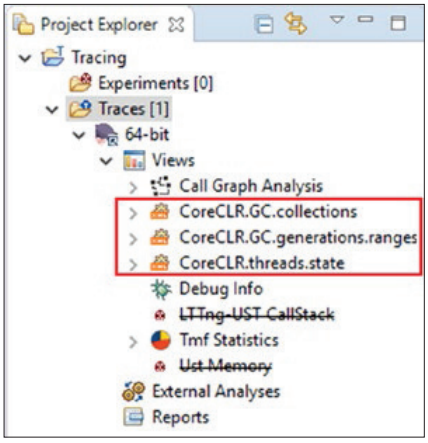


Рис. 3.45 ❖ Eclipse Trace Compass – три пользовательских представления трассы LTTng

Для этого мы просто откроем файл, содержащий представления, подготовленные для этой книги, а затем я объясню, как было создано каждое и для чего оно служит. Чтобы сделать это, закройте текущую трассировку, выбрав команду **Clear** (Очистить) из контекстного меню раздела **Tracing ➤ Traces** на вкладке **Project Explorer**. Скачайте файл `coreclr_analyses.xml`, прилагаемый к этой книге, и сохраните его где-нибудь. Затем выберите из того же контекстного меню команду **Manage XML analyses...**. В появившемся окне выберите **Import** и укажите на только что скачанный файл. Затем откройте ту же самую трассировку еще раз. Ниже элемента **Tracing ➤ Traces ➤ 64-bit ➤ Views** должны появиться три строки (рис. 3.45).

Раскрыв любое из этих представлений, вы увидите возможные дополнительные представления. Двойной щелчок по любому из них добавит его в главное представление.

Все эти представления основаны на функции Trace Compass, которая называется *анализ, управляемый данными* http://archive.eclipse.org/tracecompass/doc/stable/org.eclipse.tracecompass.doc.user/Data-driven-analysis.html#Data_driven_analysis. Она позволяет различными способами задать интерпретацию последовательности событий с помощью специальных XML-файлов.

CoreCLR.GC.collections

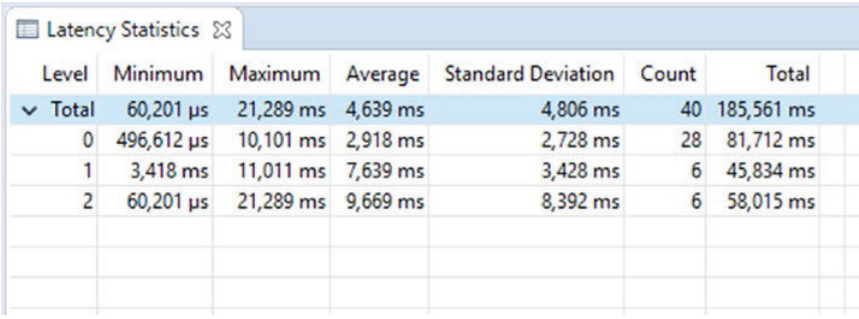
Начнем с простейшего пользовательского представления. Оно основано на простом паттерне сопоставления событий начала и конца сборки мусора. Каждая такая пара генерирует «сегмент» в терминологии Trace Compass, т. е. временной интервал, имеющий имя и, возможно, некоторые атрибуты. Такой анализ в Trace Compass выполняется с помощью конечного автомата, описывающего интересные для нас переходы (реакции на последующие события) и связанные с ними действия. В листинге 3.14 показано сокращенное описание такого анализа (для простоты я опустил часть, в которой сопоставляются начало и конец одной и той же сборки мусора).

Листинг 3.14 ❖ Фрагменты пользовательского представления CoreCLR.GC.collections для Trace Compass

```
<pattern version="0" id="CoreCLR.GC.state">
 ...
 <patternHandler initial="gcsegments">
  <action id="gc_starting">
   <stateChange>
    <stateAttribute type="constant" value="#CurrentScenario" />
    <stateAttribute type="constant" value="Generation" />
    <stateValue type="eventField" value="Depth"/>
   </stateChange>
  </action>
  <action id="gc_ending">
   <segment>
    <segType>
     <segName>
      <stateValue type="query">
       <stateAttribute type="constant" value="#CurrentScenario" />
       <stateAttribute type="constant" value="Generation" />
      </stateValue>
     </segName>
    </segType>
   </segment>
  </action>
  <fsm id="gcsegments" initial="state_before_gc">
   <state id="state_before_gc">
    <transition event="DotNETRuntime:GCStart_V2"
     target="state_during_gc" action="gc_starting"
     saveStoredFields="true" />
   </state>
   <state id="state_during_gc">
    <transition event="DotNETRuntime:GCEnd_V1"
      target="state_after_gc" action="gc_ending"
```

```
        cond="count_condition" saveStoredFields="true"
        clearStoredFields="true" />
    </state>
    <final id="state_after_gc" />
   </fsm>
  </patternHandler>
</pattern>
```

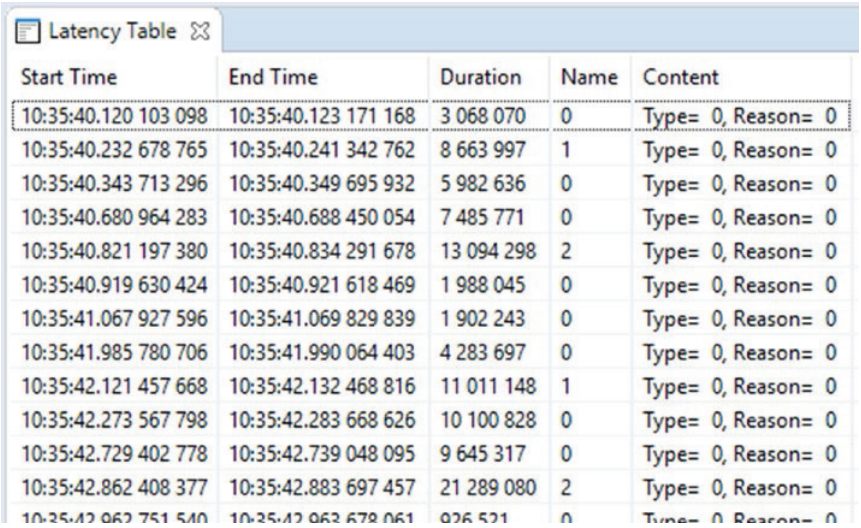
Имя каждого сегмента соответствует поколению, в котором была выполнена сборка мусора (раздел segName в приведенном выше описании). Таким образом, представления, сгенерированные в результате этого анализа, включают список всех сборок мусора по поколениям и их статистику (рис. 3.46 и 3.47) – продолжительность сегмента называется задержкой (latency).



Latency Statistics

Level	Minimum	Maximum	Average	Standard Deviation	Count	Total
Total	60,201 µs	21,289 ms	4,639 ms	4,806 ms	40	185,561 ms
0	496,612 µs	10,101 ms	2,918 ms	2,728 ms	28	81,712 ms
1	3,418 ms	11,011 ms	7,639 ms	3,428 ms	6	45,834 ms
2	60,201 µs	21,289 ms	9,669 ms	8,392 ms	6	58,015 ms

Рис. 3.46 ❖ Eclipse Trace Compass – статистика всех сборок мусора на протяжении записанной трассировки. В столбце Level указан номер поколения



Latency Table

Start Time	End Time	Duration	Name	Content
10:35:40.120 103 098	10:35:40.123 171 168	3 068 070	0	Type= 0, Reason= 0
10:35:40.232 678 765	10:35:40.241 342 762	8 663 997	1	Type= 0, Reason= 0
10:35:40.343 713 296	10:35:40.349 695 932	5 982 636	0	Type= 0, Reason= 0
10:35:40.680 964 283	10:35:40.688 450 054	7 485 771	0	Type= 0, Reason= 0
10:35:40.821 197 380	10:35:40.834 291 678	13 094 298	2	Type= 0, Reason= 0
10:35:40.919 630 424	10:35:40.921 618 469	1 988 045	0	Type= 0, Reason= 0
10:35:41.067 927 596	10:35:41.069 829 839	1 902 243	0	Type= 0, Reason= 0
10:35:41.985 780 706	10:35:41.990 064 403	4 283 697	0	Type= 0, Reason= 0
10:35:42.121 457 668	10:35:42.132 468 816	11 011 148	1	Type= 0, Reason= 0
10:35:42.273 567 798	10:35:42.283 668 626	10 100 828	0	Type= 0, Reason= 0
10:35:42.729 402 778	10:35:42.739 048 095	9 645 317	0	Type= 0, Reason= 0
10:35:42.862 408 377	10:35:42.883 697 457	21 289 080	2	Type= 0, Reason= 0
10:35:42.962 751 540	10:35:42.963 678 061	926 521	0	Type= 0, Reason= 0

Рис. 3.47 ❖ Eclipse Trace Compass – статистика всех сборок мусора на протяжении записанной трассировки. Включены дополнительные параметры Type и Reason

Это означает, что в нашей демонстрационной трассе было по шесть сборок мусора в двух поколениях и в среднем они занимали по 10 мс. Дополнительные поля Type и Reason присутствуют в событии GCStart_V2 (они еще не документированы, но имеются также в событии GCStart_V1, детали см. в https://docs.microsoft.com/en-us/dotnet/framework/performance/garbage-collection-etw-events#gcstartv1-event).

CoreCLR.threads.state

Это самое сложное из созданных мной пользовательских представлений. В нем применяется еще одна интересная функция Trace Compass – создание аналога диаграмм Ганта по собранным данным. Для открытия дважды щелкните мышью по элементу CoreCLR.threads.state.view, который расположен ниже CoreCLR.threads.state. В листинге 3.15 приведено начало определения конечного автомата.

Листинг 3.15 ❖ Фрагменты пользовательского представления CoreCLR.threads.state для Trace Compass

```
<patternHandler initial="thread">
 <test id="thread_condition">
  <if>
   <condition>
    <stateValue type="eventField" value="context._vtid"/>
    <stateValue type="query">
     <stateAttribute type="constant" value="#CurrentScenario" />
     <stateAttribute type="constant" value="ThreadId" />
    </stateValue>
   </condition>
  </if>
 </test>
 ...
 <action id="on_thread_restarting_begin">
  <stateChange>
   <stateAttribute type="constant" value="#CurrentScenario" />
   <stateAttribute type="constant" value="Status" />
   <stateValue type="int" value="11"/>
  </stateChange>
 </action>
 ...
 <fsm id="thread" initial="state_before_thread" consuming="false">
  <state id="state_before_thread">
   <transition event="DotNETRuntime:ThreadCreated"
    target="state_normal_thread" action="on_thread_starting" />
  </state>
  <state id="state_normal_thread">
   <transition event="DotNETRuntime:ThreadTerminated"
    target="state_dead_thread" action="on_thread_ending"
    cond="thread_condition" />
   <transition event="DotNETRuntime:GCSuspendEEBegin_V1"
    target="state_suspending_thread" action="on_thread_suspending_begin" />
   <transition event="DotNETRuntimePrivate:BGCBegin"
    target="state_during_bgc_nonconcurrent" action="on_bgc_
    starting_nonconcurrent" cond="thread_condition" />
```

```
  </state>
  <state id="state_during_gc">
    <transition event="DotNETRuntime:GCEnd_V1" target="state_normal_thread"
      action="on_gc_ending" cond="gc_thread_condition" />
    <transition event="DotNETRuntimePrivate:BGCBegin"
      target="state_during_gc" action="on_bgc_starting_global" />
  </state>
  ...
</patternHandler>
```

Такой относительно сложный конечный автомат реагирует на отдельные события CoreCLR (главным образом связанные с GC) изменением состояния одного из так называемых «сценариев». В данном случае сценарий соответствует одному потоку, в силу условия `thread_condition`. Иными словами, событие чаще всего изменяет состояние только одного выбранного потока, назначенного данному сценарию. Для некоторых событий это не так, например для события `GCSuspendEEBegin_V1`, которое влияет на все текущие управляемые потоки. Действия, связанные с каждым из этих событий (реакции на события), в основном изменяют поле `Status` данного сценария, значением которого является число. Позже это представление интерпретируется компонентом `timeGraphView`, показанным в листинге 3.16.

Листинг 3.16 ❖ Определение компонента timeGraphView, отображающего CoreCLR. threads.state

```
<timeGraphView id="CoreCLR.threads.state.view">
  <head>
    <analysis id="CoreCLR.threads.state" />
    <label value="CoreCLR.threads.state.view" />
  </head>

  <definedValue name="USER THREAD" value="0" color="#CCCCCC"/>
  <definedValue name="GC THREAD" value="1" color="#D6F0FF"/>
  <definedValue name="FINALIZER THREAD" value="2" color="#118811"/>
  <definedValue name="THREADPOOL THREAD" value="4" color="#A0A0A0"/>
  <definedValue name="GCWORK" value="8" color="#0000FF"/>
  <definedValue name="SUSPENDING" value="9" color="#8C5656"/>
  <definedValue name="RESTARTING" value="11" color="#758C56"/>
  <definedValue name="GCPREPARE" value="12" color="#A38A8A"/>
  <definedValue name="BGCWORK NONCONCURRENT" value="16" color="#00A4FC"/>
  <definedValue name="BGCWORK CONCURRENT" value="17" color="#000099"/>

  <entry path="scenarios/*">
    <display type="self" />
    <name type="self" />
    <entry path="*">
      <display type="constant" value="Status" />
      <name type="constant" value="ThreadId" />
    </entry>
  </entry>
</timeGraphView>
```

Этот компонент визуализирует каждый сценарий в отдельной строке, что дает нам по одной строке для каждого потока, окрашенной в цвет, соответствующий

текущему состоянию потока. В качестве имени строки используется идентификатор потока ThreadID. В результате получается симпатичное представление состояния приложения (рис. 3.48). А после увеличения будут показаны полезные детали одного выполнения GC (рис. 3.49).

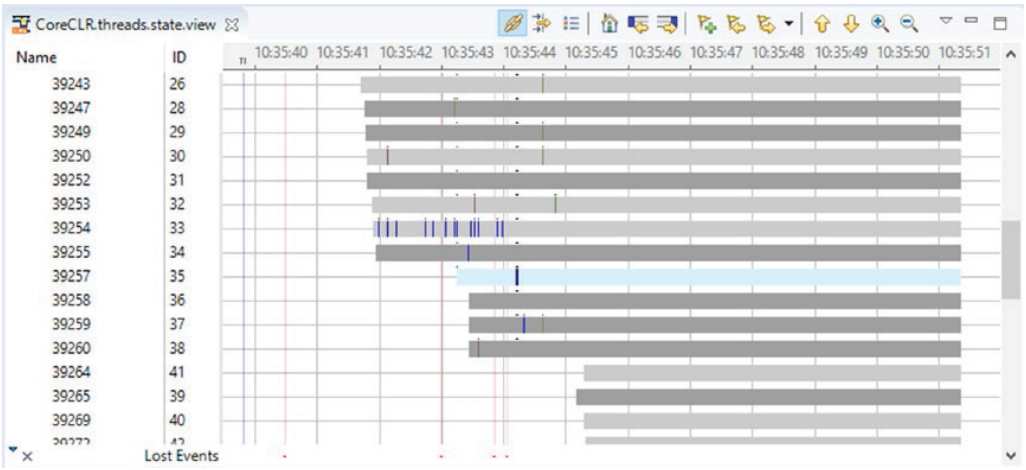


Рис. 3.48 ❖ Eclipse Trace Compass – общее представление потоков

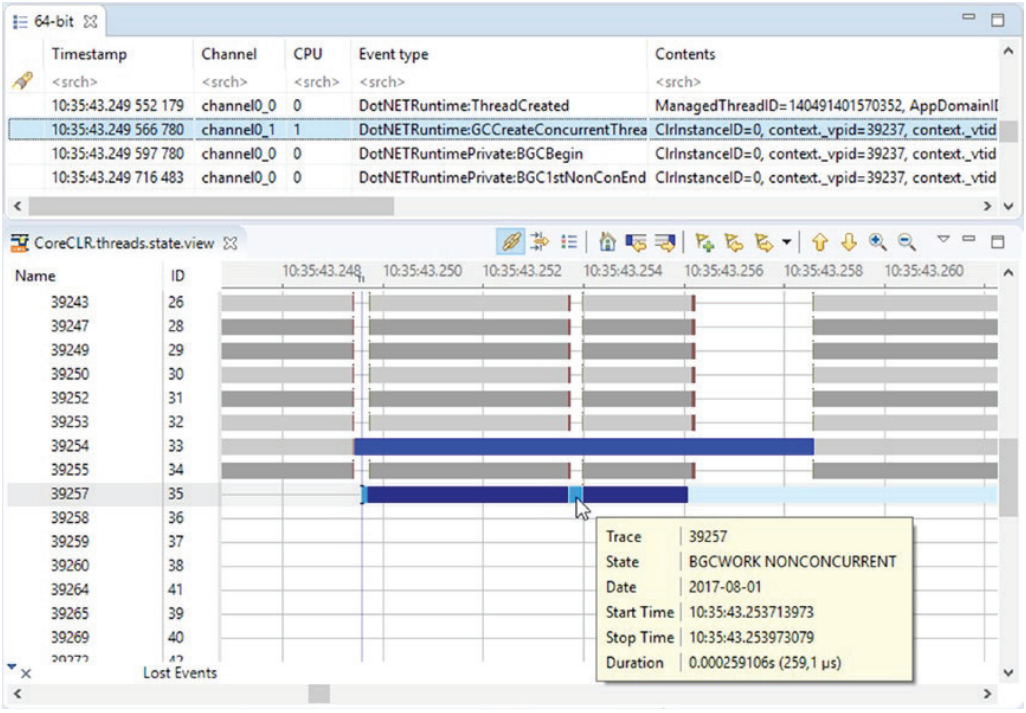


Рис. 3.49 ❖ Eclipse Trace Compass –
одна фоновая сборка мусора, происходившая в конкурентном потоке

В примерах выше показаны детали одной сборки мусора в поколении 2 – был создан фоновый поток GC для выполнения тех частей процесса, которые не допускают конкурентной работы (всё это подробно объясняется в главе 11).

CoreCLR.GC.generations.ranges

Последний способ – создать так называемые *XY-графики* (подобнее см. документ http://archive.eclipse.org/tracecompass/doc/stable/org.eclipse.tracecompass.doc.user/Data-driven-analysis.html#Defining_an_XML_XY_chart) по данным, взятым из событий. Разумеется, очень хочется визуализировать все виды измеряемых метрик, например размеры поколений и прочее. Для этого особенно полезно событие `GCGenerationRange`, генерируемое для каждого поколения в конце каждой сборки мусора (рис. 3.50).

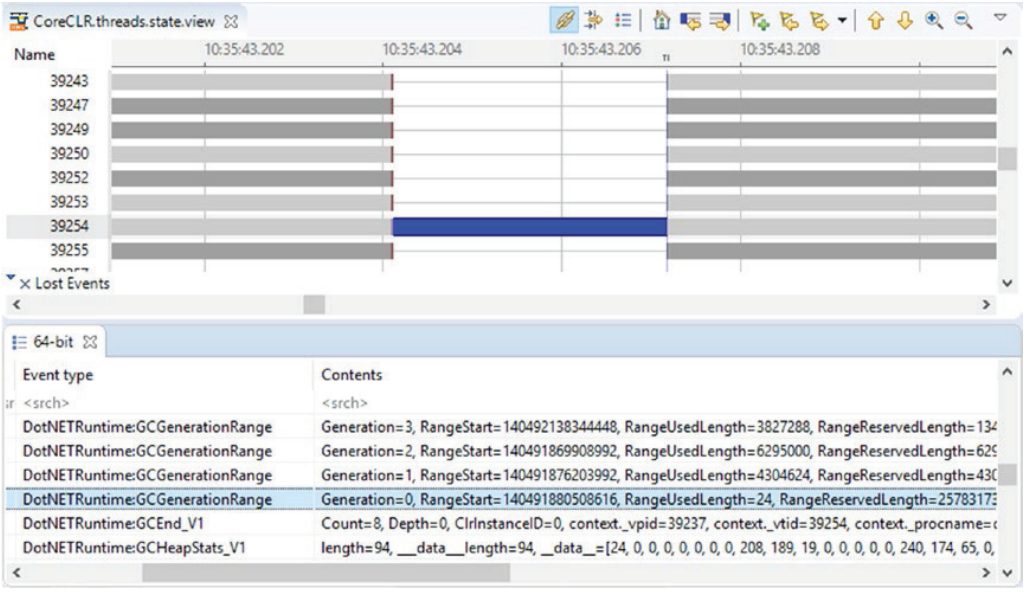


Рис. 3.50 ❖ Eclipse Trace Compass – события DotNETRuntime:GCGenerationRange, генерируемые в конце сборки мусора

Для визуализации размеров поколений можно взять поля `Generation`, `RangeUsedLength` и `RangeReservedLength`. Такой анализ основан на более простом механизме и не требует создания отдельного конечного автомата. Это просто обработчик, реагирующий на конкретное событие (листинг 3.17).

Листинг 3.17 ❖ Определение пользовательского анализа CoreCLR.GC.generations.ranges для программы Trace Compass и соответствующего ему представления

```
<stateProvider version="0" id="CoreCLR.GC.statistics">
  <head>
    <traceType id="org.eclipse.linuxtools.lttng2.ust.tracetype" />
    <label value="CoreCLR.GC.generations.ranges" />
```

```
  </head>
  <eventHandler eventName="DotNETRuntime:GCGenerationRange">
    <stateChange>
      <stateAttribute type="constant" value="Generations" />
      <stateAttribute type="eventField" value="Generation" />
      <stateValue type="eventField" value="RangeUsedLength" forcedType="long"/>
    </stateChange>
  </eventHandler>
</stateProvider>

<xyView id="CoreCLR.GC.statistics.view">
  <head>
    <analysis id="CoreCLR.GC.statistics" />
    <label value="CoreCLR.GC.statistics.view" />
  </head>

  <entry path="Generations/*">
    <display type="self" />
  </entry>
</xyView>
```

Мы получаем наглядное представление того, как размеры поколений изменяются со временем. Это может быть очень полезно для анализа (рис. 3.51).

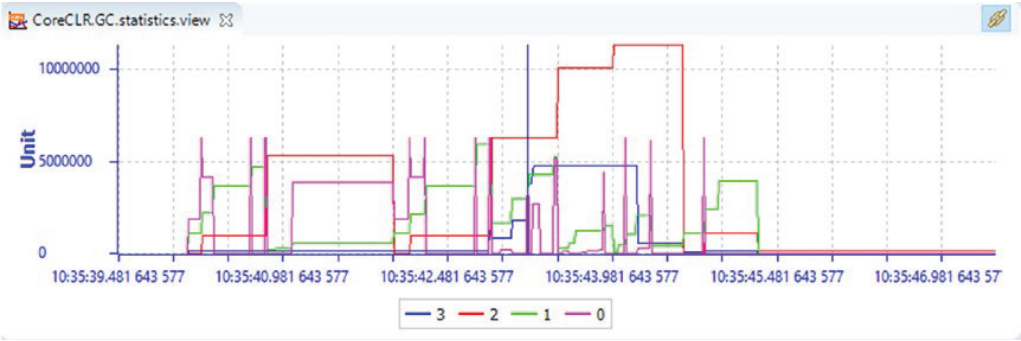


Рис. 3.51 ❖ Eclipse Trace Compass – визуализация зависимости размеров поколений от времени в виде XY-графика

ПРИМЕЧАНИЕ Существует еще одно весьма интересное событие, `DotNETRuntime: GCHeapStats_V1`, но, к сожалению, его полезная нагрузка представлена в виде массива байтов, поэтому воспользоваться им невозможно.

Окончательные результаты

Все это позволяет настроить Trace Compass для достаточно комфортного анализа собранных трассировок (рис. 3.52). Конечно, еще осталось много работы, но и такой анализ дает возможность сделать кое-какие предварительные выводы: как часто и почему выполняется сборка мусора и как изменяется со временем потребление памяти. А просмотр списка событий поможет составить представление о деталях.

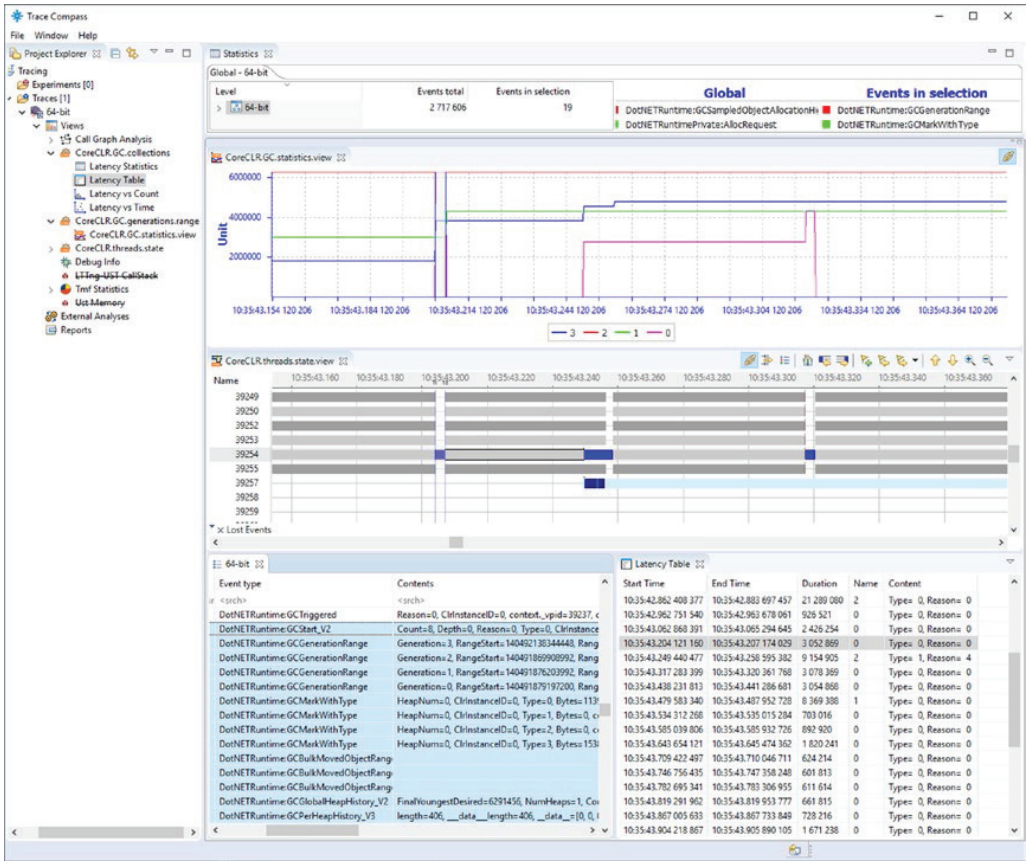


Рис. 3.52 ❖ Eclipse Trace Compass – собранные вместе пользовательские представления для анализа CoreCLR

Дампы памяти

Получение дампа памяти приложения для .NET Core концептуально производится так же, как для любой другой программы в Linux. Чтобы выгрузить дамп, выполните команду gcore, входящую в состав комплекта инструментов gdb (отладчик из проекта GNU).

Листинг 3.18 ❖ Получение дампа памяти процесса

```
$ gcore <PID>
```

Это очень похоже на использование программы Procdump в Windows.

Что касается анализа дампа, то в настоящее время для этого применяется в основном отладочное расширение SOS, поставляемое в составе самой среды выполнения .NET Core. Чтобы начать анализ, откройте файл дампа в отладчике lldb, загрузите подключаемый модуль SOS и сообщите отладчику, где находится среда выполнения CoreCLR, – с помощью команды `setclrpath` (листинг 3.19).

Листинг 3.19 ❖ Загрузка дампа памяти в lldb и задание параметров

```
> lldb --core ./path.to.coreListing 3-20.
(lldb) plugin load /usr/share/dotnet/shared/Microsoft.NETCore.App/2.0.0/
libsosplugin.soListing 3-21.
(lldb) setclrpath /usr/share/dotnet/shared/Microsoft.NETCore.App/2.0.0
```

После этого можно использовать все команды SOS, как в WinDbg.

> **ПРИМЕЧАНИЕ** Отладчик lldb основан на llvm и может считаться совершенно новой средой отладки, никак не связанной с gdb.

Резюме

В этой довольно длинной главе мы рассмотрели различные инструменты, полезные в контексте управления памятью в .NET – как для диагностики, так и для мониторинга. Обзор по необходимости получился кратким, мы не могли вдаваться в детали каждого инструмента. Но, несмотря на это, глава разрослась до впечатляющего размера. В Windows есть масса разных инструментов, в Linux немного меньше. Большинство из них – не простенькие программки, а руководства по ним – отдельные книги солидного объема. Я настоятельно рекомендую пользоваться этими инструментами в повседневной работе и рассматривать приведенный в данной главе список лишь как отправную точку для дальнейших исследований. Скачайте их и попробуйте ими воспользоваться. Конечно, какие-то вам понравятся больше, а какие-то – меньше.

Чтобы облегчить вам жизнь, в табл. 3.4 и 3.5 приведена краткая сводка рассмотренных выше инструментов.

Таблица 3.4. Сводка относящихся к .NET инструментов для Windows

Инструмент	Назначение	Плюсы и минусы
Системный монитор	Средство просмотра счетчиков производительности. Собирает и показывает данные счетчиков	+ простота использования + низкие накладные расходы – иногда вводит в заблуждение
Windows Performance Toolkit	Собирает данные ETW и позволяет их визуально анализировать. Ориентирован в основном на анализ самой Windows и драйверов	+ очень мощный + низкие накладные расходы при разумном использовании – сложен в изучении
Perfview	Собирает данные ETW и позволяет их анализировать с помощью множества предопределенных представлений. Ориентирован в основном на анализ проблем, связанных с .NET	+ очень мощный для .NET + низкие накладные расходы при разумном использовании – сложен в изучении
ProcDump, DebugDiag	Создает дамп памяти процесса по запросу или при наступлении заданных условий	+ простота использования
WinDbg	Отладка управляемого и платформенного кода. Благодаря дополнительным расширениям предлагает развитые средства анализа	+ допускает изучение процесса на очень низком уровне – очень сложен в изучении – для повседневных целей слишком низкоуровневое средство

Таблица 3.4 *(окончание)*

Инструмент	Назначение	Плюсы и минусы
dnSpy	Редактирование и отладка сборок .NET, даже когда исходный код недоступен	
BenchmarkDotNet	Позволяет писать тесты производительности для изучения времени работы и потребления ресурсов	
Visual Studio (платная)	Хорошо известная IDE общего назначения. Включает средства отладки, профилирования и анализа дампа памяти	+ хорошо знакома разработчикам для .NET – средства профилирования и анализа дампов ограничены по сравнению с другими специализированными коммерческими программами
Scitech .NET Memory Profiler (платная) JetBrains DotMemory (платная) RedGate ANTS Memory Profiler (платная)	Инструменты, предназначенные для анализа памяти .NET	+ простой пользовательский интерфейс + много готовых видов анализа – платные
Intel VTune Amplifier and AMD CodeAnalyst Performance Analyzer	Профилирование платформенного и управляемого кода на аппаратном уровне, включая использование кешей, конвейера ЦП и многое другое	+ очень глубокий анализ производительности оборудования – пожалуй, слишком подробно для типичных ситуаций – требует хотя бы базовых знаний об оборудовании
Dynatrace и Appdynamics (платные)	Инструменты непрерывного мониторинга, позволяющие производить сбор данных о .NET (зависит от инструмента)	+ глубокий анализ работающих приложений – платные

Таблица 3.5. Сводка относящихся к .NET инструментов для Linux

Инструмент	Назначение	Плюсы и минусы
Perfcollect	Скрипт для сбора и просмотра событий LTTng и perf	+ помощь в настройке сеансов LTTng и perf + очень ограниченные возможности анализа
Trace Compass	Сбор и визуальный анализ событий LTTng. Предназначен для анализа общего вида, но может быть настроен на события, относящиеся к .NET	+ возможен очень низкоуровневый анализ процесса – сложен в изучении
Intel VTune Amplifier and AMD CodeAnalyst Performance Analyzer	См. описание этих инструментов для Windows	

Некоторые инструменты, описанные в данной главе, будут использованы в книге для иллюстрации обсуждаемых вопросов. Поэтому так важно было представить их заранее. Впоследствии у нас еще будет возможность попрактиковаться в различных ситуациях. Поскольку мы пока не познакомились с деталями GC в .NET, в этой главе было рано рассматривать конкретные проблемы диагностики. Далее

мы будем пользоваться также несколькими более простыми инструментами, которые здесь не упомянуты, поскольку это заняло бы слишком много места.

Первые три главы были общим введением в управление памятью. В главе 1 мы узнали о теоретических основаниях предмета. В главе 2 рассмотрели аппаратные и системные детали. А сейчас заканчиваем это обширное введение рассказом об инструментах, которыми будем пользоваться. И переходим к той части, где описывается непосредственно .NET, ее внутреннее устройство и рекомендации по использованию. Приятного чтения!

Правило 5: измеряйте GC как можно раньше

Обоснование. Непрерывный мониторинг различных метрик позволяет ответить на вопрос «существует ли проблема с памятью?» сразу после запуска приложения. Более того, мы можем следить за тенденциями и своевременно обнаруживать снижение производительности. Конечно, этот принцип применим не только в контексте сборки мусора. Точно так же следует измерять общую производительность (например, время реакции), аспекты синхронизации (скажем, количество контекстных переключений) и т. д.

Как применять. Важно выработать у себя привычку измерять параметры GC как можно раньше, начиная с первого развертывания в средах разработки и тестирования и заканчивая непрерывным мониторингом в условиях промышленной эксплуатации приложения. Поскольку это скорее концептуальная установка, чем практический совет, способов ее использования на практике может быть очень много. Без сомнения, целью должен быть процесс непрерывного мониторинга использования памяти и операций сборки мусора в приложениях, желательно автоматический. Отправной точкой для создания такого процесса должны послужить другие правила, сформулированные в этой книге. Благодаря им мы знаем, что измерять и как интерпретировать результаты. Как конкретно будет выглядеть процесс, в значительной мере зависит от используемых инструментов. В Windows измерения, скорее всего, будут основаны на чтении относящихся к делу счетчиков производительности (раздел 3.2) или анализе событий ETW, записываемых в циклический буфер (раздел 3.3). В Linux требуется автоматизировать анализ событий perf_events и LTTng. Такие автоматизированные проверки можно встроить в процессы непрерывной интеграции и поставки, например после каждой сборки нового релиза продукта. В качестве абсолютного минимума следует вручную следить за выбранными показателями после каждого развертывания в производственной среде и сравнивать их с поведением, наблюдавшимся в предыдущих версиях. Что измерять? Зависит от обстоятельств и от полноты процесса мониторинга. Но я не могу представить себе хорошо продуманную систему, в которой не измерялись бы следующие характеристики приложения:

- сколько памяти потребляет процесс, и имеет ли место неконтролируемый рост со временем?
- как часто вызывается и сколько времени занимает сборка мусора, вызывает ли она заметные накладные расходы?

Глава 4

Фундаментальные основы .NET

Мы дошли еще только до четвертой главы, но уже многое узнали о различных аспектах управления памятью. Эти вопросы обсуждались в общем виде, чтобы дать теоретическое введение в предмет. Ссылки на .NET встречались редко, а ведь это основная тема книги. Пора уже это исправить. Начиная с этой главы и до конца книги .NET будет сопутствовать нам постоянно. В данной главе мы несколько расширим угол зрения, изучим некоторые внутренние механизмы и начнем погружение в темы, связанные с управлением памятью. Всячески рекомендую предварительно вооружиться знаниями, изложенными в трех предыдущих главах, хотя это и необязательно – решение за вами. Далее я буду также предполагать некоторое знакомство с языком ассемблера для платформ x86/x64, поскольку мы будем погружаться в .NET все глубже и глубже. Если вы почувствуете, что надо бы освежить знания в этой области, почитайте, например, прекрасную книгу Daniel Kusswurm «Modern X86 Assembly Language Programming» (Apress, 2014).

Если бы .NET Framework был человеком, то сейчас он ходил бы в среднюю школу и через несколько лет начал бы потихоньку готовиться к экзаменам на аттестат зрелости. Иными словами, этот продукт разрабатывается и используется уже примерно 15 лет. За это время и богатая коллекция вспомогательных библиотек, и сама среда исполнения значительно эволюционировали. Все .NET-разработчики должны хорошо знать базис: стандартную библиотеку и синтаксис языка C# – основного языка разработки в среде .NET (или других, например VB.NET, неуклонно теряющего популярность, или F#, столь же неуклонно ее набирающего). Это наш «хлеб насущный». Но с возрастом или, если хотите, с опытом часто приходит понимание, что неплохо бы знать больше. Вот и давайте расширим наши знания!

Помните, что эта книга посвящена управлению памятью, а остальные вопросы, связанные с .NET, упоминаются лишь вскользь. Так, например, не ждите подробного описания языковых средств C# или проблем многопоточности. Эти темы рассматриваются в других книгах и онлайновых материалах, недостатка в которых не ощущается.

Версии .NET

Среда .NET не столь однородна, как может показаться на первый взгляд. Чаще всего это слово ассоциируется с самым популярным вариантом .NET Framework,

который прошел длинный путь от версии 1.0 через 2.0, 3.5, 4.0 к текущей версии 4.7.2. Но, говоря о среде .NET, можно в действительности иметь в виду все богатство версий и реализаций. Важным решением, обеспечившим это богатство, стала стандартизация. С самого начала концепция .NET была основана на спецификации, которая называется *общеязыковой инфраструктурой* (Common Language Infrastructure – CLI). В этом основополагающем техническом стандарте (официально зарегистрированном как *ECMA 335* и *ISO/IEC 23271* в 2003 году) описываются понятия кода и среды выполнения, которая позволяет исполнять его на разных машинах без перекомпиляции. Я много раз буду ссылаться на него в этой главе, потому что лучшего источника истины не существует.

Возникает сильное искушение описать все компоненты CLI, включая все варианты реализации и различия между ними. Но мы ограничимся в основном тем, как все это влияет на предмет нашего интереса. А теперь рассмотрим различные варианты .NET с точки зрения управления памятью и сборки мусора.

- .NET Framework 1.0–4.7.2 – разрабатывался с 2002 года, коммерческий и наиболее зрелый продукт, который всем нам известен. Он существует уже много лет, так что ядро сборщика мусора разрабатывалось и улучшалось от версии к версии. Долгое время весь этот механизм предлагалось считать черным ящиком, хотя с появлением каждой новой версии .NET проскальзывали мимолетные упоминания о нем. Поскольку код коммерческой среды выполнения .NET Framework закрыт, о том, как работают ее компоненты, мы могли узнавать только из сообщений, исходящих от самой компании Майкрософт. Эта информация была весьма подробной, что позволяло нам понимать и диагностировать проблемы с памятью в наших приложениях. Но все равно разработчики испытывали неудовлетворенность, особенно в сравнении с открытостью исходного кода других платформ, например Java.
- Shared Source CLI (также Rotor) – выпущенная в 2002 (версия 1.0) и 2006 (версия 2.0) годах реализация среды выполнения для учебных и академических целей. Она никогда не предназначалась для промышленной эксплуатации, но позволяет познакомиться с различными деталями реализации CLR. Есть замечательная книга David Stutz, Ted Neward, Geoff Shilling «Shared Source CLI Essentials» (O'Reilly Media, 2003), в которой подробно описывается эта версия. Но, во-первых, в ней не полностью реализована «зрелая» версия .NET 2.0 Framework. А во-вторых, в некоторых местах реализация сильно отличается от настоящей CLR, и особенно, как это ни грустно, в части управления памятью. Сборщик мусора в ней очень упрощенный.
- .NET Compact Framework – «мобильная» версия .NET, оставшаяся со времен Windows CE/Mobile и Xbox 360. Сборщик мусора в ней сильно отличается от основной версии, он гораздо проще, например понятие поколения (о котором мы поговорим в следующей главе) в нем отсутствует. Впрочем, эта система в любом случае представляет лишь исторический интерес, и больше мы с ней не встретимся. Однако во время разработки данной версии был получен ценный опыт, особенно в области переноса на другие платформы – различные процессоры, на которых работали устройства под управлением Windows CE. Именно тогда зародилась концепция CoreCLR.
- Silverlight – модуль, подключаемый к веб-браузеру и позволяющий выполнять в нем обычные оконные приложения (с некоторыми ограничениями).

Поскольку Майкрософт начала работу над ней во времена .NET 2.0, в основу была положена тогдашняя версия среды выполнения. Если вы по-прежнему пользуетесь ей, имейте в виду, что к ней применимо многое из относящегося к текущей версии .NET. Только информация о среде выполнения устарела и осталась на уровне .NET 2.0. Это была среда выполнения, перенесенная на платформу OSX, которая легла в основу кода современной среды выполнения CoreCLR (.NET Core).

- .NET Core (соответствующая среда выполнения называется CoreCLR) – появление версии .NET с открытым исходным кодом многое изменило. Отныне имеется среда выполнения, готовая к промышленной эксплуатации, код которой мы можем изучать сколько угодно. И особенно важно, что код сборщика мусора – практически копия коммерческой версии. Похоже, что .NET Core постепенно будет обгонять по функциональности .NET Framework, изменения в котором будут последовательно переноситься в более современную среду. .NET Core также является официально поддерживаемым кросс-платформенным решением. Оно работает в Windows, в Linux и в MacOS.
- Windows Phone 7.x, Windows Phone 8.x и Windows 10 Mobile – эти устаревшие версии системы основывались на простой схеме управления памятью, заимствованной у .NET Compact Framework 3.7. В Windows Phone 8.x внесены значительные улучшения во внутренние механизмы среды выполнения .NET, основанные на версии .NET Framework 4.5 и унаследовавшие ее сборщик мусора.
- .NET Native – технология, позволяющая компилировать код CIL непосредственно в машинный код. Основана на облегченной среде выполнения CoreRT (прежнее название MRT). Код сборщика мусора тот же, что в .NET Core.
- .NET Micro Framework – отдельная реализация с открытым исходным кодом для небольших устройств. Самое популярное применение – устройство .NET Gadgeteer, содержащее собственную упрощенную версию сборщика мусора. Поскольку это решение нишевое, ориентированное на любителей-энтузиастов, мы не будем рассматривать его в данной книге.
- WinRT – новый способ раскрытия функциональности ОС разработчикам в виде набора API, используемого для создания приложений в стиле Metro на языках JavaScript, C++, C# и VB.NET. Предполагается, что заменит Win32. Среда написана на C++ и вообще не является реализацией .NET. Но она объектно-ориентированная и основана на формате метаданных .NET, поэтому выглядит как обычная библиотека для .NET (особенно если вызывается из .NET).
- Mono – совершенно отдельная кросс-платформенная реализация CLI с собственным механизмом управления памятью. Знакомство с ней мало что дает для понимания .NET. Однако на базе этой технологии существует по меньшей мере два очень популярных решения: Xamarin – платформа для создания мобильных приложений и Unity3D – популярный игровой движок. Памятуя о популярности этих проектов, мы иногда будем обращаться к Unity для сравнения.

Из этого перечня складывается довольно обнадеживающая картина – механизм управления памятью на всех основных используемых в настоящее время платформах .NET – .NET Framework, .NET Core и .NET Native – очень похож (если не сказать – один и тот же).

Эта книга изобилует объяснениями внутренних механизмов сборки мусора в .NET, основанными на исходном коде .NET Core 2.1. Как мы только что сказали, данный код весьма близок к основному коду в .NET Framework и к его мобильному варианту. Поэтому выводы, сделанные на основе исходного кода для .NET Core, очень ценны, а его изучение – способ получения достаточно полной информации. Начиная с этого момента все примеры исходного кода .NET по умолчанию взяты из .NET Core 2.1, если явно не оговорено противное. Я также ссылаюсь на так называемую «Книгу среды выполнения» – «Book of the runtime» – открытую документацию, которая разрабатывается параллельно с самой средой и выложена по адресу https://github.com/dotnet/coreclr/blob/master/Documentation/botr/README.md. В ней много ценной информации о реализации среды выполнения.

Нам следует знать о некоторых деталях внутреннего устройства .NET, чтобы полностью разобраться в механизме управления памятью. Мы рассмотрим эти детали, но будем опускать все, что не относится к интересующему нас контексту. Существует много других источников, в которых можно найти дополнительную информацию, в т. ч. знаменитая книга Джеффри Рихтера «CLR via C#» (Microsoft Press, 2012)[1], Sasha Goldshtein «Pro .NET Performance» (Apress, 2012)[2] или Ben Watson «Writing High-Performance .NET Code» (Ben Watson, 2014)[3].

Детали внутреннего устройства .NET

Когда мы пишем программу на C или C++, компилятор преобразует ее в исполняемый файл. Он может быть выполнен непосредственно на целевой машине, так как помимо библиотек, взаимодействующих с операционной системой, он содержит бинарный код, исполняемый непосредственно процессором.

С другой стороны, у среды выполнения .NET имеется много обязанностей, без которых невозможно решить интересующую нас задачу – выполнить написанное нами приложение. В отличие от программ на C или C++, когда мы пишем на C#, F# или любом другом .NET-совместимом языке, код компилируется в *общий промежуточный язык* – CIL (Common Intermediate Language). Затем этот код поступает на обработку *общеязыковой среде выполнения* (Common Language Runtime – CLR). CLR – это то место, где творится «управляемая магия». Выше CLR находится более общая концепция среды .NET, которая включает все стандартные библиотеки и инструментарий (поэтому существуют различные версии .NET, которые могут как включать в себя, так и не включать изменения, относящиеся к среде исполнения). У CLR есть ряд обязанностей, из которых мы перечислим основные.

- *Компиляция на лету* (*JIT-компиляция*) – ее задача состоит в том, чтобы преобразовать CIL-код в машинный. Этот способ исполнения управляемого кода по существу является умным сокрытием механизмов операционной системы – так, управление памятью включает стек для потоков, кучу и многое другое.

---

[1] *Джеффри Рихтер*. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. СПб.: Питер, 2019.

[2] *Саша Голдштейн, Дима Зурбалев, Идо Флатов.* Оптимизация приложений на платформе .Net. М.: ДМК Пресс, 2017.

[3] *Ватсон Бен.* Высокопроизводительный код на платформе .NET. СПб.: Питер, 2019.

- Система типов – отвечает за контроль типов и механизмы их совместимости. Среди прочего она включает *общую систему типов* (Common Type System – CTS) и метаданные (используемые механизмом отражения (Reflection).
- Обработка исключений – отвечает за обработку исключений на уровне как пользовательской программы, так и самой среды выполнения. Здесь используется как платформенный механизм *структурной обработки исключений* (SEH – Structured Exceptions Handling), так и механизм обработки исключений в C++.
- Управление памятью (часто упоминается как сборка мусора) – это крупная часть среды выполнения, отвечающая за управление памятью в интересах самой среды выполнения и нашего приложения. Очевидно, что одна из ее основных обязанностей – освобождение памяти, занятой уже ненужными объектами.

Мы часто разделяем эти обязанности на два основных блока:

- движок выполнения – включает большинство вышеперечисленных обязанностей среды выполнения, в частности JIT-компиляцию и обработку исключений. В стандарте ECMA-335 он называется *виртуальной исполняющей системой* (Virtual Execution System – VES) и описывается следующим образом: «*отвечает за загрузку и выполнение программ, написанных для CLI. Предоставляет службы и данные, необходимые для выполнения управляемого кода. Использует метаданные на этапе выполнения для объединения отдельно сгенерированных модулей*»;
- сборщик мусора – отвечает за управление памятью, выделение памяти для объектов и возврат более неиспользуемых областей памяти. В ECMA-335 описывается как «*процесс, посредством которого выделяется и освобождается память для управляемых данных*».

Все эти элементы работают совместно, как в хорошо отлаженной машине, состоящей из крупных и мелких механизмов. Трудно ожидать, что после удаления одного из них машина будет продолжать работать. Так и с управлением памятью. Можно говорить о механизмах управления памятью, но надо ясно понимать, что все остальные компоненты тесно связаны с ними. Например, JIT-компилятор порождает информацию о времени жизни переменных, которая затем используется сборщиком мусора. Система типов предоставляет информацию, необходимую для принятия ключевых решений, например определен ли в типе финализатор. В реализации обработки исключений нужно учитывать механизмы возврата памяти, например приостанавливать ее на время сборки мусора. Такие зависимости между различными компонентами CLR очень интересны.

Мы часто слышим выражение *управляемый код* в контексте .NET. Означает оно, что код, исполняемый средой выполнения, должен быть готов к кооперации с ней, чтобы среда могла обеспечить все, что от нее требуется. Стандарт ECMA-335 гласит:

> *управляемый код: код, который содержит достаточно информации, чтобы CLI могла предоставить базовые службы. Например, зная адрес метода внутри кода, CLI должна быть способна найти метаданные, описывающие этот метод. Она также должна иметь возможность проходить по стеку, обрабатывать исключения, сохранять и извлекать информацию, относящуюся к безопасности.*

На рис. 4.1 схематично показан процесс выполнения нашей программы средой выполнения .NET.

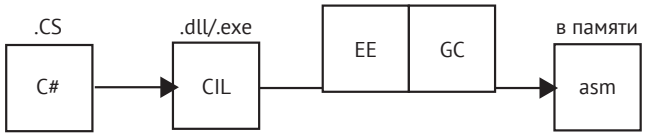


Рис. 4.1 ❖ Исходный код (текстовые файлы) компилируется в общий промежуточный язык CIL (двоичные файлы). Затем на целевой машине с установленной средой выполнения .NET этот код исполняется средой. Среда состоит из двух основных блоков: движок выполнения (EE) и сборщик мусора (GC). EE читает CIL-код из двоичных файлов и преобразует его в машинный код в памяти

Весь процесс состоит из следующих шагов.

- Мы пишем код в редакторе Visual Studio, Visual Studio Code или любом другом. В результате получается проект, состоящий из нескольких исходных файлов. Это просто текстовые файлы, содержащие код на выбранном нами языке: C#, VB.NET, F# или любом другом поддерживаемом.
- Проект компилируется с помощью подходящего компилятора, будь то компилятор, встроенный в Visual Studio (для проектов .NET Framework), или компилятор .NET Core. В результате получается набор файлов (сборок), содержащих двоичный код – команды на общем промежуточном языке. Это представление нашей программы в виде совокупности низкоуровневых команд для «виртуальной» стековой машины (см. главу 1). Могут быть и другие сборки, содержащие библиотеки, которые мы используем в нашей программе. Такой набор сборок теперь можно передать другим пользователям в виде ZIP-файла или пакета для установщика.
- Мы запускаем приложение. Очевидно, это самая важная часть, которую также можно разбить на несколько шагов:
  - для .NET Framework – исполняемый файл содержит код начальной загрузки, который загружает подходящую версию среды выполнения .NET при поддержке ОС Windows;
  - для .NET Core – многоплатформенное решение не зависит от взаимодействия с Windows. Если мы хотим запустить управляемую сборку, то должны явно вызвать подходящую команду, например `dotnet run`, находясь в каталоге с нашей программой. Это приведет к загрузке среды выполнения;
  - среда выполнения .NET загружает из файла необходимую в данный момент часть CIL-кода сборки и передает ее JIT-компилятору;
  - JIT-компилятор транслирует CIL-код в машинный код, оптимизированный для платформы, на которой он работает. Дополнительно он вставляет обращения к движку выполнения, обеспечивающие взаимодействие между нашим кодом и средой выполнения .NET;
  - начиная с этого момента ваш код исполняется как обычный неуправляемый код. Разница только в вышеупомянутом взаимодействии со средой выполнения.

Теперь самое время развеять некоторые распространенные заблуждения, относящиеся к среде .NET.

- .NET не является виртуальной машиной в обычном смысле – среда выполнения .NET не создает изолированного окружения и не имитирует какую-то конкретную архитектуру или машину. На самом деле среда выполнения .NET пользуется встроенными системными ресурсами, в т. ч. системным механизмом управления памятью с его кучей и стеком, процессами и потоками и т. д. Но она надстраивает над ними дополнительную функциональность, в частности автоматизированное управление памятью.
- Не существует единственной среды выполнения .NET, работающей на данной машине, – бинарный дистрибутив один, но он загружается и выполняется каждым запущенным .NET-приложением. Например, сборка мусора в процессе A напрямую не влияет на сборку мусора в процессе B. Очевидно, что на уровне оборудования и операционной системы какое-то разделение ресурсов существует, но среда выполнения .NET не знает ни о каких других управляемых приложениях, в которых работают их собственные экземпляры среды выполнения. На самом деле можно даже загрузить среду выполнения .NET в неуправляемое приложение (именно так и реализованы возможности CLR в SQL Server). Более того, в одном процессе может работать несколько сред выполнения .NET, хотя практической пользы от этого мало.

Разбираем пример программы

Теперь пройдем по шагам весь процесс компиляции и выполнения простого приложения Hello world (листинг 4.1), чтобы лучше понять, как работает .NET. Это позволит нам познакомиться с некоторыми базовыми концепциями, которые пригодятся позже. Каждый, кто изучал C#, наверное, узнает этот код, единственная цель которого – вывести короткий текст на консоль. Мы будем использовать его как пример, работающий под управлением среды .NET Core 2.1 в Windows. Понятно, что чересчур глубоко мы залезать не будем, потому что нас интересуют в основном вопросы, связанные с управлением памятью. Тех, кому любопытно, как среда выполнения .NET загружается, как она управляет своими типами и т. п., отсылаю к уже упомянутым книгам.

Листинг 4.1 ❖ Пример программы Hello World, написанной на C#

```
using System;
namespace HelloWorld
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Hello world!");
      }
   }
}
```

Компиляция кода в листинге 4.1 компилятором C# (Roslyn в случае использованной нами Visual Studio 2017) порождает один файл DLL, называемый `CoreCLR`.

`HelloWorld.dll`. Этот файл содержит все данные, необходимые для запуска программы. Детали можно увидеть, открыв его, например, в программе dnSpy, после чего мы сможем просмотреть различные декодированные секции файла (рис. 4.2):

- метаданные, описывающие сам файл (в терминах описания двоичного файла Windows или Linux) – в Windows это заголовки DOS и PE;
- метаданные, описывающие содержимое, относящееся к .NET, включая все объявленные в сборке типы, их методы и прочие свойства (показано как Storage Stream #0 с именем #~);
- список ссылок на другие файлы;
- бинарный поток объявленных типов и их методов на общем промежуточном языке.

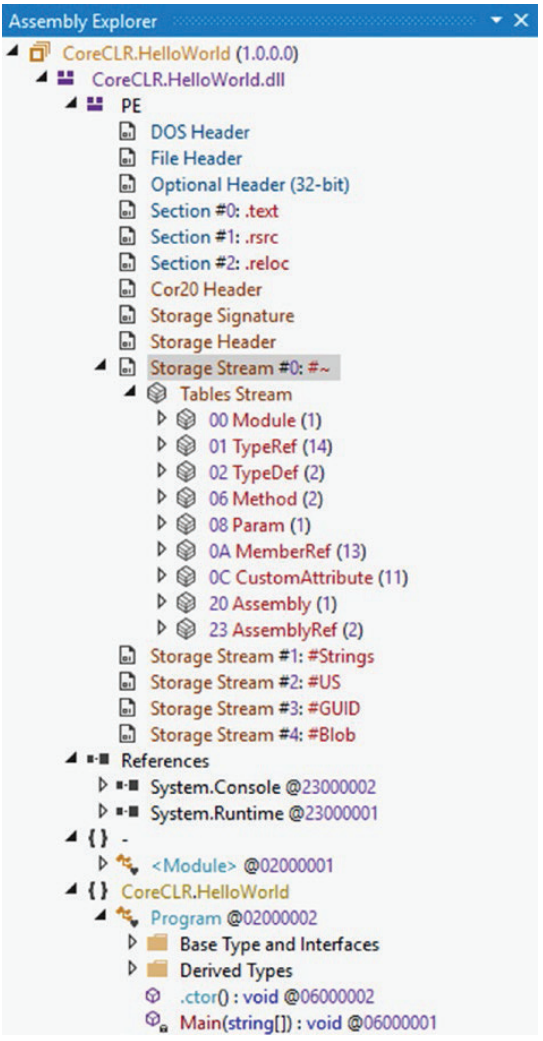


Рис. 4.2 ❖ Содержимое двоичного файла CoreCLR.HelloWorld.dll – результат компиляции программы в листинге 4.1

У каждого метода или типа имеется уникальный идентификатор – *маркер* (token), – местоположение которого в файле можно узнать из вышеупомянутых потоков метаданных. Поэтому мы можем определить участок файла, содержащий тело каждого метода. Например, чтобы просмотреть тело метода `Main`, выберите его в окне Assembly Explorer (Обозреватель сборки) и выполните команду **Show Method Body in Hex Editor** (Показать тело метода в шестнадцатеричном редакторе) из его контекстного меню (рис. 4.3).

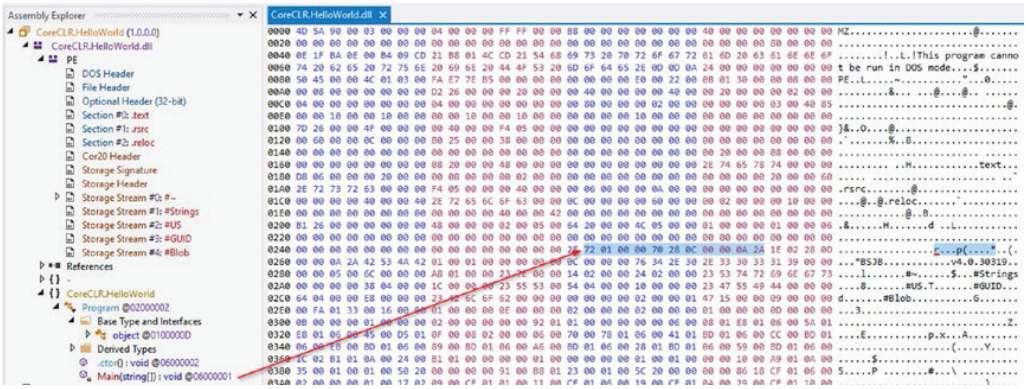


Рис. 4.3 ❖ Байты, представляющие команды метода `Program.Main` на общем промежуточном языке (стрелка добавлена, чтобы было понятнее)

Разумеется, глядя на эти байты, трудно понять, что они означают. Но мы можем декодировать CIL-код каждого метода, представив его в более понятной форме, и в этом нам поможет декомпилятор, упомянутый в главе 3. Для этого выберем метод `Main` в Assembly Explorer, а в меню dnSpy укажем в качестве языка декомпиляции IL.

Результат декомпиляции типа `Program` из файла `CoreCLR.HelloWorld.dll` показан в листинге 4.2 (конструктор для краткости опущен). В комментариях мы видим исходный байт-код команд (например, байт `2A` соответствует команде CIL `ret`), так что теперь последовательность байтов `7201000070280C00000A2A` на рис. 4.3 обретает смысл.

Изучение простого CIL-кода метода `Main` (листинг 4.2) показывает, как он компилируется в код стековой машины:

- `ldstr "Hello World!"` – ссылка на строковый литерал помещается в стек вычислений;
- `call System.Console::WriteLine` – вызывается статический метод, который берет первый аргумент из стека вычислений;
- `ret` – метод возвращает управление (никакое значение не возвращается, потому что в стеке вычислений ничего нет).

Листинг 4.2 ❖ Результат преобразования простой программы в листинге 4.1 на общий промежуточный язык. Получено с помощью программы dnSpy

```
// Маркер: 0x02000002
.class private auto ansi beforefieldinit CoreCLR.HelloWorld.Program
    extends [System.Runtime]System.Object
```

```
{
   // Маркер: 0x06000001
   .method private hidebysig static
      void Main (
         string[] args
      ) cil managed
   {
      // Размер заголовка: 1 byte
      // Размер кода: 11 (0xB) bytes
      .maxstack 8
      .entrypoint

      /* 7201000070 */ IL_0000: ldstr    "Hello World!"
      /* 280C00000A */ IL_0005: call     void [System.Console]System.
                                         Console::WriteLine(string)
      /* 2A */ IL_000A: ret
   } // окончание метода Program::Main
} // окончание класса CoreCLR.HelloWorld.Program
```

Внимательно присмотревшись к коду в листинге 4.2, вы заметите команду `.maxstack 8`, которая вроде бы относится к выполнению программы. Однако это не команда CIL. Такое описание метаданных может использоваться различными инструментами для проверки безопасности кода. `maxstack` говорит, какое максимальное количество байтов может быть выделено в стеке вследствие выполнения метода. В случае метода `Main` необходимо 8 байт для хранения ссылки на строковый литерал. Инструмент типа `PEVerify` может воспользоваться этой информацией и сопоставить ее с тем, что хочет сделать CIL-код метода. Таким образом, .NET-код допускает проверку на безопасность, и это хорошо, поскольку различные виды переполнения буфера – самая опасная угроза для программ.

Говоря о стековой машине .NET, нельзя не упомянуть о важном понятии *местоположения* (location). Для хранения различных значений, необходимых при выполнении программы, предусмотрено несколько логических местоположений:

- локальные переменные метода;
- аргументы метода;
- поле экземпляра другого значения;
- статическое поле (класса, интерфейса или модуля);
- локальный пул памяти;
- временно в стеке вычислений.

На что отображается каждое местоположение в конкретной компьютерной архитектуре, решает JIT-компилятор, чуть ниже мы рассмотрим этот вопрос.

Примечание. В экосистеме .NET в настоящее время существует несколько JIT-компиляторов:

- унаследованный x86 JIT, применявшийся в среде выполнения .NET до версии 4.5.2 и в .NET Core 1.0/1.1 для архитектуры x86 (32-разрядной);
- унаследованный x64 JIT, применявшийся в среде выполнения .NET до версии 4.5.2;
- новый RyuJIT, применяемый в .NET Core 2.0 (и более поздних версиях) и в .NET Framework 4.6 (и более поздних версиях) для 32- и 64-разрядной архитектур;
- Mono JIT для платформ x86 и x64.

Поскольку унаследованные версии постепенно заменяются, я буду говорить только о новом движке RyuJIT.

Теперь мы можем воспользоваться WinDbg, чтобы посмотреть, как JIT-компилятор скомпилировал нашу программу в случае 64-битной Windows. Конечно, нам придется запустить приложение, поскольку только после этого будет загружена среда выполнения и активирована JIT-компиляция нужных методов.

В предположении, что используется новейшая версия WinDbg, распространяемая как универсальное приложение Windows, мы можем выбрать команду **Launch executable (advanced)** (Запуск исполняемого файла (расширенный)) на панели **File** и задать следующие параметры (предполагается, что наше решение находится в папке C:\Projects):

- исполняемый файл: `C:\Program Files\dotnet\dotnet.exe`;
- аргументы: `\CoreCLR.HelloWorld.dll`;
- начальный каталог: `C:\Projects\CoreCLR.HelloWorld\bin\Release\netcoreapp2.1`.

> Многие предпочитают запускать WinDbg из командной строки. Тогда для открытия сеанса отладки можно выполнить: `windbgx C:\Program Files\dotnet\dotnet.exe C:\Projects\CoreCLR.HelloWorld\bin\x64\Release\netcoreapp2.1\ CoreCLR.HelloWorld.dll`.

После нажатия **OK** приложение Hello world будет запущено, и его выполнение будет сразу же прервано. Теперь мы можем установить точку останова, в которой программа приостановится перед завершением (после печати сообщения Hello World!). Это делается такой командой:

```
bp coreclr!EEShutDown
```

Теперь нажмите Go и немного подождите, пока выполнение дойдет до точки останова. После этого мы можем загрузить расширение SOS (см. главу 3) и найти метод `Main`, выполнив команды:

```
.loadby sos coreclr
!name2ee *!CoreCLR.HelloWorld.Program.Main
```

Вторая команда напечатает показанную ниже информацию, где говорится, что JIT-компилированный код метода `Main` находится по адресу `00007ffbca3e06b0`:

```
Module:      00007ffbca284d78
Assembly:    CoreCLR.HelloWorld.dll
Token:       0000000006000001
MethodDesc:  00007ffbca285d30
Name:        CoreCLR.HelloWorld.Program.Main(System.String[])
JITTED Code Address: 00007ffbca3e06b0
```

Мы можем воспользоваться командой `!U 00007ffbca3b0480`, чтобы посмотреть сгенерированный ассемблерный код, результат показан в листинге 4.3. Прокомментируем шаги выполнения:

- `sub rsp,28h` – сдвинуть указатель стека на 40 байт;
- `mov rcx,24D6CCA3068h` – записать адрес `24D6CCA3068h` в регистр `rcx` (это описатель строкового литерала `"Hello World!"`, используемый здесь вследствие механизма интернирования строк, который будет рассмотрен ниже);
- `mov rcx,qword ptr [rcx]` – разыменовать адрес, хранящийся в регистре `rcx`, и получить указатель на строку, содержащую наш строковый литерал;
- `call 00007ffb`ca3b0330` – вызвать статический метод `Console.WriteLine`, передав ему текст из регистра `rcx`;
- `nop`, `add rsp,28h` и `ret` – конец вызова функции.

Листинг 4.3 ❖ Машинный код, порожденный JIT-компиляцией кода в листинге 4.2

```
Normal JIT generated code
CoreCLR.HelloWorld.Program.Main(System.String[])
Begin 00007ffbca3b0480, size 1c
00007ffb`ca3b0480 4883ec28        sub     rsp,28h
00007ffb`ca3b0484 48b96830ca6c4d020000 mov rcx,24D6CCA3068h
00007ffb`ca3b048e 488b09          mov     rcx,qword ptr [rcx]
00007ffb`ca3b0491 e89afeffff      call    00007ffb`ca3b0330 (System.
Console.WriteLine(System.String), mdToken: 0000000006000083)
00007ffb`ca3b0496 90              nop
00007ffb`ca3b0497 4883c428        add     rsp,28h
00007ffb`ca3b049b c3              ret
```

Вот так простая программа на C# транслируется сначала в CIL, а потом в исполняемый код. Местоположение стека вычислений, используемое в CIL-командах `ldstr` и `call`, JIT-компилятор загрузил в регистр `rcx`. В методе `Main` не выделяется память ни в стеке, ни в куче, но имейте в виду, что какая-то память уже выделена самой средой выполнения и библиотечными сборками.

> Использовать регистры и память для вызова функций можно разными способами, но существует стандартизованный подход, называемый *соглашением о вызове*. Это соглашение определяет, как передавать аргументы, как управлять стеком в процессе вызова метода и как возвращать значение. В примерах ассемблерного кода в этой книге предполагается *соглашение о вызове Microsoft x64*. В несколько упрощенном виде, достаточном для наших целей, оно сводится к следующим правилам:
>
> – первые четыре целых и ссылочных аргумента передаются в регистрах `RCX`, `RDX`, `R8` и `R9`;
> – первые четыре аргумента с плавающей точкой передаются в регистрах `XMM0`–`XMM3`;
> – остальные аргументы передаются в стеке;
> – целые значения длиной не более 64 бит возвращаются в регистре `RAX`.
>
> Имейте в виду, что в Linux x64 соглашения о вызове иные; если хотите, можете прочитать о них в интернете.

Надеюсь, что этот очень краткий, но все же насыщенный обзор позволил вам составить представление о том, что такое среда выполнения .NET. В конечном итоге все методы JIT-компилируются в обычный исполняемый код, использующий (при необходимости) некоторые «управляемые» части среды выполнения.

Сборки и домены приложений

Основной единицей функциональности в среде .NET является *сборка*[1]. Ее можно рассматривать как упакованный CIL-код, который может быть выполнен средой выполнения .NET. Программа состоит из одной или нескольких сборок. Например, в результате компиляции кода в листинге 4.1 была создана одна сборка, представленная файлом `CoreCLR.HelloWorld.dll`. В этой программе используются

[1] Английское слово assembly (сборка) не следует путать с языком ассемблера (assembly language). Это совершенно разные понятия, по случайному совпадению обозначаемые одним словом.

и другие сборки, начиная с базовой библиотеки классов (она называется mscorlib и содержит такие важные пространства имен, как System.IO, System.Collections.Generic и т. д). Сложное .NET-приложение может состоять из большого числа сборок, содержащих наш код. В терминологии управления исходным кодом проекта существует простое соответствие – из одного проекта, входящего в состав решения, строится одна сборка. Имеется также возможность создавать динамические сборки в процессе выполнения программы (часто говорят о динамической *генерации* (emit) кода такой динамической сборки); эта функциональность находит применение во многих сериализаторах.

Иными словами, сборку можно рассматривать как единицу развертывания управляемого кода, которая в типичном случае соответствует одному DLL- или EXE-файлу (такой файл называют *модулем*).

.NET Framework дает возможность изолировать различные части управляемого кода приложения (сборки), разделив их по так называемым *доменам приложений* (обычно название сокращают до *AppDomain*, потому что так называется тип в библиотеке BCL). Такое разделение может быть желательно из соображений безопасности, надежности или версионирования. Чтобы выполнить код из сборки, ее нужно сначала загрузить в какой-нибудь домен приложения (то же самое относится и к динамически созданным сборкам).

> Между сборками и доменами приложений существует весьма сложная, но хорошо документированная связь, о которой можно прочитать по адресу https://docs.microsoft.com/en-us/dotnet/framework/appdomains/application-domains.

Чтобы уменьшить размер .NET Core, пришлось отказаться от некоторых возможностей, и одной из них были AppDomain. Они попросту слишком много весят по сравнению с предоставляемой функциональностью, да и тянут за собой довольно много. Поэтому в .NET Core нет открытого AppDomain API, относящегося к управлению доменами приложений. Однако часть отвечающего за них кода все же присутствует в CoreCLR, т. к. сама среда пользуется ими для своих целей. Разработчикам, желающим изолировать приложения для .NET Core, Майкрософт предлагает старые добрые процессы или новенькие контейнеры. Что же касается динамической загрузки сборок, то рекомендую познакомиться с новым классом `AssemblyLoadContext`.

Домены приложений попали в сферу наших интересов, потому что они влияют на структуру памяти процесса в .NET. Вообще говоря, среда выполнения может создавать несколько разных доменов приложений.

- Общий, или разделямый, домен – сюда загружается весь код, совместно используемый несколькими доменами: сборки базовой библиотеки классов, типы из пространства имен `System` и т. д.
- Системный домен – отвечает за создание и инициализацию других доменов, поскольку сюда загружаются основные компоненты среды выполнения. Здесь же хранятся интернированные строковые литералы, видимые во всем процессе (об интернировании мы будем говорить позже в этой главе).
- Домен по умолчанию (например, с именем Domain 1) – в этот домен загружается пользовательский код.
- Динамические домены – при поддержке среды выполнения .NET Framework приложение может создавать (а затем удалять) сколько угодно дополни-

тельных доменов. Например, с помощью метода `AppDomain.CreateDomain` (но, как уже было отмечено, в .NET Core эта функциональность намеренно опущена и вряд ли когда-нибудь появится).

Понятно, что в .NET Core динамические домены не создаются. За весь общий код отвечает разделяемый домен. А для всего пользовательского кода существует единственный домен по умолчанию. Системный домен физически не виден в памяти процесса, но его структуры и логика также включены.

Забираемые сборки

Загружаемые нами сборки содержат манифест, в котором указано, в каких еще сборках они нуждаются. Стандартное поведение CLR подразумевает загрузку всех требуемых сборок в главный домен приложения – тот, который будет существовать на всем протяжении работы программы. В большинстве случаев этого достаточно, но иногда требуется более точный контроль над временем жизни сборки.

- Скрипты – если мы разрешаем выполнять в своем приложении пользовательские скрипты (например, скомпилированные с помощью Roslyn API), то было бы идеально создать для такого скрипта временную сборку и удалить ее, как только необходимость в скрипте отпадет.
- Объектно-реляционное отображение (ORM) – иногда нам нужно отобразить данные из базы на объекты .NET, но не в течение всего времени работы приложения, особенно если наше приложение на короткое время подключается к разным источникам данных. В таком случае очистка предназначенных для ORM структур (для которых созданы отдельные сборки) была бы очень полезной возможностью.
- Сериализаторы – как и в предыдущем случае, может возникать необходимость в сериализации и десериализации различных сущностей (не важно, поступают они в виде файлов или HTTP-запросов), поэтому если это делается многократно, то было бы хорошо убрать ставшие ненужными сборки. Такие сборки создаются для сериализаторов из соображений производительности – типы, предназначенные для сериализации конкретных данных, создаются, чтобы не реализовывать совершенно лишний «общий» способ работы с ними.
- Подключаемые модули – наше приложение может поддерживать расширяемость путем загрузки написанных пользователями подключаемых модулей. Конечно, было бы хорошо загружать и выгружать их по мере необходимости.

В .NET Framework выгрузить сборку можно опосредованно, выгрузив весь домен приложения, в который она загружена. Так, типичный сценарий работы с пользовательскими скриптами выглядит так: создать динамический домен приложения, сгенерировать сборку, содержащую скомпилированный скрипт, загрузить ее во временный домен приложения и в конечном итоге выгрузить этот домен. В .NET Core из-за недоступности AppDomain API такой сценарий невозможен (по крайней мере, в версии .NET Core 2.1).

Хотя в .NET Framework описанное решение работает, у него есть недостатки; наибольший – накладные расходы на взаимодействие между доменами приложений.

> Именно из-за этих накладных расходов даже динамические сборки чаще всего просто загружаются в главный домен приложения, хотя это означает, что выгрузить их впоследствии не получится (для этого нужно было бы выгрузить все приложение). Так обстоит дело с популярным сериализатором `XmlSerializer`, что может приводить к утечке памяти, описанной ниже в сценарии 4.4.

Поэтому возникает идея облегченных сборок, допускающих сборку в мусор. *Забираемой сборкой* называется динамическая сборка, которую можно выгрузить, не выгружая домен приложения, в котором она находится. Это подошло бы для всех описанных выше сценариев. Но в настоящее время такая возможность отсутствует в обеих средах выполнения Microsoft .NET. Однако следите за обновлениями, поскольку в .NET Core ведется работа по реализации выгружаемого `AssemblyLoadContext`.

> В .NET Framework забираемые сборки реализованы, но лишь частично, для случая, когда код генерируется вручную методом `Reflection.Emit`. Процитируем MSDN: «Порождение отражения – это единственный поддерживаемый механизм для загрузки забираемых сборок. Сборки, загружаемые любым другим образом, выгрузить невозможно».

Области памяти процесса

Как было сказано в главе 2 и показано на рис. 2.20, среда выполнения .NET управляет несколькими областями памяти внутри процесса. Говоря об использовании памяти .NET-процессом, мы должны учитывать каждую из них. Рассмотрим эти области поочередно. Для этой цели воспользуемся великолепным инструментом VMMap, который показывает области памяти в процессе, к которому мы подключились. Представленные ниже области памяти соответствуют моменту непосредственно перед выходом из приложения в листинге 4.1.

Заглянув внутрь приложения Hello World, мы увидим области памяти, показанные на рис. 4.4. Чтобы интерпретировать результаты работы VMMap, следует вспомнить описание областей виртуальной памяти из главы 2. Как видим, в процессе почти 128 ТБ свободной памяти (это соответствует 128 ТБ виртуального адресного пространства на 64-разрядной платформе).

Type	Size	Committed	Private	Total WS	Private WS
Total	2,147,961,700 K	78,568 K	6,828 K	11,740 K	2,388 K
Image	37,924 K	37,908 K	3,436 K	9,236 K	772 K
Mapped File	4,064 K	4,064 K		388 K	
Shareable	2,147,508,516 K	33,140 K		512 K	20 K
Heap	3,828 K	2,344 K	2,280 K	1,084 K	1,080 K
Managed Heap	393,856 K	380 K	380 K	272 K	272 K
Stack	4,608 K	104 K	104 K	60 K	60 K
Private Data	7,000 K	592 K	592 K	152 K	148 K
Page Table	36 K	36 K	36 K	36 K	36 K
Unusable	1,868 K				
Free	135,290,991,744 K				

Рис. 4.4 ❖ Показанные программой VMMap области памяти работающего приложения из листинга 4.1. 64-разрядная среда выполнения .NET Core 2.0

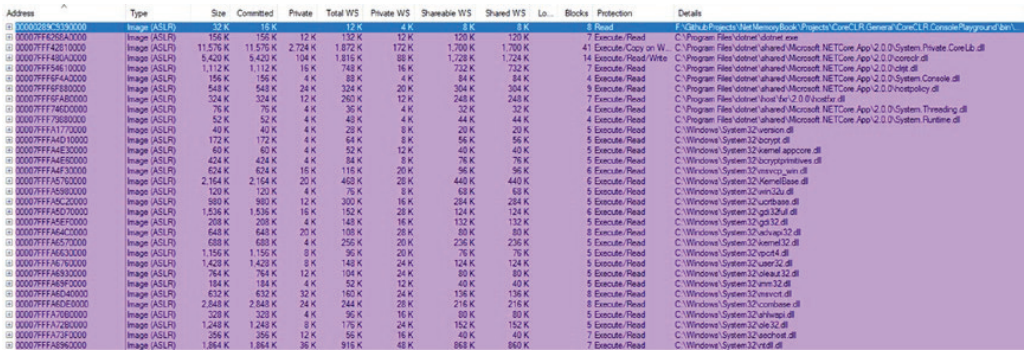
Кратко опишем эти области с точки зрения .NET:

- *Shareable* (около 2 ГиБ) – разделяемая память, которая нас не особенно интересует; передано только 32 МиБ, и лишь 20 КиБ находится в физической памяти. Эти области служат для целей системного управления, вообще не имеющих отношения к .NET;
- *Mapped File* (около 4 МиБ) – как отмечалось в главе 2, эти области содержат проецируемые файлы, в частности шрифты и файлы локализации. Хотя они и читаются средой выполнения .NET с применением различных API локализации, никаких проблем нашему приложению они создавать не должны;



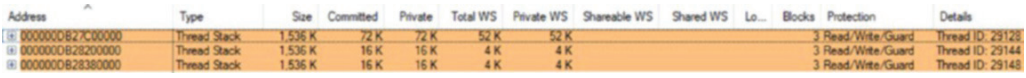
Address	Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Lo...	Blocks	Protection	Details
00000289C7990000	Mapped File	772 K	772 K		276 K		276 K	276 K		1	Read	C:\Windows\System32\locale.nls
00000289E1AD0000	Mapped File	3,292 K	3,292 K		112 K		112 K	112 K		1	Read	C:\Windows\Globalization\Sorting\SortDefault.nls

- *Image* (около 37 МиБ) – *двоичные образы*, содержащие различные исполняемые файлы .NET, включая саму среду выполнения и нашу сборку. Отметим, что большая часть этой области разделяемая, и лишь 772 КиБ входят в частный рабочий набор. Это файлы, которые читаются с диска на этапе запуска приложения;



Address	Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Lo...	Blocks	Protection	Details
00000289C5590000	Image (ASLR)	32 K	16 K		12 K	4 K	8 K	8 K		8	Read	F:\GitHub\Projects\NetMemoryBook\Projects\CoreCLR.General\CoreCLR.ConsolePlayground\bin\...
00007FF6258A0000	Image (ASLR)	156 K	156 K	12 K	132 K	12 K	120 K	120 K		7	Execute/Read	C:\Program Files\dotnet\dotnet.exe
00007FFF42810000	Image (ASLR)	11,576 K	11,576 K	2,724 K	1,872 K	172 K	1,700 K	1,700 K		41	Execute/Copy on W...	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\2.2.0\System.Private.CoreLib.dll
00007FFF480A0000	Image (ASLR)	5,420 K	5,420 K	104 K	1,816 K	88 K	1,728 K	1,724 K		14	Execute/Read/Write	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\2.2.0\coreclr.dll
00007FFF54610000	Image (ASLR)	1,112 K	1,112 K	16 K	748 K	16 K	732 K	732 K		7	Execute/Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\2.2.0\clrjit.dll
00007FFF54A20000	Image (ASLR)	156 K	156 K	4 K	88 K	4 K	84 K	84 K		5	Execute/Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\2.2.0\System.Console.dll
00007FFF6F880000	Image (ASLR)	548 K	548 K	24 K	324 K	20 K	304 K	304 K		9	Execute/Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\2.2.0\hostpolicy.dll
00007FFF6FAB0000	Image (ASLR)	324 K	324 K	12 K	260 K	12 K	248 K	248 K		7	Execute/Read	C:\Program Files\dotnet\host\fxr\2.2.0\hostfxr.dll
00007FFF746D0000	Image (ASLR)	76 K	76 K	4 K	36 K	4 K	32 K	32 K		4	Execute/Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\2.2.0\System.Threading.dll
00007FFF75680000	Image (ASLR)	52 K	52 K	4 K	48 K	4 K	44 K	44 K		4	Execute/Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\2.2.0\System.Runtime.dll
00007FFFA1770000	Image (ASLR)	40 K	40 K	4 K	28 K	8 K	20 K	20 K		5	Execute/Read	C:\Windows\System32\version.dll
00007FFFA4D10000	Image (ASLR)	172 K	172 K	4 K	64 K	8 K	56 K	56 K		5	Execute/Read	C:\Windows\System32\bcrypt.dll
00007FFFA4E30000	Image (ASLR)	60 K	60 K	4 K	52 K	12 K	40 K	40 K		5	Execute/Read	C:\Windows\System32\kernel.appcore.dll
00007FFFA4EB0000	Image (ASLR)	424 K	424 K	4 K	84 K	8 K	76 K	76 K		5	Execute/Read	C:\Windows\System32\bcryptprimitives.dll
00007FFFA4F30000	Image (ASLR)	624 K	624 K	16 K	116 K	20 K	96 K	96 K		6	Execute/Read	C:\Windows\System32\msvcp_win.dll
00007FFFA5760000	Image (ASLR)	2,164 K	2,164 K	20 K	468 K	28 K	440 K	440 K		6	Execute/Read	C:\Windows\System32\KernelBase.dll
00007FFFA5980000	Image (ASLR)	120 K	120 K	4 K	76 K	8 K	68 K	68 K		5	Execute/Read	C:\Windows\System32\win32u.dll
00007FFFA5C20000	Image (ASLR)	980 K	980 K	12 K	200 K	16 K	284 K	284 K		5	Execute/Read	C:\Windows\System32\ucrtbase.dll
00007FFFA5D70000	Image (ASLR)	1,536 K	1,536 K	16 K	152 K	28 K	124 K	124 K		6	Execute/Read	C:\Windows\System32\gdi32full.dll
00007FFFA5EF0000	Image (ASLR)	208 K	208 K	4 K	148 K	16 K	132 K	132 K		5	Execute/Read	C:\Windows\System32\gdi32.dll
00007FFFA64C0000	Image (ASLR)	648 K	648 K	20 K	108 K	28 K	80 K	80 K		8	Execute/Read	C:\Windows\System32\advapi32.dll
00007FFFA6570000	Image (ASLR)	688 K	688 K	4 K	256 K	20 K	236 K	236 K		5	Execute/Read	C:\Windows\System32\kernel32.dll
00007FFFA6630000	Image (ASLR)	1,156 K	1,156 K	8 K	96 K	20 K	76 K	76 K		5	Execute/Read	C:\Windows\System32\rpcrt4.dll
00007FFFA6760000	Image (ASLR)	1,428 K	1,428 K	8 K	148 K	24 K	124 K	124 K		5	Execute/Read	C:\Windows\System32\user32.dll
00007FFFA6930000	Image (ASLR)	764 K	764 K	12 K	104 K	24 K	80 K	80 K		5	Execute/Read	C:\Windows\System32\oleaut32.dll
00007FFFA69F0000	Image (ASLR)	184 K	184 K	4 K	52 K	12 K	40 K	40 K		5	Execute/Read	C:\Windows\System32\msvcrt.dll
00007FFFA6D40000	Image (ASLR)	632 K	632 K	32 K	160 K	24 K	136 K	136 K		8	Execute/Read	C:\Windows\System32\msvcrt.dll
00007FFFA6DE0000	Image (ASLR)	2,848 K	2,848 K	24 K	244 K	28 K	216 K	216 K		8	Execute/Read	C:\Windows\System32\combase.dll
00007FFFA70B0000	Image (ASLR)	328 K	328 K	4 K	96 K	16 K	80 K	80 K		5	Execute/Read	C:\Windows\System32\shlwapi.dll
00007FFFA72B0000	Image (ASLR)	1,248 K	1,248 K	8 K	176 K	24 K	152 K	152 K		5	Execute/Read	C:\Windows\System32\ole32.dll
00007FFFA73F0000	Image (ASLR)	356 K	356 K	12 K	56 K	16 K	40 K	40 K		7	Execute/Read	C:\Windows\System32\sechost.dll
00007FFFA8960000	Image (ASLR)	1,864 K	1,864 K	36 K	916 K	48 K	868 K	860 K		7	Execute/Read	C:\Windows\System32\ntdll.dll

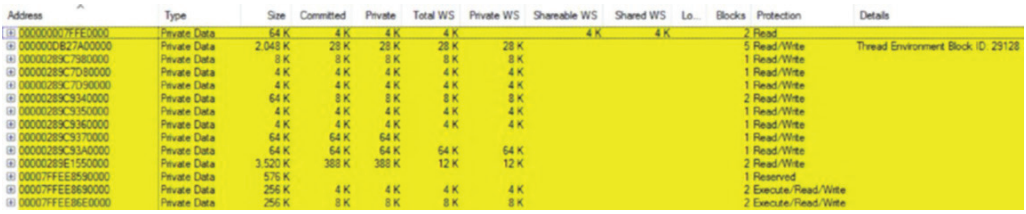
- *Stack* (около 4,5 МиБ) – в нашем приложении Hello World три потока, поэтому для них отведено три области под стеки;



Address	Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Lo...	Blocks	Protection	Details
000000DB27C00000	Thread Stack	1,536 K	72 K	72 K	52 K	52 K				3	Read/Write/Guard	Thread ID: 29128
000000DB28200000	Thread Stack	1,536 K	16 K	16 K	4 K	4 K				3	Read/Write/Guard	Thread ID: 29144
000000DB28380000	Thread Stack	1,536 K	16 K	16 K	4 K	4 K				3	Read/Write/Guard	Thread ID: 29148

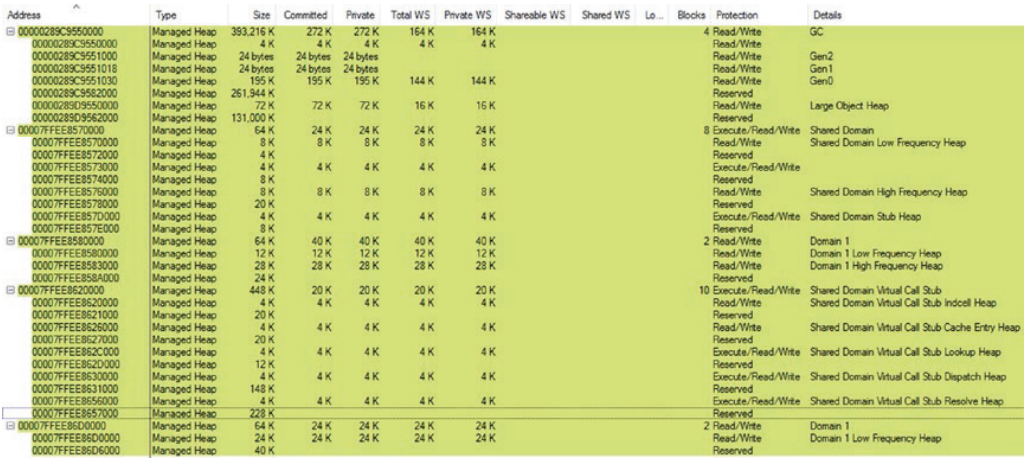
- *Heap* и *Private Data* (около 9 МиБ) – это различные области памяти, которые среда выполнения .NET использует для собственных целей. В основном здесь хранятся вещи, которые нас не интересуют (и даже понять, что это такое, невозможно без глубокого анализа исходного кода CoreCLR). Но можно отметить, что среди них есть фундаментальные структуры данных, используемые движком выполнения и сборщиком мусора, например:
  - список пометки и таблицы карт, с которыми мы познакомимся в главах 5, 8 и 11;
  - здесь хранятся данные о регистрации интернированных строк;

– обратите внимание, что две последние области памяти помечены флагами защиты Выполнение/Чтение/Запись. Сюда JIT-компилятор помещает машинный код при компиляции CIL-кода. Потому-то они и помечены флагом выполнения (Execute), поскольку этот код должен допускать вызов, как любой другой. По сути дела, эти области составляют ядро исполняемого кода нашего приложения, написанного на C# или любом другом .NET-совместимом языке. Если по какой-то причине в нашем приложении часто производится JIT-компиляция, то мы будем наблюдать постоянный рост таких частных областей с флагами Выполнение/Чтение/Запись;
– различные временные области, необходимые во время JIT-компиляции;



Address	Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Lo...	Blocks	Protection	Details
000000007FFE0000	Private Data	64 K	4 K	4 K	4 K		4 K	4 K		2	Read	
00000DB27A00000	Private Data	2.048 K	28 K	28 K	28 K	28 K				5	Read/Write	Thread Environment Block ID: 29128
00000289C7980000	Private Data	8 K	8 K	8 K	8 K	8 K				1	Read/Write	
00000289C7D80000	Private Data	4 K	4 K	4 K	4 K	4 K				1	Read/Write	
00000289C7D90000	Private Data	4 K	4 K	4 K	4 K	4 K				1	Read/Write	
00000289C9340000	Private Data	64 K	8 K	8 K	8 K	8 K				2	Read/Write	
00000289C9350000	Private Data	4 K	4 K	4 K	4 K	4 K				1	Read/Write	
00000289C9360000	Private Data	4 K	4 K	4 K	4 K	4 K				1	Read/Write	
00000289C9370000	Private Data	64 K	64 K	64 K						1	Read/Write	
00000289C93A0000	Private Data	64 K	64 K	64 K	64 K	64 K				1	Read/Write	
00000289E1550000	Private Data	3.520 K	388 K	388 K	12 K	12 K				2	Read/Write	
00007FFEE8590000	Private Data	576 K								1	Reserved	
00007FFEE8690000	Private Data	256 K	4 K	4 K	4 K	4 K				2	Execute/Read/Write	
00007FFEE86E0000	Private Data	256 K	8 K	8 K	8 K	8 K				2	Execute/Read/Write	

❍ *Managed Heap* (около 384 МиБ) – основой управления памятью в .NET является управляемая куча, поддерживаемая сборщиком мусора, и другие кучи, используемые средой выполнения. Поскольку эта область памяти для нас важнее всего, чуть ниже мы рассмотрим ее отдельно;



Address	Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Lo...	Blocks	Protection	Details
00000289C9550000	Managed Heap	393,216 K	272 K	272 K	164 K	164 K				4	Read/Write	GC
00000289C9550000	Managed Heap	4 K	4 K	4 K	4 K	4 K					Read/Write	
00000289C9551000	Managed Heap	24 bytes	24 bytes	24 bytes							Read/Write	Gen2
00000289C9551018	Managed Heap	24 bytes	24 bytes	24 bytes							Read/Write	Gen1
00000289C9551030	Managed Heap	195 K	195 K	195 K	144 K	144 K					Read/Write	Gen0
00000289C9582000	Managed Heap	261,944 K									Reserved	
00000289D9550000	Managed Heap	72 K	72 K	72 K	16 K	16 K					Read/Write	Large Object Heap
00000289D9562000	Managed Heap	131,000 K									Reserved	
00007FFEE8570000	Managed Heap	64 K	24 K	24 K	24 K	24 K				8	Execute/Read/Write	Shared Domain
00007FFEE8570000	Managed Heap	8 K	8 K	8 K	8 K	8 K					Read/Write	Shared Domain Low Frequency Heap
00007FFEE8572000	Managed Heap	4 K									Reserved	
00007FFEE8573000	Managed Heap	4 K	4 K	4 K	4 K	4 K					Execute/Read/Write	
00007FFEE8574000	Managed Heap	8 K									Reserved	
00007FFEE8576000	Managed Heap	8 K	8 K	8 K	8 K	8 K					Read/Write	Shared Domain High Frequency Heap
00007FFEE8578000	Managed Heap	20 K									Reserved	
00007FFEE857D000	Managed Heap	4 K	4 K	4 K	4 K	4 K					Execute/Read/Write	Shared Domain Stub Heap
00007FFEE857E000	Managed Heap	8 K									Reserved	
00007FFEE8580000	Managed Heap	64 K	40 K	40 K	40 K	40 K				2	Read/Write	Domain 1
00007FFEE8580000	Managed Heap	12 K	12 K	12 K	12 K	12 K					Read/Write	Domain 1 Low Frequency Heap
00007FFEE8583000	Managed Heap	28 K	28 K	28 K	28 K	28 K					Read/Write	Domain 1 High Frequency Heap
00007FFEE858A000	Managed Heap	24 K									Reserved	
00007FFEE8620000	Managed Heap	448 K	20 K	20 K	20 K	20 K				10	Execute/Read/Write	Shared Domain Virtual Call Stub
00007FFEE8620000	Managed Heap	4 K	4 K	4 K	4 K	4 K					Read/Write	Shared Domain Virtual Call Stub Indcell Heap
00007FFEE8621000	Managed Heap	20 K									Reserved	
00007FFEE8626000	Managed Heap	4 K	4 K	4 K	4 K	4 K					Read/Write	Shared Domain Virtual Call Stub Cache Entry Heap
00007FFEE8627000	Managed Heap	20 K									Reserved	
00007FFEE862C000	Managed Heap	4 K	4 K	4 K	4 K	4 K					Execute/Read/Write	Shared Domain Virtual Call Stub Lookup Heap
00007FFEE862D000	Managed Heap	12 K									Reserved	
00007FFEE8630000	Managed Heap	4 K	4 K	4 K	4 K	4 K					Execute/Read/Write	Shared Domain Virtual Call Stub Dispatch Heap
00007FFEE8631000	Managed Heap	148 K									Reserved	
00007FFEE8656000	Managed Heap	4 K	4 K	4 K	4 K	4 K					Execute/Read/Write	Shared Domain Virtual Call Stub Resolve Heap
00007FFEE8657000	Managed Heap	228 K									Reserved	
00007FFEE86D0000	Managed Heap	64 K	24 K	24 K	24 K	24 K				2	Read/Write	Domain 1
00007FFEE86D0000	Managed Heap	24 K	24 K	24 K	24 K	24 K					Read/Write	Domain 1 Low Frequency Heap
00007FFEE86D6000	Managed Heap	40 K									Reserved	

❍ *Page Table* (небольшая область размером 36 КиБ) – таблица страниц, описанная в главе 2;
❍ *Unusable* (почти 2 МиБ) – в силу гранулярности выделения страниц (см. главу 2) некоторые части памяти оказались непригодными для использования.

Область, обозначенную Managed Heap, можно разделить на несколько категорий.

- Куча GC – самая важная для нас куча, которой управляет сборщик мусора. Большая часть типов нашего приложения создается здесь, поэтому это главное для нас место и наиболее вероятный источник проблем. Начиная с главы 5 и до конца книги мы будем описывать, как GC управляет этой кучей. В терминах главы 1 это свободная память, которая управляется механизмом сборщика мусора и его распределителем. Отметим, однако, сколько всего интересного мы узнали, пока добрались до этой области памяти! А еще много глав будет посвящено ее подробному описанию.
- Кучи других доменов – каждый домен приложения имеет собственный набор куч, так что могут быть кучи разделяемого домена, системного домена, домена по умолчанию и динамически созданных доменов. И в каждой может быть несколько подобластей.
  - *Высокочастотная куча* – используется для хранения данных, к которым домен часто обращается для своих внутренних надобностей. В комментариях к коду CoreCLR сказано, что это «кучи для выделения данных, существующих на протяжении жизни AppDomain. Объекты, которые выделяются часто, должны браться из высокочастотной кучи, поскольку это улучшает управление страницами». Поэтому, например, высокочастотная куча разделяемого домена содержит такие часто используемые данные о типах, как подробное описание методов и полей. Здесь же находятся статические данные примитивных типов.
  - *Низкочастотная куча* – содержит не столь востребованные данные о типах. Среди прочего, здесь находится EEClass и другие данные, необходимые JIT-компилятору, а также механизмам отражения и загрузки типов.
  - *Куча заглушек* – как сказано в документации, она «содержит заглушки, необходимые для разграничения доступа к коду (CAS), вызова COM-оберток и механизма P/Invoke».
  - *Куча заглушек виртуальных вызовов* – содержит структуры данных и код, используемые механизмом диспетчеризации виртуальных заглушек (Virtual Stub Dispatch – VSD) (использование заглушек для вызова виртуальных методов вместо традиционной таблицы виртуальных методов) для диспетчеризации интерфейсов. Эта куча далее подразделяется на кучу записей кеша (Cache Entry Heap), кучу диспетчеризации (Dispatch Heap), кучу индексных ячеек (Indcell Heap), кучу сопоставления (Lookup Heap) и кучу разрешения (Resolve Heap). Все они включают данные различных типов, необходимые для VSD. Эти кучи невелики (сотни кибибайтов), даже если в приложении тысячи интерфейсов.
  - Высокочастотная куча, низкочастотная куча, куча заглушек и различные виды куч заглушек виртуальных вызовов все вместе называются кучей загрузчика (Loader Heap), потому что отвечают за хранение данных, необходимых системе типов (и потому за загрузку типов). Что бы вам ни говорили, не существует кучи загрузчика как специально созданной области памяти – это просто собирательное название нескольких областей.

ПРИМЕЧАНИЕ По умолчанию эти кучи совсем малы, порядка одной страницы, т. е. около 64 КиБ. В этом легко убедиться, взглянув на определения размеров по умолчанию в исходном коде CoreCLR:

```
#define LOW_FREQUENCY_HEAP_RESERVE_SIZE          (3 * GetOsPageSize())
#define LOW_FREQUENCY_HEAP_COMMIT_SIZE           (1 * GetOsPageSize())

#define HIGH_FREQUENCY_HEAP_RESERVE_SIZE         (10 * GetOsPageSize())
#define HIGH_FREQUENCY_HEAP_COMMIT_SIZE          (1 * GetOsPageSize())

#define STUB_HEAP_RESERVE_SIZE                   (3 * GetOsPageSize())
#define STUB_HEAP_COMMIT_SIZE                    (1 * GetOsPageSize())
```

Напомним, что тип, загруженный в кучу загрузчика, уже нельзя выгрузить до выгрузки всего соответствующего домена приложения. Если мы постоянно загружаем много новых типов (например, в случае динамической загрузки или генерации сборок), то потребление памяти может существенно возрасти. И помните, что домен приложения по умолчанию не выгружается до завершения программы.

Как отмечалось в главе 2, существует возможность изменить размер стека, выделяемого потокам программы по умолчанию. Для этого пригодится командная утилита `dumpbin`, входящая в дистрибутив Visual Studio. Следующая команда изменяет заголовок указанного исполняемого файла:

```
editbin DotNet.HelloWorld.exe /stack:8000000
```

Для исполняемых файлов, предназначенных для .NET Framework (как в примере выше), она пока работает, но лучше считать такой подход неподдерживаемым – не гарантируется, что в будущем .NET Framework не станет игнорировать это значение при создании потоков. В случае .NET Core исполняемым файлом является сама программа запуска среды выполнения, которая обычно находится в файле `C:\Program Files\dotnet\dotnet.exe`. Чтобы изменить размер стека потоков в приложениях для .NET Core, нужно было бы отредактировать с помощью `editbin` этот файл, что, очевидно, в большинстве случаев неприемлемо. Поэтому, хотя описанное манипулирование размером стека и возможно, полагаться на него не следует.

Теперь перейдем к одной из важнейших частей книги – нашему первому сценарию. Как и все последующие, он будет состоять из описания некоторой ситуации и описания того, как подходить к ее анализу и разрешению.

Сценарий 4.1. Сколько места в памяти занимает моя программа?

Проблема. Заказчик, для которого мы пишем .NET-приложение, спрашивает, сколько памяти оно требует и как использует эту память, поскольку подозревает, что программа «кушает» слишком много. Вопрос вызвал смятение у разработчиков, потому что, как оказалось, никто не знает ответа и даже не понимает, как правильно измерять потребление памяти. Все наперебой предлагают разные инструменты и способы интерпретации результатов. Поставим себя на место разработчиков Paint.NET (https://www.getpaint.net/)!

Решение. Чтобы дать правильный ответ заказчику, нужно понимать, как операционная система видит память нашего процесса. Вкратце это было описано в главе 2, и вы, возможно, обратили внимание, что между инструментами нет согласия в этом вопросе. На верхнем уровне мы должны сосредоточиться на следующих измерениях:

- частный рабочий набор – самое важное измерение, показывающее объем физической памяти, занятой процессом. Очевидно, главный «затык» может быть именно здесь, поэтому сюда и смотрим в первую очередь;
- байты исключительного пользования (или размер переданной памяти) – суммарный объем памяти, находящейся в ОЗУ и выгруженной на диск. Нам лишний страничный обмен ни к чему, поэтому если этот размер намного больше частного рабочего набора, то следует заподозрить неладное. Неограниченный рост файла подкачки опасен, потому что размер дисков не бесконечен;
- байты виртуальной памяти – общий объем виртуальной памяти, переданной (частной) и еще только зарезервированной, вне зависимости от того, где она находится. Это самое абстрактное из всех измерений, потому что не означает большого потребления физических ресурсов, разве что таблиц страниц (см. главу 2). Однако если размер доходит до сотен гигабайтов или постоянно растет, это должно стать поводом для беспокойства.

В Windows, чтобы узнать эти размеры, достаточно открыть вкладку **Подробности** в диспетчере задач – они показаны соответственно в столбцах **Память (частный рабочий набор)** (Memory (private working set)) и **Выделенная память** (Commit size). Байты виртуальной памяти эта программа не показывает. См. рис. 4.5.

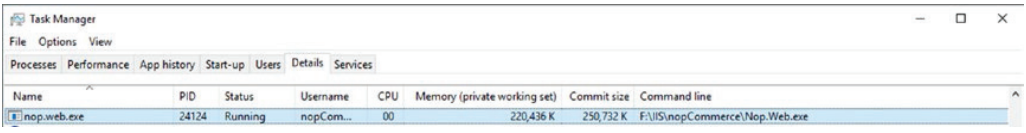


Рис. 4.5 ❖ Сведения об использовании памяти в окне диспетчера задач

Можно также последить за ними в системном мониторе (рис. 4.6). Для этого нужно добавить счетчики `\Процесс(имя процесса)\Рабочий набор (частный)` (\Process(processname)\Working Set – Private), `\Процесс(имя процесса)\Байт исключительного пользования` (\Process(processname)\Private Bytes) и `\Процесс(имя процесса)\Байт виртуальной памяти` (\Process(processname)\Virtual Bytes). В Linux можно воспользоваться программой `top`, столбцы которой были описаны в главе 2.

Для анализа памяти измеряемого процесса можно также воспользоваться программой VMMap в Windows (см. рис. 4.4). Там нужно смотреть на столбцы Private WS, Private и Size, соответствующие описанным выше показателям. Если говорить о типах памяти, то в первую очередь следует, конечно, смотреть на управляемую кучу (Managed Heap). Однако другими типами памяти тоже пренебрегать не стоит. Если вы заподозрили утечку памяти, понаблюдайте за размерами всех типов памяти в динамике и попытайтесь выяснить, какой размер постоянно растет. Утечка может быть как в вашем управляемом коде, так и в каком-то неуправляемом компоненте, которым вы пользуетесь (быть может, даже неявно, не подозревая об этом).

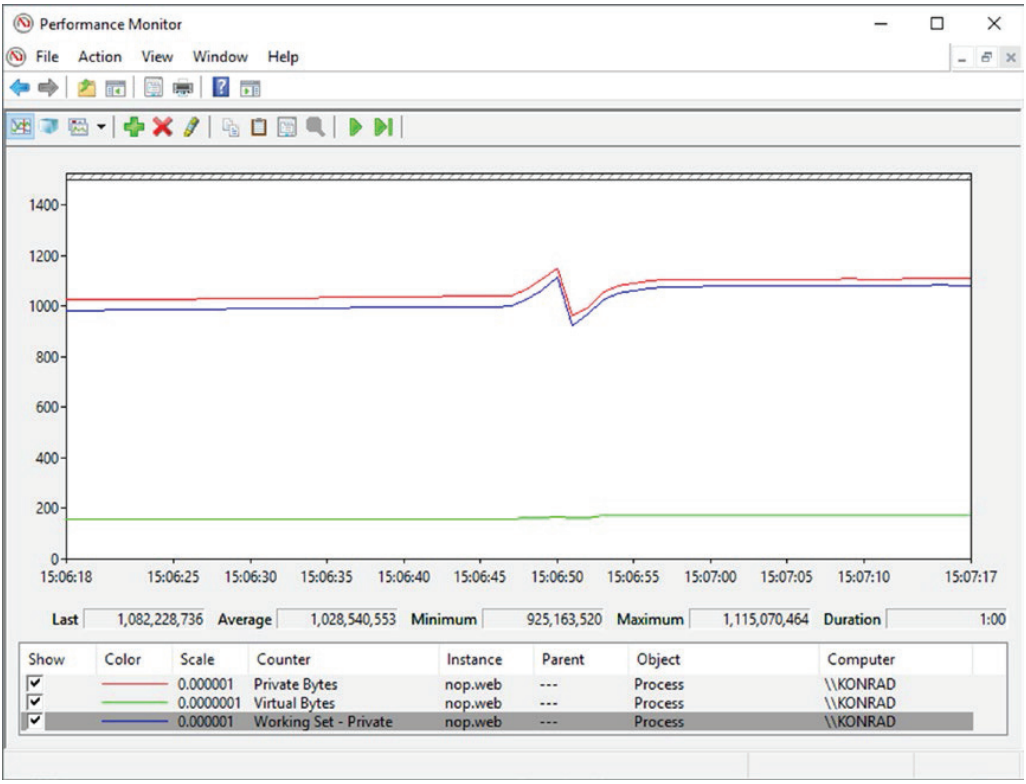


Рис. 4.6 ❖ Счетчики производительности, показывающие основные сведения об использовании памяти

Сценарий 4.2. Моя программа потребляет все больше и больше памяти

Описание. Заказчик сообщает об исключении OutOfMemory после нескольких дней непрерывной работы написанной нами службы Windows. Требуется установить причину, и, разумеется, срочно.

Решение. Если нам не предоставили полного дампа памяти процесса, то расследование следует начать с наблюдения за динамикой потребления памяти программой. Для начала можно понаблюдать за несколькими важными счетчиками в системном мониторе (рис. 4.7):

- `\Процесс(имя процесса)\Рабочий набор (частный)`;
- `\Процесс(имя процесса)\Байт исключительного пользования`;
- `\Процесс(имя процесса)\Байт виртуальной памяти`;
- `\Память CLR .NET(имя процесса)\Всего фиксировано байтов` – счетчик, дающий представление о характере использования управляемой кучи.

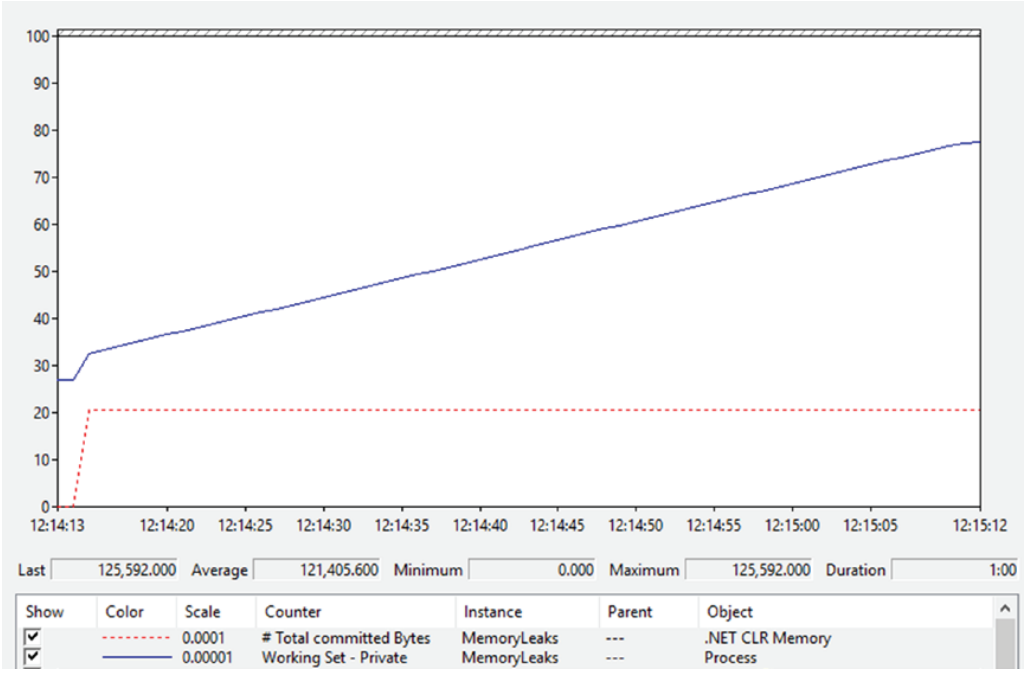


Рис. 4.7 ❖ Счетчики производительности для сценария 4.2 показывают, что размер управляемой кучи стабилен, но частный рабочий набор постоянно растет

Отсюда видно, что имеет место утечка памяти – память, занимаемая процессом, постоянно растет. Однако размер управляемой кучи очень стабилен, так что утечка, скорее всего, в неуправляемой памяти, которая к нашему коду для .NET не имеет отношения (однако может и иметь, как будет показано в сценарии 4.3!). Зная это, имеет смысл заглянуть внутрь процесса с помощью VMMap. Уже после недолгого наблюдения мы замечаем, что размер кучи Heap (Private Data) монотонно возрастает. Наша программа медленно выделяет все больше блоков памяти размером около 16 МиБ (рис. 4.8).

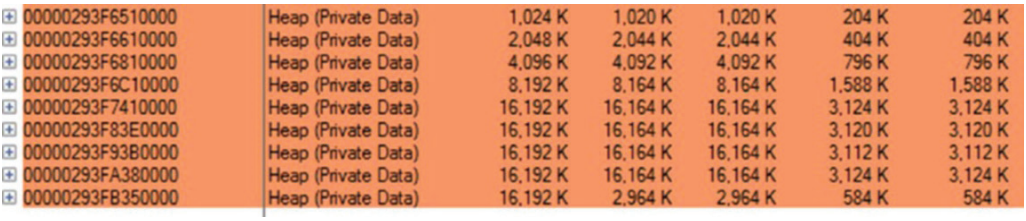


Рис. 4.8 ❖ Сценарий 4.2 – VMMap показывает области памяти в куче. Память монотонно растет, и время от времени выделяются новые блоки типа Heap (Private Data)

Это первый ключ в нашем расследовании – куча, скорее всего, растет из-за неумеренного использования Heap API (например, вызова функции `malloc` в C или оператора `new` в C++). Теперь надо определить, кто это делает. Использовать для этой цели дамп памяти процесса утомительно, поскольку анализировать неуправляемую память очень трудно (особенно для .NET-разработчиков, не привыкших к неуправляемому миру). По счастью, есть гораздо более простой способ исследования проблемы – программа PerfView. В ее диалоговом окне **Collect** введите имя исполняемого файла в поле **OS Heap Exe** или идентификатор процесса в поле **OS Heap Process** (имейте в виду, что только во втором случае вы сможете присоединиться к уже работающему процессу). Задание любого из параметров OS Heap разрешает ETW отслеживать использование Heap API. Начните сбор данных и подождите – сколько ждать, зависит от того, как быстро растет занимаемая процессом память.

Когда сбор данных остановится и вся обработка завершится, откройте элемент **Net OS Heap Alloc Stacks** в папке **Memory Group**. Постепенно раскрывайте узлы дерева, спускаясь все ниже и ниже в ту часть кода, где происходит основное выделение памяти (наибольшее значение в столбце Inc %). Возможно, в каких-то узлах понадобится загрузить символы при помощи команды **Lookup Symbols** контекстного меню. Также стоит выключить группировку модулей, выбрав из контекстного меню команду **Ungroup Module**. Очень скоро вы увидите причину более 90 % выделений памяти (рис. 4.9). Вот она – мощь ETW – к вашим услугам!

Name	Inc %	Inc
ROOT	100.0	25,161,660.0
+ Process64 MemoryLeaks (19592)	100.0	25,161,660.0
+ Thread (17656) CPU=129ms	99.4	25,015,120.0
+ module ntdll <<ntdll!RtlUserThreadStart>>	99.4	25,015,120.0
+ module kernel32 <<kernel32!BaseThreadInitThunk>>	99.4	25,015,120.0
+ module mscoree <<mscoree!_CorExeMain_Exported>>	99.4	25,015,120.0
+ module mscoreei <<mscoreei!_CorExeMain>>	99.4	25,015,120.0
+ module clr <<clr!_CorExeMain>>	99.4	25,015,120.0
+ MemoryLeaks!MemoryLeaks.Program.Main(class System.String[])	92.9	23,375,120.0
+ MemoryLeaks!MemoryLeaks.Leaks.UnmanagedLeak.Run()	92.5	23,282,020.0
+ MemoryLeaks!MemoryLeaks.Leaks.SomeService.DoSomeProcessing()	92.5	23,281,520.0
+ memoryleaks.library!MemoryLeaksLibrary.Class1.DoSomething(int32)	92.5	23,280,800.0
+ memoryleaks.library!dynamicClass.IL_STUB_PInvoke(int32,optional_modifier System.Runtime.CompilerServices.IsConst)	92.5	23,280,000.0
+ unmanagedlibrary!CUnmanagedLibrary::CalculateSomething	92.5	23,280,000.0
+ unmanagedlibrary!operator new	92.5	23,280,000.0

Рис. 4.9 ❖ Сценарий 4.2 – анализ в PerfView. Виден агрегированный стек вызовов для оператора new

Мы видим, что причиной большинства выделений памяти является вызов оператора `new` из метода `CUnmanagedLibrary::CalculateSomething`, который вызывается другими компонентами нашего .NET-приложения. Мы действительно нашли корень зла, потому что этот метод специально реализован на редкость глупо (листинг 4.4).

Листинг 4.4 ❖ Причина утечки памяти в сценарии 1.2

```
int CUnmanagedLibrary::CalculateSomething(int size)
{
   int* buffer = new int[size];
   return 2 * size;
}
```

В реальных ситуациях выделение памяти может происходить в разных местах, их все придется исследовать и высказать гипотезу о том, где может быть проблема.

Имейте в виду, что, не имея файлов символов для неуправляемых библиотек, используемых нашим приложением, мы не увидим имен методов и функций в представлении Net Virtual Alloc Stacks. Но все же мы получим указание на то, какой компонент является источником проблемы, поэтому сможем связаться с его изготовителем или поискать решение в сети. Также не стоит забывать, что трассировка Heap API средствами ETW может сопровождаться весьма заметными накладными расходами, так что будьте осторожны, включая ее в производственной среде.

Сценарий 4.3. Моя программа потребляет все больше и больше памяти

Описание. Что-то странное происходит с нашим приложением на машинах заказчика. Потребление памяти вроде бы монотонно возрастает, но это не оказывает никакого негативного влияния, и программа работает нормально. Заказчик сообщает, что приложение пожирает «гигабайты памяти», хотя у себя мы ничего такого не замечали. Никто не знает, надо ли уже бояться или еще нет.

Анализ. Снова следует начать расследование с наблюдения за динамикой потребления памяти программой. Для начала можно воспользоваться системным монитором и понаблюдать за счетчиками:

- `\Процесс(имя процесса)\Рабочий набор (частный)`;
- `\Процесс(имя процесса)\Байт исключительного пользования`;
- `\Процесс(имя процесса)\Байт виртуальной памяти`;
- `\Память CLR .NET(имя процесса)\Всего фиксировано байтов`.

Вскоре мы замечаем, что размеры управляемой кучи и частного рабочего набора стабильны. Но все время растет количество байтов исключительного пользования – быть может, большая часть выделенной памяти находится не в ОЗУ. Количество байтов виртуальной памяти тоже растет, что указывает на «потребление» гигабайтов памяти! Если просмотреть память процесса в VMMap, то причина станет ясна (рис. 4.10).

Действительно, занято свыше 40 ГБ виртуальной памяти. Только вот около 37 ГБ помечено как непригодная для использования! Это означает, что кто-то выделяет страницы крайне неэффективно (вспомните главу 2). В этом можно убедиться, взглянув на список областей памяти (рис. 4.11), где присутствует очень много страниц с непригодными для использования данными.

Type	Size	Committed	Private	Total WS	Private WS
Total	40,117,824 K	2,615,052 K	2,558,776 K	89,832 K	80,228 K
Image	52,668 K	52,656 K	5,316 K	9,220 K	356 K
Mapped File	4,064 K	4,064 K		424 K	
Shareable	24,996 K	4,808 K		308 K	
Heap	2,988 K	2,000 K	1,936 K	376 K	372 K
Managed Heap	393,856 K	4,264 K	4,264 K	4,228 K	4,228 K
Stack	12,288 K	116 K	116 K	16 K	16 K
Private Data	2,478,796 K	2,471,944 K	2,471,944 K	60 K	56 K
Page Table	75,200 K	75,200 K	75,200 K	75,200 K	75,200 K
Unusable	37,072,968 K				
Free	137,398,910,784 K				

Рис. 4.10 ❖ Сценарий 4.3 – представление процесса в VMMap. Объем виртуальной памяти (Size) очень велик, но большая ее часть непригодна для использования

Address	Type	Size	Committed	Private
⊞ 0000019492DB0000	Private Data	4 K	4 K	4 K
0000019492DB1000	Unusable	60 K		
⊞ 0000019492DC0000	Private Data	4 K	4 K	4 K
0000019492DC1000	Unusable	60 K		
⊞ 0000019492DD0000	Private Data	4 K	4 K	4 K
0000019492DD1000	Unusable	60 K		
⊞ 0000019492DE0000	Private Data	4 K	4 K	4 K
0000019492DE1000	Unusable	60 K		
⊞ 0000019492DF0000	Private Data	4 K	4 K	4 K
0000019492DF1000	Unusable	60 K		
⊞ 0000019492E00000	Private Data	4 K	4 K	4 K
0000019492E01000	Unusable	60 K		
⊞ 0000019492E10000	Private Data	4 K	4 K	4 K
0000019492E11000	Unusable	60 K		
⊞ 0000019492E20000	Private Data	4 K	4 K	4 K
0000019492E21000	Unusable	60 K		
⊞ 0000019492E30000	Private Data	4 K	4 K	4 K

Рис. 4.11 ❖ Сценарий 4.3 – представление непригодных для использования областей памяти в VMMap. Таких областей очень много, а между ними вкраплены одностраничные блоки частных данных

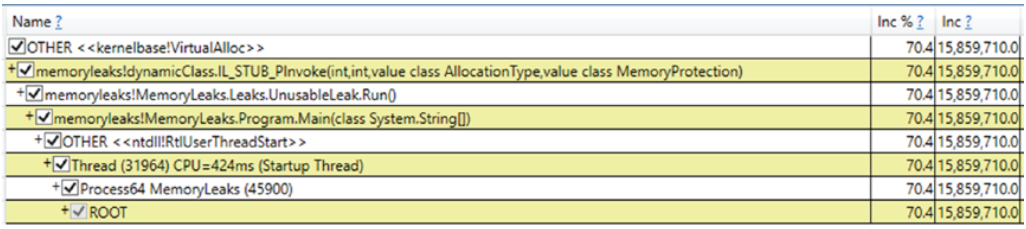
Теперь необходимо понять, где в программе страницы используются столь неподобающим образом. И снова на помощь приходит PerfView. Но сейчас нас интересует Virtual API (например, вызов функции `VirtualAlloc`), поскольку речь идет о памяти типа Private Data (а не Heap). Как и раньше, мы воспользуемся PerfView для исследования собранных данных ETW. Но в диалоговом окне **Collect** нужно отметить флажок **VirtAlloc** и начать сбор данных, пока проблемное приложение работает. Активация этого поставщика влечет за собой меньшие накладные расходы, чем в случае наблюдения за Heap API в сценарии 4.2.

Когда сбор данных остановится и вся обработка завершится, откройте элемент **Net Virtual Alloc Stacks** в папке **Memory Group**. Если утечка памяти значительна, то причина, скорее всего, будет видна уже в верхней части списка – в нашем случае 94,1 % всех выделений памяти производилось с помощью функции `VirtualAlloc` (рис. 4.12)!

Name ?	Exc % ?	Exc ?
OTHER <<kernelbase!VirtualAlloc>>	70.4	15,859,710
OTHER <<ntdll!RtlUserThreadStart>>	13.1	2,953,216
OTHER <<mscorlib.ni!System.String.FormatHelper(System.IFormatProvider, System.String, System.ParamsArray)>>	9.9	2,232,320
OTHER <<mscorlib.ni!System.Console.WriteLine(System.String)>>	2.6	593,920

Рис. 4.12 ❖ Сценарий 4.3 – анализ в PerfView показывает очень много обращений к VirtualAlloc

Если дважды щелкнуть по этой строке, то появится дерево вызовов. Раскройте узлы, дающие наибольший вклад в количество выделений памяти. При необходимости загрузите символы и выключите группировку с помощью команд **Lookup Symbols** и **Ungroup Module** в контекстном меню. Действуя таким образом, мы должны найти, где в программе больше всего обращений к выделению памяти. В нашем случае это метод `MemoryLeaks.Leaks.UnusableLeak.Run()` из модуля `MemoryLeaks` (рис. 4.13).



Name ?	Inc % ?	Inc ?
OTHER <<kernelbase!VirtualAlloc>>	70.4	15,859,710.0
+ memoryleaks!dynamicClass.IL_STUB_PInvoke(int,int,value class AllocationType,value class MemoryProtection)	70.4	15,859,710.0
+ memoryleaks!MemoryLeaks.Leaks.UnusableLeak.Run()	70.4	15,859,710.0
+ memoryleaks!MemoryLeaks.Program.Main(class System.String[])	70.4	15,859,710.0
+ OTHER <<ntdll!RtlUserThreadStart>>	70.4	15,859,710.0
+ Thread (31964) CPU=424ms (Startup Thread)	70.4	15,859,710.0
+ Process64 MemoryLeaks (45900)	70.4	15,859,710.0
+ ROOT	70.4	15,859,710.0

Рис. 4.13 ❖ Сценарий 4.3 – анализ в PerfView показывает агрегированный стек вызовов VirtualAlloc

И действительно, этот метод содержит вызов функции `VirtualAlloc`, которая выделяет только одну страницу (обычно 4 КиБ), хотя, как мы знаем, гранулярность выделения памяти в Windows составляет 64 КиБ (листинг 4.5). Стало быть, 60 КиБ памяти расходуется впустую при каждом вызове `VirtualAlloc`.

Листинг 4.5 ❖ Фрагмент кода, вызвавший проблему в сценарии 4.3

```
ulong block = (ulong)DllImports.VirtualAlloc(IntPtr.Zero, new
IntPtr(pageSize),
       DllImports.AllocationType.Commit,
       DllImports.MemoryProtection.ReadWrite);
```

В реальной ситуации какая-нибудь используемая нами неуправляемая библиотека действительно может вызывать `VirtualAlloc` так неэффективно. Но, воспользовавшись данными ETW для Virtual API, мы смогли проследить проблему до ее источника.

Сценарий 4.4. Моя программа потребляет все больше и больше памяти

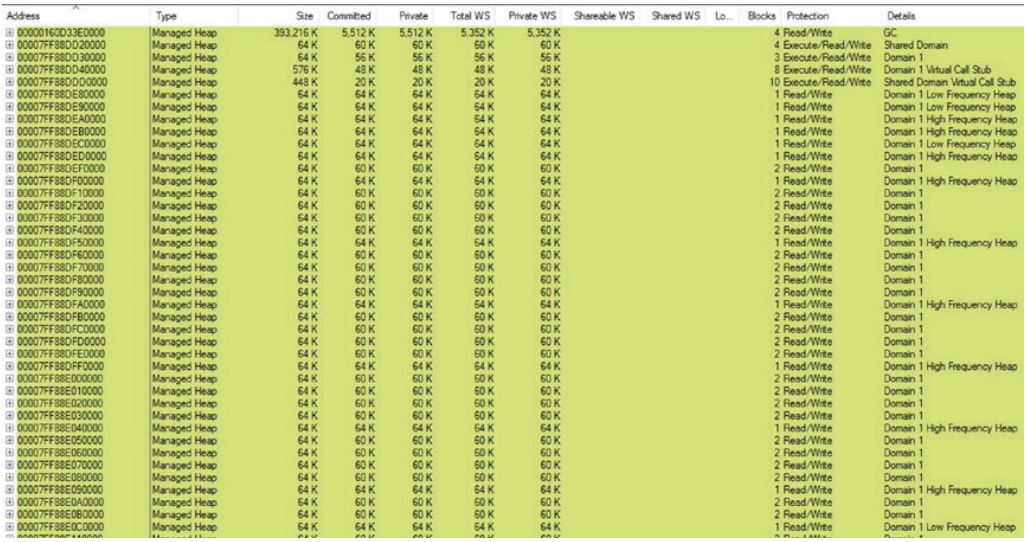
Описание. Заказчик жалуется, что программа потребляет много памяти. Раз за разом она «отъедает» гигабайты памяти, после чего «падает» с исключением OutOfMemory. Мы уверены, что у нас нет никаких неуправляемых компонентов, поэтому утечка наверняка происходит в коде на C# (хотя следует помнить, что библиотеки, которыми мы пользуемся, могут внутри себя вызывать неуправляемый код, так что ... всегда нужно проявлять осторожность и не забывать о рассмотренных выше сценариях). Заказчик прислал нам два снимка экрана диспетчера задач, на которых видно, что да, объем выделенной памяти поступательно растет.

Анализ. Начинаем с типичного наблюдения за процессом в системном мониторе и следим за четырьмя счетчиками:

- `\Процесс(имя процесса)\Рабочий набор (частный)`;
- `\Процесс(имя процесса)\Байт исключительного пользования`;
- `\Процесс(имя процесса)\Байт виртуальной памяти`;
- `\Память CLR .NET(имя процесса)\Всего фиксировано байтов`.

Как ни удивительно, размер управляемой кучи стабилен. Но все остальные размеры растут, включая и размер самого проблематичного частного рабочего набора. Инстинктивно мы решили заглянуть внутрь процесса с помощью VMMap. После нескольких минут наблюдения мы видим, что частный рабочий набор

управляемой кучи постоянно растет, так что утечка, по-видимому, где-то в .NET. Но почему она не отражается в счетчиках производительности? Глядя на список типов в управляемой куче в VMMap, мы замечаем кое-что необычное (рис. 4.14). Область управляемой кучи, помеченная как GC (та часть, где хранятся объекты, выделенные нашим приложением), растет очень медленно. С другой стороны, мы видим десятки областей типа Domain 1, Domain 1 Low Frequency Heap и Domain 1 High Frequency Heap! Это означает, что создается много дополнительных сборок, скорее всего, вследствие динамической загрузки сборок.



Address	Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Lo...	Blocks	Protection	Details
00000160D33E0000	Managed Heap	393,216 K	5,512 K	5,512 K	5,352 K	5,352 K				4	Read/Write	GC
00007FF88DD20000	Managed Heap	64 K	60 K	60 K	60 K	60 K				4	Execute/Read/Write	Shared Domain
00007FF88DD30000	Managed Heap	64 K	56 K	56 K	56 K	56 K				3	Execute/Read/Write	Domain 1
00007FF88DD40000	Managed Heap	576 K	48 K	48 K	48 K	48 K				8	Execute/Read/Write	Domain 1 Virtual Call Stub
00007FF88DDD0000	Managed Heap	448 K	20 K	20 K	20 K	20 K				10	Execute/Read/Write	Shared Domain Virtual Call Stub
00007FF88DE80000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 Low Frequency Heap
00007FF88DE90000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 Low Frequency Heap
00007FF88DEA0000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88DEB0000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88DEC0000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 Low Frequency Heap
00007FF88DED0000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88DEF0000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88DF00000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88DF10000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88DF20000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88DF30000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88DF40000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88DF50000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88DF60000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88DF70000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88DF80000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88DF90000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88DFA0000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88DFB0000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88DFC0000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88DFD0000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88DFE0000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88DFF0000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88E000000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E010000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E020000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E030000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E040000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88E050000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E060000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E070000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E080000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E090000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88E0A0000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E0B0000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E0C0000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 Low Frequency Heap

Рис. 4.14 ❖ Сценарий 4.4 – представление управляемых куч в VMMap

Мы можем подтвердить это, вернувшись в системный монитор и добавив такие счетчики:

- `\Загрузка CLR .NET(имя процесса)\Байт в куче загрузчика`;
- `\Загрузка CLR .NET(имя процесса)\Загружено классов на текущий момент`;
- `\Загрузка CLR .NET(имя процесса)\Текущее число сборок`;
- `\Загрузка CLR .NET(имя процесса)\Текущих доменов приложений`.

Первые три счетчика постоянно растут, поэтому мы, по всей видимости, нашли истинную причину утечки памяти. Какая-то часть программы загружает десятки динамических сборок. К сожалению, мы не сможем глубоко проанализировать такую утечку памяти с помощью коммерческих инструментов типа JetBrains dotMemory или .NET Memory Profiler (по крайней мере, так было на момент написания книги). Хотя эта утечка связана со средой выполнения .NET, рост памяти в этих инструментах часто показывается как «unidentified» (неопределенная память), и даже возможности углубиться в детали не предоставляется. И снова на помощь приходят ETW и PerfView! На этот раз нас интересуют события, связанные с загрузкой сборок. Отследить их можно, воспользовавшись полем **Additional Providers** (Дополнительные поставщики) в диалоговом окне **Collect**. Введите в него `Microsoft-Windows-DotNETRuntime:LoaderKeyword:Always:@StacksEnabled=true`. Это

означает, что нам интересны события, связанные с загрузчиком, и при каждом возникновении события мы хотим регистрировать стеки вызовов. Начните сбор данных и наберитесь терпения (например, подождите столько времени, чтобы в счетчике Текущее число сборок стала видна загрузка нескольких новых сборок).

Когда сбор данных остановится и вся обработка завершится, откройте список Events и найдите в нем события типа `Microsoft-Windows-DotNETRuntime/Loader/AssemblyLoad` для нашего процесса (рис. 4.15).

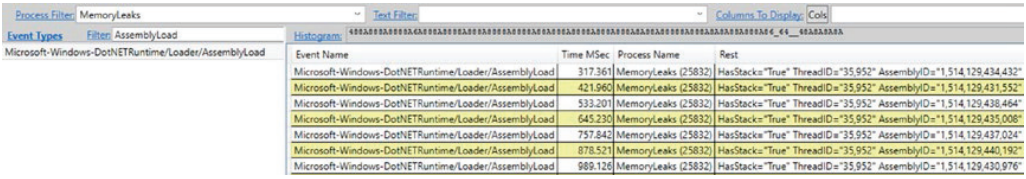


Рис. 4.15 ❖ Сценарий 4.4 – представление событий в PerfView. Видно очень много событий AssemblyLoad

Выберите одно из них и выполните команду **Open Any Stacks** (Открыть все стеки) из контекстного меню его столбца **Time MSec** (стек не отобразится, если щелкнуть правой кнопкой мыши по любому другому столбцу). Сгруппировав не интересующие нас модули (например, `clr`, `mscoree` или `mscoreei`, относящиеся к среде выполнения .NET) и развернув наши собственные модули, мы сразу же выявим источник создания динамических сборок (рис. 4.16). Это конструктор класса `XmlSerializer`, вызываемый из нашего метода `XmlSerializerLeak.Run()`.

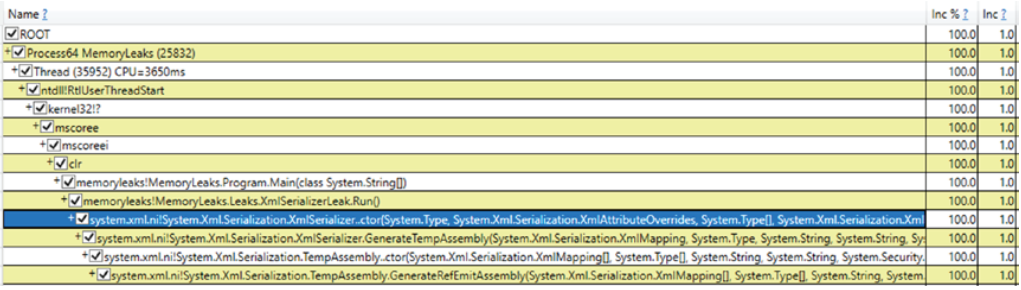


Рис. 4.16 ❖ Представление стека вызовов в PerfView для одного события AssemblyLoad указывает на конструктор XmlSerializer

Мы нашли, в чем проблема! И действительно, в документации MSDN по классу `XmlSerializer` читаем:

Для повышения производительности инфраструктура сериализации XML динамически создает сборки для сериализации и десериализации указанных типов. Инфраструктура находит и повторно использует эти сборки. Это поведение возникает только при применении следующих конструкторов:

** XmlSerializer.XmlSerializer(Type)*

** XmlSerializer.XmlSerializer(Type, String)*

При использовании любого из других конструкторов несколько версий одной и той же сборки создаются и никогда не выгружаются, что приводит к утечке памяти и снижению производительности. Самым простым решением является использование одного из упомянутых выше двух конструкторов. В противном случае необходимо кешировать сборки в Hashtable, как показано в следующем примере.

В нашем случае, как доказывает рис. 4.16, применен конструктор, который не поддерживает повторное использование ранее сгенерированных сборок, отсюда и наблюдаемая утечка памяти.

ПРИМЕЧАНИЕ Такая же проблема может возникнуть и в других ситуациях, где динамически создаются сборки, например при вызове `AppDomain.CreateDomain` без последующей выгрузки домена или при работе с различными скриптовыми движками, которые создают сборки, содержащие откомпилированные скрипты.

Система типов

Тип – фундаментальное понятие в CLI. Как написано в стандарте ECMA 335, он служит для «описания значения и формулирования контракта, который обязаны соблюдать все значения этого типа». О самой *общей системе типов* можно говорить долго. Но для целей управления памятью будет достаточно интуитивно понятного определения типа, с которым мы постоянно имеем дело при написании программ на C# или любом другом языке. Впрочем, позже мы еще вернемся к разговору о различных категориях типов в .NET.

Каждый тип в .NET описывается структурой данных, которая называется *таблица методов* (MethodTable). Она содержит много информации о типе, в том числе и той, что важна нам:

- GCInfo – структура данных, необходимая сборщику мусора (в следующих главах мы ее, конечно, рассмотрим);
- флаги – описывают различные свойства типа;
- базовый размер экземпляра – размер объекта;
- ссылка на EEClass – там хранятся «холодные» данные, которые обычно необходимы только для загрузки типа, JIT-компиляции или отражения, в т. ч. описания всех методов, полей и интерфейсов;
- описания всех методов (включая унаследованные), необходимые для их вызова;
- данные, относящиеся к статическим полям, – включают в себя данные, связанные с примитивными статическими полями (подробнее о статических полях мы поговорим ниже в этой главе).

Среда выполнения использует адрес записи в таблице MethodTable (он называется TypeHandle), когда ей нужно получить информацию о загруженном типе. Мы неоднократно встретим такие примеры в этой книге, потому что MethodTable – один из столпов взаимодействия между движком выполнения и сборщиком мусора.

Категории типов

Едва ли не в любой статье, посвященной памяти в .NET, повторяется одна и та же мантра: «*существуют типы значений, память для которых выделяется в стеке, и ссылочные типы, выделяемые в куче*». И еще одна: «*классы – это ссылочные типы, а структуры struct – типы значений*». На собеседованиях очень любят задавать вопросы на эту тему. Но это отнюдь не самый лучший способ пояснить разницу между *типами значений* и *ссылочными типами*. Почему же? Потому что при таком подходе концепция описывается с точки зрения реализации, не пытаясь объяснить истинное различие между двумя категориями типов.

Мы займемся деталями реализации позже, а пока просто отметим, что это детали реализации, и не более того. И как любая реализация некоторой абстракции, она может измениться. Важна лишь сама абстракция, предлагаемая программисту. Поэтому я отброшу подход, основанный на реализации, и попытаюсь объяснить, что за ним стоит. И лишь после этого мы сможем по-настоящему понять, при каких условиях текущая реализация возможна (и разумна).

Начнем с самого начала, каковым является стандарт ECMA 335. К сожалению, необходимые нам определения несколько расплывчаты, и легко запутаться в различных значениях слов *тип*, *значение*, *тип значения*, *значение типа* и т. д. Вообще говоря, полезно запомнить, что в этом стандарте говорится, что «любое значение, описываемое типом, называется *экземпляром* этого типа». Иными словами, мы можем говорить о *значении* (или, что то же самое, *экземпляре*) как типа значений, так и ссылочного типа. А определяются они дальше:

> *тип значений: такой тип, экземпляр которого непосредственно содержит все свои данные. (...) Значения, описываемые типом значений, не нуждаются ни в чем стороннем;*
>
> *тип ссылочный: тип, значение которого содержит ссылку на его данные. (...) Значение, описываемое ссылочным типом, указывает местоположение другого значения.*

Отсюда понятно истинное различие между абстракциями этих двух категорий типов: экземпляры (значения) типа значений содержат все его данные «здесь же» (фактически они и являются значениями), тогда как значения ссылочных типов лишь указывают на данные, находящиеся «где-то еще» (ссылаются на что-то). Но у этой абстракции местоположения есть очень важные следствия, которые соотносят ее с другими фундаментальными вещами.

Время жизни:

- значение типа значений содержит все его данные – мы можем рассматривать его как единую, замкнутую в себе сущность. Данные живут столько, сколько сам экземпляр типа значений;
- значение ссылочного типа лишь обозначает местоположение другого значения, время жизни которого не определяется временем жизни экземпляра типа.

Разделение:

- значение типа значений не может быть общим – если мы хотим использовать его в другом месте (например, хотя это и деталь реализации, в качестве аргумента метода или значения другой локальной переменной), то его нуж-

но будет скопировать байт в байт. В этом случае говорят о семантике *передачи по значению*. А поскольку в другое место передается копия значения, время жизни оригинала не изменяется;

- значение ссылочного типа может быть общим – если мы хотим использовать его в другом месте, то по умолчанию применяется семантика *передачи по ссылке*. После этого на одно и то же местоположение, где находится значение, будут указывать два экземпляра. Необходимо каким-то образом отслеживать все такие ссылки, чтобы знать время жизни значения (см. главу 1).

Идентичность:

- значения типов значений идентичны тогда и только тогда, когда состоят в точности из одних и тех же битов в одном и том же порядке;
- ссылочные типы идентичны тогда и только тогда, когда указывают на одно и то же местоположение.

Повторю, в данном контексте нет ни единого упоминания о стеке или куче. Знание этих различий и определений немного проясняет картину, хотя нужно время, чтобы к ним привыкнуть. В следующий раз, когда вас спросят на собеседовании, где хранятся типы значений, вы сможете ответить нестандартно, пустившись в пространные объяснения.

Есть еще одна категория типов, о которой следует знать, – *неизменяемые* (immutable) *типы*. Так называется тип, значение которого нельзя изменить после создания. Не больше, не меньше. Здесь нет ни слова о семантике значения или ссылки. Иначе говоря, неизменяемыми могут быть как типы значений, так и ссылочные типы. В объектно-ориентированном программировании для гарантии неизменяемости нужно просто не делать открытыми методы и свойства, которые позволили бы изменить значение объекта.

Хранение типов

Но все же, откуда следует, что для этих двух категорий типов нужно использовать именно стек и кучу? Да ниоткуда! Так Майкрософт решила реализовать стандарт .NET Framework CLI. Поскольку на протяжении многих лет эта реализация была преобладающей, рассказ о том, что «память для типов значений выделяется в стеке, а для ссылочных типов в куче», как мантра, повторялся снова и снова без глубокого осмысления. А поскольку это проектное решение оказалось очень хорошим, оно было повторено в различных реализациях CLI, которые мы обсуждали выше. Но имейте в виду, что вообще-то это утверждение не всегда верно. В следующих разделах мы увидим, что есть исключения из правила. В зависимости от местоположения к хранению значений могут быть разные подходы. И как мы скоро увидим, это в точности случай с CLI.

Тем не менее рассуждать о хранении типов значений и ссылочных типов можно только при проектировании реализации CLI для конкретной платформы. Дело в том, что на некоторых платформах стека или кучи может вообще не быть. Поскольку в подавляющем большинстве современных компьютеров есть и то, и другое, решение принять просто. Но ведь у процессора, вероятно, есть еще и регистры, а про них даже не упоминается в заклинании «память для типов значений выделяется в …», хотя это деталь реализации того же уровня, что использование стека или кучи.

Истина заключается в том, что решение о хранении того или иного типа может быть отнесено в основном к проектированию JIT-компилятора. Этот компонент проектируется для конкретной платформы, и мы знаем, какие ресурсы на ней присутствуют. В архитектуре x86/x64 к нашим услугам и стек, и куча, и регистры. Однако решение о том, где сохранять значение данного типа, необязательно оставлять на усмотрение одного лишь JIT-компилятора. Мы можем позволить компилятору влиять на это решение, основываясь на проведенном им анализе. А можем предоставить решение разработчику, вынеся его на уровень языка (как в языке C++, где память для объектов можно выделять как в стеке, так и в куче).

Существует даже более простой подход, принятый в Java, где никаких пользовательских типов значений нет вовсе, поэтому вопрос о том, где их хранить, не стоит! В языке имеется несколько встроенных примитивных типов (целые и т. п.), они называются типами значений, а все остальное выделяется в куче (без учета анализа локальности, о котором речь ниже). При проектировании .NET тоже можно было бы принять решение о том, что экземпляры всех типов выделяются в куче, и в этом не было бы ничего плохого, при условии что не нарушается семантика типа значений и ссылочного типа. В части размещения в памяти стандарт ECMA-335 дает полную свободу.

> *Четыре части состояния метода – массив входных аргументов, массив локальных переменных, пул локальной памяти и стек вычислений – специфицированы как логически различные области. Совместимая со стандартом реализация CLI вправе отобразить эти области в один непрерывный массив в памяти, хранить в кадре стека, определенном в целевой архитектуре, или использовать любой другой эквивалентный способ представления.*

Почему были приняты именно такие, а не иные решения о реализации, будет уместнее объяснить в следующих разделах, разделив обсуждение типов значений и ссылочных типов.

Осталось сделать одно важное замечание. Хотя теперь мы знаем, что упоминание стека и кучи – деталь реализации, поговорить об этом все равно имеет смысл. К сожалению, есть место, где «как должно быть» вступает в противоречие с «как практично». И это место – оптимизация производительности и использования памяти. Если мы пишем на C# код, рассчитанный на платформу x86/x64 или ARM, то точно знаем, что в определенных обстоятельствах для таких-то типов будет использована куча, стек или регистры. Так что, по закону протекающих абстракций (см. главу 2), абстракция типов значений и ссылочных типов может дать здесь течь. И если мы пожелаем, то можем воспользоваться этим из соображений производительности (что станет наглядно видно в главе 14, где описываются различные продвинутые сценарии оптимизации).

Типы значений

Как было сказано выше, тип значений «непосредственно содержит свои данные». Стандарт ECMA 335 определяет значение следующим образом:

> *Простая комбинация битов для целого числа, числа с плавающей точкой или еще чего-то подобного. У каждого значения есть тип, который описывает как*

занимаемую им область памяти, так и семантику битов в его представлении. Значения служат для представления простых типов и не-объектов в языках программирования.

В общеязыковой спецификации (Common Language Specification) определены две категории типов значений:

- *структуры* – различные встроенные целые типы (char, byte, integer и т. д.), типы с плавающей точкой и логический тип bool. И конечно, пользователь может определять собственные структуры;
- *перечисления* – это обобщение целых типов, а именно тип, состоящий из набора именованных констант. С точки зрения управления памятью, это просто целый тип, и в этой книге мы не будем специально останавливаться на них, т. к. на внутреннем уровне они рассматриваются как структуры.

Хранение типов значений

Ну а как насчет части «*типы значений хранятся в стеке*»? С точки зрения реализации, ничто не мешает хранить все типы значений в куче. За исключением одной мелочи – хранить в стеке или в регистре ЦП лучше. Как было описано в главе 1, накладные расходы на хранение в стеке очень малы. Чтобы «выделить» и «освободить» объект, нужно просто создать кадр стека подходящего размера и отбросить его, когда в объекте отпадет необходимость. Но раз стек такой быстрый, то только его и надо использовать, разве не так? Проблема в том, что это не всегда возможно, в основном из-за различия во времени жизни данных в стеке и желательного времени жизни самого значения. Именно продолжительность жизни и возможность разделения значения определяют, какой механизм использовать для хранения данных типа значений.

Теперь рассмотрим все места, где может встречаться тип значений, и подходящие для каждого места виды памяти.

- Локальные переменные в методе – их время жизни точно определено и совпадает с протяженностью вызова метода (включая все вызываемые из него методы). Мы могли бы выделить память для всех локальных переменных типа значений в куче, а затем освободить ее по завершении вызова. Но можно было бы использовать и стек, потому что мы знаем, что у каждого значения есть только один экземпляр (и никакого разделения). Поэтому нет риска, что кто-то попытается использовать значение по завершении метода или параллельно из другого потока. Так что для хранения локальных типов значений запись активации метода подойдет идеально. Кроме того, CLI ясно говорит, что «наличие управляемого указателя, ссылающегося на локальную переменную или параметр, может привести к тому, что ссылка будет жить дольше самой переменной, так что это не поддается проверке» (мы вернемся к управляемым указателям в главе 14).
- Аргументы метода – их можно рассматривать так же, как локальные переменные, и, стало быть, использовать для хранения стек, а не кучу.
- Поле экземпляра ссылочного типа – его время жизни зависит от времени жизни объемлющего значения. Очевидно, что оно может жить дольше, чем текущая или любая другая запись активации, так что стек – не подходящее место для него. Поэтому типы значений, являющиеся полями ссылочного типа (например, класса), выделяются вместе с последним в куче.

- Поле экземпляра другого типа значений. Здесь ситуация несколько сложнее. Если содержащее такой тип значение находится в стеке, то для хранения будет использоваться стек. А если оно уже находится в куче, то и значение поля будет выделено в куче.
- Статическое поле (класса, интерфейса или модуля) – ситуация аналогична полю экземпляра ссылочного типа. Время жизни статического поля совпадает со временем жизни типа, в котором оно определено. Это означает, что для его хранения стек не подходит, т. к. запись активации живет гораздо меньше.
- Локальный пул памяти – его время жизни неразрывно связано со временем жизни метода (ECMA говорит, что «локальный пул памяти возвращается при выходе из метода). Следовательно, мы можем без проблем использовать стек, и потому локальный пул памяти реализован как увеличение записи активации.
- Временно в стеке вычислений – время жизни значения стеке вычислений строго контролируется JIT-компилятором. Он точно знает, зачем нужно это значение и когда оно будет использовано. Поэтому у него есть полная свобода выбора: куча, стек или регистр. Из соображений производительности он, очевидно, предпочтет регистры или стек, если это возможно.

Итак, мы разобрались с первой частью: «типы значений хранятся в стеке». Как видим, правильнее другое утверждение: «тип значений хранится в стеке, если значение является локальной переменной или находится в локальном пуле памяти. Но он хранится в куче, если является частью другого объекта, хранящегося в куче, или статическим полем. И может храниться в регистре, если создается в процессе обработки стека вычислений». Немного сложнее, правда? Но и это еще не вся правда, поскольку, как мы увидим, *замыкания* захватывают локальные переменные, помещая их в контекст ссылочного типа, так что память для них выделяется в куче, а не в стеке.

Структуры

Структуры – пожалуй, один из самых недооцененных элементов C#, хотя они существуют с самой первой версии .NET. По-видимому, это объясняется следующими причинами:

- трудно понять, в чем смысл существования структур, если свести их к формуле «типы значений хранятся в стеке»;
- для структур действует много ограничений (невозможно определить конструктор без параметров, не допускается наследование);
- классов и так вполне хватает, и у нас не возникает потребности что-то менять в этом отношении;
- поскольку для структур реализована семантика передачи по значению, мы полагаем, что передача их в качестве параметров методам или присваивание переменным приведет к низкой производительности (что, как мы вскоре увидим, в общем случае неправда);
- их поведение не всегда очевидно, а учитывая отсутствие явной необходимости, отпадает всякое желание их использовать.

Так зачем все-таки нам могут понадобиться структуры? Вот их основные преимущества:

- память для них можно выделять в стеке, а не в куче. Да, здесь абстракция протекает, но мы можем использовать эту деталь реализации к собственной выгоде. Выделение памяти в стеке избавляет нас от накладных расходов, связанных с уборкой таких типов в мусор, что всегда хорошо;
- они меньше по размеру. Поскольку в структуре хранятся только данные без каких-либо метаданных, они занимают меньше памяти, чем классы. И хотя память дешева, при больших объемах данных это может стать весомым преимуществом;
- они улучшают локальность данных – поскольку структуры меньше, мы можем более плотно упаковать данные в коллекциях (что будет наглядно продемонстрировано ниже). А это, как мы знаем, хорошо с точки зрения использования кеша;
- доступ к ним осуществляется быстрее – данные хранятся прямо в структуре, поэтому не нужно никакого разыменования;
- они изначально обладают семантикой передачи по значению – если мы хотим создать неизменяемый тип, то структура – отличный кандидат. Но для структур возможна и семантика передачи по ссылке (в чем мы скоро убедимся), так что мы получаем лучшее из обоих миров.

Далее мы подробно рассмотрим эти преимущества, поскольку использование структур – один из самых распространенных и эффективных приемов оптимизации памяти и производительности. Мы уделим им особое внимание в главах 13 и 14, когда будем описывать передачу по ссылке с помощью ключевых слов `in`, `out` и `ref` (особенно в контексте таких типов, как `Span<T>`). Но прежде продолжим краткое общее введение.

Общие сведения о структурах

Структуру можно рассматривать просто как тип, описывающий размещение данных в памяти, вместе с методами, которые можно применять к экземплярам. Экземпляр структуры содержит только ее данные (в полном соответствии с определением типа значений), поэтому если определить структуру, как показано в листинге 4.6, то ее представление в памяти будем таким, как на рис. 4.17 (и в 32-, и в 64-разрядных архитектурах). Структуре необходимо место для четырех целых чисел, так что она занимает 16 байт.

> В 32-разрядных системах стандарт де-факто называется *ILP32*, т. е. длина значений типа `int`, `long`, а также указателя равна 32 битам. В 64-разрядных системах имеется небольшое различие между Windows и Linux. В Unix действует стандарт *LP64* – длина `long` и указателя равна 64 битам (но длина `int` по-прежнему 32 бита). В Windows 64-разрядный стандарт называется *LLP64* – специальный тип «двойной длины» (`long long`) и указатель занимают 64 бита (но длина `long` и `int` равна 32 битам).

Листинг 4.6 ❖ Определение демонстрационной структуры

```
public struct SomeStruct
{
   public int Value1;
   public int Value2;
   public int Value3;
   public int Value4;
}
```

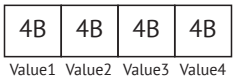


Рис. 4.17 ❖ Размещение структуры из листинга 4.6 в памяти

В зависимости от того, в каком месте программы структура используется (и от конкретной реализации), такая область памяти может находиться в стеке или в куче (и даже, как мы увидим, в регистре). Но в современных реализациях CLR она не может размещаться в управляемой куче непосредственно. В управляемой куче допускаются только объекты самодокументированных ссылочных типов. Поэтому, если возникает необходимость сохранить структуру в куче, применяется так называемая *упаковка* (boxing). Об упаковке у нас еще будет случай подробно поговорить ниже в этой главе. Мы также расскажем, как размещение данных в памяти зависит от полей типа, – здесь и в главе 13, поскольку в обоих местах затрагиваются структуры и классы.

Сейчас нам интересно использование структур с точки зрения управления памятью. Если структура подверглась упаковке (ее копия размещается в куче), то извлекать какие-то выгоды уже, пожалуй, слишком поздно. Преимущества структур в полной мере проявляются при работе с неупакованными данными. Иными словами, мы хотим воспользоваться тем фактом, что они размещены не в куче. Одно из главных правил гласит «Избегайте выделения памяти», а структуры как раз и являются одним из механизмов достижения этой цели. Кроме того, вследствие многочисленных ограничений на структуры, например запрета наследования, компилятор и (или) JIT-компилятор могут сделать различные заключения об их использовании. Наследование же подразумевает возможность виртуальных вызовов и полиморфизма, поэтому понять, как будут использоваться данные в конечном итоге, гораздо труднее[1].

Хранение структур

Рассмотрим пример класса в листинге 4.7, в котором используется структура, определенная в листинге 4.6. Мы видим, что в методе `Main` имеется локальная переменная `sd`, в которой хранится экземпляр структуры `SomeStruct`. И вот что можно сказать об этой структуре на основе уже известной нам информации:

- экземпляр `sd` передается методу `Helper` по значению, что, вероятно, означает копирование его данных. `Helper` работает с собственной копией данных, поэтому их изменение никак не отразится на оригинальном значении `sd`;
- `sd` – локальная переменная типа значений, поэтому память для нее (скорее всего) будет выделена в стеке, а не в куче.

Листинг 4.7 ❖ Пример метода, в котором используется структура из листинга 4.6

```
public class ExampleClass
{
   public int Main(int data)
   {
```

[1] Правда, на момент написания этой книги уже были планы добавить в .NET так называемую *девиртуализацию*, т. е. механизм, позволяющий во время компиляции определить, какой именно метод будет вызван.

```
      SomeStruct sd = new SomeStruct();
      sd.Value1 = data;
      return Helper(sd);
   }

   private int Helper(SomeStruct arg)
   {
      return arg.Value1;
   }
}
```

Взглянув на CIL-код метода `Main`, например с помощью dnSpy (см. листинг 4.8), мы увидим, как он компилируется в код для стековой машины, работающей со стеком вычислений, и какие команды исполняются на каждом шаге:

- `ldloca.s 0` – адрес первой локальной переменной (с индексом 0) помещается в стек вычислений;
- `initobj Samples.SomeStruct` – область памяти по адресу, полученному (и удаленному) из стека вычислений, инициализируется как значение типа `SomeStruct` (в соответствии с документацией MSDN `initobj` «Инициализирует каждое поле типа значения с определенным адресом пустой ссылкой или значением 0 соответствующего примитивного типа»);
- `ldloca.s 0` – адрес первой локальной переменной снова помещается в стек вычислений;
- `ldarg.1` – второй аргумент метода помещается в стек вычислений (это данные типа `int`, а первый аргумент по умолчанию является ссылкой на экземпляр класса);
- `stfld int32 Samples.SomeStruct::Value1` – сохранить значение первого элемента стека вычислений в поле `SomeStruct.Value1` по адресу, указанному во втором элементе стека вычислений. Оба элемента удаляются из стека вычислений;
- `ldarg.0` – первый аргумент метода (сам экземпляр класса, который в C# называется `this`) помещается в стек вычислений;
- `ldloc.0` – значение первой локальной переменной помещается в стек вычислений. Теперь мы можем считать, что все 16 байт структуры `SomeStruct` скопированы и далее метод `Helper` будет работать именно с ними;
- `call instance int32 Samples.ExampleClass::Helper(valuetype Samples.SomeStruct)` – вызвать метод `Helper` и поместить результат в стек вычислений;
- `ret` – вернуть управление вызывающей программе.

Листинг 4.8 ❖ Результат компиляции метода Main из листинга 4.7 на промежуточный язык CIL

```
.method public hidebysig instance int32 Main (int32 data) cil managed
{
   // Метод начинается в  RVA 0x2048
   // Размер кода 24 (0x18)
   .maxstack 2
   .locals init (
      [0] valuetype Samples.SomeStruct
   )

   IL_0000: ldloca.s 0
   IL_0002: initobj Samples.SomeStruct
```

```
   IL_0008: ldloca.s 0
   IL_000a: ldarg.1
   IL_000b: stfld int32 Samples.SomeStruct::Value1
   IL_0010: ldarg.0
   IL_0011: ldloc.0
   IL_0012: call instance int32 Samples.ExampleClass::Helper(valuetype
            Samples.SomeStruct)
   IL_0017: ret
} // окончание метода ExampleClass::Main
```

В листинге 4.8 структура используется в трех местах: локальная переменная, аргумент метода и сам стек вычислений. Но нигде нет выделения памяти в куче (для этого служит команда `newobj`, с которой мы познакомимся в листинге 4.13)! Это и есть желаемая оптимизация. Можно ожидать, что SomeStruct будет выделена в стеке и скопирована в запись активации `Helper` при вызове. Отсюда с очевидностью следует, что мы должны хорошенько подумать, так ли выгодно использование структур (однако смотрите примечание ниже).

> Копирование данных структуры вследствие семантики передачи по значению может свести на нет весь выигрыш, полученный благодаря отказу от выделения памяти в куче. Однако есть два момента, из-за которых все-таки имеет смысл серьезно рассматривать применение структур при написании высокопроизводительного кода:
> – зачастую JIT-компилятор оптимизирует работу с данными небольшого размера, так что используются только регистры ЦП без обращения к стеку (как показано в следующих абзацах);
> – популярный обходной способ заключается в передаче структуры по ссылке, что тоже возможно (благодаря ключевым словам `ref`, `in` и `out`, которые будут подробно объяснены позже).

Все это замечательно, и можно было бы на этом поставить точку. Однако очень интересно посмотреть, как код для абстрактной стековой машины преобразуется JIT-компилятором в настоящий машинный код. Как вышеупомянутые три места отображаются на стек, кучу и регистры процессора? Очевидно, это зависит от конкретного JIT-компилятора, поэтому будем рассматривать только наиболее популярную комбинацию: RyuJIT для .NET Framework на платформе x64. Результат, показанный в листинге 4.9, очень радует. JIT-компилятор сумел вообще избавиться от работы со стеком вычислений, заметив, что можно обойтись всего одной командой `mov`! Вот что делает этот код:

- `mov eax, edx` – перемещает второй аргумент (который передается в регистре `edx` согласно соглашению о вызове Microsoft x64) в регистр `eax`, который должен содержать результат метода;
- `ret` – возвращает управление из метода.

Обращения к методу `Helper` нет вообще (оно встроено), нет также копирования данных структуры, да и вообще структуры!

Листинг 4.9 ❖ Метод `Main` из листинга 4.7 после обработки JIT-компилятором RyuJIT x64

```
Samples.ExampleClass.Main(Int32)
0x00007FFA`5178BA40:   L0000: mov eax, edx
0x00007FFA`5178BA42:   L0002: ret
```

Можно было бы возразить, что так получилось только потому, что метод `Helper` тривиален. Но на самом деле память для `SomeStruct`, скорее всего, не будет выделена в стеке, даже если внутри Helper производится более сложная обработка и используются все поля. Это тот уровень изощренности, которого достигли современные алгоритмы JIT-компиляции.

В чем я хотел бы вас убедить? В том, что структуры – эффективные контейнеры данных, поскольку благодаря их простоте возможны далеко идущие оптимизации кода. В том, что «локальные переменные структурного типа выделяются в стеке», много истины, но, как мы видим, на практике все может обстоять даже лучше. Локальные переменные могут быть размещены в регистрах, так что к стеку вовсе не придется обращаться. Даже если мы ожидаем, что передача структуры по значению влечет за собой копирование данных, JIT-компилятор может оптимизировать код, сведя его к работе с регистрами ЦП.

> Оптимизация, показанная в листинге 4.9, производится при компиляции в режиме выпуска, поскольку именно в нем разрешены все оптимизации. Если бы мы компилировали код из листинга 4.7 в отладочном режиме, то методу `Main` соответствовал бы ассемблерный код, насчитывающий 41 строку, и метод `Helper` не был бы встроен (и занял бы еще 25 строк ассемблерного кода). То есть в отладочном режиме мы вместо двух команд имели бы 66!

Осталось сделать еще одно очень важное замечание. Среда выполнения .NET оптимизирует структуры по-разному в зависимости от их размера. Так, если бы мы добавили в `SomeStruct` еще одно целое поле, то JIT-компилятор не смог бы оптимизировать метод `Main`. Произошло бы и выделение памяти в стеке, и копирование данных. Граница, разделяющая способы обработки, – еще одна деталь реализации, но можно считать, что она проходит в районе 24 байтов. Так что можно без опаски предполагать, что описанные оптимизации имеют место для структур не длиннее 16 байт, но я думаю, что и 24 байта нормально.

> Копирование данных в таких случаях также в какой-то мере оптимизируется с использованием всех возможностей процессора. Например, на моем процессоре Intel 4-го поколения Haswell данные копируются командой `vmovdqu`. Эта команда, входящая в расширенный набор *AVX* (*Advanced Vector Extensions*) для векторных вычислений, копирует вектор целых значений из одной невыровненной области памяти в другую. Тем не менее если мы стремимся к наивысшей производительности, то нужно по возможности избегать копирования вообще.

Любопытный факт, быть может, уже известный вам, – в методе структуры можно присвоить новое значение полю `this`. Хотя, с точки зрения языка, это выглядит курьезно, в следующем примере не возникает никаких сложностей в плане управления памятью:

```
public struct SomeData
{
   public int Value1;
   public int Value2;
   public int Value3;
   public int Value4;

   public void Bizzarre()
   {
```

```
      this = new SomeData();
   }
}
```

Поскольку данные типов значений хранятся «здесь же», то такое присваивание можно рассматривать как повторную инициализацию полей структуры.

> При определении структуры, наверное, лучше всего делать ее неизменяемой. Когда объект передается методу и в нем производится присваивание новых значений полям, у читателя кода может сложиться впечатление, будто модифицируется исходное значение. Что, как мы знаем, неверно, т. к. типы значений реализуют семантику передачи по значению. Лучше сделать объект неизменяемым, чем явно писать в комментариях, что его не следует изменять. Например, можно оставить для всех полей только свойства для чтения, а в методах не изменять данные – так мы избежим неожиданного поведения.

Ссылочные типы

Как мы уже говорили, ссылочным называется тип, экземпляр которого содержит ссылку на данные. В общеязыковой спецификации определены две категории ссылочных типов.

- *Тип объекта* – как гласит стандарт ECMA 335, объект – это «ссылочный тип самодокументированного значения», и «его тип явно хранится в его представлении». К таковым относятся хорошо известные классы и делегаты. Существует несколько встроенных ссылочных типов, самым известным из них, конечно же, является `Object`.
- *Тип указателя* – это просто машинный адрес ячейки памяти (см. главу 1). Указатели могут быть управляемыми и неуправляемыми. Управляемые указатели подробно описаны в главе 13, потому что играют важную роль в реализации семантики передачи по ссылке.

При обсуждении ссылочных типов удобно считать, что они состоят из двух сущностей (рис. 4.18):

- *ссылка* – значением ссылочного типа является ссылка на его данные, т. е. адрес данных, хранящихся где-то в другом месте. Саму ссылку можно рассматривать как тип значений, потому что ее внутреннее представление – просто 32- или 64-разрядный адрес. Ссылки обладают семантикой передачи по значению, т. е. копируются;
- *данные ссылочного типа* – это область памяти, на которую указывает ссылка. Стандарт ничего не говорит о том, где должны храниться данные. Просто где-то в другом месте.

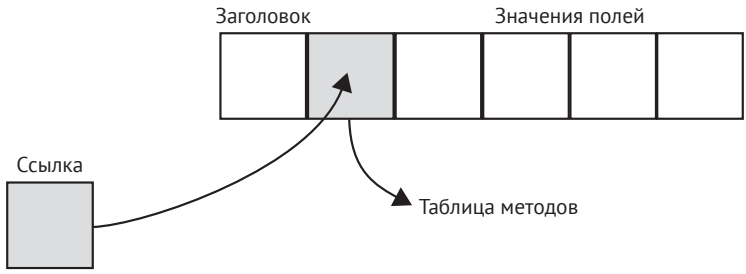


Рис. 4.18 ❖ Схематическое изображение ссылочного типа

На рис. 4.18 повторен рис. 1.10 из главы 1, на котором были показаны указатели и данные, на которые они указывают. И это не случайно, т. к. ссылки можно рассматривать как указатели с дополнительными мерами безопасности, обеспечиваемыми средой выполнения.

Рассматривать возможные способы хранения ссылочных типов проще, чем в случае типов значений. Как уже отмечалось, поскольку ссылки могут использоваться для разделения данных, время их жизни не имеет точного определения. В общем случае хранить ссылочные типы в стеке нельзя, т. к. их время жизни может оказаться значительно дольше времени жизни записи активации (существующей, пока метод не вернул управление). Поэтому напрашивается очевидное решение, о котором и говорится во фразе «ссылочные типы хранятся в куче». Но в распоряжении среды выполнения .NET имеется несколько куч, так что даже это простое предложение не вполне точно.

И раз уж мы заговорили о выделении памяти для ссылочных типов в куче, то есть одно исключение. Если бы мы могли как-то узнать, что экземпляр ссылочного типа обладает такими же характеристиками, как локальная переменная типа значений, то можно было бы выделить память для него в стеке, как обычно делается для типов значений. В частности, это означает, что нужно знать, не видна ли ссылка за пределами своей локальной области видимости (стека или потока), т. е. не разделяет ли она общие данные с другими ссылками. Проверка этого факта называется *анализом локальности* (Escape Analysis) (листинг 4.10). Она успешно реализована в Java, где особенно полезна, т. к. в этом языке по умолчанию память почти для всего выделяется в куче. На момент написания этой книги в .NET-среде анализ локальности не поддерживался[1].

Листинг 4.10 ❖ Анализ локальности для метода `Helper` может обнаружить, что локальная переменная `c` не «покидает» метод и потому может быть безопасно выделена в стеке. В настоящее время такой анализ не реализован ни в одной среде выполнения .NET

```
private int Helper(SomeData data)
{
   SomeClass c = new SomeClass();
   c.Calculate(data);
   return c.Result;
}
```

Классы

Всякий, кто программирует на .NET-совместимом языке, использует чужие и объявляет собственные классы. Класс – это ссылочный тип, определенный пользователем. Он является полноправной сущностью в CTS и краеугольным камнем любого приложения на C#. Классы могут содержать поля, свойства, методы, статические поля и статические методы и т. д. Определим аналог структуры и листинга 4.6 в виде класса, чтобы понаблюдать за различиями между тем и другим.

---

[1] Однако эта функциональность разрабатывается и, вероятно, будет включена (по крайней мере, как опциональная) в .NET Core 3.0.

Листинг 4.11 ❖ Пример определения класса (с той же функциональностью, что у структуры из листинга 4.6)

```
public class SomeClass
{
   public int Value1;
   public int Value2;
   public int Value3;
   public int Value4;
}
```

Управление памятью в .NET спроектировано так, что каждый объект в памяти включает следующие части (их размеры зависят от разрядности архитектуры, см. рис. 4.19):

- *заголовок объекта* – место для «любой дополнительной информации, которую может понадобиться присоединить к произвольному объекту» (как говорится в комментариях к исходному коду CoreCLR). Часто заголовок просто содержит нули, но обычно он используется для хранения информации о блокировке, поставленной на объект, или для кеширования значения, вычисленного методом `GetHashCode`. Заголовок используется по принципу «кто первый встал, того и тапки». Если среде выполнения он понадобился для хранения блокировки, то хеш-код в нем уже не хранится. И так далее. Кроме того, это место играет важную роль для работы сборщика мусора;
- *ссылка на таблицу методов* – как уже было сказано, «тип объекта явно хранится в его представлении», и, с точки зрения реализации, это и есть таблица методов (MethodTable). Именно сюда указывают все внешние ссылки на объект, т. е. если на объект имеются какие-то ссылки, то все они содержат адрес хранящейся в нем ссылки на таблицу методов. Поэтому говорят, что заголовок объекта имеет «отрицательный индекс». Ссылка на таблицу методов сама является указателем на часть структуры, содержащей описание типа (она находится в высокочастотной куче домена, содержащего этот тип);
- факультативный заместитель данных, если в типе нет ни одного поля. Сборщик мусора настаивает на том, чтобы в каждом объекте было место хотя бы для одного поля такой длины, как указатель. Это поле используется для разных целей: как первое поле в нормальных объектах (на рис. 4.19 показано как поле `Value1`) или как длина коллекции в случае массивов. И еще оно очень важно для GC, в чем мы убедимся в главе 7.

Таким образом, в куче не может быть объектов, размер которых меньше суммарного размера этих трех полей (см. фрагмент в листинге 4.12, взятый из исходного кода CoreCLR). Это означает, что размер самого маленького объекта (вообще без полей) в куче равен 12 байтам в 32-разрядной среде выполнения:

- 4 байта для заголовка объекта;
- 4 байта (размер указателя) для ссылки на таблицу методов;
- 4 байта (размер указателя) для внутреннего заместителя данных, –

и 24 байтам в 64-разрядной среде выполнения:

- 8 байт для заголовка объекта – из которых на самом деле используются только 4, а остальные 4 заполнены нулями (т. к. в 64-разрядной архитектуре область памяти должна быть выровнена на 8-байтовую границу);
- 8 байт (размер указателя) для ссылки на таблицу методов;
- 8 байт (размер указателя) для внутреннего заместителя данных.

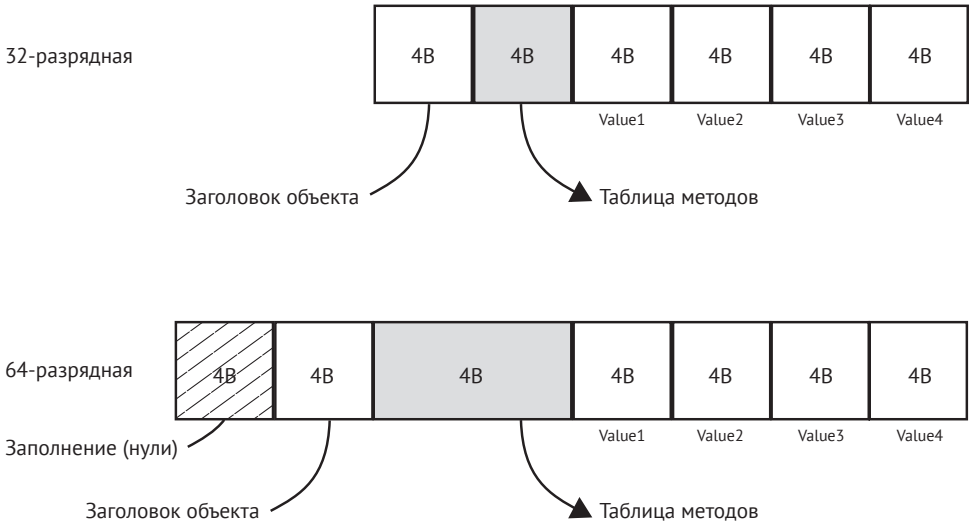


Рис. 4.19 ❖ Размещение класса из листинга 4.11 в памяти

Листинг 4.12 ❖ Минимальный размер объекта, выделенного в куче

```
// GC на основе поколений настаивает, чтобы размер каждого объекта был не меньше 12 байт
#define MIN_OBJECT_SIZE (2*sizeof(BYTE*) + sizeof(ObjHeader))
```

Мы подвергнем это различие тестированию производительности в разделе о локальности данных для типов, но издержки памяти очевидны. Структура, содержащая один байт, при размещении в стеке будет ровно столько памяти и занимать[1]. А класс, содержащий один байт, при размещении в куче будет занимать 24 байта (в случае 64-разрядной среды выполнения).

Рассмотрим теперь пример класса в листинге 4.13, в котором используется класс, определенный в листинге 4.11, – точно так же, как мы делали для структуры. Имеется метод `Main`, а в нем одна локальная переменная `sd` типа `SomeClass`. И вот что мы можем сказать по поводу этого класса.

- Данные, на которые ссылается локальная переменная `sd`, передаются методу `Helper` по ссылке, т. е. копирование данных не производится. Сама ссылка копируется, поскольку это просто адрес в памяти. Метод `Helper` работает с этой разделяемой ссылкой. Модификация значения внутри метода приведет к изменению значения `sd`.
- Данные, представленные переменной `sd`, – это локальная переменная ссылочного типа, поэтому память для них выделяется в куче, по крайней мере до тех пор, пока в .NET не будет реализован анализ локальности, который определил бы, что память можно безопасно выделить в стеке.

Листинг 4.13 ❖ Пример метода, в котором используется класс из листинга 4.11

```
public class ExampleClass
{
   public int Main(int data)
```

[1] Но выравнивание на границу в памяти может добавить накладных расходов. Размещение объекта в памяти, включая и влияние выравнивания, подробно описывается в главе 10.

```
   {
      SomeClass sd = new SomeClass();
      sd.Value1 = data;
      return Helper(sd);
   }

   private int Helper(SomeClass arg)
   {
      return arg.Value1;
   }
}
```

Теперь рассмотрим CIL-код метода `Main` (листинг 4.14). Стековая машина, работающая со стеком вычисления, шаг за шагом выполняет следующие инструкции:

- `newobj instance void Samples.SomeClass::.ctor()` – вызывается распределитель для создания нового экземпляра класса `SomeClass`, ссылка на который помещается в стек вычислений. Что тут происходит, мы будем подробно рассматривать в главе 6;
- `stloc.0` – ссылка удаляется с вершины стека и сохраняется в первой локальной переменной;
- `ldloc.0` – значение первой локальной переменной помещается в стек вычислений;
- `ldarg.1` – значение второго аргумента (напомним, что первым аргументом, как всегда, является ссылка `this`) помещается в стек вычислений;
- `stfld int32 Samples.SomeClass::Value1` – первый элемент стека вычислений сохраняется в поле `Value1` объекта, на который ссылается второй элемент стека вычислений (после чего оба элемента удаляются из стека);
- `ldarg.0` – значение первого аргумента (ссылка `this`) помещается в стек вычислений;
- `ldloc.0` – значение первой локальной переменной (ссылка на вновь созданный экземпляр `SomeClass`) помещается в стек вычислений;
- `call instance int32 Samples.ExampleClass::Helper(class Samples.SomeClass)` – вызывается метод `Helper`, который снимает два аргумента со стека вычислений (мы знаем это из его определения);
- `ret` – возврат из метода.

Листинг 4.14 ❖ Результат компиляции метода Main из листинга 4.13 на промежуточный язык CIL

```
.method public hidebysig instance int32 Main (int32 message) cil managed
{
   .locals init ([0] class Samples.SomeClass)
   IL_0000: newobj instance void Samples.SomeClass::.ctor()
   IL_0005: stloc.0
   IL_0006: ldloc.0
   IL_0007: ldarg.1
   IL_0008: stfld int32 Samples.SomeClass::Value1
   IL_000d: ldarg.0
   IL_000e: ldloc.0
   IL_000f: call instance int32 Samples.ExampleClass::Helper(class
   Samples.SomeClass)
   IL_0014: ret
} // конец метода ExampleClass::Main
```

Налицо небольшая избыточность – команда `stloc.0`, а затем сразу же `ldloc.0`. Очевидно, компилятор должен быть написан максимально общим способом, поэтому иногда встречается код, который можно было бы без труда оптимизировать.

Тем не менее ассемблерный код, сгенерированный JIT-компилятором в .NET Framework для архитектуры x64, очень прост и хорошо оптимизирован (листинг 4.15). Он мало что делает, кроме вызова внутренней функции распределителя `JIT_TrialAllocSFastMP_InlineGetThread`. И все же он гораздо сложнее тех двух строчек, которые были сгенерированы для программы из листинга 4.9, демонстрирующей использование структуры!

Листинг 4.15 ❖ Метод Main из листинга 4.13 после обработки JIT-компилятором RyuJIT x64

```
Samples.ExampleClass.Main(Int32)
0x00007FFA`5176E5A0:   L0000: push rsi
0x00007FFA`5176E5A1:   L0001: sub rsp, 0x20
0x00007FFA`5176E5A5:   L0005: mov esi, edx
0x00007FFA`5176E5A7:   L0007: mov rcx, 0x7ffa5192f838
0x00007FFA`5176E5B1:   L0011: call clr.dll!JIT_TrialAllocSFastMP_InlineGetThread+0x0
0x00007FFA`5176E5B6:   L0016: mov [rax+0x8], esi
0x00007FFA`5176E5B9:   L0019: mov eax, [rax+0x8]
0x00007FFA`5176E5BC:   L001c: add rsp, 0x20
0x00007FFA`5176E5C0:   L0020: pop rsi
0x00007FFA`5176E5C1:   L0021: ret
```

Как это различие отражается на производительности? Можно прогнать простой тест производительности для сравнения методов `Main` из листингов 4.7 и 4.13 (см. табл. 4.1). Из-за выделения памяти для объекта метод, в котором используется класс, более чем в четыре раза медленнее и, очевидно, выделяет память в куче, чего версия со структурой не делает.

Таблица 4.1. Результаты теста сравнения производительности методов Main из листингов 4.7 и 4.13. Использовалась библиотека BenchmarkDotNet на платформе .NET Framework 4.7

Метод	Среднее время	Поколение 0	Выделено
`ConsumeStruct`	0.6864 нс	–	0 Б
`ConsumeClass`	3.3206 нс	0.0076	32 Б

> В C++ синтаксис создания экземпляра класса позволяет выделять память для объекта как в стеке (`MyClass c`), так и в куче (`MyClass* c = new MyClass()`). Но в языке C++/CLI при создании экземпляра ссылочного типа якобы в стеке компилятор все равно выделяет память для него в куче (вызывая функцию `gcnew`).

Строки

Строка – хорошо известный ссылочный тип, представляющий последовательность символов, т. е. некоторый текст. В обычной .NET-программе это один из самых часто встречающихся типов, даже если мы не подозреваем об этом. А все потому, что большинство современных программ в большей или меньшей сте-

пени зависят от обработки текста. Не важно, откуда поступают данные – из базы, из веб-запросов по протоколу REST или SOAP или из XML-файлов, прочитанных с диска, – мы должны их получить, как-то обработать и вывести результат, скорее всего, в текстовой форме. Поэтому при анализе дампа памяти типичного .NET-приложения (в особенности работающего с вебом) строки всегда будут в начале списка встречающихся типов.

> Частое использование строк – типичное явление, поэтому если, анализируя потребление памяти программой, вы видите в ней много строк, не надо сразу же думать, будто это и есть корень зла. Это возможно, но необязательно. Лишь тщательный анализ связей и сравнение дампов, сделанных с некоторым интервалом, может дать точный ответ.

Строки обрабатываются в .NET особым образом, поскольку по умолчанию они неизменяемы. В отличие от неуправляемых языков типа C или C++, мы не можем изменить значение строки, после того как она создана. Поэтому код в листинге 4.16 приводит к ошибке компиляции «`Property or indexer 'string.this[int]' cannot be assigned to -- it is read only`» («Свойству или индексатору 'string.this[int]' нельзя присвоить значение – разрешено только чтение»).

Листинг 4.16 ❖ Пример неизменяемости строк

```
string s = "Hello world!";
s[6] = 'W';
```

> Имейте в виду, что фраза «строки неизменяемы, поэтому мы не можем изменить значение строки, после того как она создана» не совсем корректна. Просто базовая библиотека классов BCL не раскрывает API, который позволил бы изменить значение строки (даже с помощью отражения). Однако на уровне среды выполнения неизменяемость ничем не обеспечена. Содержимое строки – это просто непрерывный блок байтов, интерпретируемых в соответствии с заданной кодировкой. Ничто не помешает нам получить указатель на некоторые из этих байтов в небезопасном режиме и изменить их на месте. Однако такое поведение абсолютно не поддерживается, так что анализ возможных проблем остается на вашей совести.

Неизменяемость строк приводит к большой неразберихе при первом знакомстве с языком C#. Часто она иллюстрируется такими примерами, как в листинге 4.17. Метод `Greet` создает новую строку путем конкатенации строковых литералов и параметров метода. Начинающий программист на C# думает, что каждое применение оператора `+=` модифицирует переменную `result` (как это было бы в случае инкремента целой переменной).

Листинг 4.17 ❖ Конкатенация строк и пример создания скрытой временной строки

```
public string Greet(string firstName, string secondName)
{
   string result = "Hello ";
   result += firstName;
   result += " ";
   result += secondName;
   result += "!";
   return result;
}
```

Но рано или поздно он узнает, что это невозможно, потому что строки неизменяемы, а код в листинге 4.17 всякий раз создает временные строки (см. листинг 4.18). Таким образом, не сознавая того, мы создали четыре временные строки, каждая из них живет очень недолго – лишь до следующего вызова `Concat`. А как мы увидим в следующих главах, предотвращение выделения памяти – один из лучших способов улучшить код.

Листинг 4.18 ❖ CIL-код метода из листинга 4.17. Как видим, каждому оператору += соответствует вызов метода `String::Concat`, который конкатенирует две строки на вершине стека вычислений и помещает результат обратно в стек вычислений

```
.method public hidebysig instance string Write (string firstName, string secondName)
cil managed
{
   IL_0000: ldstr "Hello "
   IL_0005: ldarg.1
   IL_0006: call string [mscorlib]System.String::Concat(string, string)
   IL_000b: ldstr " "
   IL_0010: call string [mscorlib]System.String::Concat(string, string)
   IL_0015: ldarg.2
   IL_0016: call string [mscorlib]System.String::Concat(string, string)
   IL_001b: ldstr "!"
   IL_0020: call string [mscorlib]System.String::Concat(string, string)
   IL_0025: ret
}
```

Как улучшить такой код? Стандартное решение – воспользоваться типом `StringBuilder`, который имитирует поведение изменяемой строки (листинг 4.19). Внутри `StringBuilder` хранит текст в виде связанного списка блоков символов (они называются порциями, см. рис. 4.20). Можно считать, что `StringBuilder` – точка доступа к цепочке внутренних буферов. Количество и размеры буферов динамически изменяются по мере роста текста. Если в какой-то момент нам понадобится обычная строка, то можно вызвать метод `ToString`, который выделяет память для новой строки и копирует в нее данные порция за порцией.

Листинг 4.19 ❖ Создание строки с помощью типа `StringBuilder`, имитирующего «изменяемую строку», вместо конкатенации, как в листинге 4.17

```
public string Greet(string firstName, string secondName)
{
   StringBuilder sb = new StringBuilder();
   sb.Append("Hello ");
   sb.Append(firstName);
   sb.Append(" ");
   sb.Append(secondName);
   sb.Append("!");
   return sb.ToString();
}
```

Всякий раз, как требуются сложные манипуляции со строками, например при агрегировании данных в коллекциях, следует подумать об использовании StringBuilder.

Имейте в виду, что в таких простых случаях, как форматирование сообщения с несколькими параметрами, эффективнее воспользоваться методом `string.Format` или основанным на нем механизмом интерполяции строки: `public string Greet(string firstName, string secondName) => $"Hello {firstName} {secondName}!";`.

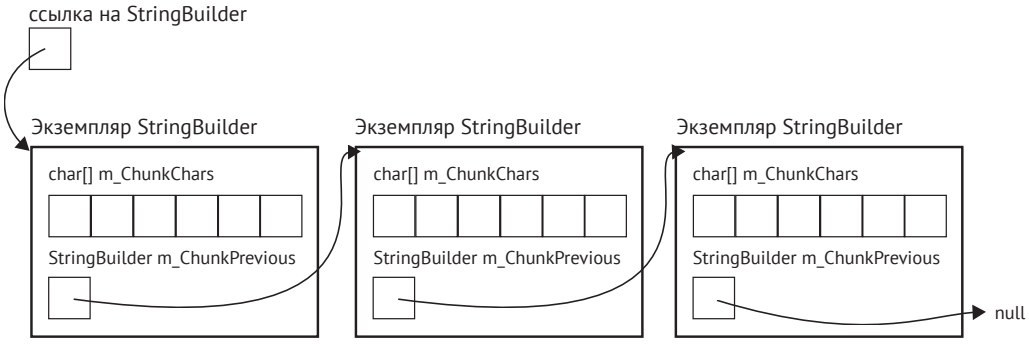


Рис. 4.20 ❖ Внутренняя структура данных StringBuilder

В таких популярных методах, как `string.Format` или `string.Join`, на самом деле используется `StringBuilder`. Они даже идут дальше и пытаются оптимизировать работу путем использования кешированных экземпляров `StringBuilder`, обернутых классом `StringBuilderCache` (листинг 4.20).

Листинг 4.20 ❖ Пример использования `StringBuilder` в методе `FormatHelper`, который вызывается из различных перегруженных вариантов `string.Format`

```
private static String FormatHelper(IFormatProvider provider, String format,
ParamsArray args) {
   ...
   return StringBuilderCache.GetStringAndRelease(
      StringBuilderCache
         .Acquire(format.Length + args.Length * 8)
         .AppendFormatHelper(provider, format, args));
}
```

Внутри себя `StringBuilderCache` хранит экземпляр `static StringBuilder` с атрибутом `ThreadStatic` (листинг 4.21). Это означает, что его можно использовать повторно, не опасаясь проблем с многопоточностью, т. к. значение в каждом потоке уникально (статическая память потока подробно рассматривается в главе 13).

Листинг 4.21 ❖ Начало класса `StringBuilderCache`, из которого понятна его внутренняя структура

```
internal static class StringBuilderCache
{
   // Значение 360 выбрано по итогам обсуждения со специалистами по
   // производительности, как компромисс между минимизацией объема
   // используемой памяти в каждом потоке и покрытием значительной
   // части операций создания короткоживущих экземпляров StringBuilder.
   private const int MAX_BUILDER_SIZE = 360;
```

```
   [ThreadStatic]
   private static StringBuilder CachedInstance;
   ...
}
```

Поскольку кешированных экземпляров `StringBuilder` существует столько же, сколько потоков в приложении, их емкость была выбрана с учетом компромисса между полезностью и издержками памяти. Это показывает нам, что подобные издержки обязательно следует принимать в расчет, когда проектируется такой востребованный API, как форматирование строк.

Различие в производительности при использовании изменяемых `StringBuilder` вместо конкатенации неизменяемых строк может быть очень существенным. В табл. 4.2 показаны результаты тестирования производительности трех методов в листинге 4.22. Здесь сравниваются два вышеупомянутых подхода и версия, использующая класс `StringBuilderCache`, который, хотя и не является открытым, легко может быть скопирован из исходного кода .NET Framework (https://referencesource.microsoft.com/#mscorlib/system/text/stringbuildercache.cs).

Листинг 4.22 ❖ Три подхода к построению сложных строк. В первом используется классическая конкатенация строк, в результате чего создается много короткоживущих строк. Во втором используется `StringBuilder`, а в третьем – кеширование экземпляров `StringBuilder` (кешированный экземпляр достаточно велик для хранения окончательного текста)

```
[Benchmark]
public static string StringConcatenation()
{
   string result = string.Empty;
   foreach (var num in Enumerable.Range(0, 64))
      result += string.Format("{0:D4}", num);
   return result;
}

[Benchmark]
public static string StringBuilder()
{
   StringBuilder sb = new StringBuilder();
   foreach (var num in Enumerable.Range(0, 64))
      sb.AppendFormat("{0:D4}", num);
   return sb.ToString();
}

[Benchmark]
public static string StringBuilderCached()
{
   StringBuilder sb = StringBuilderCache.Acquire(2 * 4 * 64);
   foreach (var num in Enumerable.Range(0, 64))
      sb.AppendFormat("{0:D4}", num);
   return StringBuilderCache.GetStringAndRelease(sb);
}
```

Таблица 4.2. Результаты измерения производительности трех методов построения строки из листинга 4.22. Использовалась программа BenchmarkDotNet на платформе .NET Core 2.1.0

Метод	Среднее время	Поколение 0	Выделено
`StringConcatenation`	12,420 мкс	6,3477	26,75 КБ
`StringBuilder`	7,708 мкс	1,7090	7,64 КБ
`StringBuilderCached`	7,630 мкс	1,4648	6,57 КБ

Как легко видеть из табл. 4.2, если не знать о подводных камнях конкатенации, то потребление памяти оказывается в четыре раза больше. И накладные расходы на сборку мусора при этом возрастают в четыре раза. В нашем тесте это не страшно, но в крупном веб-приложении, обрабатывающем тысячи запросов, разница может быть существенной.

Возникает два вопроса по поводу решений, принятых при проектировании строк.

- Почему строки неизменяемые? Если неизменяемость привносит противоречащее интуиции поведение и скрытые проблемы при выделении памяти, то зачем она вообще нужна? Ответ прост – неизменяемость столь популярного типа дает массу преимуществ ценой всего нескольких неудобств. Перечислим эти преимущества:
  - безопасность – строки широко используются в качестве элементов других структур данных. Возможность изменять их «на месте» могла бы привести к многочисленным ошибкам. Представьте себе ключи в структуре, подобной словарю. Если бы было разрешено изменять значение такого ключа, то могло бы повредиться внутреннее представление структуры (которое часто основано на различных видах самобалансирующихся деревьев). Строки также передаются различным API для задания учетных данных, имен файлов и путей к ним и т. д. Возможность изменить содержимое уже проверенной строки могла бы представлять большую опасность;
  - параллелизм – поскольку данные не изменяются, их совместное использование в нескольких потоках безопасно. Не нужно блокировок, и можно не бояться ложного разделения.
- А главный недостаток такой:
  - операции модификации приводят к появлению новых экземпляров строки (как метод `Concat` выше). Это особенно неприятно для больших текстовых данных. Представьте, что к тексту длиной несколько мегабайтов, хранящемуся в строке, применен всего один метод `Replace('a', 'b')`. В результате будет создана еще одна строка такой же длины, отличающаяся от исходной лишь несколькими символами.
- Поэтому решение о неизменяемости строк следует признать исключительно удачным. Если все-таки нужна какая-то операция, изменяющая строку, то воспользуйтесь классом `StringBuilder`. Это заставляет разработчика думать о наиболее подходящем подходе.
- Если строка неизменяема, то почему она не определена в виде структуры? Типы значений – идеальные кандидаты на роль неизменяемых сущностей, поскольку данные хранятся «здесь же», а семантика передачи по значе-

нию естественно сочетается с неизменяемостью. Так почему же строка – не структура? Но давайте немного подумаем. Хотя тип значений прекрасно выглядит в роли неизменяемого типа, обратное может быть неверно. Передача длинных строк по значению сопровождается большими накладными расходами, их гораздо эффективнее передавать по ссылке.

С другой стороны, если неизменяемость – это так хорошо, то почему бы не сделать все типы неизменяемыми по умолчанию?! На самом деле в большинстве функциональных языков так и поступают. И F# – не исключение. В языке F# изменяемость типа надо оговаривать специально при его объявлении (с помощью ключевого слова `mutable`).

Интернирование строк

В среде выполнения .NET существует механизм *интернирования строк*, который иногда вызывает больше путаницы, чем заслуживает. Это еще один вопрос, который обожают задавать на собеседовании. Интернирование строк – техника оптимизации, применяемая с целью эффективного использования памяти для хранения повторяющихся текстов. Один и тот же текст не копируется, а хранится в памяти в единственном экземпляре. Но проблема в том, что по умолчанию этот механизм применяется только к строковым литералам, но не к строкам, динамически создаваемым в процессе выполнения программы. Стандарт ECMA 335 гласит: «*по умолчанию CLI гарантирует, что две команды* `ldstr`*, ссылающиеся на маркеры метаданных с одинаковыми последовательностями символов, возвращают в точности один и тот же объект (этот процесс называется интернированием строк)*». И в листинге 4.18 мы уже видели, как команда `ldstr` применяется для загрузки строкового литерала».

Интернирование строк часто иллюстрируют примерами типа того, что показан в листинге 4.23. Здесь мы видим два строковых литерала `"Hello world!"` в разных контекстах, но с одинаковым значением. В строке 4 метода `Main` печатается `True`, поскольку среда выполнения интернировала литерал `"Hello world!"`, так что `s1` и `Global` ссылаются на один и тот же экземпляр строки.

Поскольку интернирование строк по умолчанию применяется только для литералов, этот механизм не особенно интересен разработчикам. Это скорее деталь реализации среды выполнения, имеющая целью очевидную оптимизацию – устранить дублирование «зашитого» текста. Еще раз подчеркнем: интернированию подвергаются только строковые литералы. Это ограничение также иллюстрируется в листинге 4.23. Хотя строка `s3` тоже имеет значение `"Hello world!"`, напечатанное сообщение показывает, что этот экземпляр отличается от интернированного. Стало быть, созданная динамически строка `s3` не интернируется (хотя каждый из двух литералов `"Hello "` и `"world!"` интернируется).

Листинг 4.23 ❖ Пример интернирования строк с комментариями

```
private static string Global = "Hello world!";
static void Main(string[] args)
{
   string s1 = "Hello world!";
   string s2 = "Hello ";
   string s3 = s2 + "world!";
```

```
Console.WriteLine(string.ReferenceEquals(s1, Global)); // True
Console.WriteLine(string.ReferenceEquals(s1, s3));     // False
...
```

Почему динамически созданные строки не интернируются по умолчанию? Потому что это сопряжено с большими накладными расходами. При создании новой строки среда выполнения должна была бы проверить, не интернирована ли уже эта строка. Но такая проверка обошлась бы дорого, если количество строк в программе велико. Стоимость проверки перевесила бы выигрыш от интернирования.

Однако имеется возможность явно управлять интернированием строк. Статический метод `string.IsInterned` возвращает `null`, если не существует интернированной строки с таким значением, а в противном случае – ссылку на интернированную строку. В листинге 4.24 показано продолжение метода `Main` из листинга 4.23. Если в строке 1 мы вызовем метод `string.IsInterned`, чтобы проверить, интернирована ли строка с таким значением, как у переменной `s3` (т. е. `"Hello world!"`), то получим интернированную ссылку – потому что интернированный строковый литерал `"Hello world!"` действительно существует. Это позволяет использовать интернированную версию строки, если таковая существует, а исходный экземпляр `s3` в конечном итоге будет убран в мусор, поскольку больше нигде не используется.

Мы даже можем явно интернировать строку методом `string.Intern` (см. строку 8 в листинге 4.24), который возвращает ссылку на интернированную строку. Если такое значение раньше не было интернировано, то это произойдет сейчас, и `string.Intern` вернет ссылку на интернированную строку. Иными словами, интернирование динамически созданной строки – это всего лишь запоминание ее в некоторой внутренней структуре данных. В нашем примере вызов `string.Intern` интернирует ссылку `message`, поэтому ссылки `s6` и `message` равны.

Листинг 4.24 ❖ Пример ручного интернирования строк

```
string s4 = string.IsInterned(s3);
Console.WriteLine(s4);   // Hello world!
Console.WriteLine(string.ReferenceEquals(s4, Global)); // True
string message = args[0];
string s5 = string.IsInterned(message);
Console.WriteLine(s5);   // null
string s6 = string.Intern(message);
Console.WriteLine(string.ReferenceEquals(s6, message)); // True
```

Это подводит нас к следующему вопросу. Существует разнобой мнений по поводу того, где находятся интернированные строки. Если динамически созданная строка `message` была интернирована, то где она хранится? Часто встречается утверждение, что интернированные строки хранятся в так называемом пуле интернированных строк, который располагается в куче больших объектов (Large Object Heap – LOH; мы будем говорить о ней в главе 5) – части управляемой кучи. Но проблема в том, что, как мы скоро узнаем, LOH предназначена для объектов, размер которых превышает 85 000 байт, а наша строка, очевидно, меньше. Означает ли это, что в процессе интернирования она перемещается в какой-то буфер большего размера? Иногда можно услышать, что интернированные строки хранятся в исполняемом файле, но этого никак не может быть для динамически созданных строк, правда? Истина несколько сложнее.

В памяти есть несколько мест, относящихся к интернированию строк (см. рис. 4.21). Главная из них – *карта строковых литералов* (String Literal Map), находящаяся в самом .NET Framework (в частной неуправляемой куче). Это хеш-таблица строк, распределенных по кластерам. Для каждой интернированной строки там есть отдельная запись, которая содержит вычисленный хеш и адрес записи в другой структуре – `LargeHeapHandleTable`. Эта таблица описателей, которая фактически находится в куче больших объектов, содержит ссылки на экземпляры строк. Но это экземпляры «нормальных» строк, находящихся в управляемой куче. Таким образом, мы не можем сказать, что интернированные строки обитают в каком-то специальном пуле интернированных строк. Они просто регистрируются в карте строковых литералов и в таблице описателей. Важно, что эти структуры живут столько же, сколько само .NET-приложение, поэтому ссылки на интернированные строки после регистрации уже не удаляются. В терминологии GC они всегда достижимы и потому никогда не убираются в мусор! Поскольку интернированные

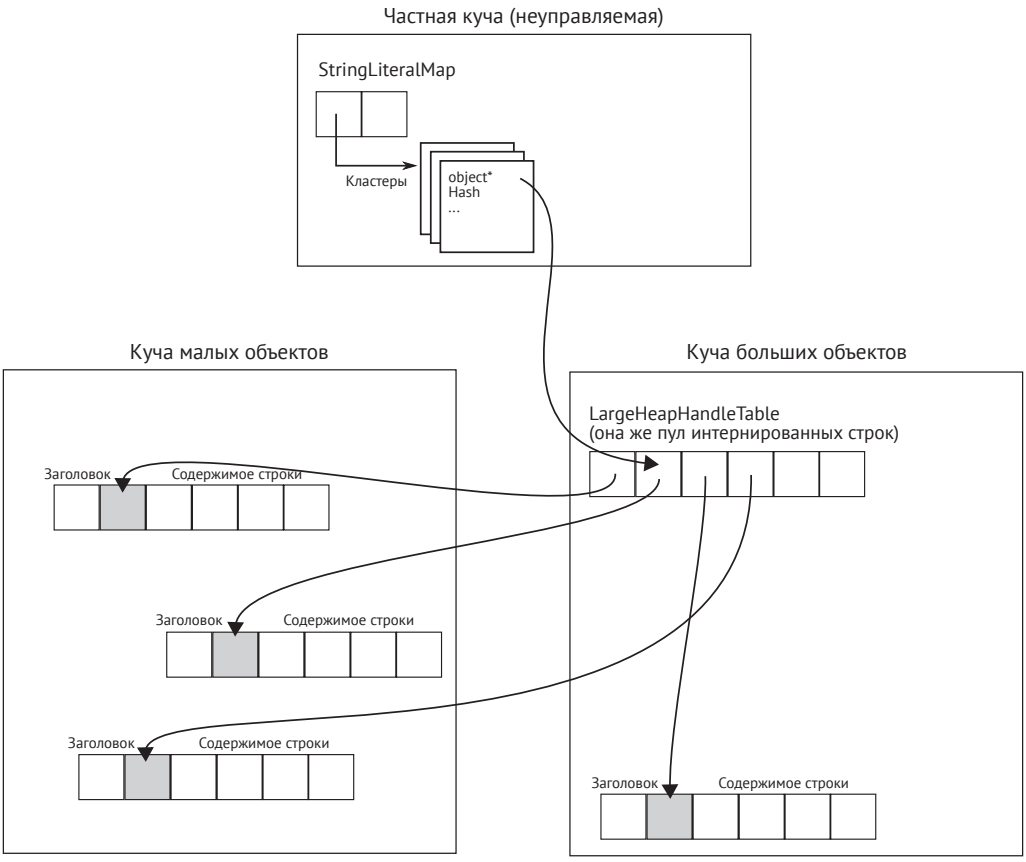


Рис. 4.21 ❖ Интернирование строк. Все интернированные строки на самом деле являются обычными экземплярами строки и хранятся в куче малых объектов или в куче больших объектов в зависимости от размера. Ссылки на них хранятся в таблице LargeHeapHandleTable, размещенной в куче больших объектов, а информация об этих описателях – во внутренних структурах данных среды выполнения .NET

строки находятся в управляемой куче, как любые другие объекты – в куче малых объектов (Small Object Heap – SOH), если занимают меньше 85 000 байт, и в куче LOH, если больше, – то рано или поздно они будут перемещены в поколение 2 и останутся там навечно.

Но что там со строковыми литералами? Интересно, что они ведут себя по существу так же. Рассмотрим простой код:

```
string s = "Hello world!";
```

При компиляции исходного кода все строковые литералы (в т. ч. и `"Hello world!"`) записываются в исполняемый файл, в так называемый поток хранения `#US` (расшифровывается как *user strings* – пользовательские строки). Показанная выше строка транслируется в уже известную нам команду CIL, аргумент которой ссылается на значение в потоке `#US` (`0x70000000`) с индексом 1 (`0x00000001`). В предположении, что наш текст «Hello world!» находится там, эта команда имеет вид:

```
ldstr 0x70000001
```

В процессе JIT-компиляции этой команды выполняются следующие действия:

- из потока #US читаются строковые данные с указанным индексом;
- проверяется наличие таких данных в карте строковых литералов. Если данные уже существуют, то будет возвращен адрес описателя. В противном случае:
  - выделяется память для новой строки. Это нормальная строка, поэтому она создается в поколении 0 (или в LOH, если размер достаточно велик);
  - в эту строку копируются данные из потока;
  - в таблице LargeHeapHandleTable создается новый описатель, указывающий на вновь созданную строку;
  - в карте строковых литералов создается новая запись.

Разработчику доступ к механизму интернирования строк дает метод `string.Intern`. Мы можем явно интернировать любую строку, в т. ч. динамически созданную. Это приводит к сумятице в умах. Когда интернирование строк может дать выигрыш? Рассмотрим некоторые за и против.

Достоинства интернирования строк:

- дедупликация строк – очевидное преимущество, которое, собственно, и стоит за идеей интернирования – устранить дубликаты строк и тем снизить накладные расходы при работе с памятью. Вполне разумно для строковых литералов, потому что отвечает за это среда выполнения на этапе JIT-компиляции. Но когда речь заходит о дедупликации динамически создаваемых строк, все становится не так очевидно. Следует проанализировать, сколько в нашем приложении дубликатов строк и какие издержки это несет. Быть может, они не настолько велики, чтобы терпеть изложенные ниже недостатки;
- производительность сравнения на равенство – для установления равенства строк требуется сравнивать их побайтно, что может оказаться очень медленно, особенно если строки велики. Но в операторах сравнения строк на равенство есть короткая ветвь, выбираемая в случае, когда сравниваются ссылки (см. листинг 4.25). Поэтому если в нашей программе приходится часто сравнивать строки, многие из которых дублируются, то такая оптимизация может дать выигрыш.

Недостатки интернирования строк:

- бессмертие – как уже было сказано, интернированные строки остаются достижимыми в течение всего времени работы приложения. Скорее всего, строки, которые мы собираемся интернировать, скоро станут недостижимыми и будут убраны в мусор. Но если мы их интернируем, то они обретут бессмертие, и нужно дважды подумать, стоит ли оно того. Вместо более эффективного использования памяти мы добьемся прямо противоположного эффекта. По существу, мы собираемся вечно хранить все строки, которые однажды встретились в приложении. Есть ли в этом смысл, зависит от степени их уникальности;
- создание временной строки – интернировать можно только уже созданную строку. Поэтому в течение короткого времени неинтернированная строка будет существовать, пусть даже только для того, чтобы проверить, имеется ли интернированная версия.

Листинг 4.25 ❖ Начало сравнения строк на равенство. Если обе строки представлены одной и той же ссылкой, то выбирается очень быстрая ветвь

```
public static bool Equals(String a, String b)
{
   if ((Object)a==(Object)b) {
      return true;
}
...
```

Читая данные из файла, веб-запроса и т. д., мы получаем экземпляры строк. Эти экземпляры не интернированы, и если они очень часто дублируются (как, например, имена тегов и атрибутов XML), то возникает искушение их интернировать. Но вот вопрос: каково время жизни таких строк? Если они просто читаются в процессе обработки входных данных, то скоро будут убраны в мусор. Если же мы их интернируем, то они останутся в памяти навечно, хотя библиотека, которой мы пользуемся, скорее всего, все равно будет создавать такие же временные строки[1]. А поскольку это нормальные строки, которые в конечном итоге переходят в поколение 2, создается дополнительная нагрузка на сборщик мусора.

Итак, мы приходим к окончательному заключению – интернирование строк может дать выигрыш в тех случаях, когда мы долгое время храним в памяти много повторяющихся строк. Это довольно необычная ситуация, потому что, как правило, приложение обрабатывает порцию текстовых данных, после чего забывает о них. Но если нам часто нужно сравнивать повторяющиеся строки на равенство, то есть повод задуматься об интернировании.

> Если вы контролируете процесс создания строк, то почему бы не реализовать собственный механизм дедупликации? Для этого нужно найти подходящее место, например там, где вы получаете поток байтов и хотите десериализовать его в строку. В таком случае процедуру дедупликации можно написать так, чтобы не создавать временных строк. Но все равно делать это стоит, только если количество повторяющихся строк в приложении действительно велико.

---

[1] Из-за неочевидных преимуществ интернирования строк даже системные библиотеки, в частности для работы с XML и HTTP, по умолчанию не пользуются интернированием.

Компромисс между плюсами и минусами иллюстрируется в следующем сценарии.

Сценарий 4.5. Моя программа потребляет слишком много памяти

Описание. Тестировщики обратили внимание, что после нескольких часов непрерывной работы процесс потребляет гигабайты памяти. Они сообщили об этом, и вы легко воспроизвели описанное поведение на своей машине с помощью средств автоматизации тестирования.

Анализ. Вы полностью контролируете среду, поэтому возможностей подступиться к проблеме много. Глядя на счетчики производительности или на результаты VMMap, легко убедиться, что размер управляемой кучи растет, и она занимает гигабайты памяти. В среде разработки мы можем присоединиться к процессу или проанализировать дамп памяти – инструментов хватает. Коммерческие программы проведут встроенные виды анализа и покажут, что огромное количество памяти расходуется на хранение повторяющихся строк (на рис. 4.22 в качестве примера приведен результат работы JetBrains dotMemory).

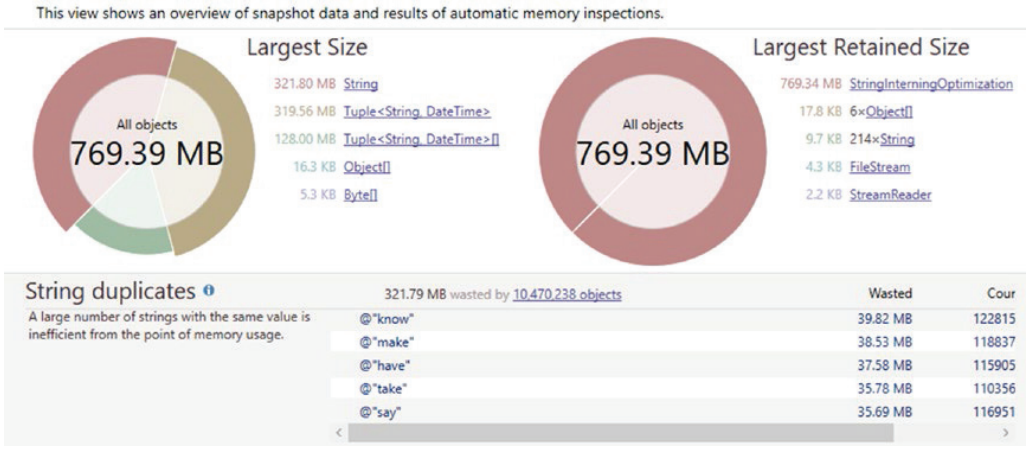


Рис. 4.22 ❖ Сценарий 4.5 – анализ дублирования строк с помощью программы JetBrains dotMemory

К такому же заключению можно прийти с помощью PerfView. В диалоговом окне Collect отметьте флажок .NET Alloc. Эта операция отслеживания обходится дорого, и вряд ли стоит включать ее в производственной среде. Но для проведения тестов на локальной машине с такими накладными расходами можно смириться. Отметим, что при включенном флажке .NET Alloc профилируемое приложение следует запускать после начала сбора данных. Когда сбор данных завершится, откройте анализ **GC Heap Net Mem** в группе **Memory Group**. Будет показан список типов, для которых чаще всего выделялась память. В нашем случае в начале списка должен быть тип `string`. Дважды щелкнув по нему, мы увидим агрегированный стек

выделений строки (рис. 4.23). Как видим, главный источник один – метод `System.IO.ReadLinesIterator.MoveNext()`.

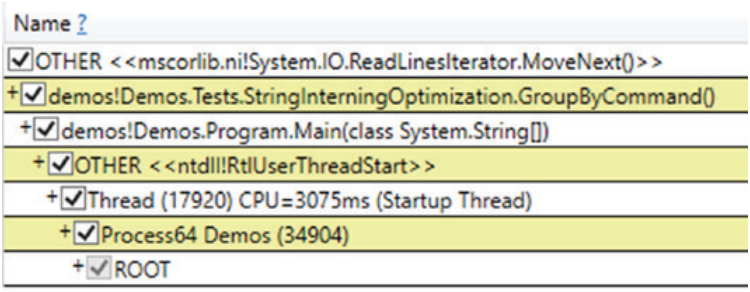


Рис. 4.23 ❖ Сценарий 4.5 – стек выделений строки в PerfView

> Если накладные расходы, связанные с флажком .NET Alloc, слишком велики, то можно провести выборочное тестирование в режиме .NET SampAlloc или даже оставить только флажок GC, часто этого хватает (если проблематичные операции выделения резко отличаются от всех прочих).

Взглянув на код, на который указывает анализ – `System.IO.ReadLinesIterator.MoveNext()` (листинг 4.26), – мы увидим, что это простой разбор файла, где мы подсчитываем, сколько раз встречается каждая строка, и сохраняем все строки в словаре вместе с временной меткой вхождения. Очевидно, что если повторяющихся строк много, то в памяти будет находиться много строк-дубликатов.

Листинг 4.26 ❖ Очень простой код для подсчета строк, иллюстрирующий возможность дублирования строк

```
foreach (var line in File.ReadLines(file))
{
   bool counted = false;
   foreach (var key in counter.Keys)
   {
      if (key == line)
      {
         counter[key]++;
       counted = true;
       break;
      }
   }
   if (!counted)
   {
      counter.Add(line, 0);
   }
   list.Add(new Tuple<string, DateTime>(line, DateTime.Now));
}
```

Изменим этот код, добавив интернирование строк. Строку можно интернировать сразу после чтения из файла (листинг 4.27). Новые строки будут выделяться для каждой строчки, прочитанной из файла, но время их жизни будет очень ко-

ротким. В словарь будут добавляться только интернированные строки. Они будут храниться на протяжении всего времени работы приложения, поэтому дедупликация даст выигрыш. Мы даже можем получить дополнительный прирост производительности, поскольку для сравнения строк на равенство теперь иногда будет использоваться сравнение ссылок.

Листинг 4.27 ❖ Код в листинге 4.26 изменен – используется явное интернирование строк

```
foreach (var line in File.ReadLines(file))
{
   var line2 = string.Intern(line); // время жизни line здесь и заканчивается
                                    // (кроме первого вхождения, когда строка
                                    // интернируется)
   bool counted = false;
   foreach (var key in counter.Keys)
   {
      if (key == line2)             // часто должно хватать сравнения ссылок, т. к.
                                    // сравниваются две интернированные строки)
      {
         counter[key]++;
         counted = true;
         break;
      }
   }
   if (!counted)
   {
      counter.Add(line2, 0);        // добавление интернированной строки
   }
   list.Add(new Tuple<string, DateTime>(line2, DateTime.Now));
}
```

Такой код приносит реальные плоды, только если в файле мало уникальных строк, но много повторяющихся, которые иначе пришлось бы долго хранить. Если это не так, то интернирование скорее приведет к снижению, а не повышению производительности.

Упаковка и распаковка

В .NET существует преобразование между типом значений и ссылочным типом. Процитируем стандарт ECMA-335:

> *Для каждого типа значений CTS определяет соответствующий ссылочный тип – упакованный тип. Обратное неверно: в общем случае ссылочному типу не соответствует никакой тип значений. Представлением значения упакованного типа (упакованным значением) является местоположение, где можно сохранить значение типа значений. Упакованный тип – это тип объекта, а упакованное значение – объект.*
>
> *(...)*

Для любого типа значений определена операция, называемая упаковкой. Упаковка значения типа значений порождает его упакованное значение, т. е. значение соответствующего упакованного типа, содержащее побитовую копию исходного значения.

Поскольку в определениях типа значений и ссылочного типа стек и куча вообще не упоминаются, таких упоминаний нет и в определении упаковки. Мы можем рассматривать упаковку как процесс преобразования экземпляра типа значений в экземпляр ссылочного типа, а стало быть, преобразования семантики этих значений.

Как я уже несколько раз говорил, касаясь деталей реализации, в некоторых случаях память для экземпляра типа значений (например, структуры) приходится выделять в куче. А с любым объектом в куче связаны дополнительные данные: заголовок и ссылка на таблицу методов. Следовательно, если требуется сохранить объект типа значений в куче, его значение необходимо обернуть, добавив эти данные. То есть операция упаковки состоит из двух шагов:

- выделение в куче памяти для упакованного типа, соответствующего типу значений (нового экземпляра упакованного типа);
- копирования данных из экземпляра типа значений во вновь созданный экземпляр ссылочного типа.

Интуиция подсказывает, что это не самая эффективная операция. Нужно выделить память для объекта и скопировать его данные – то и другое требует времени. Хуже того, экземпляр упакованного типа в какой-то момент нужно будет убрать в мусор, что оказывает дополнительную нагрузку на GC.

В листинге 4.28 приведен типичный пример упаковки. Мы видим, что значение типа значений `int` присваивается объекту ссылочного типа `object`. В таком случае необходима упаковка.

Листинг 4.28 ❖ Пример неявной упаковки

```
int i = 123;
object o = i; // неявная упаковка
```

В листинге 4.29 показан CIL-код упаковки для стековой машины. Команда `box` получает значение и помещает в стек вычислений результат упаковки (ссылку на вновь созданный экземпляр ссылочного типа).

Листинг 4.29 ❖ CIL-код, сгенерированный для кода на C# из листинга 4.28

```
IL_0000: ldc.i4.s 123
IL_0002: box System.Int32
IL_0007: ret
```

Это транслируется в двухшаговую операцию, которая была упомянута выше (листинг 4.30). Сначала выделяется память для упакованного типа `System.Int32`, а затем в нее копируется значение (в данном случае одно целое число, равное `123`, или `0x7b` в шестнадцатеричной записи).

Листинг 4.30 ❖ Ассемблерный код, сгенерированный для CIL-кода из листинга 4.29 (в режиме выпуска для платформы x64)

```
Samples.Echoer.Write(System.String)
0x00007FFB`7BE56180:   L0000: sub rsp, 0x28
0x00007FFB`7BE56184:   L0004: mov rcx, 0x7ffbd85e9288 ; (MT: System.Int32)
```

```
0x00007FFB`7BE5618E:   L000e: call clr!JIT_TrialAllocSFastMP_InlineGetThread
0x00007FFB`7BE56193:   L0013: mov dword [rax+0x8], 0x7b
0x00007FFB`7BE5619A:   L001a: add rsp, 0x28
0x00007FFB`7BE5619E:   L001e: ret
```

Одно из главных правил .NET, относящихся к памяти, – избегать упаковки. Код, где упаковка производится очень часто, действительно создает проблемы – производительность снижается. К сожалению, чаще всего упаковка производится неявно, так что мы о ней даже не подозреваем. Поэтому так важно знать, в каких местах это происходит чаще всего.

- Тип значений используется там, где ожидается объект (ссылочный тип), – значит, он должен быть упакован. Помимо несколько искусственного примера в листинге 4.28, эта ситуация часто встречается в аргументах методов, принимающих тип `object`, например `string.Format`, `string.Concat` и т. п.:

  ```
  int i = 123;
  return string.Format("{0}", i);
  ```

  Глядя на сгенерированный CIL-код, мы видим, что имеет место упаковка в тип `System.Int32`:

  ```
  IL_0003: ldstr "{0}"
  IL_0000: ldc.i4.s 123
  IL_0009: box [mscorlib]System.Int32
  IL_000e: call string [mscorlib]System.String::Format(string, object)
  ```

  К сожалению, здесь мы ничего не можем сделать, чтобы избежать упаковки. Даже более современный синтаксис интерполяции в строку (`return $"{i}"` в нашем случае) приводит к упаковке, поскольку основан на методе `string.Format`. Мы можем вызвать метод `ToString` типа значений (`string.Format("{0}", i.ToString())`), чтобы избежать упаковки, но тогда будет выделена новая строка, так что с точки зрения нагрузки на память эффект точно такой же. Общее правило – по возможности избегать методов, принимающих параметры типа `object`. До появления обобщенных типов в .NET Framework 2.0 данные во всех коллекциях хранились в виде ссылок на объекты, поскольку коллекция должна была обладать достаточной гибкостью для хранения любых данных. Поэтому существовало много методов вида `ArrayList.Add(Object value)` и т. п., а значит, упаковка производилась очень часто. Благодаря обобщенным типам этой проблемы больше не существует, поскольку обобщенный тип или метод компилируется для конкретного типа значений (например, `List<T>` превращается в `List<int>`), так что упаковка не нужна.
- Экземпляр типа значений используется в качестве типа интерфейса, реализуемого этим типом значений. Поскольку интерфейс – ссылочный тип, здесь тоже необходима упаковка. Предположим, что тип `SomeStruct` реализует интерфейс `ISomeInterface`, содержащий метод `GetMessage`:

  ```
  public string Main(string args)
  {
    SomeStruct some;
    var message = Helper(some);
    return message;
  }
  ```

```
string Helper(ISomeInterface data)
{
  return data.GetMessage();
}
```

И здесь в сгенерированном CIL-коде присутствует неявная упаковка:

```
IL_0000: ldarg.0
IL_0001: ldloc.0
IL_0002: box Samples.SomeStruct
IL_0007: call instance string Samples.Program::Helper(class Samples.ISomeInterface)
```

В таких случаях упаковки можно избежать, включив обобщенный метод, который ожидает получить желаемый интерфейс в виде параметра обобщенного типа:

```
string Helper<T>(T data) where T : ISomeInterface
{
  return data.GetMessage();
}
```

Обобщенный метод будет откомпилирован для этого конкретного типа значений, поэтому упаковка не понадобится:

```
IL_0000: ldarg.0
IL_0001: ldloc.0
IL_0002: call instance string Samples.Program::Helper<valuetype
Samples.SomeStruct>(!!0)
```

Рассмотрим один из самых распространенных источников упаковки, проистекающий из использования типа значений в качестве интерфейса, – цикл `foreach` для типа `IEnumerable<T>` (листинг 4.31). Пусть мы передаем экземпляр `List<int>` в качестве `IEnumerable<int>` методу `Print`. Цикл `foreach` в действительности основан на понятии перечислителя – он вызывает метод `GetEnumerator()` переданной коллекции, а затем в цикле вызывает его методы `Current()` и `MoveNext()`. Метод `Print` видит список как объект типа `IEnumerable<int>`, поэтому вызывает метод `IEnumerable<int>.GetEnumerator()` и ожидает, что тот вернет `IEnumerator<int>`. Класс `List<T>`, очевидно, реализует интерфейс `IEnumerable<int>`, но важно то, что `GetEnumerator()` возвращает `Enumerator`, который является … структурой. Поскольку эта структура используется как `IEnumerator<int>`, в начале цикла `foreach` производится упаковка.

Листинг 4.31 ❖ Скрытое выделение памяти из-за упаковки при использовании foreach

```
public int Main(string args)
{
   List<int> list = new List<int>() {1, 2, 3};
   Print(list);
   return list.Count;
}

public void Print(IEnumerable<int> list)
{
   foreach (var x in list)
   {
      Console.WriteLine(x);
   }
}
```

Очевидно, больших накладных расходов тут нет, поскольку единственная упаковка `Enumerator`, скорее всего, потеряется на фоне операций внутри цикла `foreach`. Как всегда в таких случаях, беда настигнет нас, если на основном пути выполнения встречается много таких циклов `foreach`. И, как всегда, «измеряйте как можно раньше», подвержено ваше приложение этой проблеме или нет, для чего выясните, сколько раз создается `Enumerator`. Если вы хотите избежать упаковки, то можете передать методу `Print` просто `List<int>` (объявив его как `public void Print(List<int> list)`). В таком случае, когда цикл `foreach` вызывает `List<int>.GetEnumerator()`, он ожидает получить структуру `List<int>.Enumerator`, и для нее будет создана локальная переменная именно такого типа. Нужды в упаковке не возникает. Это то место, где рекомендуемая практика программирования может вступать в конфликт с желанием оптимизировать код. Вообще говоря, лучше, когда метод `Print` принимает любой тип `IEnumerable<T>` и не завязан на конкретную реализацию `List<T>`. Но, с другой стороны, это влечет за собой упаковку, так что нужно выбирать между правильным проектированием и высокой производительностью.

Напрашивается вопрос: почему в таких часто используемых коллекциях, как `List<T>`, перечислители реализованы в виде структуры, если это сопряжено со скрытой упаковкой? Ответ прост, и после всего сказанного вы, наверное, уже догадались. В подавляющем большинстве случаев перечислители используются как локальные переменные, а поскольку они являются типами значений, то память для них можно быстро и дешево выделить в стеке. Это намного перевешивает возможные проблемы с упаковкой.

Для упаковки определена противоположная операция *распаковки*, т. е. преобразования упакованного значения ссылочного типа обратно в экземпляр типа значений. Эта операция привлекает гораздо меньше внимания, потому что не сопровождается столь существенными накладными расходами. Во-первых, распаковка имеет место, только если до этого была произведена упаковка. Во-вторых, с распаковкой не связано никакое выделение памяти в куче. Значение копируется из кучи обратно в стек, так что издержки, связанные с копированием в памяти, все же присутствуют. Но, как мы уже знаем, выделение памяти в стеке гораздо слабее влияет на производительность, так что опасаться распаковки нет причин.

> С распаковкой связана одна не вполне очевидная тонкость. Стандарт ECMA-335 гласит: «*Для любого упакованного типа имеется операция распаковки, которая дает управляемый указатель на битовое представление значения*». И существует команда CIL `unbox`, которая именно это и делает – помещает в стек вычислений управляемый указатель на данные в упакованном экземпляре. Таким образом, можно сказать, что распаковка в чистом виде не является ни копированием, ни выделением памяти для данных. Но возвращенный указатель следует использовать для получения фактического значения. И делает это команда `ldobj`, которая «*копирует значение, хранящееся по адресу src, в стек*». Когда компилятор C# хочет выполнить распаковку, он генерирует команду CIL `unbox.any`, эквивалентную последовательности `unbox` и `ldobj`.

Есть много мест, в которых возможна неявная упаковка, и очень трудно все время помнить о них. Что можно сделать, чтобы справиться с этой проблемой? Безусловно, следует изучить самые простые и часто встречающиеся случаи. Но, кроме того, есть и инструменты, готовые оказать помощь. Существует расширение Heap Allocations Viewer для Visual Studio и подключаемый модуль Roslyn C# Heap Allocation Analyzer для ReSharper, которые именно это и делают. Они показывают все

скрытые выделения памяти, включая и проистекающие из неявной упаковки. Настоятельно рекомендую применять эти средства в повседневной работе. Дополнительные примеры источников скрытого выделения памяти (в т. ч. и упаковки) приведены в главе 6 вместе со сценариями их исследования.

Передача по ссылке

Мы уже вкратце рассмотрели типы значений и ссылочные типы, а также связанную с ними семантику передачи по значению и по ссылке. Но есть еще один уровень контроля. Мы несколько раз упоминали, что по ссылке можно передать любое значение, будь то экземпляр типа значений или ссылочного типа.

Теперь рассмотрим оба случая.

Передача по ссылке экземпляра типа значений

Как уже неоднократно отмечалось, типы значений обладают семантикой передачи по значению, т. е. присваивание экземпляра типа значений приводит к созданию его побитовой копии. Часто это иллюстрируют примером, похожим на приведенный в листинге 4.32. Здесь используется определение структуры из листинга 4.6. Метод `Helper` принимает один аргумент типа значений. Когда мы передаем ему экземпляр `SomeStruct`, внутри `Helper` создается его локальная копия. Поэтому изменять `data.Value1` не имеет смысла – будет модифицирована лишь локальная копия, а исходный экземпляр `ss` не изменится. Метод `Main` вернет значение 10.

Листинг 4.32 ❖ Пример передачи структуры по значению

```
public int Main(int data)
{
   SomeStruct ss = new SomeStruct();
   ss.Value1 = 10;
   Helper(ss);
   return ss.Value1;
}

private void Helper(SomeStruct data)
{
   data.Value1 = 11;
}
```

Мы можем изменить это поведение, передавая экземпляр данных по ссылке с помощью ключевого слова `ref` (листинг 4.33). В таком случае будет использоваться ссылка на исходное значение в стеке. Любая его модификация в методе `Helper` отразится на оригинальном экземпляре `ss`. Поэтому метод `Main` вернет `11`.

Листинг 4.33 ❖ Пример передачи структуры по ссылке

```
public int Main(int data)
{
   SomeStruct ss = new SomeStruct();
   ss.Value1 = 10;
```

```
   Helper(ref ss);
   return ss.Value1;
}

private void Helper(ref SomeStruct data)
{
 data.Value1 = 11;
}
```

Использование структуры (типа значений) в качестве локальной переменной и передача ее по ссылке – хороший способ оптимизации. И не только потому, что мы избегаем выделения памяти в куче, но и потому, что накладные расходы на копирование данных исключаются вне зависимости от размера структуры.

> Не забывайте, что JIT-компилятор – великий мастер оптимизации. Строя версию программы из листинга 4.33 для режима выпуска, JIT-компилятор заметит, что наличие структуры даже в стеке необязательно (мы видели это в листинге 4.9). Поэтому в нашем примере ассемблерный код метода `Main` сводится к двум командам: `mov eax, 0xb` и `ret`!

Передача по ссылке экземпляра ссылочного типа

Тут возможно легкое замешательство – ведь мы говорим о передаче по ссылке ссылки на экземпляр ссылочного типа. Если вы знакомы с C/C++, то это что-то вроде указателя на указатель.

В листинге 4.34 это иллюстрируется на примере определения в листинге 4.11. Здесь по ссылке передается ссылка на экземпляр ссылочного типа `SomeClass`. Мы можем обратиться к нему, как обычно, из класса `Helper` (правда, это будет несколько медленнее простого доступа по ссылке из-за необходимости дополнительного разыменования указателя). Но, имея ссылку на ссылочный тип, мы можем модифицировать ее, так что эта ссылка будет указывать на какой-то другой экземпляр ссылочного типа. В нашем примере метод `Main` вернет `11`. Если бы `SomeClass` был передан просто по ссылке, то метод `Helper` перезаписал бы локальную ссылку, создав новый экземпляр. Но снаружи такие изменения были бы не видны. Возможно, вам стоит немного задержаться, чтобы уложить это в голове.

Листинг 4.34 ❖ Пример передачи ссылочного типа по ссылке

```
public int Main(int data)
{
   SomeClass sc = new SomeClass();
   sc.Value1 = 10;
   Helper(ref sc);
   return sc.Value1;
}

private void Helper(ref SomeClass data)
{
   data = new SomeClass();
   data.Value1 = 11;
}
```

Мы будем много говорить о передаче по ссылке в главе 14. Это большая и очень интересная тема. Это также один из самых действенных приемов оптимизации, применяемых для повышения производительности. Если перед вами стоит задача написать суперэффективную библиотеку, обладающую максимально возможной производительностью, то, безусловно, стоит обратить внимание на этот вид оптимизации. Именно так пишутся программы, от которых ожидают наивысшей производительности, например компилятор Roslyn или сервер Kestrel. А пока просто запомним, что это прекрасный способ повысить производительность структур и классов и избежать выделения памяти в куче.

> Передача по ссылке настолько важна для оптимизации различных библиотек, что привлекает все больше внимания со стороны создателей .NET и языка C#. Начиная с версии C# 7.0 появилась возможность объявлять локальные ссылочные переменные и возвращать значение по ссылке. В версиях C# 7.1 и 7.2 есть возможность передавать данные по постоянной ссылке (с помощью ключевого слова `in` вместо `ref`), чтобы выразить тот факт, что ссылка передается только для доступа к данным, но не позволяет их модифицировать. Все это мы рассмотрим в главе 14.

Локальность типов данных

Не будучи обременены дополнительными данными, структуры очень компактны. Это желательное свойство по двум причинам.

- Всегда лучше обрабатывать поменьше данных – этот факт очевиден и не нуждается в комментариях. Даже во времена дешевой памяти мы выигрываем во времени.
- Всегда хорошо использовать кеш максимально эффективно – благодаря меньшему размеру объектов мы можем загрузить их больше в одну строку кеша, а значит, получить заметный прирост производительности. В главе 2 мы видели, как окупается размещение данных в памяти таким образом, чтобы получить максимально много данных в одной строке кеша. В этом как раз и помогают структуры.

Применение структур обеспечивает более плотное размещение данных в памяти, поскольку отсутствуют накладные расходы, присущие ссылочным типам. И, что еще важнее, массив структур расположен в памяти непрерывно, тогда как в случае ссылочных типов непрерывную область занимают лишь ссылки на данные. А значения, на которые они указывают, могут быть разбросаны по всем управляемым кучам, и мы никак не можем это контролировать (рис. 4.24).

Разница в производительности из-за различий в локальности данных иллюстрируется с помощью программы в листинге 4.35. Она всего лишь вычисляет сумму первых полей всех элементов массива, один раз для массива структур и один раз для массива классов.

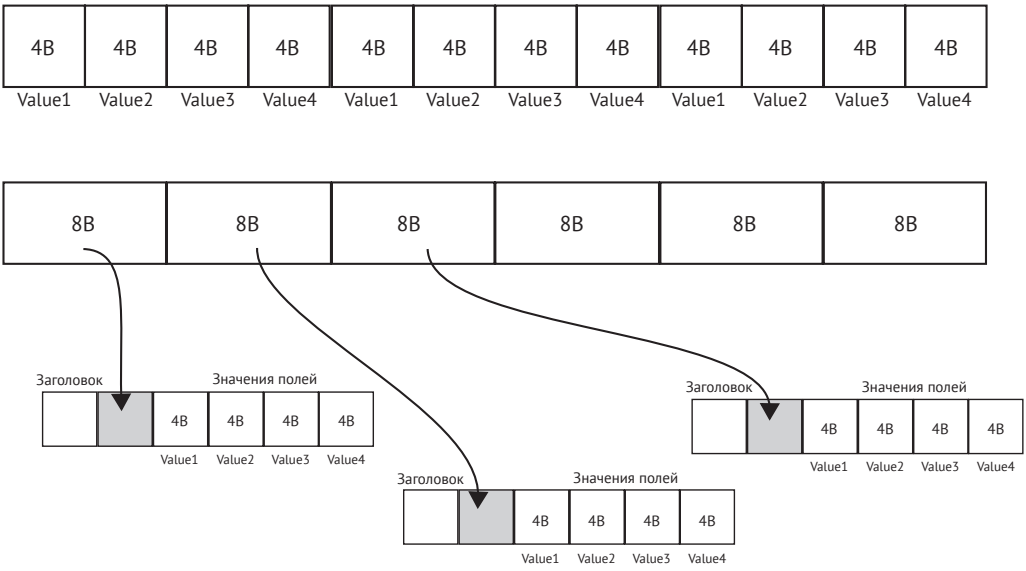


Рис. 4.24 ❖ Массив структур (вверху) занимает непрерывную область памяти, потому что данные типа значений хранятся в самом типе. Массив классов (внизу) представляет собой непрерывную область ссылок, указывающих на объекты, расположенные где-то в куче

Листинг 4.35 ❖ Измерение разницы в производительности доступа к массиву структур и массиву классов

```
public struct SmallStruct
{
   public int Value1;
   public int Value2;
}

public class SmallClass
{
   public int Value1;
   public int Value2;
}

// каждый массив содержит миллион элементов
private SmallClass[] classes;
private SmallStruct[] structs;
[Benchmark]
public int StructArrayAccess()
{
   int result = 0;
   for (int i = 0; i < items; i++)
      result += Helper1(structs, i);
   return result;
}
```

```
[Benchmark]
public int ClassArrayAccess()
{
   int result = 0;
   for (int i = 0; i < items; i++)
      result += Helper2(classes, i);
   return result;
}

public int Helper1(SmallStruct [] data, int index)
{
   return data[index].Value1;
}

public int Helper2(SmallClass [] data, int index)
{
   return data[index].Value1;
}
```

Интересно, что разница между этими двумя подходами проявляется только в коде методов `Helper`, сгенерированном JIT-компилятором (см. листинг 4.36). И состоит она в том, что при доступе к массиву структур в методе `Helper1` используется только одно разыменование адреса – адрес элемента в массиве вычисляется путем умножения его индекса на размер структуры. А затем значение по этому адресу копируется в регистр. В методе `Helper2` разыменование производится дважды – первый раз, чтобы получить ссылку на объект с заданным индексом, а второй раз, чтобы получить значение, на которое указывает эта ссылка.

Листинг 4.36 ❖ Фрагменты ассемблерного кода, сгенерированного JIT-компилятором для методов Helper в листинге 4.35. Регистр `rdx` содержит адрес массива объектов, а `rax` – индекс элемента в этом массиве

```
Helper1(Samples.SomeStruct[], Int32)
...
0x00007FFA`526A0E8D:   L000d: mov eax, [rdx+rax*8+0x10]
...
Helper2(Samples.SomeClass[], Int32)
...
0x00007FFA`526A0E4D:   L000d: mov rax, [rdx+rax*8+0x10]
0x00007FFA`526A0E52:   L0012: mov eax, [rax+0x8]
...
```

> **ПРИМЕЧАНИЕ** Код методов Helper не самом деле встроен в тестируемые методы, но здесь он представлен в исходной форме для большей ясности.

В табл. 4.3 показаны результаты обоих тестов. Столь большую разницу невозможно объяснить только лишним разыменованием адреса. Дополнительные накладные расходы связаны с гораздо худшей локальностью данных из-за того, что экземпляры класса не находятся по соседству в памяти. Поэтому в процессе вычислений приходится загружать больше строк кеша.

Таблица 4.3. Результаты тестирования производительности доступа к массивам структур и классов. Использовалась программа BenchmarkDotNet в версии .NET Core 2.0.0

Метод	Среднее время	Выделено
`StructArrayAccess`	618,7 мкс	0 Б
`ClassArrayAccess`	1816,6 мкс	0 Б

Статические данные

Статические данные можно рассматривать как разновидность глобальных переменных в программе. И хотя в рекомендациях по проектированию глобальные переменные не приветствуются, они все же бывают полезны. В языке C# существует только один вид статических данных – статические поля. VB.NET позволяет объявлять статические переменные в функциях, но это просто синтаксический сахар поверх обычного статического поля (при использовании в Shared-функции). Рассмотрим статические поля более подробно.

Статические поля

Любой программирующий для .NET прекрасно знаком со статическими полями – их значение общее во всех экземплярах данного типа. Для доступа к ним нужно указать имя типа, что можно сделать в любом месте, где этот тип доступен (см. листинг 4.37). Это вполне разумно и не нуждается в дальнейших объяснениях.

Листинг 4.37 ❖ Пример статического поля

```
public class C {
   public void Method1()
   {
      S.Value = 10;
   }
   public void Method2() {
     Console.WriteLine(S.Value);
   }
}

public class S
{
   public static int Value;
}
```

Однако с точки зрения использования памяти следует сделать еще несколько замечаний.

- Область видимости статических данных – домен приложения, т. е. если загрузить одну и ту же сборку в несколько доменов приложения, то будет существовать несколько экземпляров одних и тех же статических данных.

- Статические данные типов, определенных в сборке, живут столько же, сколько домен приложения, в который была загружена сборка, т. е. до тех пор, пока сборка не выгружена, все статические данные и объекты, на которые они ссылаются, остаются достижимыми (не убираются в мусор).
- Хотя это и детали реализации, следует помнить, что:
  - статические данные примитивных типов (например, числа) хранятся в высокочастотной куче соответствующего домена приложения (это часть его кучи загрузчика);
  - статические экземпляры ссылочных типов (объекты) находятся в регулярной куче GC – от обычных объектов они отличаются наличием дополнительных ссылок из внутренней «таблицы статических данных». Поскольку такие объекты, очевидно, живут долго, в конечном итоге они окажутся в поколении 2 и там и останутся[1];
  - статические экземпляры пользовательских типов значений (структуры) также находятся в регулярной куче GC в упакованной форме.

Если вас интересует, как в действительности реализованы статические данные в .NET, прочитайте следующий раздел.

Внутреннее устройство статических данных

Любой домен приложения в .NET представлен набором внутренних структур данных (рис. 4.25). Каждому модулю, принадлежащему одной из загруженных сборок, соответствует структура данных `DomainLocalModule`. Она содержит две области, особенно важные с точки зрения реализации статических данных:

- для полей ссылочных типов и структур (в упакованной форме) – ссылка, указывающая на начало части таблицы `Object[]`, в которой хранятся ссылки из данного модуля (`m_pGCstatics` на рис. 4.25). Эта таблица `Object[]` является общей для всех модулей и сборок в этом домене приложения;
- для полей примитивных типов – значения, сгруппированные по типам, в которых определены, включая заполнение, обусловленное требованиями к выравниванию памяти (*область статики* на рис. 4.25).

На вышеупомянутый разделяемый массив `Object[]` ведет ссылка из внутренней структуры данных `LargeHeapHandleTable` (уже упоминавшейся в разделе об интернировании строк, для которого она тоже используется), а выделен он в куче больших объектов (и закреплен, чтобы можно было безопасно хранить адреса, указывающие на его элементы). Такая таблица описателей хранит массивы кластерами: когда текущий массив заполнен, создается новый кластер и соответствующий ему массив (это может случиться, например, если нужно сконструировать новый обобщенный тип со статическими полями).

> Заметим, что все структуры данных, показанные на рис. 4.25, в конечном итоге будут удалены при удалении соответствующего домена приложения (включая все статические данные в загруженных сборках). В случае забираемых сборок, упомянутых выше в этой главе, удаляется только `DomainLocalModule`, и все соответствующие ему элементы исключаются из разделяемой таблицы описателей. В любом случае все статические экземпляры ссылочных типов (и объекты, на которые они ссылаются) станут при этом недоступными и в конце концов будут убраны в мусор.

[1] Если только это не большой объект, который начинает жизнь в куче больших объектов.

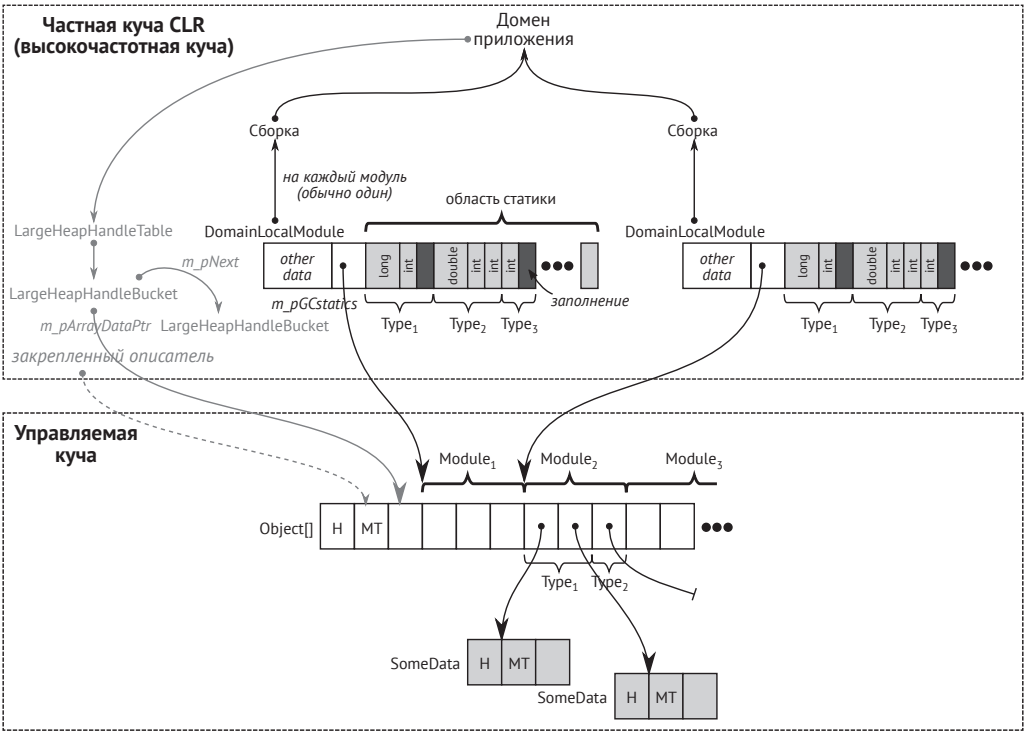


Рис. 4.25 ❖ Механизм хранения статических полей в .NET Core (при наличии одного домена приложения двух загруженных в него сборок). Места, где хранятся статические данные, закрашены серым (а все остальные видимые структуры можно считать вспомогательными данными). В .NET Framework область статики хранится рядом с таблицей методов данного типа

Дополнительно при построении статических данных вычисляются смещения всех статических полей и сохраняются в соответствующем поле таблицы методов. Когда JIT-компилятор порождает код, обращающийся к статическому полю, он следующим образом использует эти данные:

- для статического поля примитивного типа – зная адрес соответствующего `DomainLocalModule` и смещение интересующего поля в его области статики, компилятор вычисляет абсолютный адрес данных;
- для статического поля ссылочного типа (включая структуры, которые хранятся в куче в упакованной форме) – зная адрес (из `LargeHeapHandleTable` и ее кластеров) соответствующего массива `Object[]` и смещение интересующего поля от его начала, вычисляется абсолютный адрес соответствующего элемента массива (в котором находится ссылка, указывающая на нужный объект).

Взяв в качестве примера несколько простых типов, определенных в листинге 4.38, мы можем увидеть структуры, изображенные на рис. 4.25, в действии.

Листинг 4.38 ❖ Простые типы, используемые в следующих примерах кода

```
public class ExampleClass
{
   public static int StaticPrimitive;
```

```
   public static S StaticStruct;
   public static R StaticObject = new R();
}

public class R
{
   public int Value;
}

public struct S
{
   public int Value;
}
```

Ассемблерный код, генерируемый при доступе к статическому полю примитивного типа (листинг 4.39), очень простой (листинг 4.40) – он сводится всего лишь к чтению данного значения по адресу, находящемуся внутри области статики. Таким образом, доступ к примитивным статическим данным можно считать очень быстрой операций без всяких накладных расходов (по крайней мере, до тех пор, пока мы не пытаемся сделать ее потокобезопасной, навесив блокировки).

Листинг 4.39 ❖ Тривиальный пример доступа к примитивному статическому полю

```
[MethodImpl(MethodImplOptions.NoInlining)]
public void Method1()
{
   Console.WriteLine(ExampleClass.StaticPrimitive);
}
```

Листинг 4.40 ❖ Результат JIT-компиляции кода из листинга 4.39 (только относящаяся к делу часть)

```
...
mov ecx,dword ptr [00007ff9`3c8a4bd8] ;   адрес в высокочастотной куче
                                      ;   (внутри statics blob)
call 00007ff9`3c9c1380 (System.Console.WriteLine(Int32), mdToken:000000000600007e)
...
```

Структуры, являющиеся статическими полями, выделяются в куче в упакованной форме, поэтому их можно рассматривать как любой другой объект. Для доступа к данным в таком статическом поле (листинг 4.41) JIT-компилятор генерирует код, который обращается к таблице описателей за адресом выделенного в куче GC экземпляра структуры (листинг 4.42). Следует иметь в виду эти дополнительные накладные расходы на разыменование описателя, поскольку априори можно было бы подумать, что структуры хранятся в области статики как значения примитивных типов.

Листинг 4.41 ❖ Тривиальный пример доступа к статическому полю определенного пользователем типа значений

```
[MethodImpl(MethodImplOptions.NoInlining)]
public void Method2()
{
   Console.WriteLine(ExampleClass.StaticStruct.Value);
}
```

Листинг 4.42 ❖ Результат JIT-компиляции кода из листинга 4.41

```
...
mov  rcx,19510002938h      ; адрес в LOH (внутри таблицы описателей)
mov  rcx,qword ptr [rcx]   ; разыменовать описатель (rcx содержит адрес упакованной структуры)
mov  ecx,dword ptr [rcx+8] ; обратиться к первому полю упакованной структуры
call 00007ff9`3c9c2b60 (System.Console.WriteLine(Int32), mdToken:000000000600007e)
...
```

Доступ к данным в статическом поле ссылочного типа (листинг 4.43) порождает в точности такой же код, как выше: обращение к таблице описателей за адресом объекта (листинг 4.44). Здесь снова имеют место накладные расходы на разыменование, но в случае ссылочного типа они хотя бы ожидаемы.

Листинг 4.43 ❖ Тривиальный пример доступа к статическому полю ссылочного типа

```
[MethodImpl(MethodImplOptions.NoInlining)]
public void Method3()
{
   Console.WriteLine(ExampleClass.StaticObject.Value);
}
```

Листинг 4.44 ❖ Результат JIT-компиляции кода из листинга 4.43

```
mov  rcx,19510002940h      ; адрес в LOH (внутри таблицы описателей)
mov  rcx,qword ptr [rcx]   ; разыменовать описатель (rcx содержит адрес упакованной структуры)
mov  ecx,dword ptr [rcx+8] ; обратиться к первому полю упакованной структуры
call 00007ff9`3c9c2b60 (System.Console.WriteLine(Int32), mdToken:000000000600007e)
```

> Точно такой же код (естественно, с другими адресами) был бы сгенерирован, если бы тип `ExampleClass` был структурой. Важен тип самого статического поля, а не то, частью какого типа оно является.

Резюме

Первые три главы были мало связаны с .NET. Мы рассмотрели некоторые вопросы, относящиеся к алгоритмам и архитектуре компьютеров. Но в этой главе все изменилось. Мы приступили к предметному изучению .NET. Начали с краткого исторического обзора, а затем перешли к внутреннему устройству .NET. Несколько страниц были посвящены описанию различных областей памяти .NET-процесса. Некоторые из них мы обсудили более глубоко, например рассказали о том, как диагностировать возникающие в них проблемы. С этой целью мы ввели новый способ изложения – сценарии, которые призваны описать проблему и возможные подходы к ее анализу. Надеюсь, вы почувствовали, что изучаете не только теорию, но и вполне практические вопросы управления памятью в .NET.

Хотя мы уже достаточно далеко углубились в тему, самого сборщика мусора мы даже не коснулись. К некоторым моментам, упомянутым в этой главе, мы еще будем иногда возвращаться. Но нетрудно заметить, что большая часть этой главы посвящена системе типов и различным аспектам тех или иных категорий типов в .NET. После того как мы так много говорили о структурах и классах, уместно будет коротко подвести итог их сильным и слабым сторонам.

Структуры

- Улучшенная локальность данных – все данные находятся в одном месте, а их хранение не сопровождается накладными расходами, поэтому кеш используется гораздо эффективнее.
- Память можно выделять в стеке – в некоторых ситуациях структурные локальные переменные размещаются в стеке, что гораздо дешевле и не требует внимания GC (и всех связанных с этим издержек) в будущем.
- Допускают очень эффективную оптимизацию – как мы видели, иногда в машинном коде не остается никаких следов от структуры, а вся обработка производится только с помощью регистров.
- Опасность непреднамеренной упаковки – при беспечном отношении структуры могут подвергаться упаковке, что приводит к скрытому выделению памяти.
- Труднее для понимания – семантика передачи по значению и другие тонкие моменты иногда кажутся не столь интуитивно очевидными, как хорошо знакомое поведение классов.
- Большинство достоинств в плане производительности зависит от реализации – сейчас работает, но не гарантируется, что будет работать в будущем, если детали реализации изменятся.

Классы

- «Просто работают» – классы являются основными строительными блоками, написанный с их помощью код «просто работает». Мы привыкли к ним и воспринимаем как должное.
- Накладные расходы на сборку мусора – память для экземпляров класса выделяется в куче, что добавляет работы сборщику мусора.

Теперь самое время ввести дополнительные правила, относящиеся к материалу этой главы. Их несколько, потому что рассматриваемые темы все ближе и ближе к практике. Отметим, что правило «Избегайте скрытого выделения памяти», тесно связанное с конкатенацией строк, описанной в этой главе, будет сформулировано в главе 5.

Правило 6: подвергайте программу измерениям

Обоснование. Трудно сказать, много ли памяти потребляет программа, если вы не знаете, как это измерить. Ответить на вопрос, сколько памяти занимает программа, тоже нелегко. Существуют различные метрики, и без их глубокого понимания мы просто застынем с разведенными руками – не будем знать, как сравнить разные программы по размеру и какие указания дать заказчику относительно проверки.

Как применять. Пользуясь знаниями, полученными во второй и четвертой главах, мы можем точно сказать, что означает каждая метрика, относящаяся к размеру программы. Анализ любой проблемы, относящейся к памяти, следует начинать с выяснения размера программы и динамики его изменения. В первую очередь внимание нужно обращать на самый важный размер – сколько занято физической памяти. Также интерес представляют байты исключительного пользования и бай-

ты виртуальной памяти. Лишь располагая результатами всех этих измерений, мы будем иметь достаточно широкий контекст для дальнейшего анализа.

Иллюстрирующие сценарии: 4.1.

Правило 7: не считайте, что утечек памяти не бывает

Обоснование. Соблазнительно думать, что в управляемой среде .NET никаких утечек памяти быть не может. Память освобождается автоматически, так о чем же беспокоиться? Это почти всегда правда и выдающееся инженерное достижение создателей среды выполнения .NET. Но все же существуют ситуации (и не одна), когда можно получить удар в спину в самый неподходящий момент. И конечно же, они проявятся в производственной среде заказчика.

Как применять. Просто не делайте такого предположения. Подвергайте свою программу измерениям (правило 6) и измеряйте GC как можно раньше (правило 5). Следите за подозрительной динамикой, особенно когда какой-то из наблюдаемых размеров демонстрирует тенденцию к неограниченному росту.

Иллюстрирующие сценарии: 4.2, 4.3 и 4.4.

Правило 8: подумайте об использовании структур

Обоснование. Классы в объектно-ориентированном программировании на C# настолько широко распространены, что мы используем их по умолчанию, даже не задумываясь. Классы «просто работают», так что еще нужно? Однако структуры были придуманы не без причины. Включите их в свой повседневный арсенал. Не надо сразу же начинать использовать их повсюду. Просто обдумывайте эту возможность, вооружившись полученными знаниями.

Как применять. Почитайте о структурах. Оцените их плюсы и минусы. Разберитесь в семантике передачи по значению и по ссылке. Измеряйте как можно раньше, чтобы понять, имеет ли смысл тратить силы на оптимизацию изучаемой части кода. Если да, попробуйте воспользоваться утечками абстракции структур – выделением памяти в стеке, JIT-оптимизацией и т. д. И если решите применить структуры в своем коде, помните о возможности передавать их по ссылке – параметрах с ключевым словом `ref`, локальных ссылочных переменных и возврате значений по ссылке. Это позволит еще сильнее повысить производительность. Также не забывайте, что стек – ограниченный ресурс, не надо думать, что в нем можно хранить огромный объем данных.

Правило 9: подумайте об интернировании строк

Обоснование. Строки – один из наиболее часто используемых типов почти в любой программе. А хранить в памяти много повторяющихся строк, конечно же, неэффективно. Для строковых литералов об устранении дубликатов позаботится среда выполнения .NET. А если мы хотим сделать то же самое для динамически создаваемых строк (например, загруженных из файла или полученных из HTTP-запроса), то можем интернировать строки вручную.

Как применять. Измерьте, много ли в программе повторяющихся строк. Примите во внимание время их жизни и степень уникальности. Много ли дубликатов строк существует в процессе в течение нескольких минут или часов? Или речь идет о кратковременных всплесках временных строк в течение обработки каких-

то входных данных? У интернирования строк есть недостатки, выигрыш можно получить только в первом случае. Помните, что интернированная строка будет жить до завершения приложения, так что это рискованное решение, которое надо принимать, все тщательно взвесив.

Иллюстрирующие сценарии: 4.5.

Правило 10: избегайте упаковки

Обоснование. Операция упаковки преобразует тип значений в ссылочный тип. При этом производится скрытое выделение памяти, потому что ссылочные типы размещаются в куче. Правило 14 «Избегайте выделения памяти» – один из важнейших подходов к оптимизации, поэтому всюду, где возможно, следует избегать упаковки, особенно ввиду того, что часто это делается неявно, без нашего ведома.

Как применять. Узнайте о типичных случаях неявной упаковки и старайтесь избегать их. Можно измерить GC на ранних этапах разработки (правило 5) с целью понять, часто ли программа выделяет память и не является ли упаковка одной из причин. Найти места неявной упаковки поможет расширение Heap Allocations Viewer для Visual Studio и подключаемый модуль Roslyn C# Heap Allocation Analyzer для ReSharper.

Глава 5

Разделение памяти на части

Из предыдущей главы мы уже узнали кое-что о внутреннем устройстве памяти в .NET. Мы заглянули внутрь памяти процесса, исполняющего управляемый код. Как мы видели, эта память разделена на много разных частей. Одни используются самим .NET Framework. Другие служат для взаимодействия с операционной системой. Но для нас важнее области, которые называются управляемой кучей.

Как было сказано в главе 4, некоторые из них содержат различные данные, необходимые движку выполнения, например описание типов. Это доменная куча, низкочастотная и высокочастотная кучи. Но среди всех этих куч есть одна, которая используется исключительно для нужд сборщика мусора (рис. 5.1). Это сегменты памяти, содержащие кучу (или свободную память) в смысле, определенном в главе 1 с точки зрения CLI. Будем называть эти области управляемой кучей сборщика мусора (или для краткости кучей GC).

0000008BF8800000	Private Data	2,048 K	52 K	52 K	52 K	52 K	3 Read/Write	Thread Environment Block ID: 18376
0000008BF8A00000	Thread Stack	1,536 K	20 K	20 K	8 K	8 K	3 Read/Write/Guard	Thread ID: 17980
0000008BF8B80000	Thread Stack	1,536 K	20 K	20 K	8 K	8 K	3 Read/Write/Guard	Thread ID: 2432
0000008BF8D00000	Thread Stack	1,536 K	20 K	20 K	8 K	8 K	3 Read/Write/Guard	Thread ID: 14436
0000008BF8E80000	Thread Stack	1,536 K	20 K	20 K	8 K	8 K	3 Read/Write/Guard	Thread ID: 19740
0000008BF9000000	Thread Stack	1,536 K	20 K	20 K	8 K	8 K	3 Read/Write/Guard	Thread ID: 18712
0000008BF9180000	Free	1,992,407,552 K						
0000026700000000	Managed Heap	393,216 K	336 K	336 K	224 K	224 K	4 Read/Write	GC
0000026718000000	Free	1,610,688 K						
00002677A4F0000	Heap (Shareable)	64 K	64 K		4 K		1 Read/Write	Heap ID: 2 [COMPATABILITY]
00002677A500000	Private Data	4 K	4 K	4 K	4 K	4 K	1 Read/Write	
00002677A501000	Unusable	60 K						
00002677A510000	Shareable	88 K	88 K		88 K		1 Read	

Рис. 5.1 ❖ Среди различных куч в процессе, исполняющем .NET-приложение, есть одна наиболее интересная для нас – куча GC, содержащая все объекты, созданные нашей программой

Во время работы приложения распределитель среды выполнения .NET выделяет память в куче GC. Сборщик, реализованный в среде выполнения .NET, отслеживает достижимость объектов в куче GC и освобождает память, занятую теми объектами, которые уже недостижимы.

Как мы видели в предыдущей главе, странное поведение любой из вышеупомянутых куч может быть признаком какой-то проблемы. Тем не менее, с точки зрения .NET разработчика, наибольший интерес представляет куча GC. Таким образом, можно сказать, что оставшаяся часть книги будет посвящена этой области памяти.

Стратегии разделения памяти

Размер кучи GC может увеличиваться до многих гигабайтов. С точки зрения распределителя, это не обязательно проблема. Но, принимая во внимание столь большие размеры, трудно представить, как сборщик может обрабатывать так много данных единообразно. Трудно справиться с гигабайтами данных, уложившись в приемлемые временные рамки. При проектировании сборщика мусора один из самых важных параметров – его накладные расходы. В числе прочего, например, на какое время приостанавливается работа потоков для сборки мусора. Или сколько ресурсов процессора занимает сборка. Желательно, чтобы пауза длилась не дольше миллисекунды. Но вследствие задержек доступа к памяти, описанных в главе 2, за время порядка миллисекунд можно прочитать мегабайты, а не гигабайты данных. Именно поэтому одно из самых важных решений для любой реализации сборщика мусора – *стратегия разделения памяти*.

Проще говоря, мы хотим разделить кучу GC на меньшие части, чтобы иметь возможность работать с ними независимо. Если сделать это разумно, то можно очень сильно ускорить работу сборщика мусора, поскольку, как выясняется, нет никакой нужды трактовать все данные одинаково во время выполнения программы.

Существует много разных стратегий разделения. Обычно они основаны на одном из свойств существующего объекта.

- Размер – мы можем разделить кучу GC на части, по размеру объектов. Например, можно обрабатывать малые объекты иначе, чем большие. Особенно это важно при использовании уплотняющей сборки. Копирование больших объектов может быть сопряжено со значительными накладными расходами, поэтому мы можем уплотнять только области малых объектов, а для больших использовать сборку очисткой.
- Время жизни – эта характеристика объекта очень важна. Интуитивно кажется, что обрабатывать короткоживущие объекты стоит иначе, чем те, которые существуют в течение всего времени работы приложения. Понятно, что будущее нам неизвестно, но, по крайней мере, можно различать объекты, живущие долго, и те, что созданы недавно. Области памяти для объектов с разным временем жизни называются *молодыми/старыми* поколениями – или поколениями с порядковыми номерами.
- Изменяемость – это одно из самых важных свойств объектов. Объекты, которые нельзя изменять после создания, имеет смысл обрабатывать иначе, чем изменяемые.
- Тип – можно по-разному обрабатывать объекты разных типов. Быть может, нужно поддерживать разные кучи для строк, целых чисел и других специальных классов, реализаций интерфейсов или атрибутов? Тут есть разные варианты.
- Вид – объекты можно классифицировать по-разному и соответственно разделять память. Например, содержит ли объект какие-нибудь указатели (исходящие ссылки)? Если нет, то о них необязательно думать при уплотнении других объектов. Был ли объект *закреплен* (закрепление описывается в главе 7), так что его не будут перемещать даже во время уплотнения? Если да, то, быть может, надо бы переместить его в другой раздел памяти, чтобы

избежать накладных расходов, связанных с перемещением других объектов в обход закрепленных экземпляров[1].

В реализациях Microsoft .NET и Mono выбраны только первые две стратегии. В обоих случаях сборщики мусора не особенно интересуются типом или изменяемостью объекта, а просто выделяют запрошенное число байтов (удовлетворяют запросы вида «дай мне N байт для нового объекта»). Но дизайн GC постоянно изменяется, и кто знает, быть может, в будущем в .NET или в Mono будет реализована какая-то дополнительная стратегия.

А теперь внимательно изучим обе стратегии разделения. Как всегда, наибольшее внимание уделим реализации Майкрософт, лишь попутно делая замечания о Mono и других средах выполнения.

Разделение по размеру

Первая стратегия заключается в том, чтобы по-разному обрабатывать объекты разного размера. Как уже отмечалось, главная причина, стоящая за ней, – накладные расходы на копирование памяти в случае сборки с уплотнением. Поскольку не видно никакого обоснования в пользу разделения на несколько областей по размеру, было выбрано одно пороговое значение, определяющее границу между большим и малым объектами. И куча GC разделяется на две физически разнесенные области памяти:

- *куча малых объектов* (Small Object Heap – SOH) – здесь создаются объекты, по размеру меньшие 85 000 байт;
- *куча больших объектов* (Large Object Heap – LOH) – здесь создаются объекты, по размеру большие или равные 85 000 байт.

Логика работы и код у них в основном общий, но есть и важные различия. Заметим, что порог равен 85 000 байт, но многие неправильно считают, что это 85×1024 байт, т. е. 85 КиБ (или, по старинке, 85 КБ).

Поскольку мы таким образом разделили «малые» и «большие» объекты, то можем обрабатывать обе кучи по-разному:

- для SOH можно использовать сборку с уплотнением, потому что для малых объектов копирование не так страшно. В главе 7 мы увидим, что в случае кучи для малых объектов Майкрософт реализовала сборку очисткой и уплотнением на месте. На дополнительном этапе планирования принимается решение, какую из них выбрать;
- в LOH используется только сборка очисткой, поскольку стоимость уплотнения (копирования) больших объектов велика (хотя пользователь может явно активировать уплотнение LOH).

В версии Mono 5.4 пороговое значение – 8000 байт. Все более крупные объекты создаются в области, которая в Mono называется памятью больших объектов (Large Ob-

[1] Однако это гораздо труднее, чем кажется. Объекты в .NET при создании не закреплены, мы можем закреплять и откреплять их в любой момент. Поэтому создавать отдельную область для закрепленных в настоящий момент объектов может быть только хуже, поскольку во время закрепления и открепления придется копировать объекты туда и обратно.

> ject Store – LOS), а все объекты меньшего размера – в области, называемой детской (Nursery). Как и в Microsoft .NET, область малых объектов можно уплотнить, тогда как LOS освобождается только очисткой.

Возникает вопрос, почему выбран порог 85 000 байт, а не какой-то другой. Как мы уже неоднократно видели и еще не раз увидим, тому есть целый ряд как инженерных, так и исторических причин. Самый простой ответ – это значение было выбрано экспериментально на основе многочисленных тестов, проведенных в начале разработки .NET. Ходят слухи, что эти тесты проводились в контексте программы SharePoint, но слухи эти неподтвержденные. Прорабатывались самые разные сценарии с привлечением внутренних и внешних команд. А с тех пор не появилось никаких фактов в пользу того, что изменение этого значения принесло бы какие-то выгоды.

> Также интересно, к какому именно размеру относится порог 85 000 байт. Очевидно, речь идет о поверхностном размере объекта – ссылки считаются как ссылки, а не как размеры объектов, на которые они указывают. Поэтому в LOH мы чаще всего встречаем … массивы. Трудно вообразить объект, у которого так много больших полей, что его поверхностный размер превосходит 85 000 байт. Следует также иметь в виду, что объект, одним из полей которого является большой массив, сам большим не является; это поле – всего лишь короткая ссылка на массив.

Стоит упомянуть одну важную деталь реализации. SOH предъявляет разные требования к выравниванию памяти на различных платформах. В 32-разрядной среде выполнения граница выравнивания равна 4 байтам. Это означает, что начальные адреса всех объектов кратны 4. При этом никогда не происходит обращения к невыровненной памяти, что привело бы к заметному снижению производительности. На 64-разрядной платформе адреса в SOH выравниваются на границу 8 байт. LOH в этом смысле отличается, потому что граница выравнивания всегда равна 8 байтам вне зависимости от разрядности. Для 64-разрядной платформы это кажется естественным. Но почему на 32-разрядной адреса выравниваются на границу 8, а не 4 байт? Так было сделано в основном ради массивов чисел с двойной точностью, чтобы доступ к ним был выровнен (объяснения скоро последуют). А поскольку 8 байт – очень мало по сравнению с размером объекта в LOH, такое выравнивание не приносит неудобств.

Куча малых объектов

Куча малых объектов – безусловно, самая популярная область памяти, потому что размер большей части создаваемых нами объектов меньше 85 000 байт. Поэтому обычно количество объектов, находящихся в SOH, на несколько порядков превосходит количество объектов в LOH. Но когда объектов слишком много, возможны проблемы (например, при обходе большого графа на этапе пометки), а значит, имеет смысл подумать о разделении этой области еще на несколько частей. Такое решение – разделять объекты по времени жизни – принято в большинстве известных систем с автоматическим управлением памятью.

Поскольку организация кучи малых объектов тесно связана с разделением по времени жизни, все последующие детали будут изложены позже.

Куча больших объектов

Кучу больших объектов иногда называют поколением 3 (индексы 0, 1 и 2 зарезервированы для трех поколений, находящихся в SOH). Идея этой кучи проста – хранить все объекты размером, равным или больше 85 000 байт.

> С точки зрения сборщика, большие объекты в LOH логически принадлежат поколению 2, поскольку убираются в мусор, только когда производится сборка для поколения 2.

Предполагается, что для больших объектов память выделяется нечасто, потому что в большинстве программ большие структуры данных нужны не так часто. Иногда это предположение неверно, что может привести к падению производительности (см. правило 15 «Избегайте чрезмерного выделения памяти в LOH» в главе 6). Вообще говоря, в LOH действительно размещаются только объекты размера больше 85 000 байт. Но есть несколько исключений.

Куча больших объектов – массивы чисел типа double

Самое заметное исключение – массивы чисел типа `double` в 32-разрядной среде выполнения (даже если она работает на 64-разрядной машине). Такие массивы считаются «большими объектами», и потому память для них выделяется в LOH, если количество элементов больше или равно 1000 (см. листинг 5.1). Поскольку длина типа `double` равна 8 байтам, это означает, что LOH содержит массивы, начиная с размера 8000 байт, что нарушает общее правило 85 000 байт.

Листинг 5.1 ❖ В 32-разрядной среде выполнения .NET массив, содержащий 1000 или более элементов типа double, размещается в LOH, поэтому следующая программа печатает 0 и 3 соответственно

```
double[] array1 = new double[999];
Console.WriteLine(GC.GetGeneration(array1)); // печатается 0
double[] array2 = new double[1000];
Console.WriteLine(GC.GetGeneration(array2)); // печатается 3
```

С чего бы такое странное и узкоспециальное исключение? В данном случае причина связана с выравниванием, а не с издержками копирования. Длина типа `double` равна 8 байтам. Невыровненный доступ к `double` обходится очень дорого (гораздо дороже, чем для целых типов). В 64-разрядной среде это не страшно, потому что граница выравнивания и в SOH, и в LOH равна 8 байтам. Но в 32-разрядной среде, где адреса в SOH выравниваются на границу 4 байт, это может составлять проблему.

Поэтому имеет смысл использовать LOH, где, как уже сказано, адреса всегда выравниваются на границу 8 байт. Тогда мы избежим высокой стоимости невыровненного доступа для больших массивов. Но раз так, то почему не размещать все массивы типа `double` в LOH на 32-разрядной платформе? У размещения в LOH есть свои недостатки – поскольку эта куча не уплотняется, большое количество мелких структур может привести к нежелательной фрагментации. Поэтому решение размещать там только массивы больше определенного размера – компромисс между стоимостью невыровненного доступа и фрагментацией. Ну а значение 1000 было выбрано экспериментально.

При работе с 32-разрядной платформой все равно следует помнить о фрагментации, вызванной массивами чисел типа double. Например, для некоторых видов обработки сигналов характерно создание и освобождение большого числа таких массивов, содержащих больше 1000 элементов. В подобной ситуации следует организовать повторно используемый буфер (пул) массивов, а не создавать каждый раз новый массив. Дополнительные сведения см. в сценарии 6.1.

Куча больших объектов – внутренние данные CLR

Других исключений из правила выделения в LOH объектов размером больше порогового нет. Но LOH также используется самим .NET Framework для хранения некоторых дополнительных данных. Мы уже дважды упоминали это в главе 4, в контексте интернирования строк и статических полей. Мы имеем в виду структуру LargeHeapHandleTable. Пора сказать о ней несколько слов.

LARGEHEAPHANDLETABLE LargeHeapHandleTable – структура данных, обслуживаемая самой средой выполнения .NET. Она служит для управления массивами, выделенными в куче больших объектов для внутренних целей самой среды. Она организована в виде кластеров (как выглядит эта структура в CoreCLR, показано на рис. 5.2). Каждый кластер представляет один массив Object[], выделенный в LOH. Эти массивы закреплены, т. е. сборщику мусора запрещено их перемещать. Объясняется это тем, что различные неуправляемые части CLR могут хранить указатели на элементы массива, поэтому их перемещение повлекло бы за собой необходимость обновлять указатели, что очень нелегко.

В каждом кластере хранится закрепленный описатель (handle) соответствующего массива. Там же (для удобства) хранится прямой указатель на начало данных массива (m_pArrayDataPtr) и текущий индекс первого еще не использованного элемента (m_currentPos), поскольку эти массивы создаются с запасом. Если все элементы массива уже использованы, то создается новый кластер (и, стало быть, новый массив Object[] в LOH). Кластеры, хранящиеся в LargeHeapHandleTable, связаны в односвязный список (в каждом кластере хранится указатель m_pNext на следующий кластер или null, если элемент списка последний).

Как уже было сказано, у структуры LargeHeapHandleTable двоякое назначение. Вот выдержка из исходного кода CoreCLR:

```
// Таблицу LargeHeapHandleTable можно найти в двух местах:
// 1) в каждом BaseDomain, используется для отслеживания статических членов в этом домене;
// 2) в системном домене, используется для карты GlobalStringLiteralMap
```

Все это также показано на рис. 5.2. Иными словами, в LOH хранятся:

- один или более массивов Object[] для карты глобальных строковых литералов (или пула интернированных строк) – управляются одной структурой LargeHeapHandleTable, состоящей как минимум из одного кластера;
- один или более массивов Object[] для каждого домена, используемые для статических данных, – управляются структурой LargeHeapHandleTable в домене BaseDomain, состоящей как минимум из одного кластера.

Класс SystemDomain описывает домен и наследует BaseDomain, поэтому содержит член m_pLargeHeapHandleTable, но он не используется – системный домен не содержит управляемых модулей, поэтому и в хранении статических членов нужды не возникает.

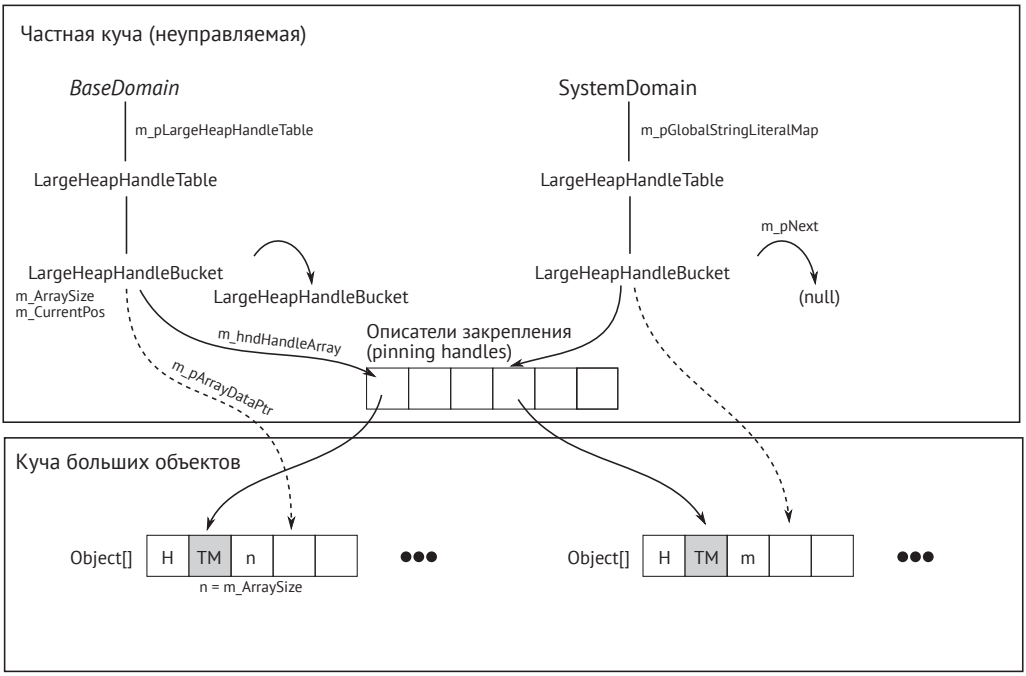


Рис. 5.2 ❖ Структура LargeHeapHandleTable

Массивы, содержащие таблицы описателей, можно увидеть, к примеру, в WinDbg. После присоединения к .NET-процессу загрузите расширение SOS и выведите список всех относящихся к GC областей памяти командой `eeheap` (листинг 5.2). Найдя диапазон адресов, соответствующий LOH, выполните команду `dumpheap`, чтобы вывести список всех содержащихся в нем объектов. В листинге 5.2 приведены также результаты для простой консольной программы «Hello world». Как видим, в этой программе есть только три массива `Object[]` (столбец со значением `00007ffb8f34a5b8` соответствует таблице методов `Object[]`).

Листинг 5.2 ❖ Применение WinDbg и расширения SOS для перечисления таблиц описателей в куче больших объектов

```
> .loadby sos clr
> !eeheap
...
Large object heap starts at 0x000001e5ad231000
         segment            begin        allocated              size
000001e5ad230000 000001e5ad231000 000001e5ad235480 0x4480(17536)
> !dumpheap 000001e5ad231000 000001e5ad235480
         Address               MT   Size
000001e5ad231000 000001e59afc2ff0     24 Free
000001e5ad231018 000001e59afc2ff0     30 Free
000001e5ad231038 00007ffb8f34a5b8   8184
000001e5ad233030 000001e59afc2ff0     30 Free
000001e5ad233050 00007ffb8f34a5b8   1048
000001e5ad233468 000001e59afc2ff0     30 Free
000001e5ad233488 00007ffb8f34a5b8   8184
```

Вот эти три массива:

- по адресу `000001e5ad231038` – таблица описателей для домена Domain 1 (который содержит большинство библиотек и модулей, в т. ч. саму программу);
- по адресу `000001e5ad233050` – пул интернированных строк;
- по адресу `000001e5ad233488` – таблица описателей для разделяемого домена (который в случае простого консольного приложения может содержать только модуль `System.Private.CoreLib.dll`).

> Если вам интересно, почему в листинге 5.2 видны также очень маленькие области Free, то ответ вы найдете в разделе главы 6, посвященном выделению памяти в куче больших объектов.

К сожалению, в настоящее время нет уверенного способа узнать, какой массив для чего предназначен, – это можно понять, в основном глядя на содержимое каждого из них (выполнив команду `dumparray` для каждого адреса).

Понятно, что пул интернированных строк содержит ссылки на интернированные строки. Остальные два массива содержат различные статические члены используемых библиотек и нашего кода. Там же находятся строки, созданные в процессе разрешения строковых литералов из сборок, созданных программой NGEN (без использования интернирования строк из-за параметра `NoStringIntern`).

У хранящихся в таблицах описателей есть еще одно применение – среда выполнения использует их для хранения различных данных, относящихся к отражению (reflection). Если используются `GetType`, `typeof` или еще какие-то элементы Reflection API, то описатели `RuntimeType` и прочей информации также хранятся в таблицах. Это позволяет найти разнообразные связанные с типами объекты по ссылкам в этих массивах.

Крайне маловероятно, что `LargeHeapHandleTable` создаст какие-то проблемы приложению. Для этого нужно было бы создать уйму статических членов (динамически) или загрузить много динамических доменов приложений. Еще одной причиной могло бы быть интернирование большого числа строк. Если вы увидите много больших массивов `Object[]` в куче больших объектов, единственным корнем которых является закрепленный описатель, то это может означать, что вы столкнулись с одной из этих редких ситуаций. Однако, поскольку в этих массивах хранятся только ссылки, вы, вероятно, сначала заметите много таких объектов где-то еще.

Разделение по времени жизни

Как уже было сказано, из-за возможности размещения очень большого числа объектов в куче малых объектов было принято решение разделить ее на части по времени жизни объектов. Эта концепция называется *сборкой мусора на основе поколений*, поскольку к одному поколению относятся объекты со схожим временем жизни. Есть много определений времени жизни, но мы будем придерживаться двух наиболее очевидных.

- Абсолютное время – мы можем каким-то образом связать время жизни объекта с реальным временем. Самый простой способ – взять число тактов процессора с момента создания объекта. Но у этого подхода есть и недостатки. Что считать «долгой жизнью»? А короткой? Секунда – это долго или корот-

ко? Дать общий ответ почти невозможно, поскольку он зависит от характеристик конкретной программы – сколько объектов она создает, как часто они должны убираться в мусор и т. д. Можно было бы придумать механизм самообучения, который вычисляет границу между короткой и долгой жизнями, но это чересчур сложно.

- Относительное время – вместо реального времени можно связать время жизни объекта с каким-то конкретным событием, например с самой сборкой мусора. Тогда мы подсчитываем, сколько сборок мусора пережил объект. Можно в каждом объекте завести внутренний счетчик, в котором будет храниться количество пережитых сборок. Если оно превосходит некий заданный (или вычисленный) порог, то такой объект рассматривается как «пожилой».

Можно представить себе и менее очевидные способы определения времени жизни объекта. Например, если сборщик и распределитель спроектированы так, что объекты никогда не вставляются в области памяти с адресами, меньшими уже достигнутого, то можно вычислять возраст объекта как разность между его адресом и каким-то другим местом в памяти.

Интересно отметить, что многие описания сборки мусора в .NET начинаются с того, что она основана на поколениях. Но, как видите, прежде чем добраться до этой детали реализации, мы много чего изучили.

Но в чем вообще смысл сборки мусора на основе поколений? Почему разделение объектов по возрасту и разные подходы к обработке поколений дают какой-то эффект? Этот вывод следует из наблюдения, называемого *гипотезой о поколениях*. На самом деле есть два варианта этой гипотезы: слабый (менее общий) и сильный (более общий), которые в совокупности составляют теоретическую основу сборщиков мусора на основе поколений. Правда, они противоречат интуитивным представлениям о человеческой жизни.

- *Слабая гипотеза о поколениях* – самые молодые объекты живут недолго. Иными словами, большинство объектов, созданных программой, быстро выходят из употребления. Это временные объекты, представленные локальными переменными, скрытым выделением памяти и участвующие в любого рода быстрой обработке. Эта гипотеза подтверждена различными исследованиями.
- *Сильная гипотеза о поколениях* – чем дольше живет объект, тем более вероятно, что он проживет еще. Это такие долгоживущие объекты, как кеши, «диспетчеры», «помощники», пулы объектов, технологические процессы и т. д. Но исследования не подтверждают эту гипотезу в полной мере, поскольку характеристики времени жизни объекта бывают гораздо сложнее, чем можно выразить в одном предложении. Даже общепринятой формулировки этой гипотезы не существует.

Мы можем выгодно использовать это знание о распределении объектов по возрасту (рис. 5.3). Память, занятую молодыми объектами, имеет смысл освобождать как можно раньше (выделив их в «молодое» поколение), если большая их часть быстро умирает. А освобождать память, занятую старыми объектами, следует гораздо реже (для них поддерживается «старое» поколение), коль скоро они умирают редко. И конечно, можно создать сколько угодно «временных» промежуточных поколений между ними.

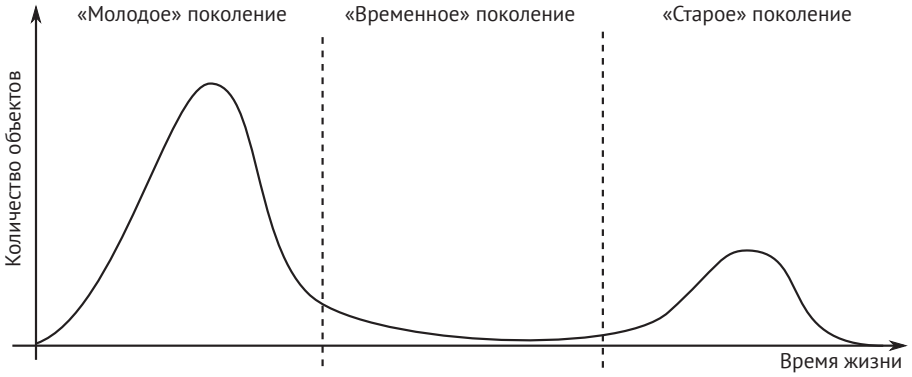


Рис. 5.3 ❖ Иллюстрация слабой и сильной гипотез о поколениях в виде количества живых (или менее точно – достижимых) объектов в зависимости от возраста

Объекты, принадлежащие разным поколениям, можно обрабатывать по-разному. Например, можно производить сборку мусора только для самого молодого поколения или только для самого старого. Можно также собирать мусор во всех поколениях, что обычно называют *полной сборкой мусора*.

Когда объект достигает порогового возраста, он переводится в следующее поколение, т. е. считается принадлежащим более старому поколению. Что в действительности означает такой перевод, и почему его реализация в разных вариантах GC существенно различается?

Один из подходов – копирование в другую область памяти. В таком случае реализуется вариант, упомянутый в главе 1 (рис. 1.18). Предположим, что поколения организованы, как показано на рис. 5.4, т. е. имеется три разные области памяти для поколений 0, 1 и 2. Тогда последовательность действий могла бы быть такой:

- проработав некоторое время, программа создала объекты A, B и C – они размещены в самом молодом поколении 0 (рис. 5.4a);
- спустя некоторое время началась сборка мусора, предположим, что к этому моменту объект A стал недостижимым. Тогда в поколение 1 следует скопировать только объекты B и C (рис. 5.4b);
- спустя еще некоторое время создан объект D – память для него выделена в поколении 0 (рис. 5.4c);
- затем сборка мусора была запущена снова, предположим, что теперь объект B недостижим. Таким образом, объекты C и D скопированы в старшие поколения (рис. 5.4d);
- через какое-то время мы создали объект E и разместили его в поколении 0 (рис. 5.4e).

Иногда можно встретить утверждение, что в Microsoft .NET поколения работают именно таким интуитивно понятным образом. Хорошенько запомните, что это не так. Майкрософтовская реализация CLR устроена несколько иначе – она сложнее, но эффективнее (см. главу 7).

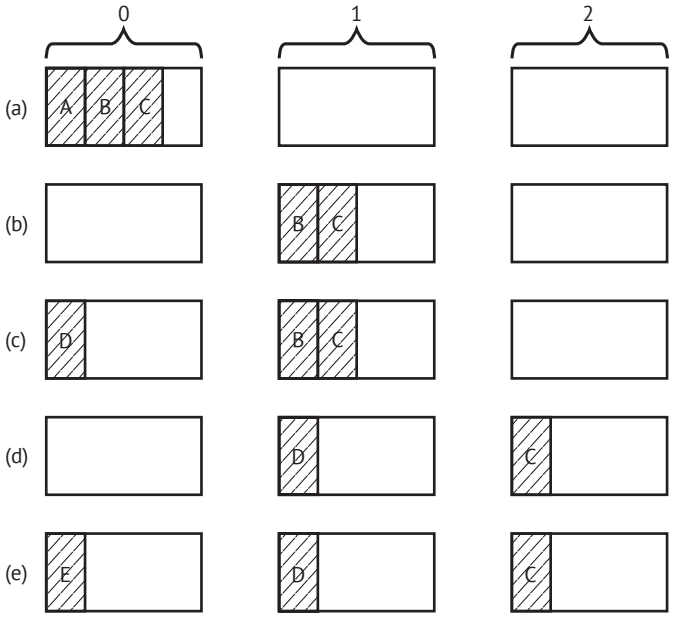


Рис. 5.4 ❖ В случае сборки мусора с копированием поколения являются отдельными областями памяти. Перевод в следующее поколение означает, что объект копируется в другую область

При другом подходе поколения можно определить логически – границами адресов. Тогда перевод означает сдвиг границы, а не копирование самих объектов (рис. 5.5). Это гораздо быстрее, чем копирование, потому что сдвиг логических границ почти не занимает времени. Дополнительно можно уплотнить выжившие объекты, хотя это необязательно (и заметно усложнит реализацию). Пусть поколения организованы, как показано на рис. 5.5, где имеется один непрерывный блок памяти. Тогда возможна такая последовательность действий:

- проработав некоторое время, программа создала объекты A, B и C – пока что существует только одно, самое молодое поколение 0 (рис. 5.5a). Границы поколений 1 и 2 вырождены, размер этих поколений равен нулю или очень мал (это зависит от конкретных деталей реализации);
- спустя некоторое время началась сборка мусора, предположим, что к этому моменту объект A стал недостижимым. Предположим также, что выполняется простая сборка очисткой. Память объекта A возвращена системе. И поскольку теперь объекты B и C принадлежат более старому поколению 1, то мы сдвигаем границу, устанавливая ее после объекта C (рис. 5.5b). При этом граница поколения 0 тоже корректируется. Никакого копирования не потребовалось;
- спустя еще некоторое время создан объект D – память для него выделена в поколении 0 (рис. 5.5c);

- затем сборка очисткой была запущена снова, предположим, что теперь объект B недостижим, поэтому его память возвращена. Мы должны снова скорректировать границы поколений. Теперь объект D принадлежит поколению 1, а C – поколению 2 (рис. 5.5d). Граница поколения 0 тоже сдвигается;
- через какое-то время мы создали объект E и разместили его в поколении 0 (рис. 5.5e).

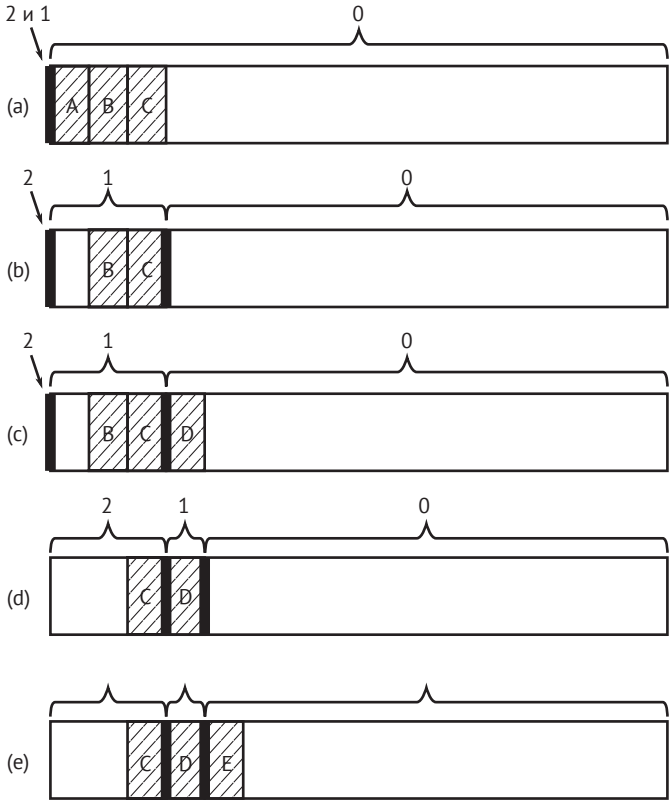


Рис. 5.5 ❖ Поколения как логические границы внутри одной непрерывной области памяти. Перевод в следующее поколение – это всего лишь логический переход вследствие изменения границы поколений

Именно так устроена работа с поколениями в средах выполнения Microsoft .NET. Было решено создать три поколения с последовательными номерами, как в примерах выше. То есть мы имеем поколение 0 (молодое), поколение 1 (временное) и поколение 2 (старое). Еще одно решение связано с вычислением границ между поколениями. В Microsoft .NET это очень просто – объект переводится в следующее поколение, если переживает сборку мусора.

> Из этого правила есть исключения, которые мы называем *оставлением* (demotion) (или просто неперемещением). Почему так может случиться, будет описано в следующих главах, поскольку это тесно связано с механизмами работы распределителя и сборщика.

Иными словами, когда объект переживает поколение *N*, он становится частью поколения *N* + 1 (говорят, что он переведен в поколение *N* + 1). Это также означает, что после всего двух последовательных GC он может оказаться в поколении 2 и оставаться там до тех пор, пока в нем не отпадет необходимость.

> В Mono, основной альтернативе Microsoft .NET, принята аналогичная организация для малых объектов (меньше 8000 байт). Различается только два поколения – «молодое», называемое детской (Nursery), и старое, которое называется старой памятью или просто главной кучей. Применяется также более простой из двух описанных выше механизмов переноса объекта в старшее поколение – если объект в детской переживает сборку мусора, то он копируется в старое поколение.

Но у сборки мусора на основе поколений есть один важный недостаток. Если поведение объектов в нашем приложении не согласуется с гипотезами о поколениях, то приложение может серьезно страдать. Это приводит к важному заключению – в здоровой системе, согласующейся с гипотезами о поколениях, чем старше поколение, тем реже в нем производится сборка мусора. Мы должны стремиться к соблюдению правила 18 «Избегайте кризиса среднего возраста» (см. главу 7).

Однако большой интерес могут представлять также размеры поколений. Именно так легко проверить, имеется в приложении утечка памяти или нет. Проще всего следить за размерами поколений с помощью счетчиков производительности или механизмов ETW (см. табл. 5.1). То и другое отражает состояние кучи сразу после сборки мусора. Есть лишь два небольших подводных камня.

- По историческим причинам счетчик `\Память CLR .NET(имя процесса)\Размер кучи поколения 0` (`\.NET CLR Memory(processname)\Gen 0 heap size`) показывает не истинный размер поколения 0, а нечто, именуемое *бюджетом выделения* (проще говоря, сколько байтов нужно выделить в поколении, перед тем как в нем начнется сборка мусора). Поэтому решение, основанное на этом счетчике, может быть ошибочным.
- Следует помнить, что минимальное время между сбором информации в системном мониторе составляет одну секунду, пусть даже данные обновляются чаще. Поэтому если сборка мусора происходит чаще, чем раз в секунду, то некоторые измерения будут пропущены.

Таблица 5.1. Результаты измерения размеров поколений (где имя процесса – на самом деле имя экземпляра счетчика, соответствующего процессу)

Поколение	ETW (событие GCHeapStats_V1)	Счетчик производительности (\Память CLR .NET(имя процесса))
0	`GenerationSize0`	`Размер кучи поколения 0 (Gen 0 heap size)` (бюджет выделения)
1	`GenerationSize1`	`Размер кучи поколения 1 (Gen 1 heap size)`
2	`GenerationSize2`	`Размер кучи поколения 2 (Gen 2 heap size)`
3 (LOH)	`GenerationSize2`	`Размер кучи для массивных объектов (Large Object Heap size)`

Однако эти тонкости не слишком раздражают, потому что обычно в поколениях 0 и 1 мусора немного, так что проблем это не вызывает.

Сценарий 5.1. Как чувствует себя моя программа? Динамика размеров поколений

Описание. Мы хотим последить за размерами поколений в процессе выполнения веб-приложения. В идеале хотелось бы сделать это во время выполнения нагрузочных тестов и так, чтобы не вмешиваться в работу программы. Это дало бы субъективную уверенность в отсутствии утечек памяти. Тестированию будет подвергнуто приложение nopCommerce 4.0 – универсальная платформа электронной торговли с открытым исходным кодом, написанная на ASP.NET (обратите также внимание на сценарий 5.2, где аналогичный тест выполняется при несколько иных условиях).

Анализ. Пропустим технические детали подготовки к нагрузочному тестированию и будем считать, что все необходимые инструменты и процедуры имеются в наличии. Тест будет создавать нагрузку 7 запросов в секунду и продолжаться 170 минут, чтобы мы могли заметить утечку памяти, если она существует. Приложение nopCommerce размещается в IIS с помощью .NET Core Windows Server Hosting. Это означает, что процесс `w3wp.exe`, представляющий пул приложений, присутствует, но всего лишь передает запросы саморазмещаемому (self-hosted) веб-приложению для .NET Core. В нашем случае этот процесс называется `Nop.Web.exe`.

Во-первых, мы хотим посмотреть на общее потребление памяти, как в сценарии 4.1 из главы 4. То есть нас интересуют счетчики `Рабочий набор (частный)` (Working Set – Private), `Байт исключительного пользования` (Private Bytes) и `Байт виртуальной памяти (Virtual Bytes)` в группе `Процесс(Nop.Web)`, а также счетчик `\Память CLR .NET(Nop.Web)\Всего фиксировано байтов (\.NET CLR Memory(Nop.Web)\# Total committed Bytes)`.

Во-вторых, для наблюдения за счетчиками, перечисленными в табл. 5.1, проще всего воспользоваться системным монитором. Результаты показаны на рис. 5.6 в графической форме и в табл. 5.2 в табличной. Для наглядности поколения изображены в разных масштабах. Можно заметить, что:

- размер поколения 0 (тонкая сплошная линия) постоянно меняется между значениями 4 194 300 и 6 291 456 байт. Как было сказано выше, это не реальный размер поколения, а его бюджет выделения. Но хотя это не истинный показатель, его можно интерпретировать как признак хорошего состояния приложения. Размер поколения стабилен. Если бы он рос со временем, то рос бы и этот счетчик (хотя он отражает лишь величину, связанную с размером);
- размер поколения 1 (штриховая линия) сильно изменяется, что согласуется с его промежуточной природой. Поскольку никакого восходящего тренда не наблюдается, этот показатель тоже свидетельствует о нормальной работе приложения;
- размер поколения 2 (жирная сплошная линия) демонстрирует типичную треугольную форму – объекты накапливаются в старейшем поколении и время от времени убираются в мусор. Обычно полную сборку мусора откладывают до момента, когда она действительно необходима, поэтому периодический рост размера старых данных – ожидаемое явление. В случае веб-приложений достижимость многих объектов связана со временем жизни сеанса пользователя и, возможно, с кешированием данных. Поэтому на-

личие таких периодических треугольников нормально. Однако это все же косвенный признак возможных проблем, который нужно взять на заметку как повод для дальнейшего расследования. Следующий наш шаг – понаблюдать за этим паттерном подольше и посмотреть, нет ли тенденции к росту максимального размера поколения 2. Следует также понаблюдать за счетчиком `\Память CLR .NET(Nop.Web)\% времени в GC` (`\.NET CLR Memory(Nop.Web)\% Time in GC`) (подробнее см. сценарий 7.1), чтобы оценить накладные расходы на GC в процессе.

Заметим также, что оба поколения 0 и 1 вместе занимают совсем немного памяти, так что волноваться тут не о чем. Это типичный сценарий, поскольку утечки памяти можно опознать по монотонному росту самого старого поколения (накапливаются долгоживущие объекты).

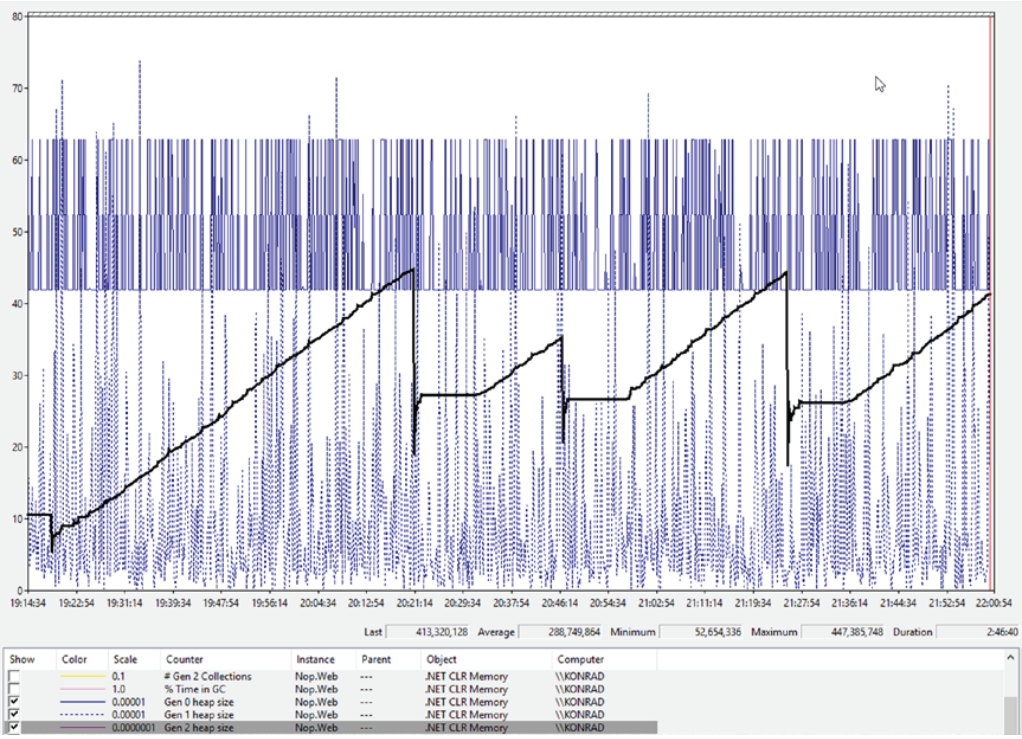


Рис. 5.6 ❖ Системный монитор – представление размеров поколений в приложении ASP.NET за три часа работы нагрузочного теста

Таблица 5.2. Сводка результатов измерений, показанных на рис. 5.6

Поколение	Минимум	Максимум
0	4 194 300	6 291 456
1	~18 268	7 384 704
2	52 654 336	447 385 748
LOH	0	38 826 368

Интересно также сравнить данные ETW с данными счетчиков производительности. Напомним, что последние отбираются раз в секунду, тогда как ETW позволяет записывать каждое событие (событие `GCHeapStats_V1` генерируется в конце сборки мусора). На рис. 5.7a, b и c это различие проиллюстрировано в случае 20-секундных промежутков времени (временной отрезок тут гораздо меньше, чтобы сделать картину более наглядной). Полученные от ETW размеры поколений записывались программой Perfview, в которой был включен режим GC Collect Only (только сборка мусора) с низкими накладными расходами. Затем данные из событий `GCHeapStats_V1` были экспортированы в CSV-файл. Счетчики производительности собирались с помощью механизма групп сборщиков данных, который позволяет записывать данные сеанса в файл (в т. ч. в формате CSV), а не строить графики в режиме реального времени. Вот что мы наблюдаем:

- данные счетчиков производительности действительно отбираются каждую секунду. Поскольку во время тестирования сайт подвергался высокой нагрузке, сборка мусора производилась гораздо чаще. Поэтому количество событий ETW намного больше;
- для поколения 0 разница между данными, собранными двумя способами, огромна (рис. 5.7a). Это объясняется вышеупомянутыми историческими причинами. Если нас действительно интересует динамика размера поколения 0, то следует использовать ETW;
- для поколения 1 видно, что некоторые измерения от счетчика производительности соответствуют данным ETW (рис. 5.7b). Однако между ними происходит много всякого. Видно, как сильно изменяется размер поколения 1. Впрочем, нам это знание в общем-то ни к чему. Измерения раз в секунду может быть вполне достаточно. В большинстве приложений GC происходит не так часто, поэтому различия может и вообще не быть (если частота GC меньше одного раза в секунду). Но знать об этом различии все-таки полезно;
- для поколения 2 данные почти совпадают (рис. 5.7c), поскольку полная сборка мусора производится гораздо реже, так что даже при использовании счетчиков производительности данные почти не теряются.

Общий вердикт положительный. Можно считать, что приложение функционирует нормально. Длительное наблюдение за счетчиками производительности не показало ничего тревожного. В этом сценарии мы представили лишь небольшую часть данных ETW, чтобы продемонстрировать различие в результатах измерений. Анализ всего набора данных ETW тоже не показал бы ничего настораживающего. Но надо предпринять дальнейшие шаги для измерения накладных расходов (см. сценарий 7.1 в главе 7).

Запомненные наборы (Remembered sets)

Мы теперь знаем, что объекты в куче SOH разделены на поколения, и каждое поколение можно обрабатывать по-разному. В частности, это означает, что есть возможность запускать сборку мусора для каждого поколения в отдельности. Можно было бы собирать только объекты в «молодом» поколении. Или только в «старом» поколении. Однако это чересчур упрощенный взгляд на вещи.

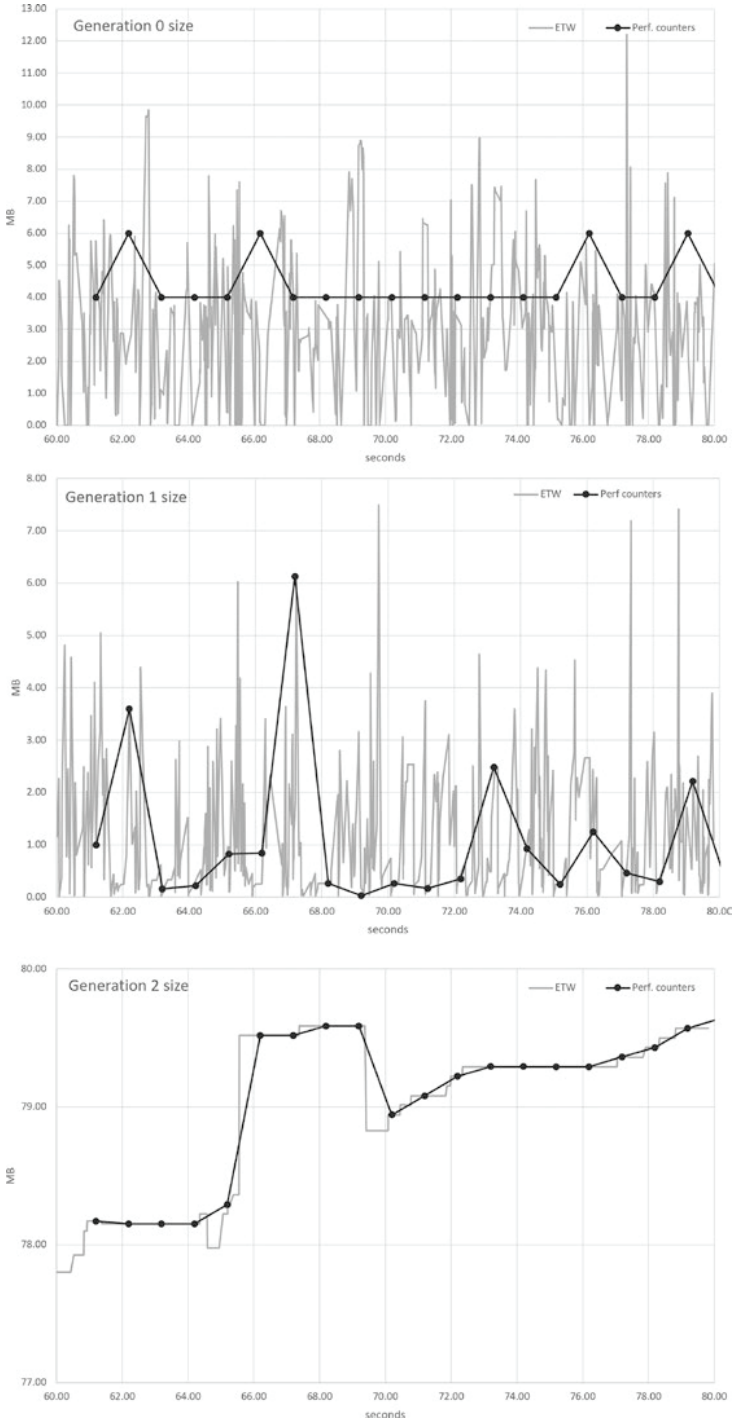


Рис. 5.7 ❖ Графики размеров поколений, построенные по данным ETW и счетчиков производительности, экспортированным в формате CSV

Вспомним, что в общем механизме сборки мусора, описанном в главе 1, есть этап пометки, используемый сборщиком. На этом этапе определяется достижимость объектов, для чего производится обход графа объектов, начиная с корней. Во время обхода GC проходит по ссылкам, исходящим из посещенных объектов. Это прекрасно работает, если обходится весь граф объектов, созданных в приложении. Но что, если требуется собрать только их часть, например лишь «молодое» поколение? Рассмотрим ситуацию, изображенную на рис. 5.8. Здесь показано состояние сборщика мусора с тремя поколениями в некоторый момент времени:

- поколение 0 содержит объекты A, B, C, D. На A есть прямая ссылка из корня (скорее всего, она хранится в локальной переменной, находящейся в стеке), и в нем есть поля, ссылающиеся на объект B. На объект C имеется только ссылка из более старого поколения. На объект D вообще нет ссылок (т. е. он действительно недостижим);
- поколение 1 содержит объекты E, F, G. На E есть прямая ссылка из корня, а в нем самом есть поле, ссылающееся на объект C (принадлежащий молодому поколению). На объект F нет ссылок (это еще один недостижимый объект). На объект G имеется ссылка из объекта D, принадлежащего молодому поколению;
- в поколении 2 нет ни одного объекта, чтобы не загромождать объяснение. Но механизм не меняется от того, что понимать под «более старым» поколением – поколение 1 или 2.

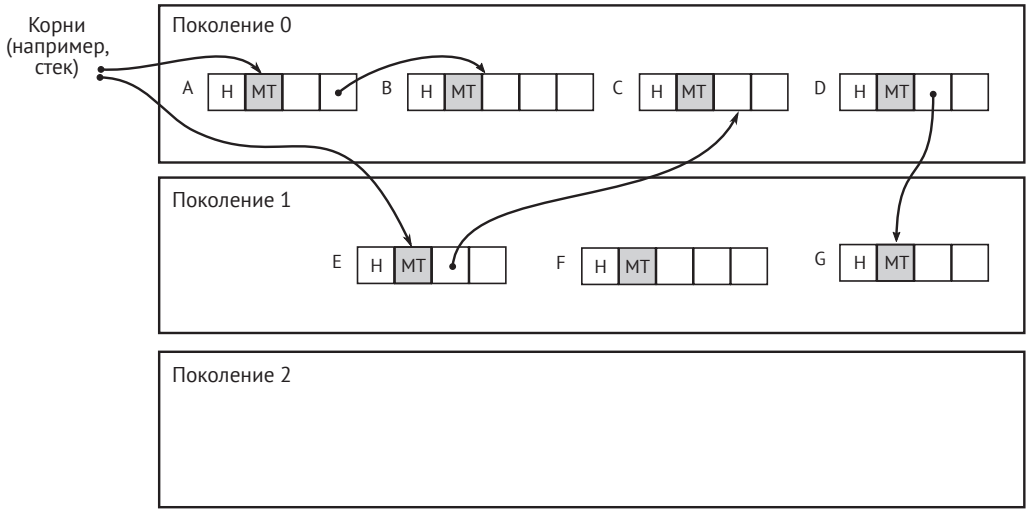


Рис. 5.8 ❖ Ссылки между поколениями

На рис. 5.8 показаны самые типичные ссылки в приложении. Ссылки между поколениями – вовсе не аномалия.

- Ссылки из молодого поколения на старое – недавно созданный объект может содержать ссылку на объект, который существует уже долго (объекты D и G).
- Ссылки из старого поколения на молодое – в объект, созданный давно, может быть записана ссылка на вновь созданный объект (объекты E и C).

Такие *межпоколенческие ссылки* следует как-то обрабатывать на этапе пометки. Конечно, можно было бы обойти весь граф объектов, чтобы определить, какие из объектов A, B, C, D, E, F, G достижимы. Но это, очевидно, сводит на нет всю идею поколений. Раз так, то рассмотрим наивный подход – помечать только «молодое» поколение, т. е. обходить лишь принадлежащие ему объекты. Точнее, мы будем начинать с корней и продолжать обход, пока не встретится объект из какого-то другого поколения, кроме молодого. Понятно, что при этом получаются неверные результаты.

Начиная с корней, мы пометим как достижимые только объекты A и B. Объект E, хотя он и корневой, будет проигнорирован, т. к. находится в «старом» поколении. Мы не посетим объект C, поскольку на него не ссылается никакой корень или другой «молодой» объект. Мы просто не заметим, что на объект C ведет ссылка из E. В результате объекты C и D будут считаться недостижимыми. Объект D и в самом деле недостижим, его можно удалить. Но объект C падет жертвой сборщика мусора, хотя все еще используется объектом E – мы же этого не заметили! Таким образом, очевидно, что ссылки из старого поколения на молодое нужно как-то обрабатывать. Мы должны включать их в рассмотрение достижимости объектов молодого поколения, даже если хотим произвести сборку мусора только в молодом поколении.

Для этого применяется метод *запомненных наборов*. Вообще говоря, запомненный набор – это отдельно управляемая коллекция ссылок между разными множествами объектов. В нашем случае в одном наборе запоминаются ссылки из более старого поколения на более молодое. Затем они проверяются на этапе пометки.

В нашем примере сборки мусора в молодом поколении мы будем обходить объекты, начиная с корней и со ссылок, хранящихся в запомненном наборе, и в том числе увидим ссылку из E на C. Тогда результат будет таким, как нужно.

Заметим, что наличие ссылок из молодого поколения на старое может доставлять проблемы, только если производится сборка мусора в одном лишь старом поколении (без сборки в более молодых поколениях). С другой стороны, если в нашем примере мы производим только сборку в молодом поколении, то можем убрать объект D, даже если он на что-то указывает. Мы просто временно оставим объект G осиротевшим. Позже при выполнении сборки мусора в более старом поколении он будет помечен как недостижимый. Так что в конечном итоге оба объекта D и G будут убраны.

Но при попытке выполнить сборку мусора только в старом поколении мы столкнемся с той же проблемой. Мы не заметим, что на G имеется ссылка из D. Поэтому придется создать еще один запомненный набор для хранения ссылок из молодого поколения на старое. Как мы увидим, реализация запомненных наборов – дело не тривиальное, поэтому было принято более простое решение. В документации Майкрософт написано: «Сборка мусора в некотором поколении означает также сборку мусора во всех более молодых поколениях». И это приводит нас к одному из самых важных положений, касающихся управления памятью в .NET. Сборка мусора в .NET может производиться:

- только для поколения 0;
- для поколений 0 и 1;
- для всех поколений 0, 1, 2 и кучи больших объектов (полная сборка мусора).

Но как обслуживается запомненный набор? Когда в него добавляются ссылки, а когда удаляются? Общее решение – запоминать ссылку, когда она создается, а происходит это в основном в момент присваивания полю (листинг 5.3). Это действие может активироваться прямо (если поле не закрытое) или косвенно – путем присваивания свойству, в конструкторе или при вызове метода.

Листинг 5.3 ❖ Присваивание открытому полю как пример создания ссылки из старого поколения на молодое (предполагается, что объект `e` находится в более старом поколении, чем объект `c`)

```
E e = new E();
...
C c = new C();
e.SomeField = c;
```

Последняя строка листинга 5.3 – самое подходящее место для добавления вновь созданной ссылки в запомненный набор. Однако следует взглянуть на проблему с более общей точки зрения. Поля в смысле C# – лишь один из возможных способов хранения ссылок, вытекающий из спецификации C#. Но мы не должны связывать механизм запомненных наборов с одним конкретным языком. В будущем могут появиться и другие способы хранения ссылок – в C# или каком-то новом, еще не существующем языке.

Поэтому для реализации этого механизма следует прибегнуть к механизмам среды выполнения более низкого уровня – понятию барьера записи, упомянутому в главе 1. Мы можем добавить код барьера записи в операцию `Mutator.Write` (см. листинг 1.7). Эта операция выполняется модификатором всякий раз, как мы хотим сохранить какое-то значение по указанному адресу. Понятно, что выполняется она очень часто, поэтому добавление в нее чего-либо может привести к гигантским издержкам. Поэтому при проектировании барьера записи следует быть крайне осторожным. Хорошо, что дополнять операцию `Write` барьером записи нужно только при выполнении определенных условий (описывающих сохранение ссылки):

- значение является ссылкой на управляемый объект;
- адрес находится в управляемой куче и представляет некоторое допустимое поле объекта;
- адрес принадлежит поколению, более старому, чем поколение, содержащее объект, на который ссылается значение.

Получается схематическая реализация, показанная в листинге 5.4, которая проверяет эти условия и запоминает ссылку, если это необходимо. На этапе пометки мы должны будем рассматривать ссылки, включенные в `RememberedSet`, наряду с другими корнями.

Листинг 5.4 ❖ Очень простой схематичный псевдокод барьера записи с поддержкой запомненных наборов

```
Mutator.Write(address, value)
{
   *address = value;
   if (AreWriteBarrierConditionMeet(address, value))
   {
```

```
            RememberedSet.AddOrUpdate(address, value);
        }
    }
```

Эта общая идея иллюстрирует возможную реализацию в среде выполнения .NET. Очевидно, что проверка всех условий при каждой операции привнесла бы огромные накладные расходы. Но, подумав, мы можем заметить много возможностей для оптимизации. Большая их часть связана с тем, что эти условия можно проверить заранее на этапе JIT-компиляции. JIT-компилятор точно знает из IL-кода, собираемся ли мы сохранить ссылку на управляемый объект в поле другого управляемого объекта. Генерируя ассемблерный код, компилятор может включить подходящую версию `Mutator.Write` в зависимости от того, нужен барьер записи или нет. Именно такой подход и принят в .NET.

> Если вас интересуют дополнительные детали, можете заглянуть в код метода `CodeGen::genCodeForTreeNode` (в CoreCLR) для значения параметра `GT_STOREIND`. Он вызывает метод `CodeGen::genCodeForStoreInd`, который решает (посредством обращения к `gcIsWriteBarrierCandidate`), требуется барьер записи или нет. Если требуется, то вызывается метод `CodeGen::genGCWriteBarrier`. Этот метод генерирует ассемблерный код одной из двух вспомогательных функций: `CORINFO_HELP_ASSIGN_REF` или `CORINFO_HELP_CHECKED_ASSIGN_REF` (вторая используется, если JIT-компилятор знает, что может не проверять нахождение целевого адреса в управляемой куче, первая – в противном случае). Эти две вспомогательные функции – не что иное, как ассемблерные функции `JIT_WriteBarrier` и `JIT_CheckedWriteBarrier`, код которых находится в файле `.\src\vm\amd64\JitHelpers_Fast.asm`. Имейте в виду, что все это происходит на этапе JIT-компиляции, а во время выполнения просто вызывается одна из функций – `JIT_WriteBarrier` или `JIT_CheckedWriteBarrier`. Также имейте в виду, что все сказанное относится только к случаю среды выполнения на платформе x64. В случае платформы x86 обработка барьеров записи концептуально похожа, но путь выполнения иной, и для краткости он здесь не описан.

Рассмотрим подробнее, как можно увидеть барьер записи в наших .NET-приложениях. Начнем с очень простого кода на C#, приведенного в листинге 5.5. Он создает два объекта и присваивает второй полю первого.

Листинг 5.5 ❖ Код для иллюстрации барьеров записи в .NET

```
ClassA someClass = new ClassA();
ClassB otherClass = new ClassB();
someClass.FieldB = otherClass;
```

Этот код компилируется в CIL-код на рис. 5.6 (он немного упрощен, но все важные детали сохранены). Мы видим, как создаются объекты типов `ClassA` и `ClassB`. Оба экземпляра находятся в стеке вычислений. Затем выполняется команда `stfld`, которая сохраняет значение на вершине стека вычислений в поле объекта (второго значения в стеке).

Листинг 5.6 ❖ Результат компиляции кода из листинга 5.5 в CIL

```
newobj CoreCLR.WriteBarrier.ClassA::.ctor
newobj CoreCLR.WriteBarrier.ClassB::.ctor
stfld CoreCLR.WriteBarrier.ClassA::FieldB
```

При выполнении JIT-компиляции этот код может транслироваться в ассемблерный код в листинге 5.7. Мы не можем с уверенностью утверждать, что результат будет выглядеть именно так, потому что опустились на очень низкий уровень реализации. Точный вид кода зависит от многих факторов, включая версию среды выполнения. Но общего вида достаточно для иллюстрации обсуждаемого вопроса. Как видим, команда `stfld` транслируется в вызов функции `JIT_WriteBarrier` (версия с проверкой не используется, потому что JIT-компилятор знает, что работает с управляемым объектом).

Листинг 5.7 ❖ Результат компиляции CIL-кода из листинга 5.6 в ассемблерный код на платформе x64

```
; Эти строки соответствуют выделению памяти для объекта ClassA и вызову
; его конструктора
mov   rcx,7FFCC4BA6600h (MT: CoreCLR.WriteBarrier.ClassA)
call  CoreCLR!JIT_TrialAllocSFastMP_InlineGetThread (00007ffd`241d2130)
mov   rdi,rax ; rdi содержит ссылку на ClassA
mov   rcx,rdi
call System_Private_CoreLib+0xc04060 (00007ffd`22e44060) (System.
Object..ctor(), mdToken: 0000000006000103)

; Эти строки соответствуют выделению памяти для объекта ClassB и вызову
; его конструктора
mov   rcx,7FFCC4BA67B8h (MT: CoreCLR.WriteBarrier.ClassB)
call  CoreCLR!JIT_TrialAllocSFastMP_InlineGetThread (00007ffd`241d2130)
mov   rsi,rax ; rsi содержит ссылку на ClassB
mov   rcx,rsi
call System_Private_CoreLib+0xc04060 (00007ffd`22e44060) (System.
Object..ctor(), mdToken: 0000000006000103)

; В этих строках вызывается WriteBarrier, которая сохраняет ссылку
; в запомненном наборе
lea   rcx,[rdi+8] ; rcx содержит адрес поля FieldB объекта ClassA
mov   rdx,rsi ; rdx содержит ссылку на ClassB
call  CoreCLR!JIT_WriteBarrier (00007ffd`2403fae0)
```

Мы заглянем в код функции `JIT_WriteBarrier`, но сначала должны узнать еще об одной важной технике, которая называется *таблицей карт* (card table).

Таблицы карт (Card tables)

Возможно, вы обратили внимание на серьезный недостаток идеи хранить каждую ссылку в запомненном наборе. В примере на рис. 5.8 запомненный набор мал (содержит всего-то одну ссылку). Но как быть с реальными приложениями, где имеются сотни, тысячи и даже миллионы ссылающихся друг на друга объектов? Хуже того, в .NET имеется три поколения, поэтому количество возможных межпоколенческих ссылок еще больше. К тому же изменение ссылок между объектами – вполне обычная операция. Поэтому наивное управление запомненным набором как коллекцией всех без исключения межпоколенческих ссылок привело бы к слишком большим затратам.

Как часто бывает, для решения проблемы следует пойти на компромисс. Чтобы уменьшить издержки управления коллекцией, мы не отслеживаем каждую

ссылку по отдельности, соглашаясь на потерю точности. Вместо этого отслеживаются предопределенные области памяти, для чего применяется техника *таблиц карт*.

Чтобы объяснить ее, вернемся к моменту, предшествующему тому, что изображен на рис. 5.8 (см. рис. 5.9a). Здесь показан момент, перед тем как в объект E записана межпоколенческая ссылка на объект C. Идея таблицы карт очень проста – мы разбиваем старшее поколение на участки одинакового размера (непрерывные участки, содержащие заданное число байтов). В примере на рис. 5.9a показано четыре таких участка и часть пятого. В первом участке нет ни одного объекта, во втором – только один объект, в третьем – часть объекта (так получилось, что объект располагается на границе двух участков). Четвертый участок содержит оставшуюся часть того же объекта и часть другого. И так далее.

Каждый участок представлен одной *картой* в таблице карт. Вначале все карты *чистые*, т. е. флаг в записи, соответствующей карте, установлен в положение «чистая» (например, некоторый бит сброшен в 0). Это означает, что в соответствующем участке памяти нет ни одной ссылки из старшего поколения на младшее.

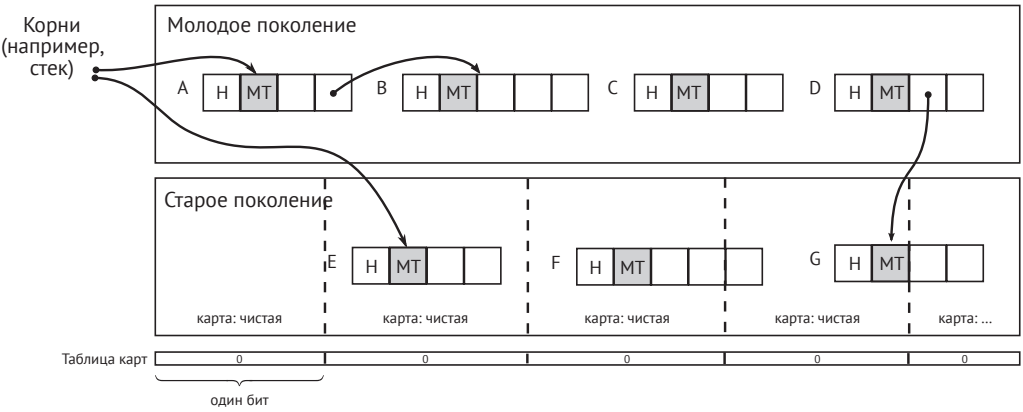


Рис. 5.9a ❖ Таблица карт управляет ссылками из старшего поколения на младшее. Показан момент, предшествующий изображенному на рис. 5.8. Все карты чистые (межпоколенческих ссылок еще нет)

Когда приложение запишет объект C в поле объекта E, мы окажемся в ситуации, изображенной на рис. 5.9b. Мы вычисляем, в какой карте находится объект E, и помечаем ее целиком флагом «грязная», для чего достаточно установить один бит в 1. Такая карта называется *помеченной* (set card).

Начиная с этого момента все объекты в помеченной карте считаются потенциальными дополнительными корнями. Иными словами, когда наступит момент для сборки мусора в молодом поколении, мы будем обходить граф объектов, начиная с корней и всех объектов в помеченных картах (и тогда выяснится, что объект C достижим, потому что E принадлежит помеченной карте).

> Внимательный читатель может спросить, что произойдет, если мы изменим последнее поле объекта F, которое находится в четвертой карте, хотя сам объект F начинает-

ся в третьей? Какую карту нужно пометить в этом случае? Поскольку барьер записи должен быть максимально простым, помечена будет четвертая карта, т. к. именно она соответствует измененному адресу. Позже, на этапе пометки, система найдет, где начинается объект (в нашем случае F), применяя технику таблицы кирпичей, описанную в главе 9.

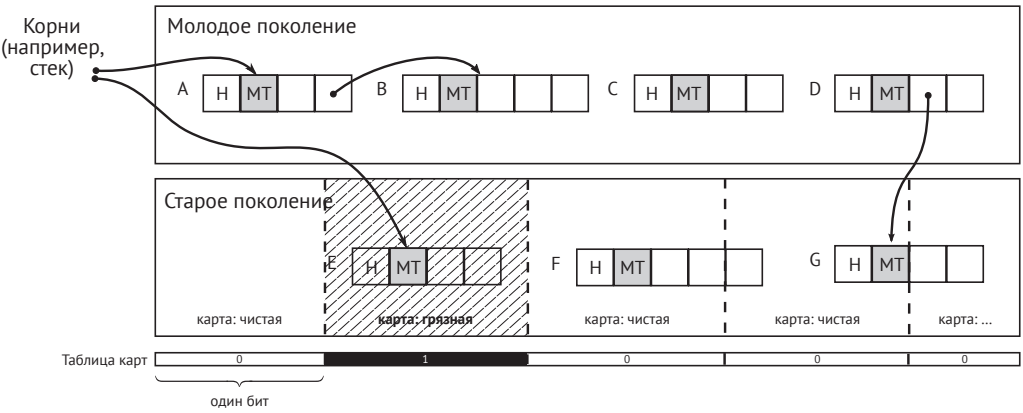


Рис. 5.9b ❖ Таблица карт управляет ссылками из старшего поколения на младшее. После присваивания объекта C полю объекта E соответствующая карта в таблице помечена как «грязная»

Очевидно, что при таком подходе неизбежны накладные расходы. Из-за единственной межпоколенческой ссылки мы должны будем посетить все объекты, находящиеся в соответствующей карте, и проследовать по ссылкам, ведущим из них. Но это компромисс между производительностью и точностью. Его можно настроить, выбрав размер карты. Если карта содержит не более одного объекта, то мы возвращаемся к наивному способу обработки запомненного набора, когда отслеживается каждая ссылка. Если же карта настолько велика, что содержит все поколение, то мы будем обходить граф объектов целиком.

В среде выполнения .NET размер карты равен 256 байт на 64-разрядной машине и 128 байт на 32-разрядной. Каждая карта представлена одним битовым флагом. Если в какую-то часть карты записывается межпоколенческая ссылка, этот флаг поднимается. Биты, естественно, группируются в байты, т. е. один байт представляет область памяти размером 8×256 (2048) байт. Карты собираются в группы по 32 элемента, которые называются *карточным словом* (card word). Следовательно, длина одного карточного слова равна 4 байтам, т. е. слово имеет тип DWORD (unsigned long) и представляет 8192 байта памяти. Это показано на рис. 5.10 (для 64-разрядной платформы).

Вооруженные этими знаниями, мы можем перейти к вышеупомянутой функции `JIT_WriteBarrier`. Интересно, что участок памяти, отведенный под функцию `JIT_WriteBarrier`, рассматривается как область для размещения одной из ее конкретных реализаций. Барьеры могут меняться во время выполнения, для чего нужно заменить одну реализацию другой (разумеется, это делается, когда выполнение программы приостановлено). Размер этой области равен размеру самой длинной реализации функции, так чтобы туда поместилась любая реализация.

Мы рассмотрим самую простую версию (листинг 5.8), но они мало чем отличаются, так что знакомства с одной будет вполне достаточно (ниже изложены дополнительные детали).

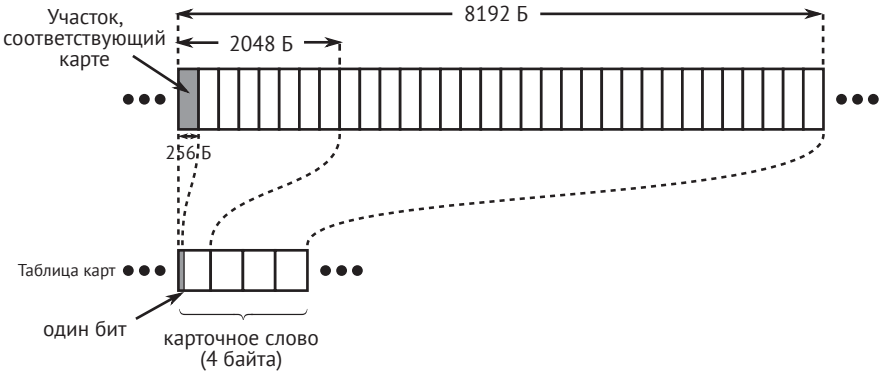


Рис. 5.10 ❖ Организация таблицы карт в среде выполнения .NET (64-разрядная версия). Каждый бит в таблице карт представляет 256 байт памяти. Эти биты сгруппированы в байты (так что каждый байт представляет участок длиной 2048 байт), а байты объединены в карточные слова, представляющие участок в 4 раза больше

Различные реализации `JIT_WriteBarrier` можно найти в файле `.\src\vm\amd64\JitHelpers_FastWriteBarriers.asm` в исходном коде CoreCLR (для процессоров семейства amd64). Там имеются следующие версии:

- `JIT_WriteBarrier_PreGrow64` и `JIT_WriteBarrier_PostGrow64` – используются в режиме GC для рабочей станции. Первая применяется, когда поколения 0 и 1 находятся по адресам, подразумеваемым по умолчанию. По истечении некоторого времени среда выполнения может переместить их в другое место, и тогда будет использоваться вторая версия;
- `JIT_WriteBarrier_SVR64` – используется в серверном режиме GC, когда имеется несколько куч и, стало быть, несколько поколений 0 и 1. В этом случае проверка принадлежности значения одному их них была бы слишком медленной, поэтому карта помечается безусловно;
- `JIT_WriteBarrier_WriteWatch_PreGrow64`, `JIT_WriteBarrier_WriteWatch_PostGrow64` и `JIT_WriteBarrier_WriteWatch_SVR64` – версии описанных выше функций с использованием реализованной в CLR техники наблюдения за записью, которую мы скоро опишем (если эта техника не реализована в ОС).

Когда среда выполнения решает изменить барьер записи, она вызывает следующий метод:

```
int WriteBarrierManager::ChangeWriteBarrierTo(WriteBarrierType newWriteBarrier,
bool isRuntimeSuspended)
{
  ...
  memcpy((PVOID)JIT_WriteBarrier,
     (LPVOID)GetCurrentWriteBarrierCode(), GetCurrentWriteBarrierSize());
  ...
}
```

Детали смотрите в исходном коде методов `StompWriteBarrierResize` и `StompWriteBarrierEphemeral` в файле `.\src\vm\amd64\JITInterfaceAMD64.cpp`.

Как следует из листинга 5.8, код барьера записи на самом деле очень простой.

- Аргумент, передаваемый в регистре `rcx`, содержит конечный адрес (`address` в нашем примере `Mutator.Write`), а регистр `rdx` содержит ссылку на источник (`value` в нашем `Mutator.Write`).
- В строке 3 производится основная работа – запись указанного значения в память по указанному адресу. Мы хотим манипулировать таблицей карт (пометить карту) только в случае, когда адрес в `rdx` действительно принадлежит молодому поколению, поскольку среде выполнения интересны только ссылки из старшего поколения на младшее (при этом если старшим является поколение 2, то оба поколения – 0 и 1 – считаются младшими).
- Поэтому в строках с 6 по 14 проверяется, верно ли, что ссылка на источник принадлежит так называемой *эфемерной области* (т. е. обоим поколениям – 0 и 1). Если нет, то работа функции заканчивается. Если да, то проверяется таблица карт, если она еще не помечена. Для нас эти строки представляют наибольший интерес.
- В строке 16 адрес таблицы карт записывается в регистр `rax` (странная константа `0F0F0F0F0F0F0F0Fh` будет заменена во время выполнения правильным значением).
- В строке 17 конечный адрес (находящийся в `rcx`) делится на 2048[1].
- В строках с 18 по 22 байт в таблице карт сравнивается со значением `FFh`, и если он еще не установлен, то устанавливается.

Листинг 5.8 ❖ Реализация функции JIT_WriteBarrier_PostGrow64. Исходные комментарии заменены другими

```
01. LEAF_ENTRY JIT_WriteBarrier_PostGrow64, _TEXT
02.      align 8
03.      mov [rcx], rdx ; сохранить значение из регистра rdx по адресу,
                        ; в регистре rcx
04.      NOP_3_BYTE     ; заполнение для выравнивания константы
05. PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_Lower
06.      mov rax, 0F0F0F0F0F0F0F0Fh ; 0F0F0F0F0F0F0F0Fh будет заменена
                        ; во время выполнения правильным адресом
07.      cmp rdx, rax   ; сравнить с нижней границей эфемерной области
                        ; (если rdx < rax, перейти к Exit)
08.      jb Exit
09.      nop            ; заполнение для выравнивания константы
10. PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_Upper
11.      mov r8, 0F0F0F0F0F0F0F0Fh ; 0F0F0F0F0F0F0F0Fh будет заменена
                        ; во время выполнения правильным адресом
12.      cmp rdx, r8    ; сравнить с нижней границей эфемерной области
                        ; (если rdx >= r8, перейти к Exit)
13.      jae Exit
14.      nop            ; заполнение для выравнивания константы
15. PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_CardTable
16.      mov rax, 0F0F0F0F0F0F0F0Fh ; 0F0F0F0F0F0F0F0Fh будет заменена
                        ; во время выполнения правильным адресом таблицы карт
```

[1] Команда `shr rcx, 0Bh` сдвигает значение в регистре `rcx` на `0Bh` (т. е. на 11) бит вправо. Сдвиг на *n* бит вправо эквивалентен делению на 2^*n*. 2^11 равно 2048.

```
17.      shr rcx, 0Bh   ; пометить запись в таблице карт, если она еще
                        ; не помечена
18.      cmp byte ptr [rcx + rax], 0FFh
19.      jne UpdateCardTable
20. REPRET
21.   UpdateCardTable:
22.      mov byte ptr [rcx + rax], 0FFh
23.      ret
24.      align 16
25.   Exit:
26.      REPRET
27. LEAF_END_MARKED JIT_WriteBarrier_PostGrow64, _TEXT
```

Нам особенно важно, что устанавливается весь байт, представляющий восемь карт, хотя можно было бы установить в нем всего один бит. Это сделано из соображений производительности. Гораздо эффективнее сравнивать и сохранять целый байт (для чего, как мы видим, хватает одной команды), чем заниматься манипулированием битами (для этого нужно было бы подготовить битовые маски и работать с ними).

Конечно, без дополнительных накладных расходов тут не обходится. Вместо пометки всего одной карты (участок памяти размером 256 байт) мы помечаем сразу 8 карт, т. е. участок длиной 2048 байт. Это еще один пример компромисса, принятого в процессе проектирования.

> Имейте в виду, что текущие реализации барьера записи, в т. ч. и пример в листинге 5.8, проверяют только, принадлежит ли ссылка на источник молодому поколению. Не проверяется, что ссылка действительно исходит из адреса, принадлежащего старому поколению. Поэтому карта будет помечена как грязная также и для ссылок из молодого поколения на молодое. Но в этом нет ничего страшного, потому что:
>
> – на этапе пометки таблица карт проверяется только для адресов, принадлежащих старшим поколениям, а значит, ссылки из молодого поколения на него же попросту игнорируются;
>
> – проверять во время выполнения барьера записи, принадлежит ли адрес в `rcx` старшему поколению, было бы слишком долго. Быстрее пометить карту, чем производить все необходимые проверки.

Связки карт

Техника *таблиц карт* позволяет оптимизировать использование запомненных наборов. Вместо того чтобы отслеживать каждую межпоколенческую ссылку по отдельности, мы отслеживаем группы ссылок. Как мы видели, в случае 64-разрядного .NET Framework ведется наблюдение за участками памяти длиной 256 байт, покрываемыми одной картой. Если хотя бы один объект внутри такого блока был модифицирован и теперь содержит ссылку на молодое поколение, то весь блок считается «грязным», о чем свидетельствует установленный бит в таблице. И даже более того, в целях оптимизации помечается не один бит, а целый байт, которому соответствует участок памяти длиной 2048 байт. Но есть и еще одна возможность для оптимизации.

Рассмотрим типичное веб-приложение, работающее на сервере. Оно может потреблять несколько гигабайтов памяти. Предположим, что размер старого поко-

ления равен 2 ГБ. Каждый байт в таблице карт представляет 2 КБ. Поэтому, чтобы покрыть все старое поколение, нужна таблица карт размером 1 МБ. На первый взгляд, не так много. Но эти байты придется просматривать при каждой сборке в младших поколениях (чтобы найти все ссылки из старшего поколения на младшее). Сборка в младших поколениях должна завершаться очень быстро, поэтому просмотр такой большой таблицы карт следует рассматривать как чрезмерные затраты, пусть даже он занимает всего несколько миллисекунд. За это время нужно завершить весь процесс сборки мусора, а не только просмотр таблицы карт. Ко всему прочему, таблица карт часто сильно разрежена – имеется много непомеченных карт, а между ними изредка встречаются помеченные.

По этой причине добавлен еще один уровень наблюдения – *связки карт* (card bundle). Если в одном карточном слове сгруппировано несколько карт, то в одном слове связки карт – несколько карточных слов. Они покрывают гораздо большие области памяти (рис. 5.11). Один бит в слове связки карт представляет 32 карточных слова (т. е. покрывает область размером 256 КБ). Таким образом, каждый байт представляет область размером 2 МБ, а все четырехбайтовое слово связки карт – область размером 8 МБ.

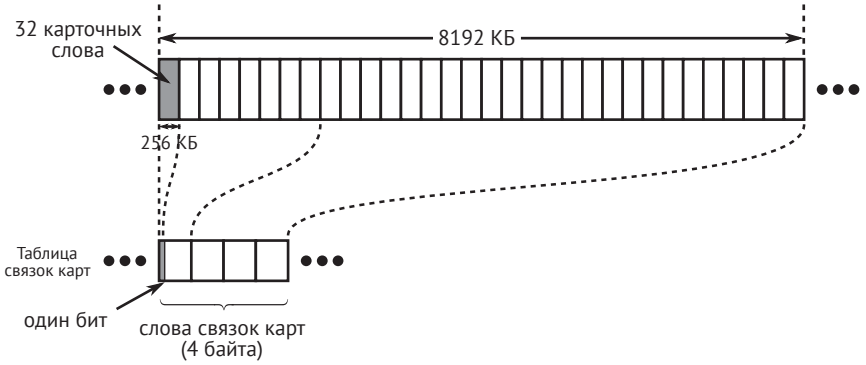


Рис. 5.11 ❖ Организация таблицы связок карт в среде выполнения .NET (64-разрядная версия). Каждый бит в таблице связок карт представляет 32 карточных слова (256 КБ). Эти биты сгруппированы в байты (так что каждый байт представляет участок длиной 2048 КБ), а байты объединены в слова связок карт, представляющие участок в 4 раза больше (8 МБ)

Такая организация допускает очень быстрый (быть может, даже кешированный) просмотр помеченных карт. Сначала в таблице связок карт ищутся большие «грязные» области, и только в них производится полный просмотр таблицы карт. В нашем примере, когда старое поколение занимает 2 ГБ, для его представления необходимо только 1024 байта в таблице связок карт. Если в нем найдется установленный бит, то будет выполнен просмотр 32 карточных слов в таблице карт для поиска помеченных карт.

Но когда именно связка карт помечается как «грязная»? В коде барьера записи мы ничего такого не видели. Механизм зависит от операционной системы.

В Windows используется механизм наблюдения за записью, упомянутый в главе 1. Когда с помощью Virtual API резервируются страницы для области, занятой

таблицей карт, в них поднимается флаг MEM_WRITE_WATCH. В таком случае при любой последующей модификации (вследствие того, что барьер записи пометил какую-то карту) страница помечается как измененная в специальной структуре Windows. После этого мы можем запросить список измененных страниц с помощью функции GetWriteWatch, являющейся частью WinAPI. Среда выполнения .NET вызывает эту функцию в начале этапа пометки – из метода gc_heap::update_card_table_bundle(). Этот метод получает от системы список всех измененных страниц и устанавливает соответствующие биты в таблице связок карт.

В Linux команда разработки .NET Core не смогла найти эквивалентного механизма в операционной системе. Но преимущества верхнего уровня управления картами настолько ощутимы, что было решено реализовать замену ему самостоятельно. Поэтому в Linux механизм наблюдения за записью реализован в барьере записи. Соответствующий код находится в файле .\src\amd64\jithelpers_fastwritebarriers.S (в листинге 5.9 показана существенная часть одной из функций).

Листинг 5.9 ❖ Часть ассемблерного кода барьера записи для среды выполнения .NET в Linux. Показана реализация механизма наблюдения за записью, необходимого для управления связками карт

```
#ifdef FEATURE_MANUALLY_MANAGED_CARD_BUNDLES
       NOP_6_BYTE // заполнение для выравнивания константы
PATCH_LABEL JIT_WriteBarrier_PreGrow64_Patch_Label_CardBundleTable
      movabs rax, 0xF0F0F0F0F0F0F0F0
      // пометить связку карт, если она еще не грязная
      // rdi уже сдвинут на 0xB, сдвинем еще на 0xA
      shr rdi, 0x0A
      cmp byte ptr [rdi + rax], 0FFh
      .byte 0x75, 0x02
      // jne UpdateCardBundle_PreGrow64
      REPRET
  Upd ateCardBundle_PreGrow64:
     mov byte ptr [rdi + rax], 0FFh
#endif
```

Как видим, здесь весь байт тоже помечается как «грязный», поэтому в .NET Core для Linux таблицы карт работают с гранулярностью 2 МБ.

> Есть еще один интересный вопрос – таблицы карт и обработка массивов. Представьте себе большой массив объектов, находящийся в старом поколении. Он настолько велик, что захватывает много карт и даже их связок. Пусть теперь мы присваиваем одному из элементов массива вновь созданный объект. Что будет? Лишь один байт в карточном слове будет сделан «грязным», равно как соответствующий ему бит в слове связок карт. Но как впоследствии этой информацией воспользуется процесс пометки? Какие элементы массива он будет просматривать? Только те, что соответствуют карте, или, быть может, весь массив? Ответ простой – просматриваются лишь части массива, соответствующие помеченным картам.

Мы многое узнали о запомненных наборах, таблицах и связках карт в среде выполнения .NET. Этой теме уделено так много места, потому что это один из ключевых механизмов, обеспечивающих сборку мусора в .NET. С другой стороны, в литературе он обсуждается не очень подробно, быть может, потому что это глубоко

скрытая деталь реализации. Он прекрасно оптимизирован, поэтому не вызывает никаких проблем и не обязателен для общего понимания. Однако я полагаю, что нет лучшего места для объяснения этого вопроса, чем в книге по управлению памятью в .NET. Полученные знания позволят нам понять правило, сформулированное в конце этой главы: «Избегайте ссылок в куче, без которых можно обойтись».

Физическое разделение

Мы уже знаем, что управляемая память разделена на две области. В куче больших объектов находятся объекты, размер которых превышает 85 000 байт (и еще коечто). В куче малых объектов находятся объекты меньшего размера, она дополнительно разделена на поколения. Мы знаем, что все это обитает в области памяти, которая является кучей с точки зрения операционной системы (как показано на рис. 5.1 в начале этой главы). Чего мы еще не знаем, так это как точно организована управляемая куча GC, содержащая LOH и SOH со всеми ее поколениями. Сейчас мы рассмотрим вопрос о физической организации кучи GC и соберем воедино все, что узнали до сих пор.

Физически управляемая куча состоит из множества *сегментов кучи*. Сегмент принадлежит либо LOH, либо SOH. Если сегментов, принадлежащих SOH, несколько, то все они относятся к поколению 2, за исключением одного сегмента, называемого эфемерным, который содержит объекты из поколений 0 и 1 (и, быть может, из поколения 2). Важно отметить, что сборщик мусора от Майкрософт может работать в одном из двух, существенно различных, режимов:

- режим *рабочей станции* – имеется одна управляемая куча (и, значит, одна SOH и одна LOH);
- *серверный* режим – имеется несколько управляемых куч (и, значит, несколько SOH и LOH). По умолчанию их столько, сколько имеется логических ядер на машине, исполняющей .NET-приложение.

Различия между этими режимами мы подробно обсудим в следующих главах. А пока достаточно отметить, что количество управляемых куч разное.

Все эти концепции лучше всего объяснить на примере создания отдельных элементов на стадии запуска среды выполнения .NET. На рис. 5.12 показано три этапа создания управляемой кучи в простейшей ситуации (в режиме рабочей станции). Более сложные сценарии будут описаны ниже.

В этом случае выполняются следующие действия:

- среда выполнения .NET пытается выделить (зарезервировать) один непрерывный блок памяти (рис. 5.12a) для начальных сегментов; это оптимизация, благодаря которой сегменты располагаются рядом. Если такого большого сплошного куска в виртуальном адресном пространстве нет, то сегменты будут располагаться в несмежных областях;
- затем среда должна создать два сегмента для SOH и LOH. Они создаются в только что зарезервированном блоке и логически разделяют его на две части (рис. 5.12b);
- внутри сегмента SOH создаются поколения 0, 1 и 2, каждому из которых передается определенное количество памяти. Сколько-то памяти передается также LOH (рис. 5.12c).

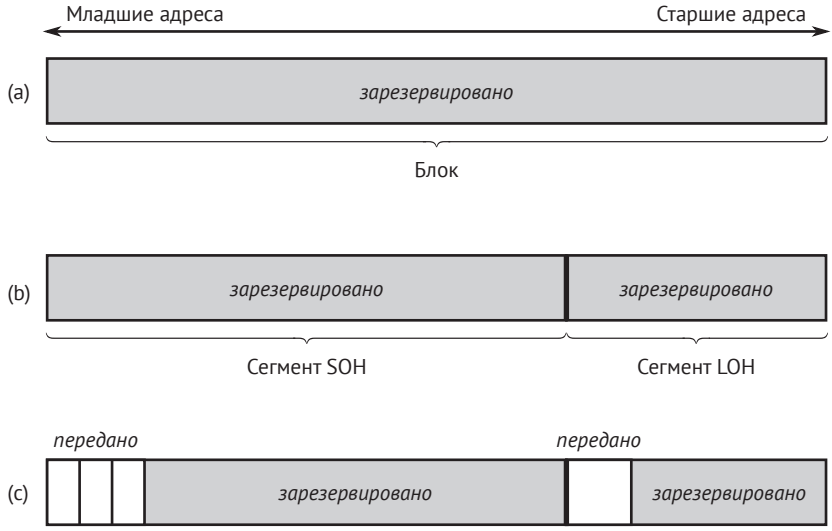


Рис. 5.12 ❖ Блоки и сегменты в простейшей ситуации – один блок содержит сегменты SOH и LOH

> Сегменты представлены в среде выполнения .NET объектами `heap_segment`, которые мы будем подробно изучать в последующих главах. В них запоминается информация об адресах в памяти, о том, сколько памяти зарезервировано и сколько передано, и т. д. Как мы увидим в следующей главе, сегмент кучи потребляется от младших адресов к старшим. Чем больше объектов выделено, тем больше памяти должно быть передано (committed) внутри сегмента.

Ситуацию, изображенную на рис. 5.12, легко наблюдать в реальности для простого консольного приложения с помощью программы VMMap. Если раскрыть блок GC Managed Heap, показанный на рис. 5.1, то мы увидим распределение (рис. 5.13), согласующееся с описанием выше и его иллюстрацией на рис. 5.12c. Видны следующие области памяти:

- примерно 260 КБ выделено для Gen0 (259 КБ), Gen1 (24 байта) и Gen2 (24 байта);
- почти 256 МБ зарезервировано для остатка сегмента SOH;
- 72 КБ передано куче больших объектов;
- почти 128 МБ зарезервировано для остатка сегмента LOH.

⊟ 0000026700000000	Managed Heap	393,216 K	336 K	336 K	224 K	224 K	4 Read/Write	GC
0000026700000000	Managed Heap	4 K	4 K	4 K	4 K	4 K	Read/Write	
0000026700001000	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Gen2
0000026700001018	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Gen1
0000026700001030	Managed Heap	259 K	259 K	259 K	204 K	204 K	Read/Write	Gen0
0000026700042000	Managed Heap	261,880 K					Reserved	
0000026710000000	Managed Heap	72 K	72 K	72 K	16 K	16 K	Read/Write	Large Object Heap
0000026710012000	Managed Heap	131,000 K					Reserved	

Рис. 5.13 ❖ VMMap показывает один блок в консольном .NET-приложении, который содержит два сегмента (SOH и LOH)

Как уже было сказано, сегмент, который содержит поколения 0 и 1, называется *эфемерным*. Это важное отличие, которое проявляется в реализации GC во многих местах. А значит, и мы еще не раз вернемся к нему в этой книге.

Список всех сегментов и поколений можно получить в WinDbg с расширением SOS, выполнив команду eeheap (листинг 5.10). Мы видим информацию о двух сегментах в полном соответствии с картинкой на рис. 5.13. Заметим, что поколение начинается со смещения 0x1000 от начала сегмента. Почему это так, будет объяснено ниже.

Листинг 5.10 ❖ Список сегментов и поколений, выведенный командой eeheap из расширения SOS программы WinDbg. Показано состояние того же процесса, что на рис. 5.13

```
> !eeheap
Number of GC Heaps: 1
generation 0 starts at 0x0000026700001030
generation 1 starts at 0x0000026700001018
generation 2 starts at 0x0000026700001000
ephemeral segment allocation context: none
         segment             begin         allocated             size
0000026700000000  0000026700001000  0000026700033b18  0x32b18(207640)
Large object heap starts at 0x0000026710001000
         segment             begin         allocated           size
0000026710000000  0000026710001000  0000026710005480  0x4480(17536)
Total Size:              Size: 0x36f98 (225176) bytes.
------------------------------
GC Heap Size:            Size: 0x36f98 (225176) bytes.
```

Для сегментов размеры по умолчанию зависят от нескольких факторов. Один из самых важных – режим работы GC. Второй – разрядность среды выполнения. Данные сведены в табл. 5.3. Например, консольное приложение, показанное на рис. 5.9 и 5.10, исполнялось в 64-разрядной среде в режиме рабочей станции. Поэтому сегмент SOH занимал 256 МБ, а сегмент LOH – 128 МБ. Также легко видеть, что в серверном режиме размеры сегментов SOH по умолчанию зависят от количества логических ядер (чем больше ядер, тем меньше сегмент).

Таблица 5.3. Размеры сегментов по умолчанию для различных условий

	Рабочая станция		**Сервер**	
	32-разрядная	**64-разрядная**	**32-разрядная**	**64-разрядная**
SOH	16 МБ	256 МБ	64 МБ (#ЦП<=4) 32 МБ (#ЦП<=8) 16 МБ (#ЦП>8)	4 ГБ (#ЦП<=4) 2 ГБ (#ЦП<=8) 1 ГБ (#ЦП>8)
LOH	16 МБ	128 МБ	32 МБ	256 МБ

На рис. 5.14 показаны сегменты в представлении VMMap для приложения ASP.NET 4.5, запущенного на машине с 8 ядрами и 64-разрядной средой выполнения .NET, работающей в серверном режиме. Как видим, зарезервирован один очень большой непрерывный блок. Он содержит 8 сегментов SOH и вслед за ними 8 сегментов LOH. Размеры сегментов такие, как указано в табл. 5.3 (2 ГБ для SOH и 256 МБ для LOH).

Теперь легко понять, как важно различие между зарезервированной и переданной (committed) памятью, описанное в главе 2. Хотя на первый взгляд кажется, что

управляемая куча в веб-приложении на рис. 5.14 занимает аж 18 ГБ (зарезервированная память), реально используется только около 8 МБ (переданная память).

⊟ 000001A971E50000	Managed Heap	18,874,368 K	8,704 K	8,704 K	8,348 K	8,348 K	32	Read/Write	GC
000001A971E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K		Read/Write	
000001A971E51000	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen2
000001A971E51018	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen1
000001A971E51030	Managed Heap	1,987 K	1,987 K	1,987 K	1,964 K	1,964 K		Read/Write	Gen0
000001A972042000	Managed Heap	2,095,160 K						Reserved	
000001A9F1E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K		Read/Write	
000001A9F1E51000	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen2
000001A9F1E51018	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen1
000001A9F1E51030	Managed Heap	1,155 K	1,155 K	1,155 K	1,124 K	1,124 K		Read/Write	Gen0
000001A9F1F72000	Managed Heap	2,095,992 K						Reserved	
000001AA71E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K		Read/Write	
000001AA71E51000	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen2
000001AA71E51018	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen1
000001AA71E51030	Managed Heap	1,475 K	1,475 K	1,475 K	1,428 K	1,428 K		Read/Write	Gen0
000001AA71FC2000	Managed Heap	2,095,672 K						Reserved	
000001AAF1E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K		Read/Write	
000001AAF1E51000	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen2
000001AAF1E51018	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen1
000001AAF1E51030	Managed Heap	195 K	195 K	195 K	180 K	180 K		Read/Write	Gen0
000001AAF1E82000	Managed Heap	2,096,952 K						Reserved	
000001AB71E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K		Read/Write	
000001AB71E51000	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen2
000001AB71E51018	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen1
000001AB71E51030	Managed Heap	1,027 K	1,027 K	1,027 K	1,012 K	1,012 K		Read/Write	Gen0
000001AB71F52000	Managed Heap	2,096,120 K						Reserved	
000001ABF1E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K		Read/Write	
000001ABF1E51000	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen2
000001ABF1E51018	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen1
000001ABF1E51030	Managed Heap	771 K	771 K	771 K	716 K	716 K		Read/Write	Gen0
000001ABF1F12000	Managed Heap	2,096,376 K						Reserved	
000001AC71E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K		Read/Write	
000001AC71E51000	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen2
000001AC71E51018	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen1
000001AC71E51030	Managed Heap	1,027 K	1,027 K	1,027 K	980 K	980 K		Read/Write	Gen0
000001AC71F52000	Managed Heap	2,096,120 K						Reserved	
000001ACF1E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K		Read/Write	
000001ACF1E51000	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen2
000001ACF1E51018	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen1
000001ACF1E51030	Managed Heap	707 K	707 K	707 K	676 K	676 K		Read/Write	Gen0
000001ACF1F02000	Managed Heap	2,096,440 K						Reserved	
000001AD71E50000	Managed Heap	264 K	264 K	264 K	180 K	180 K		Read/Write	Large Object Heap
000001AD71E92000	Managed Heap	261,880 K						Reserved	
000001AD81E50000	Managed Heap	8 K	8 K	8 K	8 K	8 K		Read/Write	Large Object Heap
000001AD81E52000	Managed Heap	262,136 K						Reserved	
000001AD91E50000	Managed Heap	8 K	8 K	8 K	8 K	8 K		Read/Write	Large Object Heap
000001AD91E52000	Managed Heap	262,136 K						Reserved	
000001ADA1E50000	Managed Heap	8 K	8 K	8 K	8 K	8 K		Read/Write	Large Object Heap
000001ADA1E52000	Managed Heap	262,136 K						Reserved	
000001ADB1E50000	Managed Heap	8 K	8 K	8 K	8 K	8 K		Read/Write	Large Object Heap
000001ADB1E52000	Managed Heap	262,136 K						Reserved	
000001ADC1E50000	Managed Heap	8 K	8 K	8 K	8 K	8 K		Read/Write	Large Object Heap
000001ADC1E52000	Managed Heap	262,136 K						Reserved	
000001ADD1E50000	Managed Heap	8 K	8 K	8 K	8 K	8 K		Read/Write	Large Object Heap
000001ADD1E52000	Managed Heap	262,136 K						Reserved	
000001ADE1E50000	Managed Heap	8 K	8 K	8 K	8 K	8 K		Read/Write	Large Object Heap
000001ADE1E52000	Managed Heap	262,136 K						Reserved	

Рис. 5.14 ❖ Представление VMMap – один гигантский блок в приложении ASP.NET содержит восемь сегментов (SOH и LOH). Приложение исполнялось на машине с 8 логическими ядрами (четыре физических ядра плюс включенный режим гипертрединга) и 64-разрядной средой выполнения, работающей в серверном режиме

Оба показанных выше сценария обладают одним общим свойством – все сегменты созданы внутри одного непрерывного блока. Это наиболее типичная схема начального выделения памяти, которая получила название *все сразу* (показана на рис. 5.15a и 5.16a). Но есть и две другие схемы:

- *двухступенчатая* – имеется два отдельных блока: для сегментов SOH и LOH (рис. 5.15b и 5.16b);
- *каждый блок* – для каждого сегмента выделяется отдельный блок (рис. 5.16c).

Такие схемы могут быть выбраны, если среде выполнения .NET не удалось зарезервировать один непрерывный блок виртуальной памяти. В таком случае сна-

чала она попробует двухступенчатую схему, а если и это не получится, то в серверном режиме будет выбрана схема с выделением блока для каждого сегмента.

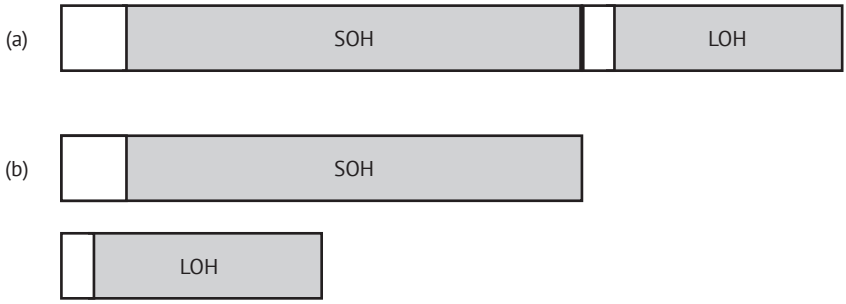


Рис. 5.15 ❖ Возможные конфигурации начальных сегментов в режиме рабочей станции: (a) все сразу; (b) двухступенчатая (совпадает с конфигурацией «каждый блок»)

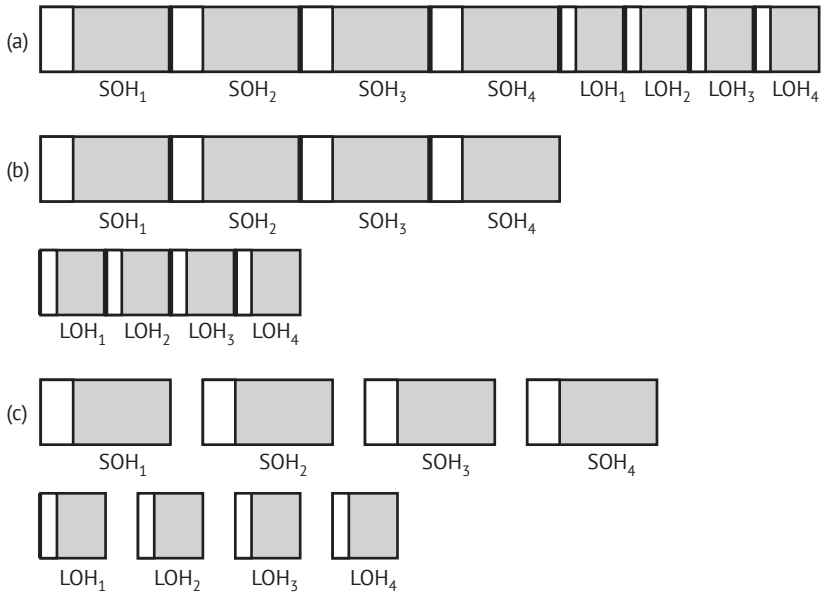


Рис. 5.16 ❖ Возможные конфигурации начальных сегментов в серверном режиме: (a) все сразу; (b) двухступенчатая; (c) каждый блок

Если во время работы приложение создает много объектов, то эфемерный сегмент или LOH может переполниться. В таком случае выделяется дополнительный сегмент. В главе 6 мы увидим, как обрабатываются подобные ситуации. Заметим, что описанные выше конфигурации сегментов одинаковы для версий .NET Core в Windows и Linux.

В Mono (версии 5.4) физическая организация поколений несколько различается:

– небольшие объекты хранятся в областях памяти двух видов. Детская (молодое поколение) занимает непрерывный блок памяти размера 4 МБ. Она не может динамически менять размер, но начальный размер может быть задан в конфигурационном

файле. Для выделения памяти применяется быстрый метод сдвига указателя (bump pointer). Старшее поколение организовано в виде блоков размера 16 КБ (но во избежание фрагментации память выделяется порциями большего размера);

– большие объекты, размещаемые в памяти для больших объектов, организованы секциями по 1 МБ, а объекты еще большего размера – путем прямого обращения к Virtual API, они запоминаются в односвязном списке.

Сегменты бывают трех типов:

- куча малых объектов;
- куча больших объектов;
- постоянная куча (только для чтения).

Третий тип объявлен нерекомендуемым в .NET Framework, начиная с версии 3.5, и в .NET Core. Но в других средах исполнения он все еще может использоваться (в настоящее время только в .NET Native), поэтому упоминается в разных местах, в т. ч. в исходном коде CoreCLR, событиях ETW и документации (в главе 3 мы уже видели такое упоминание в виде значения `ReadOnlyHeapMapMessage` перечисления `CSegmentType`, когда рассматривали данные событий ETW). Сегменты постоянной кучи нужны для реализации функции *замораживания объектов*, для активации которой следует пометить сборку атрибутом `StringFreezingAttribute`.

Когда такая сборка преобразуется в образ в машинном коде с помощью программы Ngen.exe, все строковые литералы помещаются в сгенерированный образ (в управляемой форме) на этапе предкомпиляции. Затем область памяти образа, содержащую такие строки (или вообще любые объекты, хотя для работы с ними не существует никакого API), можно зарегистрировать как постоянный сегмент и сразу же начать использовать (поскольку находящиеся в нем объекты уже приведены к управляемой форме и для них выделена память).

Отметим, что это не то же самое, что интернирование строк (см. главу 4), для которого необходимо обычное выделение памяти для строк во время выполнения. Кроме того, в MSDN написано: «Обратите внимание, что среда CLR не может выгрузить любой машинный образ с замороженной строкой, так как любой объект в куче может ссылаться на замороженную строку. Поэтому следует использовать класс StringFreezingAttribute только в тех случаях, когда машинный образ, содержащий замороженную строку, является общим».

Сценарий 5.2. Утечка памяти в nopCommerce?

Описание. Мы только что скачали дистрибутив nopCommerce – платформы электронной коммерции с открытым исходным кодом, написанной на ASP.NET. В документации по ZIP-файлу сказано: «скачайте этот пакет, если хотите развернуть готовый к работе сайт на веб-сервере, сведя количество необходимых файлов к минимуму». Установка проста: «чтобы использовать IIS, скопируйте содержимое распакованной папки nopCommerce в виртуальный каталог IIS (или в корень сайта)». Мы хотим проверить производительность nopCommerce, в т. ч. характер использования памяти. Мы подготовили простой нагрузочный тест для JMeter 3.2 – популярного средства нагрузочного тестирования с открытым исходным кодом. Он в цикле выполняет три действия: посетить домашнюю страницу, одну из категорий («Computers») и одну из меток («awesome» – потрясающий). Между соседними запросами мы добавили паузы, чтобы смоделировать поведение реального пользователя – время на размышление. Тест выполняется в течение часа.

Примечание. Этот сценарий довольно длинный, поскольку в нем демонстрируется несколько подходов к задаче. Кроме того, была выбрана стабильная и проверенная на практике программа nopCommerce. В нее были специально внесены ошибки, чтобы показать, как решаются различные проблемы. Все сказанное ниже не следует рассматривать как оценку продукта nopCommerce.

Анализ. Этот сценарий похож на сценарий 5.1, поэтому и анализ можно начать точно так же, т. е. понаблюдать в системном мониторе (в реальном времени или в режиме групп сборщика данных) за следующими счетчиками:

- `\Процесс(Nop.Web)\Рабочий набор (частный)(\Process(Nop.Web)\Working Set - Private)`;
- `\Процесс(Nop.Web)\Байты исключительного пользования (\Process(Nop.Web)\Private Bytes)`;
- `\Процесс(Nop.Web)\Байты виртуальной памяти (\Process(Nop.Web)\Virtual Bytes)`;
- `\Память CLR .NET(Nop.Web)\Всего фиксировано байтов (\.NET CLR Memory(Nop.Web)\# Total committed Bytes)`;
- `\Память CLR .NET(Nop.Web)\Размер кучи поколения 0 (\.NET CLR Memory(Nop.Web)\Gen 0 heap size)`;
- `\Память CLR .NET(Nop.Web)\Размер кучи поколения 1 (\.NET CLR Memory(Nop.Web)\Gen 1 heap size)`;
- `\Память CLR .NET(Nop.Web)\Размер кучи поколения 2 (\.NET CLR Memory(Nop.Web)\Gen 2 heap size)`;
- `\Память CLR .NET(Nop.Web)\Размер кучи для массивных объектов (\.NET CLR Memory(Nop.Web)\Large Object Heap size)`.

Мы замечаем, что счетчик `Всего фиксировано байтов (# Total committed Bytes)` быстро растет в течение первых 20 минут теста. Затем объем занятой памяти внезапно падает и снова начинает очень быстро расти. Эта закономерность повторяется раз за разом. Размеры поколений, зафиксированные системным монитором, выглядят так (рис 5.7):

- размер поколения 0 (линия с длинными штрихами) постоянно изменяется между 4 194 300 и 6 291 456. Как мы уже знаем, это не настоящий размер поколения 0, а «бюджет выделения». Однако он изменяется стабильно, поэтому с поколением 0 проблем нет;
- размер поколения 1 (линия с короткими штрихами) изменяется динамически, но тоже стабильно. Не видно никакой тенденции к росту, поэтому можно предположить, что и здесь проблем нет;
- размер поколения 2 (тонкая сплошная линия), очевидно, выделяется на этом фоне. Кривая потребления памяти характеризуется странным треугольным паттерном. И это вызывает опасения, поскольку она достигает максимального значения 1 314 381 592 байт. Придется копнуть глубже, чтобы добраться до истинной причины проблемы;
- размер кучи больших объектов (жирная сплошная линия) растет очень медленно. Это может указывать на ту же проблему, но вряд ли является ее причиной. Отметим, что эта «утечка памяти» не особенно обременительна. LOH достигает размера 38 МБ (с небольшими пиками 46 МБ) после часа интенсивной работы. По сравнению с более чем гигабайтом памяти, занятым поколением 2, это ерунда.

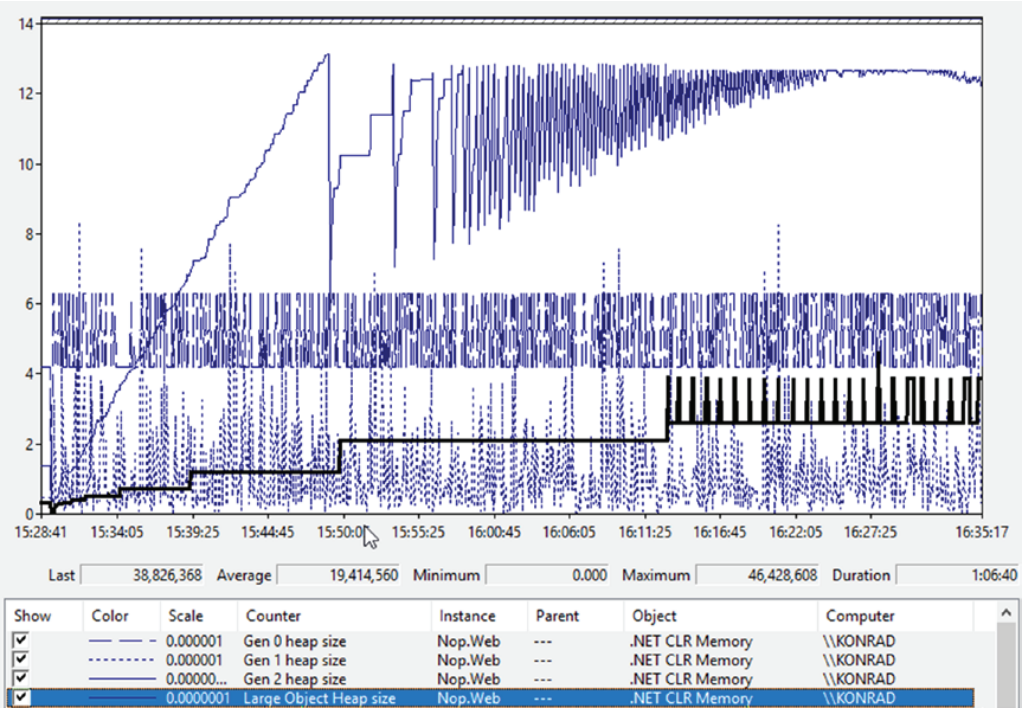


Рис. 5.17 ❖ Системный монитор – размеры поколений памяти после часа работы нагрузочного теста для приложения ASP.NET Core

Если во время работы теста взглянуть на состояние процесса `Nop.Web.exe` в программе VMMap, то мы наткнемся на первый ключ. Мы увидим множество строк вида Domain 1 Low Frequency Heap и Domain 1 High Frequency Heap, относящихся к низкочастотным и высокочастотным кучам в домене 1 (на рис. 5.18а показана только малая их часть). Раз их так много, то, вероятно, создается много динамических типов, например в результате применения отражения или загрузки большого числа сборок. Можно вспомнить сценарий 4.4, где точно такая же картина была связана с использованием класса `XmlSerializer`.

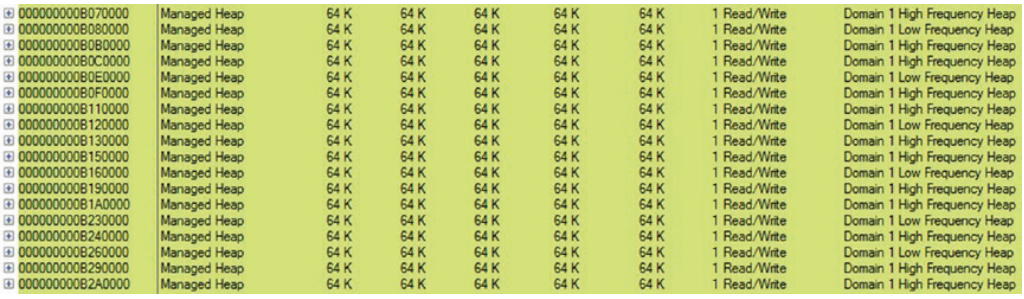


Рис. 5.18а ❖ Небольшая часть представления в VMMap процесса Nop.Web.exe во время тестирования. Видно много низкочастотных и высокочастотных куч в домене 1

Однако не будем спешить с выводами. Как и в сценарии 4.4, подтвердим свои подозрения, добавив к наблюдению следующие счетчики:

- `\Загрузка CLR .NET(Nop.Web)\Байт в куче загрузчика (\.NET CLR Loading(Nop.Web)\Bytes in Loader Heap)`;
- `\Загрузка CLR .NET(Nop.Web)\Всего загружено классов (\.NET CLR Loading(Nop.Web)\Current Classes Loaded)`;
- `\Загрузка CLR .NET(Nop.Web)\Текущее число сборок (\.NET CLR Loading(Nop.Web)\Current Assemblies)`;
- `\Загрузка CLR .NET(Nop.Web)\Текущих доменов приложений (\.NET CLR Loading(Nop.Web)\Current appdomains)`.

Удивительно, что эти счетчики не изменяются, даже если тестировать несколько часов. Наша догадка оказалась ложной. Вообще, даже большое количество низкочастотных и высокочастотных куч еще не означает, что имеется проблема. Если время от времени заглядывать в VMMap, мы увидим, что их число не меняется. Мы позволили обмануть себя. Наверное, их так много, потому что nopCommerce создает много динамических типов. Но на этой стадии более глубокое исследование не имеет смысла.

Отказавшись от этого пути, поинтересуемся основным подозреваемым – поколением 2. Снова воспользовавшись VMMap, мы можем отсортировать области управляемой кучи по столбцу Details, так чтобы все кучи GC располагались рядом (рис. 5.18b). Просматривая их, мы замечаем много сегментов, содержащих только объекты второго поколения. И еще можно обратить внимание на следующие факты:

- адреса короткие (старшая половина равна нулю) – это значит, что в процессе используется 32-разрядная среда выполнения .NET, но это мы знали уже, когда развертывали систему;
- существует всего один сегмент, содержащий объекты поколений 0 и 1 (эфемерный сегмент), – это означает, что, скорее всего, GC работает в режиме рабочей станции;
- размер сегментов, содержащих поколение 2, равен 16 МБ – согласно табл. 5.3, такое возможно только для 32-разрядного сборщика мусора в режиме рабочей станции, что подтверждает два предыдущих умозаключения.

Address	Type	Size	Committed	Private	Total WS	Private WS	...	Protection	Details
⊞ 000000000FB20000	Managed Heap	16,384 K	16,384 K	16,384 K	15,948 K	15,948 K	1	Read/Write	GC
⊟ 00000000122B0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,372 K	16,372 K	1	Read/Write	GC
00000000122B0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,372 K	16,372 K		Read/Write	Gen2
⊟ 000000000142F0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC
00000000142F0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K		Read/Write	Gen2
⊟ 0000000016C90000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC
0000000016C90000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K		Read/Write	Gen2
⊟ 0000000017C90000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC
0000000017C90000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K		Read/Write	Gen2
⊟ 000000001A6E0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC
000000001A6E0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K		Read/Write	Gen2
⊟ 000000001BCC0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC
000000001BCC0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K		Read/Write	Gen2
⊟ 000000001D2C0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC
000000001D2C0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K		Read/Write	Gen2
⊟ 000000001E2C0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC
000000001E2C0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K		Read/Write	Gen2
⊞ 000000001F2C0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC
⊞ 0000000202C0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC
⊞ 0000000212C0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC
⊞ 0000000224A0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC
⊞ 0000000234A0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC
⊞ 0000000244A0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC

Рис. 5.18b ❖ Небольшая часть представления в VMMap процесса Nop.Web.exe во время тестирования. Видно много управляемых куч GC, содержащих поколение 2

Веб-приложение, сконфигурированное для работы в 32-разрядной среде выполнения .NET с GC в режиме рабочей станции, – не самый оптимальный вариант. Пусть даже это интересный факт, о котором мы до сих пор не знали, наблюдаемую утечку памяти он не объясняет. Расследование нужно продолжить[1].

Программа VMMap включена в этот сценарий главным образом для того, чтобы показать физическую структуру .NET-приложения в свете знаний, полученных в этой главе. Кроме того, мы продемонстрировали потенциальные подводные камни, подстерегающие тех, кто решит ей воспользоваться (и подумает, что много высокочастотных куч – серьезная проблема). Иметь в своем арсенале VMMap полезно. Но не с VMMap обычно начинают поиск проблем, подобных этой. Посмотрев на счетчики производительности, надо было сразу перейти к WinDbg или PerfView.

Вот и нам пришло время достать другие инструменты. Первое, что приходит на ум, – WinDbg с расширением SOS. Получим полный дамп памяти `Nop.Web.exe` с помощью программы ProcDump. После загрузки его в WinDbg загрузим SOS командой `.loadby sos clr`. Затем можно выполнить еще две команды: `eeversion` (печать сведений о среде выполнения .NET) и `lmf` (вывести список всех загруженных модулей) – см. листинг 5.11. Как видим, в процессе используется .NET Framework 4.7 и GC в режиме рабочей станции. Он загрузил 32-разрядную версию `clr.dll` (64-разрядная версия находится в каталоге `C:\Windows\Microsoft.NET\Framework64`). Это окончательно подтверждает все наши предыдущие находки.

Листинг 5.11 ❖ Загружена WinDbg с расширением SOS. Команды `eeversion` и `lmf` показывают, что в процессе используется 32-разрядная версия .NET Framework и GC работает в режиме рабочей станции

```
> !eeversion
4.7.2117.0 retail
Workstation mode
SOS Version: 4.7.2117.0 retail build
> lmf
...
72f70000 73656000   clr   C:\Windows\Microsoft.NET\Framework\v4.0.30319\clr.dll
...
```

Исследование поколения 2 начнем с того, что выполним команды `heapstat` и `eeheap` (листинг 5.12). Как видим, поколение 2 действительно огромно (1 217 024 356 байт) и содержит не так много свободной памяти (10 981 728 байт). Фрагментация, по-видимому, отсутствует. Команда `eeheap` выводит подробные сведения о сегментах, которые мы уже видели в представлении VMMap.

Листинг 5.12 ❖ Загружена WinDbg с расширением SOS. Команды `heapstat` и `eeheap` выводят подробные сведения об управляемой куче GC. Вывод `eeheap` сокращен, показано только несколько интересных для нас строк

```
> !heapstat
Heap            Gen0       Gen1        Gen2        LOH
Heap0        9719400     280232  1217024356   38826368
Free space:                                        Percentage
Heap0        7042304       1152    10981728    12587408 SOH: 1% LOH: 32%
```

[1] И кстати, существуют другие, более простые способы узнать конфигурацию GC в работающем приложении. Они описаны в главе 8 (и прежде всего в сценарии 8.1).

```
> !eeheap
segment      begin  allocated        size
024c0000  024c1000  034bffe4   0xffefe4(16773092)
0a070000  0a071000  0b06ffe0   0xffefe0(16773088)
0fb20000  0fb21000  10b1ffdc   0xffefdc(16773084)
122b0000  122b1000  132affe0   0xffefe0(16773088)
142f0000  142f1000  152effe0   0xffefe0(16773088)
...
41820000  41821000  4281ffec   0xffefec(16773100)
43820000  43821000  4410ea14   0x8eda14(9361940)
42820000  42821000  431aa510   0x989510(9999632)
```

Зная диапазон адресов сегментов, мы можем изучить их содержимое с помощью команды dumpheap. Поскольку утечка памяти выглядит очень сильной и объекты живут долго, рассмотрим содержимое одного из первых сегментов (скорее всего, он один из самых старых). В листинге 5.3 показан результат команды dumpheap со статистикой объектов в четвертом сегменте. Большинство строк опущено, оставлены только последние. Как видим, имеется очень много объектов из пространства имен Microsoft.Extensions.Caching.Memory. И похоже, самый интересный класс, CacheEntry, свидетельствует о наличии каких-то проблем с кешированием.

Листинг 5.13 ❖ Загружена WinDbg с расширением SOS. Команда dumpheap показывает статистические сведения об объектах в одном из сегментов (большая часть строк опущена)

```
> !dumpheap -stat 122b1000 132affe0
      MT    Count TotalSize Class Name
...
04aa58e4    33795    946260  Microsoft.Extensions.Primitives.IChangeToken[]
0b542680    33808    946624  Microsoft.Extensions.Caching.Memory.
PostEvictionCallbackRegistration[]
089f26fc    33818   1082176  Microsoft.Extensions.Caching.Memory.PostEvictionDelegate
71f91d64    34858   4327314  System.String
089e2b70    33786   4459752  Microsoft.Extensions.Caching.Memory.CacheEntry
Total 431540 objects
```

Теперь можно начать утомительный процесс исследования различных экземпляров класса CacheEntry. Его таблица методов размещена по адресу 089e2b70, поэтому мы можем выполнить другую команду dumpheap, которая покажет только экземпляры класса Microsoft.Extensions.Caching.Memory.CacheEntry в четвертом сегменте (листинг 5.14). В ответ будет выведен огромный список, содержащий 33 786 экземпляров, поэтому мы оставили только последние несколько строк.

Листинг 5.14 ❖ Загружена WinDbg с расширением SOS. Команда dumpheap выводит все объекты в указанном сегменте с заданной таблицей методов

```
> !dumpheap -mt 089e2b70 122b1000 132affe0
 Address       MT    Size
...
132af460 089e2b70     132
132af64c 089e2b70     132
132af98c 089e2b70     132
132afd08 089e2b70     132
```

```
Statistics:
      MT    Count  TotalSize Class Name
089e2b70    33786    4459752 Microsoft.Extensions.Caching.Memory.CacheEntry
Total 33786 objects
```

Мы можем исследовать каждый экземпляр с помощью команды `DumpObj`, сообщив ей адрес (листинг 5.15). Одно из полей называется `<Key>k__BackingField`, и это наводит на мысль, что мы можем узнать ключ записи в кеше (также см. листинг 5.15). Оказывается, что он равен `Nop.pres.widget-79740-1-left_side_column_after_category_navigation-DefaultClean`, – похоже на данные, кешированные для какого-то виджета на странице.

Листинг 5.15 ❖ Загружена WinDbg с расширением SOS. Команда `DumpObj` выводит подробные сведения об одном из объектов, перечисленных в листинге 5.13

```
> !DumpObj 132afd08
Name:        Microsoft.Extensions.Caching.Memory.CacheEntry
MethodTable: 089e2b70
EEClass:     089c4f2c
Size:        132(0x84) bytes
File:        F:\IIS\nopCommerce\Microsoft.Extensions.Caching.Memory.dll
Fields:
...
71f81404  400000b         34 ...ffset, mscorlib]]   1 instance 132afd3c _
absoluteExpiration
...
71f92104  4000012         20        System.Object   0 instance 132afc18
<Key>k__BackingField
...
> !DumpObj 132afc18
Name: System.String
...
String: Nop.pres.widget-79740-1-left_side_column_after_category_
navigation-DefaultClean
```

Просматривать таким образом все экземпляры `CacheEntry` в сегменте было бы долго и утомительно. К счастью, для этой цели можно воспользоваться расширением netext, упомянутым в главе 3. Имеющаяся в нем команда `wfrom` позволяет писать SQL-подобные (или LINQ-подобные, если вам так больше нравится) запросы к объектам. Мы можем попросить вывести только поле `_Key_k__BackingField` объектов с заданным адресом таблицы методов, отфильтровав их по адресу интересующего нас сегмента (листинг 5.16).

> **ПРИМЕЧАНИЕ** Netext выводит имена полей несколько иначе, поэтому мы видим поле `_Key_k__BackingField` вместо `<Key>k__BackingField`.

Листинг 5.16 ❖ Загружена WinDbg с расширением netext. Команда `wfrom` выводит поле `_Key_k__BackingField` объектов с таблицей методов по адресу `089e2b70` в указанном диапазоне адресов

```
> !wfrom -mt 089e2b70 where (($addr() > 122b1000) && ($addr() < 132affe0))
select _Key_k__BackingField
...
```

```
_Key_k__BackingField: Nop.pres.widget-74954-1-mob_header_menu_after-
DefaultClean
_Key_k__BackingField: Nop.pres.widget-76130-1-header_menu_before-
DefaultClean
_Key_k__BackingField: Nop.pres.widget-75965-1-body_start_html_tag_after-
DefaultClean
_Key_k__BackingField: Nop.pres.widget-75369-1-searchbox_
before_search_
button-DefaultClean
_Key_k__BackingField: Nop.pres.widget-75965-1-searchbox_
before_search_
button-DefaultClean
_Key_k__BackingField: Nop.pres.widget-75867-1-header_selectors-DefaultClean
_Key_k__BackingField: Nop.pres.widget-75965-1-header_menu_before-
DefaultClean
_Key_k__BackingField: Nop.pres.widget-75573-1-body_start_html_tag_after-
DefaultClean
_Key_k__BackingField: Nop.pres.widget-75680-1-mob_header_menu_after-
DefaultClean
...
```

Просматривая результаты, мы сразу же видим бросающуюся в глаза закономерность. Почти все имена начинаются строкой `Nop.pres.widget`, за которой следуют цифры и (иногда) имя виджета. Теперь у нас есть уверенность, что проблема – в кешировании данных виджета. Возникает вопрос, почему так много похожих кешированных записей. Откуда берутся почти идентичные записи, различающиеся только первым числом? И сразу же второй запрос: кешируются ли они для каждого запроса?

Взглянув на несколько графов ссылок с помощью команды `gcroot`, можно заметить, что ссылки на эти записи удерживаются объектом `MemoryCacheManager` внутри объекта `ProductTagService` или ему подобных (листинг 5.17).

Листинг 5.17 ❖ Загружена WinDbg с расширением SOS. Команда `gcroot` показывает последовательность ссылок, ведущих к экземпляру `CacheEntry`. Поскольку этот путь очень длинный, оставлено только несколько интересных нам узлов

```
> !gcroot 132afd08
Thread 6d5c:
  0bc8f128 71ec99fa System.Threading.ExecutionContext.RunInternal(System.
Threading.ExecutionContext, System.Threading.ContextCallback, System.
Object, Boolean)
    ebp+4c: 0bc8f13c
        ->  0348777c System.Threading.Thread
        ->  025416d8 System.Runtime.Remoting.Contexts.Context
        ->  024c12e0 System.AppDomain
             ...
        ->  0ac5df50 Nop.Services.Catalog.ProductTagService
        ->  033dbacc Nop.Core.Caching.MemoryCacheManager
        ->  033db504 Microsoft.Extensions.Caching.Memory.MemoryCache
             ...
        -> 132afd08 Microsoft.Extensions.Caching.Memory.CacheEntry
```

Это самая трудная часть головоломки, решить ее без доступа к исходному коду нелегко. По счастью, мы обычно анализируем свое собственное приложение, поэтому код доступен и хорошо нам знаком. В этом конкретном случае выясняется, что ключ кеша содержит идентификатор пользователя, который для анонимных пользователей берется из cookie. Но в нашем тестовом сценарии для JMeter нет элементов диспетчера HTTP-cookies, который управлял бы ими! Иначе говоря, каждый HTTP-запрос рассматривается так, будто он отправлен новым пользователем, для которого cookie еще не установлен. Разумеется, такой сценарий нежелателен, это результат нашей ошибки при подготовке скрипта нагрузочного тестирования.

Исходный код nopCommerce открыт, поэтому причину проблемы найти легко.

- Поискав имя из примера среди ключей кеша (например, идентификатор `mob_header_menu_after`), мы найдем следующую строку в файле `./src/Presentation/Nop.Web/Views/Shared/Components/TopMenu/Default.cshtml`:

```
@await Component.InvokeAsync("Widget", new { widgetZone = "mob_
header_menu_after" })
```

- Виджет, определенный в файле `./src/Presentation/Nop.Web/Components/Widget.cs`, содержит простой метод `Invoke`, обращающийся к фабрике виджетов:

```
var model = _widgetModelFactory.PrepareRenderWidgetModel(widgetZone, additionalData);
```

- Метод `PrepareRenderWidgetModel` класса `WidgetModelFactory` конструирует ключ кеша `cacheKey` следующим образом:

```
var cacheKey = string.Format(ModelCacheEventConsumer.WIDGET_MODEL_KEY,
_workContext.CurrentCustomer.Id,
_storeContext.CurrentStore.Id,
widgetZone,
_themeContext.WorkingThemeName);
```

Как видим, виджеты пользуются идентификатором `CurrentCustomer.Id`, который для анонимного пользователя строится на основе cookie. Если cookie не существует, то берется новое целое значение.

Мы привели этот сценарий, чтобы показать, что, зная о поколениях и сегментах, можно заметить проблему и воспользоваться низкоуровневыми инструментами для поиска ее причины. Конечно, в реальных ситуациях причины могут быть очень разными. Ошибки при настройке нагрузочного теста, пожалуй, относятся к числу самых редких. Но цель упражнения заключалась в том, чтобы продемонстрировать не столько конкретную проблему, сколько подход к ее решению. Для анализа такой утечки памяти можно было бы использовать и более удобные инструменты, например PerfView или какую-нибудь коммерческую программу. Этот подход мы применим в следующих сценариях.

Сценарий 5.3. Растранжиривание кучи больших объектов?

Описание. Наше приложение для 64-разрядной рабочей станции обрабатывает очень большие списки объектов – допустим, это какие-то «большие данные». Но увы, проработав некоторое время, программа завершается с исключением `OutOfMemoryException`, и нам не удается обработать все данные. Процесс начинается

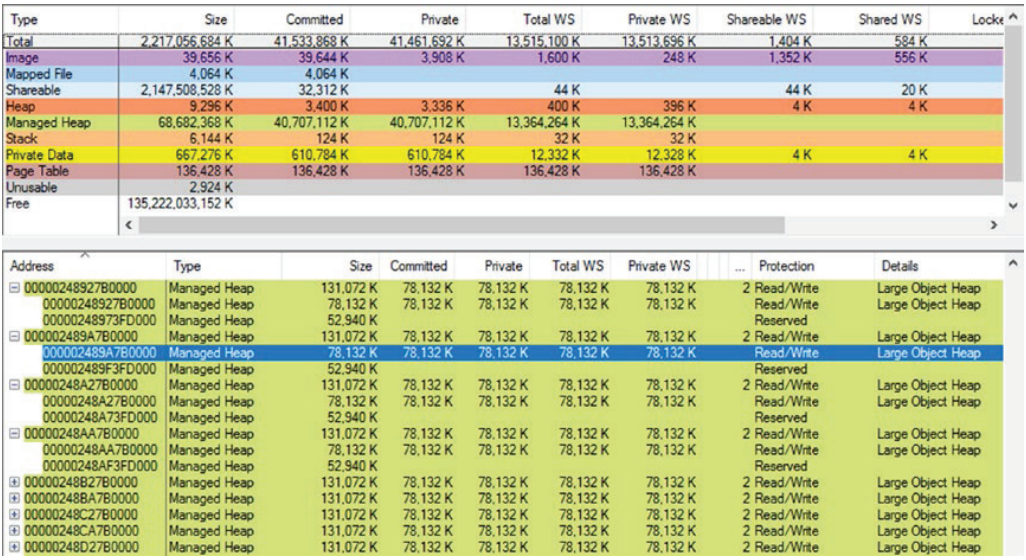
этапом предобработки – создается список больших массивов предобработанных объектов. Каждый такой блок содержит 10 000 000 ссылок на объекты, расположенные где-то в другом месте. Исключение `OutOfMemoryException` возникает при выделении памяти для этих массивов. Мы хотим, чтобы обработка была доведена до конца, поэтому начинаем расследование.

Анализ. Имеет смысл для начала посмотреть на процесс в VMMap в момент, непосредственно предшествующий исключению `OutOfMemoryException` (рис. 5.19). Видно, что действительно потребляется очень много памяти. Частный рабочий набор (private working set) процесса занимает примерно 15 ГБ, т. е. почти всю доступную физическую память (в машине установлено 16 ГБ ОЗУ). Более того, взглянув на страничный файл системы, мы увидим, что размер `pagefile.sys` составляет почти 32 ГБ – максимально возможный размер, заданный системным администратором. Это означает, что памяти для массивов не осталось, и поделать с этим ничего нельзя (разве что добавить еще ОЗУ или увеличить размер страничного файла).

Однако можно заметить тревожные симптомы в характере потребления сегментов. Количество сегментов LOH очень велико, и в каждом только половина памяти передана, а вторая половина лишь зарезервирована. Почему так происходит? Заглянув в табл. 5.3, мы увидим, что для 64-разрядной среды выполнения с GC, работающим в режиме рабочей станции, размер сегментов LOH составляет 128 МБ. Мы создаем массивы, содержащие 10 000 000 ссылок. Длина каждой ссылки 8 байт, поэтому для всего массива требуется приблизительно 76 МБ. Вновь создаваемый массив не помещается в уже имеющийся сегмент LOH, т. к. в нем осталось всего примерно 52 МБ. Поэтому для каждого массива приходится создавать новый сегмент. В результате 52 МБ в каждом сегменте LOH выбрасываются на ветер (предполагается, что наше приложение не создает объектов поменьше, которые можно было бы разместить в этой области).

Но вдумчивый читатель заметит ошибку в наших рассуждениях. Вспомните, о чем говорилось в главе 2: резервирование виртуальной памяти не приводит к потреблению физической памяти (нужно только запомнить несколько небольших дескрипторов резервирования). Внимательно приглядевшись к рис. 5.19, мы заметим, что зарезервированные части сегментов LOH не учитываются ни в столбце **Committed** (Передано), ни в столбце **Private** (Байты исключительного пользования). Вряд ли это можно назвать «растранжириванием» памяти. Но не будем обманываться этими измерениями. Фактически мы действительно заняли всю доступную память и ничего не можем с этим поделать (кроме выделения меньшего числа массивов за раз).

Однако такая пустая трата ресурсов из-за непригодной для использования зарезервированной памяти в сегментах LOH не составляет проблемы только на 64-разрядной машине, где виртуальное адресное пространство очень велико. А в 32-разрядной среде выполнения .NET, где виртуальное адресное пространство гораздо меньше, это может стать проблемой – да еще какой. Если вы столкнетесь с ней, то попробуйте разбить данные на массивы меньшей длины, чтобы лучше использовать каждый сегмент LOH и избежать фрагментации.



Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locke
Total	2,217,056,684 K	41,533,868 K	41,461,692 K	13,515,100 K	13,513,696 K	1,404 K	584 K	
Image	39,656 K	39,644 K	3,908 K	1,600 K	248 K	1,352 K	556 K	
Mapped File	4,064 K	4,064 K						
Shareable	2,147,508,528 K	32,312 K		44 K		44 K	20 K	
Heap	9,296 K	3,400 K	3,336 K	400 K	396 K	4 K	4 K	
Managed Heap	68,682,368 K	40,707,112 K	40,707,112 K	13,364,264 K	13,364,264 K			
Stack	6,144 K	124 K	124 K	32 K	32 K			
Private Data	667,276 K	610,784 K	610,784 K	12,332 K	12,328 K	4 K	4 K	
Page Table	136,428 K	136,428 K	136,428 K	136,428 K	136,428 K			
Unusable	2,924 K							
Free	135,222,033,152 K							

Address	Type	Size	Committed	Private	Total WS	Private WS	...	Protection	Details
00000248927B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
00000248927B0000	Managed Heap	78,132 K	78,132 K	78,132 K	78,132 K	78,132 K		Read/Write	Large Object Heap
00000248973FD000	Managed Heap	52,940 K						Reserved	
00000248927B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
00000248927B0000	Managed Heap	78,132 K	78,132 K	78,132 K	78,132 K	78,132 K		Read/Write	Large Object Heap
00000248973FD000	Managed Heap	52,940 K						Reserved	
00000248A27B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
00000248A27B0000	Managed Heap	78,132 K	78,132 K	78,132 K	78,132 K	78,132 K		Read/Write	Large Object Heap
00000248A73FD000	Managed Heap	52,940 K						Reserved	
00000248AA7B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
00000248AA7B0000	Managed Heap	78,132 K	78,132 K	78,132 K	78,132 K	78,132 K		Read/Write	Large Object Heap
00000248AF3FD000	Managed Heap	52,940 K						Reserved	
00000248B27B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
00000248BA7B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
00000248C27B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
00000248CA7B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
00000248D27B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap

Рис. 5.19 ❖ Часть представления процесса в VMMap
за несколько мгновений до исключения OutOfMemoryException

Анатомия сегментов и кучи

Ниже мы объясним, что сегмент – это физическое представление управляемой кучи. Его внутренняя структура проста, но знать ее полезно (рис. 5.20). В листинге 5.10 мы видели пример программы с эфемерным сегментом по адресу `0x0000026700000000`, который «начинается» с адреса `0x0000026700001000`. Эти начальные 4096 байт (шестнадцатеричное `0x1000`) предназначены для хранения информации о сегменте, которая нужна среде выполнения. Объекты создаются дальше. Любой сегмент SOH и LOH имеет следующую структуру.

- В начале хранится *информация о сегменте* (экземпляр класса `heap_segment`). Хотя этот экземпляр занимает всего несколько десятков байтов, в большинстве случаев под него отводится целая страница. Это оптимизация, используемая, если среда выполнения поддерживает популярную фоновую сборку мусора (см. главу 11). А к таковым относятся все среды выполнения для общего пользования, существовавшие на момент написания книги. Начало этой структуры (и самого сегмента) отображается как адрес сегмента рассмотренной выше командой `eeheap`.
- Объекты выделяются из адресного пространства, которое в исходном коде .NET называется `mem`. Однако команда `eeheap` называет этот адрес `begin`. В главе 6 мы увидим, что зарезервированная для сегмента память передается заранее (не только для одного объекта), поэтому переданной памяти будет немного больше, чем необходимо для уже размещенных объектов.
- Адрес, на котором кончаются размещенные в настоящий момент объекты, называется `allocated`.

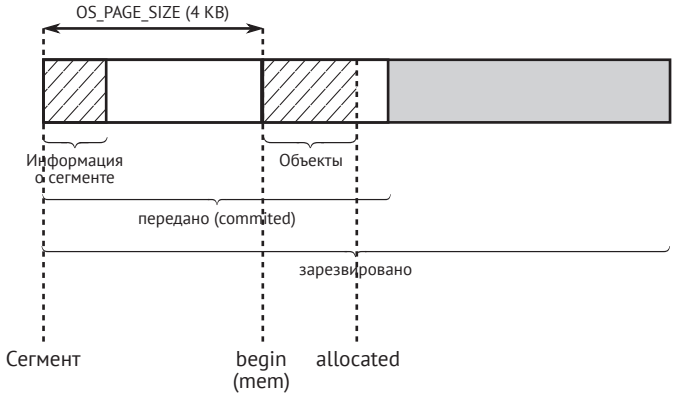


Рис. 5.20 ❖ Внутренняя структура сегмента кучи

Хотя в повседневной работе это и не так важно, при анализе кода .NET Core полезно знать связь между несколькими основополагающими классами, представляющими описанные здесь сущности. Это облегчит вам собственное путешествие в мир исходного кода CoreCLR, если вы когда-нибудь решитесь на него.

В реализации основной функциональности сборки мусора принимают участие следующие важные классы (рис. 5.21):

- `GCHeap` – всегда существует по крайней мере один экземпляр этого высокоуровневого API, который используется как интерфейс между сборщиком мусора и движком выполнения (оба они хранят ссылки на глобальный экземпляр – `g_pGCHeap` и `g_theGCHeap`). Этот класс содержит, в частности, методы `Alloc` и `GarbageCollect`. В серверном режиме каждая управляемая куча представляется дополнительным экземпляром `GCHeap`. Таким образом, в режиме рабочей станции имеется один экземпляр, а в серверном режиме еще по одному на каждое логическое ядро;
- `gc_heap` – низкоуровневый API единственной управляемой кучи, используемый `GCHeap`. Именно в нем находятся методы, выполняющие всю трудную работу GC: `allocate`, `garbage_collect`, `make_gc_heap`, `make_heap_segment` и т. д. В серверном режиме экземпляр `GCHeap` работает с соответствующим экземпляром `gc_heap`. В режиме рабочей станции все релевантные методы `gc_heap` статические, так что необходимости создавать экземпляр класса вообще нет. Поэтому в режиме рабочей станции его экземпляров нет вовсе, а в серверном режиме их столько, сколько логических ядер;
- `generation` – представляет одно поколение. Хранит информацию о сегментах, содержащих поколение, разнообразную информацию о выделении памяти и прочие данные;
- `heap_segment` – представляет один сегмент. Все сегменты образуют односвязный список, т. е. каждый сегмент (кроме последнего) содержит указатель на следующий.

Зная все вышеизложенное, мы теперь можем понять, как устроена реализация метода `GC.GetGeneration`, которым нам уже доводилось воспользоваться (листинг 5.18).

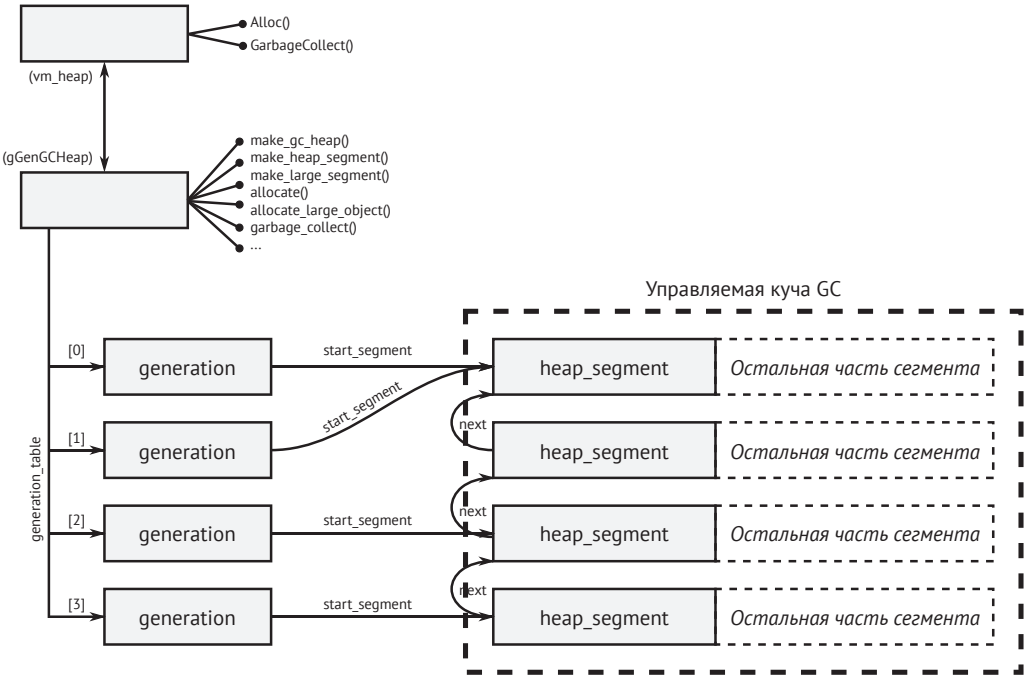


Рис. 5.21 ❖ Связи между основополагающими классами .NET, относящимися к сборке мусора (на основе кода .NET Core). Экземпляры `heap_segment` находятся в управляемой куче, в начале каждого сегмента, как описано выше. Все остальные данные находятся в частных кучах среды выполнения

Листинг 5.18 ❖ Метод класса `gc_heap`, который вызывается при выполнении метода `GC.GetGeneration`

```
// вернуть номер поколения объекта.
// Предполагается, что объект действительный.
// Отметим, что для объекта из LOH этот метод возвращает max_generation
int gc_heap::object_gennum (uint8_t* o)
{
   if (in_range_for_segment (o, ephemeral_heap_segment) &&
      (o >= generation_allocation_start (generation_of (max_generation-1))))
   {
      // в эфемерном сегменте.
      for ( int i = 0; i < max_generation-1; i++)
      {
         if ((o >= generation_allocation_start (generation_of (i))))
            return i;
      }
      return max_generation-1;
   }
   else
   {
      return max_generation;
   }
}
```

Повторное использование сегментов

Во время выполнения программы могут создаваться все новые и новые сегменты для размещения выделяемых объектов. А удаляются ли они когда-нибудь? Да, удаляются. Но, как часто бывает, ответ более сложен, чем просто «да».

Для начала подумаем, когда среда выполнения .NET может принять решение об удалении сегмента. На самом деле причина только одна – сегмент опустел после сборки мусора (в нем не осталось объектов). Мы увидим, когда это происходит, при обсуждении деталей сборки мусора.

Далее, что вообще означает «удалить сегмент»? В простейшем понимании это может быть вызов VirtualFree (или аналога в Linux) для всей зарезервированной для сегмента области памяти. То есть мы возвращаем память операционной системе. Представим себе ситуацию, изображенную на рис. 5.22a. В нашей программе имеется четыре сегмента. Поколение 2 настолько большое, что занимает два сегмента. Как уже было сказано, передано больше памяти (белые участки), чем необходимо для текущих объектов (заштрихованные участки), поскольку память подготавливается заблаговременно. По прошествии некоторого времени может произойти сборка мусора с уплотнением, в результате чего из поколения 2 будет удалено много объектов (рис. 5.22b) – настолько много, что один из сегментов, содержащих поколение 2, опустел (не содержит ни одного объекта). Но в этот момент вся память по-прежнему передана. В таком случае проще всего эту память освободить (рис. 5.22c).

Хотя этот подход кажется вполне разумным, у него есть серьезный недостаток. Из-за постоянного создания и удаления сегментов может возникнуть фрагментация. Особенно опасно это в 32-разрядных приложениях, где виртуальной памяти не так много, и прежде всего для долго работающих веб-приложений. Так обстояло дело во времена .NET 2.0 и ASP.NET 2.0, поэтому была реализована более изощренная обработка сегментов, получившая название *придерживание виртуальной памяти* (VM Hoarding). Идея очень проста. Вместо того чтобы полностью освобождать сегмент, мы можем сохранить (*придержать*) его для последующего использования (рис. 5.22d). В таком случае:

- вся память сегмента остается зарезервированной;
- большая часть физической памяти сегмента возвращается, а переданной остается только небольшая область в начале, включающая информацию о сегменте;
- сегмент помещается в список сегментов, допускающих повторное использование (`segment_standby_list` в случае CoreCLR). Когда понадобится новый сегмент, система сначала просмотрит этот список, и если найдет, то инициализирует его как новый.

Для 64-разрядных движков выполнения придерживание не так важно, поскольку виртуальной памяти гораздо больше. С другой стороны, при особенно динамичной работе, когда создается и удаляется много сегментов, все равно быстрее использовать уже зарезервированную память повторно, чем просить у системы новую. Так что даже в 64-разрядной ситуации этот механизм стоит использовать.

Однако по умолчанию придерживание сегментов выключено, потому что среда выполнения .NET не хочет придерживать виртуальную память, которую не использует (даже если она всего лишь зарезервирована). В обычном приложении

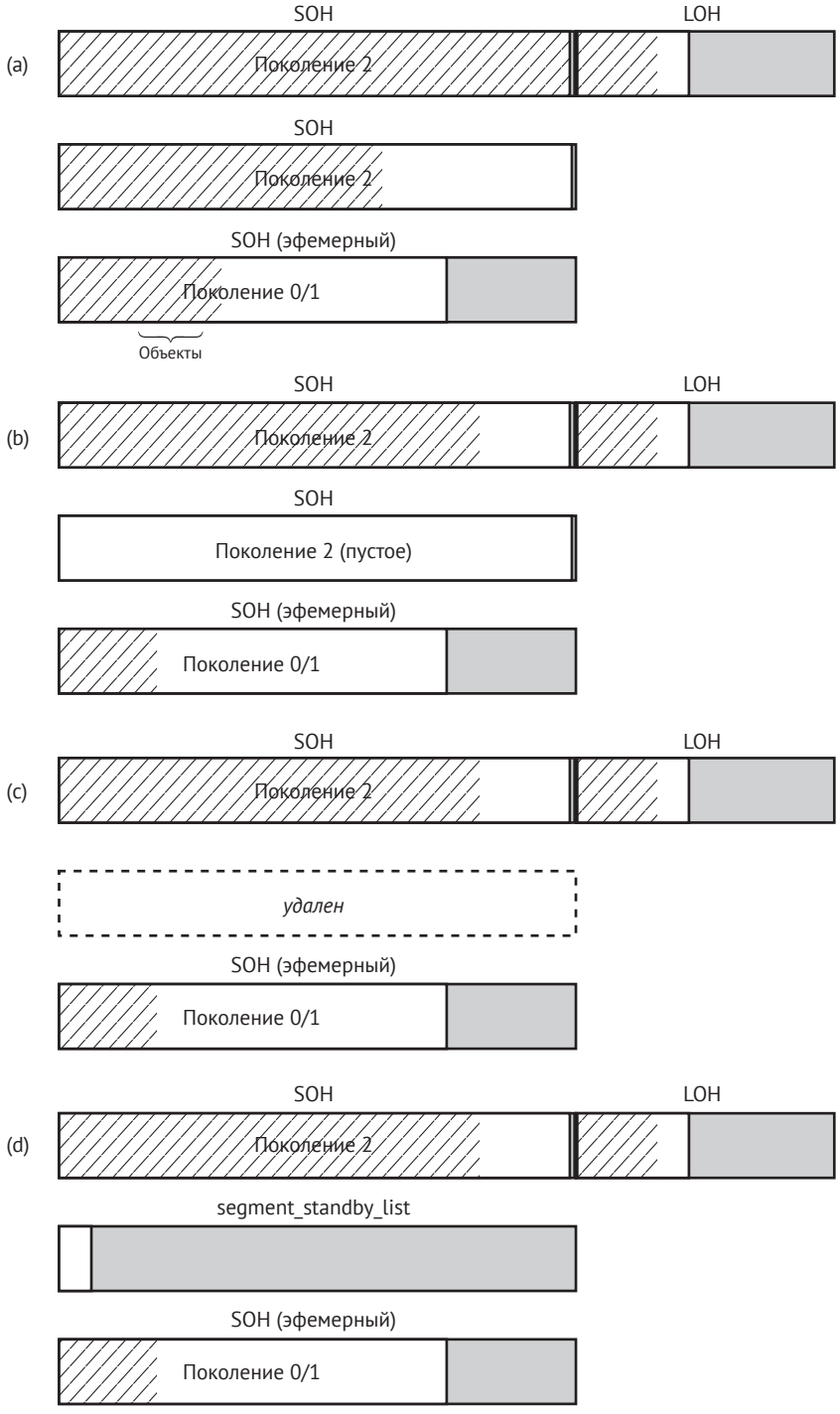


Рис. 5.22 ❖ Стратегии повторного использования сегмента

для рабочего стола или консольном .NET-приложении (не размещенном во внешнем процессе) придерживание виртуальной памяти, скорее всего, так и останется выключенным. Это поведение можно изменить, задав параметр `GCRetainVM` в переменной среды или в реестре (для Windows). Кроме того, процесс, в котором размещена среда выполнения .NET, может включить придерживание с помощью конфигурационного параметра `System.GC.RetainVM`. Именно так делают веб-приложения для ASP.NET, размещаемые в IIS, который включает придерживание по умолчанию. Мы тоже можем включить его вручную, если будем размещать среду выполнения .NET в своем приложении с помощью Hosting API (детали см. в главе 15).

Если вы хотите понаблюдать за тем, когда и почему создаются и удаляются сегменты в приложении, то проще всего воспользоваться событиями ETW (включив сбор стеков вызовов):

- `GCCreateSegment_V1` – показывает поля `Address`, `Size`, `ClrInstanceID` и `Type`;
- `GCFreeSegment_V1` – показывает поля `Address` и `ClrInstanceID`.

Поле `Type` может принимать одно из двух значений: `SmallObjectHeap` или `LargeObjectHeap`. Возможно также значение `ReadOnlyHeap`, упоминавшееся выше, но не в средах выполнения .NET Runtime и .NET Core, поскольку в них постоянные сегменты не используются.

Резюме

В этой главе мы рассмотрели много вопросов и приблизились к более глубокому пониманию того, как работает управление памятью в .NET. Мы описали, как физически и логически организована память, управляемая сборщиком мусора. В сочетании с материалом предыдущих глав это объясняет не только то, как выполняется некоторое действие, но и почему. Надеюсь, что теперь вы лучше понимаете, для чего нужно разделение на поколения, кучу малых объектов и кучу больших объектов.

В этой главе довольно подробно описаны различные аспекты организации памяти внутри управляемой кучи. Некоторые из них фундаментальны; не зная их, трудно понять .NET вообще. К таким относится прежде всего концепция сборки мусора на основе поколений. Поколения – основополагающее понятие, которое почти всегда встречается в контексте управления памятью в .NET-приложениях. Так что эти вопросы следует считать в высшей мере практическими. С другой стороны, были описаны также темы, представляющие меньший утилитарный интерес, но позволяющие глубже проникнуть во внутреннее устройство CLR и понять, как реализованы некоторые вещи.

Были также приведены три сценария решения проблем, так или иначе связанных с обсуждаемыми темами. Это дает возможность взглянуть на вопрос поколений и сегментов под более практическим углом зрения.

Правило 11: следите за размерами поколений

Обоснование. Слабая гипотеза о поколениях – фундамент сборщика мусора на основе поколений, реализованного в среде выполнения .NET. Если программа – вследствие необычных или ошибочных паттернов создания объектов – нарушает

эту гипотезу, то это может привести к серьезным последствиям в части производительности GC.

Как применять. Согласно правилам 5 и 6, мы должны измерять характеристики программы, наблюдая за поведением системы управления памятью. Один из самых важных показателей – динамика изменения размера поколений. Следует знать (если уж не вести непрерывный мониторинг), насколько велики поколения 0, 1, 2 и LOH. Нас должны насторожить два необычных поведения:

- одно или несколько поколений постоянно растет (даже если этот процесс растянут во времени и становится заметен после большого числа циклов сборки мусора) – это может свидетельствовать о сильной или умеренной утечке памяти;
- размер одного или нескольких поколений часто флуктуирует – это может быть признаком чрезмерно активной работы с памятью, в результате чего запускается дорогостоящий процесс сборки мусора.

Понятно, что одних лишь размеров поколений недостаточно. Можно представить себе, что размер поколения 2 стабилен, но внутри него постоянно происходит движение (т. е. одни объекты очень часто заменяются другими), поэтому мы тратим много времени на сборку мусора в поколении 2. Поэтому не менее важно измерять, как используется процессор (например, с помощью счетчика «% времени в GC»).

Иллюстрирующие сценарии: 5.1, 5.2.

Правило 12: избегайте лишних ссылок в куче

Обоснование. В сборке мусора на основе поколений существует специальная техника для отслеживания ссылок между поколениями – запомненные наборы (remembered sets). В среде выполнения .NET для ее реализации используются барьеры записи и таблицы карт. Как описано в этой главе, реализация сложна и делает все возможное, чтобы минимизировать накладные расходы. Однако лучшая межпоколенческая ссылка – та, которой не существует. Мы можем помочь GC в уменьшении накладных расходов, постаравшись не создавать слишком много ссылок, часть которых, возможно, и не нужна.

Как применять. Когда программа создает разного рода долгоживущие буферы или кеши, она зачастую помещает в них вновь создаваемые объекты. Это может приводить к появлению межпоколенческих ссылок (и запускать механизм таблиц карт). Однако иногда таких ссылок можно избежать. Например, в двоичном дереве можно хранить не ссылки на узлы, как здесь:

```
class Node
{
   Data d;
   Node left;
   Node right;
};
```

а индексы узлов; при этом сами узлы хранятся в массиве:

```
class Node
{
```

```
    Data d;
    uint left_index;
    uint right_index;
};
```

Но имейте в виду, что такое изменение влечет за собой куда больше последствий, чем ослабление нагрузки на механизм таблиц карт. Например, как будет выделяться память для массивов узлов? Как это изменение отразится на скорости обхода графа, ведь теперь нужно будет лишний раз заглядывать в массив для каждого узла? Лишь обстоятельное измерение и тестирование может ответить, принесло ли соблюдение этого правила выигрыш или наоборот.

Правило 13: наблюдайте за использованием сегментов

Обоснование. Сегменты – это деталь реализации, относящаяся к организации управляемой кучи. В большинстве случаев она надежно скрыта, так что и думать о ней не надо. Но, как всегда, имеются исключения. Сами сегменты и характер их размещения могут дать дополнительные ключи для анализа проблем, связанных с памятью. Редко, но бывает, что они сами становятся источником таких проблем, особенно в 32-разрядной среде, где виртуальной памяти не так много.

Как применять. Иногда полезно взглянуть на исследуемый процесс с помощью команд WinDbg (или таких инструментов, как VMMap). Анализ сегментов, созданных GC, может навести на мысли о причинах проблемы. Знать, как поколения располагаются в сегментах, особенно полезно при выполнении низкоуровневого анализа, например в WinDbg.

Иллюстрирующие сценарии: 5.2, 5.3.

Глава 6

Выделение памяти

В первых главах этой книги мы достаточно подробно изучили устройство памяти, а также рассмотрели ее низкоуровневые аспекты. Начиная с главы 4 мы все глубже погружаемся в реализацию управления памятью в .NET. Пока что мы занимались в основном деталями внутреннего устройства .NET (глава 4) и структурной организацией памяти (глава 5). На основе полученных знаний пора перейти к наиболее важным темам этой книги – принципам работы сборщика мусора. И чем ближе к этим темам, тем больше будет практических знаний, полезных как для диагностики, так и для программирования. Ну и, конечно, мы не обойдем стороной детали реализации.

Начнем с механизма, без которого невозможна работа ни одной программы, – с механизма выделения памяти. С его помощью мы получаем память для создаваемых приложением объектов. А создавать объекты необходимо, как бы мы ни пытались этого избежать. Даже простейшее консольное приложение создает множество вспомогательных объектов еще до того, как будет выполнена первая строчка нашего кода. Из-за особой важности и активного использования распределителя памяти в .NET были предприняты все усилия, чтобы сделать его максимально эффективным.

Вы, наверное, еще помните краткое описание концепции распределителя в главе 1 как «сущности, отвечающей за управление динамическим выделением и освобождением памяти». Там же был определен метод `Allocator.Allocate(amount)`, который отвечает за получение запрошенного объема памяти. На этом уровне абстракции распределителю не важен тип объекта, он просто предоставляет требуемое количество байтов (которые среда выполнения затем заполнит правильными значениями).

Введение в распределение памяти

Понятно, что абстрактный метод `Allocator.Allocate(amount)` – лишь вершина айсберга. Вся эта глава посвящена деталям реализации одного этого метода и вытекающим из них практическим советам.

Как мы помним из главы 2, операционная система предоставляет собственные механизмы выделения памяти. В неуправляемой среде, например в C/C++, именно они и используются. В Windows для этого служит Heap API, в Linux – комбинация вызовов `mmap` и `sbrk`. Но .NET может выиграть от появления дополнительного слоя между операционной системой и исполняемой программой, каковым

является среда выполнения. Чаще всего управляемые среды типа .NET заранее запрашивают непрерывный блок памяти и внутри него реализуют свой механизм распределения. Это гораздо быстрее, чем запрашивать у операционной системы дополнительную память при создании каждого нового объекта. Вызовы ОС обходятся дорого, да и, как мы увидим, нас устроят гораздо более простые механизмы.

Из предыдущей главы мы знаем, что управляемая куча GC состоит из сегментов. Это и есть то место, в котором происходит описанное в данной главе выделение памяти. Хотя мы пока не сказали об этом явно, вы, наверное, уже поняли, что выделение памяти для объектов имеет место:

- в поколении 0 в случае кучи малых объектов. Это было показано на рис. 5.4 и 5.5 в предыдущей главе. Физически память выделяется из эфемерного сегмента (содержащего поколения 0 и 1);
- непосредственно в куче больших объектов, поскольку она не делится на поколения. Физически это происходит в одном из сегментов, содержащих LOH.

В «Книге среды выполнения» это подытожено в следующих словах: «всякий раз, как выделяется память для большого объекта, рассматривается вся куча больших объектов. При выделении памяти для малых объектов рассматривается только эфемерный сегмент».

Существует два популярных способа реализации распределителя, и оба используются в .NET. Они упоминались в главе 1: *последовательное выделение памяти* и *выделение памяти из списка свободных блоков (free-list allocation)*. Теперь рассмотрим их внимательнее.

Выделение памяти сдвигом указателя

В распоряжении распределителя имеются сегменты. Самый простой и быстрый способ выделить память внутри сегмента – сдвинуть указатель на конец занятой памяти. Этот указатель называется *указателем выделения памяти*. Стоит сдвинуть его на количество байтов, соответствующих размеру создаваемого объекта, как можно принимать поздравления – память для объекта выделена! Эта идея иллюстрируется на рис. 6.1. Предположим, что сколько-то объектов уже создано (рис. 6.1a). Указатель выделения памяти показывает, где эти объекты кончаются. В этом месте будет расположен следующий объект. Когда программа запросит память для нового объекта A, значение указателя станет адресом этого объекта. А распределителю останется только сдвинуть указатель дальше на указанное число байтов (рис. 6.1b).

Псевдокод в листинге 6.1 иллюстрирует эту простую, но эффективную технику. Как мы увидим позже, такая реализация – лишь одна из стратегий выделения памяти, реализованных в CLR. На ассемблере код этой простой функции займет всего несколько команд, что делает ее чрезвычайно эффективной.

Листинг 6.1 ❖ Реализация простого распределителя методом сдвига указателя

```
Allocator.Allocate(amount)
{
   PTR result = alloc_ptr;
   alloc_ptr += amount;
   return result;
}
```

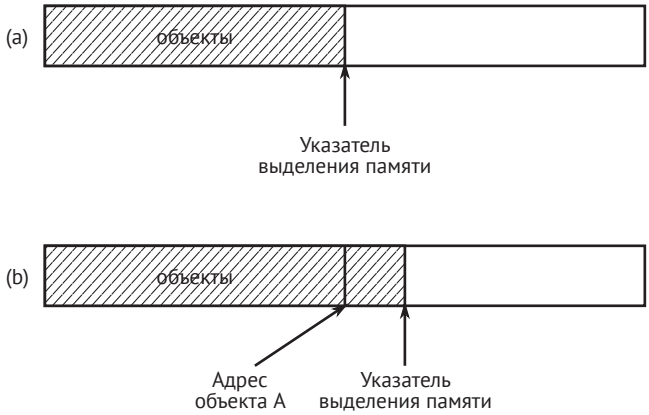


Рис. 6.1 ❖ Простая реализация последовательного распределителя

Этот способ иногда называют *выделением памяти методом сдвига указателя*, поскольку для выделения памяти достаточно время от времени сдвигать указатель вправо. Отметим два свойства такого алгоритма.

- Во-первых, как следует из названия, алгоритм последовательный: при выделении памяти мы просто сдвигаемся в одном направлении. Это может привести к хорошей локальности данных. Если программа сразу создает группу объектов, не исключено, что они представляют собой какие-то взаимно согласованные и зависящие друг от друга структуры данных, поэтому будет неплохо, если они и физически будут находиться рядом. Иначе говоря, данные, созданные примерно в одно и то же время, возможно, и будут использоваться одновременно (а, как мы помним из главы 2, архитектура ЦП устроена так, что максимальная эффективность достигается в случае временной и пространственной локальности данных).
- Во-вторых, эта модель предполагает, что память бесконечна. Ясное дело, такое предположение чрезмерно оптимистично. Мне бы хотелось иметь ОЗУ бесконечного объема, но, к сожалению, у меня всего 16 ГБ. Но разве идею последовательного выделения памяти из-за этого нужно отбросить? Нет, конечно, поскольку мы можем что-то сделать с объектами, находящимися слева от указателя. Например, удалить неиспользуемые и уплотнить оставшиеся. Тут-то в игру и вступает сборка мусора: после того как неиспользуемые объекты будут собраны, мы можем по мере необходимости перемещать обратно указатель выделения памяти.

Возникает вопрос: что происходит с содержимым той памяти, где находится объект A? Чтобы только что созданный объект был в начальном состоянии, эту память, конечно, нужно обнулить (какие-то поля будут затем инициализированы конструктором, но это уже задача движка выполнения, а не сборщика мусора). Поэтому в методе `Allocate` необходим дополнительный вызов для очистки (листинг 6.2).

Листинг 6.2 ❖ Реализация простого последовательного распределителя (с обнулением памяти)

```
Allocator.Allocate(amount)
{
   PTR result = alloc_ptr;
```

```
   ZeroMemory(alloc_ptr, amount);
   alloc_ptr += amount;
   return result;
}
```

Но обнуление памяти влечет за собой новые накладные расходы, которыми нельзя пренебречь в такой важной и частой операции, как создание новых объектов. Поэтому, чтобы выделение было максимально быстрым, имеет смысл заранее подготовить сколько-то памяти, заполненной нулями. Это позволит использовать код из листинга 6.1 для быстрого выделения памяти и вернуться к варианту с обнулением при необходимости. Заблаговременное обнуление памяти заодно повысит эффективность использования кеша ЦП, потому что доступ к этой области «прогреет» кеш.

Вводится еще один указатель, называемый *пределом выделения памяти*, который указывает на конец заполненной нулями области памяти. Эта область называется *контекстом выделения памяти* (рис. 6.2), и это то место, где возможно быстрое выделение памяти методом сдвига указателя.

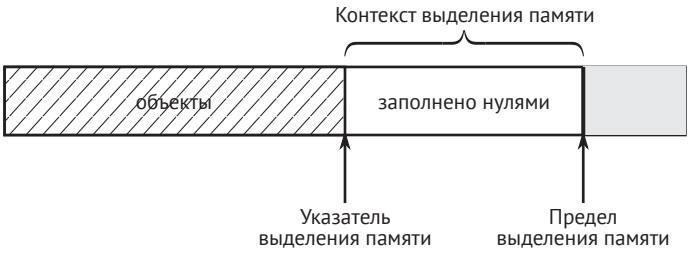


Рис. 6.2 ❖ Контекст выделения памяти занимает место между указателем выделения памяти и пределом выделения памяти. Он содержит заполненную нулями память, готовую к использованию

Если в контексте выделения памяти не хватает места для запрошенного количества байтов, то активируется резервный механизм (листинг 6.3), который может быть сколь угодно сложным. В случае CLR он содержит довольно запутанный конечный автомат, который, как мы увидим в следующих разделах, подробно описывает выделение памяти для куч больших и малых объектов. Одна из самых очевидных возможностей – увеличить размер контекста выделения памяти или получить новый нужного размера. Типичная величина такого приращения называется *квантом выделения памяти*. Иначе говоря, если в контексте выделения памяти недостаточно места, то типичный сценарий – увеличить его как минимум на размер кванта выделения памяти (или больше, если требуется больше памяти).

Листинг 6.3 ❖ Более реалистичный распределитель методом сдвига указателя с контекстом выделения памяти, предварительно заполненным нулями

```
Allocator.Allocate(amount)
{
   if (alloc_ptr + amount <= alloc_limit)
   {
      // Быстрый способ – памяти достаточно для сдвига указателя
```

```
      PTR result = alloc_ptr;
      alloc_ptr += amount;
      return result;
   }
   else
   {
      // Медленный способ - контекст указателя необходимо изменить,
      // чтобы удовлетворить запрос (т. е. увеличить по крайней мере на
      // величину кванта выделения памяти)
      if (!try_allocate_more_space())
      {
         throw OutOfMemoryException;
      }
      PTR result = alloc_ptr;
      alloc_ptr += amount;
      return result;
   }
}
```

В предыдущей главе отмечалось, что в GC уже есть один механизм подготовки памяти – двухступенчатое построение сегментов. Сначала резервируется большой блок памяти, а затем по мере необходимости выделяются страницы. Но когда сегмент растет в результате выделения страниц, эти страницы не обязательно сразу же обнуляются. Иными словами, контекст выделения памяти может не распространяться до конца выделенной памяти (рис. 6.3). Это компромисс между выигрышем от подготовки памяти и стоимостью ее обнуления. Например, для кучи малых объектов квант выделения памяти по умолчанию равен 8 КБ, тогда как при увеличении размера сегмента выделяется сразу 16 страниц (т. е. 64 КБ при стандартной длине страницы).

> Хотя по умолчанию размер кванта выделения памяти равен 8 КБ, при некоторых условиях он может динамически изменяться. В текущей реализации CLR эта величина может принимать значения от 1024 до 8096 байт в зависимости от интенсивности выделения памяти и количества активных контекстов.

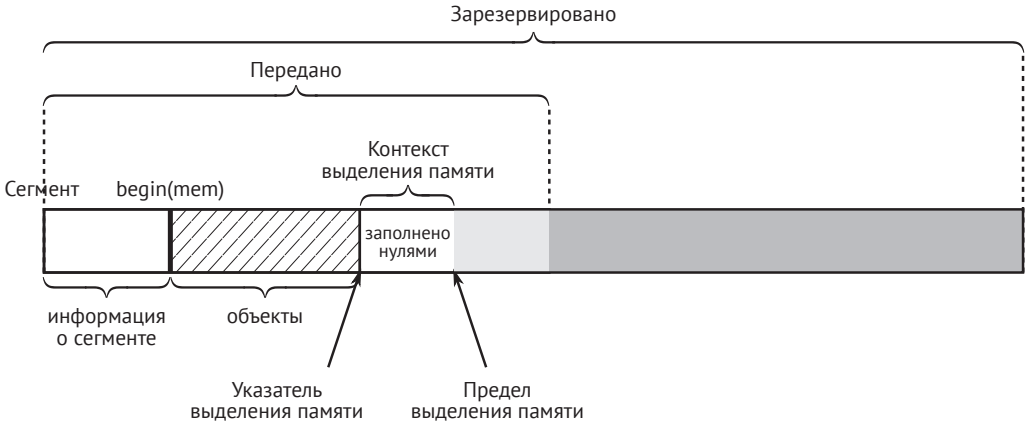


Рис. 6.3 ❖ Контекст выделения внутри сегмента находится после всех уже выделенных объектов

При таком подходе приходится гораздо реже обращаться к операционной системе для выделения страниц, при этом растет лишь контекст выделения памяти. Это куда более разумный способ получения памяти, чем выделение для каждого объекта, которое было бы совсем не эффективно.

Контекст выделения памяти можно размещать и в других местах, а не только в конце сегмента. Он может располагаться внутри свободного участка между существующими объектами (рис. 6.4). В таком случае указатель выделения памяти будет совпадать с началом свободного пространства, а предел выделения памяти – с его концом.

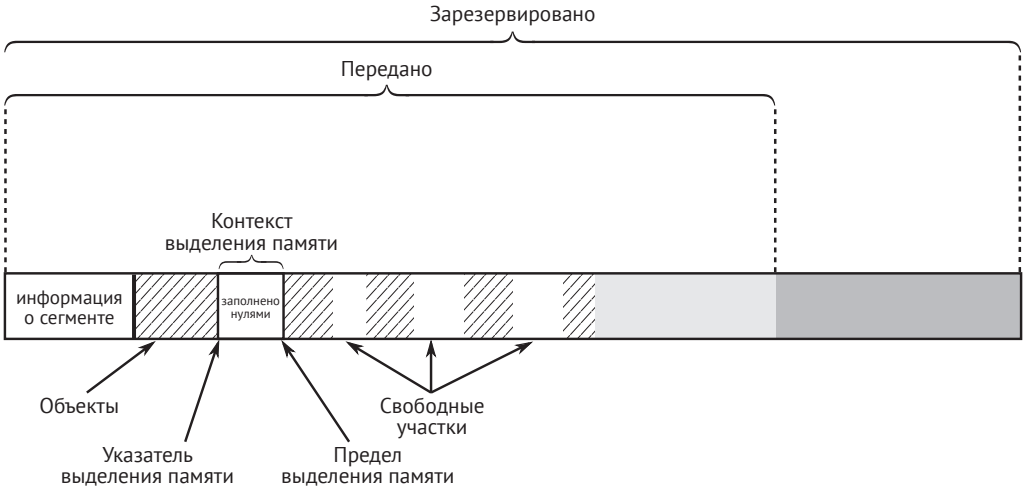


Рис. 6.4 ❖ Контекст выделения памяти внутри сегмента находится внутри свободного участка

Один из наиболее важных моментов заключается в том, что контекст выделения памяти обладает *привязкой к потоку* (thread affinity). Это означает, что у каждого управляемого потока (исполняющего код .NET) есть свой контекст выделения памяти. В «Книге среды выполнения» читаем: «Привязка к потоку контекстов и квантов выделения памяти гарантирует, что существует только один поток, который пишет в данный контекст выделения памяти. Поэтому нет необходимости ставить блокировку при выделении памяти для объекта до тех пор, пока текущий контекст выделения памяти не исчерпается».

Это свойство очень важно с точки зрения производительности. Если бы контекст выделения памяти совместно использовался несколькими потоками, то метод `Allocate` сопровождался бы накладными расходами на синхронизацию. Но поскольку у каждого потока свой контекст, простой метод сдвига указателя можно использовать, не опасаясь, что кто-то еще модифицирует указатель или предел выделения памяти. Механизм хранения одного контекста выделения памяти на каждый поток основан на *локальной памяти потока* (thread-local storage – TLS). В общем случае эту технику можно встретить под названием *локальный буфер выделения памяти потока (thread local allocation buffer)*.

ПРИМЕЧАНИЕ На машине с одним логическим процессором будет всего один контекст выделения памяти. Поэтому доступ к нему обязательно должен быть синхронизирован, поскольку к нему могут одновременно обращаться разные потоки. Но в таком случае синхронизация обходится очень дешево, потому что в каждый момент может работать только один поток.

Наличие нескольких контекстов выделения памяти немного усложняет рис. 6.3 и 6.4. Не существует единственного контекста в конце сегмента, как мы упрощенно рисовали ранее. В приложении работает много потоков, поэтому чаще можно встретить ситуацию, когда в одном сегменте расположено несколько контекстов выделения памяти (рис. 6.5). В процессе работы программы некоторые из них окажутся в конце сегмента, а другие будут использовать свободные участки между объектами.

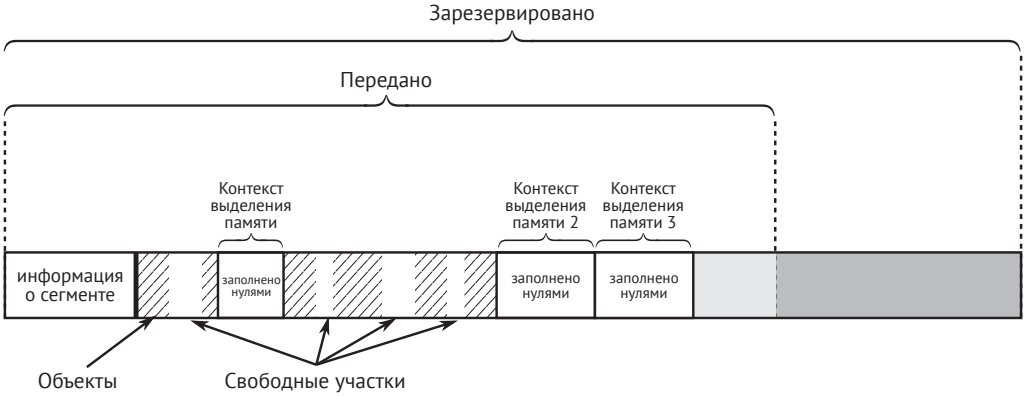


Рис. 6.5 ❖ Несколько контекстов выделения памяти внутри сегмента – по одному для каждого потока

Контекст выделения памяти находится в эфемерном сегменте – в том, что содержит поколения 0 и 1. Так что на рис. 6.5 показана структура эфемерного сегмента, а «объекты» могут распределяться между поколениями 1 и 0 (и 2, если оно мало, например в начале работы программы).

К этому моменту мы уже довольно хорошо знакомы с организацией памяти в .NET, но еще раз вкратце напомним ее на рис. 6.6. Помните, поколения – чисто логические подвижные границы внутри сегмента.

Выделение памяти методом сдвига указателя в своем оригинальном виде имеет один недостаток. Если выполнить сборку мусора очисткой для уже выделенных объектов, то, очевидно, возникнет фрагментация. Слева от указателя выделения памяти появится много пустых мест (рис. 6.7а). Наивный метод сдвига указателя (не тот, что применяется в .NET) не умеет обрабатывать такие случаи. Он может только запрашивать все новую и новую память. Понятно, никто не стал бы разрабатывать GC, который очищает кучу, но даже не пытается использовать освободившееся в результате место, чтобы разместить в нем новые объекты. Простейший выход – запустить сборку мусора с уплотнением, разместив выжившие объекты вплотную друг к другу и сдвинув весь контекст выделения памяти влево

(рис. 6.7b). Но существует куда лучшее решение, опирающееся не только на уплотнение.

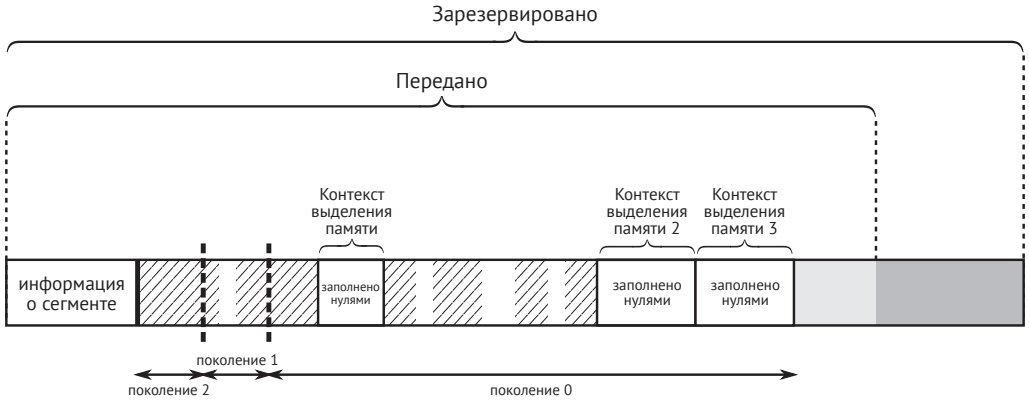


Рис. 6.6 ❖ Организация эфемерного сегмента

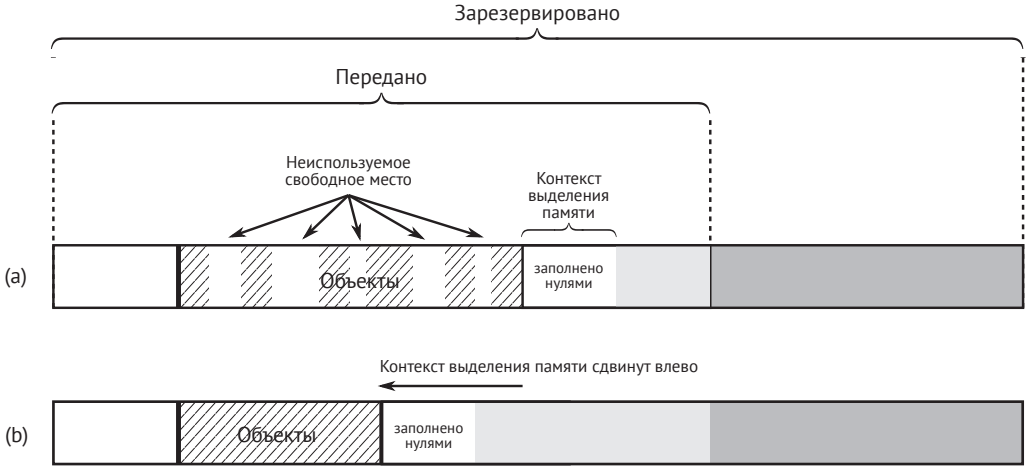


Рис. 6.7 ❖ Выделение памяти методом сдвига указателя и проблема фрагментации: (a) сборка мусора очисткой приводит к фрагментации, а контекст выделения памяти не знает о наличии свободной памяти; (b) сборка мусора с уплотнением возвращает память, сдвигая контекст выделения памяти влево, но при этом требует большого объема копирования

К счастью, .NET использует разумное сочетание последовательного выделения памяти и контекста выделения памяти, и, как показано на рис. 6.4 и 6.5, контекст выделения памяти может быть создан в свободном пространстве (т. е. фрагментация используется во благо). А время от времени GC решает выполнить уплотнение, и тогда все контексты выделения памяти перемещаются на свое естественное место в конец сегмента (рис. 6.8).

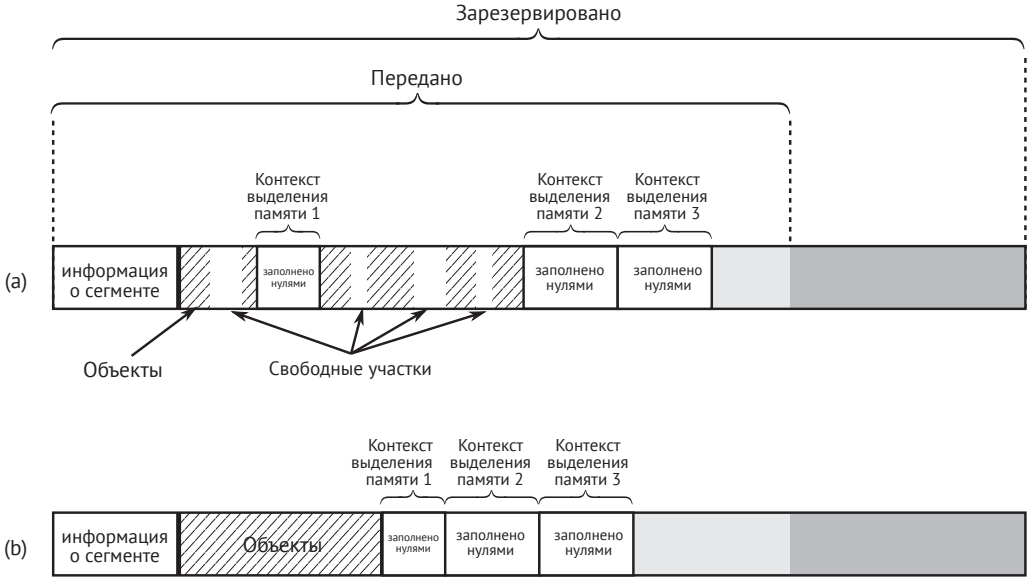


Рис. 6.8 ❖ Уплотняющий сборщик мусора может реорганизовать все контексты выделения памяти по завершении работы: (a) в начале три контекста выделения памяти разбросаны по сегменту; (b) после уплотнения контексты выделения памяти организованы оптимально

Выделение памяти из списка свободных блоков

Идея выделения памяти из списка свободных блоков тривиальна. Всякий раз, как среда выполнения просит GC выделить заданное число байтов, GC просматривает список свободных блоков памяти в поисках достаточно большого свободного промежутка. Как было сказано в главе 1, существует две стратегии просмотра:

- лучшее соответствие – ищется свободный промежуток, наилучшим образом отвечающий запросу (т. е. наименьшего размера, большего или равного запрошенному). Смысл в том, чтобы остаток был наименьшим из возможных. При наивном подходе пришлось бы просматривать весь список свободных блоков памяти, но ниже объясняется типичное решение, основанное на кластерах (buckets);
- первое соответствие – просмотр прекращается, как только будет найден первый достаточно большой промежуток. Это быстрее по времени, но хуже с точки зрения фрагментации.

В реализации Microsoft .NET для управления списками свободных блоков разного размера используются кластеры. В этом случае можно просматривать список быстро, не тратя лишнего времени на дефрагментацию. Подбирая количество кластеров (диапазонов размеров свободных блоков памяти), можно соблюсти баланс между производительностью и уменьшением фрагментации. Если кластер всего один (в него попадают все промежутки независимо от размера), то мы приходим к наивной стратегии первого соответствия. С другой стороны, если класте-

ров очень много (очень высокая гранулярность размеров), то мы приближаемся к стратегии лучшего соответствия. Как мы увидим, количество кластеров зависит от поколения.

Списки свободных блоков частично организуются прямо в куче GC благодаря способу представления свободного пространства. Свободное пространство между используемыми объектами представляется так, будто это (почти) обычный массив. Поэтому его структура очень похожа на структуру обычного объекта (рис. 6.9). Существует специальная таблица методов, представляющая такой «свободный объект». После указателя на таблицу методов хранится количество «элементов» свободного пространства, как в обычном массиве. Под «свободным объектом» понимается однобайтовый элемент, поэтому количество элементов в таком массиве – это попросту размер свободного пространства, выраженный в байтах. А вместо обычного заголовка объекта (который для «свободного объекта» не нужен) присутствует элемент, называемый «отмена» (undo). В нем, как мы увидим, во время обработки списка временно хранится адрес другого элемента списка свободных блоков.

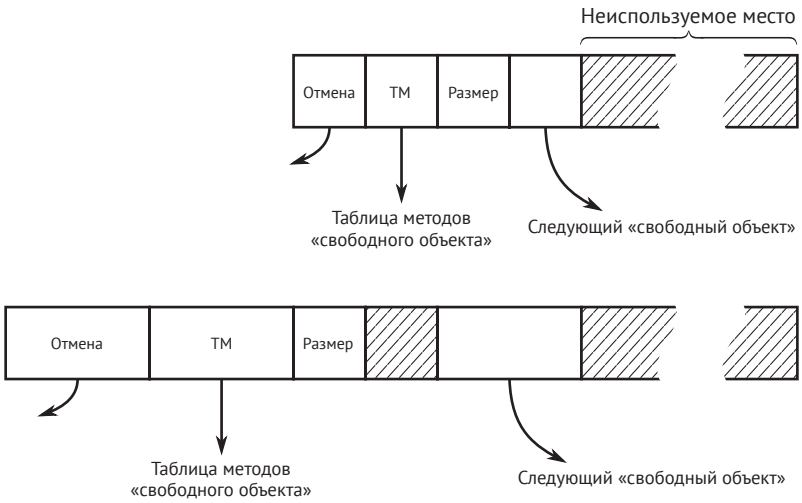


Рис. 6.9 ❖ Структура «свободного объекта», представляющая свободное пространство в куче GC

ПРИМЕЧАНИЕ Если вас интересует код CoreCLR, относящийся к «свободному объекту», начните с метода `gc_heap::make_unused_array`, в котором такой объект подготавливается. Вы увидите, что в качестве таблицы методов используется статический глобальный указатель на `g_pFreeObjectMethodTable`. Далее промежуток добавляется в список свободных блоков путем вызова метода `generation_allocator(gen)->thread_item (gap_start, size)`. Однако добавление в список производится только для промежутков, размер которых превышает удвоенный минимальный размер объекта. Это позволяет избежать накладных расходов на обработку совсем небольших промежутков.

Распределитель для каждого поколения управляет списком кластеров (рис. 6.10). Первый кластер представляет собой список свободных блоков, размер которых меньше `first_bucket_size`. В каждом следующем кластере размер удваивается,

а последний кластер предназначен для промежутков сколь угодно большого размера. В каждом кластере хранится описание соответствующего списка свободных блоков и в первую очередь его заголовок. Однако, как мы видим на рис. 6.10, сам список реализован как односвязный список «свободных объектов», которые находятся прямо в куче GC. Это позволяет быстро обходить его, поскольку по крайней мере часть кучи уже находится в кеше. Ведение отдельного списка в данном случае совершенно излишне.

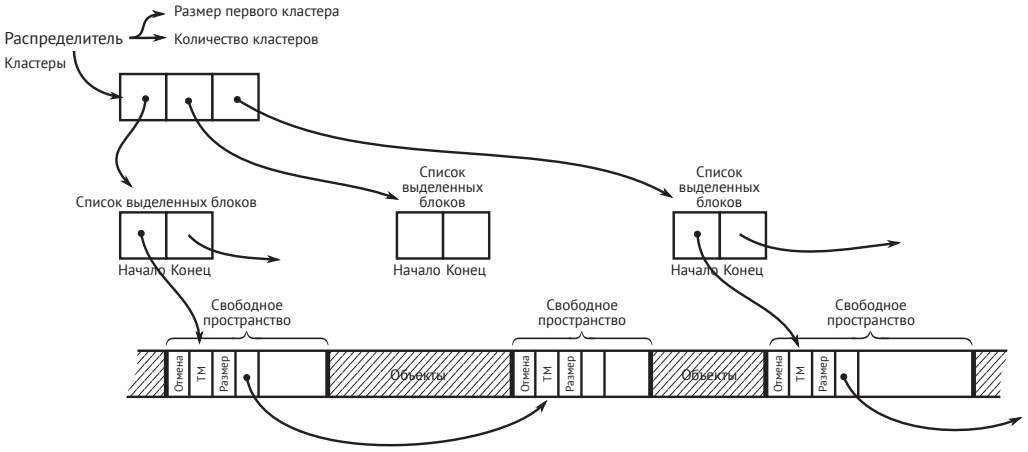


Рис. 6.10 ❖ Реализация списка свободных блоков на основе кластеров в CLR

Вас, возможно, удивит, что у каждого поколения есть собственный распределитель, поскольку ранее было ясно сказано, что выделение объектов производится либо в поколении 0 кучи малых объектов, либо в куче больших объектов. Это правда, пользовательские объекты так и создаются. Но когда GC переводит выжившие объекты в следующее поколение, он должен выделять в этом поколении память.

Для каждого поколения задаются свои значения количества и размера кластеров (см. табл. 6.1). Как видим, в обоих эфемерных поколениях есть только один кластер для всех размеров. Конфигурация поколения 2 различна для 32- и 64-разрядных сред. Так, в 64-разрядной среде выполнения GC поддерживает кластеры для размеров меньше 256 Б, 512 Б, 1 КБ, 2 КБ, 4 КБ, 8 КБ и более 8 КБ.

Таблица 6.1. Конфигурация кластеров списков свободных блоков в каждом поколении

Область	Размер первого кластера	Количество кластеров
Поколение 0	`Int.Max`	1
Поколение 1	`Int.Max`	1
Поколение 2	256 Б (64-разрядная) 128 Б (32-разрядная)	12 12
LOH	64 КБ	7

Выделение памяти с помощью списков свободных блоков, собранных в кластер, производится очень просто (листинг 6.4). Мы должны начать с первого подходящего кластера и поискать первый подходящий свободный блок в соответствующем

списке. После выделения нужного количества памяти из найденного промежутка в нем может остаться место. Если оно больше удвоенного минимального размера объекта (48 байт на 64-разрядной платформе), то создается новый «свободный объект» и включается в список. В противном случае эта небольшая область памяти будет считаться непригодным к использованию артефактом фрагментации.

Листинг 6.4 ❖ Реализация выделения памяти из списка свободных блоков на псевдокоде

```
Allocator.Allocate(amount)
{
   foreach (bucket in buckets)
   {
      if (amount < bucket.BucketSize) // пропускаем кластеры, содержащие слишком
                                      // маленькие промежутки
      {
         foreach (freeItem in bucket.FreeItemList)
         {
            if (size < freeItem.Size)
            {
               UnlinkItem(freeItem);
               ZeroMemory(freeItem.Start, amount);
               if (RemainingFreeSpaceBigEnough())
                  ThreadRemainingFreeSpace(freeItem, amount);
               return freeItem.Start;
            }
         }
      }
   }
}
```

Заметим, что обнуление памяти в листинге 6.4 необходимо только для объектов, созданных пользователем (поскольку при создании они должны находиться в своем начальном состоянии), но при переводе из нового поколения в старое эту операцию можно опустить (поскольку память все равно будет переписана содержимым перемещенного объекта). В .NET именно так и реализовано. Кроме того, в случае поколений 0 и 1 «свободный объект» отбрасывается (становится непригодным для использования из-за фрагментации), если не отвечает требованиям к размеру. Это означает, что в этих двух поколениях каждый свободный объект проверяется только один раз. Это еще один компромисс между стоимостью поддержания списка свободных блоков и платой за допущение фрагментации. Два младших поколения часто уплотняются, поэтому список свободных блоков нередко перестраивается.

Вышеупомянутое поле «отмена» свободного объекта нужно сборщику мусора на этапе планирования, когда он решает использовать один из свободных элементов для выделения памяти. Точнее, речь идет о выделении в старшем поколении памяти для перемещаемого объекта в том случае, когда GC хочет задействовать для этой цели свободный элемент. Тогда GC исключает использованный элемент из списка свободных блоков с помощью стандартных манипуляций с указателями (рис. 6.11).

- Адрес удаленного элемента сохраняется в поле «отмена» предыдущего элемента (если таковой имеется).

- В поле «следующий» предыдущего элемента записывается адрес следующего свободного элемента (того, на который указывал удаленный из списка элемент).

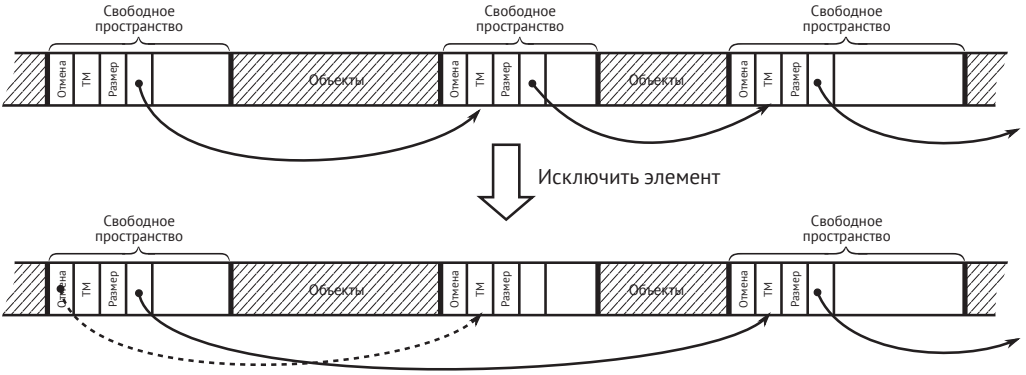


Рис. 6.11 ❖ Исключение элемента из списка свободных блоков

Но, как было сказано, это делается на этапе планирования, и впоследствии GC может решить, что надо произвести очистку. Тогда нужно отменить использование элементов списка свободных блоков (поскольку в случае очистки старшее поколение остается нетронутым, так что все запланированные операции выделения памяти следует отменить). Благодаря сохранению адреса свободного элемента в поле «отмена» мы можем восстановить исходное состояние списка. Более подробно об этапах планирования, уплотнения и очистки мы узнаем в главе 7.

Создание нового объекта

Зная два основных способа выделения памяти для объектов, мы можем перейти к описанию того, как они используются совместно в .NET. Между выделением памяти в куче больших и малых объектов есть существенные различия, поэтому описание разделено на две части.

Созданию нового объекта ссылочного типа (например, с помощью оператора `new` в C#, см. листинг 6.5) в CIL соответствует команда `newobj` (листинг 6.6).

Листинг 6.5 ❖ Создание объекта в C#

```
var obj = new SomeClass();
```

Листинг 6.6 ❖ Создание объекта на общем промежуточном языке

```
newobj instance void SomeClass::.ctor()
```

JIT-компилятор генерирует для команды `newobj` вызов подходящей функции в зависимости от различных условий. Чаще всего используется один из *методов выделения памяти (allocation helpers)*. На рис. 6.12 показано соответствующее дерево решений. Все решения основаны на условиях, которые известны во время JIT-компиляции или даже раньше, на этапе запуска среды выполнения. Отметим две основные возможности:

- если размер объекта превосходит пороговое значение (объект создается в LOH) или у объекта есть финализатор (специальный метод, который мы подробно обсудим в главе 12), будет выбран общий и более медленный метод выделения памяти `JIT_New`;
- в противном случае выбирается более быстрый – какой именно, зависит от платформы и режима работы GC.

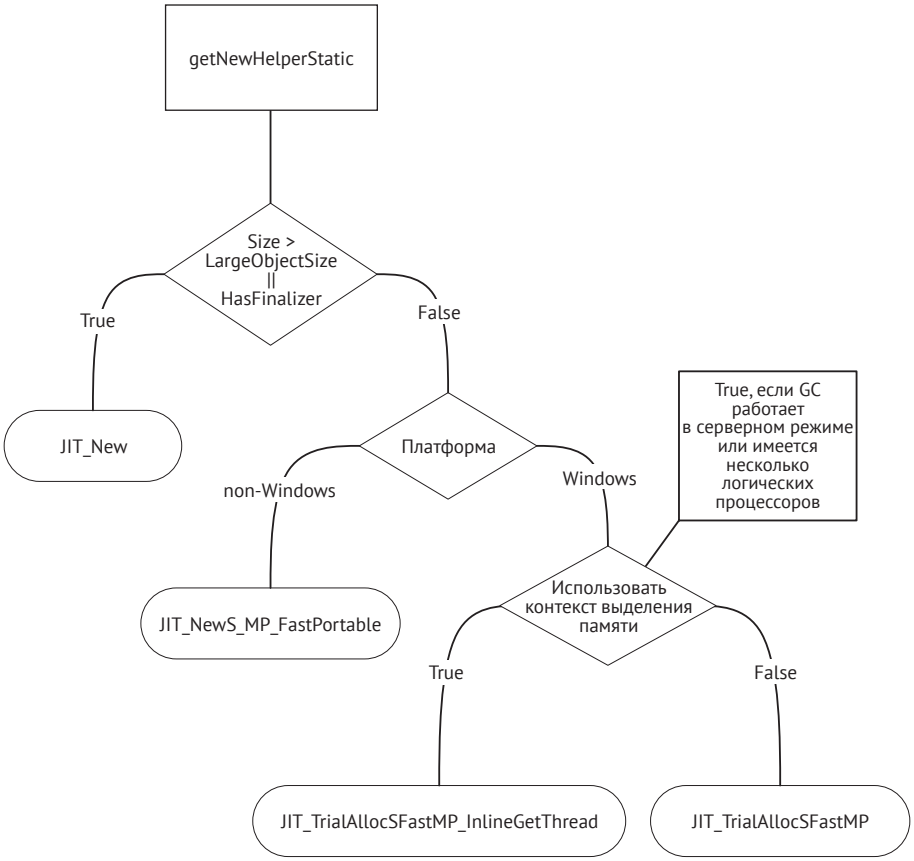


Рис. 6.12 ❖ Дерево решений для выбора метода выделения памяти на этапе JIT-компиляции (имена функций взяты из кода CoreCLR)

Важно помнить, что дерево решений используется во время JIT-компиляции, и в результате его обработки генерируется подходящий метод. Поэтому на этапе выполнения программы нет никаких накладных расходов – просто позже будет вызван один из вышеперечисленных методов.

> **ПРИМЕЧАНИЕ** Созданию массивов соответствует команда CIL `newarr`, для которой есть свои варианты, например оптимизированный для создания одномерных массивов объектов или для создания одномерных массивов типов значений. Но, поскольку реализация выделения памяти для каждого из них по существу такая же, мы для краткости опустили этот момент.

Если вы хотите покопаться в деталях выделения памяти в коде CoreCLR, то начните с реакции JIT-компилятора на код операции `CEE_NEWOBJ` (файл `importer.cpp`, метод `Compiler::impImportBlockCode`). Тут принимается решение, что делать: создать массив, строку, тип значений или ссылочный тип. Для ссылочных типов, отличных от строк и массивов, вызывается метод `CEEInfo::getNewHelper`, который обрабатывает часть дерева решений на рис. 6.12. Более медленный общий метод представлен константой `CORINFO_HELP_NEWFAST`, а более быстрый – константой `CORINFO_HELP_NEWSFAST`. Какие функции реализуют эти методы, решается на этапе запуска среды выполнения в методе `InitJITHelpers1`. Он обрабатывает оставшуюся часть дерева решений.

Выделение памяти в куче малых объектов

Выделение памяти для объектов, размещаемых в куче малых объектов, основано в основном на стратегии сдвига указателя. Цель состоит в том, чтобы по возможности выделять память для объектов с помощью стратегии сдвига указателя в контексте выделения памяти, как описано выше в этой главе. И только если это не получается, будет выбран более медленный способ (см. ниже).

Самый быстрый метод выделения памяти для SOH реализует стратегию, показанную в листинге 6.3, и занимает всего несколько машинных команд (см. листинг 6.7). Он используется для всех объектов в SOH, не имеющих финализатора (как следует из дерева решений на рис. 6.12), в случае когда GC работает в серверном режиме, или вообще на машине с несколькими логическими процессорами.

Вариант для однопроцессорной машины называется `JIT_TrialAllocSFastSP`. Он включает механизм блокировки, гарантирующий безопасный доступ к единственному глобальному контексту синхронизации.

Этот код действительно очень эффективен – всего несколько команд сравнения и сложения. Поэтому часто можно услышать, что «выделение памяти в .NET обходится дешево». И да, как следует из комментариев, быстрая оптимистическая ветвь «выделяет» память для объекта очень быстро – нужно лишь увеличить указатель в обнуленной области памяти в пределах контекста выделения памяти (память для которого уже передана).

Листинг 6.7 ❖ Самый быстрый помощник выделения

```
; На входе rcx содержит указатель на таблицу методов
; На выходе rax содержит адрес нового объекта
LEAF_ENTRY JIT_TrialAllocSFastMP_InlineGetThread, _TEXT
   ; Прочитать размер объекта в edx
   mov edx, [rcx + OFFSET__MethodTable__m_BaseSize]
   ; Гарантируется, что m_BaseSize кратно 8.
   ; Прочитать адрес локальной памяти потока (TLS) в r11
   INLINE_GETTHREAD r11
   ; Прочитать предел выделения памяти в r10
   mov r10, [r11 + OFFSET__Thread__m_alloc_context__alloc_limit]
   ; Прочитать указатель выделения в rax
   mov rax, [r11 + OFFSET__Thread__m_alloc_context__alloc_ptr]
   add rdx, rax             ; rdx = alloc_ptr + size
   cmp rdx, r10             ; rdx меньше alloc_limit?
   ja AllocFailed
```

```
    ; Обновить alloc_ptr в TLS
    mov [r11 + OFFSET__Thread__m_alloc_context__alloc_ptr], rdx
    ; Сохранить таблицу методов по адресу alloc_ptr (нового объекта)
    mov [rax], rcx
    ret
AllocFailed:
    jmp JIT_NEW             ; быстро выделить не получилось, пойдем по медленному пути
LEAF_END JIT_TrialAllocSFastMP_InlineGetThread, _TEXT
```

Если в текущем контексте выделения памяти не хватает места, то быстрый, написанный на ассемблере распределитель вызывает более общий метод `JIT_NEW` (тот, который используется для объектов с финализатором или размещаемых в LOH). В этом методе выделение памяти будет проводиться более медленным способом. Из-за вынужденного отказа от быстрого пути фраза «выделение памяти обходится дешево» не всегда правдива. Медленный способ реализован в виде сложного конечного автомата, который ищет область требуемого размера.

Насколько сложен медленный способ? На рис. 6.13 показан соответствующий ему конечный автомат. Он начинается в состоянии `_state_start`, когда описанная выше попытка быстрого выделения закончилась неудачно. Из этого состояния ведет безусловный переход в состояние `_state_try_fit`, где вызывается метод `gc_heap::soh_try_fit()` (рис. 6.14). Здесь все и начинается. Возможных решений много, мы перечислим лишь самые важные.

- Сначала делается попытка использовать уже имеющееся неиспользуемое место в эфемерном сегменте (см. рис. 6.14, где описан метод `soh_try_fit`). Для этого метод:
  - пробует найти в списке свободных блоков подходящий промежуток для размещения нового контекста выделения памяти (вспомните рис. 6.4);
  - пробует изменить предел выделения памяти в уже переданной (committed) памяти;
  - пробует дополнительно выделить память из зарезервированной и соответственно изменить предел выделения памяти.
- Если ничто из вышеперечисленного не сработало, запускается сборка мусора. В зависимости от условий сборщик мусора может быть вызван несколько раз.
- Если и это не помогло, значит, распределитель не может выделить запрошенную память. Это критическая ситуация, поэтому начинается обработка исключения `OutOfMemoryException`.

Код медленного способа выделения памяти в куче малых объектов находится в методе CoreCLR `gc_heap::allocate_small`, логика которого показана на рис. 6.13.

В данных ETW запуск GC из-за попытки выделения памяти в SOH (самый частый) обозначается значением AllocSmall в поле причины.

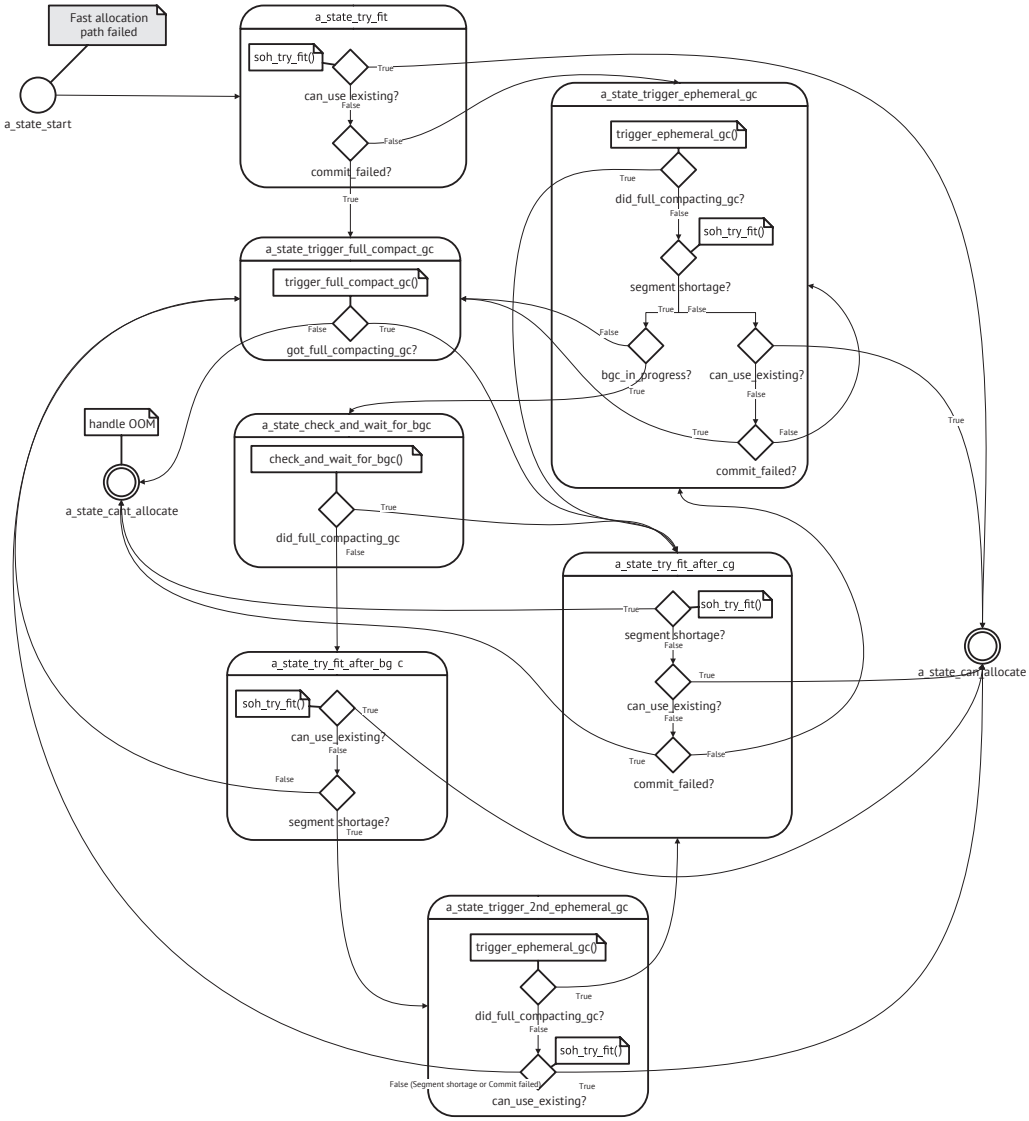


Рис. 6.13 ❖ Сложный конечный автомат, соответствующий медленному способу выделения памяти в куче малых объектов

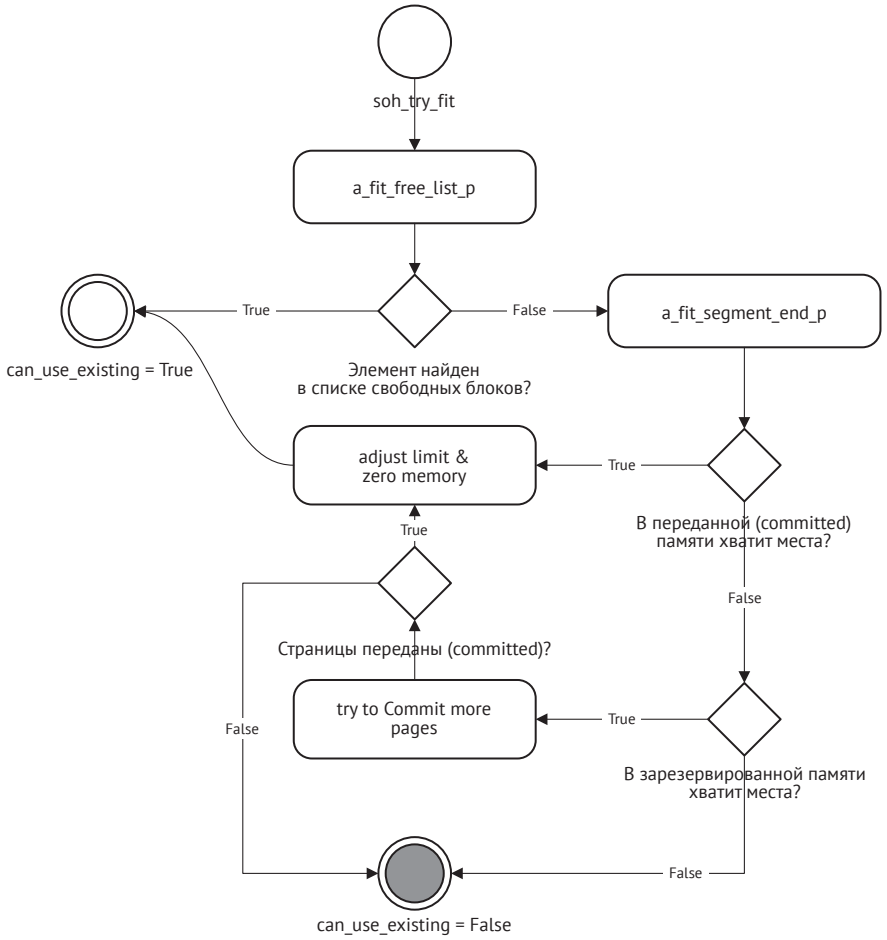


Рис. 6.14 ❖ Дерево решений в методе `soh_try_fit`

Описывать весь конечный автомат с рис. 6.13 особого смысла не имеет. Все это очень глубокие детали реализации, которые могут измениться еще до того, как книга выйдет из печати (однако я все равно рекомендую потратить некоторое время на самостоятельное изучение). Но важно отметить, насколько сложен медленный способ выделения памяти по сравнению с быстрым (сначала попытка найти место в списке свободных блоков, потом запуск одной или нескольких сборок мусора). Так что будем помнить, что фраза «выделение памяти обходится дешево» верна лишь в определенной степени. Надо понимать, какие действия производятся при выделении памяти, и пользоваться этим аккуратно: не созда-

вать объекты без необходимости и не использовать библиотеку, активно потребляющую память, не понимая, что она делает. Как видим, даже без запуска GC медленный способ выделения памяти недешев. Если производительность стоит на первом месте, то лучшее правило, относящееся к выделению памяти, – не выделяйте память совсем (что приводит нас к правилу 14 «Избегайте выделения памяти»).

Также имейте в виду, что для объектов с финализаторами по умолчанию используются более общие методы выделения памяти. С механизмом финализации сопряжены дополнительные накладные расходы, описанные в главе 12. Отсюда правило 25 «Избегайте финализаторов».

Выделение памяти в куче больших объектов

Выделение памяти для объектов, размещаемых в куче больших объектов, основано на стратегии выделения памяти из списка свободных блоков в сочетании с упрощенной стратегией сдвига указателя до конца сегмента (без использования контекста выделения памяти). Контекст выделения памяти и связанные с ним оптимизации не так важны, поскольку преобладает стоимость очистки большого объекта и, значит, нет смысла тратить силы на оптимизацию того, что не даст заметного эффекта. Вместо этого лучше позаботиться о потенциальной фрагментации, происходящей из-за того, что в LOH применяется только сборка мусора очисткой (если мы явно не попросим произвести уплотнение).

Поэтому в распределителе памяти в LOH нет разделения на быстрый и медленный способы выделения памяти. Всегда выбирается один и тот же способ, очень похожий на медленный путь для SOH (рис. 6.15):

- сначала распределитель пробует воспользоваться имеющимся неиспользованным пространством (см. рис. 6.16, где описывается метод `loh_try_fit`). Для этого метод:
- пробует найти подходящий для размещения объекта промежуток в списке свободных блоков памяти.

В каждом сегменте, содержащем LOH:

- он пробует изменить предел выделения памяти в уже переданной (committed) памяти;
- пробует дополнительно использовать зарезервированную память и соответственно изменить предел выделения памяти.
- Если ничто из вышеперечисленного не сработало, запускается сборка мусора. В зависимости от условий сборщик может быть вызван несколько раз.
- Если и это не помогло, значит, распределитель не может выделить запрошенную память. Это критическая ситуация, поэтому начинается обработка исключения `OutOfMemoryException`.

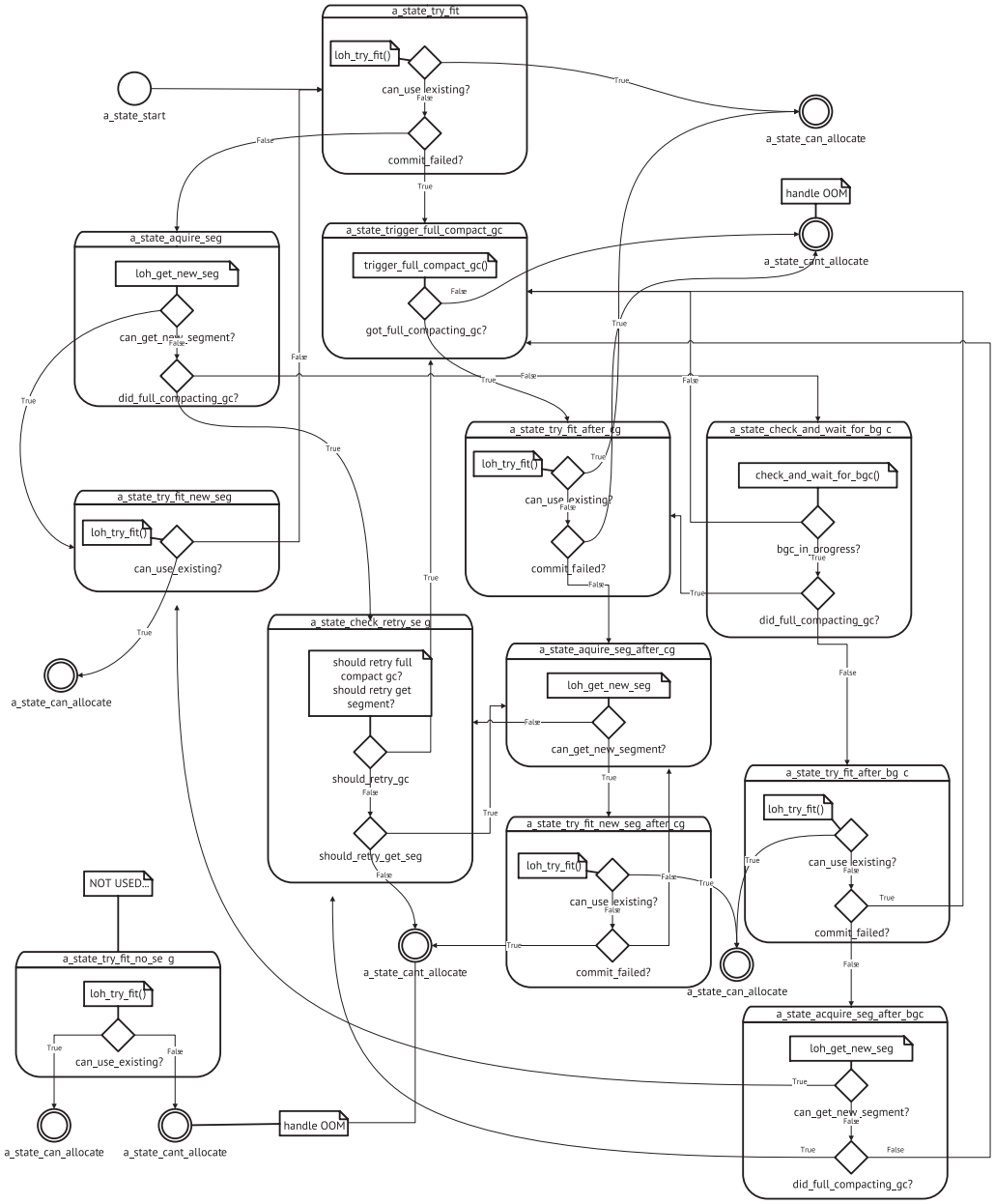


Рис. 6.15 ❖ Сложный конечный автомат выделения памяти в куче больших объектов

Код медленного выделения памяти в куче больших объектов находится в методе CoreCLR `gc_heap::allocate_large`, логика которого показана на рис. 6.16.

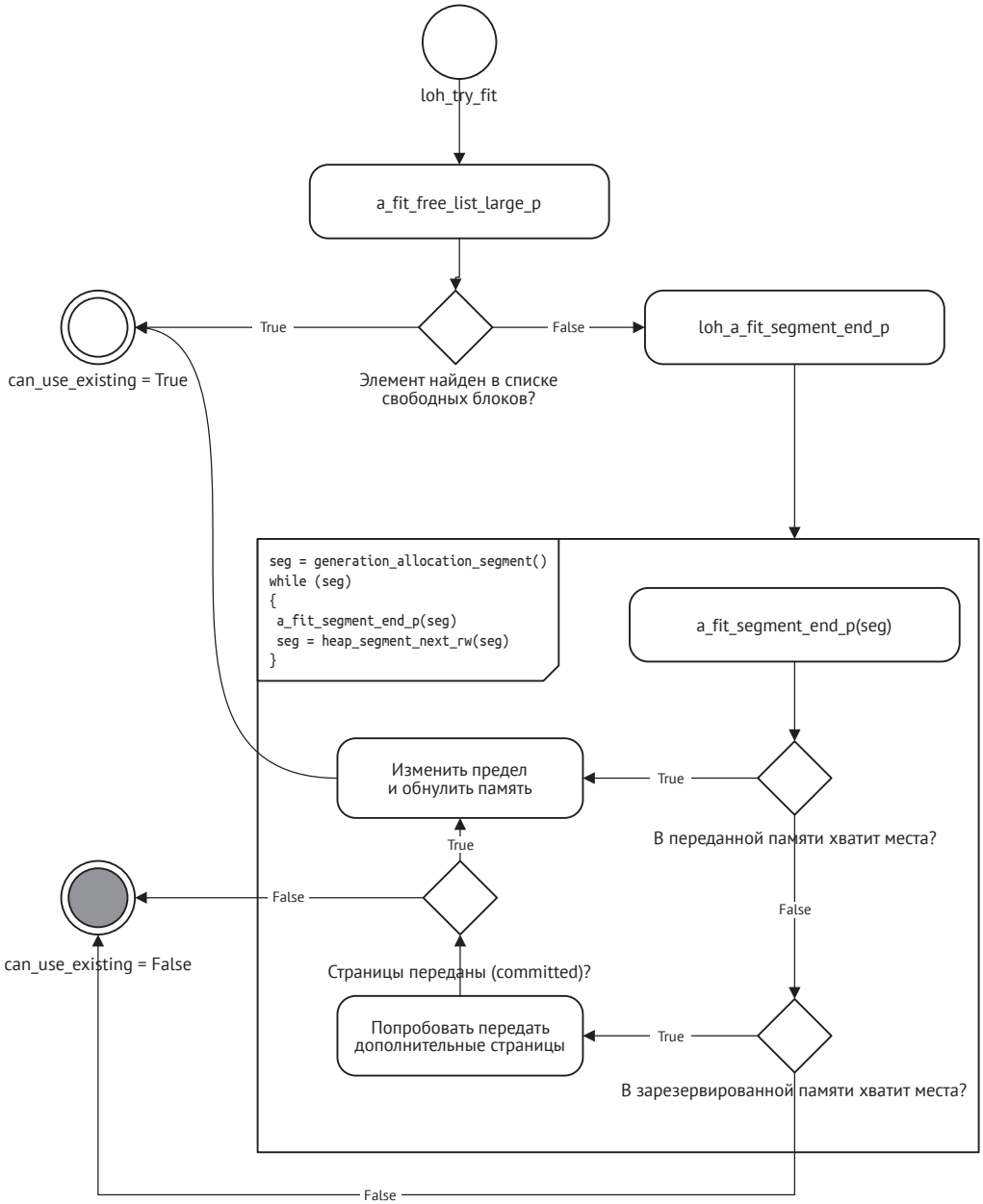


Рис. 6.16 ❖ Дерево решений в методе `loh_try_fit`

Как видим, конечный автомат для LOH еще сложнее, чем тот, что показан на рис. 6.13. Как и в том случае, описывать все возможные состояния и действия не имеет смысла. Заметим, однако, что в LOH контекст выделения памяти не используется. Но распределитель тем не менее должен гарантировать, что сразу после создания объект находится в своем начальном состоянии, поэтому выделенную

для него память необходимо обнулить. Стоимость обнуления памяти больших объектов может быть весьма значительной. Учитывая задержки доступа к памяти, описанные в табл. 4.2 из главы 2, обнуление объекта размером несколько мегабайтов может занимать десятки миллисекунд. В зависимости от приложения это может оказаться очень долго.

Поэтому запомним, что выделение памяти в LOH обходится еще дороже, чем в SOH. Поэтому его тем более нужно избегать, что подводит нас к правилу 15 «Избегайте чрезмерного выделения памяти в LOH». Проще всего решить эту проблему, создав пул повторно используемых объектов.

> **ПРИМЕЧАНИЕ** Сборщик мусора в .NET GC постоянно совершенствуется, и зачастую новая версия среды выполнения приносит заметные улучшения. Например, начиная с версии .NET 4.5 (и, значит, и .NET Core 1.0) распределитель памяти в LOH подвергся значительной переработке и теперь лучше использует список свободных блоков благодаря применению описанного выше подхода на основе кластеров.

Возникает интересный вопрос: чему равен максимальный размер объекта, который мы можем создать в .NET? В самой первой версии .NET он был равен 2 ГБ. И хотя мы редко создаем настолько большие объекты, бывает, что требуется массив, занимающий еще больше места. До выхода .NET 4.5 обойти это ограничение было невозможно. А в версии 4.5 появился новый параметр `gcAllowVeryLargeObjects` (листинг 6.8), который позволяет создавать объекты, размер которых можно выразить 64-разрядным числом со знаком (на самом деле чуть меньше, но это несущественно). И хотя теперь можно создавать массивы, большие 2 ГБ, другие ограничения на размер объекта или массива не изменились:

- максимальное количество элементов массива равно `UInt32.MaxValue` (2 147 483 591);
- максимальный индекс по любому измерению равен 2 147 483 591 (0x7FFFFFC7) для массивов байтов и однобайтовых структур и 2 146 435 071 (0X7FEFFFFF) для массивов других типов;
- максимальный размер строк и других объектов, отличных от массивов, не изменился.

Листинг 6.8 ❖ Задание конфигурационного параметра `gcAllowVeryLargeObjects` (по умолчанию выключен)

```
<configuration>
   <runtime>
      <gcAllowVeryLargeObjects enabled="true" />
   </runtime>
</configuration>
```

Где же создается такой огромный объект? Понятно, что в одном из сегментов LOH, поскольку его размер больше пороговой величины. Скорее всего, для него будет создан новый сегмент, поскольку маловероятно, что среди уже имеющихся найдется готовый вместить такой большой объект. И имейте в виду: выделение памяти для такого объекта может занять несколько секунд из-за задержек доступа к памяти!

Балансировка кучи

Мы уже несколько раз говорили, что в серверном режиме GC управляет несколькими кучами – по одной на каждое логическое ядро, доступное среде выполнения. Раз есть несколько управляемых куч, то эфемерных сегментов и сегментов LOH тоже несколько. С другой стороны, в приложении работает несколько управляемых потоков. Как они уживаются друг с другом? Как куча назначается потоку?

Для ответа на этот вопрос нужно сначала ответить на другой: как кучи назначаются логическим процессорам? При обсуждении этой темы нам понадобятся знания из главы 4 о взаимодействии процессора и памяти. Понятно, что CLR хочет, чтобы управляемая куча находилась как можно «ближе» (в терминах времени доступа) к конкретному логическому процессору (ядру). И конечно, ей хотелось бы избежать накладных расходов на синхронизацию ядер. Поэтому были приняты следующие проектные решения:

- если ОС предоставляет информацию о том, какое ядро исполняет текущий поток (это верно для Windows и, вероятно, для большинства версий Linux и macOS), то логические ЦП назначаются управляемым кучам по порядку, и это назначение никогда не изменяется. Это позволяет соответственно заполнять кеши ЦП во время выполнения программы и не вытеснять содержимое слишком часто. С другой стороны, управляемая куча никогда не используется совместно несколькими ядрами, чтобы избежать накладных расходов на работу протоколов когерентности кешей[1];
- если ОС не предоставляет информацию, то выполняется микротест производительности, чтобы эмпирически определить, какая куча ближе к конкретному ядру с точки зрения времени доступа;
- если в компьютере используются группы NUMA (см. главу 2), то назначение куч производится внутри одной группы.

> Если вас интересует, как выполняется микротест производительности, начните с метода `heap_select::access_time`.

Когда управляемый поток начинает выделять память, ему назначается куча – та, которая была назначена процессору, на котором выполняется поток. Типичная взаимосвязь между управляемыми кучами, потоками и логическими ядрами показана на рис. 6.17. Два логических процессора потребляют управляемую память, при выделении которой применялась стратегия «все сразу», описанная в предыдущей главе. Первому ЦП назначены сегменты SOH_1 и LOH_1, второму – сегменты SOH_2 и LOH_2 (т. е. ни один сегмент не разделяется между процессорами). Заметим, что процессоры просто используют некоторые области памяти (изолированные благодаря концепции сегментов), но не существует никакого волшебного механизма, который отделил бы эти области друг от друга с помощью какой-то поддержки со стороны ОС или оборудования. Однако такая изоляция обеспечивает хорошее использование кешей, т. к. каждый ЦП работает с этими сегментами часто и монопольно.

---

[1] Разделение кучи между ядрами тем не менее может произойти, если по какой-то причине мы сконфигурировали GC так, что куч больше, чем логических процессоров.

У потоков, работающих на ЦП 1 (обозначены T_1 и T_2), контекст выделения памяти находится в SOH_1. Потоки, работающие на втором ЦП (в данном случае поток один, обозначенный T_3), пользуются второй кучей и т. д. В LOH контекста выделения памяти нет, поэтому он не показан.

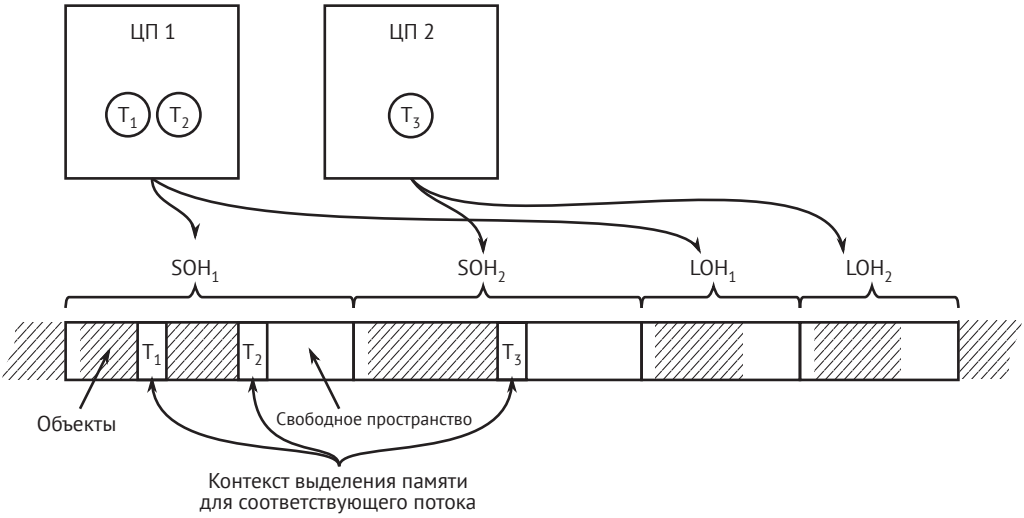


Рис. 6.17 ❖ Взаимосвязи между логическими процессорами, потоками и управляемыми кучами

При создании потока операционная система решает, на каком логическом процессоре он будет исполняться. И все хорошо, если все управляемые потоки приложения выделяют более-менее одинаковое количество памяти. Но бывают ситуации, когда один или несколько потоков начинают выделять гораздо больше памяти, чем остальные. Тогда возникают *несбалансированные кучи*, показанные на рис. 6.18. Потоки 3 и 4 выделяют гораздо больше памяти, чем потоки 1 и 2 (поэтому в SOH_2 осталось намного меньше места). Такая ситуация нежелательна по двум причинам:

- во второй SOH скоро кончится память. Тогда будет запущена сборка мусора, и в конечном итоге, возможно, придется создавать новый сегмент;
- использование кеша процессора не сбалансировано.

Сборщик мусора периодически (во время выделения памяти) проверяет сбалансированность куч. Если он заметит дисбаланс, то переназначит кучу самому активному потоку, т. е. переместит его контекст выделения памяти в другую кучу. Очевидно, это нарушает описанные выше паттерны проектирования, потому что поток, исполняемый одним ядром, будет использовать кучу, назначенную другому ядру. Поэтому GC немедленно просит операционную систему перенести исполнение такого потока на соответствующее куче ядро. В настоящее время подобное поведение поддерживается только в Windows с помощью функции `SetThreadIdealProcessor` (другие операционные системы просто не предоставляют эквивалентного API). В результате ситуация, показанная на рис. 6.18, будет сбалансирована и приведена к виду, показанному на рис. 6.19.

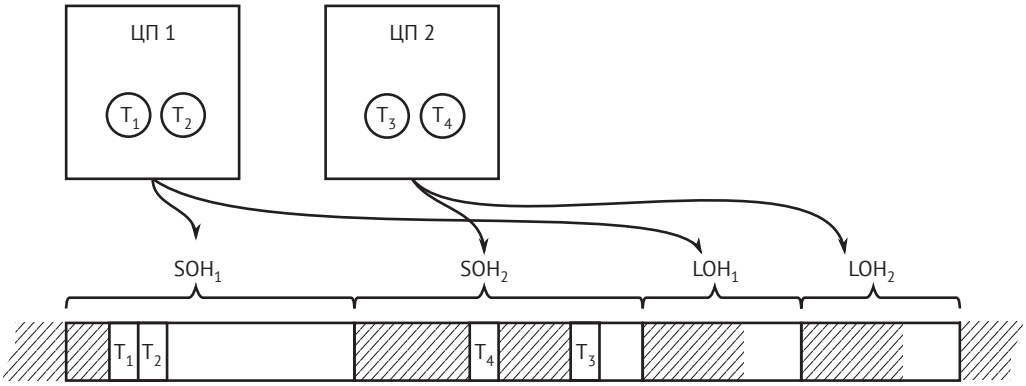


Рис. 6.18 ❖ Несбалансированные кучи в ситуации, когда несколько потоков выделяют гораздо больше памяти, чем все остальные

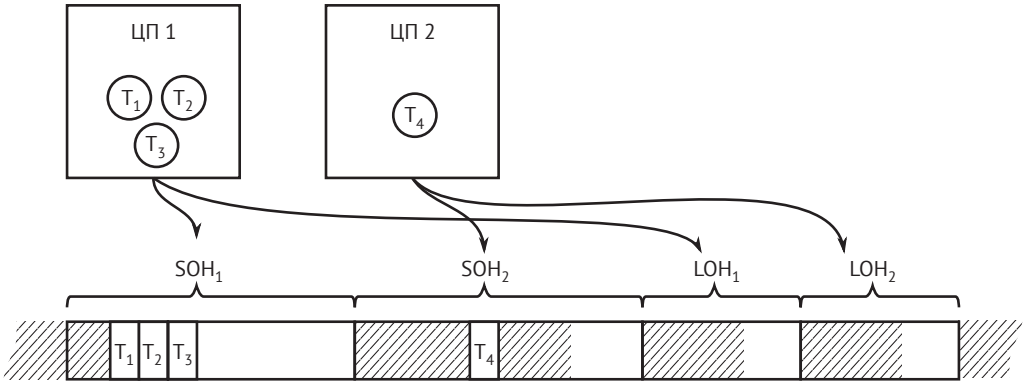


Рис. 6.19 ❖ После балансирования куч, изображенных на рис. 6.18

Начиная с версии .NET 4.5 кучи больших объектов сбалансированы, что существенно повысило производительность выделения памяти. Техника балансирования LOH такая же, как для SOH, поэтому мы не будем на ней останавливаться.

Исключение OutOfMemoryException

При рассмотрении деревьев решений распределителя мы видели, что иногда нет никакой возможности выделить запрошенное количество памяти. Давайте ненадолго задержимся на этом вопросе и обсудим часто возникающие в связи с ним недоразумения.

Прежде всего: когда возникает исключение `OutOfMemoryException`? Поскольку это последнее решение в деревьях на рис. с 6.12 по 6.15, то:

- сборщик мусора уже был запущен, быть может, даже не один раз. Была произведена полная сборка мусора с уплотнением, так что фрагментация SOH не должна быть проблемой. Есть небольшой шанс, что ваша проблема на-

столько нестабильна, что запуск GC еще раз (в дополнение к тем, что были инициированы распределителем) мог бы помочь. Разумеется, исключение `OutOfMemoryException` произошло не потому, что среда выполнения .NET забыла вызвать GC для возвращения памяти. С другой стороны, если `OutOfMemoryException` произошло во время выделения памяти в LOH, то можно попробовать явно запросить уплотнение (как описано в главе 7) и запустить GC еще раз;

- распределитель не смог подготовить область памяти заданного размера. Это могло случиться по двум причинам:
  - виртуальная память закончилась, поэтому распределитель не смог зарезервировать достаточно большой участок (например, чтобы создать новый сегмент). Это может быть связано прежде всего с фрагментацией виртуальной памяти, особенно в 32-разрядной среде. Из-за фрагментации памяти мы не знаем, сколько памяти в действительности было использовано, поэтому если `OutOfMemoryException` происходит в такой ситуации, вполне возможно, что в системе еще много свободной памяти. Вспомните размеры виртуального адресного пространства, показанные в табл. 2.5. 32-разрядная среда выполнения располагает всего 2 или 3 ГБ виртуальной памяти даже в 64-разрядной системе, не испытывающей недостатка в ОЗУ!
  - физическая память (имеется в виду как ОЗУ, так и файл подкачки) закончилась, поэтому распределитель не может выделить достаточное количество памяти (например, чтобы увеличить уже существующий сегмент). Заметим, что операционная система управляет памятью, принимая во внимание все работающие процессы, а не только ваше приложение. Вполне может случиться, что свободная память еще осталась, и вы ее видите, но ваше приложение исчерпало свой лимит (и в ОЗУ, и на диске), поэтому система отклоняет его запросы на передачу дополнительной памяти.

Я хотел бы подчеркнуть два важных вывода, вытекающих из сказанного выше:

- запуск GC вручную вряд ли поможет, если возникло исключение `OutOfMemoryException` (если только это не произошло при выделении памяти для большого объекта, когда имеет смысл самостоятельно запустить уплотнение LOH);
- нормально, если после возникновения `OutOfMemoryException` вы видите, что имеется свободная оперативная память.

Как можно исправить приложение, в котором возникает `OutOfMemoryException`? Попробуйте предпринять следующие шаги:

- создавайте меньше объектов: изучите, как используется память в вашем приложении и можно ли сократить лишние операции выделения. Далее в этой главе мы увидим, что есть много источников выделения памяти, о некоторых из них вы, возможно, даже не подозреваете;
- используйте пул объектов: одно из решений состоит в том, чтобы не выделять память для новых объектов, а повторно использовать существующую. Мы покажем, что для этой цели существуют готовые библиотеки (и вы всегда можете написать свою собственную);
- используйте придерживание виртуальной памяти (VM Hoarding), описанное в главе 5 (особенно в 32-разрядных средах);

- перекомпилируйте приложение для 64-разрядной платформы – этого вполне может оказаться достаточно, поскольку при этом вы получите большое виртуальное адресное пространство.

Сценарий 6.1. Нехватка памяти

Описание. Один из процессов в .NET Core время от времени «падает» в условиях промышленной эксплуатации приложения с исключением `OutOfMemoryException`. В других средах мы не можем воспроизвести эту ситуацию. Это бывает так редко, что подключить «навороченное» средство мониторинга невозможно. Мы хотели бы получить полный дамп и проанализировать потребление памяти, но нельзя предсказать, когда произойдет `OutOfMemoryException`.

Анализ. Есть и хорошая новость: существует возможность автоматически создавать полный дамп памяти в момент возникновения `OutOfMemoryException`! Этот метод работает в Windows и для .NET Framework, и для .NET Core. Нужно выполнить следующие действия:

- с помощью программы `regedit` добавить в раздел реестра `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework` (или установить, если он уже существует) параметр `GCBreakOnOOM` типа `REG_DWORD` со значением `0x2`. Он означает, что при возникновении `OutOfMemoryException` генерируется контрольное исключение (Breakpoint Exception). Это исключение может перехватить программа DebugDiag;
- настройте правило DebugDiag:
  - добавьте новое правило типа Crash;
  - выберите «A specific process» и укажите процесс, который вас интересует;
  - в разделе **Advanced Settings** нажмите кнопку **Exceptions**, затем **Add Exception**;
  - в списке исключений выберите элемент **80000003 Breakpoint Exception**;
  - в списке **Action Type** выберите элемент **Full userdump** и введите в поле **Action limit** значение 1;
  - нажмите кнопку **Save & Close**;
  - задайте имя правила и место для хранения файлов дампов;
  - выберите команду **Activate the rule now** и нажмите кнопку **Finish**;
- начиная с этого момента система будет наблюдать за вашим процессом и создаст полный дамп при возникновении исключения `OutOfMemoryException`;
- когда это наконец случится, у вас будет несколько возможностей проанализировать дамп. Для начала можно открыть его в WinDbg. Прежде всего загрузите расширение SOS, а затем введите команду `analyzeoom`, которая напечатает подробные сведения об исключении `OutOfMemoryException` (листинг 6.9).

Листинг 6.9 ❖ Анализ полного дампа памяти в WinDbg: сведения об исключении `OutOfMemoryException`

```
> .loadby sos coreclr
> !analyzeoom
Managed OOM occurred after GC #4 (Requested to allocate 0 bytes)
Reason: Didn't have enough memory to allocate an LOH segment
Detail: LOH: Failed to reserve memory (50331648 bytes)
```

Дополнительно можно исследовать потоки в момент создания дампа и найти тот, который стал причиной появления `OutOfMemoryException`. Для этого воспользуйтесь командой `threads` и затем `clrstack` (листинг 6.10). Это приведет вас прямо в то место, где произошла ошибка.

Листинг 6.10 ❖ Анализ полного дампа памяти в WinDbg: потоки

```
> !threads
ThreadCount:      3
UnstartedThread:  0
BackgroundThread: 2
PendingThread:    0
DeadThread:       0
Hosted Runtime:   no

                                                                  Lock
  ID  OSID ThreadOBJ   State GC Mode     GC Alloc Context  Domain
        Count Apt Exception
0  1  3a5c 00a09c60    20020 Preemptive  0715D9C8:00000000 00a0c2e0
        0     Ukn  System.OutOfMemoryException 0715d954
2  2 512c  00a9ba78    21220 Preemptive  00000000:00000000 00a0c2e0
        0     Ukn  (Finalizer)
4  3 5660  00aa7758    21220 Preemptive  00000000:00000000 00a0c2e0
        0     Ukn
> ~0s
> !clrstack
OS Thread Id: 0x3a5c (0)
Child SP       IP Call Site
0097ead8 73e008b2 [HelperMethodFrame: 0097ead8]
0097eb5c 06b404bf CoreCLR.LOHWaste.Program.Main(System.String[])
0097ecf0 0f8b926f [GCFrame: 0097ecf0]
0097f004 0f8b926f [GCFrame: 0097f004]
```

Можно продолжить анализ дампа с помощью других инструментов, упомянутых в этой книге, в т. ч. исследовать сегменты и кучи. Имейте в виду, что код, который привел к появлению `OutOfMemoryException`, может и не быть истинной причиной проблемы. Возможно, это всего лишь какой-то поток, который исполнялся в момент, когда распределитель не смог найти место для нового объекта. А сам источник потребления памяти может быть совсем в другом месте. Поэтому имеет смысл внимательно изучить дамп на предмет самых многочисленных объектов, самых больших объектов, их распределения по поколениям и т. д.

Выделение памяти в стеке

До сих пор мы говорили только о создании объектов в управляемой куче. Конечно, это самый популярный и часто встречающийся подход. Мы видели, какие грандиозные усилия предпринимались, чтобы сделать выделение памяти из кучи максимально быстрым. Однако, как мы помним из предыдущих глав, выделение и освобождение памяти в стеке в принципе гораздо быстрее. Требуется лишь сдвинуть указатель стека, не опасаясь никаких накладных расходов со стороны GC.

Как уже было сказано, память для типов значений при определенных условиях выделяется в стеке. Это хорошо, хотя можно и явно запросить выделение памяти в стеке. С учетом правила 14 «Избегайте выделения памяти в куче» это очень полезная возможность.

В C# для явного выделения памяти в стеке служит оператор `stackalloc` (листинг 6.11). Он возвращает указатель на область памяти запрошенного размера, которая будет размещена в стеке. Поскольку мы получаем указатель, этот код должен находиться в небезопасном контексте (если только не используется тип `Span<T>`, как будет показано ниже). Содержимое вновь выделенной области памяти не определено, и никаких предположений (например, что она заполнена нулями) по этому поводу делать нельзя.

Листинг 6.11 ❖ Применение `stackalloc` для явного выделения памяти в стеке

```
static unsafe void Test(int t)
{
   SomeStruct* array = stackalloc SomeStruct[20];
}
```

Оператор `stackalloc` очень редко встречается в коде на C#. В основном это связано с незнанием и недопониманием со стороны программистов. Его можно использовать, когда, например, требуется очень высокая скорость обработки данных, но мы не хотим выделять память для огромных таблиц в куче. У такого решения два преимущества:

- освобождение памяти производится так же быстро, как для любого объекта в стеке: нет никакого метода выделения памяти из кучи, нет медленного пути, GC вообще в этом не участвует;
- адрес такого объекта неявно закреплен (т. е. не может быть перемещен), поскольку кадры стека никогда не перемещаются, а значит, мы можем, ничего не опасаясь, передать указатель на такие данные неуправляемому коду, не понеся при этом накладных расходов на закрепление.

Оператор `stackalloc` соответствует команде CIL `localloc` (листинг 6.12). В стандарте ECMA сказано (часть текста опущена), что она «выделяет `size` байт из локального динамического пула памяти. После возврата из текущего метода место в локальном пуле памяти может быть использовано повторно». Обратите внимание, что здесь ничего явно не говорится о стеке, а употребляется более общее понятие «локальный пул памяти» (оно упоминалось в главе 4). И как мы уже видели в главе 4, стандарт ECMA не зависит от технологий, поэтому понятия стека или кучи в нем нет.

Листинг 6.12 ❖ Часть CIL-кода, сгенерированная для кода из листинга 6.11. Показано, что оператор `stackalloc` транслируется в команду `localloc`

```
IL_0000: ldc.i4.s 10
IL_0002: conv.u
IL_0003: sizeof SomeStruct
IL_0009: mul.ovf.un
IL_000a: localloc
```

Но что именно можно разместить в стеке таким образом? Стандарт ECMA ничего не говорит по этому поводу и лишь обещает, что команда `localloc` выделит ука-

занное число байтов. Поскольку CIL гарантирует лишь получение блока памяти, CLR в настоящее время может использовать его только как контейнер для простых типов данных. В определении оператора `stackalloc` в спецификации языка C# эти ограничения описаны более подробно. Сказано, что допускается только массив элементов «неуправляемого типа». Под *неуправляемыми типами* понимаются:

- примитивные типы: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`;
- любой тип перечисления (enum);
- любой тип указателя;
- любая определенная пользователем структура, не являющаяся конструируемым типом[1] и содержащая только поля *неуправляемых типов*.

Следует помнить, что не существует способа явно освободить память, выделенную оператором `stackalloc`. Она освобождается неявно при выходе из метода. Также следует помнить, что интенсивное использование стека вследствие большого количества долго выполняющихся методов может закончиться исключением `StackOverflowException`.

> Команда `localloc` транслируется JIT-компилятором в последовательность команд `push` и `sub rsp, [size]`, чтобы увеличить размер кадра стека. Величина приращения округляется до 8 или 16 байт в 32- и 64-разрядной версии .NET соответственно. Поэтому даже если вы попросите `stackalloc` выделить место для массива из двух целых чисел (обычно 8 байт), то кадр стека будет расширен на 16 байт (в 64-разрядной версии .NET). Это связано с тем, что в архитектуре x64 стек должен быть выровнен на границу, кратную 16 байтам. Дополнительные сведения смотрите в документации по адресу https://docs.microsoft.com/en-us/cpp/build/stack-allocation.

Как уже отмечалось, мы не обязаны использовать небезопасный код при работе с оператором `stackalloc`. Начиная с версии C# 7.2 и .NET Core 2.1 появился тип `Span<T>` (он очень подробно описан в главе 15), с помощью которого мы можем безопасно писать такой код, как в листинге 6.11.

Листинг 6.13 ❖ Благодаря поддержке `Span<T>` оператор `stackalloc` можно применять в безопасном коде для явного выделения памяти в стеке

```
static void Test(int t)
{
   Span<SomeStruct> array = stackalloc SomeStruct[20];
}
```

Избегание выделения памяти

Уже много было сказано о механизмах выделения памяти. Мы теперь понимаем, что утверждение «выделение памяти обходится дешево» может быть верным благодаря методу сдвига указателя внутри контекста выделения памяти. Но с несколькими оговорками:

- выделение памяти обходится дешево, если удается выбрать быстрый способ для этого. Но иногда в точках, которые программа не может предсказать

[1] Обобщенный тип, включающий аргументы-типы.

заранее, приходится изменять контекст выделения памяти, что становится причиной использования более сложного и медленного способа;
- эти более сложные способы выделения памяти периодически становятся причиной запуска сборщика мусора;
- выделение памяти для больших объектов в LOH медленнее из-за затрат на обнуление памяти;
- чем больше создано объектов, тем больше работы для сборщика мусора – этот факт может показаться очевидным, но он очень важен. Если создается много временных объектов, то все они должны быть удалены сборщиком мусора. И чем больше объектов создается, тем больше шансов нарушить гипотезу о поколениях, относящуюся ко времени жизни объекта.

Из всего сказанного выше следует, что один из самых эффективных способов оптимизации работы с памятью в .NET – вообще избегать выделения или, по крайней мере, точно знать, где и когда выделяется память. Меньше выделений памяти – меньше нагрузка на GC, дешевле доступ к памяти, меньше взаимодействий с операционной системой. Поэтому разработчик для .NET, думающий о производительности, обязательно должен знать источники выделения памяти и способы их устранения или минимизации.

В этом разделе перечислены самые распространенные источники выделения памяти и способы справиться с этой проблемой. Но прошу вас иметь в виду одно очень важное замечание – к вопросу о минимизации выделения памяти следует подходить ответственно и обладая полной информацией на этот счет. Известно популярное высказывание, которое часто цитируют: «преждевременная оптимизация – корень всех зол». Конечно, анализировать каждую строку кода на предмет количества выделений памяти необязательно. Это может парализовать всю работу, не дав ничего взамен. Так ли важно, чтобы строка кода, выполняемая раз в минуту, выделяла 200 байт вместо 800? Пожалуй, нет. Все зависит от требований, предъявляемых к коду. Поэтому начинать имеет смысл с анализа выделения памяти в тех участках программы, которые находятся в критических с точки зрения производительности точках, поскольку это может дать наибольший эффект.

Во-первых, следует узнать о наиболее распространенных источниках выделения памяти, чтобы избежать очевидных ошибок. Или, по крайней мере, знать, насколько требователен к памяти код, который вы пишете в данный момент. Зная контекст всего приложения и требования к этой конкретной части, мы сможем решить, хорошо это или нет. Во-вторых, знать источники выделения будет полезно, когда мы применим (а это обязательно нужно сделать!) правило 2 «Измеряйте GC как можно раньше». Только измерения позволят нам избежать преждевременной оптимизации не там, где это нужно. Лишь с помощью измерений мы сможем понять, нужно ли вообще минимизировать количество выделений памяти. И найти, на какой части кода следует сконцентрировать усилия.

Ниже приведен перечень наиболее часто встречающихся источников выделения памяти. Одни из них очевидны, другие не очень. Наряду с информацией о самих источниках приводятся сведения о том, стоит ли их избегать и как именно.

> Читая описания некоторых механизмов, используемых компилятором C#, полезно видеть, как они преобразуют наш исходный код. Это позволит лучше понять, что происходит под капотом, и проверить себя, если мы не уверены. Для этого мы снова воспользовались замечательной программой dnSpy. Рекомендую поэкспериментировать

с ней, чтобы лучше ориентироваться в излагаемом ниже материале. Попробуйте изменить код, декомпилируйте его и посмотрите, как это влияет на окончательный код, исполняемый средой выполнения.

Явное выделение памяти для ссылочных типов

Большинство случаев выделения очевидны – мы явно создаем объекты. Но это не значит, что к данному источнику следует относиться как к тривиальному. Можно задуматься над тем, действительно ли нам нужен ссылочный тип, который будет создан в куче. Ниже рассмотрен ряд возможных сценариев и предложены решения для них.

Общий случай: рассмотрите возможность использования структуры

Быть может, мы используем классы только потому, что даже не думали об альтернативах. Самые типичные сценарии, когда можно использовать структуры вместо классов, – это передача небольшого количества данных методу в качестве аргументов и возврат этих данных из метода. В листинге 4.7 из главы 4 иллюстрируется такой случай и ясно показано, как можно было бы сгенерировать оптимальный код (см. листинги 4.8 и 4.9), вместо того чтобы создавать небольшой объект в куче. Результаты тестирования производительности в табл. 4.1 показывают существенное различие в производительности двух подходов.

Поэтому имеет смысл подумать об использовании структур для передачи небольшого количества данных методу и получения данных от него в том случае, когда данные локальны (т. е. не хранятся внутри данных, размещенных в куче). На самом деле большая часть бизнес-логики соответствует этим требованиям: мы получаем какие-то данные, обрабатываем их локально и возвращаем результат. Пример в листинге 6.14 возвращает полные имена всех работников, проживающих в пределах указанного расстояния от указанного места. Демонстрируется типичное использование коллекции, полученной от внешней службы (или из внешнего репозитория). Однако при этом явно создается множество объектов:

- список объектов типа `PersonDataClass` и сами объекты `PersonDataClass`;
- объект, представляющий сотрудника, который возвращает внешняя служба.

Листинг 6.14 ❖ Пример простой бизнес-логики, основанный исключительно на классах

```
[Benchmark]
public List<string> PeopleEmployeedWithinLocation_Classes(int amount,
      LocationClass location)
{
   List<string> result = new List<string>();
   List<PersonDataClass> input = service.GetPersonsInBatchClasses(amount);
   DateTime now = DateTime.Now;
   for (int i = 0; i < input.Count; ++i)
   {
      PersonDataClass item = input[i];
      if (now.Subtract(item.BirthDate).TotalDays > 18 * 365)
      {
         var employee = service.GetEmployeeClass(item.EmployeeId);
```

```
        if (locationService.DistanceWithClass(location, employee.Address) < 10.0)
        {
           string name = string.Format("{0} {1}", item.Firstname, item.Lastname);
           result.Add(name);
        }
     }
  }
  return result;
}
internal List<PersonDataClass> GetPersonsInBatchClasses(int amount)
{
  List<PersonDataClass> result = new List<PersonDataClass>(amount);
  // Заполнить список данными из внешнего источника
  return result;
}
```

А что, если переписать код из листинга 6.14, используя везде, где возможно, структуры? На самом деле данные о людях и сотрудниках не выходят за пределы метода `PeopleEmployeedWithinLocation_Classes`, поэтому их можно безопасно хранить в стеке как структуры (листинг 6.15). Метод `GetPersonsInBatch` может вернуть массив структур, при этом улучшается локальность данных и снижаются накладные расходы (см. главу 4). Внешняя служба, такая как метод `GetEmployeeStruct`, может возвращать небольшие структуры, а не объекты классов. Она и аргументы типа значений может получать по ссылке (как метод `DistanceWithStruct`), чтобы явно избежать копирования.

Листинг 6.15 ❖ Пример простой бизнес-логики, в котором везде, где возможно, используются структуры

```
[Benchmark]
public List<string> PeopleEmployeedWithinLocation_Structs(int amount,
     LocationStruct location)
{
  List<string> result = new List<string>();
  PersonDataStruct[] input = service.GetPersonsInBatchStructs(amount);
  DateTime now = DateTime.Now;
  for (int i = 0; i < input.Length; ++i)
  {
     ref PersonDataStruct item = ref input[i];
     if (now.Subtract(item.BirthDate).TotalDays > 18 * 365)
     {
        var employee = service.GetEmployeeStruct(item.EmployeeId);
        if (locationService.DistanceWithStruct(ref location, employee.Address)
           < 10.0)
        {
           string name = string.Format("{0} {1}", item.Firstname, item.Lastname);
           result.Add(name);
        }
     }
  }
  return result;
}
```

```
internal PersonDataStruct[] GetPersonsInBatchStructs(int amount)
{
   PersonDataStruct[] result = new PersonDataStruct[amount];
   // Заполнить список данными из внешнего источника
   return result;
}
```

Кажется, что код в листинге 6.15 менее красивый, чем в листинге 6.14? Ну, может быть, самую малость из-за передачи по ссылке (и использования ссылочной локальной переменной, объясняемой в главе 14). Впрочем, это дело вкуса. Код в листинге 6.15 по-прежнему удобочитаем и не нуждается в комментариях. А взамен мы получили вполне ощутимую разницу в объеме выделенной памяти и, стало быть, количество запусков сборки мусора (см. табл. 6.2). Код с использованием структур выделяет в куче вполовину меньше памяти, чем код с использованием классов. Если он вызывается часто, выигрыш может быть весьма значительным!

Таблица 6.2. Результаты тестирования производительности кода в листингах 6.14 и 6.15. Предполагается, что значение `amount` ***равно 1000 (обрабатывается тысяча объектов или структур)***

Метод	Среднее время	Поколение 0	Выделено
PeopleEmployeedWithinLocation_Classes	348,8 мкс	15,1367	62,60 КБ
PeopleEmployeedWithinLocation_Structs	344,7 мкс	9,2773	39,13 КБ

Кортежи: используйте вместо них структуру ValueTuple

Часто возникает необходимость вернуть или передать в качестве аргумента очень простую структуру данных, содержащую всего несколько полей. Если этот тип встречается только один раз, то возникает искушение использовать кортеж или анонимный тип, вместо того чтобы определять класс (листинг 6.16). Однако следует иметь в виду, что и тип `Tuple`, и анонимные типы являются ссылочными, поэтому размещаются в куче.

Листинг 6.16 ❖ Кортежи и анонимные типы, используемые всего один раз

```
var tuple1 = new Tuple<int, double>(0, 0.0);
var tuple2 = Tuple.Create(0, 0.0);
var tuple3 = new {A = 1, B = 0.0};
```

Как следует из предыдущего раздела, в таком случае стоит рассмотреть возможность создания пользовательских структур. Однако в версии C# 7.0 появился новый тип значений, называемый *кортежем значений* (value tuple) и представленный структурой `ValueTuple` (листинг 6.17). Он может служить отличной заменой вышеупомянутым классам и в некоторых случаях избавляет нас от необходимости определять собственные структуры.

Листинг 6.17 ❖ Кортежи значений, появившиеся в C# 7.0

```
var tuple4 = (0, 0.0);
var tuple5 = (A: 0, B: 0.0);
tuple5.A = 3;
```

Чаще всего они используются для возврата нескольких значений из метода. Обычно мы создаем для этой цели кортеж Tuple (или специальный класс) и помещаем в него результаты метода (см. метод ProcessData1 в листинге 6.18). Но можно вместо этого воспользоваться кортежем значений, который содержит другие структуры (см. метод ProcessData2 в листинге 6.18).

Листинг 6.18 ❖ Сравнение кортежа значений и класса Tuple для возврата нескольких значений из метода

```
public static Tuple<ResultDesc, ResultData> ProcessData1(IEnumerable<SomeClass> data)
{
   // Какая-то обработка
   return new Tuple<ResultDesc, ResultData >(new ResultDesc() { ... },
                                             new ResultData() { ... });
   // Или:
   // return Tuple.Create(new ResultDesc() { ... }, new ResultData() {
   // Average = 0.0, Sum = 10.0 });
}

public static (ResultDescStruct, ResultDataStruct) ProcessData2(IEnumerable
      <SomeClass> data)
{
   // Какая-то обработка
   return (new ResultDescStruct() { ... }, new ResultDataStruct() { ... });
}

public class ResultDesc
{
   public int Count;
}

public class ResultData
{
   public double Sum;
   public double Average;
}

public struct ResultDescStruct
{
   public int Count;
}

public struct ResultDataStruct
{
   public double Sum;
   public double Average;
}
```

Это может заметно снизить накладные расходы при возврате нескольких значений из метода (см. табл. 6.3). Поскольку используются только структуры, в методе ProcessData2 нет вообще ни одного выделения памяти! И работает он в два раза быстрее.

Таблица 6.3. Результаты тестирования производительности кода в листинге 6.18

Метод	Среднее время	Выделено
ProcessData1	11,326 нс	88 Б
ProcessData2	5,207 нс	0 Б

У кортежей значений есть еще одна приятная особенность – *деконструкция*. Она позволяет разобрать на составные части кортеж, возвращенный из метода, прямо на месте. Есть также возможность явно *отбросить* часть элементов кортежа, не представляющих для нас интереса (рис. 6.15). Это бывает полезно, потому что компилятор и JIT-компилятор могут использовать эту информацию для дополнительной оптимизации структур.

Листинг 6.19 ❖ Деконструкция кортежа с отбрасыванием

```
(ResultDescStruct desc, _) = ProcessData2(list);
```

> Планируются и, возможно, скоро появятся изменения в системах объектно-реляционного отображения (ORM), которые дадут возможность представить результаты запроса к базе данных в виде кортежей значений и структур. Тогда работать с ними станет гораздо удобнее. Следите за анонсами ORM или проголосуйте за такие изменения!

Небольшие временные локальные данные: подумайте об использовании stackalloc

Мы уже показали, что использование структур вместо классов может принести ощутимые преимущества при работе с локальными временными данными. Вместо списка объектов можно создать массив структур. Но помните, что массив структур все равно размещается в куче, и единственное, что мы выигрываем, – плотность упаковки данных. Однако можно пойти еще дальше и вообще избавиться от выделения памяти в куче, воспользовавшись оператором `stackalloc`.

Рассмотрим простой метод, который принимает список объектов, преобразует его во временный список и обрабатывает последний для сбора какой-то статистики. Типичный подход с применением LINQ показан в листинге 6.20, и надо полагать, вы сможете обобщить его на более сложные случаи. Этот метод производит много выделений памяти для целого списка временных объектов.

Листинг 6.20 ❖ Пример простой обработки списка, основанный исключительно на классах

```
public double ProcessEnumerable(List<BigData> list)
{
   double avg = ProcessData1(list.Select(x => new DataClass()
   {
      Age = x.Age,
      Sex = Helper(x.Description) ? Sex.Female : Sex.Male
   }));
   _logger.Debug("Результат: {0}", avg / _items);
   return avg;
}

public double ProcessData1(IEnumerable<DataClass> list)
```

```
{
   // Обработать элементы списка
   return result;
}

public class BigData
{
   public string Description;
   public double Age;
}
```

Здесь, как и в предыдущих примерах, можно было бы использовать массив структур. Однако мы вместо этого воспользуемся оператором `stackalloc` в сочетании со `Span<T>`, чтобы не вводить в код небезопасные участки.

Листинг 6.21 ❖ Пример простой обработки списка, основанный на структурах и stackalloc

```
public double ProcessStackalloc(List<BigData> list)
{
   // Опасно!
   Span<DataStruct> data = stackalloc DataStruct[list.Count];
   for (int i = 0; i < list.Count; ++i)
   {
      data[i].Age = list[i].Age;
      data[i].Sex = Helper(list[i].Description) ? Sex.Female : Sex.Male;
   }
   double result = ProcessData2(new ReadOnlySpan<DataStruct>(data));
   return result;
}

// Передать Span только для чтения, чтобы сразу было видно, что модифицировать его не следует
public double ProcessData2(ReadOnlySpan<DataStruct> list)
{
   // Обработать элемент списка list[i]
   return result;
}
```

Новый вариант кода меняет все (см. табл. 6.4). Он вообще не выделяет память в куче и работает почти в четыре раза быстрее! Безусловно, этот подход стоит рассмотреть, если подобный код находится в критическом участке программы.

Таблица 6.4. Результаты тестирования производительности кода из листингов 6.20 и 6.21, обрабатывается 100 элементов

Метод	Среднее время	Выделено
ProcessEnumerable	2206,6 нс	3272 Б
ProcessStackalloc	542,9 нс	0 Б

Но имейте в виду, что `stackalloc` следует использовать для небольших буферов (порядка 1 КБ). Главная опасность при его использовании – исключение `StackOverflowException`, которое может произойти, если в стеке недостаточно места. Это одно из неперехватываемых исключений, которые могут привести к аварийному завершению приложения без возможности что-то исправить. Таким образом, за-

водить слишком большие буферы рискованно, и именно поэтому строка, содержащая stackalloc, является опасной (см. комментарий в листинге).

Выделять память для большого объема данных в стеке не очень хорошо и с точки зрения производительности, потому что это приведет к попаданию большого количества страниц в рабочий набор (working set) потока (что сопровождается отказами страниц (page fault)). Но эти страницы не разделяются с другими потоками, поэтому такой подход очень расточителен.

Если вы собираетесь использовать `stackalloc` и хотите быть на 100 % уверены, что исключения `StackOverflowException` не будет, может возникнуть соблазн воспользоваться одним из методов – `RuntimeHelpers.TryEnsureSufficientExecutionStack()` или `RuntimeHelpers.EnsureSufficientExecutionStack()`. В документации сказано, что оба они «гарантируют, что в стеке будет достаточно места для выполнения средней функции .NET». Текущее значение размера стека равно 128 КБ для 64-разрядной среды и 64 КБ для 32-разрядной. Иными словами, если `RuntimeHelpers.TryEnsureSufficientExecutionStack()` возвращает `true`, то, вероятно, будет безопасно выделить в стеке буфер размером не более 128 КБ с помощью `stackalloc`. «Вероятно» – потому что значения эти являются деталями реализации и не гарантируются, сказано лишь о «средней функции .NET Framework», которая, надо думать, не подразумевает вызова `stackalloc` с большим размером буфера. То есть использовать `stackalloc` безопасно только для выделения маленьких буферов (вышеупомянутое значение 1 КБ представляется разумным).

Создание массивов: используйте ArrayPool

В табл. 6.2 мы уже видели, что работа с временными массивами структур, а не с коллекциями объектов может дать ощутимый выигрыш. Но каждый раз создавать массив структур накладно – с точки зрения как быстродействия, так и работы с памятью. Особенно это заметно, когда буферы большие. В таких случаях лучшим решением будет использование пула предварительно созданных объектов. Именно для этого и предназначен класс `ArrayPool` (находится в пакете `System.Buffers`) – пул повторно используемых управляемых массивов.

Этот класс управляет массивами заданного типа различной длины, сгруппированными в кластеры. Допускаются массивы ссылочных типов и типов значений. Для типов значений пул массивов может быть более эффективен, поскольку в нем сохраняется как сам массив, так и его элементы.

По умолчанию каждый из 17 кластеров в объекте `ArrayPool` содержит массивы, размер которых в два раза больше, чем в предыдущем кластере, а именно: 16, 32, 64, 128, 256, 512, 1,024, 2048, 4096, 8192, 16 384, 32 768, 65 536, 131 072, 262 144, 524 288 и 1 048 576. Заметим, что эти массивы создаются по требованию, поэтому нет необходимости выделять такое количество памяти сразу.

Такой пул массивов по умолчанию доступен в виде статистического объекта типа `ArrayPool<T>.Shared`. Если нам нужен массив, мы вызываем метод `Rent` этого объекта. А когда необходимость в нем отпадет, возвращаем его в пул методом `Return` (листинг 6.22).

Листинг 6.22 ❖ Пример использования ArrayPool

```
var pool = ArrayPool<int>.Shared;
int[] buffer = pool.Rent(minLength);
```

```
try
{
   Consume(buffer);
}
finally
{
   pool.Return(buffer);
}
```

Заметим, что метод Rent гарантирует, что длина арендованного массива будет не меньше указанной. Скорее всего, она будет даже больше из-за округления до размера ближайшего кластера.

> Объект ArrayPool<T>.Shared на самом деле имеет тип класса TlsOverPerCoreLockedStacksArrayPool<T>, в котором используется довольно сложная техника кеширования – есть небольшой локальный кеш потока массивов каждого размера и кеш, общий для всех массивов, разбитый на стеки, привязанные к каждому ядру (отсюда и название). Мы ненадолго вернемся к этому вопросу, когда будем обсуждать локальную память потока (TLS) в главе 13.

Теперь немного изменим пример PeopleEmployeedWithinLocation_Structs в листинге 6.15, воспользовавшись пулом ArrayPool. На этот раз вместо создания массива для каждого запроса будем арендовать массив из пула по умолчанию.

Листинг 6.23 ❖ Пример реализации простой бизнес-логики с помощью структур и пула ArrayPool

```
public List<string> PeopleEmployeedWithinLocation_ArrayPoolStructs(int amount,
LocationStruct location)
{
   List<string> result = new List<string>();
   PersonDataStruct[] input = service.GetDataArrayPoolStructs(amount);
   DateTime now = DateTime.Now;
   for (int i = 0; i < amount; ++i)
   {
      ref PersonDataStruct item = ref input[i];
      if (now.Subtract(item.BirthDate).TotalDays > Constants.MaturityDays)
      {
         var employee = service.GetEmployeeStruct(item.EmployeeId);
         if (locationService.DistanceWithStruct(ref location, employee.Address) <
Constants.DistanceOfInterest)
         {
            string name = string.Format("{0} {1}", item.Firstname, item.Lastname);
            result.Add(name);
         }
      }
   }
   ArrayPool<InputDataStruct>.Shared.Return(input);
   return result;
}

internal PersonDataStruct[] GetDataArrayPoolStructs(int amount)
{
   PersonDataStruct[] result = ArrayPool<PersonDataStruct>.Shared.
```

```
    Rent(amount);
    // Заполнить массив данными из внешнего источника
    return result;
}
```

Сравнение этого кода с кодом из листингов 6.14 (коллекция объектов) и 6.15 (создание массивов структур) показывает, какой выигрыш можно получить от использования `ArrayPool` (см. табл. 6.5). Новая версия выделяет лишь 3,5 % памяти от того объема, который нужен в варианте с массивами (и во время тестирования производительности не произошло ни одной сборки мусора). Это может оказаться очень важным в ситуации, когда на потребление памяти наложены жесткие ограничения. Вспомните, что все эти килобайты пришлось бы освобождать сборщику мусора!

Таблица 6.5. Результаты тестирования производительности кода из листингов 6.14, 6.15 и 6.23. Предполагается, что значение `amount` ***равно 1000 (обрабатывается тысяча объектов или структур)***

Метод	Среднее время	Поколение 0	Выделено
PeopleEmployeedWithinLocation_Classes	348,8 мкс	15,1367	62,60 КБ
PeopleEmployeedWithinLocation_Structs	344,7 мкс	9,2773	39,13 КБ
PeopleEmployeedWithinLocation_ArrayPoolStructs	343.4 мкс	–	1,35 КБ

> Результаты в табл. 6.5 интересны, но следует помнить, что они могут вводить в заблуждение, поскольку искусственные тесты на производительность методов не всегда отражают поведение реальных программ. Например, если одновременно выполняются сотни таких операций, то лишь малой части из них действительно удастся получить массив из пула, остальным придется сначала потратить время на поиск в пуле, а затем все-таки пойти по резервному пути – выделить память для массива из кучи. Показанные результаты следует рассматривать как «лучший случай», понимая, что в реальной многопоточной программе выигрыш может быть не столь впечатляющим.

Класс `ArrayPool` может быть выбором по умолчанию, если программа часто работает с большими буферами. Вместо того чтобы снова и снова выделять для них память, используйте их повторно. Количество библиотек, поддерживающих `ArrayPool`, постоянно растет (и, как мы уже говорили, он используется во многих местах стандартной библиотеки .NET). В качестве примера назовем очень популярную библиотеку Json.NET. Можно использовать ее стандартным образом, применяя классы `JsonTextReader` или `JsonTextWriter` (листинг 6.24). Но начиная с версии 8.0 Json.NET поддерживает пулы массивов, поэтому мы можем задать реализацию интерфейса `IArrayPool`, основанную на классе `ArrayPool` (см. класс `JsonArrayPool` в листинге 6.25).

Листинг 6.24 ❖ Пример стандартного использования библиотеки Json.NET

```
public IList<int> ReadPlain()
{
    IList<int> value;
    JsonSerializer serializer = new JsonSerializer();
    using (JsonTextReader reader = new JsonTextReader(new StringReader(Input)))
    {
        value = serializer.Deserialize<IList<int>>(reader);
```

```
      return value;
   }
}
```

Листинг 6.25 ❖ Пример использования ArrayPool в библиотеке Json.NET

```
public int[] ReadWithArrayPool()
{
   JsonSerializer serializer = new JsonSerializer();
   using (JsonTextReader reader = new JsonTextReader(new StringReader(Input)))
   {
      // читатель получает буфер из пула массивов
      reader.ArrayPool = JsonArrayPool.Instance;
      var value = serializer.Deserialize<int[]>(reader);
      return value;
   }
}

public class JsonArrayPool : IArrayPool<char>
{
   public static readonly JsonArrayPool Instance = new JsonArrayPool();
   public char[] Rent(int minimumLength)
   {
      // получить массив char из разделяемого пула System.Buffers
      return ArrayPool<char>.Shared.Rent(minimumLength);
   }

   public void Return(char[] array)
   {
      // вернуть массив char в разделяемый пул System.Buffers
      ArrayPool<char>.Shared.Return(array);
   }
}
```

Предоставив `ArrayPool` сериализатору Json.NET, можно существенно уменьшить количество выделений памяти (см. табл. 6.6). Заметим, что этот буфер используется внутри Json.NET для хранения массива символов. В настоящее время невозможно десериализовать что-то в массив (`int[]` в нашем примере), хотя это тоже было бы весьма желательно.

Таблица 6.6. Результаты тестирования производительности кода из листингов 6.24 и 6.25

Метод	Среднее время	Выделено
`ReadPlain`	14,58 мкс	6,10 КБ
`ReadWithArrayPool`	13,37 мкс	4,42 КБ

Сделаем одно важное замечание. Экземпляр `ArrayPool<T>` можно создать также методом `ArrayPool<T>.Create(int maxArrayLength, int maxArraysPerBucket)`, у него будет тип `ConfigurableArrayPool<T>`. Его реализация несколько проще; она тоже основана на кластерах, но без использования локальной памяти потока. Зато, как видно из сигнатуры метода `Create`, можно самостоятельно задать количество массивов в каждом кластере и максимальный размер кешированного массива (так что ко-

личество кластеров вычисляется автоматически). По умолчанию максимальная длина массива в таком пуле равна 1024*1024 (1 048 576), и в каждом кластере находится 50 массивов.

Если в программе применяется `ArrayPool` (разделяемый или созданный вручную), стоит последить за тем, как он используется с помощью поставщика ETW `System.Buffers.ArrayPoolEventSource`. Например, можно собирать данные в PerfView. При задании свойств коллекции в диалоговом окне **Collect** введите в поле **Additional Providers** значение:

- `*System.Buffers.ArrayPoolEventSource` – если хотите собирать только данные событий;
- `*System.Buffers.ArrayPoolEventSource:::@StacksEnabled=true` – если хотите также собирать стеки вызовов событий.

При этом мы сможем увидеть все операции аренды и выделения массивов (рис. 6.20). Особенно интересно событие `BufferAllocated` с причиной `OverMaximumSize` (размер больше максимального) или `PoolExhausted` (пул исчерпан). Если они происходят часто, то текущая конфигурация `ArrayPool`, вероятно, не отвечает потребностям программы. Если часто встречается `OverMaximumSize`, то слишком мал заданный максимальный размер массива. А в случае `PoolExhausted` стоит увеличить количество массивов в кластере. Имеется также причина `Pooled`, но она встречается только для пула типа `ConfigurableArrayPool` и означает, что в кластере пришлось выделить память для нового массива.

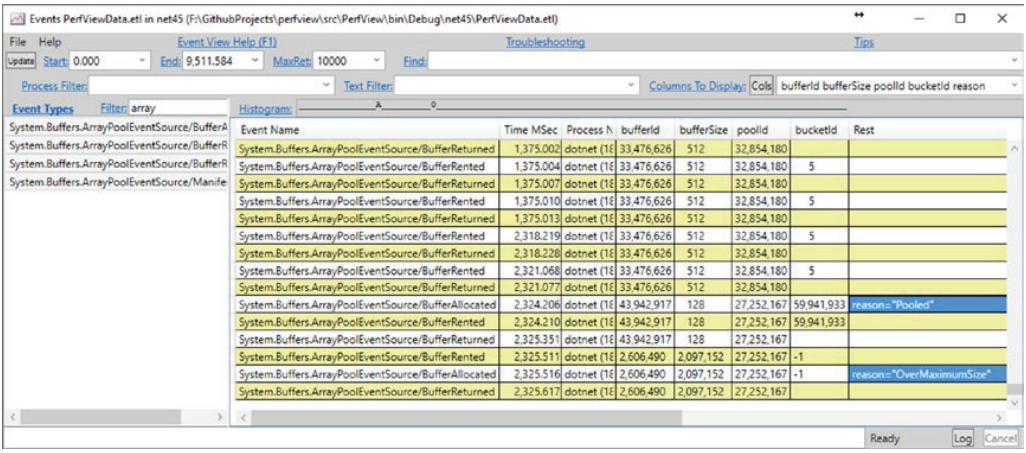


Рис. 6.20 ❖ PerfView – просмотр событий ETW, генерируемых классом ArrayPool

При работе с `ArrayPool` есть один подводный камень. Напомним, что массивы в пуле живут вечно – никакой механизм тайм-аута не предусмотрен. Это хорошо, если массивы используются постоянно на протяжении всего времени работы программы. Но если это кратковременное явление, то вы только увеличите размер рабочего набора (working set), не получив почти никаких преимуществ. Поэтому, размышляя о том, стоит ли воспользоваться классом `ArrayPool`, принимайте во внимание «коэффициент повторного использования».

Имейте в виду, что ArrayPool – одно из активно разрабатываемых усовершенствований в .NET Core (например, механизм был значительно улучшен при переходе от версии .NET Core 2.0 к 2.1). И хотя приведенное здесь общее описание не изменится, детали реализации могут поменяться. В частности, в будущем может появиться механизм вытеснения из пула по тайм-ауту.

Создание потоков: класс RecyclableMemoryStream

Если в вашем приложении часто используется класс System.IO.MemoryStream, то стоит подумать о пуле таких объектов. Этот механизм реализован в классах RecyclableMemoryStream и RecylableMemoryStreamManager из пакета Microsoft.IO.RecyclableMemoryStream. В комментариях к этим классам ясно написано, что интенсивное использование MemoryStream чревато следующими нежелательными эффектами:

- выделение памяти в LOH – поскольку размер внутренних буферов MemoryStream велик, они размещаются в LOH, что дорого с точки зрения выделения и освобождения памяти;
- расточительное использование памяти, т. к. размер внутреннего буфера MemoryStream удваивается, если в нем не хватает памяти. Это приводит к постоянному росту потребления памяти;
- копирование памяти: при каждом увеличении буфера MemoryStream все байты копируются из него в новый буфер, что влечет за собой интенсивное обращение к памяти;
- из-за повторного создания внутренних буферов возможна фрагментация.

Класс RecyclableMemoryStream предназначен для решения этих проблем. Процитируем комментарий к коду этого класса: «Поток реализован как надстройка над последовательностью блоков одинакового размера. По мере роста потока у диспетчера памяти запрашиваются дополнительные блоки. В пул помещаются именно эти блоки, а не сам объект потока.

Проблема этой реализации – в вызове метода GetBuffer(). Он хочет получить один непрерывный буфер. Если используется только один блок, то он и возвращается. Если же используется несколько блоков, то мы запрашиваем буфер большего размера у диспетчера памяти. Эти крупные буферы тоже сохраняются в пуле и распределяются по размеру – все они кратны размеру порции (chunk), которая по умолчанию 1 МБ».

В листинге 6.26 показано стандартное применение класса MemoryStream для сериализации объекта. Помимо создания объектов XmlWriter и DataContractSerializer (которые должны кешироваться), создается также новый объект MemoryStream. Это может приводить к вышеупомянутым проблемам, если сериализуемые объекты велики, а сериализация производится часто.

Листинг 6.26 ❖ Пример XML-сериализации с помощью DataContractSerializer и MemoryStream

```
public string SerializeXmlWithMemoryStream(object obj)
{
   using (var ms = new MemoryStream())
   {
      using (var xw = XmlWriter.Create(ms, XmlWriterSettings))
      {
         var serializer = new DataContractSerializer(obj.GetType());
```

```
        // можно было бы кешировать!
        serializer.WriteObject(xw, obj);
        xw.Flush();
        ms.Seek(0, SeekOrigin.Begin);
        var reader = new StreamReader(ms);
        return reader.ReadToEnd();
      }
   }
}
```

Если потоки используются часто, то следует рассмотреть класс `RecyclableMemoryStream` (листинг 6.27). Сначала нужно создать объект `RecyclableMemoryStreamManager`, а затем можно обращаться к его методу `GetStream`, чтобы получить поток из пула. Полученный поток реализует интерфейс `IDisposable` таким образом, что занятая им память возвращается в пул в момент уничтожения объекта. Конструктору диспетчера можно передать параметры (в листинге 6.27 показаны значения по умолчанию):

- `blockSize` – размер блока, помещаемого в пул;
- `largeBufferMultiple` – размер крупного буфера будет кратен этой величине;
- `maximumBufferSize` – буферы большего размера в пул не помещаются.

Листинг 6.26 ❖ Пример XML-сериализации с помощью `DataContractSerializer` и `RecyclableMemoryStream`

```
static RecyclableMemoryStreamManager manager =
   new RecyclableMemoryStreamManager(blockSize: 128 * 1024,
                                     largeBufferMultiple: 1024 * 1024,
                                     maximumBufferSize: 128 * 1024 * 1024);
public string SerializeXmlWithRecyclableMemoryStream<T>(T obj)
{
   using (var ms = manager.GetStream())
   {
      using (var xw = XmlWriter.Create(ms, XmlWriterSettings))
      {
         var serializer = new DataContractSerializer(obj.GetType());
         // можно было бы кешировать!
         serializer.WriteObject(xw, obj);
         xw.Flush();
         ms.Seek(0, SeekOrigin.Begin);
         var reader = new StreamReader(ms);
         return reader.ReadToEnd();
      }
   }
}
```

За тем, как используется `RecyclableMemoryStream`, рекомендуется следить с помощью поставщика ETW `Microsoft-IO-RecyclableMemoryStream`. Данные можно собирать в PerfView. При задании свойств коллекции в диалоговом окне **Collect** введите в поле **Additional Providers** значение:

- `*Microsoft-IO-RecyclableMemoryStream` – если хотите собирать только данные событий;
- `*Microsoft-IO-RecyclableMemoryStream:::@StacksEnabled=true` – если хотите также собирать стеки вызовов событий.

ПРИМЕЧАНИЕ В экспериментах с `RecyclableMemoryStream` я обнаружил, что задание поставщика ETW по имени работает некорректно. Пришлось идентифицировать его Guid'ом. Так что, возможно, вам придется набрать в поле **Additional Provider** `B80CD4E4-890E-468D-9CBA-90EB7C82DFC7` вместо `*Microsoft-IO-RecyclableMemoryStream`.

Класс `RecyclableMemoryStream` может предоставить весьма детальную информацию об использовании своего пула (рис. 6.21). Особенно интересно событие `MemoryStreamOverCapacity`, которое возникает, когда запрашивается буфер больше максимального размера.

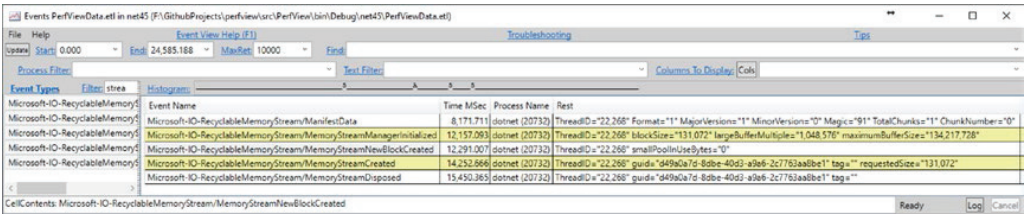


Рис. 6.21 ❖ Просмотр событий ETW, генерируемых классом RecyclableMemoryStream с помощью PerfView

ПРИМЕЧАНИЕ Если потоки используются интенсивно, то подумайте об API `System.IO.Pipelines`. Он предлагает гораздо более эффективную замену потокам с существенно меньшим количеством операций выделения памяти. Подробно этот API описывается в главе 14.

Создание большого числа объектов: используйте пул объектов

Как и в случае коллекций, при интенсивном использовании объектов какого-то типа можно подумать о создании пула таких объектов. Но имейте в виду, что создание большого числа объектов только для кратковременного использования не противоречит гипотезе о поколениях. Поэтому, может быть, и не стоит ничего предпринимать. Сборщик мусора уберет их из поколения 0 очень быстро. Пул объектов следует рассматривать в одной из следующих ситуаций:

- объекты создаются на таком критически важном участке кода, что в счет идет каждый такт процессора – в таком случае предотвращение выделения памяти (особенно его медленного способа) посредством использования более предсказуемого механизма может дать выигрыш. Хорошо написанный пул объектов должен задействовать кеш процессора, поэтому получение объектов из пула должно происходить очень быстро;
- объекты настолько велики, что стоит подумать о стоимости выделения памяти – в таком случае хотелось бы избежать накладных расходов на обнуление (особенно для объектов в LOH). К тому же, помимо собственно выделения, нас может беспокоить стоимость инициализации объекта: если инициализировать его поля очень сложно, то лучше не создавать все новые и новые объекты, а повторно использовать уже инициализированный (если контекст позволяет).

Написать хороший пул объектов – нетривиальная задача. Она могла быть проще, если бы мы ограничились только однопоточным окружением. Но сделать пул потокобезопасным, избежав накладных расходов на синхронизацию, совсем не-

легко. Тривиальная реализация может навредить производительности больше, чем выделение памяти, которое она призвана сократить. В листинге 6.28 показана хорошо протестированная реализация, основанная на замечательном классе `ObjectPool` из исходного кода компилятора C# Roslyn (с сохранением оригинальных комментариев, объясняющих детали, относящиеся к производительности).

Листинг 6.28 ❖ Реализация пула объектов на основе класса ObjectPool, входящего в код компилятора Roslyn

```
public class ObjectPool<T> where T : class
{
   private T firstItem;
   private readonly T[] items;
   private readonly Func<T> generator;

   public ObjectPool(Func<T> generator, int size)
   {
      this.generator = generator ?? throw new ArgumentNullException("generator");
      this.items = new T[size - 1];
   }

   public T Rent()
   {
      // PERF: сначала проверяем первый элемент. Если не получается, RentSlow
      // рассмотрит остальные элементы.
      // Заметим, что первое чтение не синхронизируется. Это
      // сделано сознательно.
      // Блокировка производится, только когда есть кандидат. В худшем
      // случае мы пропустим какие-то недавно возвращенные объекты. Ничего
      // страшного.
      T inst = firstItem;
      if (inst == null || inst != Interlocked.CompareExchange
         (ref firstItem, null, inst))
      {
         inst = RentSlow();
      }
      return inst;
   }

   public void Return(T item)
   {
      if (firstItem == null)
      {
         // Здесь блокировка опущена сознательно.
         // В худшем случае два объекта будут сохранены в одном месте.
         // Это крайне маловероятно, но если случится, то один из объектов
         // будет убран в мусор.
         firstItem = item;
      }
      else
      {
         ReturnSlow(item);
      }
   }

   private T RentSlow()
```

```
   {
      for (int i = 0; i < items.Length; i++)
      {
         // Первое чтение не синхронизируется. Это
         // сделано сознательно.
         // Блокировка производится, только когда есть кандидат. В худшем
         // случае мы пропустим какие-то недавно возвращенные объекты. Ничего
         // страшного.
         T inst = items[i];
         if (inst != null)
         {
            if (inst == Interlocked.CompareExchange(ref items[i],
                null, inst))
            {
               return inst;
            }
         }
      }
      return generator();
   }

   private void ReturnSlow(T obj)
   {
      for (int i = 0; i < items.Length; i++)
      {
         if (items[i] == null)
         {
            // Здесь блокировка опущена сознательно.
            // В худшем случае два объекта будут сохранены в одном месте.
            // Это крайне маловероятно, но если случится, то один из объектов
            // будет убран в мусор.
            items[i] = obj;
            break;
         }
      }
   }
}
```

Асинхронные методы, возвращающие Task: используйте ValueTask

С тех пор как в C# 5.0 был введен механизм асинхронности, он превратился чуть ли не в канонический способ программирования. Мы встречаем асинхронный код повсюду. Неплохо бы знать, как он соотносится с потреблением памяти. Взять, к примеру, простой асинхронный код для чтения всего содержимого файла (листинг 6.29). Сначала он проверяет, существует ли файл, и только если да, асинхронно ждет завершения операции.

Листинг 6.29 ❖ Пример асинхронного метода

```
public async Task<string> ReadFileAsync(string filename)
{
   if (!File.Exists(filename))
      return string.Empty;
   return await File.ReadAllTextAsync(filename);
}
```

Наверное, большинство .NET разработчиков уже знают, что ключевое слово async преобразует этот метод в довольно сложный конечный автомат, отвечающий за правильную обработку запланированных шагов при завершении последующих асинхронных действий. В листинге 6.30 представлен код, сгенерированный компилятором для метода ReadFileAsync в листинге 6.29. Метод был трансформирован в код, который запускает конечный автомат, представленный объектом с загадочным названием Program.<ReadFileAsync>d__14. Существует немало хороших описаний этого механизма, поэтому не будем отвлекаться.

Листинг 6.30 ❖ Метод ReadFileAsync из листинга 6.20 после преобразования компилятором

```
[AsyncStateMachine(typeof(Program.<ReadFileAsync>d__14))]
public Task<string> ReadFileAsync(string filename)
{
   Program.<ReadFileAsync>d__14 <ReadFileAsync>d__;
   <ReadFileAsync>d__.filename = filename;
   <ReadFileAsync>d__.<>t__builder = AsyncTaskMethodBuilder<string>.Create();
   <ReadFileAsync>d__.<>1__state = -1;
   AsyncTaskMethodBuilder<string> <>t__builder = <ReadFileAsync>d__.<>t__builder;
   <>t__builder.Start<Program.<ReadFileAsync>d__14>(ref <ReadFileAsync>d__);
   return <ReadFileAsync>d__.<>t__builder.Task;
}
```

Для нас важны следующие факты (которые подтверждаются кодом в листинге 6.31):

- в сгенерированном компилятором коде в листинге 6.30 используются только структуры (включая Program.<ReadFileAsync>d__14 и AsyncTaskMethodBuilder<string>) – это прекрасный пример осознанного применения структур там, где было бы соблазнительно, недолго думая, воспользоваться классами;
- <ReadFileAsync>d__14 – это сгенерированная компилятором структура, представляющая конечный автомат. Она будет упакована, если асинхронная операция не завершится мгновенно (что происходит внутри метода AwaitUnsafeOnCompleted, присутствующего в листинге 6.31)[1], – а в таком случае «состояние» должно покинуть текущий метод, поскольку асинхронная операция может продолжиться не в том потоке, в котором началась. А значит, «состояние» должно оказаться в куче, а не в стеке текущего потока. Однако делать <ReadFileAsync>d__14 структурой все равно имеет смысл, потому что могут существовать часто выбираемые пути, на которых упаковка не производится (см. случай, когда File.Exists возвращает false в листинге 6.31);
- сгенерированная компилятором структура, представляющая конечный автомат, запоминает (захватывает) все необходимые локальные переменные метода (в нашем примере filename) – об этом нужно знать, потому что таким образом мы можем значительно продлить их жизнь, если память для конечного автомата (<ReadFileAsync>d__14) выделяется в куче.

[1] Начиная с версии .NET Core 2.1 это работает по-другому. Структура перемещается в кучу, но как строго типизированное поле класса, а не в упакованном виде.

Листинг 6.31 ❖ Структура, представляющая конечный автомат метода ReadFileAsync из листинга 6.30

```
[CompilerGenerated]
[StructLayout(LayoutKind.Auto)]
private struct <ReadFileAsync>d__14 : IAsyncStateMachine
{
   void IAsyncStateMachine.MoveNext()
   {
      int num = this.<>1__state;
      string result;
      try
      {
         TaskAwaiter<string> awaiter;
         if (num != 0)
         {
            if (!File.Exists(this.filename))
            {
               result = string.Empty;
               goto IL_A4;
            }
            awaiter = File.ReadAllTextAsync(this.filename,
               default(CancellationToken)).GetAwaiter();
            if (!awaiter.get_IsCompleted())
            {
               this.<>1__state = 0;
               this.<>u__1 = awaiter;
               this.<>t__builder.AwaitUnsafeOnCompleted<TaskAwaiter
                 <string>, Program.<ReadFileAsync>d__14>(ref awaiter, ref this);
               return;
            }
         }
         else
         {
            awaiter = this.<>u__1;
            this.<>u__1 = default(TaskAwaiter<string>);
            this.<>1__state = -1;
         }
         result = awaiter.GetResult();
      }
      catch (Exception exception)
      {
         this.<>1__state = -2;
         this.<>t__builder.SetException(exception);
         return;
      }
      IL_A4:
      this.<>1__state = -2;
      this.<>t__builder.SetResult(result);
   }
}
```

Помимо возможных накладных расходов, связанных с размещением конечного автомата в куче, у асинхронных методов есть еще один подвох. Если проследить, что происходит в коде из листинга 6.31 в случае, когда файл не существует, то мы увидим, что после выполнения `goto` вызывается метод `SetResult` структуры `AsyncTaskMethodBuilder<string>`. Теоретически это очень быстрый синхронный путь, на котором нет издержек, обусловленных асинхронным ожиданием. Однако метод `SetResult` выделяет память для объекта `Task`, который будет содержать результат метода (листинг 6.32).

Листинг 6.32 ❖ Структура AsyncTaskMethodBuilder

```
public struct AsyncTaskMethodBuilder<TResult>
{
   public static AsyncTaskMethodBuilder<TResult> Create()
   {
      return default(AsyncTaskMethodBuilder<TResult>);
   }

   public void Start<TStateMachine>(ref TStateMachine stateMachine) where
      TStateMachine : IAsyncStateMachine
   {
      // ...
      stateMachine.MoveNext();
   }

   // ...

   public void SetResult(TResult result)
   {
      Task<TResult> task = this.m_task;
      if (task == null)
      {
         this.m_task = this.GetTaskForResult(result);
         return;
      }
      // ...
   }

   public Task<TResult> Task
   {
      get
      {
         Task<TResult> task = this.m_task;
         if (task == null)
         {
            task = (this.m_task = new Task<TResult>());
         }
         return task;
      }
   }
}
```

Метод `GetTaskForResult`, вызываемый из `SetResult`, скорее всего, выделит память для нового объекта `Task`, оборачивающего результат выполнения, но с некоторыми оптимизациями:

- в случае Task<bool> возвращается один из двух кешированных объектов (для значений true и false);
- в случае Task<int> возвращается кешированный объект для значений от –1 до 9, а для других значений создается новый объект Task;
- в случае многих числовых типов T будет возвращен кешированный объект Task<T>, если значение равно 0;
- для ссылочных типов возвращается кешированный Task, если значение равно null;
- во всех остальных случаях создается новый объект Task.

Неэффективно выделять память для объекта Task только для того, чтобы передать значение результата. Если наш асинхронный метод вызывается очень часто и такой синхронный ответ встречается неоднократно, то мы произведем много ненужных выделений памяти для Task. Именно для этого случая предложена упрощенная версия Task под названием ValueTask. На самом деле это структура, реализованная как размеченное объединение (discriminated union) – тип, который может принимать одно из трех возможных значений (листинг 6.33):

- готовый к использованию результат (если операция успешно завершилась синхронно);
- обычный объект Task, который можно ждать;
- обертка IValueTaskSource<T>, которую могут реализовать произвольные объекты, желающие быть представленными типом ValueTask<T> (доступно только в версии .NET Core 2.1). Затем эти объекты можно поместить в пул и использовать повторно для минимизации выделения памяти.

Листинг 6.33 ❖ Структура ValueTask, появившаяся в C# 7.0 (версия в .NET Core 2.1)

```
public struct ValueTask<TResult>
{
   // null, если _result содержит результат, в противном случае
   // Task<TResult> или IValueTaskSource<TResult>
   internal readonly object _obj;
   internal readonly TResult _result;
}
```

Соответственно метод SetResult класса AsyncValueTaskMethodBuilder<TResult> присваивает результат (если он уже доступен) или просто создает объект Task обычным описанным выше способом (если продолжается обычное асинхронное выполнение). Следовательно, мы можем полностью избежать выделения памяти, когда асинхронный метод получает ответ синхронно. Для этого всего-то и нужно, что изменить тип возвращаемого значения с Task<T> на ValueTask<T> (листинг 6.34). Компилятор позаботится обо всем остальном, выбрав AsyncValueTaskMethodBuilder вместо AsyncTaskMethodBuilder.

Листинг 6.34 ❖ Пример использования ValueTask

```
public async ValueTask<string> ReadFileAsync2(string filename)
{
   if (!File.Exists(filename))
      return string.Empty;
   return await File.ReadAllTextAsync(filename);
}
```

Пользуясь асинхронными методами, возвращающими ValueTask, мы можем просто ждать результата с помощью await как для любого регулярного асинхронного метода. И лишь в самом кратчайшем цикле или на критическом с точки зрения производительности участке программы можно дополнительно проверить, завершился ли уже метод, и если да, то использовать Result (см. листинг 6.35). Поскольку все основано на структурах, то никакого выделения памяти при этом не будет. А если задача еще не завершилась, то она начнет выполняться как обычный Task.

Листинг 6.35 ❖ Использование асинхронного метода, возвращающего ValueTask

```
var valueTask = ReadFileAsync2();
if(valueTask.IsCompleted)
{
   return valueTask.Result;
}
else
{
   return await valueTask.AsTask();
}
```

Возможна еще одна оптимизация. Как уже было сказано, в случае асинхронного пути объект Task все равно должен быть создан. Но если это часто происходит на критическом участке кода, хорошо бы устранить и это выделение памяти. Поэтому и был введен вышеупомянутый интерфейс IValueTaskSource. С тех пор мы можем создавать структуру ValueTask, оборачивающую экземпляр реализации этого интерфейса, что выгодно, если такой экземпляр кешируется или хранится в пуле. Иными словами, асинхронная операция впоследствии будет представлена экземпляром из кеша или пула (листинг 6.36). Поэтому необходимости в выделении памяти для Task не возникает.

Листинг 6.36 ❖ Пример использования ValueTask, поддержанного реализацией IValueTaskSource

```
public ValueTask<string> ReadFileAsync3(string filename)
{
   if (!File.Exists(filename))
      return new ValueTask<string>("!");
   var cachedOp = pool.Rent();
   return cachedOp.RunAsync(filename, pool);
}

private ObjectPool<PooledValueTaskSource> pool =
   new ObjectPool<PooledValueTaskSource>(() => new PooledValueTaskSource (), 10);
```

Для реализации интерфейса IValueTaskSource нужно написать три метода:

- GetResult вызывается один раз, когда асинхронный конечный автомат должен получить результат операции;
- GetStatus вызывается асинхронным конечным автоматом для проверки состояния операции;

- `OnCompleted` вызывается асинхронным конечным автоматом, когда ожидается обертывание `ValueTask`. Здесь мы должны запомнить продолжение (continuation), которое следует вызвать, когда операция завершится (но если она уже завершилась, то нужно вызвать продолжение сразу).

Дополнительно для удобства такой тип должен предоставить методы для запуска операции и для реагирования на завершение операции.

> Но следует понимать, что полнофункциональная и потокобезопасная реализация интерфейса `IValueTaskSource` – далеко не тривиальное занятие. Места в книге не хватило бы, чтобы включить полную реализацию `PooledValueTaskSource` (использованную в листинге 6.36) со всеми необходимыми пояснениями. К тому же в действительности найдется немного разработчиков, которым нужно реализовывать этот интерфейс. Впрочем, полная реализация `PooledValueTaskSource` (с развернутыми комментариями) есть в сопроводительном репозитории на Github и в блоге по адресу http://tooslowexception.com/implementingcustom-ivaluetasksource-async-without-allocations/.

Заметим, что не следует заменять `Task` на `ValueTask` во всех рассмотренных выше местах. Чаще всего выигрыш в производительности не стоит затраченных усилий. Однако они могут окупиться, если в очень интенсивно используемом коде наш асинхронный метод часто завершается синхронно. Компромиссы при использовании `ValueTask` взамен `Task` прекрасно объяснены в описании API `ValueTask`:

- «Хотя `ValueTask<TResult>` помогает избежать выделения памяти в случае, когда метод успешно завершается синхронно, он содержит два поля, тогда как `Task<TResult>`, будучи ссылочным типом, представляет собой одно поле. Это означает, что вызов метода завершается возвратом двух полей данных вместо одного, а значит, объем копирования возрастает. Кроме того, если метод, возвращающий `ValueTask<TResult>`, находится внутри асинхронного метода, то конечный автомат для этого `async`-метода будет больше из-за необходимости хранить структуру с двумя полями вместо одной ссылки»;
- «Далее, если требуется не просто получить результат асинхронной операции с помощью `await`, `ValueTask<TResult>` приведет к более запутанной модели программирования, что может привести к большему количеству выделений памяти. Рассмотрим, к примеру, метод, который может вернуть или кешированную задачу `Task<TResult>`, или `ValueTask<TResult>`. Если потребитель (consumer) хочет использовать результат как `Task<TResult>`, например для вызова таких методов, как `Task.WhenAll` или `Task.WhenAny`, то `ValueTask<TResult>` нужно будет сначала преобразовать в `Task<TResult>` с помощью метода `AsTask`, что требует выделения памяти, которого можно было бы избежать, если с самого начала использовать кешированную `Task<TResult>`».

Скрытое выделение памяти

Помимо явного создания, объекты нередко создаются неявно в процессе выполнения некоторых операций. Часто это называют *скрытым выделением памяти*, и разработчики прилагают много усилий, чтобы избежать его. Понятно, что в скрытом выделении памяти приятного мало – его не увидишь в коде, если не знаешь заранее, что оно там есть.

Выделение памяти для делегата

Всякий раз при создании делегата (в т. ч. таких популярных, как `Func` и `Action`), скорее всего, происходит скрытое выделение памяти. Это может случиться как для делегата так называемой группы методов (метод, на который ссылка производится по имени, см. листинг 6.37), так и для делегата лямбда-выражения (в таком случае лямбда-выражение преобразуется в метод, генерируемый компилятором, см. листинг 6.38).

Листинг 6.37 ❖ Выделение памяти для делегата группы методов

```
Func<double> action = ProcessWithLogging; // скрытое
Func<double> action = new Func<double>(this.ProcessWithLogging); // явное
```

Листинг 6.38 ❖ Выделение памяти для делегата лямбда-выражения

```
Func<double> action = () => ProcessWithLogging(); // скрытое
Func<double> action = new Func<double>(this.<SomeMethod>b__31_0)(); // явное
```

Избежать таких выделений нет никакой возможности, но, зная о них, мы сможем писать код более осмысленно (например, не создавать делегатов многократно внутри цикла).

> С лямбда-выражениями связана важная оптимизация. Если они не захватывают никаких данных, то, скорее всего, компилятор C# сгенерирует код для кеширования экземпляра такого делегата в статическом поле (поэтому память будет выделена только один раз, при первом использовании).

Упаковка

Упаковка была описана в главе 4. Там мы упоминали два наиболее распространенных источника упаковки, а здесь вкратце напомним о них:

- тип значений используется там, где ожидается объект (ссылочный тип) (листинг 6.39), сюда относятся многие неявные преобразования;
- экземпляр типа значений используется в качестве типа интерфейса, реализованного этим типом значений (листинг 6.40).

Листинг 6.39 ❖ Типичные источники упаковки: преобразования типов

```
object obj = 0; // структура Int32 упаковывается
FooBar(0); // 0 будет упакован
static void FooBar(object obj)
{
}
```

Листинг 6.40 ❖ Типичные источники упаковки: передача в качестве интерфейса

```
// ValueTuple в ITuple
FooBar(new ValueTuple() {A = 1});
static void FooBar(ITuple tuple)
{
   // ValueTuple будет упакован
}
```

Избежать упаковки первого вида не всегда получится. Однако если тип `object` используется как способ сообщить, что методу можно передать произвольный

объект (как в случае FooBar в листинге 6.39), то лучше использовать обобщенные типы (см. листинг 6.41).

Листинг 6.41 ❖ Предотвращение упаковки путем использования обобщенного метода

```
void FooBar<T>(T obj)
{
   // FooBar<Int32> вызывается без упаковки
}
```

Второй источник выделения памяти можно обойти, воспользовавшись обобщенным методом с ограничением на тип аргумента (листинг 6.42).

Листинг 6.42 ❖ Предотвращение упаковки путем использования обобщенного метода с ограничением на тип аргумента

```
void FooBar<T>(T tuple) where T : ITuple
{
   // ValueTuple не упаковывается
   Console.WriteLine($"число элементов: {tuple.Length}");
   Console.WriteLine($"предпоследний элемент: {tuple[tuple.Length - 2]}");
}
```

Еще три источника упаковки типов значений не так широко известны:

- вызовы виртуальных методов valueType.GetHashCode() и valueType.ToString(), которые не переопределены в типе valueType;
- вызов valueType.GetType() всегда приводит к упаковке valueType;
- делегат, созданный для метода типа значений, всегда упаковывается (см. листинги 6.43 и 6.44).

Листинг 6.43 ❖ Выделение памяти для делегата группы методов типа значений

```
SomeStruct valueType;
Func<double> action2 = valueType.SomeMethod;
```

Листинг 6.44 ❖ IL-код, сгенерированный для кода из листинга 6.39

```
ldarg.1
box       CoreCLR.Program.SomeStruct
ldftn     instance float64 CoreCLR.Program.SomeStruct::SomeMethod()
newobj    instance void class [System.Runtime]System.Func`1<float64>::.ctor(object, native
int)
callvirt  instance !0 class [System.Runtime]System.Func`1<float64>::Invoke()
```

Замыкания

Замыкания – это механизм управления состоянием вычислений, т. е. «функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся ее параметрами» («Википедия»). Чтобы лучше понять это, возьмем в качестве примера простой метод с использованием LINQ, где применяются лямбда-выражения для фильтрации и выборки значений из списка (листинг 6.45). Если вы читаете эту главу последовательно, то, наверное, заметили два места в методе Closures, где возможно выделение памяти:

для лямбда-выражений могут быть созданы два делегата, т. к. методы Where и Select ожидают в качестве параметров Func<>[1].

Листинг 6.45 ❖ Пример кода с использованием лямбда-выражений

```
private IEnumerable<string> Closures(int value)
{
   var filteredList = _list.Where(x => x > value);
   var result = filteredList.Select(x => x.ToString());
   return result;
}
```

Существует еще один важный источник выделения памяти. Код в листинге 6.45 транслируется в более сложную конструкцию с использованием дополнительного класса <>c__DisplayClass1_0 (листинг 6.46). Этот класс реализует вышеупомянутое замыкание. Он содержит как подлежащую выполнению функцию (под внутренним именем <Closures>b__0), так и все переменные, необходимые для ее выполнения (в нашем случае value). Отметим следующие факты:

- замыкание реализовано в виде класса, что влечет за собой выделение памяти: в нашем примере для Program.<>c__DisplayClass1_0 будет выделяться память при каждом выполнении метода Closures;
- локальные переменные, хранящиеся в замыкании (захваченные им), учитываются при подсчете размера, занимаемого замыканием в куче, и в нашем случае захвачено целое значение. Чем больше таких переменных, тем больше «класс замыкания».

Листинг 6.46 ❖ Результат компиляции кода с лямбда-выражениями

```
private IEnumerable<string> Closures(int value)
{
   Program.<>c__DisplayClass1_0 <>c__DisplayClass1_ = new Program.<>c__DisplayClass1_0();
   <>c__DisplayClass1_.value = value;
   IEnumerable<int> arg_43_0 = this._list.Where(new Func<int, bool>(
       <>c__DisplayClass1_.<Closures>b__0));
   Func<int, string> arg_43_1;
   if ((arg_43_1 = Program.<>c.<>9__1_1) == null)
   {
      arg_43_1 = (Program.<>c.<>9__1_1 = new Func<int, string>(
          Program.<>c.<>9.<Closures>b__1_1));
   }
   return arg_43_0.Select(arg_43_1);
}

[CompilerGenerated]
private sealed class <>c__DisplayClass1_0
{
   public <>c__DisplayClass1_0()
```

---

[1] Однако благодаря вышеупомянутой оптимизации замыканий, скорее всего, будет выделена память только для одного делегата – того, который передается методу Where. Лямбда-выражение, передаваемое методу Select, не захватывает никакого состояния, потому компилятор C# генерирует код для кеширования делегата. В листинге 6.46 ему соответствует поле arg_43_1.

```
   {
   }
   internal bool <Closures>b__0(int x)
   {
      return x > this.value;
   }
   public int value;
}
```

О выделении памяти для замыканий следует помнить, когда наша цель – написать код, потребляющий как можно меньше памяти: чем меньше переменных захватывает замыкание, тем лучше. Это всегда можно проверить, например изучив декомпилированный код в программе dnSpy.

В листинге 6.47 приведены дополнительные иллюстрации на тему того, что и когда захватывается. Но предупреждаю: в значительной мере это результат оптимизаций, выполняемых компилятором. Правил и исключений так много, что иногда расследование того, что и когда захватывается, завершается выводом: все это магия (или, если серьезно, глубоко скрытые детали реализации применяемых в текущей версии оптимизаций). Имейте в виду, что все примеры в листинге 6.47 могут содержать скрытое выделение памяти для делегата лямбда-выражения.

Листинг 6.47 ❖ Примеры различных ситуаций, возможных, когда замыкание захватывает состояние

```
// Замыкания нет, потому что нечего захватывать (this не захватывается)
Func<double> action1 = () => InstanceMethodNotUsingThis();

// Замыкания нет, потому что нечего захватывать (this по-прежнему не захватывается)
Func<double> action2 = () => InstanceMethodUsingThis();

// Нечего захватывать
Func<double> action3 = () => StaticMethod();

// Захватывается ss
Func<double> action3 = () => StaticMethodUsingLocalVariable(ss);

// Замыкание захватывает ss и this (для вызова this.<>4__this.ProcessSomeStruct(this.ss);
// внутри)
// если бы аргумента ss не было, то ничего бы не захватывалось (один только this
// не захватывался бы)
Func<double> action6 = () => InstanceMethodUsingLocalVariable(ss);
```

Чтобы избавиться от замыканий, мы должны писать код, в котором лямбда-выражения не захватывают никаких переменных или вовсе отсутствуют. В листинге 6.48 показано, как можно было бы переписать метод из листинга 6.45. Заметим, однако, что теперь этот метод должен выделить память для списка результатов, а это может оказаться еще менее эффективно, чем выделение памяти в замыкании.

Листинг 6.48 ❖ Пример кода без лямбда-выражений и замыканий

```
private IEnumerable<string> WithoutClosures(int value)
{
   List<string> result = new List<string>();
   foreach (int x in _list)
```

```
      if (x > value)
         result.Add(x.ToString());
   return result;
}
```

Локальные функции, появившиеся в C# 7.0, по существу аналогичны лямбда-выражениям и могут выделять память под замыкание. Код в листинге 6.45 можно преобразовать в код с двумя локальными функциями (листинг 6.49). Но таким способом мы не избежим захвата переменной `value`.

Листинг 6.49 ❖ Код из листинга 6.45, переписанный с использованием локальных функций

```
private IEnumerable<string> ClosuresWithLocalFunction(int value)
{
   bool WhereCondition(int x) => x > value;
   string SelectAction(int x) => x.ToString();
   var filteredList = _list.Where(WhereCondition);
   var result = filteredList.Select(SelectAction);
   return result;
}
```

Код, сгенерированный компилятором (листинг 6.50), по-прежнему содержит замыкание, захватывающее `value`.

Листинг 6.50 ❖ Результат компиляции кода с использованием локальных функций

```
private IEnumerable<string> ClosuresWithLocalFunction(int value)
{
   Program.<>c__DisplayClass26_0 <>c__DisplayClass26_ = new Program.<>c__DisplayClass26_0();
   <>c__DisplayClass26_.value = value;
   return this._list.Where(new Func<int, bool>(<>c__DisplayClass26_.
      <ClosuresWithLocalFunction>g__WhereCondition0)).Select(new Func<int, string>
      (Program.<>c.<>9.<ClosuresWithLocalFunction>g__SelectAction26_1));
}
```

Yield return

Помимо асинхронных методов и замыканий, есть еще одна конструкция, которая приводит к скрытому выделению памяти для вспомогательных классов, генерируемых компилятором, – механизм `yield return`. Он используется для быстрого и удобного создания итераторных методов. Вся трудная работа по созданию класса итератора, содержащего внутреннее состояние, перекладывается на компилятор. Например, переписав метод из листинга 6.45 с применением оператора `yield`, мы избавимся от лямбда-выражений (листинг 6.51).

Листинг 6.51 ❖ Пример кода с использованием yield return

```
private IEnumerable<string> WithoutClosures(int value)
{
   foreach (int x in _list)
      if (x > value)
         yield return x.ToString();
}
```

Однако это не позволяет полностью избавиться от выделения памяти для временного объекта, который представляет собой состояние итератора (листинг 6.52). Как видим, он также захватывает переменную `value`, да еще и ссылку `this`. Но, принимая во внимание, что помимо замыканий, код в листинге 6.45 выделяет еще память для перечисляемых объектов, которые используются в методах `Where` и `Select`, этот вариант все же более экономичен.

Листинг 6.52 ❖ Результат компиляции кода с использованием оператора yield

```
[IteratorStateMachine(typeof(Program.<WithoutClosures>d__26))]
private IEnumerable<string> WithoutClosures(int value)
{
   Program.<WithoutClosures>d__26 expr_07 = new Program.<WithoutClosures>d__26(-2);
   expr_07.<>4__this = this;
   expr_07.<>3__value = value;
   return expr_07;
}
```

Массив параметров

Еще со времен C# 2.0 можно написать метод с переменным числом параметров, для чего служит ключевое слово `params` (листинг 6.53). Следует знать, что это не более чем синтаксический сахар, под которым скрывается обычный массив объектов, передаваемый методу в качестве последнего аргумента.

Листинг 6.53 ❖ Пример метода с переменным числом параметров

```
public void MethodWithParams(string str, params object[] args)
{
   Console.WriteLine(str, args);
}
```

Таким образом, при передаче методу аргументов с помощью `params` мы неявно создаем новый массив типа `object[]`. В случае когда не было передано ни одного параметра, компилятор применяет простую оптимизацию (листинг 6.54).

Листинг 6.54 ❖ Использование метода с параметрами

```
SomeClass sc;
MethodWithParams("Log {0}", sc); // Выделяется память для object[] с одним элементом sc

int counter;
MethodWithParams("Counter {0}", counter); // Упаковывается целое, и выделяется память для
                                          // object[] с одним элементом counter

p.MethodWithParams("Hello!"); // Память не выделяется, используется статический объект
                              // Array.Empty<object>()
```

Чтобы обойти этот источник скрытого выделения памяти, для многих методов с переменным числом параметров предоставляются перегруженные варианты для наиболее распространенных случаев с небольшим фиксированным числом параметров типа `object` или обобщенных типов (листинг 6.55).

Листинг 6.55 ❖ Пример перегруженных вариантов метода с переменным числом параметров

```
public void MethodWithParams(string str, object arg1)
{
   Console.WriteLine(str, arg1);
}

public void MethodWithParams(string str, object arg1, object arg2)
{
   Console.WriteLine(str, arg1, arg2);
}

public void GenericMethodWithParams<T1>(string str, T1 arg1)
{
   Console.WriteLine(str, arg1);
}

public void GenericMethodWithParams<T1,T2>(string str, T1 arg1, T2 arg2)
{
   Console.WriteLine(str, arg1, arg2);
}
```

Конкатенация строк

Конкатенация строк и соображения, в силу которых класс `string` сделан неизменяемым, описаны в главе 4. Для полноты картины напомним типичные примеры, в которых выделяется память для временных строк (листинг 6.56).

Листинг 6.56 ❖ Примеры типичных операций со строками

```
// Создается временная строка "Hello " + otherString
string str = "Hello " + otherString + "!";

// Создается строка str + " you are welcome" (предыдущее значение str становится мусором)
str += " you are welcome";
```

Как было сказано в главе 4, для операций со строками среднего размера лучше использовать перегруженные варианты метода `String.Format`, поскольку в них применяется кешированный объект `StringBuilder`. Для создания длинных текстов путем дописывания коротких строк `StringBuilder` – это оптимальный выбор. Но в простейших случаях, когда конкатенируется всего две-три строки, лучше всего пользоваться оператором плюс (как в первой строке примера в листинге 6.56), т. к. он основан на эффективной реализации метода `string.Concat` (листинг 6.57), которая манипулирует данными строки напрямую (или можно явно вызвать метод `Concat`).

Листинг 6.57 ❖ Эффективная реализация `string.Concat` (метод `FillStringChecked` напрямую манипулирует внутренними данными строки)

```
public static String Concat(String str0, String str1)
{
   if (IsNullOrEmpty(str0)) {
      if (IsNullOrEmpty(str1)) {
         return String.Empty;
      }
```

```
      return str1;
   }
   if (IsNullOrEmpty(str1)) {
      return str0;
   }
   int str0Length = str0.Length;
   String result = FastAllocateString(str0Length + str1.Length);
   FillStringChecked(result, 0, str0);
   FillStringChecked(result, str0Length, str1);
   return result;
}
```

> Если код форматирования строк является критичным с точки зрения производительности и вы хотите вообще избежать выделения памяти, то стоит присмотреться к внешним библиотекам, например StringFormatter (https://github.com/MikePopoloski/StringFormatter). В этой библиотеке выделения памяти нет в принципе, а API очень похож на `string.Format`. Существуют даже надстроенные над ней высокоуровневые библиотеки, например библиотека протоколирования ZeroLog (https://github.com/Abc-Arbitrage/ZeroLog). Начиная с версии .NET Core 2.1 можно также использовать все API, связанные с типом `Span<T>`, для манипулирования строками (см. главу 14).

Скрытое выделение памяти в библиотеках

Очевидно, что из-за многочисленных источников выделения памяти (явных и скрытых) использование внешних библиотек сопряжено с риском. Невозможно описать все ситуации, поскольку для этого пришлось бы подробно описывать большинство популярных библиотек, которыми мы пользуемся. Поэтому мы рассмотрим только самые распространенные источники выделения памяти этого типа.

Коллекции из пространства имен System.Collections.Generic

Некоторые часто используемые коллекции из пространства имен `System.Collections.Generic` можно рассматривать как обертки над массивами. В качестве примера рассмотрим популярный класс `List<T>` (листинг 6.58). В нем хранится массив элементов предопределенного размера (если в конструкторе не была указана емкость). По мере роста списка (в результате вызова метода `Add`) этого массива может не хватить. Тогда создается новый, и в него копируются все элементы из старого.

Листинг 6.58 ❖ Начало реализации класса `List<T>` (из исходного кода .NET)

```
public class List<T> : IList<T>, System.Collections.IList, IReadOnlyList<T>
{
   private const int _defaultCapacity = 4;
   private T[] _items;
   ...
```

Таким образом, при заполнении `List<T>`, а также коллекций `Stack<T>`, `SortedList<T>`, `Queue<T>` и им подобных размер внутренних массивов может много раз меняться. Если приблизительный окончательный размер известен заранее, то лучше воспользоваться перегруженным конструктором, в котором задается емкость. Вообще, всегда рекомендуется задавать ожидаемую емкость, если это возможно,

и не думать, как коллекция ей воспользуется, – будем считать, что она сделает это оптимальным образом.

LINQ: делегаты

LINQ элегантен, и работать с ним – одно удовольствие. Достаточно всего нескольких строчек кода, чтобы лаконично записать сложные манипуляции данными. Но LINQ – один из тех механизмов в C#, которые требуют выделения большого количества памяти. В нем очень много скрытых источников выделения памяти (вроде упомянутых в разделе о замыканиях). Один из самых распространенных мы уже описывали – выделение памяти для делегатов. Поскольку методы LINQ основаны на делегатах, при их использовании таковых создается много (листинг 6.59).

Листинг 6.59 ❖ Пример выделения памяти для делегата при выполнении LINQ-запроса

```
// Создает делегат лямбда-выражения
var linq = list.Where(x => x.X > 0);
```

Но, как было сказано выше, если выполняемая функция ничего не захватывает, то такие делегаты кешируются. Это значит, что благодаря оптимизации компилятора память для них выделяется только один раз (см. листинг 6.60).

Листинг 6.60 ❖ Преобразованный компилятором код из листинга 6.59: память для делегата выделяется только один раз

```
Func<SomeClass, bool> arg_152_1;
if ((arg_152_1 = Program.<>c.<>9__0_0) == null)
{
   arg_152_1 = (Program.<>c.<>9__0_0 = new Func<SomeClass, bool>
      (Program.<>c.<>9.<Main>b__0_0));
}
arg_152_0.Where(arg_152_1);
```

LINQ: создание анонимных типов

При написании LINQ-запросов возникает искушение создать временные анонимные типы, что только увеличивает и так уже немалые затраты на выделение памяти. В листинге 6.61 приведен искусственный пример простого LINQ-запроса, синтаксически напоминающего SQL.

Листинг 6.61 ❖ Пример простого LINQ-запроса

```
public IEnumerable<Double> Main(List<SomeClass> list) {
   var linq = from x in list
      let s = x.X + x.Y
      select s;
return linq;
```

Следует понимать, что ключевое слово `let` – не что иное, как создание анонимного временного объекта (см. сгенерированный компилятором метод `<Main>b__0_0` в листинге 6.62).

Листинг 6.62 ❖ Преобразованный компилятором простой LINQ-запрос

```
[CompilerGenerated]
private sealed class <>c
{
   internal <>f__AnonymousType0<SomeClass, double> <Main>b__0_0(SomeClass x)
   {
      return new <>f__AnonymousType0<SomeClass, double>(x, x.X + x.Y);
   }
   ...
}

public IEnumerable<double> Main(List<SomeClass> list)
{
   return list.Select( <>c.<>9__0_0 ?? (<>c.<>9__0_0 = <>c.<>9.<Main>b__0_0))
      .Select( <>c.<>9__0_1 ?? (<>c.<>9__0_1 = <>c.<>9.<Main>b__0_1));
}
```

Иногда такие временные типы нужны, чтобы LINQ-запросы выглядели элегантно. Но всегда стоит задуматься, действительно ли они необходимы или используются лишь для удобства и красоты. В нашем примере временный тип, очевидно, лишний, потому что сумму можно вернуть непосредственно (листинг 6.63), но при этом генерируется гораздо более простой код без выделения памяти (листинг 6.64).

Листинг 6.63 ❖ Пример простого LINQ-запроса с синтаксисом метода

```
public IEnumerable<Double> Main(List<SomeClass> list) {
   var linq = list.Select(x => x.X + x.Y);
   return linq;
}
```

Листинг 6.64 ❖ Преобразованный компилятором LING-запрос из листинга 6.63

```
[CompilerGenerated]
private sealed class <>c
{
   internal double <Main>b__0_0(SomeClass x)
   {
      return x.X + x.Y;
   }
   ...
}

public IEnumerable<double> Main(List<SomeClass> list)
{
   return list.Select( <>c.<>9__0_0 ?? (<>c.<>9__0_0 = <>c.<>9.<Main>b__0_0));
}
```

LINQ: объекты перечисляемых типов

Хотя мы можем и не знать об этом, методы LINQ на самом деле строят цепочку перечисляемых объектов, тип которых допускает перечисление элементов коллекции. Понятно, что для этих объектов нужно выделить память. Это делают даже

самые простые методы, например статический метод `Enumerable.Range` создает *итератор*, один из конкретных способов реализации перечисляемого типа (листинг 6.65).

Листинг 6.65 ❖ Простой пример скрытого выделения памяти для итератора

```
// Выделяется память для System.Linq.Enumerable/'<RangeIterator>d__111'
var range = Enumerable.Range(0, 100);
```

Такие популярные методы, как `Where` или `Select`, также выделяют память для своих итераторов. Например, метод `Where` может создавать следующие итераторы:

- `WhereArrayIterator` вызывается для массива;
- `WhereListIterator` вызывается для списка;
- `WhereEnumerableIterator` вызывается в остальных случаях.

Каждый из этих итераторов занимает примерно 48 байт, поскольку содержит ссылку на исходную коллекцию, делегат для выборки, идентификатор потока и т. д. Выделение несколько раз по 48 байт в одном методе из-за использования LINQ может считаться или не считаться проблемой – все зависит от применяемого критерия производительности.

Внутри LINQ имеются дополнительные оптимизации для комбинирования итераторов, если это возможно, но, к сожалению, это не устраняет выделение памяти. Например, если, как это часто бывает, используется пара методов `Where` и `Select`, создается комбинированный итератор `WhereSelectArrayIterator` (или `WhereSelectListIterator`, или `WhereSelectEnumerableIterator`), но вместе с ним – промежуточный объект `WhereArrayIterator` (или другой, в зависимости от типа коллекции).

Рассмотрим тривиальный пример фильтрации строк (листинг 6.66). В нем создается два итератора:

- `WhereArrayIterator` размером 48 байт с очень коротким временем жизни, поскольку вскоре он будет заменен следующим итератором;
- `WhereSelectArrayIterator` размером 56 байт.

Листинг 6.66 ❖ Простой пример скрытого выделения памяти для итератора

```
string[] FilterStrings(string[] inputs, int min, int max, int charIndex)
{
   var results = inputs.Where(x => x.Length >= min && x.Length <= max)
      .Select(x => x.ToLower());
   return results.ToArray();
}
```

Кроме того, выделяется память для делегата и замыкания, которое захватывает два целых числа (`min` и `max`).

> Можно совместить приятное с полезным, воспользовавшись какой-нибудь библиотекой, которая автоматически переписывает LINQ-запросы в виде более процедурного кода. Самые популярные – roslyn-linq-rewrite (https://github.com/antiufo/roslyn-linq-rewrite) и LinqOptimizer (http://nessos.github.io/LinqOptimizer).

> **Примечание.** В настоящее время в среде .NET все большую популярность завоевывает функциональное программирование – благодаря распространению языка F# и общего всплеска интереса к функциональным языкам. Один из основных принципов функционального программирования – неизменяемость данных. В таких языках, как

F#, выполнение функций не изменяет уже существующие данные, а возвращает новые. Разумеется, при этом могут возникать вопросы по поводу производительности. Опыт работы с C# научил нас, что из-за неизменяемости строк создается целая серия непрошеных временных объектов. В своем воображении мы уже видим горы создаваемых объектов, между которыми копируются данные, ведь именно это, наверное, и происходит в F#, разве не так? Но вообще при работе с неизменяемыми типами и функциональным программированием нужно подвергнуть пересмотру весь прошлый опыт. Да, неизменяемые типы могут оказаться гораздо медленнее в обычной ситуации, где данные разрешено изменять. Достаточно измерить скорость добавления большого количества объектов в изменяемый список `List<T>` и его неизменяемый аналог. Понятно, что неизменяемая коллекция при каждом добавлении элемента будет создавать свою копию, так что операция окажется гораздо медленнее (и, кстати, проектировщики функциональных языков, наверное, приложили немало усилий, чтобы такие операции выполнялись не в лоб; например, можно повторно использовать общие части коллекций). Но сравнивать эти коллекции следует по-другому. Неизменяемость дает очень важные преимущества, особенно в многопоточных программах, которые используются все чаще. Безопасный, не требующий блокировки доступ к данным, допускающим только чтение, может дать значительный выигрыш в ситуациях с высокой степенью конкурентности (когда много потоков конкурируют за доступ к разделяемому ресурсу), и это перевешивает накладные расходы на обеспечение неизменяемости. Поэтому неизменяемые типы – отличный выбор для многопоточной и (или) параллельной обработки. К тому же неизменяемость позволяет более эффективно использовать кеш процессора без накладных расходов на когерентность кешей. Все эти соображения относятся к неизменяемым коллекциям в C#, которые находятся в пространстве имен `System.Collections.Immutable` (например, `ImmutableArray<T>`, `ImmutableList<T>` и т. д.). Поэтому все дело в выборе подходящего инструмента для решения конкретной задачи. Не придавайте чрезмерной важности тестам, показывающим, что для значительной модификации состояния неизменяемой коллекции действительно требуется много времени. Конечно, требуется, если они используются не для того, для чего предназначены!

Сценарий 6.2. Исследование выделения памяти

Описание. После развертывания новой версии веб-приложения ASP.NET Core мы заметили существенный рост потребления памяти, наблюдая за счетчиками `Рабочий набор (частный)`, `Байт исключительного пользования` и `Байт виртуальной памяти` в группе `Process(dotnet)`, а также за счетчиком `\Память CLR.NET(dotnet)\Всего фиксировано байтов`. Разработчики не могут найти подозрительное место в измененном коде, которое могло бы стать источником повышенного выделения памяти. Мы хотим помочь им, проанализировав развернутое приложение.

Анализ. Один из лучших инструментов для исследования выделения памяти – PerfView. В главе 4 описаны три метода выборки. Для получения максимально точных результатов следует использовать метод .NET Alloc, если это возможно. При этом с помощью .NET Profiling API в исследуемый процесс встраивается библиотека EtwCorProfiler, что позволяет зарегистрировать каждую операцию выделения. Понятно, что накладные расходы при этом велики, так что это можно делать только на локальной машине или в строго контролируемой среде разработки. В противном случае можно воспользоваться методом .NET SampAlloc, в котором та же техника применяется с меньшей гранулярностью. С другой стороны, основанный на ETW, метод .NET Alloc вносит совсем небольшие накладные расходы, поэтому его можно без опаски использовать даже в условиях промышленной

работы приложения. Однако помните, что последние два метода видят не все операции выделения, а только выборку, так что результаты получаются более грубые.

> В методах .NET Alloc и .NET SampAlloc, реализованных в PerfView, для трассировки выделения памяти применяется API профилирования CLR, а точнее, обратный вызов `ICorProfilerCallback3::ObjectAllocated`, который среда выполнения выполняет при каждом выделении памяти для нового объекта. Чтобы это сделать, JIT-компилятор отключает быстрый способ выделения памяти, написанный на ассемблере. Так что уже по этой причине исследуемая программа будет работать немного медленнее.

Исследуем выделение памяти методом .NET Alloc:

- запустите **PerfView**;
- в диалоговом окне **Collect** выберите вариант **.NET Alloc**;
- запустите веб-приложение – важно, чтобы это было сделано после запуска сбора данных в режиме *.NET Alloc* (или **.NET SampAlloc**);
- погуляйте по сайту – наибольший интерес, наверное, представляют части, затронутые последними изменениями;
- остановите сбор данных;
- в **PerfView** выберите строку **GC Heap Net Mem Stacks** из группы **Memory**;
- выберите приложение **dotnet.exe**.

Далее можно направить расследование по одному из двух путей.

1. Получить высокоуровневое представление выделений памяти:
   - на вкладке **By Name** отсортируйте данные по столбцу **Exc** в порядке убывания – так вы быстро увидите самые существенные источники выделения памяти (рис. 6.22). Обратите внимание, что часто большая часть выделений приходится на строку `Type <Unknown>`. К сожалению, ETWClrProfiler не всегда может получить информацию о типе от среды выполнения. В таких случаях он подставляет тип `<Unknown>`.

By Name ? | Caller-Callee ? | CallTree ? | Callers ? | Callees ? | Flame Graph ? | Notes ?

Name ?	Exc % ?	Exc ?	Exc Ct ?	Inc % ?	Inc ?	Inc Ct ?
Type <Unknown>	98.7	24,549,600	405,260	98.7	24,549,600.0	405,260
Type System.Collections.Immutable.SortedInt32KeyNode`1	1.0	238,560	4,260	1.0	238,560.0	4,260
Type System.Collections.Immutable.ImmutableHashSet`1	0.1	19,760	494	0.1	19,760.0	494
Type <>c__DisplayClass42_0	0.1	16,160	505	0.1	16,160.0	505
Type ?[]	0.0	3,992	52	0.0	3,992.0	52
Type Microsoft.AspNetCore.Razor.Language.Extensions.DefaultTagHelperPropertyIntermediateNode	0.0	3,960	33	0.0	3,960.0	33
Type Microsoft.CodeAnalysis.CSharp.Syntax.CastExpressionSyntax	0.0	3,840	60	0.0	3,840.0	60
Type Microsoft.AspNetCore.Razor.Language.Extensions.PreallocatedTagHelperPropertyIntermediateN	0.0	3,600	30	0.0	3,600.0	30
Type Microsoft.AspNetCore.Razor.Language.Intermediate.TagHelperPropertyIntermediateNode	0.0	3,432	33	0.0	3,432.0	33
Type Microsoft.Net.Http.Headers.EntityTagHeaderValue[]	0.0	3,384	61	0.0	3,384.0	61
Type Microsoft.Net.Http.Headers.EntityTagHeaderValue	0.0	2,880	72	0.0	2,880.0	72
Type Microsoft.AspNetCore.Razor.Language.Extensions.PreallocatedTagHelperPropertyValueIntermed	0.0	2,016	21	0.0	2,016.0	21
Type <GetEnumerator>d__7	0.0	1,920	48	0.0	1,920.0	48

Рис. 6.22 ❖ Высокоуровневое представление выделения памяти в веб-приложении для ASP.NET Core

Но сам тип – не единственная информация, не менее полезны агрегированные источники выделения (стеки вызовов). Например, чтобы исследовать источники выделения типов `<Unknown>`, выберите команду **Goto ➤ Goto Item in Callers** из контекстного меню найденного элемента. Напомним, что по ходу расследования:

- всегда можно попробовать загрузить символы для неименованных модулей (имена которых заканчиваются строкой `?!`, например `<<microsoft.codeanalysis.csharp!?>>`), воспользовавшись командой **Lookup Symbols** из контекстного меню;
- можно сгруппировать модули с помощью команды **Grouping ➤ Group Module** из контекстного меню. Таким образом можно объединить в одну группу большинство модулей типа `<Unknown>` (рис. 6.23).

Methods that call Type <Unknown>

Name	Inc %	Inc	Inc Ct
Type <Unknown>	98.7	24,549,600.0	405,260
+ ntdll!NtTraceEvent	98.7	24,549,590.0	405,260
+ ntdll!EtwpEventWriteFull	98.7	24,549,590.0	405,260
+ ntdll!EtwEventWrite	98.7	24,549,590.0	405,260
+ etwclrprofiler!Template_xxxx	98.7	24,549,590.0	405,260
+ etwclrprofiler!CorProfilerTracer::ObjectAllocated	98.7	24,549,590.0	405,260
+ coreclr!EEToProfInterfaceImpl::ObjectAllocated	98.7	24,549,590.0	405,260
+ coreclr!ProfilerObjectAllocatedCallback	98.7	24,549,590.0	405,260
+ coreclr!JIT_New	62.7	15,604,170.0	315,857
+ microsoft.codeanalysis.csharp	19.7	4,913,008.0	67,713
+ microsoft.aspnetcore.mvc.razor.extensions	19.1	4,743,088.0	115,424
+ system.linq	7.5	1,871,536.0	32,889
+ microsoft.aspnetcore.razor.language	5.9	1,465,576.0	42,897
+ system.private.corelib	3.2	796,712.0	19,524

Рис. 6.23 ❖ Наиболее распространенные источники выделения типа <Unknown>

Следует тщательно проанализировать часто создаваемые объекты. К сожалению, эта задача довольно утомительная. Чтобы найти подозрительную область, заслуживающую анализа, мы можем сравнить снимки кучи, созданные PerfView, и выявить объекты, вызывающие наибольшее «движение» в памяти.

2. Чтобы исследовать операции выделения памяти, выполненные каким-то конкретным методом:
   - на вкладке **By Name** выберите команду **[No grouping]** в **GroupPats** – чтобы разобрать группу на отдельные элементы;
   - в поле **Find** введите имя своей функции, скажем `HomeController.Contact`;
   - выберите команду **Goto ➤ Goto Item in Callees** из контекстного меню элемента – вы увидите все выделения памяти, произведенные самим этим методом и теми, которые из него вызываются (рис. 6.24).

Methods that are called by CoreCLR.AspNetCore!CoreCLR.AspNetCore.Controllers.HomeController.Contact()

Name	Inc %	Inc	Inc Ct
CoreCLR.AspNetCore!CoreCLR.AspNetCore.Controllers.HomeController.Contact()	0.0	132.0	2
+ system.private.corelib!System.Collections.Generic.Dictionary`2[System.__Canon,System.__Canon].set_Item(System.__Canon, S	0.0	132.0	2
+ system.private.corelib!System.Collections.Generic.Dictionary`2[System.__Canon,System.__Canon].TryInsert(System.__Canon,	0.0	132.0	2
+ system.private.corelib!System.Collections.Generic.Dictionary`2[System.__Canon,System.__Canon].Initialize(Int32)	0.0	132.0	2
+ coreclr!JIT_NewArr1	0.0	132.0	2
+ coreclr!AllocateArrayEx	0.0	96.0	1
+ coreclr!??FastAllocatePrimitiveArray	0.0	36.0	1

Рис. 6.24 ❖ Выделения памяти, произведенные указанным методом и вызываемыми из него

Как видим, метод `HomeController.Contact` создает два массива в методе `System.Collections.Generic.Dictionary<>.Initialize`. Сам метод `Contact` в нашем примере тривиален, он всего лишь записывает значение одного элемента в словарь `View-`

Data (листинг 6.67). Но если бы мы заглянули в код метода Dictionary<TKey,TValue>.Initialize, то увидели бы, что на самом деле он выделяет память для двух массивов – кластеров и записей (buckets and entries). Конечно, это лишь один пример того, как можно использовать полученную подробную информацию. В процессе расследования вас будет интересовать выделение памяти, произведенное вашим кодом, поэтому имеет смысл сгруппировать все внешние модули.

Листинг 6.67 ❖ Метод HomeController.Contact

```
public IActionResult Contact()
{
   ViewData["Message"] = "Your contact page.";
   return View();
}
```

Заметим, что в Linux диагностировать выделение памяти не так легко и приятно. PerfView со своим профилировщиком здесь не поможет. API профилирования .NET для Linux не настолько зрелый, поэтому основанных на нем и хорошо протестированных инструментов пока нет. Для выборки выделений памяти можно использовать событие LTTng GCAllocationTick, так мы получим статистическую информацию об объектах, которые выделяются чаще всего. Но механизм LTTng не позволяет получить стек вызовов в момент события. Их можно получить с помощью perf, собирая данные о генерирующей события функции EventXplatGCEnabledAllocationTick в библиотеке libcoreclr.so. Однако таким способом мы достигнем противоположной цели – сможем проанализировать стек вызовов, но ничего не будем знать о типе. В настоящее время не существует механизма, который позволил бы сделать и то, и другое. И коммерческие программы тоже не предоставляют приличной поддержки для такого рода диагностики.

Сценарий 6.3. Функции Azure

Описание. Потребление ресурсов функциями Azure (Azure Functions) тарифицируется посекундно и изменяется в гигабайт-секундах (ГБ-с) и количестве выполнений. В прейскуранте Майкрософт написано: «Память, потребляемая функцией, округляется до ближайших 128 МБ вплоть до максимального размера 1536 МБ, а время выполнения округляется до 1 мс. Минимальное время выполнения и объем памяти для одной функции составляют 100 мс и 128 МБ соответственно». Это означает, что каждая функция потребляет по меньшей мере 0,0125 ГБ-с (100 мс * 128 МБ, т. е. 0,1 с * 0,125 ГБ). Кроме того, бесплатно предоставляется 400 000 ГБ-с и 1 млн выполнений в месяц.

С учетом сказанного понятно, что было бы желательно минимизировать потребление памяти. Если наша функция Azure неэффективно расходует память, то мы можем выйти за пределы бесплатного лимита. Стоимость возрастает всякий раз, как потребление памяти превышает очередные 128 МБ. В мире .NET трудно найти другое место, где потребление памяти было бы так непосредственно связано с денежными затратами.

Анализ. В Azure есть сервис Application Insights, который позволяет следить за потреблением ресурсов функциями Azure. Мы можем отслеживать так называемые единицы выполнения функций (Function Execution Unit). Они измеряются

в мегабайт-миллисекундах (МБ-мс), так что нужно перевести их в ГБ-с. Отслеживание единиц выполнения функций позволяет вести учет затрат, но, к сожалению, детального анализа потребления памяти функциями мы так не получим. Стало быть, анализом и оптимизацией функции лучше заниматься во время разработки приложения. Благодаря инструментам Azure Functions Core Tools мы можем выполнять функции локально, поэтому исследовать выделение памяти можно так же просто, как в сценарии 6.2. Нужно только профилировать процесс `func.exe` (так называется исполняемый файл инфраструктуры Azure Function CLI, внутри которой размещаются наши функции).

> **ПРИМЕЧАНИЕ** Если вы хотите следить за интенсивностью выделения памяти внутри своей программы, то одно из простейших решений – воспользоваться статическим методом `GC.GetAllocatedBytesForCurrentThread`. Он позволяет получить точную информацию о том, сколько байтов памяти было выделено с момента начала работы текущего потока.

Резюме

В этой главе мы подробно рассмотрели создание объектов в .NET. Теперь мы знаем, что выделение памяти для объекта действительно может быть очень быстрым, но также вполне возможен вариант с более сложным поиском необходимого места, включающим запуск сборки мусора.

В первой части главы были представлены детали реализации распределителя памяти в .NET. Чтобы добиться максимальной скорости выделения, разработчикам пришлось пуститься на разные ухищрения. Но разбираться в этих деталях было очень интересно и поучительно. В результате мы поняли, насколько сложна эта задача вообще и как замечательно она решена в .NET CLR.

Вторая часть главы была посвящена в основном практическому обзору одного из самых важных аспектов эффективного управления памятью – тому, как избежать выделения памяти. Если нам удастся это сделать, то, очевидно, мы избежим также накладных расходов на сборку мусора. Так что это одна из основных оптимизаций в мире .NET. Мы представили довольно обширный (но, конечно, не исчерпывающий) список возможных источников выделения памяти и потенциальных способов обойти их (там, где это возможно).

Мы также включили три сценария решения проблем, относящихся к выделению памяти. Они дают возможность взглянуть на вопрос о создании новых объектов под более практическим, ориентированным на диагностику, углом зрения.

Правило 14: избегайте выделения памяти в куче на критических с точки зрения производительности участках программы

Обоснование. Говорят, что выделение памяти в .NET обходится дешево. Но в этой главе показано, что это не всегда правда. Следует знать о потенциальной стоимости выделения. Считать затраты значительными или нет, зависит от контекста. Просто имейте в виду, что выделение может повлечь за собой значительный объ-

ем операций с памятью, взаимодействие с операционной системой и запуск сборки мусора. Чем больше объектов мы создаем, тем больше нагрузка на GC. Поэтому в тех частях кода, которые должны работать особенно быстро, лучше вообще избегать выделения памяти.

Как применять. Способов избежать выделения памяти столько же, сколько ситуаций, в которых выделение памяти может иметь место. Они были подробно описаны в разделе «Избегание выделения памяти». Иногда выделение памяти производится явно, т. е. мы знаем о нем все. Но и от таких выделений можно избавиться, воспользовавшись пулами объектов или типами значений. Другие операции выделения скрыты – различные библиотеки могут делать это без нашего ведома. Чтобы избежать скрытых выделений, их необходимо выявить. Желательно знать о наиболее распространенных источниках скрытого выделения памяти, чтобы можно было быстро определить их места в своей программе. Нетривиальные источники следует исследовать с помощью диагностических средств.

Иллюстрирующие сценарии: 6.2, 6.3.

Правило 15: избегайте дорогостоящего выделения памяти в LOH

Обоснование. Выделение памяти в .NET не всегда обходится дешево, а для выделения памяти в куче больших объектов это тем более справедливо. Предположение о том, что объекты создаются в LOH нечасто, и тот факт, что такие объекты велики, диктуют проектное решение – не выделять для них память заранее. Поэтому при выделении памяти в LOH преобладает стоимость ее обнуления. Если мы создаем большие объекты часто, то имеет смысл организовать пул с целью их повторного использования. В таком случае характер работы с памятью будет более стабильным, и это поможет не только сократить затраты на выделение, но и немного снизить нагрузку на GC.

Как применять. Даже если большие объекты создаются часто, может оказаться, что совсем отказаться от выделения нельзя. Использование типов значений не поможет из-за ограничений на место в стеке. Лучше всего в таких случаях прибегнуть к пулу объектов – см. соответствующие части раздела «Избегание выделения памяти».

Правило 16: по возможности выделяйте память в стеке

Обоснование. Классы – это фундаментальные типы данных в .NET. Они сопровождают нас с самого начала изучения C#. Стоит подумать о структуре данных, как сразу же на ум приходит класс. Мы создаем классы по умолчанию, когда разрабатываем приложения. С другой стороны, структуры кажутся какой-то экзотикой, о которой мы узнаем в начале обучения, а потом забываем. Они кажутся странными и непонятными. Однако так вовсе не должно быть: у них есть весьма ценные свойства, например улучшенная локальность памяти, предотвращение выделения памяти в куче и различные полезные оптимизации со стороны компилятора и JIT-компилятора.

Как применять. Нужно узнать, что могут структуры, и постараться добавить их в свой арсенал. Реализуя новую функциональность, задавайте себе следующие вопросы. Так ли нужен нашему методу класс или структуры вполне достаточно? Нужна ли нам коллекция объектов или, быть может, хватит простого массива структур? Не бойтесь копирования структур – пользуйтесь новыми и более развитыми возможностями C# для передачи структур по ссылке в различных ситуациях. Но, конечно, не нужно стрелять из пушки по воробьям. Все это стоит делать только в тех участках кода, где производительность критически важна, которые исполняются часто и оказывают заметное влияние на потребление ресурсов.

Глава 7

Сборка мусора – введение

Добро пожаловать в самую важную часть этой книги. В предыдущих главах мы описывали предмет управления памятью широкими мазками. Было дано введение в теорию и аппаратное обеспечение. Мы также узнали о многих деталях организации памяти в среде .NET – как она разделена на сегменты и поколения и как вся эта инфраструктура взаимодействует с операционной системой. Эти знания ценны и сами по себе, например они помогают диагностировать проблемы, вызванные слишком большим количеством выделений памяти, и использовать различные методы предотвращения таких проблем.

Однако нельзя отрицать, что управление памятью в .NET неразрывно связано с ее автоматическим освобождением. Мы уже говорили о распределителе, поэтому знаем, как объекты создаются. Теперь же пришло время узнать о том, как и когда они удаляются, а занятая ими память автоматически возвращается системе.

Эта и три последующие главы содержат длинную историю о том, как в .NET работает сборка мусора. Она разделена на четыре главы, чтобы не вываливать всю информацию на голову бедного читателя сразу. Однако эти главы взаимосвязаны, и, чтобы составить полное представление, их надо прочитать целиком.

Заметим также, что эти главы основаны на ранее изложенном материале. Поэтому если вы читаете книгу не подряд, то я настоятельно рекомендую хотя бы мельком просмотреть предыдущие главы (а особенно главы 5 и 6).

В этой главе мы расскажем о многих ситуациях, в которых может иметь место сборка мусора. Мы точно опишем, из каких этапов она состоит, и начнем глубоко изучать первые шаги. Все это будет сопровождаться пояснениями и примерами, которые дадут возможность не только получить интеллектуальное удовольствие от приобретения новых знаний, но и применить их на практике.

Общее описание

Прежде чем двигаться дальше, взглянем на реализацию сборщика мусора в среде выполнения Microsoft .NET с высоты птичьего полета. Самый важный факт уже упоминался в предыдущих главах: GC может работать в двух режимах:

- *режим рабочей станции* – предназначен для минимизации задержек GC с точки зрения управляемых потоков. Можно сказать, что это стратегия, при которой сборка мусора производится чаще, поэтому объем работы и воспринимаемая пауза меньше. Этот режим особенно полезен в настольных приложениях, когда воспринимаемая задержка – важное условие удовле-

творенности пользователя: мы не хотим, чтобы все приложение зависало, пока выполняется длительная сборка мусора;

- *серверный режим* – предназначен для максимизации пропускной способности приложения. Стратегия заключается в том, чтобы запускать GC реже, так что каждая сборка будет продолжаться дольше. Это также означает, что будет потребляться больше памяти – чем реже запускается сборка, тем больше памяти ждет освобождения. Однако паузы и повышенный расход памяти не столь важны, поскольку предпочтение отдается статистической итоговой пропускной способности – сколько данных было обработано в единицу времени.

Между обоими режимами существуют важные проектные различия. Одно из самых важных – количество управляемых куч. В главе 5 отмечалось, что в режиме рабочей станции есть всего одна управляемая куча, тогда как в серверном режиме их может быть столько, сколько логических ядер имеется в компьютере.

Кроме того, у каждого режима имеется два подрежима:

- *неконкурентный* – во время выполнения GC все управляемые потоки приложения приостанавливаются;
- *конкурентный* – некоторые части GC выполняются одновременно с управляемыми потоками.

Таким образом, получаем четыре способа конфигурирования GC в приложении. Подробно эти комбинации описаны в главе 11 вместе с обсуждением того, когда применять каждую из них. Для простоты усвоения материала в главах 7–10 рассматривается только простейший случай – неконкурентный режим рабочей станции. Так мы сможем разобраться в большинстве аспектов GC, не загромождая изложение деталями. Остальные режимы отличаются только некоторыми нюансами, поэтому материал, изложенный в этой и трех следующих главах, в полной мере относится и к ним.

Напомним важный факт, касающийся поведения двух областей управляемой кучи:

- в куче малых объектов может применяться сборка очисткой или с уплотнением – решение принимает сам GC, но мы можем попросить его выбрать конкретный режим, если захотим запустить сборку вручную;
- в куче больших объектов по умолчанию применяется только сборка очисткой, но мы можем явно запросить однократную сборку с уплотнением.

Далее для желающих исследовать рассматриваемые вопросы самостоятельно приводятся детали из исходного кода CoreCLR. Процесс сборки мусора описывается несколькими параметрами. Один из самых важных – перечисление `collection_mode`, с помощью которого можно установить следующие флаги:

- `collection_non_blocking` – неблокирующая (конкурентная) сборка;
- `collection_blocking` – блокирующая («остановка мира») сборка;
- `collection_optimized` – GC продолжится, только если это необходимо (когда исчерпан *бюджет выделения* указанного поколения);
- `collection_compacting` – сборка с уплотнением кучи малых объектов;
- `collection_gcstress` – внутренний режим нагрузочного тестирования CLR.

Все эти варианты будут описаны позже, а пока сконцентрируемся на простейшем неконкурентном режиме рабочей станции.

Пример процесса сборки мусора

Думаю, что сейчас самое время явно огласить факты, которые мельком упоминались в нескольких местах. Это позволит составить наглядное представление о работе GC.

Прежде всего сборка мусора производится в контексте конкретного поколения, которое часто называют *выбранным* (condemned generation). Вся идеология поколений опирается на тот факт, что мы можем решить, в каком поколении собирать объекты. В главе 5 объяснялось, что при проектировании было принято решение производить сборку мусора также в поколениях младше выбранного. Кроме того, вся куча больших объектов обрабатывается как поколение 2. Отсюда вытекают следующие возможные сценарии:

- выбрано поколение 0 – сборка мусора производится только в поколении 0;
- выбрано поколение 1 – сборка мусора производится только в поколениях 0 и 1;
- выбрано поколение 2 – сборка мусора производится во всех трех поколениях 0, 1 и 2, а также в куче больших объектов. Такую ситуацию называют *полной сборкой мусора* (или полной GC).

В процессе работы сборщик мусора проверяет достижимость объектов (и помечает их) только в выбранном и младших поколениях. И каждый раз решает, что он будет делать: только очистку или также уплотнение.

Представим все возможные случаи наглядно, как на рис. 5.5 из главы 5. Потратьте какое-то время, чтобы полностью уложить в голове эти сценарии, потому что в них заключена самая суть работы GC в .NET.

Сначала предположим, что в какой-то момент память программы выглядит, как на рис. 7.1. Памятуя изложенное в главе 5, мы легко распознаем типичную ситуацию – один блок памяти, который содержит сегменты SOH (эфемерный) и LOH. Сегмент SOH разбит на поколения 0, 1 и 2. В каждом поколении имеются объекты и обозначены границы между поколениями.

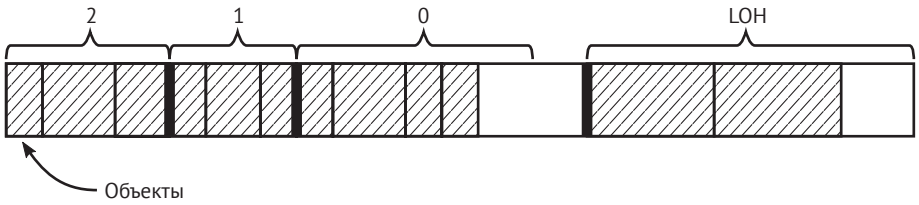


Рис. 7.1 ❖ Начальное состояние памяти, предполагаемое на следующих трех рисунках. Объекты обозначены косой штриховкой. В конце поколения 0 есть свободное место. Сегмент SOH еще не полностью занят поколениями

Теперь рассмотрим случай, когда для сборки выбрано поколение 0 (рис. 7.2). Тогда на этапе пометки анализируется только достижимость объектов в поколении 0. Предположим, что всего один объект в поколении 0 помечен как достижимый (см. рис. 7.2a, на котором помеченные объекты закрашены серым цветом). Теперь GC должен принять решение о способе очистки:

- сборка очисткой (рис. 7.2b) – в таком случае все недостижимые объекты из поколения 0 считаются свободным пространством. Граница поколения 1

сдвинулась, чтобы вместить переведенный достижимый объект (наш единственный помеченный объект переведен в поколение 1). Как часто бывает при сборке очисткой, фрагментация поколения 1 сильно увеличилась – в нем образовалась большая дыра, не содержащая ни одного объекта[1];

- сборка с уплотнением (рис. 7.2c) – в таком случае достижимые объекты в поколении 0 уплотняются и включаются в растущее поколение 1. Фрагментация, конечно, отсутствует, но вся операция сложнее (требуется копировать память и обновлять ссылки на перемещенные объекты).

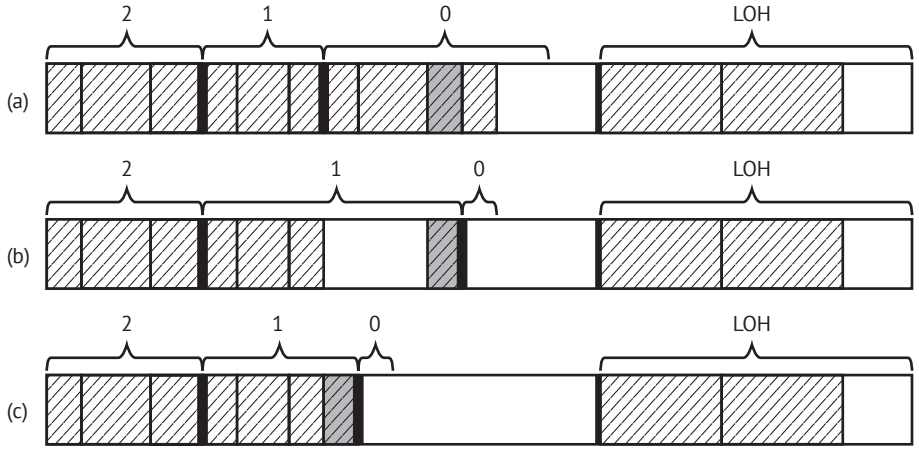


Рис. 7.2 ❖ Сборка мусора в выбранном поколении 0:
(a) объекты помечены как достижимые; (b) сборка очисткой; (c) сборка с уплотнением

Итак, в процессе сборки мусора в поколении 0:

- на достижимость проверяются только объекты в поколении 0 (помечаются);
- поколение 0 опустошается (в нем остается лишь немного места, и это сделано намеренно) – это поведение по умолчанию. Все объекты самого молодого поколения либо собираются, либо переводятся в старшее поколение. Как мы далее увидим, из этого правила есть исключения. Но пока будем рассматривать самый простой случай;
- достижимые объекты переводятся из поколения 0 в поколение 1;
- поколение 1 растет как в случае сборки очисткой (рост обусловлен увеличением фрагментации), так и в случае уплотнения (рост есть, но меньше);
- поколение 2 и LOH не изменяются. Однако они анализируются, и запоминается, на какие объекты поколения 0 из них есть ссылки (с помощью таблиц карт, описанных в главе 5).

Теперь рассмотрим случай, когда для сборки мусора выбрано поколение 1 (рис. 7.3). Тогда на этапе пометки анализируется достижимость объектов в поко-

[1] Из предыдущей главы мы знаем, что это место все-таки пригодно для использования – им управляет распределитель на основе списка свободных блоков. Но в поколениях 0 и 1 элементы такого списка проверяются только один раз, а затем отбрасываются, так что это свободное место очень быстро становится бесполезным. Однако следует помнить, что в поколениях 0 и 1 сборка производится очень часто, т. е. они и перестраиваются часто.

лениях 0 и 1. Снова предположим, что в поколении 0 достижимым помечен всего один объект, а в поколении 1 еще два объекта (рис. 7.3a). Теперь GC должна выбрать способ сборки:

- сборка очисткой (рис. 7.3b) – в таком случае все недостижимые объекты из поколений 0 и 1 считаются свободным местом. Границы поколений 2 и 1 сдвигаются, чтобы вместить переведенные достижимые объекты. При этом, как и прежде, образуется сильная фрагментация (в основном в поколении 1, но и поколение 2 тоже фрагментируется;
- сборка с уплотнением (рис. 7.3c) – в таком случае достижимые объекты в поколениях 0 и 1 уплотняются и включаются в поколения 1 и 2 путем сдвига их границ.

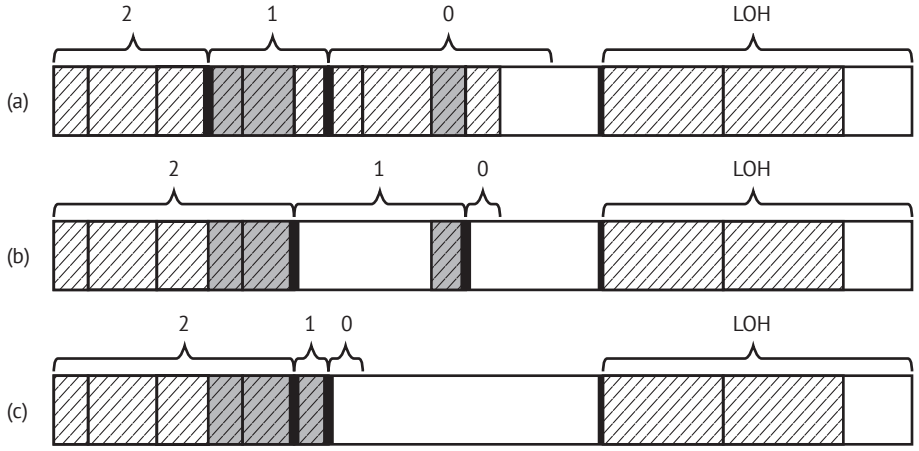


Рис. 7.3 ❖ Сборка мусора в поколении 1:
(a) объекты помечены как достижимые; (b) сборка очисткой; (c) сборка с уплотнением

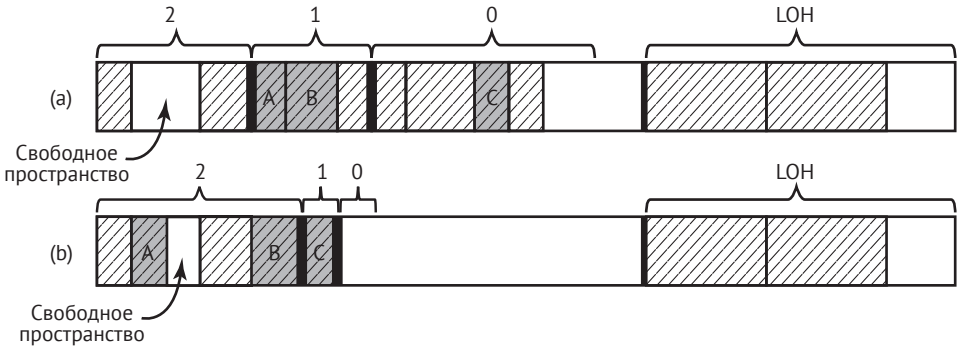
Итак, в процессе сборки мусора в поколении 1:

- на достижимость проверяются объекты в поколениях 0 и 1 (помечаются);
- поколение 0 опустошается;
- достижимые объекты из поколения 0 переводятся в поколение 1;
- достижимые объекты из поколения 1 переводятся в поколение 2;
- поколение 1 может как вырасти, так и уменьшиться – в зависимости от выбранного способа сборки. Это интересно, потому что теоретически поколение 1 может вырасти… в результате сборки в поколении 1. Понятно, что это вызвано фрагментацией, и маловероятно, что в нашем примере GC решит выбрать сборку очисткой. Но тем не менее теоретически и технически такой вариант исключать нельзя;
- поколение 2 растет;
- LOH не изменяется, но анализируется и запоминается, на какие объекты поколений 0, 1 и 2 из нее есть ссылки;
- сборка в поколении 1 несколько отличается от сборки в поколении 0 с точки зрения производительности. Ясно, что предстоит проанализировать и, возможно, переместить или модифицировать больше объектов. Однако в обо-

их случаях GC работает в пределах эфемерного сегмента (скорее всего, частично находящегося в кеше процессора), поэтому наблюдаемое различие не должно быть уж очень большим.

Если выбрано поколение 0 или 1, то можно воспользоваться еще одним способом перевода объектов. Вместо расширения следующего поколения с целью включить объекты, переведенные из выбранного, GC может «разместить их в старшем поколении», использовав для этой цели имеющееся там свободное место (управляемое списком свободных). Это позволяет воспользоваться фрагментацией (а заодно и уменьшить ее), а не слепо раздвигать границы области, занятой поколением.

В примере на рис. 7.3 один из объектов можно было бы разместить в доступном свободном пространстве:



Очевидно, что это имеет смысл только в случае сборки с уплотнением. В случае сборки очисткой объекты не копируются в другое место, поэтому нет никакой возможности поместить их в имеющееся свободное пространство.

И последний пример – собирается поколение 2 (рис 7.4). При такой полной сборке анализируется гораздо больше объектов, чем в двух предыдущих случаях. Поэтому надо стараться, чтобы полная сборка производилась как можно реже; мы обсудим этот вопрос ниже. В случае полной сборки на этапе пометки анализируется вся управляемая куча – поколения 0, 1, 2 и LOH. На рис. 7.4a показаны помеченные объекты в нашем примере. Теперь GC должен выбрать способ сборки:

- сборка очисткой (рис. 7.4b) – все недостижимые объекты из всех поколений (включая LOH) считаются свободным местом. Границы всех поколений сдвигаются соответственно. Заметим, что резко возросла фрагментация в основном в поколениях 2, 1 и LOH;
- сборка с уплотнением (рис. 7.4c) – все объекты в SOH уплотняются (напомним, что LOH автоматически не уплотняется). Это оптимально с точки зрения использования памяти, но, очевидно, требует больше времени для копирования большого количества объектов.

Итак, в процессе сборки мусора в поколении 2:

- на достижимость проверяются все объекты во всех поколениях и в LOH;
- поколение 0 опустошается;
- достижимые объекты из поколений 0 и 1 переводятся соответственно в поколения 1 и 2;

- достижимые объекты в поколении 2 остаются в нем же;
- в LOH также производится сборка мусора, но без уплотнения – появляется фрагментация, но это свободное пространство может быть использовано распределителем из списка свободных в LOH.

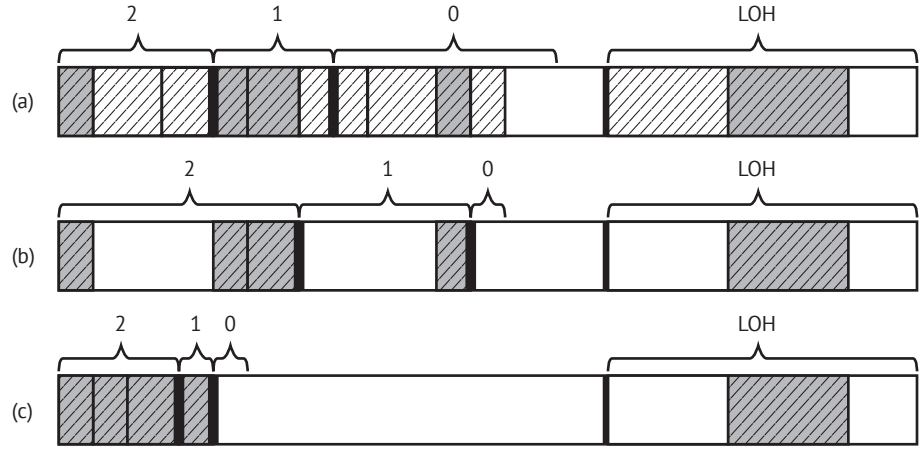


Рис. 7.4 ❖ Сборка мусора в поколении 2 (полная сборка):
(a) объекты помечены как достижимые; (b) сборка очисткой; (c) сборка с уплотнением

Внимательный читатель, вероятно, заметил, что после каждой сборки мусора в поколении 1 или 2 поколение 2 может вырасти внутри сегмента (если существует много долгоживущих объектов, которые не освобождаются). В конце концов, может наступить момент, когда оно станет настолько большим, что в поколениях 0 и 1 не останется места (рис. 7.5a). В таком случае простой сборки очисткой или с уплотнением будет недостаточно. GC может принять решение произвести сборку с уплотнением со следующими дополнительными шагами (рис. 7.5b):

- текущие эфемерные сегменты преобразуются в сегменты, содержащие только поколение 2 – все достижимые объекты из поколений 1 и 2 уплотняются там;
- создается новый эфемерный сегмент – все достижимые объекты из поколения 0 уплотняются там (становятся объектами поколения 1);
- в LOH производится сборка очисткой, как обычно.

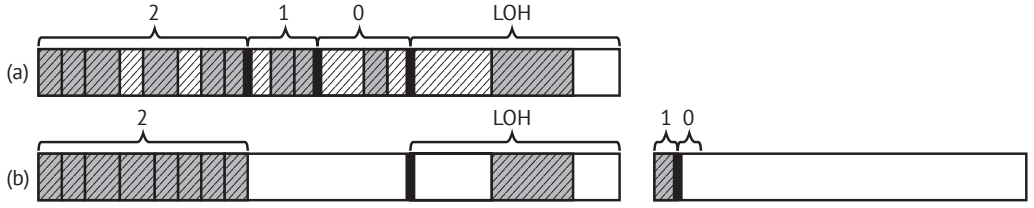


Рис. 7.5 ❖ Сборка мусора в выбранном поколении 2 (полная сборка), когда поколение 2 велико: (a) объекты помечены как достижимые; (b) сборка с уплотнением и созданием нового эфемерного сегмента

Таким образом, поколение 2 может расти «бесконечно». Если такая ситуация повторится в новом эфемерном сегменте, то он будет преобразован в сегмент, содержащий только поколение 2, и ситуация может развиваться по одному из трех сценариев:

- новый эфемерный сегмент создается путем резервирования и передачи памяти для нового сегмента, как показано на рис. 7.5;
- новый эфемерный сегмент создается из сегмента, находящегося в резервном списке сегментов, если этот список не пуст, – построение резервного списка сегментов было показано на рис. 5.22 (глава 5), где обсуждалось повторное использование сегментов. Для этого должно быть активировано придерживание виртуальной памяти, что бывает не всегда;
- существующий сегмент, содержащий только поколение 2 небольшого размера, может быть сделан новым эфемерным сегментом (рис. 7.6). В этом случае, даже если придерживание виртуальной памяти не активировано, создавать новый сегмент не придется. При этом старый эфемерный сегмент станет сегментом, содержащим только поколение 2.

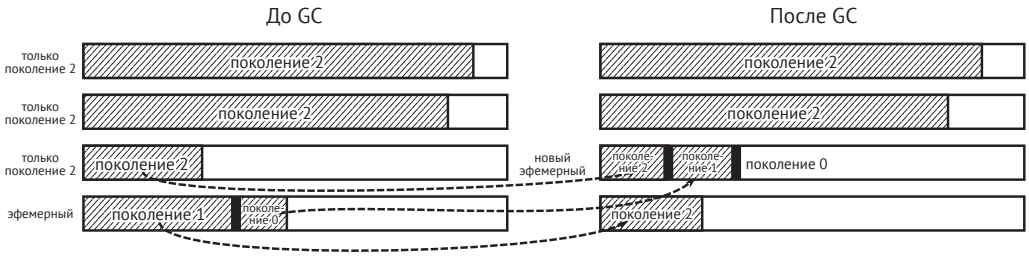


Рис. 7.6 ❖ Сборка мусора в поколении 2 (полная сборка). Сборка с уплотнением и преобразованием сегмента, содержащего только поколение 2, в новый эфемерный сегмент

Заметим, что преобразование текущего эфемерного сегмента в сегмент, содержащий только поколение 2, и образование нового эфемерного сегмента путем повторного использования существующего или создания нового могут быть вызваны чрезмерно большим количеством закрепленных объектов (pinned objects) – из-за этого эфемерные сегменты становится трудно использовать, поэтому весь сегмент отдается под поколение 2. С точки зрения требований к закреплению это вполне нормально, потому что адреса закрепленных объектов при этом не изменяются – просто такая область логически начинает представлять поколение 2.

Подчеркнем еще раз: полная сборка мусора включает пометку всех объектов во всех поколениях и в LOH. Они могут располагаться в нескольких сегментах, и если сборку переживает много объектов, то она обходится очень дорого. Кроме того, при этом может быть повторно использован сегмент, содержащий только поколение 2, или создан совсем новый сегмент. Таким образом, накладные расходы на полную сборку мусора могут быть во много раз больше (иногда на несколько порядков), чем при сборке в поколении 0 или 1, которая производится лишь в одном эфемерном сегменте, скорее всего, частично хранящемся в кеше процессора. Полной сборки мусора следует избегать любой ценой!

Шаги процесса сборки мусора

Познакомившись с тем, как в общем выглядит результат работы сборщика мусора, посмотрим, из каких шагов состоит этот процесс. Можно выделить следующие шаги верхнего уровня.

1. Запуск сборки мусора – какое-то событие привело к необходимости запустить GC.
2. Приостановка управляемых потоков – движку выполнения поступил запрос приостановить все потоки, исполняющие управляемый код (в случае неконкурентной GC – на все время сборки мусора).
3. Пользовательский поток начинает исполнять код GC – поток, запустивший GC, начинает исполнять код сборщика мусора.
4. Выбор поколения – первым делом GC на основе анализа различных условий решает, какое поколение выбрать для сборки мусора.
5. Пометка – производится пометка достижимых объектов в выбранном и более молодых поколениях.
6. Планирование – GC решает, стоит ли заниматься уплотнением или достаточно просто очистки. Хотя, на первый взгляд, это не очевидно, именно на этом шаге выполняется большая часть вычислений, необходимых для завершения GC.
7. Очистка или уплотнение – после того как решение принято, производится очистка или уплотнение с использованием той информации, которая была собрана на этапе планирования. Если было выбрано уплотнение, то необходимо выполнить дополнительный шаг перемещения ссылок, т. е. обновить адреса перемещенных объектов.
8. Возобновление управляемых потоков – движку выполнения поступает запрос возобновить все потоки, исполняющие управляемый код.

Это все шаги, выполняемые сборщиком мусора, и в главах с 8 по 10 мы их подробно опишем. Можно считать это картой, которая приведет нас к концу пути.

Часть диагностической информации генерируется сразу же во время выполнения шага, а часть – в конце процесса. Для этого используются хорошо знакомые механизмы счетчиков производительности и событий ETW/LTTng. Некоторые данные доступны командам SOS, так что для их получения нужно использовать WinDbg. Далее в этой главе мы воспользуемся этими данными и командами SOS.

Сценарий 7.1. Анализ использования GC

Описание. Мы хотим вести наблюдение за GC во время работы веб-приложения. Желательно сделать это без вмешательства в его работу во время нагрузочных тестов до развертывания в производственной среде. Тестируется приложение nopCommerce 4.0 – универсальная платформа электронной торговли с открытым исходным кодом, написанная для ASP.NET Core. Это продолжение сценария 5.1 из главы 5.

Анализ. Пропустим техническую часть, связанную с подготовкой к нагрузочному тестированию, будем считать, что все необходимые действия выполнены и инструменты в наличии. Для подготовки и выполнения нагрузочных тестов используется программа JMeter. Она отправляет около 7 запросов в секунду по простому сценарию (посетить домашнюю страницу, одну страницу продукта и одну страницу метки). Это в точности тот же тест, что был описан в сценарии 5.1. Но

на этот раз анализ продолжается всего 2 минуты, чтобы быстро оценить характер использования GC. Наблюдение ведется за самостоятельно размещенным (self hosted) веб-приложением .NET (процесс с именем `Nop.Web.exe`).

Для начала мы хотим посмотреть на общее потребление памяти .NET и использование GC в приложении. Для этого нам понадобятся следующие счетчики:

- `\Память CLR .NET (Nop.Web)\Размер кучи поколения 0 (\.NET CLR Memory(Nop.Web)\Gen 0 heap size)` (а на самом деле бюджет выделения поколения 0, как было объяснено в предыдущих главах);
- `\Память CLR .NET (Nop.Web)\Размер кучи поколения 1 (\.NET CLR Memory(Nop.Web)\Gen 1 heap size)`;
- `\Память CLR .NET (Nop.Web)\Размер кучи поколения 2 (\.NET CLR Memory(Nop.Web)\Gen 2 heap size)`;
- `\Память CLR .NET (Nop.Web)\Размер кучи массивных объектов (\.NET CLR Memory(Nop.Web)\Large Object Heap size)`;
- `\Память CLR .NET (Nop.Web)\% времени в GC (\.NET CLR Memory(Nop.Web)\% Time in GC)`.

Результаты первых двух минут работы приложения показаны на рис. 7.7 и 7.8. Видно, что размеры поколений вполне стабильны – размеры эфемерных поколений изменяются часто, но не растут. Размер самого старого поколения стабилизировался на уровне 89 520 308 байт. Однако процент времени в GC настораживает. Среднее значение равно 24 % (это ясно видно на рис. 7.8), а это означает, что четверть всего времени процесс тратит на сборку мусора. Это значительные издержки.

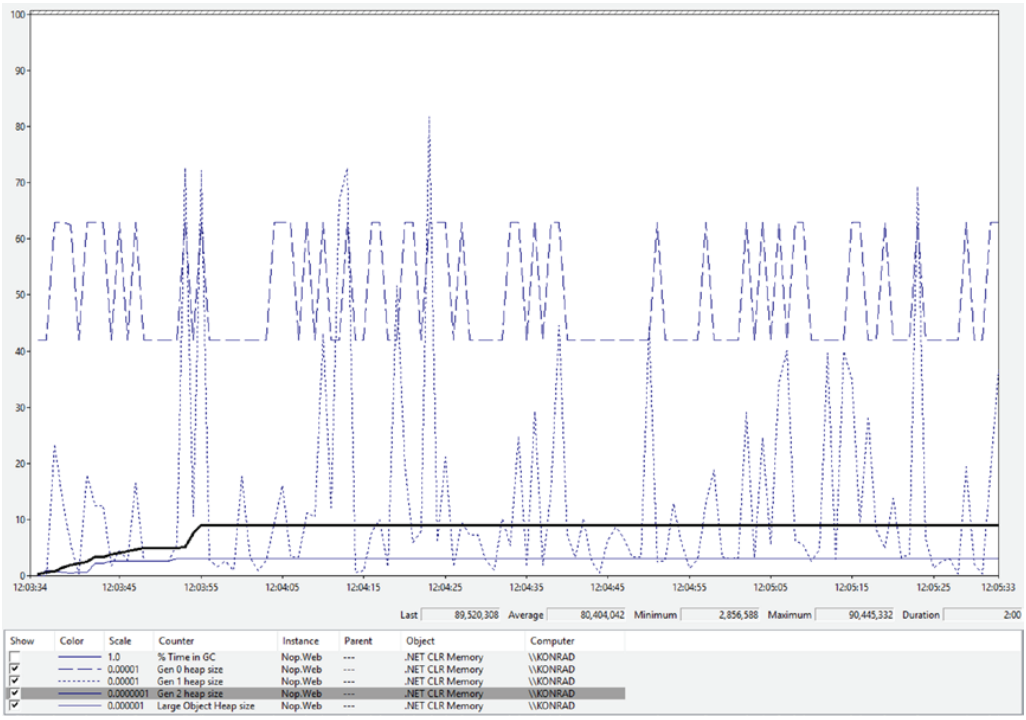


Рис. 7.7 ❖ Размер поколений в мониторе производительности на протяжении двухминутного нагрузочного теста приложения для ASP.NET Core

Анализ ситуации можно продолжить с помощью просмотра событий ETW в PerfView. Если в диалоговом окне **Collect** указать вариант **GC Collect Only**, то будут регистрироваться события от поставщиков Microsoft-Windows-DotNETRuntime с ключевым словом GC. Когда сборка мусора остановится и обработка завершится, мы сможем исследовать работу GC с помощью отчета **GCStats** в папке **Memory Group**.

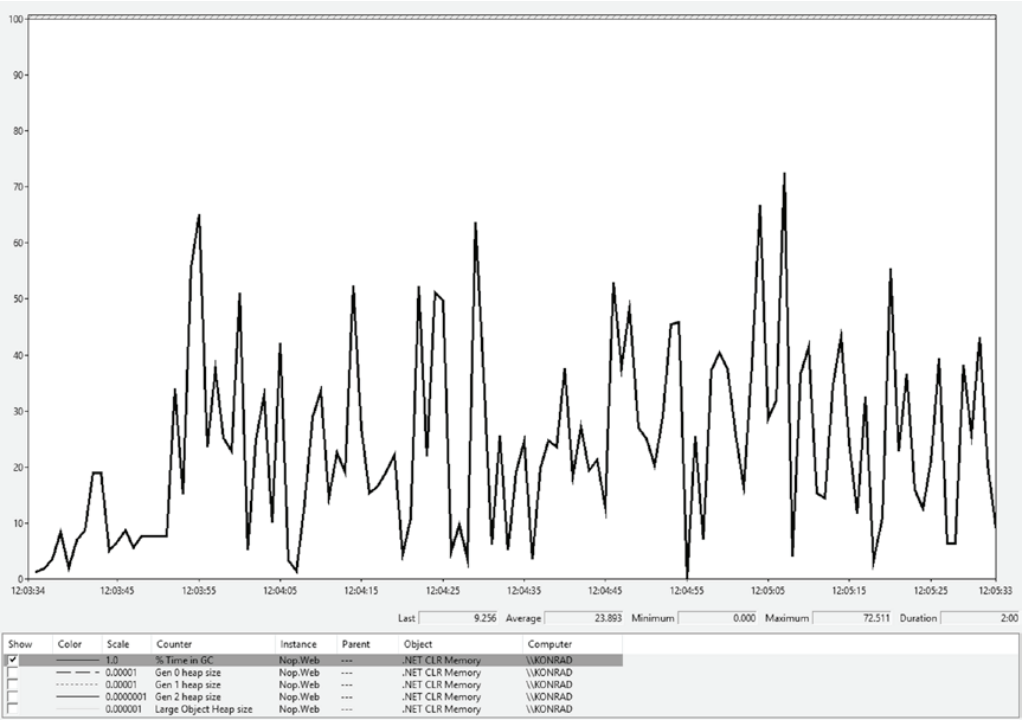


Рис. 7.8 ❖ Использование GC в мониторе производительности на протяжении двухминутного нагрузочного теста приложения для ASP.NET Core

В отчете GCStats представлена подробная статистика относящихся к GC событий от всех поставщиков среды выполнения .NET на протяжении сеанса записи. В начале отчета перечислены все поставщики, мы выбираем процесс `Nop.Web`. Сначала представлены различные диагностические данные (рис. 7.9). Например, `None` в строке CLR Startup Flags означает, что сборка мусора была запущена в неконкурентном режиме рабочей станции.

GC Stats for Process 8724: nop.web

- CommandLine: F:\IIS\nopCommerce\Nop.Web.exe
- Runtime Version: V 4.0.30319.0
- CLR Startup Flags: None
- Total CPU Time: 0 msec
- Total GC CPU Time: 0 msec
- Total Allocs : 14,483.212 MB
- GC CPU MSec/MB Alloc : 0.000 MSec/MB
- Total GC Pause: 12,350.2 msec
- % Time paused for Garbage Collection: 10.2%
- % CPU Time spent Garbage Collecting: NaN%
- Max GC Heap Size: 86.492 MB
- GC Perf Users Guide
- GCs that > 200 msec Events
- LOH allocation pause (due to background GC) > 200 msec Events
- GCs that were Gen2
- Individual GC Events
  - View in Excel
- Per Generation GC Events in Excel
- Raw Data XML file (for debugging)
- *No finalized object counts available. No objects were finalized and/or the trace did not include the necessary information.*

Рис. 7.9 ❖ Начало отчета GCStats для процесса Nop.Web

Нам интереснее следующая таблица – **GC Rollup by Generation** (Сводка GC по поколениям) (рис. 7.10). В ней показана сводная информация обо всех сборках мусора, имевших место в выбранном процессе на протяжении двухминутного сеанса ETW. Как видим, за это время произошло 3016 сборок мусора (т. е. в среднем 25 сборок в секунду). Общее время приостановки потоков составило свыше 12 секунд. Это означает, что примерно 10 % времени было потрачено в GC, тогда как в типичной ситуации эта величина никак не должна превышать нескольких процентов. Обратите также внимание на сборки мусора в поколении 2, которые занимают значительно больше времени, чем сборки в младших поколениях (столбец **Mean Pause** (Средняя пауза) на рис. 7.10).

GC Rollup By Generation										
All times are in msec.										
Gen	Count	Max Pause	Max Peak MB	Max Alloc MB/sec	Total Pause	Total Alloc MB	Alloc MB/ MSec GC	Survived MB/ MSec GC	Mean Pause	Induced
ALL	3016	91.4	86.5	3,100.147	12,350.2	14,483.2	1.2	∞	4.1	0
0	1932	13.0	78.5	3,033.417	5,087.4	8,248.1	0.4	∞	2.6	0
1	1059	22.2	86.5	3,100.147	5,147.7	6,083.1	0.4	∞	4.9	0
2	25	91.4	84.2	2,264.039	2,115.1	152.0	0.1	∞	84.6	0

Рис. 7.10 ❖ Сводка GC по поколениям из отчета GCStats для процесса Nop.Web

Следует обратить внимание на очень большое число операций выделения памяти. Всего под объекты выделено выше 12 ГБ! И хотя, как мы видели на рис. 7.7, размеры поколений стабильны, очевидно, что это свидетельствует о создании огромного количества короткоживущих временных данных, которые быстро становятся мусором.

Для дальнейшего анализа можно воспользоваться весьма интересной таблицей **GC Events by a Time** (Распределение событий GC по времени) из того же отчета GCStats (рис. 7.11). В ней приведены исключительно полезные сведения обо всех сборках мусора на протяжении сеанса. В случае длинного сеанса таблица усекается (как на рисунке), но всегда можно получить также исходные данные в формате CSV и просмотреть их, например, в Excel.

GC Events by Time

All times are in msec. Hover over columns for help.

GC Index	Pause Start	Trigger Reason	Gen	Suspend Msec	Pause MSec	% Pause Time	% GC	Gen0 Alloc MB	Gen0 Alloc Rate MB/sec	Peak MB	After MB	Ratio Peak/After	Promoted MB	Gen0 MB	Gen0 Survival Rate %	Gen0 Frag %	Gen1 MB	Gen1 Survival Rate %	Gen1 Frag %	Gen2 MB	Gen2 Survival Rate %	Gen2 Frag %	LOH MB	LOH Survival Rate %	LOH Frag %	Finalizable Surv MB	Pinned Obj
2016 Beginning entries truncated, use View in Excel to view all...																											
2184	83,609.139	AllocSmall	1N	0.091	5.867	50.3	NaN	4.147	714.17	70.049	70.102	1.00	0.439	12.961	7	99.22	0.209	58	0.00	53.857	NaN	0.01	3.155	NaN	4.70	0.01	15
2185	83,645.296	AllocSmall	1N	0.012	4.875	13.9	NaN	4.135	136.47	70.182	70.312	1.00	0.503	12.897	8	99.20	0.273	62	0.00	53.987	NaN	0.01	3.155	NaN	4.70	0.02	16
2186	83,720.348	AllocSmall	1N	0.009	4.851	6.5	NaN	4.153	59.18	70.312	70.530	1.00	0.572	12.928	8	99.11	0.242	79	0.00	54.206	NaN	0.01	3.155	NaN	4.70	0.02	16
2187	83,733.031	AllocSmall	1N	0.094	4.631	37.1	NaN	4.148	529.28	70.530	61.478	1.15	0.878	3.277	16	98.71	0.682	65	0.00	54.364	NaN	0.01	3.155	NaN	4.70	0.00	12
2188	83,741.551	AllocSmall	0N	0.065	5.152	56.9	NaN	4.105	1,053.70	62.451	61.478	1.02	0.691	2.596	16	99.46	1.363	NaN	0.00	54.364	NaN	0.00	3.155	NaN	0.00	0.00	4
2189	83,751.447	AllocSmall	1N	0.126	8.001	62.7	NaN	8.214	1,728.88	67.284	59.627	1.13	2.095	0.000	11	0.00	0.921	86	0.00	55.551	NaN	0.01	3.155	NaN	4.70	0.00	0

Рис. 7.11 ❖ Таблица GC Events by Time из отчета GCStats для процесса Nop.Web

Из показанного фрагмента таблицы (таблица целиком, по понятным причинам, не приведена) можно установить ряд интересных фактов:

- все GC были запущены по причине AllocSmall, т. е. вследствие выделения памяти в SOH;
- многие GC запускались в течение одной секунды (см. изменения в столбце Pause Start), и памяти выделялось очень много (см. столбец Gen0 Alloc MB) – это подтверждает наше подозрение о чрезмерном выделении.

На этой стадии нужно бы исследовать, что именно выделяется так часто, как мы делали в сценарии 6.2 из главы 6.

В последующих сценариях мы еще будем возвращаться к различным столбцам таблицы GC Events by Time. По мере продвижения вперед мы станем лучше понимать все новые части отчета GCStats. И в конечном итоге он станет понятен целиком.

Обратите внимание на столбец Gen, в котором указано не только выбранное поколение, но и тип GC:

- N – неконкурентная GC (блокирующая);
- B – фоновая GC;
- F – приоритетная GC (блокирующая сборка эфемерных поколений в ходе фоновой сборки);
- I – принудительная (запущенная вручную) блокирующая GC;
- i – принудительная неблокирующая GC.

Профилирование GC

Чтобы составить приблизительное представление о сравнительной стоимости отдельных шагов, взгляните на рис. 7.12, где с помощью средств профилирова-

ния ETW CPU собраны данные во время простого нагрузочного теста (описанного в сценарии выше). В столбце Inc показано полное время (в миллисекундах), проведенное в методе (и методах, которые он вызывает). В тестируемом приложении сборщик мусора работал в режиме рабочей станции. На протяжении теста имело место 627 сборок мусора (как следует из отчета ETW, который здесь не показан), т. е. среднее время паузы составило 4,33 мс.

Methods that are called by clr!WKS::gc_heap::garbage_collect

Name ?	Inc % ?	Inc ?	Exc % ?	Exc ?
☑ clr!WKS::gc_heap::garbage_collect	6.1	2,717.8	0.0	0
+ ☑ clr!WKS::gc_heap::gc1	6.0	2,700.8	0.0	0
I + ☑ clr!WKS::gc_heap::plan_phase	3.1	1,408.8	0.6	282
I I + ☐ clr!WKS::gc_heap::relocate_phase	1.9	864.6	0.0	0
I I + ☐ clr!WKS::gc_heap::compact_phase	0.3	143.8	0.0	2
I + ☐ clr!WKS::gc_heap::mark_phase	2.9	1,276.4	0.0	0

Рис. 7.12 ❖ Данные профилирования различных шагов GC, полученные для сборщика мусора в режиме рабочей станции

Стоимость шагов пометки и планирования примерно одинакова. Код GC устроен так, что этап планирования содержит шаги уплотнения и перемещения ссылок. Удивительно, что перемещение ссылок (обновление адресов) занимает больше времени, чем само уплотнение (перемещение объектов).

Но не надо придавать этим цифрам слишком большое значение. Они могут существенно изменяться в зависимости от различных условий, в т. ч. коэффициента выживания объектов, количества ссылок между объектами или общего количества объектов. Если вам это интересно, проведите собственное расследование конкретно для своего сценария. Нужно лишь использовать PerfView в два этапа:

- запустить сеанс ETW, включив профилирование процессора, для чего нужно выбрать режим CPU Samples. Возможно, вы захотите уменьшить интервал опроса в поле CPU Sample Interval MSec (по умолчанию он равен 1 мс), чтобы повысить точность результатов;
- проанализировать собранные данные в представлении CPU Stacks – вероятно, вам потребуется выполнить два простых изменения (не забудьте отменить всякую группировку):
  - найдите строку `clr?!` или `coreclr?!` (для полного .NET Framework и для .NET Core соответственно) и выполните для нее команду **Lookup Symbols**;
  - найдите метод `garbage_collect` и приступайте к исследованию, выполнив команду **Goto Item in Callees**.

Может возникнуть несколько вопросов по поводу природы операций GC, особенно о влиянии следующих условий на общую стоимость GC (в терминах использования ресурсов процессора и времени работы):

- большое общее количество объектов – чем больше объектов, тем больше работы предстоит сделать на этапе планирования. Он заключается в просмотре всей управляемой кучи, поэтому естественно, что увеличение количества объектов увеличивает и время планирования. Впрочем, поскольк-

ку доступ к памяти строго линейный (объект за объектом), это увеличение смягчается механизмами кеширования;

- большое количество выживших объектов – чем больше объектов выжило, тем больше работы будет на этапе пометки. При этом обход управляемой кучи будет неструктурированным (плохо поддается кешированию). Накладные расходы будут тем выше, чем больше существует ссылок между объектами. Кроме того, если выполняется сборка с уплотнением, то большое количество выживших объектов означает, что велико количество операций копирования памяти и обновления многочисленных ссылок. Этап планирования не столь чувствителен к количеству выживших объектов – он работает с «заполненными блоками» (см. главу 9), состоящими из многих живых объектов, так что стоимость амортизируется.

Следствия этого просты и интуитивно понятны – чем меньше создано объектов, тем лучше. Например, лучше создать один большой массив в LOH и многократно использовать его участки (например, с помощью `Span<T>`), чем создавать много мелких массивов.

Данные для настройки производительности сборки мусора

Прежде чем пускаться в путешествие по последующим этапам работы GC, стоит обратить внимание на данные, которые он использует. Часто приходится слышать о различных «эвристиках» или «внутренних настройках» GC. Вот о них-то мы и поговорим в этом разделе.

Данные, используемые GC, можно разделить на две большие группы: статические и динамические. Обе играют важную роль в том, что и как делает GC. Описывать их очень уж подробно не имеет особого смысла, т. к. это глубоко скрытая деталь реализации. Нет никаких гарантий, что состав этих данных и их значения останутся неизменными в следующих версиях .NET.

С другой стороны, эти данные настолько важны и так сильно влияют на работу GC, что совсем не упомянуть их при описании процесса в целом было бы неправильно. Трудно ожидать, что в ближайшем будущем функционирование наиболее важных индикаторов претерпит существенные изменения. Их мы и рассмотрим ниже.

Статические данные

Статические данные представляют конфигурацию, которая задана при запуске среды выполнения и больше не изменяется. В их состав входят следующие атрибуты для каждого поколения:

- минимальный размер – минимальный бюджет выделения (этот термин подробно объясняется несколькими абзацами ниже);
- максимальный размер – максимальный бюджет выделения;

- предельная фрагментация и предельный коэффициент фрагментации – используются для решения о том, производить ли уплотнение;
- лимит и максимальный лимит – используется, когда нужно вычислить увеличение бюджета выделения для поколения;
- time_clock – время, по истечении которого нужно производить сборку мусора в поколении, указывается в терминах счетчика производительности (см. QueryPerformanceCounter);
- gc_clock – количество GC, по достижении которого производить сборку мусора в текущем поколении.

> В случае CoreCLR описанные выше статические данные представлены структурой `static_data`, определенной в файле `.\src\gc\gcpriv.h`. Затем в файле `.\src\gc\gc.cpp` инициализируется статическая таблица `static_data_table` для двух режимов задержки. Некоторые значения вычисляются при запуске среды выполнения в методе `gc_heap::init_static_data`.

Статические данные настраиваются в зависимости от конфигурации уровня задержки GC (обсуждается в главе 11). В настоящее время существует два режима, статические данные для них отличаются в основном размерами поколений:

- *сбалансированный* – паузы чаще и более предсказуемы, оптимизирован баланс между задержкой и объемом занятой памяти. Это режим по умолчанию;
- *минимального потребления памяти* – оптимизирован с целью минимизации потребляемой памяти, паузы чаще и длительнее.

В табл. 7.1 и 7.2 приведены значения статических данных для обоих режимов задержки (в предположении, что компьютер оснащен L3-кешем размера 8 МБ). Из них можно почерпнуть интересную информацию:

- минимальный бюджет поколения 0 тесно связан с размером кеша процессора, и это понятно, если вспомнить (см. главу 2), как важен кеш. Этот параметр гарантирует, что наиболее востребованному поколению 0 достанется приличная часть кеша;
- максимальные бюджеты выделения обоих эфемерных поколений тесно связаны с размером эфемерного сегмента. Если вспомнить физическую организацию памяти (см. главу 5), то и это обретет смысл. Эти параметры особенно важны в режиме рабочей станции и в серверном режиме для 32-разрядной среды, поскольку тогда эфемерный сегмент относительно мал (см. табл. 5.3);
- максимальный бюджет выделения поколения 2 и кучи больших объектов ограничен только максимальным адресом (`SSIZE_T_MAX` равно половине размера слова), и это тоже вполне разумно, потому что в этих двух местах собираются все долгоживущие объекты. Поэтому пространство для них должно быть логически «не ограничено», чтобы был возможен любой сценарий использования памяти. Понятно, что эти размеры лимитированы физическими ресурсами (объемом ОЗУ и размером файла подкачки, пределами адресации).

Таблица 7.1. Статические данные GC – сбалансированный режим (в предположении, что имеется L3-кеш размером 8 МБ)

	Минимальный бюджет выделения	Максимальный бюджет выделения	Предельная фрагментация	Предельная доля фрагментации	Лимит	Максимальный лимит	time_clock	gc_clock
Gen0	1) 4/15 МБ	2) 6–200 МБ	40 000	0,5	9,0	20,0	1000 мс	1
Gen1	160 КБ	3) не менее 6 МБ	80 000	0,5	2,0	7,0	10 000 мс	10
Gen2	256 КБ	SSIZE_T_MAX	200 000	0,5	1,2	1,8	100 000 мс	100
LOH	3 МБ	SSIZE_T_MAX	0	0,0	1,25	4,5	0 мс	0

Таблица 7.2. Статические данные GC – режим минимального потребления памяти (в предположении, что имеется L3-кеш размером 8 МБ)

	Минимальный бюджет выделения	Максимальный бюджет выделения	Предельная фрагментация	Предельная доля фрагментации	Лимит	Максимальный лимит	time_clock	gc_clock
Gen0	1) 4/15 МБ	2) 6–200 МБ	40 000	0,5	4) 9,0/20,0	4) 20,0/40,0	1000 мс	1
Gen1	288 КБ	3) не менее 6 МБ	80 000	0,5	2,0	7,0	10 000 мс	10
Gen2	256 КБ	SSIZE_T_MAX	200 000	0,5	1,2	1,8	100 000 мс	100
LOH	3 МБ	SSIZE_T_MAX	0	0,0	1,25	4,5	0 мс	0

1. Минимальный бюджет выделения связан с размером кеша процессора (здесь предполагается 8 МБ) и вычисляется по-разному для разных микросхем (производителями оборудования). В общем случае он немного меньше для режима рабочей станции (первое число), чем для серверного режима (второе число).
2. В конкурентном режиме рабочей станции – 6 МБ. В серверном режиме и в неконкурентном режиме рабочей станции – половина размера эфемерного сегмента (см. табл. 5.3), но не менее 6 МБ и не более 200 МБ.
3. В конкурентном режиме рабочей станции – 6 МБ. В серверном режиме и в неконкурентном режиме рабочей станции – половина размера эфемерного сегмента (см. табл. 5.3), но не менее 6 МБ.
4. Значения для режима рабочей станции и для серверного режима соответственно.

Все эти лимиты, особенно минимальный и максимальный размеры каждого поколения, будут подробно объяснены ниже в этой главе.

Сборщик мусора использует эти данные для принятия различных решений. Мы еще не раз вернемся к ним впоследствии.

Динамические данные

Динамические данные представляют текущее состояние управляемой кучи с точки зрения поколения. Они обновляются в процессе сборки мусора, чтобы вычислить данные, необходимые для принятия различных решений (в т. ч. о том, использовать уплотнение или нет, является ли поколение «заполненным» и, стало быть, нуждающимся в сборке мусора и т. д.). В состав динамических данных входят различные атрибуты для каждого поколения, перечислим самые важные:

- *бюджет выделения* (или «желательное выделение») – сколько места GC хотел бы потратить на новые выделения перед запуском следующей сборки мусора;
- *новое выделение* – сколько места остается для выделения до следующей сборки мусора при текущем бюджете выделения;
- *фрагментация* – сколько всего места занято свободными объектами в этом поколении;
- *размер выживших* – суммарный размер выживших объектов;
- *размер закрепленных выживших* – суммарный размер выживших закрепленных заполненных блоков (подробно описывается ниже в этой главе);
- *коэффициент выживания* – отношение количества выживших байтов к общему количеству байтов;
- *текущий размер* – суммарный размер всех объектов после сборки мусора (не включает фрагментированную память);
- *gc «clock»* – количество сборок мусора для этого поколения;
- *time_clock* – время начала последней сборки мусора в этом поколении.

Атрибут *новое выделение* важен для кооперации распределителя и сборщика мусора. Он отслеживает, какая часть бюджета выделения в данном поколении уже потрачена. Если величина становится отрицательной, значит, бюджет выделения превышен и в поколении запускается сборка мусора.

Это подводит нас к одному из самых важных атрибутов – *бюджету выделения*. Это размер памяти, который GC готов потратить на выделение под новые объекты в данном поколении. Как мы помним (см. главу 6), пользовательский код может инициировать выделение памяти только в поколении 0 и в LOH. Однако бюджет выделения вычисляется для каждого поколения. Это кажущееся противоречие легко объяснить, если вспомнить, что перевод объектов из младшего поколения рассматривается как выделение памяти в старшем. При описании этапа планирования мы увидим, что GC пользуется внутренним распределителем, чтобы найти место для переведенных объектов (а заодно поймем, что эта фраза – лишь упрощение, на которое мы пошли ради краткости). В обоих случаях на выделение памяти расходуется бюджет.

Бюджет выделения динамически изменяется после каждой сборки мусора в данном поколении. Новое значение рассчитывается, главным образом исходя из коэффициента выживания. Если он высок (много объектов пережили сборку), то бюджет выделения увеличивается более агрессивно в надежде, что во время следующей сборки мусора в этом поколении отношение числа мертвых объектов к числу живых улучшится. В конце GC пересчитывается на основе коэффициента выживания – отношения суммарного размера выживших объектов к суммарному размеру объектов в начале GC (без учета фрагментации). Если коэффициент пре-

вышает некоторый порог, то новый бюджет выделения будет совпадать с максимальным бюджетом. А если он достаточно низок, то в качестве нового бюджета будет выбрано значение, близкое к минимальному. Иногда новое значение уточняется с применением линейной модели, которая вычисляет линейную комбинацию текущего и предыдущего бюджетов выделения.

На рис. 7.13 показан график зависимости нового бюджета выделения от коэффициента выживания. Крутизна кривой, порог выхода на плато и некоторые менее важные свойства функции зависят от статических параметров limit и max_limit, представленных в табл. 7.1 и 7.2. Чем меньше значения этих лимитов, тем круче кривая и тем быстрее она выходит на плато.

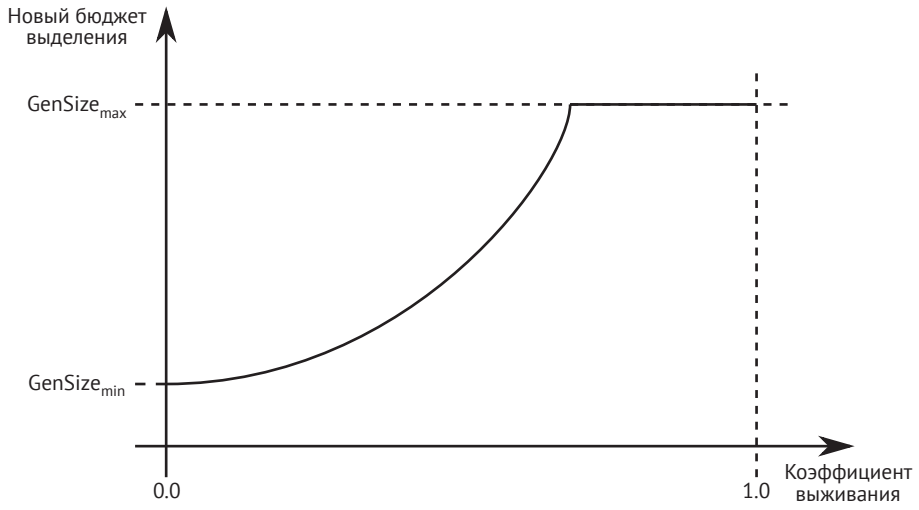


Рис. 7.13 ❖ График типичной функции, описывающей связь между коэффициентом выживания и новым бюджетом выделения

Глядя на значения в табл. 7.1 и 7.2, мы понимаем, что младшие поколения гораздо более динамично реагируют на коэффициент выживания, чем старшие. А поколение 0 так живо отвечает на него, что новый бюджет выделения чаще всего совпадает с одним из граничных значений – минимальным или максимальным размером поколения.

> Именно поэтому счетчик производительности `Память CLR .NET/Размер кучи поколения 0`, который, по историческим причинам, отражает бюджет выделения поколения 0, очень часто принимает только два значения в течение всего времени работы приложения. Это отчетливо видно на рис. 5.6 и 5.7 из главы 5 и на рис. 7.7, где величина этого счетчика постоянно перескакивает со значения 4 МБ на значение 6 МБ и обратно. Это, в свою очередь, означает, что GC работал в конкурентном режиме рабочей станции (см. значения параметров в табл. 7.1 и 7.2).

На этапе инициализации среды выполнения бюджету выделения для каждого поколения присваивается минимальная величина в соответствии со статическими данными (см. табл. 7.1 и 7.2). Как связаны размер поколения и его бюджет выделения? Тут важно понимать, что бюджет выделения – это логическая величина.

Он представляет лимит выделения в данном поколении, который может быть исчерпан, но в будущем изменится, применяясь к новым условиям. Быстрое выделение памяти в поколении прекращается, когда лимит исчерпан, но сам лимит может изменяться. Можно считать, что бюджет выделения динамически реагирует на коэффициент выживания, и в результате размер поколения динамически изменяется, стремясь принять оптимальное значение.

> Заметим, что часто задаваемый вопрос о «размере поколения по умолчанию» поставлен неправильно. В момент создания поколения просто пусты, нет такого понятия, как «размер по умолчанию». По мере создания и перевода объектов поколения растут в соответствии с бюджетами выделения.

Связь между новым выделением, бюджетом выделения и размером поколения проще всего описать с помощью метода `current_generation_size` из исходного кода CoreCLR (листинг 7.1). В любой момент приблизительный размер данных в поколении (без учета фрагментации) равен текущему размеру данных плюс разность между бюджетом выделения и новым выделением. В конце сборки мусора новое выделение устанавливается равным бюджету выделения. При создании объектов в поколении 0 и в LOH новое выделение соответственно уменьшается. Таким образом, объем выделения с момента последней сборки мусора равен разности между этими двумя значениями.

Листинг 7.1 ❖ Метод вычисления текущего размера поколения (взят из исходного кода CoreCLR)

```
size_t gc_heap::current_generation_size (int gen_number)
{
   dynamic_data* dd = dynamic_data_of (gen_number);
   size_t gen_size = (dd_current_size (dd) + dd_desired_allocation (dd) -
                      dd_new_allocation (dd));
   return gen_size;
}
```

У внимательного читателя мог возникнуть вопрос: «Как это – новое выделение обновляется после каждого создания объекта?» В главе 6 об этом ничего не было сказано. Также трудно ожидать, что это может происходить на быстром пути выделения, показанном в листинге 6.7, или где-то по дороге. Что ж, подозрение обоснованное. На самом деле новое выделение уменьшается только в результате создания или роста контекста выделения памяти – единицы памяти, выдаваемой GC.

Если вы хотите лучше разобраться в том, как бюджет выделения влияет на работу GC и как он связан с размером поколения, ознакомьтесь со сценарием 7.2, в котором продемонстрированы первые пять сборок мусора во взятом для примера процессе.

Сценарий 7.2. Демонстрация бюджета выделения

Описание. Мы хотим лучше понять концепцию бюджета выделения, в особенности его связь с размером поколения и общее влияние на работу GC. Это полезно не только в целях обучения. К такому подробному анализу можно прибегнуть, когда требуется выяснить, что именно активирует сборку мусора в процессе.

Анализ. Для анализа нет ничего лучше хорошего сеанса отладки. Была подготовлена простая программа на C# (листинг 7.2). Она в цикле создает миллион массивов байтов и сохраняет ссылки на них в дополнительном массиве, поэтому все объекты достижимы (переживают GC) на протяжении всего времени работы приложения. Размер каждого массива равен 25 024 байтам (25 000 байт данных плюс 8 байт для длины массива и 16 байт для метаданных объекта).

Листинг 7.2 ❖ Программа, использованная в этом сценарии

```
1 static void Main(string[] args)
2 {
3   Console.ReadLine();
4   Console.WriteLine("Hello, Windows");
5   Console.WriteLine("С любовью от CoreCLR.");
6   GC.Collect();
7   Console.ReadLine();
8   const int LEN = 1_000_000;
9   byte[][] list = new byte[LEN][];
10    for (int i = 0; i < LEN; ++i)
11    {
12       list[i] = new byte[25000];
13       if (i % 100 == 0)
14       {
15          Console.WriteLine("Выделено 100 массивов");
16       }
17    }
18 }
```

Благодаря средствам отладки в Visual Studio и протоколированию событий ETW первые пять сборок мусора можно полностью описать в терминах бюджета выделения, что позволяет лучше понять смысл этой концепции.

В нашем эксперименте упор сделан на простейшем варианте GC – неконкурентная рабочая станция в режиме минимизации потребляемой памяти. Значения статических данных на машине автора показаны в табл. 7.3 (вычислены на основе данных в табл. 7.2). Максимальный размер эфемерных поколений равен 128 МБ, потому что в этой конфигурации размер эфемерного сегмента составляет 256 МБ (см. табл. 5.3 в главе 5).

Таблица 7.3. Пример статических данных GC – неконкурентный GC на 64-разрядной рабочей станции в режиме минимизации потребляемой памяти (предполагается, что компьютер оснащен L3-кешем размером 8 МБ)

	min_size	max_size	limit	max_limit
Gen0	4 МБ	128 МБ	9,0	20,0
Gen1	288 КБ	128 МБ	2,0	7,0
Gen2	256 КБ	SSIZE_T_MAX	1,2	1,8
LOH	3 МБ	SSIZE_T_MAX	1,25	4,5

Для получения полной информации мы запустили среду выполнения CoreCLR в режиме отладки и расставили несколько точек останова в процессе, чтобы по-

смотреть значение «нового выделения» для каждого поколения. Понятно, что для обычного анализа проблемы это не нужно, а можно было бы ограничиться только данными ETW, как показано ниже.

В PerfView можно получить следующие данные, собранные во время сеанса ETW, для чего следует экспортировать в Excel таблицу **Per Generation GC Events** (События GC для каждого поколения) из отчета GCStats:

- размеры поколений в начале GC (*Begin size*) – из столбцов Before0/1/2/3;
- бюджеты выделения (*Allocation budget*) – из столбцов Budget0/1/2/3. Дополнительно бюджет поколения 0 показан в поле `FinalYoungestDesired` события `Microsoft-Windows-DotNETRuntime/GC/GlobalHeapHistory` и, как уже было сказано, в счетчике производительности `Память CLR .NET/Размер кучи поколения 0 (.NET Memory/Gen 0 heap size)`;
- размеры объектов, переведенных в следующее поколение (*Promoted size*) – из столбцов Surv0/1/2/3. Их также можно прочитать из события `Microsoft-Windows-DotNETRuntime/GC/HeapStats`;
- размеры поколений в конце GC (*Final size*) – из столбцов After0/1/2/3.

Таблица Per Generation GC Events содержит также данные о начале и завершении GC, выбранном поколении и фрагментации.

Ниже приводится подробное описание внутренней работы сборщика мусора в этом эксперименте. Заметим, что в данном сценарии мы используем также таблицу **All GC Events** (Все события GC) из отчета **GCStats** в **PerfView** (она уже была представлена в сценарии 7.1).

ДО GC В начале работы приложения – еще до создания первого объекта – установлены минимальные значения бюджетов выделения (см. табл. 7.3). Ниже приведены эти начальные значения (в байтах):

	Gen0	Gen1	Gen2	LOH
Бюджет выделения	4 194 304	294 912	262 144	3 145 728
Новое выделение	4 194 304	294 912	262 144	3 145 728
Начальный размер	24	24	24	24

Как уже было сказано, новому выделению для каждого поколения присваиваются такие же значения, отражающие доступное для выделения пространство. Физически каждое поколение в начале процесса пусто, его размер просто совпадает с минимальным размером объекта.

GC 1 – ЗАПУСКАЕТСЯ В РЕЗУЛЬТАТЕ ЯВНОГО ОБРАЩЕНИЯ К GC.Collect() Первая сборка мусора в нашей демонстрационной программе запускается явно (в строке 6 листинга 7.2). Ниже приведен соответствующий кусок таблицы **All GC Events** из отчета **GCStats** в **PerfView**:

GCIndex	Trigger Reason	Gen	Gen0 Alloc [MB]	Promoted [MB]	Gen0 Survival Rate [%]	Gen1 [MB]	Gen1 Survival Rate [%]	LOH [MB]	Gen0 Survival Rate [%]
1	Induced	2NI	0,213	0,082	33	0,192	0	0,018	99

Это подтверждает тот факт, что была запущена принудительная (ручная) неконкурентная полная сборка мусора (2NI). С начала работы программы было вы-

делено 0,213 МБ в SOH и 0,018 МБ в LOH. Это отражено в значениях начального размера и нового выделения в начале сборки мусора[1]:

- новое выделение для поколения 0 и LOH соответственно уменьшилось, а для поколений 1 и 2 осталось неизменным;
- начальный размер для поколения 0 и LOH увеличился, а для поколений 1 и 2 остался неизменным.

	Gen0	Gen1	Gen2	LOH
Новое выделение	3 995 024	294 912	262 144	3 128 216
Начальный размер	192 256	24	24	17 512

Суммарные размеры переведенных объектов для каждого поколения показаны в следующей таблице:

	Gen0	Gen1	Gen2	LOH
Размер переведенных в следующее поколение	64 088	0	0	17 440

Это означает, что в поколении 0 из 192 256 выделенных байтов 64 088 достижимы и будут переведены в поколение 1 (коэффициент выживания приблизительно равен 33 %, как видно из столбца Gen0 Survival Rate % в таблице All GC Events). Кроме того, большая часть объектов, выделенных в LOH, выжила (17 440 байт из общего количества 17 512, т. е. коэффициент выживания равен 99 %).

В этом месте вычисляются новые бюджеты выделения для поколения, в котором была запущена сборка мусора, и для всех более молодых (в случае полной сборки это вообще все поколения) – главным образом на основе коэффициента выживания. Поскольку эти коэффициенты равны нулю для поколений 1 и 2, их бюджеты остаются равными минимальной величине. Коэффициент выживания для поколения 0 высокий, это нормально для начальной стадии процесса – обычно GC пытается достичь коэффициента выживания порядка нескольких процентов в самом младшем поколении. В результате новые бюджеты выделения получаются такими:

	Gen0	Gen1	Gen2	LOH
Бюджет выделения	4 194 304	294 912	262 144	3 145 728

Значения нового выделения для каждого поколения также сделаны совпадающими с бюджетом выделения.

И конечные размеры поколений зависят от физически переведенных объектов.

---

[1] Заметим, что эти величины выражены в терминах изменения контекста выделения памяти, поскольку именно такие единицы измерения памяти сообщает GC. Кроме того, могут быть небольшие расхождения между значением нового выделения (которое читается в точке останова в Visual Studio) и значениями из различных событий ETW – это объясняется округлением в обоих источниках данных.

[2] Поколение 1 больше, чем ожидалось. Его размер стал равен 192 304 байтам, хотя из поколения 0 переведено только 64 088 байт. Это связано с высокой фрагментацией после этой сборки мусора. Об этом можно судить по большому значению столбца Gen1 Frag % (66,69 %) в таблице GC Events in Time отчета GCStats в PerfView. Очевидно, это указывает на то, что была произведена сборка очисткой, без уплотнения.

	Gen0	Gen1	Gen2	LOH
Конечный размер	24	192 304[1]	24	17 536

GC 2 – ЗАПУЩЕНА В РЕЗУЛЬТАТЕ ВЫДЕЛЕНИЯ ПАМЯТИ Вторая и все последующие сборки мусора произошли из-за создания массивов `byte[]` в цикле. Ниже приведен соответствующий кусок из таблицы All GC Events:

GCIndex	Trigger Reason	Gen	Gen0 Alloc [MB]	Promoted [MB]	Gen0 Survival Rate [%]	Gen1 [MB]	Gen1 Survival Rate [%]	LOH [MB]	Gen0 Survival Rate [%]
2	AllocSmall	2N	4,204	12,986	99	4,204	100	8,018	99

Мы видим, что с момента последней сборки мусора:

- в поколении 0 было выделено 4,204 МБ – из-за создания большого числа самих байтовых массивов;
- в куче больших объектов выделено около 8 МБ – из-за создания массива `byte[1_000_000][]` в строке 9, который содержит миллион 8-байтовых ссылок.

В результате таких выделений мы можем ожидать, что:

- после выделения 4,204 МБ бюджет поколения 0, очевидно, будет превышен (он был равен 4 194 304 байтам);
- выделение 8 МБ в LOH также превышает бюджет (он был равен 3 МБ).

Это можно подтвердить, взглянув на отрицательное новое выделение в поколении 0 и LOH в начале GC:

	Gen0	Gen1	Gen2	LOH
Новое выделение	–21 952	294 912	262 144	–4 854 328
Начальный размер	4 204 064	64 040	0	8 017 472

Бюджет LOH был превышен, так что эта сборка мусора превратилась в полную, хотя вначале можно было бы произвести сборку только в поколении 0 (отсюда значение 2N в столбце Gen приведенной выше таблицы событий).

Ниже показаны суммарные размеры переведенных объектов:

	Gen0	Gen1	Gen2	LOH
Размер переведенных в следующее поколение	4 204 032	64 088	0	8 017 464

Мы можем сделать следующие наблюдения:

- поколение 0 переведено целиком, поскольку все созданные массивы байтов достижимы (ссылки хранятся в массиве типа `byte[][]`);
- из поколения 1 переводятся данные, которые были переведены в него на предыдущем шаге.

Что касается бюджета выделения, то можно заметить следующие изменения:

	Gen0	Gen1	Gen2	LOH
Бюджет выделения	84 080 640	448 616	262 144	28 061 128

Эти значения бюджета можно объяснить следующим образом:

- коэффициент выживания в поколении 0 теперь очень близок к 100 %, поэтому его бюджет существенно увеличен;
- коэффициент выживания в поколении 1 также равен 100 % (см. столбец Gen1 Survival Rate % в таблице событий), поэтому его бюджет тоже увеличен;
- бюджет выделения в поколении 2 не изменился, потому что начальный размер данных в нем равен 0;
- бюджет выделения в LOH увеличился в 3,5 раза (этот коэффициент вычисляется функцией, похожей на ту, что изображена на рис. 7.13).

Наконец, конечные размеры поколений зависят от размера физически переведенных объектов.

	Gen0	Gen1	Gen2	LOH
Конечный размер	24	4 204 088	192 328	8 017 592

Самое время остановиться и оглядеться. После двух последовательных сборок мусора мы оказались в следующей ситуации:

- бюджет выделения в поколении 0 вырос приблизительно до 80 МБ из-за высокого коэффициента выживания – большое число объектов пережило сборку мусора в самом молодом поколении, так что дальше их может стать еще больше, поэтому имеет смысл увеличить бюджет. С новым бюджетом мы можем ожидать, что следующая сборка произойдет после выделения еще примерно 80 МБ памяти в SOH;
- бюджет выделения в поколении 0 меньше фактического размера поколения – это возможно, потому что GC еще не приспособился к высокому темпу выделения и перевода в следующее поколение. Впоследствии GC исправится, либо стабилизировав бюджет выделения (в случае если это был единичный всплеск в выделении памяти), либо постоянно увеличивая его (если потребление памяти устойчиво растет). Это наглядно демонстрирует логическую природу бюджетов выделения и его альтернативное название – желаемое выделение. Данная величина не является фактическим размером поколения;
- бюджет выделения в поколении 2 не изменился, но вместе с объектами в него была перенесена высокая фрагментация. Об этом свидетельствует большое значение (66,69 %) в столбце Gen2 Frag % таблицы GC Events in Time;
- бюджет выделения в LOH увеличился, приспосабливаясь к новому выделению памяти для больших объектов.

GC 3 – ЗАПУЩЕНА В РЕЗУЛЬТАТЕ ВЫДЕЛЕНИЯ ПАМЯТИ Третья сборка мусора произошла из-за дальнейшего создания массивов `byte[]`. Ниже приведен соответствующий кусок из таблицы All GC Events:

GCIndex	Trigger Reason	Gen	Gen0 Alloc [MB]	Promoted [MB]	Gen0 Survival Rate [%]	Gen1 [MB]	Gen1 Survival Rate [%]	LOH [MB]	Gen0 Survival Rate [%]
3	AllocSmall	0N	84,081	84,081	99	88,285	–	8,018	–

Мы видим, что с момента последней сборки мусора в поколении 0 было выделено примерно 84 МБ. Это должно исчерпать ее бюджет выделения. Сборка производится только в поколении 0 (значение 0N в столбце Gen), это типичная сборка мусора в одном лишь младшем поколении, инициированная выделением памяти в SOH.

Это можно подтвердить, взглянув на отрицательное новое выделение в поколении 0 в начале GC:

	Gen0	Gen1	Gen2	LOH
Новое выделение	–5 496	448 616	262 144	28 061 128
Начальный размер	84 080 640	–	–	–

Ниже показаны суммарные размеры переведенных объектов:

	Gen0	Gen1	Gen2	LOH
Размер переведенных в следующее поколение	84 080 640	–	–	–

Это приводит к интересной ситуации при вычислении новых бюджетов выделения:

	Gen0	Gen1	Gen2	LOH
Бюджет выделения	134 217 728	–83 632 024	262 144	28 061 128

Как видим, произошли следующие изменения:

- в полном соответствии с высоким коэффициентом выживания новый бюджет выделения в поколении 0 установлен равным максимальному размеру поколения (128 МБ);
- бюджет выделения в поколении 1 уменьшен на суммарный размер объектов, переведенных из поколения 0, – при этом бюджет оказался превышен, поэтому ожидается, что поколение 1 будет включено в следующую сборку мусора.

Напомним, что значения нового выделения для каждого поколения также динамически пересчитываются.

Наконец, конечные размеры поколений отражают интуитивно ожидаемые значения, соответствующие предыдущему размеру и размеру переведенных объектов, – изменились только размеры поколений 0 и 1:

	Gen0	Gen1	Gen2	LOH
Конечный размер	24	88 284 752	192 328	8 017 592

GC 4 – ЗАПУЩЕНА В РЕЗУЛЬТАТЕ ВЫДЕЛЕНИЯ ПАМЯТИ Третья сборка мусора также произошла из-за дальнейшего создания массивов `byte[]` и превышения бюджета выделения в поколении 0. Ниже приведен соответствующий кусок из таблицы All GC Events:

GCIndex	Trigger Reason	Gen	Gen0 Alloc [MB]	Promoted [MB]	Gen0 Survival Rate [%]	Gen1 [MB]	Gen1 Survival Rate [%]	LOH [MB]	Gen0 Survival Rate [%]
4	AllocSmall	1N	134,229	222,513	99	134,229	99	8,018	–

Как видим, действительно было выделено 134 229 МБ памяти, что должно превзойти ранее установленный бюджет выделения для поколения 0. Но, как мы помним, бюджет поколения 1 тоже должен быть превышен вследствие перевода объектов, оставшихся от предыдущей сборки мусора. Таким образом, сборка не ограничивается поколением 0, а включает также поколение 1 (в столбце Gen находится значение 1N).

Это можно подтвердить, взглянув на отрицательное новое выделение в поколениях 0 и 1 в начале GC (значение для поколения 1 было установлено по окончании предыдущей GC):

	Gen0	Gen1	Gen2	LOH
Новое выделение	–14 504	–83 632 024	262 144	28 061 128
Начальный размер	134 228 736	88 284 672	–	–

Ниже показаны суммарные размеры переведенных объектов:

	Gen0	Gen1	Gen2	LOH
Размер переведенных в следующее поколение	134 228 736	88 284 672	–	–

Поскольку сборка мусора производилась в обоих поколениях 0 и 1 и они содержат только достижимые массивы байтов, все объекты из них переводятся (высокий коэффициент выживания – 99 % – в поколениях 0 и 1).

А бюджеты выделения теперь стали такими:

	Gen0	Gen1	Gen2	LOH
Бюджет выделения	134 217 728	134 217 728	–88 022 528	28 061 128

С ними произошли следующие изменения:

- бюджет выделения в поколении 0 остался прежним – несмотря на высокий коэффициент выживания, его нельзя увеличить, т. к. он уже достиг максимального значения;
- бюджет выделения в поколении 1 увеличен до максимального значения – это реакция на высокий коэффициент выживания и большой суммарный размер переведенных объектов;
- бюджет выделения в поколении 2 уменьшен на суммарный размер объектов, переведенных из поколения 1, – это означает, что бюджет поколения 2 превышен, поэтому ожидается, что оно будет включено в следующую сборку мусора.

Наконец, размеры поколения отражают интуитивно ожидаемые значения, соответствующие предыдущему размеру и размеру переведенных объектов – размеры всех поколений в SOH изменились:

	Gen0	Gen1	Gen2	LOH
Конечный размер	24	134 228 760	88 477 048	8 017 592

GC 5 – ЗАПУЩЕНА В РЕЗУЛЬТАТЕ ВЫДЕЛЕНИЯ ПАМЯТИ Внимательный читатель, вероятно, ожидает обычной сборки мусора, инициированной выделением

памяти в SOH. Однако прежде возникает другое условие, запускающее GC. Ниже приведен кусок из таблицы All GC Events:

GCIndex	Trigger Reason	Gen	Gen0 Alloc [MB]	Promoted [MB]	Gen0 Survival Rate [%]	Gen1 [MB]	Gen1 Survival Rate [%]	LOH [MB]	Gen0 Survival Rate [%]
5	OutOfSpaceSOH	2*N*	134,179	364,174	99	134,179	99	8,018	–

Как видим, полная сборка (значение 2N в столбце Gen) началась по причине OutOfSpaceSOH. И это легко объяснить, глядя на внутренние данные о сборке мусора:

	Gen0	Gen1	Gen2	LOH
Новое выделение	35 592	134 217 728	–88 022 528	28 061 128
Начальный размер	134 178 688	134 228 736	88 348 760	8 017 440

Бюджет выделения превышен только для поколения 2 (из-за перевода объектов при прошлой сборке мусора), но не это причина запуска GC. Истинная причина заключается в том, что суммарный размер обоих эфемерных поколений (начальный размер) превышает максимальный размер эфемерного сегмента (256 МБ). В таком случае запускается сборка мусора по крайней мере в эфемерных поколениях. А поскольку бюджет поколения 2 превышен, эта сборка превращается в полную.

Ниже показаны суммарные размеры переведенных объектов:

	Gen0	Gen1	Gen2	LOH
Размер переведенных в следующее поколение	134 178 688	134 228 736	88 348 760	–

Из-за высокого коэффициента выживания бюджеты выделения в поколениях 0 и 1 сохраняют максимальные значения:

	Gen0	Gen1	Gen2	LOH
Бюджет выделения	134 217 728	134 217 728	178 062 152	28 061 128

Бюджет выделения в поколении 2 увеличился вдвое (этот коэффициент вычислен функцией, похожей на ту, что изображена на рис. 7.13) – в соответствии с высоким коэффициентом выживания.

И вот каковы новые размеры поколений:

	Gen0	Gen1	Gen2	LOH
Конечный размер	24	134 178 712	222 705 808	8 017 592

Это вполне ожидаемо. Поколение 0 пусто, промежуточное поколение 1 достигло максимума, а в поколении 2 собрались все остальные объекты, выделенные в SOH.

ПОСЛЕДУЮЩИЕ GC Поскольку наша программа потребляет память в постоянном темпе, следующие сборки мусора будут проходить по описанной выше схеме.

Сборка будет попеременно запускаться по одной из двух причин: AllocSmall (превышен бюджет поколения 0) и OutOfSpaceSOH (превышен максимальный размер эфемерного сегмента). Размер поколения 2 будет постепенно расти, а два других поколения будут оставаться на том же уровне.

Статические данные вкупе с регулярно обновляемыми динамическими данными управляют работой GC. Они контролируют момент запуска GC, выбор поколения и способ сборки: очисткой или с уплотнением. Полезно представлять себе, каковы эти данные и как они влияют на процесс.

Надеюсь, что подробное описание в сценарии 7.2 смогло проиллюстрировать связь между статическими и динамическими данными и их влияние на выделение памяти. Размеры поколений можно считать динамическими величинами, которые управляются бюджетами выделения соответствующих поколений, вычисляемыми, исходя из коэффициента выживания. В результате GC постоянно подстраивает размеры поколений в соответствии с текущими паттернами выделения и выживания, применяя статические данные, показанные в табл. 7.1 и 7.2 (от которых зависит характер важной функции на рис. 7.13).

Напомним, что всё это глубоко скрытые детали реализации. Нет никаких гарантий, что через несколько лет эти параметры будут оказывать такое же влияние на работу GC. Впрочем, на мой взгляд, маловероятно, что концепция бюджета выделения претерпит кардинальные изменения.

> В коде CoreCLR описанные выше динамические данные представлены классом `dynamic_data` в файле `.\src\gc\gcpriv.h`. Легко сопоставить каждому приведенному атрибуту поле этого класса. Самым важным является бюджет выделения, которому соответствует поле `desired_allocation`. В конце сборки оно вычисляется методом `gc_heap::desired_new_allocation` с применением различных эвристик (в основном опирающихся на коэффициент выживания, как на рис. 7.13, с линейной поправкой, которую вычисляет метод `gc_heap::linear_allocation_model` на основе заполненности поколения). Дальнейшее изучение этого поля можно начать с метода `gc_heap::compute_new_dynamic_data`, вызываемого в конце GC.

Инициаторы сборки мусора

Первый вопрос, который возникает в связи со сборкой мусора: когда она может запускаться? Что является инициатором? Но прежде чем дать конкретный ответ, полезно разобраться в проектных решениях, положенных в основу реализации GC, – они вполне доступно изложены в «Книге среды выполнения» (Book Of The Runtime):

- GC должна происходить достаточно часто, чтобы в управляемой куче не накапливалось много (в абсолютных или относительных цифрах) выделенных, но не используемых объектов (мусора), которые без надобности занимают память;
- GC должна происходить как можно реже, чтобы не занимать полезное для других целей время процессора, пусть даже частые сборки мусора снижают потребление памяти;
- GC должна быть продуктивной. Если GC освобождает мало памяти, то она была запущена зря (и на нее впустую было потрачено процессорное время);

- каждая GC должна быть быстрой. Во многих задачах малая величина задержки – непреложное требование;
- разработчики управляемого кода не должны знать все детали GC, чтобы обеспечить приемлемое использование памяти (в своей задаче), – сборщик мусора сам должен подстраиваться под различные схемы потребления памяти.

С учетом этих проектных решений ответ звучит так: «GC должна вызываться как можно реже, но давать оптимальные результаты». Разумеется, принимая во внимание бесчисленные возможные случаи и быстро изменяющиеся условия, спроектировать такой самонастраивающийся сборщик мусора – невероятно трудная задача. Очевидно, что для реализации поставленных целей идею вызова в цикле следует сразу отвергнуть. А ведь именно эта мысль первой приходит в голову неопытного разработчика для .NET – быть может, GC вызывается периодически, скажем, через какое-то количество миллисекунд? Краткий ответ – нет. Было бы непродуктивно просто вызвать GC и «посмотреть, что будет».

Есть много разных причин для запуска сборки мусора. Далее в этом разделе мы рассмотрим их, объединив в группы.

> Причины запуска GC в коде CoreCLR представлены перечислением `gc_reason`. С него и начните, если захотите исследовать эту тему самостоятельно.

Запуск по причине выделения памяти

В главе 6 мы видели, что распределители памяти в куче малых и больших объектов могут запускать сборку мусора, если не сумеют найти место для создаваемого объекта. В зависимости от условий может быть запущена сборка в одном или обоих эфемерных поколениях (когда собирается поколение 0 или 1) либо полная сборка.

Это самая частая причина запуска GC в приложении. Существует четыре причины такого рода (в скобках приведены имена, используемые в отчетах PerfView, мы это уже видели):

- выделение памяти для малого объекта (AllocSmall) – во время выделения памяти для объекта закончился бюджет поколения 0 (как указано в главе 6). Это самый частый случай;
- выделение памяти для большого объекта (AllocLarge) – во время выделения памяти для большого объекта закончился бюджет LOH;
- выделение памяти для малого объекта на медленном пути (OutOfSpaceSOH) – у распределителя кончилось место на «медленном пути» выделения памяти в SOH. Даже реорганизация сегмента и, быть может, даже предшествующий запуск GC не помогли – свободного места нужного размера нет. В 64-разрядной среде выполнения с большой виртуальной памятью такая причина должна встречаться редко. Но даже в этом случае такое бывает в случае GC в режиме рабочей станции, как показано в сценарии 7.2;
- выделение памяти для большого объекта на медленном пути (OutOfSpaceLOH) – у распределителя закончилось место на «медленном пути» выделения памяти в LOH. Как и OutOfSpaceSOH, эта причина не должна встречаться часто.

Разумеется, хорошее управление памятью обычно означает создание как можно меньшего числа объектов. Поэтому запуск по причине выделения памяти – лучшее место для оптимизации GC: если нет выделения, то такой инициатор никогда не срабатывает. Нет выделения – нет GC!

Явный запуск

Иногда возникает желание запросить явный запуск GC. Такую сборку мусора часто называют *принудительной*. Открытый API позволяет сделать это несколькими способами. Самый распространенный – вызов метода `GC.Collect`. У него есть несколько перегруженных вариантов, отличающихся уровнем контроля:

- `GC.Collect()` – запросить полную сборку мусора, но без принудительного уплотнения;
- `GC.Collect(int generation)` – запустить GC в указанном поколении, блокирующую, но без принудительного уплотнения;
- `GC.Collect(int generation, System.GCCollectionMode mode)` – запустить блокирующую GC в указанном поколении в указанном режиме: `Forced` (немедленно и принудительно) или `Optimized` (решение о том, в какой момент запускать сборку, остается за сборщиком мусора);
- `GC.Collect(int generation, System.GCCollectionMode mode, bool blocking)` – запустить GC в указанном поколении в указанном режиме, явно указав, должна она быть блокирующей или нет, и задав режим: `Forced` или `Optimized` (решение о выборе момента остается за GC);
- `GC.Collect(int generation, System.GCCollectionMode mode, bool blocking, bool compacting)` – запросить сборку, задав все параметры явно.

Как мы объясним позже, GC включает шаг проверки ряда условий, чтобы определить, какая сборка мусора будет самой продуктивной. Таким образом, даже если мы явно укажем номер поколения при вызове `GC.Collect`, в этом поколении (и во всех младших), конечно, будет произведена сборка, но выбрано может быть и более старое поколение, например если старшее поколение превысило свой бюджет.

> Вызов `GC.Collect(2, GCCollectionMode.Forced, blocking: false, compacting: true)` может показаться странным. В главе 11 мы узнаем, что неблокирующая (конкурентная) полная сборка производится без уплотнения, поэтому такие аргументы вроде бы противоречивы. В таком случае запущенная сборка будет неблокирующей и неуплотняющей или блокирующей и уплотняющей (решение остается за сборщиком мусора).

Вызов `GC.Collect` редко бывает оправдан. Вся эта книга – о том, что сборщик мусора в .NET – сложная и хорошо оптимизированная система. В ней ведется различная статистика для эвристических решений о том, нужно ли производить сборку мусора, и если да, то сборка в каком поколении будет наиболее продуктивна. Явные вызовы `GC.Collect` идут вразрез с такими эвристиками.

Да и вообще трудно подыскать обоснование этому методу. В главах 8 и 10 мы увидим, что CLR прилагает максимум усилий, чтобы убрать ненужные объекты как можно быстрее. Для определения того, какие объекты подлежат сборке, применяется пометка. Если объект не убран в мусор, значит, кто-то хранит на него ссылку. И вызов `GC.Collect` тут не поможет. Не поможет это и в ситуации типа «а вдруг CLR забыла вызвать сборщик мусора, так я ей напомню». GC не вызывается, если

он окажется непродуктивным. Стало быть, и явный вызов `GC.Collect` будет непродуктивным. Когда вы в следующий раз увидите в чужом коде вызов `GC.Collect`, это может означать одно из двух: либо автор кода не в курсе всех описанных выше тонкостей, либо он из тех знающих людей, кто осознанно применяет этот метод в тех крайне немногочисленных ситуациях, когда он действительно что-то дает.

Посмотрим, в каких ситуациях мы могли бы захотеть произвести сборку мусора в каждом поколении по отдельности.

- Поколение 0 – вы полагаете, что в самом молодом поколении скопилось много мертвых объектов, и хотите убрать их принудительно. Но если вы создаете какие-то объекты в своем приложении, то сборка мусора в этом поколении все равно производится очень часто. Согласно настройкам CLR (см. табл. 7.1) поколение 0 не может вырасти слишком сильно. Так что лучше позволить GC делать свою работу. Благодаря самонастройке, основанной на коэффициенте выживания, он оптимизирует частоту сборки в поколении 0. Явный вызов может только испортить подстроенные сборщиком параметры.
- Поколение 1 – это промежуточное поколение. Трудно сказать, когда и какие объекты в него попадают и как долго там остаются, потому что все это сильно зависит от динамических условий, складывающихся в приложении. Поколение 1 существует для того, чтобы молодые объекты не переводились сразу в поколение 2, где будут обретаться довольно долго. Оно дает шанс убрать объекты до попадания в поколение 2. Темп выделения и коэффициент выживания, отслеживаемые GC, помогают в этом. Явно запуская сборку в поколении 1, вы пускаете все это насмарку. Все еще достижимые объекты переводятся в поколение 2, некоторые – преждевременно и без особой необходимости. А предотвращение перевода в старшее поколение – одна из тех вещей, которые должны стоять на первом месте. Явный запуск сборки в эфемерном поколении все же может показаться соблазнительным, потому что это быстро и является последней попыткой избежать полной сборки мусора. Однако я предлагаю вам вместо этого переосмыслить свои структуры данных и постараться укоротить их жизнь.
- Поколение 2 – полная сборка мусора гораздо дороже прочих, но со своей работой она справляется – все, что может быть убрано, будет убрано. Поводом к явному запуску может быть то, что поколение 2 «велико» или «постоянно растет». Но чаще всего это случается из-за наличия корней, о которых вы не знаете, а не потому, что GC пренебрегает своими обязанностями. На самом деле полные сборки, скорее всего, и так уже производятся из-за повышенного потребления памяти. Добавление еще и явных вызовов только приведет к дополнительным накладным расходам, но не даст положительного эффекта. Вместо того чтобы запускать GC, перепроектируйте свое приложение, чтобы оно не порождало долгоживущих объектов, заканчивающих жизнь в старшем поколении. Хранение слишком объемной информации о состоянии, слишком долгое нахождение объектов в кеше или чтение чрезмерно большого количества данных из базы – вот на что стоит обратить внимание, начиная оптимизацию.

И не забудьте, что какой бы номер поколения ни был указан в аргументе `GC.Collect`, дело все равно может закончиться полной сборкой мусора!

Но каковы же все-таки те крайне немногочисленные ситуации, в которых вызов `GC.Collect` оправдан? Их можно отнести к нескольким категориям.

- Вам известна природа некоторого редко возникающего в приложении поведения, которое GC вряд ли понимает (но вы-то знаете жизненный цикл своего приложения). Например, это может быть эпизодическая пакетная обработка, вызвавшая создание большого количества объектов, в итоге оказавшихся в поколении 2. Если такие всплески выделения памяти нечасты, то GC может еще долго не приступить к сборке мусора в поколении 2. Поэтому весь мусор, накопившийся во время пакетной обработки, там и останется, и общее потребление памяти возрастет. Это не катастрофа (накладные расходы на сборку мусора не увеличиваются), но процесс долгое время остается «разбухшим», хотя вы знаете, что этого можно было бы избежать. Другой пример – очистка памяти перед ожидаемым всплеском потребления, например вышеупомянутой эпизодической пакетной обработкой, загрузкой нового уровня игры и т. д. Также это может быть полезно, перед тем как перевести приложение в режим работы с низкой задержкой, в котором накладные расходы на GC (и на среду выполнения вообще) должны быть минимальны.

  Все эти примеры прямо противоположны, к примеру, стабильно работающему веб-приложению, обрабатывающему более-менее постоянное число запросов. В этом случае GC располагает хорошей оценкой характеристик выделения и выживания, которая позволяет ему принимать решения лучше, чем смогли бы мы сами.
- Превентивная очистка в сознательно выбранных точках программы. Как и в первом случае, мы можем воспользоваться знанием специфики приложения, чтобы заранее собрать мусор в моменты, когда это будет незаметно пользователю. Типичный пример – сборка мусора в ожидании ввода данных пользователем или пока отображается какая-то заставка. Но эта причина не так убедительна, как первая. Нужна твердая убежденность в осмысленности подобных вызовов. Продуктивны ли они или делаются просто «на всякий случай»? Помните, что они пускают на ветер усилия GC по самонастройке.
- Очистка для тестирования производительности. Любое измерение требует тщательно подготовленной среды. Чтобы убедиться в повторяемости накладных расходов на GC, мы должны подготовить тестовую среду, которая будет находиться в одном и том же состоянии перед каждым тестом. Для этого, в частности, нужно очистить память от всего, что можно очистить. И запускать полную сборку мусора в этом случае – обычная практика.
- В специальных модульных и интеграционных тестах – например, в тех, где используется класс `WeakReference` (пример приведен в главе 12) или сторонний код, который подозревается в утечке памяти. Явно вызывая `GC.Collect` до и после прогона теста (чтобы очистить все, что можно), мы обеспечиваем повторяемость результатов тестирования.
- В качестве решения проблемы плохих характеристик работы с памятью в сторонней библиотеке – здесь есть сходство с первыми двумя причинами. Мы не в состоянии контролировать такой код, а единственное, что можем сделать (если не считать перехода на другую библиотеку), – убрать мусор до и (или) после ее использования.

Так или иначе, вызывать `GC.Collect` следует лишь эпизодически. Вызов в цикле для преодоления проблем означает попытку замести мусор под ковер. Как правило, настоящая проблема заключается в кризисе среднего возраста или постоянном присутствии каких-то корней, а эти проблемы явным запуском GC не решить.

Есть еще один способ почти явно запустить GC – метод `GC.AddMemoryPressure` (описан в главе 15). Он сообщает GC, что некоторые управляемые объекты удерживают определенный объем неуправляемой памяти. Поскольку, с точки зрения сборщика мусора, такие неуправляемые данные не отслеживаются кучей GC, он не может учесть их размер в решениях, касающихся использования памяти. Если общий размер неуправляемой памяти, указанный в вызовах `GC.AddMemoryPressure`, превышает динамически настраиваемый порог, то запускается неблокирующая сборка мусора в поколении, которое определяется с помощью внутренних эвристик.

> В текущей реализации начальное значение порога равно 100 000 байт (и никогда не опускается ниже). Затем оно динамически изменяется, исходя из размеров, переданных методу `GC.AddMemoryPressure` (увеличивается в 8 раз или на 10 % в зависимости от того, какой результат больше) и методу `GC.RemoveMemoryPressure`. Также учитывается, сколько раз производилась сборка в каждом поколении. Хотя эти детали реализации, скорее всего, будут меняться, стоит отметить, что логика работы с неуправляемой памятью основана на внутренних эвристиках, которые никак не связаны с эвристиками в основной логике GC.

Сценарий 7.3. Анализ явных вызовов GC

Описание. Мы разрабатываем приложение с использованием WPF. С учетом высказанных выше замечаний мы хотим проверить, запускает ли оно сборку мусора явно. Разумеется, если доступен исходный код, то проще всего поискать обращения к `GC.Collect`. Но, во-первых, наше приложение состоит из разных компонентов, и не для всех исходный код имеется. А во-вторых, само присутствие обращения к `GC.Collect` мало что говорит о том, как в действительности этот метод используется – вызывается ли вообще, и если да, то как часто. В качестве примера мы взглянем на работу приложения dnSpy – свободного и открытого отладчика .NET и редактора сборок, который уже рассматривался в предыдущих главах.

Анализ. Начнем анализ программы с проверки того, запускает ли она вообще сборку мусора явно. Самый простой и быстрый способ – воспользоваться счетчиком производительности `Память CLR .NET (dnSpy)\Принудительных GC (.\NET CLR Memory(dnSpy)\# Induced GC)`, который подсчитывает все вызовы GC такого типа (рис. 7.14). Как видим, принудительные GC встречаются (шесть раз за период наблюдения, равный одной минуте). Наблюдая за графиком на протяжении всего теста, мы замечаем, что счетчик увеличивается при каждом открытии новой сборки на панели **Assembly Explorer** (Обозреватель сборок).

Убедившись, что вызовы действительно есть, проанализируем, где они встречаются. С этой целью снова воспользуемся программой PerfView и соберем данные вместе со стеками вызовов. Для этого нужно ввести параметр `Microsoft-Windows-DotNETRuntime:GCKeyword:Informational:@StacksEnabled=true` в поле **Additional Providers** в диалоговом окне **Collect**.

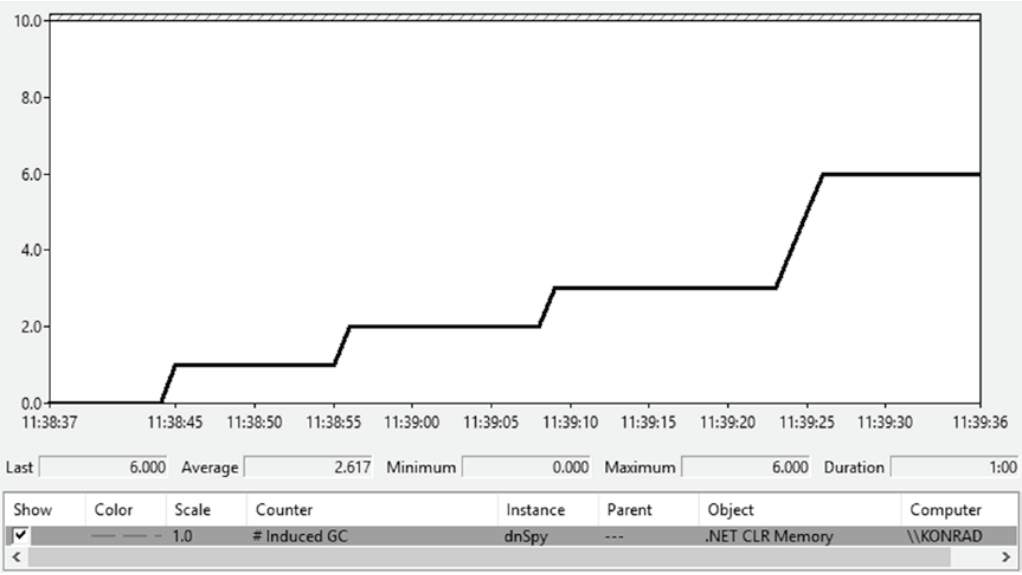


Рис. 7.14 ❖ Счетчик производительности Память CLR .NET (dnSpy)\Принудительных GC (.\NET CLR Memory(dnSpy)\# Induced GC) за первую минуту работы приложения dnSpy

Записав сеанс, откройте отчет **GCStats** из группы **Memory**. В таблице GC Rollup By Generation для процесса dnSpy мы найдем подтверждение того, что принудительные вызовы GC имели место (см. столбец Induced на рис. 7.15).

GC Rollup By Generation

All times are in msec.

Gen	Count	Max Pause	Max Peak MB	Max Alloc MB/sec	Total Pause	Total Alloc MB	Alloc MB/ MSec GC	Survived MB/ MSec GC	Mean Pause	Induced
ALL	8	27.7	216.9	654.519	95.6	304.8	3.2	0.496	11.9	8
0	4	12.9	178.0	654.519	30.2	151.5	0.1	0.173	7.5	4
1	2	27.7	216.9	19.184	35.6	89.0	0.1	0.200	17.8	2
2	2	26.5	150.3	6.746	29.8	64.3	0.1	0.995	14.9	2

Рис. 7.15 ❖ Таблица GC Rollup By Generation из отчета GCStats для процесса dnSpy

Теперь откройте панель **Events** записанного сеанса и найдите события `Microsoft-Windows-DotNETRuntime/GC/Triggered`, которые генерируются при явном запуске GC. Поскольку параметр `StacksEnabled` был включен, мы имеем стек вызова для каждого события (рис. 7.16).

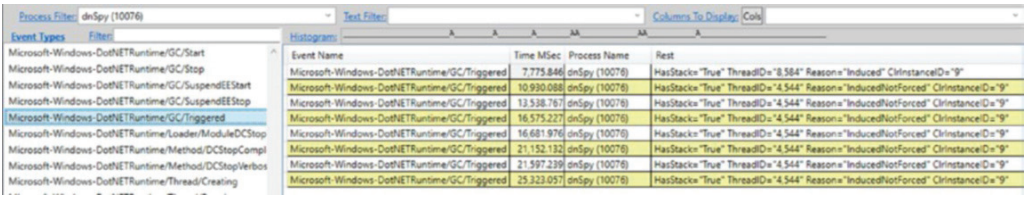


Рис. 7.16 ❖ События, связанные с процессом dnSpy

В поле **Reason** могут встречаться три значения:

- Induced – явная GC без указания предпочтительных режимов блокирования и уплотнения;
- InducedNotForced – явная GC, которая необязательно должна быть блокирующей;
- InducedCompacting – явная GC с уплотнением (но только для SOH, напомним, что уплотнение LOH явно задается другим параметром).

Выбрав команду **Open Any Stacks** из контекстного меню столбца **Time MSec**, мы сможем увидеть точный стек вызова в момент каждого явного запуска GC.

> Может показаться, что в данном случае лучше начинать анализ с события `Microsoft-Windows-DotNETRuntime/GC/Start`. Однако оно генерируется в месте, где фактически начинает работать сборщик мусора. У нас же большинство GC обрабатываются отдельным потоком в фоновом режиме. Стек вызова, соответствующий такому событию, просто укажет то место в выделенном потоке GC, где был получен сигнал начать работу.

Анализ стеков вызова позволяет определить два основных источника явного запуска GC (исходный код dnSpy можно скачать по адресу https://github.com/0xd4d/dnSpy).

1. Очистка памяти после декомпиляции сборки (рис. 7.17). После этого во временном кеше могут храниться уже ненужные данные. Чтобы убрать их как можно быстрее, производится явное обращение к `GC.Collect`.

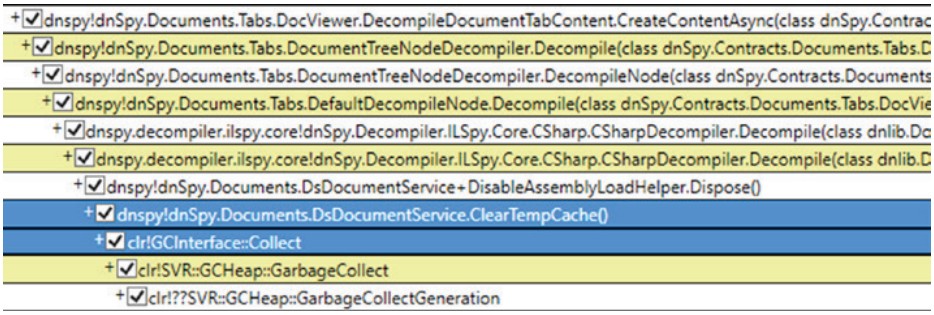


Рис. 7.17 ❖ Стек вызовов для явного вызова GC.Collect первого вида

В листинге 7.3 приведен соответствующий фрагмент кода. Это пример обертывания объекта, интенсивно использующего ресурсы (в нашем случае экземпляра `DsDocumentService`), вспомогательным объектом, который реализует интерфейс `IDisposable`. Он применяет очень простую технику подсчета ссылок для наблюдения за использованием обернутого объекта. Если он больше не используется, то производится явное освобождение тяжеловесных ресурсов.

Листинг 7.3 ❖ Пример явного запуска GC в коде dnSpy

```
sealed class DsDocumentService : IDsDocumentService {
   int counter_DisableAssemblyLoad;
   // ...
   public IDisposable DisableAssemblyLoad() => new DisableAssemblyLoadHelper(this);
   sealed class DisableAssemblyLoadHelper : IDisposable
   {
      readonly DsDocumentService documentService;
      public DisableAssemblyLoadHelper(DsDocumentService documentService) {
         this.documentService = documentService;
         Interlocked.Increment(ref documentService.counter_DisableAssemblyLoad);
      }
      public void Dispose() {
         int value = Interlocked.Decrement(ref documentService.counter_DisableAssemblyLoad);
         if (value == 0)
            documentService.ClearTempCache();
      }
   }
   // ...
   void ClearTempCache() {
      bool collect;
      lock (tempCache) {
         collect = tempCache.Count > 0;
         tempCache.Clear();
      }
      if (collect) {
         GC.Collect();
         GC.WaitForPendingFinalizers();
      }
   }
   // ...
}
```

В листинге 7.4 приведен пример использования такого класса.

Листинг 7.4 ❖ Пример использования класса из листинга 7.3

```
using (context.DisableAssemblyLoad()) {
   // Внутри этого блока счетчик ссылок во вспомогательном объекте увеличивается.
   // context содержит ссылку на экземпляр DsDocumentService
```

Код в листинге 7.3 – лишь один из примеров реализации защитной очистки памяти. Вместо подсчета ссылок можно было бы просто вызвать метод `GC.Collect` в какой-то точно определенный момент времени, когда приложение замечает, что сборка была декомпилирована (например, получено событие от интерфейса пользователя). Также возникает искушение непосредственно реализовать интер-

фейс IDisposable в классе DsDocumentService и вызывать GC.Collect из метода Dispose. Но это означало бы изменение семантики использования DsDocumentService, что не всегда приемлемо. Еще одно возможное решение – вызывать GC.Collect из финализатора DsDocumentService.

Представленная только что ручная очистка памяти – пример первого из списка перечисленных выше случаев. Разработчик решил явно запустить GC, потому что знает, что это эпизодическое, инициированное пользователем действие, для выполнения которого требуется много временных данных.

2. Контроль над неуправляемой памятью, связанной с растровыми изображениями (рис. 7.18). Как видим, на этот раз GC был запущен из Windows Presentation Foundation (файл PresentationCore.dll – часть каркаса WPF) вследствие загрузки изображения.

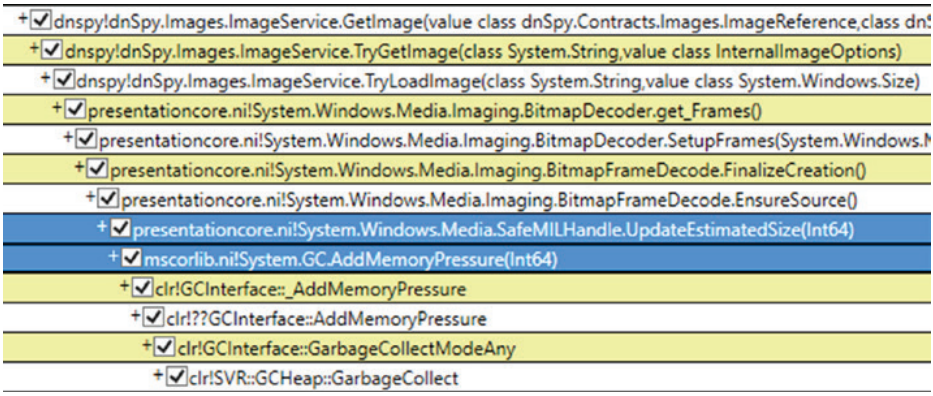


Рис. 7.18 ❖ Стек вызова для явного вызова GC.Collect второго вида

Как выясняется, это известная проблема. Растровые изображения – в WPF они представлены классом BitmapSource – это небольшие управляемые объекты, в которых данные изображения хранятся в неуправляемой памяти. Поэтому сборщику мусора они кажутся малыми, т. к. неуправляемые данные не учитываются в размере объекта. Проблему можно было бы решить, если реализовать интерфейс IDisposable и вызывать методы GC.AddMemoryPressure и GC.RemoveMemoryPressure соответственно в конструкторе и методе Dispose. К сожалению, проектировщики выбрали другое решение.

Данные растрового изображения удерживаются дополнительным описателем с подсчетом ссылок, который отвечает за вызовы GC.AddMemoryPressure и GC.RemoveMemoryPressure (листинг 7.5). Как было сказано выше, метод AddMemoryPressure может запускать GC, если превышены некоторые пороговые значения; именно это здесь и происходит.

Листинг 7.5 ❖ Класс SafeMILHandleMemoryPressure из библиотеки PresentationCore.dll (Windows Presentation Foundation)

```
namespace System.Windows.Media
{
   internal class SafeMILHandleMemoryPressure
```

```
   {
      [SecurityCritical]
      internal SafeMILHandleMemoryPressure(long gcPressure)
      {
         this._gcPressure = gcPressure;
         this._refCount = 0;
         GC.AddMemoryPressure(this._gcPressure);
      }

      internal void AddRef()
      {
         Interlocked.Increment(ref this._refCount);
      }

      [SecurityCritical]
      internal void Release()
      {
         if (Interlocked.Decrement(ref this._refCount) == 0)
         {
            GC.RemoveMemoryPressure(this._gcPressure);
            this._gcPressure = 0L;
         }
      }
      private long _gcPressure;
      private int _refCount;
   }
}
```

В этом примере применен такой же подход, как в примере с декомпиляцией: обертка с подсчетом ссылок. Но на этот раз обертка не запускает сборщик мусора явно, а только информирует его о наличии дополнительной неуправляемой памяти. Решение о запуске сборки мусора GC принимает самостоятельно.

Без этой уловки сборка мусора происходила бы гораздо реже, чем нужно, и приложение в течение длительного времени занимало бы много памяти без всякой необходимости. Проблема осложняется, когда загружается много изображений. Скорее всего, вы увидите такие принудительные вызовы GC в своих собственных WPF-приложениях. Если сопутствующие накладные расходы (например, значение в столбце % Time) невелики, то все нормально. Но если они значительны, то изменить код WPF мы, конечно, не можем. На уровне приложения можно применить обходной маневр – создать пул объектов типа `WriteableBitmap` и использовать их повторно.

> Исторически класс `SafeMILHandleMemoryPressure` хранил собственный набор счетчиков для контроля над использованием памяти и явно вызывал `GC.Collect`, чтобы произвести полную сборку, когда счетчики превышали пороговую величину. Но у такого решения было больше недостатков, чем достоинств. Начиная с .NET Framework 4.6.2 эта логика была перемещена в GC благодаря методам `AddMemoryPressure/RemoveMemoryPressure`.

> **Примечание 2.** Если CLR размещена самостоятельно, то открывается еще одна возможность запустить GC явно – метод `ICLRGCManager::Collect`. При этом производится блокирующая полная сборка в указанном поколении.

Запуск по причине нехватки памяти у системы

Сборка мусора может быть запущена «извне». Если операционная система замечает, что память скоро закончится, она может разослать всем процессам сигнал «уведомление о недостатке памяти». Корректно написанные приложения могут (но не обязаны) прослушивать такое уведомление и попытаться помочь системе, отреагировав на него. Они могут уменьшить свой рабочий набор так, как считают нужным. Но, конечно, могут и проигнорировать, если полагают, что это правильно.

Среда выполнения .NET прослушивает этот сигнал. Получив его, она запускает сборку мусора в эфемерных поколениях (которая может быть преобразована в полную сборку, если потребление памяти велико). Кроме того, во время такой сборки GC ведет себя более агрессивно. Например, полная сборка будет произведена с большей вероятностью. Выгоды обоюдны, потому что сокращение потребления памяти благотворно сказывается на всех работающих в системе приложениях (включая и написанные для .NET).

> Механизм уведомления о недостатке памяти в настоящее время поддерживает только Windows. Для этого используется функция WinApi `CreateMemoryResourceNotification`. На это уведомление реагирует поток финализации (мы познакомимся с ним в главе 11), который был выбран, потому что он гарантированно трудится на всем протяжении работы приложения. Получив уведомление, этот поток запускает GC. Согласно комментарию во внутреннем классе `System.Runtime.Caching.PhysicalMemoryMonitor`, который, в свою очередь, основан на комментариях в коде самой Windows, уведомление о недостатке памяти посылается, когда занято от 97 до 99 % физической памяти (в зависимости от объема ОЗУ в системе).

Чтобы проверить, запускает ли уведомление о недостатке памяти сборку мусора в нашем приложении, проще всего записать сеанс ETW и поискать в отчетах событие `Microsoft-Windows-DotNETRuntime/GC/Start` для GC со следующими причинами:

- LowMemory – операционная система отправила уведомление о недостатке памяти;
- InducedLowMemory – операционная система отправила уведомление о недостатке памяти (и среда выполнения запросила блокирующий GC);
- LowMemoryHost – хост-процесс отправил уведомление о недостатке памяти (в настоящее время не используется).

Запуск по различным внутренним причинам

В среде выполнения и в стандартных библиотеках есть немало мест, в которых запрашивается сборка мусора. Такие сборки чаще всего помечаются как принудительные (как и при явном вызове), поскольку, с точки зрения сборщика мусора, не важно, кто вызвал: пользователь, среда выполнения или управляемая библиотека.

Перечислим наиболее распространенные причины этого типа.

- Выгрузка домена приложения – очистка объектов, связанных с доменом приложения, – достаточная причина для сборки мусора. В этом случае запускается блокирующая полная сборка.
- Очистка объектов `Thread`, соответствующих мертвым потокам, – в долго работающем приложении могут создаваться и удаляться различные потоки.

Каждый такой поток представлен управляемым объектом. В этом случае запускается неблокирующая сборка в поколении, где находится больше всего мертвых объектов `Thread`, но не чаще чем раз в 30 минут (период по умолчанию).

- Перед входом в область NoGC (см. главу 15) – участок кода, в котором запрещен запуск сборки мусора, может заметно увеличить потребление памяти, поэтому разумно будет все почистить заранее. В этом случае запускается блокирующая полная сборка мусора с целью освободить всю память, занятую мертвыми объектами.

Существует также внутренний механизм, используемый разработчиками .NET и известный под названием стресс-теста GC. Он включает частый запуск GC для диагностических целей, в основном для обнаружения так называемых *дыр GC*, например вещей, о которых необходимо было известить GC, но этого не произошло.

Большинству перечисленных выше внутренних триггеров в данных ETW будет соответствовать причина Induced. Кроме нее, существуют еще такие причины:

- Internal – внутренняя причина, соответствующая стресс-тесту со стороны среды выполнения;
- Empty и PMFullGC – в настоящее время не используются.

Приостановка движка выполнения

В работе сборщика мусора существуют такие промежутки времени, когда потоки, исполняющие код приложения, не должны работать, т. к. могли бы читать и модифицировать те области памяти, к которым обращается сам GC. В зависимости от того, в каком режиме работает GC, эти промежутки могут быть короче или длиннее. В неконкурентном режиме пользовательские потоки приостановлены на все время работы GC. Но и в конкурентном режиме (описанном в главе 11) лишь некоторые части GC могут работать одновременно с другими потоками, так что даже в этом случае возникает необходимость приостанавливать пользовательские потоки.

Процесс приостановки всех потоков, исполняющих пользовательский код, называется *приостановкой движка выполнения* (EE suspension), имеется в виду *приостановка управляемых потоков*. В неконкурентном режиме, описанном ниже, GC в самом начале работы просит службу приостановки приостановить все управляемые потоки, а по завершении – возобновить их. Такой подход, характеризуемый очевидным вмешательством, часто называют «остановка мира» (stop the world), потому что, с точки зрения приложения, весь мир приостанавливается на время сборки мусора.

Как сказано в «Книге среды выполнения» (Book Of The Runtime): «CLR должна гарантировать, что все управляемые потоки остановлены (чтобы они не модифицировали кучу), только тогда она может безопасно и надежно найти все управляемые объекты. Остановка производится лишь в безопасных точках, когда можно проверить наличие живых ссылок в регистрах и стеке».

Таким образом, *безопасной точкой* называется такое место в коде, где можно проверить наличие живых ссылок в регистрах и стеке. Реализация безопасных точек – занятие нетривиальное. Очевидно, что приостановка должна быть еще

и очень эффективной, потому что время приостановки и возобновления потоков учитывается в общем времени паузы на сборку мусора. Впрочем, с точки зрения управления памятью в .NET и, стало быть, этой книги в целом, эти детали реализации не особенно важны. Приостановка потока – вообще не часть GC. Однако полезно познакомиться с применяемыми терминами, которые могут встречаться в различных инструментах для анализа потребления памяти (особенно в WinDbg). К тому же, как мы скоро увидим, логика приостановки тесно связана с жизнеспособностью локальных данных.

С точки зрения GC, каждый управляемый поток может работать в одном из двух режимов:

- *кооперативный* – в комментариях к исходному коду CoreCLR написано: «кооперативный режим по существу означает, что поток потенциально может модифицировать ссылки, с которыми работает GC, поэтому среда выполнения должна кооперироваться с ним, чтобы достичь 'безопасной относительно GC' точки, в которой можно перечислить эти ссылки». В таком режиме потоки, исполняющие управляемый код, работают большинство времени;
- *вытесняющий* – в этом режиме службе приостановки не о чем беспокоиться, поскольку гарантируется, что поток находится в точке, где можно начинать GC, потому что исполняет код, который не читает и не изменяет ссылки, с которыми работает GC. Как правило, это просто означает, что поток знает, как приостановить себя.

Вместе с тем приостановку движка выполнения можно определить как принудительный переход к ситуации, когда все управляемые потоки находятся в вытесняющем режиме. Переход от кооперативного режима к вытесняющему может происходить только в безопасных точках. В каждой безопасной точке запоминается представление состояния потока – стек и регистры, поскольку они могут содержать ссылки на объекты (и тогда становятся корнями графа объектов). Такие данные называются *информацией для GC (GC info)*. Если считать безопасными точками все инструкции приложения (тогда можно будет вытеснить поток при выполнении любой команды), то понадобилось бы хранить информацию для GC в каждой из них. Это потребовало бы громадного объема памяти.

Поэтому, как часто бывает в подобных случаях, было принято компромиссное решение. Управляемый код можно JIT-компилировать в код одного из двух видов:

- *частично прерываемый* – безопасными точками считаются только вызовы других методов (включая явные опросы для проверки того, есть ли ожидающая сборка мусора[1]). Количество команд между вызовами методов в среднем методе .NET невелико. Поэтому при таком подходе обеспечивается достаточно высокая плотность безопасных точек и вместе с тем разумные накладные расходы на хранение информации для GC. Генерация частично прерываемого кода – предпочтительный выбор JIT-компилятора;
- *полностью прерываемый* – безопасной точкой считается каждая команда метода (весь код вытесняемый), за исключением пролога и эпилога (небольших фрагментов в начале и в конце метода соответственно). JIT-ком-

[1] Такие опросы GC встречаются в разных местах самой среды выполнения и в некоторых случаях генерируются JIT-компилятором. Но они редки, потому что опрос – не особенно эффективная техника, и обычно достаточно просто подождать первого вызова метода, который также является безопасной точкой.

пилятор должен каким-то образом сохранить информацию для GC для каждой команды, зато полностью прерываемый код можно очень быстро приостановить. Из-за высоких накладных расходов JIT-компилятор старается избегать такого подхода. Один из случаев, когда он все-таки прибегает к нему, – циклы с неизвестным количеством итераций без единого вызова метода внутри (не гарантируется, что они быстро завершатся, что может привести к откладыванию GC надолго). Одно из типичных решений этой проблемы – вставка опроса GC перед переходом на начало цикла. Но эффективность таких избыточных опросов невысока.

Если вас интересует дополнительная информация, поищите в коде CoreCLR макросы `FC_GC_POLL` и `FC_GC_POLL_RET`, реализующие вышеупомянутый опрос GC.

В «Книге среды выполнения» (Book Of The Runtime) написано: «JIT-компилятор эвристически выбирает режим генерации полностью или частично прерываемого кода, стремясь найти наилучший компромисс между качеством кода, размером информации для GC и задержкой перед приостановкой для сборки мусора».

В процессе приостановки движка выполнения среда пытается скоординировать все потоки, работающие в кооперативном режиме, заставив их перейти в вытесняющий режим в безопасных точках[1]. Прежде всего вызывается функция операционной системы, которая приостанавливает платформенный поток, стоящий за управляемым (в случае Windows API это функция `SuspendThread`), а затем:

- в случае полностью прерываемого кода все легко. Поток уже находится в безопасной точке, так что его можно просто оставить приостановленным;
- в случае частично прерываемого кода нам может повезти, тогда поток будет приостановлен в безопасной точке, и мы оставляем его приостановленным, как и выше. Если же поток был приостановлен не в безопасной точке (что более вероятно), то адрес возврата в текущем кадре стека подменяется адресом специальной заглушки, которая «запаркует» его в безопасной точке, после чего поток ненадолго возобновляется (возможно, в это время он и сам наткнется на свою безопасную точку).

Возобновление потоков гораздо проще приостановки. По завершении работы GC все приостановленные потоки пробуждаются отправкой события о конце приостановки и продолжают выполнение как ни в чем не бывало.

Наблюдать за приостановкой и возобновлением потоков в интересах GC позволяют пары событий ETW `GCSuspendEE_V1/GCSuspendEEEnd_V1` и `GCRestartEEBegin_V1/GCRestartEEEnd_V1` соответственно.

Сценарий 7.4. Анализ времени приостановки GC

Описание. В процессе разработки приложения для .NET мы хотели бы из чистого любопытства узнать, сколько времени в действительности занимает приостановка для сборки мусора. Никаких проблем мы не предвидим, просто интересно.

---

[1] Это описание для краткости упрощено. Если вас интересуют глубокие детали реализации, обратитесь к разделу «CLR Threading Overview» «Книги среды выполнения» (Book Of The Runtimr) и развернутым комментариям в начале файла `.\src\vm\threads.h` в исходном коде CoreCLR.

Анализ. Благодаря вышеупомянутым событиям ETW легко вычислить время приостановки и возобновления в целях GC. Проще всего заглянуть в отчет GCStats в PerfView. В таблице GC Events by Time приведены сводные данные по каждому событию, в т. ч. время, потраченное на приостановку потоков и выполнение GC (столбцы Suspend Msec и Pause MSec на рис. 7.19). Видно, что приостановка занимает гораздо меньше времени, чем сама сборка мусора.

GC Index	Pause Start	Trigger Reason	Gen	Suspend Msec	Pause MSec	% Pause Time	% GC	Gen0 Alloc MB	Gen0 Alloc Rate MB/sec
4	3,877.501	AllocSmall	2N	0.006	4.822	32.7	NaN	4.194	422.62
33	4,929.946	AllocSmall	2N	0.006	13.602	27.8	NaN	6.291	177.97
72	6,241.174	AllocSmall	2B	0.011	2.851	7.6	NaN	0.000	0.00
92	7,748.723	AllocSmall	2B	0.012	1.096	8.0	NaN	0.000	0.00
109	10,513.568	AllocSmall	2B	0.020	7.109	7.4	NaN	0.000	0.00
230	21,402.063	AllocSmall	2B	0.021	1.940	12.7	NaN	0.000	0.00
275	22,536.014	AllocSmall	2B	0.249	5.347	55.4	NaN	0.000	0.00
306	23,672.918	AllocSmall	2B	0.085	2.890	10.1	NaN	0.000	0.00
321	24,249.275	AllocSmall	2B	0.017	3.835	9.0	NaN	0.000	0.00
375	25,915.426	AllocSmall	2B	0.099	3.191	33.2	NaN	0.000	0.00
487	28,916.898	AllocSmall	2B	0.529	7.109	50.3	NaN	0.000	0.00
506	29,400.041	AllocSmall	2B	0.581	6.658	37.0	NaN	0.000	0.00
628	33,148.673	AllocSmall	2B	0.255	3.313	40.2	NaN	0.000	0.00
668	34,169.166	AllocSmall	2B	0.153	2.612	23.5	NaN	0.000	0.00

Рис. 7.19 ❖ Время, потраченное на приостановку потоков и выполнение GC. Таблица GC Events by Time в отчете GCStats в PerfView

Во время выполнения приложения не должно наблюдаться заметной задержки на приостановку потоков. Это, скорее всего, означало бы дефект в среде выполнения, поскольку у нас нет прямого контроля над механизмом приостановки. Например, в .NET 2.0 существовала ошибка, проявлявшаяся при редком стечении обстоятельств (компактные циклы, в которых выполняется один и тот же код без единой безопасной точки), из-за которых время приостановки могло увеличиваться до нескольких секунд. Ошибка была исправлена в .NET 4.0. В обычных приложениях чуть более длительное время приостановки (скажем, дольше 1 мс) может наблюдаться в случае длинной операции ввода-вывода или неудачно выбранных приоритетов потоков.

> Неуправляемые потоки не приостанавливаются и не возобновляются. Если вы создали фоновый платформенный поток, например для обратного вызова по таймеру, то он будет работать независимо от приостановки движка выполнения. Однако механизм P/Invoke должен быть заблокирован при возврате из неуправляемого кода в управляемый.

Выбор поколения для сборки

Если сборщик мусора запускается с указанием конкретного поколения (в частности, в результате вызванного вами метода `GC.Collect`), то он может решить, что сборке подлежит поколение старше указанного. Решение основано на различных внутренних эвристиках – каких именно, мы видели в разделе о статических и динамических данных GC.

В этом разделе приведен список причин, по которым может быть выбрано другое поколение. Это позволит лучше понять, какой GC имеет место в наших приложениях и почему.

Примите во внимание, что порядок описанных ниже решений важен. Каждое следующее решение (эвристика) может увеличить номер выбранного поколения, но никогда не уменьшает его. Например, если по результатам какой-то проверки решено выбрать поколение 2, а следующая проверка укажет на поколение 1, то в итоге будет выбрано более старое поколение (т. е. рекомендация выбрать поколение 1 будет проигнорирована).

Ниже приведен полный перечень причин для изменения выбранного поколения (в скобках указаны имена, под которыми причины фигурируют в таблице Condemned reasons for GCs в PerfView – лучшем и единственном месте, где можно проанализировать этот этап).

- Превышен бюджет выделения (Generation Budget Exceeded) – будет выбрано самое старое поколение, превысившее бюджет. Если бюджет превышен в куче больших объектов, то выбрано будет поколение 2 (так что запустится полная сборка мусора), но только если уже не запущена фоновая сборка мусора. К примеру, из-за превышения бюджета может быть выбрано старшее поколение, даже если первоначально в этом было замечено только поколение 0 – во время выделения памяти для объекта. Типичную ситуацию такого рода мы видели в сценарии 7.2.
- Оптимизация по времени (Time Tuning) – удивительно, но GC обращает внимание и на соотношение числа сборок в каждом поколении – по времени и по количеству. Делается это только в режиме рабочей станции и только в случае режима задержки Interactive или SustainedLowLatency. GC может выбрать поколение, если прошло достаточно много времени с момента последней сборки мусора в нем и количество сборок мусора в предыдущем поколении превысило некоторый порог. Величины порогов были приведены в табл. 7.1 и 7.2 в столбцах time_clock и gc_clock. В частности, это означает, что:
  - поколение 1 может быть выбрано, если в нем не производилась сборка в течение 10 секунд и в поколении 0 уже было произведено 10 сборок;
  - поколение 2 (полная сборка) может быть выбрано, если в нем не производилась сборка в течение 100 секунд и в поколении 1 уже было произведено 100 сборок.

  Это учитывает тот факт, что процессы, для которых GC работает в режиме рабочей станции, не так регулярны, как работающие в серверном режиме, поэтому GC нужен шанс заметить паттерны выделения и выживания как можно раньше.

  Иногда можно встретить фразу *Золотое правило* GC, согласно которому в нормально работающем приложении соотношение между количеством

сборок в поколениях должно быть равно 1:10:100. Заметим, однако, что это правило применимо только к GC в режиме рабочей станции и больше не считается справедливым в общем случае. «Нормальные» пропорции GC гораздо сложнее и динамичнее, чем эти простые отношения.

- Низкая эффективность таблицы карт (Internal Tuning) – в таблице карт накопилось слишком много «отказов поколения». Напомним (см. главу 5), что с таблицами карт сопряжены накладные расходы. Каждая карта представляет непрерывный участок памяти, в котором могут находиться объекты. Каждый такой объект может содержать ссылки на другие объекты, но лишь некоторые из них действительно будут межпоколенческими. Отношение количества полезных ссылок к общему количеству ссылок называется эффективностью таблицы карт. Если эффективность низкая, то мы напрасно обходим много объектов. Когда она опускается ниже некоторого порога, имеет смысл выбрать поколение 1. Тогда долгоживущие объекты предположительно должны оказаться в одном поколении, так что большинство межпоколенческих ссылок пропадет.
- Закончилось место в эфемерном сегменте (Ephemeral Low и Ephemeral Low with Very Fragmented Gen2) – не хватает места в сегменте, содержащем поколения 0 и 1 (точнее, в зарезервированной для сегмента памяти нет места для размещения объектов общим размером больше удвоенного минимального размера поколения 0). В таком случае поколение 1 выбирается с целью освободить эфемерную память (по причине Ephemeral Low). Кроме того, если фрагментация в поколении 2 настолько сильна, что может вместить (после уплотнения) все поколение 1, то выбирается поколение 2, что влечет за собой полную сборку мусора (по причине Ephemeral Low with Very Fragmented Gen2). Вообще говоря, это означает, что если в эфемерном сегменте кончается место, то сборщик мусора ведет себя более агрессивно (т. е. производит больше сборок в поколении 1), чтобы избежать выделения нового сегмента для кучи (или расширения существующего).
- Эфемерное поколение слишком фрагментировано (Fragmented Ephemeral) – если в эфемерном поколении (0 или 1) превышен порог фрагментации, то оно выбирается.
- Из-за исчерпания места в эфемерном сегменте требуется расширить его (Expand Heap) – если не существует другого способа разместить растущие эфемерные поколения, кроме как расширить сегмент, то выбирается поколение 2 (полная блокирующая сборка мусора).
- Исчерпание места во время выделения памяти (Compacting Full-GC) – последняя попытка перед возбуждением исключения `OutOfMemoryException`, запускается полная блокирующая сборка с уплотнением.
- Физическая память в системе занята более чем на 90 %[1] или операционная система отправила уведомление о недостатке памяти (High Memory) – если поколение 2 сильно фрагментировано или уже занято более чем на 10 % своего бюджета выделения, оно выбирается (во многих случаях производится блокирующая сборка). Это означает, что CLR может игнорировать

[1] Это значение используется в большинстве случаев. Для мощных машин с большим количеством логических ядер порог может быть повышен до 97 %.

уведомление о недостатке памяти от ОС и не запускать GC, если решит, что это не даст ощутимого результата. Но, вообще говоря, недостаток памяти у системы делает GC более агрессивным, если это с большой вероятностью сможет освободить память. Важно предотвращать лишний страничный обмен в системе в целом.

- Поколение 2 слишком сильно фрагментировано (Fragmented Gen2) – превышен порог фрагментации поколения 2, и оно будет выбрано.
- Поколение 2 или LOH слишком мало для фоновой сборки мусора (Small Heap) – в таком случае запускается полная блокирующая сборка.
- В режиме низкой задержки выбрать можно только поколение 0 или 1 (при этом все предыдущие решения отменяются).

И есть еще одна специальная причина, относящаяся к фоновой сборке мусора (описывается в главе 11), – запуск GC в эфемерных поколениях перед фоновой GC (Ephemeral Before BGC).

В некоторых описанных выше решениях важную роль играет превышение порога фрагментации. Спрашивается, что это такое. Для каждого поколения определен собственный порог, состоящий из двух значений, взятых из статических данных (табл. 7.1 и 7.2):

- суммарный размер памяти, потраченной впустую из-за фрагментации, непригодной для дальнейшего использования, а именно:
  - неиспользуемого свободного пространства, не управляемого распределителем поколения – сюда входят небольшие промежутки, образовавшиеся в процессе очистки (как будет показано ниже), – и места в эфемерных поколениях, отброшенного после неудачной попытки подобрать нужный размер (как было сказано в главе 5, элементы списка, свободные в этих поколениях, проверяются только один раз, а затем исключаются из списка);
  - ожидаемая эффективность распределителя – насколько хорошо удавалось повторно использовать элемент из списка свободных до сих пор;
- это значение представлено в столбце «Предельная фрагментация» в табл. 7.1 и 7.2 (см. сводку в табл. 7.4);

Таблица 7.4. Пороги фрагментации для различных поколений

	Предельная фрагментация	**Коэффициент фрагментации**
Gen0	40 000	75 %
Gen1	80 000	75 %
Gen2	200 000	50 %

- коэффициент фрагментации – отношение размера непригодного для использования фрагментированного пространства ко всему размеру поколения. Это значение вычисляется путем удвоения столбца «Предельная доля фрагментации» в табл. 7.1 и 7.2 с отсечением по уровню 75 % (см. сводку в табл. 7.4).

Например, поколение 2 будет считаться чрезмерно фрагментированным, если размер непригодной для использования фрагментированной памяти превышает 200 000 байт и эта величина больше 50 % всего размера поколения.

Сценарий 7.5. Анализ выбираемых поколений

Описание. Мы хотим понять, каковы типичные причины сборки мусора в нашем приложении, а также узнать, какие поколения выбираются и почему. Это очень глубокий вид анализа, нужный только в довольно специфических ситуациях (например, производится слишком много полных сборок, и нужно понять, почему они полные).

Анализ. В настоящее время нет лучшего способа понять причины выбора поколения, чем отчет GCStats в PerfView. После записи даже простейшего сеанса в режиме GC Collect Only мы увидим таблицу Condemned reasons for GCs, которая объясняет практически все (рис. 7.20). Этот сценарий является продолжением сценария 7.2, в котором были подробно проанализированы первые пять сборок мусора. Теперь мы можем посмотреть, как они описаны в PerfView. В процессе анализа сверяйтесь с именами в приведенном выше списке причин выбора поколения.

Condemned reasons for GCs

This table gives a more detailed account of exactly why a GC decided to collect that generation. Hover over the column headings for more info.

GC Index	Initial Requested Generation	Final Generation	Generation Budget Exceeded	Time Tuning	Induced	Ephemeral Low	Expand Heap	Fragmented Ephemeral	Very Fragmented Ephemeral	Fragmented Gen2	High Memory	Compacting Full GC	Small Heap	Ephemeral Before BGC	Internal Tuning
1	2	2	0	0	Blocking	0	0	0	0	0	0	0	1	0	0
2	0	2	2	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
5	1	2	2	0	0	1	0	0	0	0	0	0	0	0	0
6	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
7	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0
8	0	2	2	0	0	1	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0

Рис. 7.20 ❖ Таблица Condemned reasons for GCs из отчета GCStats в PerfView

Мы действительно находим здесь подтверждение нашего анализа первых пяти сборок в сценарии 7.2:

- GC 1 запущена принудительно (значение Blocking в столбце Induced) как полная (значение 2 в столбце Initial Requested Generation). И действительно, выполнена полная сборка мусора (значение 2 в столбце Final Generation);
- GC 2 изначально была запрошена для поколения 0 (значение 0 в столбце Initial Requested Generation) из-за превышения в нем бюджета выделения. Однако она была преобразована в полную сборку мусора, потому что превышен также бюджет выделения в поколении 2 (значение 2 в столбце Final Generation). Как мы знаем, на самом деле превышен бюджет выделения в LOH, но мы объясняли, что LOH в этом случае рассматривается как поколение 2;
- GC 3 изначально была запрошена для поколения 0 и для него и выполнена. Нет никаких причин выбирать другое поколение;
- GC 4 изначально была запрошена для поколения 0, но из-за превышения бюджета в поколении 1 в итоге было выбрано поколение 1;
- GC 5 изначально была запрошена для поколения 1 по причине OutOfSpaceSOH (значение 1 в столбце Ephemeral Low). Однако, поскольку бюджет поколения 2 исчерпан, она была преобразована в полную сборку.

Внимательный анализ таблиц Condemned reasons for GCs и GC Events by Time может многое прояснить в поведении сборщика мусора в вашем приложении. Однако это скучная и утомительная работа. Можно поискать в таблице Condemned reasons for GCs общие паттерны, часто повторяющиеся причины и т. д. К сожалению, на данный момент нет инструмента, который попытался бы обобщить и проанализировать причины выбора поколения в целом.

Определенно стоит обратить внимание на следующие столбцы, которые могут свидетельствовать о проблемах в вашем коде:

- Induced – явные вызовы GC редко бывают оправданны. Если они происходят часто, то надо бы выяснить, в чем дело (см. сценарий 7.3);
- Fragmented Ephemeral и Fragmented Gen2 – если значения в них велики, значит, имеют место проблемы с фрагментацией. Вероятно, стоит лучше разобраться в паттернах выделения памяти приложением (см. сценарии 5.2 и 6.2).

> Если вы хотите самостоятельно поизучать код CoreCLR, то внимательно познакомьтесь с методом `gc_heap::generation_to_condemn`. Все описанные причины выбора проверяются здесь одна за другой.

Резюме

В этой главе мы приступили к глубокому изучению самого сердца управления памятью в .NET – сборщика мусора. Мы начали с верхнего уровня – представили общую концепцию работы GC и объяснили назначение всех шагов. Затем мы подробно рассмотрели все основные этапы. В трех последующих главах все они будут описаны во всех деталях, а в этой мы ограничились первыми тремя:

- механизмы, запускающие сборку мусора;
- как среда выполнения организует приостановку движка выполнения, т. е. приостанавливает все управляемые потоки;
- как GC выбирает, в каком поколении производить сборку мусора.

Поскольку эти темы исключительно важны, мы рассмотрели также пять практических сценариев, в частности как анализировать использование GC и находить явные вызовы GC.

Располагая знаниями, полученными в этой главе, мы можем перейти к изучению следующих этапов GC. В следующей главе содержится подробное объяснение фазы пометки.

Глава 8

Сборка мусора – этап пометки

В предыдущей главе мы узнали о некоторых общих вопросах сборки мусора, например когда она запускается и как принимается решение о том, в каком поколении производить сборку. Теперь перейдем к деталям реализации первого этапа GC – пометки.

На этом этапе GC уже знает, в каком поколении собирать мусор. Настало время выяснить, какие объекты можно освободить. Как уже упоминалось, в CLR реализован отслеживающий сборщик мусора. Он начинает просмотр с корней и рекурсивно обходит весь текущий граф объектов. Те объекты, которых невозможно достичь из корней, считаются мертвыми (вспомните рис. 1.15).

В случае неконкурентного GC, описанного в этой главе, в начале этапа пометки все управляемые потоки приостанавливаются. Гарантируется, что управляемая куча не будет изменяться и останется в полном распоряжении сборщика мусора. Поэтому он может, ничего не опасаясь, начать поиск достижимых объектов.

Обход и пометка объектов

Хотя корней много, механизм поиска достижимых объектов общий. Зная адрес корня, процедура обхода выполняет следующие действия:

- преобразовать его в адрес управляемого объекта – в случае так называемого *внутреннего указателя* (который указывает не на начало, а на место внутри управляемого объекта). Это можно сделать эффективно благодаря описанному ниже механизму кирпичной кладки;
- установить флаг закрепления – если объект закреплен (а об этом можно узнать по тому, что обход начался из таблицы закрепленных описателей, или по флагу, передаваемому GC), в заголовке объекта устанавливается соответствующий бит;
- начать обход, следуя по ссылкам, хранящимся в объектах. Благодаря информации о типе (хранится в таблице методов) GC знает, каким смещениям (полям) соответствуют исходящие ссылки. Обход производится в глубину, т. е. строится коллекция объектов, подлежащих посещению. Эта коллекция называется *стеком пометки*, потому что она организована в виде стека с операциями заталкивания (push) и выталкивания (pop). При посещении объекта:

- уже посещенный объект просто пропускается;
- еще не посещенный объект помечается – для этого в указателе на таблицу методов объекта устанавливается один бит[1];
- хранящиеся в объекте исходящие ссылки добавляются в стек пометки.

Обход завершается, когда в стеке пометки не остается объектов.

> Типичная реализация обхода в глубину основана на рекурсивных вызовах. Однако трудно гарантировать, что при этом не произойдет переполнения стека. Проще и безопаснее заменить рекурсию итерацией, самостоятельно выделяя в куче память для стекоподобной структуры (стека пометки в случае CLR), которая может расти практически неограниченно.

Заметим, что флаги закрепления и пометки устанавливаются на этапе пометки, а используются и сбрасываются на этапе планирования. Когда объект живет нормальной жизнью (т. е. управляемые потоки работают), закрепление и пометка никак не отражаются ни в заголовке объекта, ни в указателе на таблицу методов.

> Если вас интересуют детали и вы хотите изучить код CoreCLR, начните с метода `GCHeap::Promote`. Он вызывает макрос `go_through_object_cl`, который запускает обход объектов по ссылкам. Основная работа ложится на метод `gc_heap::mark_object_simple1`, который реализует обход графа объектов в глубину с применением вспомогательной стекоподобной структуры `mark_stack_array` (индексы `mark_stack_bos` и `mark_stack_tos` указывают на дно и вершину стека).

Понимая общую структуру процесса пометки, давайте теперь подробно рассмотрим, какие корни GC могут существовать в приложении. Это знание чрезвычайно полезно для управления памятью в .NET. Корни могут удерживать целые подграфы достижимых объектов, что ведет к типичным проблемам:

- избыточное потребление памяти – мы даже можем не знать о существовании некоторых корней, из-за которых доступными оказываются гораздо больше объектов, чем мы ожидаем. Но вообще, существование большого числа объектов может быть дополнительными накладными расходами для GC;
- утечка памяти – хуже того, корни могут вызывать непрерывный рост удерживаемого ими подграфа объектов, т. е. постоянно увеличивающееся потребление памяти.

Корни – локальные переменные

Локальные переменные – один из самых распространенных типов корней. Некоторые из них существуют совсем недолго (листинг 8.1), тогда как другие – на протяжении всего времени жизни приложения (листинг 8.2). Мы постоянно создаем локальные переменные то тут, то там.

---

[1] Это не повреждает указатель на таблицу методов, потому что ее адрес в памяти выровнен на границу слова (кратен 4 или 8 байтам), так что два младших бита не используются (всегда равны 0). Для получения истинного указателя на ТМ из такого модифицированного указателя нужно только обнулить два младших бита – см. метод `GetMethodTable` в исходном коде CoreCLR.

Листинг 8.1 ❖ Пример локальной переменной fullPath с очень коротким временем жизни

```
public static void Delete(string path)
{
   string fullPath = Path.GetFullPath(path);
   FileSystem.Current.DeleteFile(fullPath);
}
```

Листинг 8.2 ❖ Пример локальной переменной host, живущей столько же, сколько приложение для ASP.NET

```
public static void Main(string[] args)
{
   var host = BuildWebHost(args);
   host.Run();
}
```

Часто мы создаем локальные переменные явно (как в листингах 8.1 и 8.2), но бывает, что они создаются неявно (как в листинге 8.3).

Листинг 8.3 ❖ Пример долгоживущей локальной переменной host, созданной неявно (по существу, это тот же код, что в листинге 8.2)

```
public static void Main(string[] args)
{
   BuildWebHost(args).Run();
}
```

Локальная переменная может представлять тип-значение (например, структуру) или ссылку на значение ссылочного типа[1] (вспомните важное различие между «ссылкой» и «данными ссылочного типа», обсуждавшееся в главе 4). В этом разделе рассматривается сборка мусора для объектов, размещенных в куче, поэтому нас будут интересовать детали, касающиеся локальных переменных, содержащих ссылки (независимо от того, является ли это типичным ссылочным типом как класс или упакованным (boxed) типом-значением).

Хранилище локальных переменных

Присваивая ссылку на управляемый объект локальной переменной, мы создаем корень, как в листинге 8.4, где ссылка на вновь созданный объект типа `SomeClass` присвоена локальной переменной `c`. Начиная с этого момента мы должны рассматривать данный объект как достижимый. Таким образом, предполагая, что `c` – единственный корень, мы не можем убрать объект в мусор, пока метод `Helper` не завершится, т. к. локальная переменная `c` используется на всем протяжении этого метода.

Листинг 8.4 ❖ Пример удержания объекта локальной переменной

```
private int Helper(SomeData data)
{
   SomeClass c = new SomeClass();
```

[1] Локальная переменная может представлять и примитивный тип (например, число), но такие типы нас не интересуют, поскольку для них не выделяется память в куче.

```
    c.Calculate(data);
    return c.Result;
}
```

Ситуация из листинга 8.4 показана на рис. 1.8 в главе 1. Мы привычно считаем, что локальные переменные находятся в стеке (поскольку ссылку можно рассматривать как тип-значение). Стало быть, ситуацию в листинге 8.4 можно представлять себе следующим образом: распределитель создает объект в управляемой куче, а локальная переменная `c` хранится в стеке в записи активации метода `Helper`. Но в главе 4 мы видели, что локальные переменные могут находиться и в регистрах процессора благодаря оптимизации, выполняемой JIT-компилятором. И этот факт настолько важен, что его можно повторять бесконечно: корни, представленные локальными переменными, могут находиться как в стеке, так и в регистрах. JIT-компилятор всячески стремится использовать регистры и место в стеке максимально эффективно.

Корни на стеке

Корни, описанные в этом разделе, называются *стековыми*. В разделе «Fundamentals of Garbage Collection» Руководства по .NET они описываются так: «стековые переменные, предоставляемые JIT-компилятором и обходчиком стека». Звучит туманно. Мы знаем, что речь идет о локальных переменных, определенных в исполняемом в данный момент методе, а также о локальных переменных, определенных во всех методах в текущем стеке вызовов. Именно к стеку вызовов относится термин «стековые корни». Но не забывайте, что такие «стековые корни» могут находиться в стеке или в регистре.

Во время приостановки движка выполнения необходимо обследовать стеки вызовов во всех управляемых потоках и найти все локальные переменные, потому что они могут являться стековыми корнями. Это и делает вышеупомянутый *обходчик стека*. Если в методе из стека вызовов определена локальная переменная, содержащая ссылку на управляемый объект, то этот объект считается живым, и с него начинается обход графа объектов. Однако не так-то просто ответить на вопрос, где находится локальная переменная, встречающаяся в данной строке метода (а точнее, в команде с известным адресом), и содержит она ссылку на объект или нет.

Как описано в разделе о приостановке в главе 7, потоки можно приостанавливать только в безопасных точках, а в их число входят почти все команды (в случае полностью прерываемых методов) или только вызовы методов (в случае частично прерываемых методов). Это приводит нас к выводу, что GC должен где-то хранить информацию о живых «стековых корнях» (как в стеке, так и в регистре) для каждой безопасной точки метода. Именно в этом и состоит смысл вышеупомянутой информации для GC (GC Info).

Лексическая область видимости

В C# от понятия локальной переменной неотделимо понятие *лексической области видимости*. Проще говоря, она определяет участок кода, в котором видна данная

переменная – с учетом всех вложенных блоков и т. д. В листинге 8.5 определены три локальные переменные:

- `c1` – представляет ссылку на управляемый объект типа `ClassOne`. Лексическая область видимости `c1` охватывает весь метод `LexicalScopeExample` – `c1` доступна в каждой точке метода, потому что объявлена в самой внешней области видимости внутри него;
- `c2` – представляет ссылку на управляемый объект типа `ClassTwo`. Лексическая область видимости `c2` ограничена блоком условия;
- `data` – локальная переменная примитивного (целого) типа.

Листинг 8.5 ❖ Пример двух локальных переменных с разными лексическими областями видимости

```
1  private int LexicalScopeExample(int value)
2  {
3    ClassOne c1 = new ClassOne();
4    if (c1.Check())
5    {
6       ClassTwo c2 = new ClassTwo();
7       int data = c2.CalculateSomething(value);
8       DoSomeLongRunningCall(data);
9       return 1;
10   }
11     return 0;
12 }
```

Теперь разберемся, как достижимость объекта, созданного внутри метода, связана с его лексической областью видимости.

Живые стековые корни и лексическая область видимости

Когда встает вопрос о достижимости объекта, представленного локальной переменной, сразу приходит на ум интуитивно понятное решение – ассоциировать ее с лексической областью видимости локальной переменной. Возьмем в качестве примера листинг 8.5.

- Созданный экземпляр класса `ClassOne` должен быть доступен на протяжении всего времени жизни метода – с момента создания (строка 3) до завершения метода (строка 11). Иными словами, локальная переменная `c1` является живым стековым корнем со строки 3 по строку 11.
- Созданный экземпляр класса `ClassTwo` достижим только внутри блока условия – с момента создания (строка 6) до конца блока (строка 9). То есть локальная переменная `c2` является живым стековым корнем со строки 6 по строку 9.

Приняв такой подход, мы можем описать информацию для GC для метода `LexicalScopeExample` из листинга 8.5:

- для полностью прерываемого случая, представленного в листинге 8.6, – с каждой строкой связана своя информация (понятно, что она создается на уровне ассемблера, но для краткости будем работать со строками кода на C#);

- для частично прерываемого случая, представленного в листинге 8.7, – информация хранится только для строк, в которых вызывается метод (в том числе конструктор, выделяющий память).

Листинг 8.6 ❖ Наглядное представление информации для GC для метода `LexicalScopeExample` из листинга 8.5 в полностью прерываемом случае. С каждой строкой ассоциированы живые стековые корни

```
1  Нет живых корней
2  Нет живых корней
3  Живой корень c1
4  Живой корень c1
5  Живой корень c1
6  Живой корень c1, живой корень c2
7  Живой корень c1, живой корень c2
8  Живой корень c1, живой корень c2
9  Живой корень c1, живой корень c2
10 Живой корень c1
11 Живой корень c1
12 Нет живых корней
```

Листинг 8.7 ❖ Наглядное представление информации для GC для метода `LexicalScopeExample` из листинга 8.5 в частично прерываемом случае. С каждой строкой ассоциированы живые стековые корни

```
3  Живой корень c1
4  Живой корень c1
6  Живой корень c1, живой корень c2
7  Живой корень c1, живой корень c2
8  Живой корень c1, живой корень c2
```

В информации для GC (GC Info) хранятся данные, относящиеся к ассемблерному коду, сгенерированному JIT-компилятором. Так что в ней, конечно, фигурируют не имена переменных C#, а позиции в стеке и регистры процессора. Так, в листинге 8.8 показана более реалистичная картина, чем в листинге 8.6 (предполагается, что JIT-компилятор назначил регистр `rax` локальной переменной `c1` и регистр `rbx` локальной переменной `c2`).

Листинг 8.8 ❖ Наглядное представление информации для GC (на уровне ассемблера) для метода `LexicalScopeExample` из листинга 8.5 в полностью прерываемом случае

```
1  Нет живых корней
2  Нет живых корней
3  Живые корни: rax
4  Живые корни: rax
5  Живые корни: rax
6  Живые корни: rax, rbx
7  Живые корни: rax, rbx
8  Живые корни: rax, rbx
9  Живые корни: rax, rbx
10 Живые корни: rax
11 Живые корни: rax
12 Нет живых корней
```

Представим себе, что на время сборки мусора среда выполнения приостановила поток, исполняющий метод `LexicalScopeExample`, на строке 7 (предполагается, что метод был JIT-компилирован как полностью прерываемый). Благодаря информации для GC, представленной в листинге 8.8, сборщик мусора знает, что живые стековые корни находятся в регистрах процессора `rax` и `rbx`. Процесс пометки можно начать с адресов, хранящихся в этих регистрах.

Это совершенно корректный подход, который дает правильные результаты. Лексическая область видимости ссылочной локальной переменной, очевидно, учитывается при установлении достижимости объекта ссылочного типа. Похожий, но более либеральный подход принимается при компиляции в режиме отладки. JIT-компилятор продлевает область достижимости всех локальных переменных до конца метода. Это очень полезно для отладки (например, просмотра значений переменных). Но в режиме выпуска (release mode) можно сделать больше для оптимизации использования памяти.

Живые стековые корни с ранней сборкой корней

Еще раз взглянув на код в листинге 8.5, мы можем заметить, что лексическая область видимости – не оптимальное представление достижимости. Из того факта, что локальная переменная может использоваться в лексической области видимости, еще не следует, что она и вправду используется. На самом деле важно только, используется переменная или нет. Взглянув на метод `LexicalScopeExample` под таким углом зрения, мы заметим, что:

- созданный экземпляр класса `ClassOne` не используется, начиная со строки 5, поэтому вопреки лексической области видимости переменной `c1` живым стековым корнем она является только в строках 3 и 4;
- созданный экземпляр класса `ClassTwo` используется только в строках 6 и 7, поэтому вопреки лексической области видимости переменной `c2` живым стековым корнем она является только в этих строках.

Иными словами, компилятор C# может понять, где реально используется каждый объект (посредством ссылающихся на него локальных переменных), и сохранить эту информацию. Затем JIT-компилятор сможет использовать ее в процессе назначения мест корням (переменным с более коротким временем жизни разрешено использовать ценные регистры) и при генерации информации для GC. Такая информация для GC гораздо эффективнее (листинг 8.9).

Листинг 8.9 ❖ Наглядное представление информации для GC (на уровне ассемблера) для метода `LexicalScopeExample` из листинга 8.5 в полностью прерываемом случае с применением ранней сборки корней

```
1  Нет живых корней
2  Нет живых корней
3  Живые корни: rax
4  Живые корни: rax
5  Нет живых корней
6  Живые корни: rax
7  Живые корни: rax
8  Нет живых корней
9  Нет живых корней
```

```
10 Нет живых корней
11 Нет живых корней
12 Нет живых корней
```

Новая информация для GC показывает, что в большей части метода живых стековых корней нет. Объект считается недостижимым из локальных переменных, когда он действительно уже не используется. Техника, позволяющая убирать объект как можно быстрее, называется *ранней сборкой корней*. Очевидно, что с точки зрения потребления памяти она эффективнее, потому что сокращает время жизни объекта до абсолютного минимума. А это, в свою очередь, означает, что регистры используются более плотно, поскольку их можно переназначать чаще (как, например, повторное использование регистра `rax` в листинге 8.9). Сгенерированная информация для GC в случае частично прерываемых методов оказывается еще короче (листинг 8.10).

Листинг 8.10 ❖ Наглядное представление информации для GC для метода `LexicalScopeExample` из листинга 8.5 в частично прерываемом случае с применением ранней сборки корней

```
3  Живые корни: rax
4  Живые корни: rax
6  Живые корни: rax
7  Живые корни: rax
```

> Во всех примерах из этого раздела использовались только регистровые корни. Это типичный сценарий, потому что JIT-компилятор делает все возможное, чтобы задействовать только ультраскоростные регистры процессора, а не стек. При некоторых условиях он может размещать корни в стеке, но описанные здесь механизмы не меняются, просто вместо имени регистра будет присутствовать позиция в стеке. Стековые корни представлены в виде смещения от адреса в регистре `rsp` или `rbp` (в зависимости от того, какой из них используется в методе). Поэтому в информации для GC всегда сохраняются текущие значения регистров `rsp` и `rbp` для каждой безопасной точки. Более того, в 64-разрядной среде JIT-компилятор с гораздо большей вероятностью будет использовать регистры, потому что на этой платформе добавлено восемь регистров общего назначения (`r8`–`r15`).

Во время приостановки потока сохраняется его текущий контекст (включая содержимое регистров). Так, если приостановка метода `LexicalScopeExample` произошла на строке 6, то в соответствии с информацией для GC имеется один живой стековый корень, и он находится в регистре `rax` (сохраненном в контексте). Та же логика повторяется для всех методов в стеке вызовов, для чего стек просматривается кадр за кадром (и восстанавливается контекст потока благодаря информации внутри записей активации, в частности значения регистров).

Ранняя сборка корней используется JIT-компилятором при компиляции в режиме выпуска (release mode). Иногда это может приводить к удивительному и даже сбивающему с толку поведению. Вопросы по этому поводу чаще всего начинаются словами: «В режиме отладки программа делает X. А в режиме выпуска – Y…»

Во-первых, присваивать локальной переменной значение `null` с целью «проинформировать» GC, что мы больше не будем использовать данный объект, в большинстве случаев необязательно (см. листинг 8.11). Даже перед вызовом долго ра-

ботающих методов – чтобы сказать GC: «Слушай, я сейчас запущу один метод, он будет работать долго, так что ты имей в виду, что вот этот объект уже не нужен, его можно убрать». Благодаря ранней сборке мусора компилятор и JIT-компилятор сами прекрасно разберутся в том, каковы реальные области видимости наших переменных, и явно об этом можно не сообщать. В этом отношении код в листингах 8.5 и 8.11 в точности эквивалентен, т. е. порождает одну и ту же информацию для GC и одинаковый ассемблерный код (JIT-компилятор вообще уберет лишние присваивания `null` переменным в процессе оптимизации).

Листинг 8.11 ❖ Пример излишнего присваивания null переменным

```
private int LexicalScopeExample(int value)
{
   ClassOne c1 = new ClassOne();
   if (c1.Check())
   {
      c1 = null;
      ClassTwo c2 = new ClassTwo();
      int data = c2.CalculateSomething(value);
      c2 = null;
      DoSomeLongRunningCall(data);
      return 1;
   }
   return 0;
}
```

> Из этого правила есть одно исключение – так называемые неотслеживаемые переменные (объясняются позже), для которых время жизни совпадает со временем жизни всего метода. Так что, если речь идет о чрезвычайно критических ресурсах, можно все-таки присвоить локальной переменной значение `null`, чтобы помочь JIT-компилятору.

Во-вторых, ранняя сборка корней может приводить к странным результатам, если применяется к объектам, методы которых имеют побочные эффекты. Мы можем ожидать, что у объекта будут эти побочные эффекты в течение времени жизни, основанного на лексической области видимости, тогда как в действительности время жизни напрямую с областью видимости не связано. Типичные случаи – использование таймеров, примитивов синхронизации (например, `Mutex`) или доступ к системным ресурсам (скажем, файлам).

В листинге 8.12 показано поведение, которое трудно объяснить, если не знать о ранней сборке корней. Интуитивно кажется, что время жизни объекта `Timer` должно совпадать с лексической областью видимости переменной `timer`. И следовательно, программа должна печатать текущее время, пока мы не нажмем какую-нибудь клавишу. Такое поведение наблюдается в отладочной сборке. Однако в режиме выпуска в игру вступает ранняя сборка корней. Поскольку объект `Timer` не используется начиная со строки 3, JIT-компилятор включит его в информацию для GC. После строки 3 объект `Timer` становится недостижимым! Если сборка мусора начнется, когда метод `Main` будет исполнять код после этой строки, то он будет убран (и печать текущего времени прекратится). В зависимости от того, как часто запускается сборка мусора, таймер может успеть напечатать текущее время несколько раз.

Листинг 8.12 ❖ Пример неожиданного поведения объекта Timer из-за ранней сборки корней

```
1  static void Main(string[] args)
2  {
3     Timer timer = new Timer((obj) => Console.WriteLine(DateTime.Now.ToString()), null, 0,
         100);
4     Console.WriteLine("Hello World!");
5     GC.Collect();  // смоделировать начало сборки мусора
6     Console.ReadLine();
7  }
Вот результат работы программы:
Hello World!
28/03/2018 14:29:01
```

В нашем примере сборка мусора инициирована явно, чтобы результаты были воспроизводимы. В настоящей программе сборка может произойти из-за выделения памяти в других потоках.

Более того, ранняя сборка корней настолько агрессивна, что объект может быть сочтен недостижимым, даже когда один из его методов все еще работает (если только этот метод не обращается к `this`). В листинге 8.13 моделируется такой сценарий. Когда метод `DoSomething` еще работает, начинается сборка мусора (и снова для демонстрации мы запускаем ее явно). Дополнительно в классе `SomeClass` определен метод-финализатор (что такое финализация, мы подробно объясним в главе 12), который выполняется, когда объект становится жертвой сборщика мусора.

Листинг 8.13 ❖ Пример неожиданного поведения объекта из-за ранней сборки корней

```
static void Main(string[] args)
{
   SomeClass sc = new SomeClass();
   sc.DoSomething("Hello world!");
   Console.ReadKey();
}

class SomeClass
{
   public void DoSomething(string msg)
   {
      GC.Collect();
      Console.WriteLine(msg);
   }

   ~SomeClass()
   {
      Console.WriteLine("Killing...");
   }
}
```

Вот что печатает программа:

```
Killing...
Hello world!
```

Удивительно, но программа выводит сообщение, из которого следует, что объект умер еще до того, как выполнился его метод. Дело в том, что метод `DoSomething` не обращается к `this`, так что ему в общем-то и не нужен экземпляр, от имени которого он вызван!

Но и это еще не все – при некоторых условиях ранняя сборка корней может «убрать» объект, когда какой-то из его методов работает и его код ссылается на `this`! Пример в листинге 8.14 моделирует такое поведение. Хотя метод `DoSomethingElse` ссылается на `this`, экземпляр `SomeClass` все равно быстро убирается, как в предыдущем примере.

Листинг 8.14 ❖ Пример неожиданного поведения объекта из-за ранней сборки корней

```
static void Main(string[] args)
{
   SomeClass sc = new SomeClass() { Field = new Random().Next() };
   sc.DoSomethingElse();
   Console.ReadKey();
}

class SomeClass
{
   public int Field;

   public void DoSomethingElse()
   {
      Console.WriteLine(this.Field.ToString());
      // продолжение кода
      Console.WriteLine("Am I dead?");
   }

   ~SomeClass()
   {
      Console.WriteLine("Killing...");
   }
}
```

Вот результат программы:

```
615323
Killing...
Am I dead?
```

Как такое возможно? А благодаря встраиванию метода (method inlining). Если JIT-компилятор решает встроить метод, то он становится частью вызывающего метода (листинг 8.15). Это может повлечь за собой дальнейшие оптимизации. Например, `DoSomethingElse` использует поле `this.Field` только в самом начале. После встраивания в метод `Main` `sc.Field` окажется последней ссылкой на объект, так что оставшаяся часть кода может выполняться, даже если этот объект будет убран сборщиком мусора.

Листинг 8.15 ❖ Пример неожиданного поведения объекта из-за ранней сборки корней

```
static void Main(string[] args)
{
```

```
   SomeClass sc = new SomeClass() { Field = new Random().Next() };
   Console.WriteLine(sc.Field.ToString());
   // продолжение кода
   Console.WriteLine("Am I dead?");
   Console.ReadKey();
}
```

Имейте в виду, что такие оптимизации встречаются очень часто, потому что JIT-компилятор крайне агрессивен в своем стремлении максимально сократить жизнь локальных переменных. В большинстве случаев применяемые приемы не изменяют логику программы. Неожиданное поведение случается очень редко и только в случае объектов с побочными эффектами, как-то связанными со временем их жизни.

Иногда нам по какой-то причине необходимо точнее контролировать время жизни объекта. Так, в листинге 8.12 может понадобиться, чтобы таймер работал на протяжении всего времени жизни приложения. Для таких случаев предназначен метод `GC.KeepAlive` (листинг 8.16).

Листинг 8.16 ❖ Исправление неожиданного поведения таймера, обусловленного ранней сборкой корней (см. листинг 8.12)

```
static void Main(string[] args)
{
   Timer timer = new Timer((obj) => Console.WriteLine(DateTime.Now.ToString()), null, 0, 100);
   Console.WriteLine("Hello World!");
   GC.Collect();     // смоделировать начало сборки мусора
   Console.ReadLine();
   GC.KeepAlive(timer);
}
```

Метод `GC.KeepAlive` – на самом деле простой трюк для продления времени жизни стекового корня. Его реализация не содержит никакого кода (см. листинг 8.17), но снабжена атрибутом `MethodImplOptions.NoInlining`. Он запрещает встраивание `KeepAlive`, что заставляет компилятор рассматривать переданный аргумент как используемый (и, следовательно, достижимый). Поэтому если используется `GC.KeepAlive`, то в информации для GC время жизни переданного ему объекта продлевается до точки вызова этого метода.

Листинг 8.17 ❖ Реализация метода `GC.KeepAlive` в библиотеке базовых классов

```
[MethodImplAttribute(MethodImplOptions.NoInlining)] // запретить оптимизации
public static void KeepAlive(Object obj)
{
}
```

> **ПРИМЕЧАНИЕ** В большинстве случаев объекты с побочными эффектами (например, `Mutex` и `Timer`) реализуют интерфейс `IDisposable`. Поэтому достаточно вызова `timer.Dispose()` в конце метода `Main` (или оператора `using`), чтобы продлить время его жизни, не прибегая к методу `GC.KeepAlive`. Но помнить о подводных камнях ранней сборки корней все равно стоит.

Информация для GC (GC Info)

Наглядные представления информации для GC в листингах с 8.6 по 8.10 были намеренными упрощениями. На самом деле информация для GC – очень плотно упакованная двоичная структура данных. Детали реализации интересны, но к нашим целям отношения не имеют. Идея же остается такой, как изложено выше.

В настоящее время просмотреть информацию для GC позволяет только отладчик WinDbg с расширением SOS. Чтобы увидеть ее в дампе памяти или в процессе, к которому присоединен WinDbg, найдите описание интересующего вас метода (MethodDesc) (листинг 8.18).

Листинг 8.18 ❖ Поиск управляемой кучи в WinDbg с загруженным расширением SOS

```
> .loadby sos coreclr
> !name2ee *!Scenarios.EagerRootCollection.LexicalScopeExample
...
Module:     00007ffea9944f30
Assembly:   Scenarios.dll
Token:      000000000600000d
MethodDesc: 00007ffea9948598
Name:       Scenarios.EagerRootCollection.LexicalScopeExample(Int32)
JITTED Code Address: 00007ffea9a63310
...
```

Затем информацию для GC можно просмотреть с помощью команды `!gcinfo <MethodDesc>` (листинг 8.19). Проанализируем с ее помощью метода `LexicalScopeExample` из листинга 8.5. Команда печатает разнообразные общие сведения о выбранном методе (вид возвращаемого значения, используется ли переменное число аргументов и т. д.). Но нам важно, что перечисляются также все безопасные точки вместе с живыми стековыми корнями в каждой из них (если таковые имеются). Для каждой безопасной точки указывается смещение команды от начала метода.

Листинг 8.19 ❖ Результат команды !gcinfo для метода LexicalScopeExample

```
> !gcinfo 00007ffea9948598
entry point 00007ffea9a63310
Normal JIT generated code
GC info 00007ffea9b29188
Pointer table:
Prolog size: 0
Security object: <none>
GS cookie: <none>
PSPSym: <none>
Generics inst context: <none>
PSP slot: <none>
GenericInst slot: <none>
Varargs: 0
Frame pointer: <none>
Wants Report Only Leaf: 0
Size of parameter area: 0
Return Kind: Scalar
```

```
Code size: 71
00000017 is a safepoint:
00000022 is a safepoint:
00000021 +rdi
0000002d is a safepoint:
00000040 is a safepoint:
0000004b is a safepoint:
0000004a +rdi
00000055 is a safepoint:
0000005c is a safepoint:
```

В информации о методе `LexicalScopeExample` в листинге 8.19 показано семь безопасных точек. Это, в частности, означает, что данный метод был JIT-компилирован как частично прерываемый. Для полностью прерываемых методов сохраняются только изменения стековых корней, а безопасные точки не перечисляются (как мы вскоре увидим). В листинге 8.19 только две безопасные точки содержат по одному стековому корню (хранящемуся в регистре `rdi`). Каждая безопасная точка отменяет все прочие стековые корни. Следовательно, листинг 8.19 позволяет заключить, что:

- регистр `rdi` является живым стековым корнем, начиная с команды со смещением 21 и до команды со смещением 2d;
- регистр `rdi` снова становится живым стековым корнем, начиная с команды со смещением 4a и до команды со смещением 55.

> Для полностью прерываемого метода может потребоваться значительный объем памяти (сравнимый с размером самого кода). В стремлении добиться компромисса между временем декодирования и эффективностью хранения основная часть информации для GC хранится в виде группы битов, представляющей изменения в составе живых стековых корней при переходе от одного участка кода к другому. Кроме того, для каждой группы запоминается начальное состояние жизнеспособности. Таким образом, чтобы декодировать сведения о жизнеспособности стековых корней для конкретного смещения в коде, сначала анализируется начальное состояние группы, а затем последовательно применяются изменения, пока не встретится интересующее смещение. Расширение WinDbg SOS проделывает это несколько раз (для каждого допустимого смещения команды в методе), и в итоге получается удобная сводка, которую мы видим в листингах.

Однако такая информация для GC безо всяких ссылок на код мало что говорит. К счастью, существует еще одна команда, которая чередует JIT-компилированный код с информацией для GC, – `!u -gcinfo <MethodDesc>` (листинг 8.20).

Листинг 8.20 ❖ Результат команды !u -gcinfo для метода LexicalScopeExample

```
> !u -gcinfo 00007ffea9948598
Normal JIT generated code
Scenarios.EagerRootCollection.LexicalScopeExample(Int32)
Begin 00007ff81c5e3310, size 71
push  rdi
push  rsi
sub   rsp,28h
mov   esi,edx
mov   rcx,7FF81C69AD08h (MT: Scenarios.EagerRootCollection+ClassOne )
call  CoreCLR!JIT_New
```

```
0017 is a safepoint:
mov   rdi,rax
mov   rcx,rdi
call  System_Private_CoreLib+0xc890f0 (System.Object..ctor())
0022 is a safepoint:
0021 +rdi
mov   dword ptr [rdi+8],esi
mov   rcx,rdi
call  00007ff8`1c5e2bb8 (Scenarios.EagerRootCollection+ClassOne .Check())
002d is a safepoint:
test  eax,eax
je    00007ff8`1c5e3378
mov   rcx,7FF81C69AFE8h (MT: Scenarios.EagerRootCollection+ClassTwo )
call  CoreCLR!JIT_TrialAllocSFastMP_InlineGetThread
0040 is a safepoint:
mov   rdi,rax
mov   rcx,rdi
call  System_Private_CoreLib+0xc890f0 (System.Object..ctor())
004b is a safepoint:
004a +rdi
mov   rcx,rdi
mov   edx,esi
call  00007ff8`1c5e2be0 (Scenarios.EagerRootCollection+ClassTwo.CalculateSomething(Int32),)
0055 is a safepoint:
mov   ecx,eax
call  00007ff8`1c5e2d70 (Scenarios.EagerRootCollection.DoSomeLongRunningCall(Int32))
005c is a safepoint:
mov   eax,1
add   rsp,28h
pop   rsi
pop   rdi
ret
xor   eax,eax
add   rsp,28h
pop   rsi
pop   rdi
ret
```

Анализ результата команды `!u -gcinfo` подтверждает, что безопасные точки устанавливаются только в местах вызова методов – как внутренних методов среды выполнения (распределителей), так и других управляемых методов (включая конструкторы). Информация для GC, показанная в листинге 8.20, очень похожа на ту, что мы видели в листинге 8.10. Можно заметить, что:

- во-первых, регистр `rdi` становится живым в команде со смещением 21 и остается таковым до следующей безопасной точки со смещением 2d, – это тот диапазон, в котором удерживается ссылка на объект класса `ClassOne` с момента его конструирования и до момента вызова метода `Check`;
- во-вторых, регистр `rdi` снова становится живым в команде со смещением 4a и остается таковым до следующей безопасной точки со смещением 55 – это тот диапазон, в котором удерживается ссылка на объект класса `ClassTwo` с момента его конструирования и до момента вызова метода `CalculateSomething`.

Чтобы увидеть, как информация для GC отображается для полностью прерываемого метода, нужно написать таковой. Как было сказано выше, только сам JIT-компилятор решает, какой код генерировать: полностью или частично прерываемый. Но если написать нетривиальный цикл с динамически изменяющимся количеством итераций, то с большей вероятностью будет сделан выбор в пользу полностью прерываемой версии (листинг 8.21).

Листинг 8.21 ❖ Пример метода, который, скорее всего, будет JIT-компилирован в полностью прерываемый код

```
private int RegisterMap(int value)
{
   int total = 0;
   SomeClass local = new SomeClass();
   for (int i = 0; i < value; ++i)
   {
      total += local.DoSomeStuff(i);
   }
   return total;
}

public int DoSomeStuff(int value)
{
   return value * value;
}
```

Взглянув на метод `RegisterMap` в WinDbg с помощью команды `!u -gcinfo`, мы действительно увидим, что сгенерирован полностью прерываемый код (листинг 8.22). Напомним, что это решение принимается на основе внутренних эвристик JIT-компилятора и может зависеть от версии, среды выполнения и других условий. Поэтому возможно, что для получения полностью прерываемого кода метод `RegisterMap` придется как-то модифицировать.

Листинг 8.22 ❖ Результат команды `!u -gcinfo` для полностью прерываемого метода `RegisterMap`

```
> !u -gcinfo 00007fff42c18518
Normal JIT generated code
Scenarios.EagerRootCollection.RegisterMap(Int32)
Begin 00007fff42d32f20, size 3d
push  rdi
push  rsi
sub   rsp,28h
mov   esi,edx
00000008 interruptible
xor   edi,edi
mov   rcx,7FFF42DEAAC8h (MT: Scenarios.EagerRootCollection+SomeClass)
call  CoreCLR!JIT_TrialAllocSFastMP_InlineGetThread
00000019 +rax
mov   rcx,rax
0000001c +rcx
call  System_Private_CoreLib+0xc890f0 (System.Object..ctor())
00000021 -rcx -rax
xor   eax,eax
```

```
test  esi,esi
jle   00007fff`42d32f54
mov   edx,eax
imul  edx,eax
add   edi,edx
inc   eax
cmp   eax,esi
jl    00007fff`42d32f47
mov   eax,edi
00000036 not interruptible
add   rsp,28h
pop   rsi
pop   rdi
ret
```

Даже внутри полностью прерываемого кода существуют непрерываемые участки (по умолчанию к ним относятся пролог и эпилог функции), и это отражено в листинге выше – прерываемый код начинается со смещения 8 и продолжается до смещения 36. Вместо безопасных точек в местах вызова методов мы видим, как изменяется жизнеспособность корней (в данном случае регистров `rax` и `rcx`). Поскольку все команды в прерываемых участках являются безопасными точками, печатать эту информацию нет смысла. Вооруженные знаниями, полученными в этой главе, и немного разбираясь в языке ассемблера, вы легко поймете, почему сгенерирована именно такая информация для GC. Например, имейте в виду, что благодаря встраиванию метода `DoSomeStuff` в цикл корни, связанные с объектом `SomeClass`, становятся мертвыми еще до начала цикла.

При использовании команд `!gcinfo` и `!u -gcinfo` можно столкнуться с так называемыми *неотслеживаемыми корнями*. Они представляют аргумент или локальную переменную, содержащие ссылку, но их время жизни неизвестно на этапе выполнения. GC предполагает, что неотслеживаемые корни живы на протяжении всего тела метода (если только не равны `null`).

> Если вы хотите исследовать этап пометки с точки зрения стековых корней, то начните с метода `gc_heap::mark_phase` и его обращения к методу `GCScan::GcScanRoots`. Он вызывает метод `Thread::StackWalkFrames`, передавая функцию обратного вызова `GCHeap::Promote`, которая будет вызываться для кадров текущего стека вызовов (в каждом управляемом потоке). Анализ функции `Promote` – отличная отправная точка для понимания алгоритма пометки в целом.

Закрепленные локальные переменные

Закрепленная локальная переменная – особый случай локальной переменной. Она неявно создается в C# и F#, когда используется ключевое слово `fixed` (листинг 8.23). В VB.NET закрепленных переменных нет, потому что в этом языке вообще не допускаются указатели.

Листинг 8.23 ❖ Пример использования ключевого слова `fixed` в C#

```
public class Program
{
   private List<byte[]> list = new List<byte[]>();
```

```
   public unsafe int Run()
   {
      // ...
      fixed (byte* array = list[7])
      {
         // ...
         Console.ReadLine();
      }
   }
}
```

Взглянув на CIL-код, сгенерированный для метода `Run` из листинга 8.23, мы увидим закрепленную (pinned) локальную переменную – в нашем случае она имеет индекс 2 (листинг 8.24). Ключевое слово pinned означает, что GC не должен перемещать такую переменную в другое место.

Листинг 8.24 ❖ Начало CIL-кода, сгенерированного для программы в листинге 8.23

```
.method public final hidebysig newslot virtual
instance int32 Run () cil managed
{
   // Header Size: 12 bytes
   // Code Size: 166 (0xA6) bytes
   // LocalVarSig Token: 0x11000016 RID: 22
   .maxstack 4
   .locals init (
      [0] int32 i,
      [1] uint8[] bigArray,
      [2] uint8& pinned 'array',
      [3] uint8[],
      [4] int32 i)
   // ...
   // IL code
}
```

Информация о закрепленных локальных переменных используется JIT-компилятором при генерировании информации для GC. Вместе с информацией о самом корне сохраняются сведения о том, что он закреплен. Это можно увидеть, взглянув на информацию для GC, сгенерированную для метода `Run` в листинге 8.24 (листинг 8.25). Позиция стека по адресу `sp+20` (т. е. с указанным смещением относительно указателя стека в начале выполнения метода) помечена как неотслеживаемая и закрепленная. Это означает, что на этапе пометки этот адрес в стеке будет рассматриваться как закрепленный корень, если поток приостановится в методе Run.

Листинг 8.25 ❖ Дизассемблированные фрагменты метода из листинга 8.24 (с информацией для GC)

```
> !u -gcinfo 00007ff9fa9277d8
Normal JIT generated code
CoreCLR.CollectScenarios.Scenarios.SOHCompactionWithPinning.Run()
Begin 00007ff9faa43070, size 103
Untracked: +sp+20(pinned)(interior)
```

```
00007ff9`faa43070 57              push    rdi
00007ff9`faa43071 56              push    rsi
00007ff9`faa43072 4883ec28        sub     rsp,28h
00007ff9`faa43076 33c0            xor     eax,eax
00007ff9`faa43078 4889442420      mov     qword ptr [rsp+20h],rax
...
00007ff9`faa430bd 488b4e08        mov     rcx,qword ptr [rsi+8]
00007ff9`faa430c1 ba07000000      mov     edx,7
00007ff9`faa430c6 3909            cmp     dword ptr [rcx],ecx
00007ff9`faa430c8 e8830d155e      call    System.Collections.Generic.List`1.get_Item(Int32)
...
00007ff9`faa430eb 4883c010        add     rax,10h
00007ff9`faa430ef 4889442420      mov     qword ptr [rsp+20h],rax
```

В листинге 8.25 приведены относящиеся к делу фрагменты метода. В начале выполнения метода позиция стека `sp+20` обнуляется. Затем вызывается метод `get_Item` обобщенного класса `List<T>`, и его результат (ссылка на седьмой элемент списка, содержащий ссылку на байтовый массив) сохраняется в регистре `rax`. Несколькими командами позже `rax` модифицируется с целью получить адрес, с которого начинаются данные в объекте-массиве. И в последней строке этот адрес сохраняется в позиции стека по адресу `sp+20`. Если поток будет приостановлен после этой строки, то GC увидит данный адрес и будет рассматривать весь объект как закрепленный.

> Именно по этой причине корень в позиции `sp+20` помечен как внутренний (interior). Хранящийся в этой позиции адрес фактически указывает внутрь объекта-массива и впоследствии интерпретируется сборщиком мусора соответствующим образом.

Такие закрепленные корни видны в течение короткого промежутка времени – пока исполняется содержащий их метод. Да и как закрепленные они помечаются только на время выполнения GC – на этапе просмотра корней, основанного на информации для GC. А на этапе планирования бит закрепления сбрасывается. Поэтому найти источники закрепления не тривиально. Так, маловероятно, что дамп памяти будет создан в середине сборки мусора. А если он создан в момент нормального выполнения приложения, то никаких следов закрепления мы не увидим.

Но есть инструменты, которые умеют перечислять источники закрепления. Имея информацию для GC для всех методов, выполняемых в текущих потоках, и зная состояние всех их локальных переменных, можно проверить, какие переменные были бы закреплены, если бы дамп памяти был создан в момент сборки мусора. Конечно, данные получатся приблизительными, потому что на время сборки потоки приостанавливаются в безопасных точках, а не обязательно там, где они были в момент создания дампа. Не забывайте также, что дамп – это всего лишь снимок состояния памяти в некоторый момент времени. Вовсе необязательно, что этот снимок состояния скажет что-то полезное о закреплении локальных переменных в целом. Чтобы составить более полное представление, нам понадобится много таких снимков.

По счастью, существует событие ETW `PinObjectAtGCTime`, которое генерируется всякий раз при закреплении объекта. Это ценный источник информации о закреплении объектов, в т. ч. локальных переменных.

Ниже будет показано, что в WinDbg можно перечислить закрепленные описатели. Однако это не то же самое, что обсуждаемые здесь закрепленные локальные переменные. Разницу можно увидеть, изучив счетчик \Память CLR .NET \Закрепленных объектов (\.NET CLR Memory\# of Pinned Objects), – он подсчитывает все закрепленные (неперемещаемые) объекты в процессе сборки мусора, тогда как в WinDbg мы видим только список закрепленных описателей. С другой стороны, PerfView умеет перечислять оба типа закрепленных корней в снимках состояния кучи. Все это, включая событие ETW PinObjectAtGCTime, мы изучим на практике в сценарии 9.2.

Просмотр стековых корней

Располагая изложенной выше информацией, легко понять, как информация для GC помогает находить стековые корни. Когда все потоки приостановлены в своих безопасных точках, мы можем декодировать информацию для GC и понять, какие живые корни существуют. Каждый из них (в стеке или в регистре) рассматривается как корень, с которого можно начинать обход графа объектов для пометки.

Может возникнуть вопрос, как в контексте стековых корней обрабатывается оператор goto. Он позволяет передать управление помеченной инструкции, т. е. выполнить безусловный переход. Это могло бы нарушить весь описанный выше механизм работы с информацией для GC – внезапно поток начинает выполнять команды, расположенные совершенно в другом блоке кода. Но оператор goto все же не настолько всемогущ. Вот что говорится о метках (целях goto) в спецификации языка C#: «На метку можно ссылаться в операторах goto (§ 8.9.3), находящихся в области видимости метки. Это означает, что оператор goto может передавать управление внутри блока или за пределы блока, но только не внутрь блока». Стало быть, оператор goto не может передать управление в другой метод. Не может он и совершить переход внутрь вложенного блока, пропустив весь промежуточный код. Иными словами, на оператор goto наложены ограничения, делающие его безопасным. Это полезно и с точки зрения информации для GC. При имеющихся ограничениях выполнение goto – это просто изменение указателя команды в пределах текущего метода.

Корни финализации

Финализация – это механизм, позволяющий определить некоторое поведение в момент сборки объекта в мусор. Чаще всего она применяется для того, чтобы гарантировать освобождение неуправляемых ресурсов, удерживаемых объектом. Из-за важности этого механизма и некоторых его подводных камней мы отложим подробное описание механизма финализации до главы 12.

А пока скажем лишь, что для отслеживания объектов, которые предстоит «финализировать», сборщик мусора ведет специальные очереди, в которых хранятся ссылки на объекты, «готовые к финализации». Они также являются корнями, которые следует просмотреть.

Просмотр очереди объектов, готовых к финализации, производится просто: GC перебирает объекты один за другим и для каждого начинает обход графа.

Если вы хотите посмотреть в исходном коде CLR, как обрабатываются корни финализации, начните с вызова метода CFinalize::GcScanRoots из метода gc_heap::mark_phase (с функцией обратного вызова GCHeap::Promote).

Больше практических, с точки зрения разработки, знаний о финализации вы найдете в главе 12.

Внутренние корни GC

В главе 5 было сказано, что в случае частичной сборки мусора необходимо рассматривать ссылки из старшего поколения на младшее (см. рис. 5.8). Этот шаг включает обход ссылок внутри объектов, хранящихся в межпоколенческих запомненных наборах (remembered sets), посредством механизма карт. Карточные слова и связки карт, описанные в главе 5, помогают быстро находить участки памяти, которые могут быть источниками таких ссылок. Они называются внутренними корнями GC, поскольку не связаны непосредственно с пользовательским кодом.

Имея информацию о картах, можно довольно просто организовать просмотр карт:

- во внешнем цикле ищутся непрерывные участки помеченных карт – они представляют области памяти, которые содержат объекты с межпоколенческими ссылками (назовем их областями помеченных карт). Для каждой такой области:
  - ищется первый объект (в случае кучи малых объектов с помощью техники кирпичей, описанной в главе 9);
  - в этой области объекты просматриваются поочередно – если объект содержит ссылки, то проверяется, действительно ли они межпоколенческие. Если да, то объект считается корнем, и с него начинается обход графа.

Попутно GC вычисляет коэффициент эффективности карты – отношение количества карт, указывающих именно на поколение 0, к количеству карт, указывающих на оба эфемерных поколения. Затем этот коэффициент используется для решения о том, в каком поколении производить сборку: если он слишком мал, то GC предпочитает выбрать поколение 1 вместо 0.

Просмотр карточных корней производится после описанного выше просмотра стековых корней. Это означает, что большинство объектов, скорее всего, уже посещались, так что просмотр карточных корней добавит немного.

> Для пометки с помощью карт служат методы `gc_heap::mark_through_cards_for_segments` (для SOH) и `gc_heap::mark_through_cards_for_large_objects` (для LOH), вызываемые из метода `gc_heap::mark_phase`.
>
> В версии для SOH используется метод `gc_heap::find_card`, чтобы найти области помеченных карт, и метод `gc_heap::find_first_object`, чтобы найти первый объект в такой области. Для найденных подобным образом объектов (содержащих исходящие ссылки) вызывается метод `gc_heap::mark_through_cards_helper`, который пробегается по полям, содержащим ссылки. Для межпоколенческих ссылок вызывается переданный в качестве функции обратного вызова метод `gc_heap::mark_object_simple`, который начинает обход графа.
>
> В версии для LOH логика аналогична и основана на методах `gc_heap::find_card` и `gc::heap::mark_through_cards_helper`. Основное отличие в том, что «грязные» области просматриваются объект за объектом, поскольку кирпичей там нет.

Низкая эффективность карт может стать причиной для выбора более старого поколения, чем запрошенное. Мы уже отмечали этот факт в разделе «Выбор поколения для сборки» главы 7. Такие ситуации можно наблюдать в PerfView с помощью таблицы Condemned reasons for the GCs в отчете GCStats – в этом случае столбец Internal Tuning (Внутренняя оптимизация) укажет, какое поколение было выбрано по этой причине.

Регулярные внутренние оптимизации – дело совершенно естественное, и беспокоиться по этому поводу не надо. С точки зрения пользователя, единственное последствие состоит в том, что сборка мусора будет произведена в поколении 1, а не в поколении 0, но разница невелика.

Корни – описатели GC

И последний тип корней – различные описатели (handle) GC. Мы уже встречались с ними в главе 4. Существуют разные типы описателей, но все они хранятся в одной глобальной таблице описателей. В этой таблице ищутся описатели разных типов, и хранящиеся в них адреса считаются корнями, с которых можно начинать обход графа. Упомянем два наиболее важных типа описателей:

- сильные описатели похожи на обычные ссылки. Их можно создать явно обращением к методу `GCHandle.Alloc`. Они также используются внутри CLR, например для хранения заранее созданных объектов исключений: `Exception`, `OutOfMemoryException`, `ExecutionEngineException` и т. п.;
- закрепленные описатели – подкатегория сильных описателей. Когда объект закрепляется с помощью типа описателя Pinned (посредством подходящего вызова метода `GCHandle.Alloc`), создается новый описатель типа «закрепленный» с данным объектом в качестве целевого. На этапе пометки такие описатели считаются корнями, а объекты, на которые они указывают, – закрепленными, т. е. в заголовке объекта поднимается «бит закрепления». Впоследствии он используется на этапе планирования (и сбрасывается до окончания сборки мусора).

Существует также важная разновидность типа закрепленного описателя – *асинхронный закрепленный описатель*. Семантика у него такая же, как у обычного закрепленного описателя (сделать объект неперемещаемым), но используется для внутренних надобностей CLR при выполнении асинхронного ввода-вывода (например, для чтения или записи в файл или сокет). У такого описателя есть дополнительное свойство – объект автоматически открепляется сразу по завершении асинхронной операции ввода-вывода (не дожидаясь явного освобождения описателя из программы). Это позволяет максимально сокращать время закрепления для таких популярных операций, что, конечно, хорошо с точки зрения снижения накладных расходов. Однако, поскольку этот механизм используется только для внутренних надобностей .NET, он не слишком интересен в повседневной работе. По крайней мере, если наша программа не выполняет настолько большое количество длительных асинхронных операций ввода-вывода, что возникающее в результате закрепление начинает вызывать проблемы (например, становится причиной высокой фрагментации).

Заметим, что описанное здесь закрепление по описателю (посредством вызова метода `GCHandle.Alloc(obj, GCHandleType.Pinned)`) отличается от закрепления с помощью ключевого слова `fixed` (описано выше в разделе «Закрепленные локальные переменные»). Результат одинаковый – объект не будет перемещаться во время уплотнения кучи, а единственная разница в том, где находятся такие объекты: в таблице описателей в случае метода `GCHandle` и в стеке в случае ключевого слова `fixed`.

Заметим, что корни-описатели играют гораздо более важную роль в текущей реализации среды выполнения, чем может показаться на первый взгляд. В куче больших объектов хранятся два важнейших массива (для каждого домена приложения): ссылок на интернированные строки и ссылок на статические объекты (рис. 8.1). Эти массивы закрепляет сама среда выполнения. Это существенно, поскольку в различных внутренних структурах CLR хранятся адреса элементов этих массивов. Например, на рис. 8.1 показана карта строковых литералов и хорошо видно, что она является не корнем для интернированных строк, а лишь вспомогательной структурой данных для быстрого поиска (ссылается на элементы массива, в котором хранятся ссылки на интернированные строки).

Интересно наблюдать за тем, как некоторые механизмы используются внутри CLR для реализации логики управления памятью!

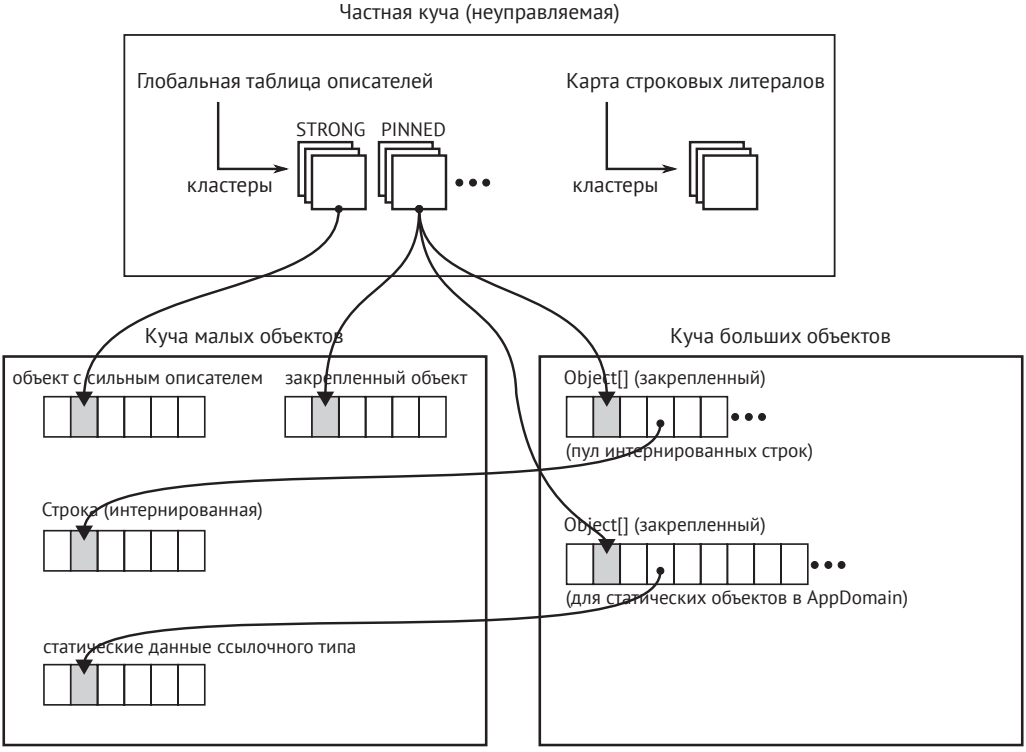


Рис. 8.1 ❖ Таблицы описателей как корни для различных управляемых объектов

> В исходном коде CoreCLR все начинается с метода `GCScan::GcScanHandles` (с функцией обратного вызова `GCHeap::Promote`), который вызывает `Ref_TracePinningRoots` (для типов `HNDTYPE_PINNED` и `HNDTYPE_ASYNCPINNED`), `Ref_TraceNormalRoots` (для типов `HNDTYPE_STRONG`, `HNDTYPE_SIZEDREF` и `HNDTYPE_REFCOUNTED`) и `Ref_ScanDependentHandlesForRelocation`.

Чтобы посмотреть на корни-описатели в действии, воспользуемся WinDbg с расширением SOS. Взяв очень простой код из листинга 8.26 в качестве примера, изучим, как отображаются корни различных объектов. Это знание окажется очень полезным, если придется анализировать различные ситуации неконтролируемого роста памяти, – мы должны понимать, каковы корни растущего графа объектов.

Листинг 8.26 ❖ Пример программы для демонстрации различных корней-описателей

```
public int Run()
{
   Normal normal = new Normal();

   Pinned onlyPinned = new Pinned();
   GCHandle handle = GCHandle.Alloc(onlyPinned, GCHandleType.Pinned);

   ObjectWithStatic obj = new ObjectWithStatic();
   Console.WriteLine(ObjectWithStatic.StaticField);

   Marked strong = new Marked();
   GCHandle strongHandle = GCHandle.Alloc(strong, GCHandleType.Normal);

   string literal = "Hello world!";
   GCHandle literalHandle = GCHandle.Alloc(literal, GCHandleType.Normal);

   Console.ReadLine();
   GC.KeepAlive(obj);
   // ... освободить описатели
   return 0;
}

public class Normal
{
}

[StructLayout(LayoutKind.Sequential)]
public class Pinned
{
   public long F1 = 301;
}

public class Marked
{
   public long F1 = 401;
}

public class ObjectWithStatic
{
   public static Static StaticField = new Static();
}

public class Static
{
   public long F1 = 501;
}
```

Присоединившись с помощью WinDbg к приложению, исполняющему этот код, в момент, когда оно ждет ввода в строке `Console.ReadLine`, мы можем исследовать различные корни объектов с помощью команды `!gcroot`. Для начала убедимся, что объект `normal` уже рассматривается как недостижимый, потому что JIT-компилятор (благодаря ранней сборке корней) должен заметить, что в этот момент он больше не используется (листинг 8.27).

Листинг 8.27 ❖ Normal-объект – недостижим вследствие ранней сборки корней

```
> !dumpheap -type CoreCLR.CollectScenarios.Scenarios.VariousRoots+Normal
         Address               MT   Size
000001c6b4dd26a0 00007fff8e84bce0     24

> !gcroot 000001c6b4dd26a0
Found 0 unique roots
```

Далее посмотрим, какие корни существуют для явно закрепленного объекта `onlyPinned` (листинг 8.28). Заметим, что результат согласуется с рис. 8.1 – сообщается, что корень является описателем (закрепленным) в таблице `HandleTable`, т. е. в неуправляемой внутренней структуре данных CLR (листинг 8.28).

Листинг 8.28 ❖ Закрепленный объект – достижим из таблицы закрепленных объектов (неуправляемой)

```
> !dumpheap -type CoreCLR.CollectScenarios.Scenarios.VariousRoots+Pinned
         Address               MT    Size
000001c6b4dd26b8 00007fff8e84be80      24

> !gcroot 000001c6b4dd26b8
HandleTable:
   000001c6b0d015d8 (pinned handle)
   -> 000001c6b4dd26b8 CoreCLR.CollectScenarios.Scenarios.
      VariousRoots+Pinned
Found 1 unique roots

> !gcwhere 000001c6b0d015d8
Address 0x1c6b0d015d8 not found in the managed heap.
```

Статические данные ссылочного типа представлены полем `ObjectWithStatic.StaticField` типа `Static`. Корни, отображаемые для такого объекта, также согласуются с рис. 8.1. Ссылка на экземпляр хранится в массиве, находящемся в LOH (здесь LOH названа поколением 3), который удерживается закрепленным описателем в таблице `HandleTable` (листинг 8.29).

Листинг 8.29 ❖ Статический объект достижим из закрепленного массива в LOH (на который ведет указатель из неуправляемой таблицы закрепленных описателей)

```
> !dumpheap -type CoreCLR.CollectScenarios.Scenarios.VariousRoots+Static
         Address               MT    Size
000001c6b4dd2700 00007fff8e84c3b0      24

> !gcroot 000001c6b4dd2700
HandleTable:
    000001c6b0d015f8 (pinned handle)
    -> 000001c6c4dc1038 System.Object[]
```

```
    -> 000001c6b4dd2700 CoreCLR.CollectScenarios.Scenarios.VariousRoots+Static
Found 1 unique roots

> !gcwhere 000001c6c4dc1038
Address          Gen   Heap            segment               begin
allocated              size
000001c6c4dc1038   3      0   000001c6c4dc0000    000001c6c4dc1000
000001c6c4dc5480    0x1ff8(8184)
```

Такие массивы типа `System.Object[]` нередко можно встретить в качестве корней различных объектов, но не дайте сбить себя с толку. Чаще всего это связано с тем, что речь идет о статических объектах или интернированных строках, как в нашем примере.

Сильный описатель похож на закрепленный – сообщается, что на объект `strong` из листинга 8.26 ссылается описатель (типа string) из таблицы `HandleTable` (листинг 8.30).

Листинг 8.30 ❖ Объект с сильным описателем достижим из таблицы описателей (неуправляемой)

```
> !dumpheap -type CoreCLR.CollectScenarios.Scenarios.VariousRoots+Marked
         Address               MT    Size
000001c6b4dd26d0 00007fff8e84c020      24

> !gcroot 000001c6b4dd26d0
HandleTable:
    000001c6b0d01190 (strong handle)
    -> 000001c6b4dd26d0 CoreCLR.CollectScenarios.Scenarios.
       VariousRoots+Marked
Found 1 unique roots
```

У строкового литерала из листинга 8.26 должно быть два корня. Один из них – пул интернированных строк (закрепленный массив в LOH, содержащий ссылки на интернированные строки), другой – созданный явно сильный описатель. Результат команды `!gcroot` это подтверждает (листинг 8.31).

Листинг 8.31 ❖ Строковый литерал с дополнительным сильным описателем (экземпляр, найденный командой `!dumpheap -mt 00007fffed021400 -min 32 -max 32`)

```
> !do 000001c6b4dd2650
Name:        System.String
MethodTable: 00007fffed021400
EEClass:     00007fffebcdddc0
Size:        50(0x32) bytes
File:        F:\GithubProjects\coreclr\bin\Product\Windows_NT.x64.Debug\
             System.Private.CoreLib.dll
String:      Hello world!

> !gcroot 000001c6b4dd2650
HandleTable:
    000001c6b0d01198 (strong handle)
    -> 000001c6b4dd2650 System.String

    000001c6b0d015e8 (pinned handle)
```

```
    -> 000001c6c4dc3050 System.Object[]
    -> 000001c6b4dd2650 System.String
Found 2 unique roots
```

Дополнительно мы можем проверить, что у обычного объекта `ObjectWithStatic` нет корней-описателей, а есть только стековые корни (листинг 8.32), точнее, один корень в регистре `rsi`.

Листинг 8.32 ❖ Нормальный объект все еще достижим по стековому корню (хранящемуся в регистре `rsi`) благодаря вызову `GC.KeepAlive`

```
> !dumpheap -type CoreCLR.CollectScenarios.Scenarios.VariousRoots+ObjectWithStatic
         Address               MT    Size
000001c6b4dd26e8 00007fff8e84c200      24

> !gcroot 000001c6b4dd26e8
Thread 273c:
    000000793097d530 00007fff8e79319d CoreCLR.CollectScenarios.Scenarios.VariousRoots.Run()
       rsi:
         -> 000001c6b4dd26e8 CoreCLR.CollectScenarios.Scenarios.VariousRoots+ObjectWithStatic
Found 1 unique roots
```

Иногда очень полезно перечислить все описатели (или только описатели определенного типа) в приложении, в чем нам поможет команда `!gchandles` (листинг 8.33).

Листинг 8.33 ❖ Команда `!gchandles` перечисляет все описатели в приложении (с возможностью фильтрации)

```
> !gchandles
         Handle Type                                Object           Size
Data Type
000001c6b0d013e8 WeakShort              000001c6b4dc1e20            152
System.Buffers.ArrayPoolEventSource
000001c6b0d017a8 WeakLong               000001c6b4dd2740            152
System.RuntimeType+RuntimeTypeCache
000001c6b0d017f8 WeakLong               000001c6b4dc2878             64
Microsoft.Win32.UnsafeNativeMethods+ManifestEtw+EtwEnableCallback
000001c6b0d01190 Strong                 000001c6b4dd26d0             24
CoreCLR.CollectScenarios.Scenarios.VariousRoots+Marked
000001c6b0d01198 Strong                 000001c6b4dd2650             50
System.String
000001c6b0d011a0 Strong                 000001c6b4dc2de0             32
System.Object[]
000001c6b0d011a8 Strong                 000001c6b4dc2d78            104
System.Object[]
000001c6b0d011b0 Strong                 000001c6b4dc13e0             24
System.SharedStatics
000001c6b0d011b8 Strong                 000001c6b4dc1300            144
System.Threading.ThreadAbortException
000001c6b0d011c0 Strong                 000001c6b4dc1270            144
System.Threading.ThreadAbortException
000001c6b0d011c8 Strong                 000001c6b4dc11e0            144
System.ExecutionEngineException
```

```
000001c6b0d011d0 Strong                 000001c6b4dc1150            144
System.StackOverflowException
000001c6b0d011d8 Strong                 000001c6b4dc10c0            144
System.OutOfMemoryException
000001c6b0d011e0 Strong                 000001c6b4dc1030            144
System.Exception
000001c6b0d011f8 Strong                 000001c6b4dc13f8            128
System.AppDomain
000001c6b0d015d8 Pinned                 000001c6b4dd26b8             24
CoreCLR.CollectScenarios.Scenarios.VariousRoots+Pinned
000001c6b0d015e0 Pinned                 000001c6c4dc3488           8184
System.Object[]
000001c6b0d015e8 Pinned                 000001c6c4dc3050           1048
System.Object[]
000001c6b0d015f0 Pinned                 000001c6b4dc13a8             24
System.Object
000001c6b0d015f8 Pinned                 000001c6c4dc1038           8184
System.Object[]
// ...
// статистические данные
```

> Существуют и другие типы описателей, например слабый описатель, о котором будет рассказано в главе 12. Но с точки зрения этой главы они ничем не отличаются, так что для краткости мы их опустили.

Анализ утечек памяти

Вам доводилось наблюдать, как потребление памяти .NET-приложением со временем растет? Конечно, механизм пометки весьма сложный, но не надо сразу подозревать его в ошибках. Иначе говоря, растущее потребление памяти и утечки в приложении вызваны не ошибками в алгоритме определения достижимости объектов! Если утечка памяти имеет место, то, скорее всего, она связана с тем, что кто-то удерживает ссылки на какие-то объекты. Поэтому самая типичная проблема в части управления памятью в .NET – найти источник утечки памяти, т. е. понять, какие корни не дают ее освободить.

Но прежде всего необходимо выяснить, а есть ли вообще утечка, и если да, то находится ли она в управляемом коде. Поэтому расследование должно начинаться следующими двумя шагами:

- проверить, какая часть памяти процесса растет. Может оказаться, что это ошибка в какой-то неуправляемой библиотеке или вы неправильно ее используете, так что утекает неуправляемая память. Как проводится такая диагностика, описано в главе 4;
- если утечка неуправляемой памяти исключена, то остается только разбираться с управляемой, а как это делается, описано ниже.

Единственным верным признаком того, что управляемая память течет, является постоянный рост потребления, несмотря на регулярные полные сборки мусора в поколении 2. В противном случае может просто оказаться, что сборщик мусора еще не решил, что пора производить полную сборку. Бывает так, что память, занятая поколением 2, растет, но полная сборка не запускается, потому что еще не

выполнены некие условия – с точки зрения GC, в этом попросту нет необходимости (например, свободной памяти еще достаточно). Возможно также, что фоновая полная сборка производилась, но без уплотнения, так что потребление памяти растет из-за фрагментации. Только если полные сборки с уплотнением производятся, но не останавливают рост памяти, мы можем заподозрить утечку.

Чтобы различить эти два случая, следует начать с общих измерений динамики GC – производятся ли сборки мусора в поколении 2 и сколько их? Для этого можете использовать тот инструмент, который больше нравится, например системный монитор или отчет GCStats в PerfView. Располагая полученными из этой книги знаниями, вы сможете определить, почему не запускаются полные сборки мусора, изучив таблицы GC Events by Time и Condemned reasons for GCs из отчета GCStats.

Убедившись, что полная сборка в поколении 2 все же производится, можно заняться исследованием причин утечки памяти. Какие корни удерживают все больше и больше объектов, мешая их удалению сборщиком мусора? Ответить на этот вопрос нелегко. В простых приложениях иногда достаточно тщательно проанализировать изменения, зафиксированные в системе управления исходным кодом, – поскольку проблема чаще всего проявляется после развертывания новой версии приложения. Но рассчитывать на такую удачу не стоит.

В более крупных приложениях, где количество объектов измеряется десятками тысяч или миллионами, причем они постоянно создаются и убираются, найти истинный источник утечки по-настоящему трудно. Запутанный лабиринт связей между объектами не облегчает задачу. Есть два основных подхода к диагностике утечек памяти.

- Первый проще, но сильнее зависит от удачливости и включает анализ одного дампа памяти приложения. Мы ищем большое количество объектов, на которые приходится много занятой памяти. Предварив анализ дополнительными измерениями, мы сможем определить, к примеру, конкретное поколение (почти всегда это будет поколение 2 или LOH), что сузит область поиска. Можно заметить, что много объектов создается в какой-то определенной части приложения (конкретная прикладная логика, какая-то сквозная функциональность или некая технология, например доступ к базе данных). В таком случае очень полезны будут знания о структуре приложения и организации исходного кода. Но это не отменяет того факта, что для подобного анализа необходима развитая интуиция. В приложении может быть много больших групп объектов, и не все из них обязательно должны быть источниками утечки. Некоторые просто необходимы для нормальной работы. Поэтому анализ утечек памяти превращается в трудоемкую, но достойную награды работу детектива. Именно такой подход представлен в сценарии 5.2 из главы 5 и в сценарии 8.1 ниже. Иногда помогает анализ нескольких дампов памяти процесса, созданных в моменты растущего потребления. Их последовательное рассмотрение может подстегнуть нашу интуицию. Но подсказку можно получить и в результате автоматического сравнения таких снимков памяти. И это подводит нас ко второму методу анализа.
- Второй, более предпочтительный, подход подразумевает анализ двух и более последовательных дампов памяти с акцентом на различия между ними. Это упрощает задачу – в сложной системе взаимосвязанных объектов мы,

возможно, сумеем выделить растущие группы объектов. Чем лучше используемый инструмент, тем проще будет такой анализ. Но все равно необходима интуиция и знание структуры приложения, потому что растущих групп может быть несколько (и лишь одна из них растет непреднамеренно). Этот подход тоже будет представлен в сценарии 8.1.

Поскольку анализ памяти – утомительное и непростое занятие, не существует рецепта на все случаи жизни. Чаще всего анализ реальной проблемы включает все три вышеупомянутые методики и направлен на поиск причины слой за слоем.

И еще один совет напоследок. Вне зависимости от специфики приложения и источника утечки памяти, в анализируемом дампе самыми многочисленными объектами, скорее всего, будут строки. Это характерно для всех современных приложений, они обрабатывают текст – полученный из файлов, HTTP-запросов, баз данных и т. д. Так что имейте строки в виду, но начинать с них анализ необязательно. Быть может, строки приведут к источнику проблемы, а быть может, и нет, но не исключено, что они укажут на него, потому что накапливающиеся объекты вполне могут содержать какие-то строковые данные.

Сценарий 8.1. Утечка памяти в nopCommerce?

Описание. У нас установлено приложение nopCommerce – платформа электронной торговли с открытым исходным кодом, написанная на ASP.NET. Мы хотим проанализировать производительность nopCommerce, в том числе паттерны использования памяти. Мы подготовили простой сценарий нагрузочного тестирования для программы JMeter 3.2 – популярного приложения для нагрузочного тестирования с открытым кодом. Здесь будут в цикле выполняться три действия: посещение домашней страницы, одной категории (Computers) и поиск по одной метке («awesome»). Мы добавили паузу между запросами, чтобы смоделировать поведение реального пользователя. В процессе тестирования мы обратили внимание, что потребление памяти растет, хотя сборка мусора в поколении 2 производится регулярно. Похоже, имеет место утечка памяти! Ниже описывается альтернативный подход к той же проблеме, что и в сценарии 5.2.

Анализ. Мы знаем, что управляемая память течет во время нагрузочного тестирования (рис. 8.2). Полная сборка мусора производится, но создается впечатление, что в поколении 2 накапливаются долгоживущие объекты. Мы попытаемся найти причину, воспользовавшись описанными выше методами анализа утечек памяти. В качестве инструмента возьмем PerfView, потому что эта программа обладает великолепными средствами для сбора и анализа снимков памяти. Прежде чем использовать ее для этой цели, прочитайте очень подробную и хорошо написанную справку по темам Collecting GC Heap Data (Сбор данных о сборке мусора в куче) и Understanding GC Heap Data (Как устроена управляемая куча GC), на которые есть ссылки в диалоговом окне **Collecting Memory Data**.

> **ПРИМЕЧАНИЕ** Безусловно, прежде чем приступать к анализу памяти .NET, необходимо убедиться, что мы действительно имеем дело с утечкой памяти. Смотрите сценарии 4.2, 4.3, 4.4.

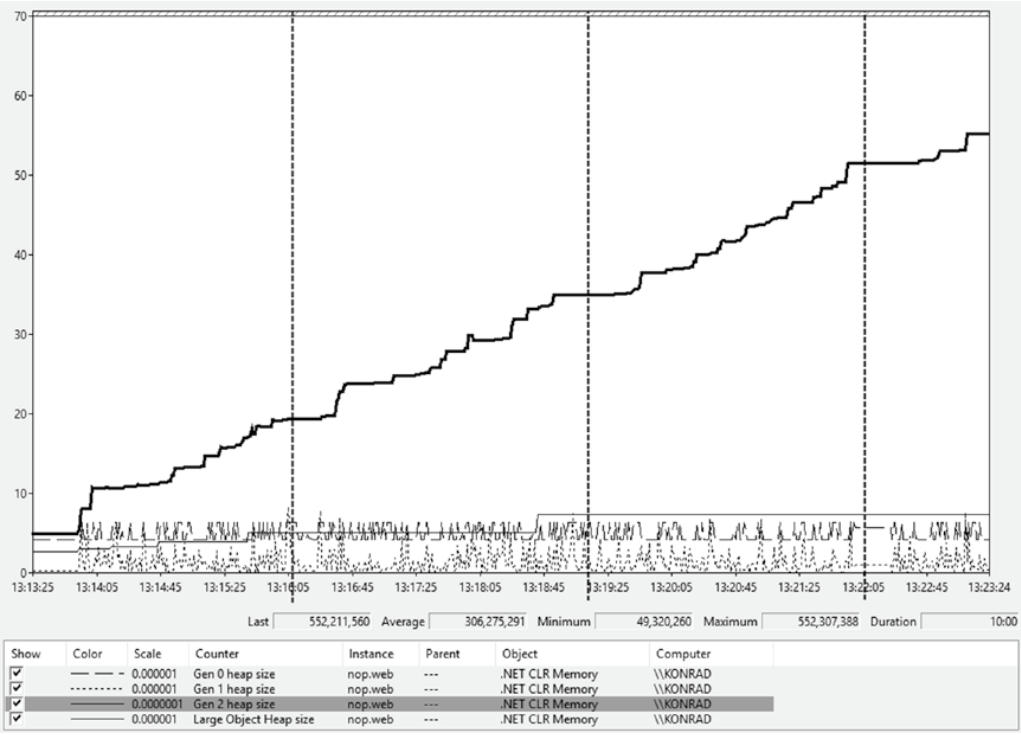


Рис. 8.2 ❖ График счетчиков производительности
за первые 10 минут нагрузочного тестирования – представлены все поколения.
Отмечены моменты создания дампов памяти

ПОДХОД 1 – АНАЛИЗ ОДНОГО СНИМКА ПАМЯТИ В этом случае мы берем один снимок памяти, созданный в PerfView (**Memory ➤ Take Heap Snapshot option**), первый из тех, что показаны на рис. 8.2. Изучение статистики объектов открывает интересные вещи. На рис. 8.3 показаны сводные сведения о том, как используется память различными объектами (включая дочерние), список упорядочен по общему размеру занятой памяти. Понятно, что корни [.NET Roots] ссылаются на все данные, так что вместе со своими потомками занимают 100 % памяти (см. столбец Inc). Большая часть корней – статические (что само по себе интересно), однако похоже, что память в основном занята контейнером Autofac IoC (он удерживает 74 % всех объектов). Быть может, это и есть корень зла, а возможно, и нет – неудивительно, что в приложении, использующем контейнер зависимостей, большинство объектов находятся в нем. Впрочем, это хоть какой-то след. Помимо этого, велика доля объектов, связанных с кешем в памяти.

Еще более наглядное представление об этих данных можно получить из пламенной диаграммы (рис. 8.4). Она подтверждает сделанные наблюдения, и создается впечатление, что посредством контейнера Autofac в памяти удерживается очень много объектов класса `Microsoft.Extensions.Caching.Memory.CacheEntry`.

Name ?	Exc % ?	Exc ?	Exc Ct ?	Inc % ?	Inc ?	Inc Ct ?	Fold ?	Fold Ct ?
[.NET Roots]	0.3	407,591	8,736	100.0	144,170,600.0	3,724,871	407,591	8,735
ROOT	0.0	0	0	100.0	144,170,600.0	3,724,871	0	0
[static vars]	5.4	7,779,199	176,526	97.5	140,536,300.0	3,664,079	7,779,199	176,525
[static var Nop.Core.Infrastructure.Singleton.allSingletons]	0.0	0	1	74.0	106,755,100.0	2,792,296	0	0
LIB <<mscorlib!Dictionary<Type,Object>>>	0.0	1,404	46	74.0	106,755,100.0	2,792,295	1,284	43
Nop.Core!Nop.Core.Infrastructure.NopEngine	0.0	12	1	74.0	106,753,700.0	2,792,249	0	0
Autofac.Extensions.DependencyInjection!Autofac.Extensions.DependencyInjection.AutofacServiceProvider	0.0	138	12	74.0	106,753,700.0	2,792,248	0	0
Autofac!Autofac.Core.Container	0.3	476,568	14,590	74.0	106,753,600.0	2,792,237	476,548	14,589
Autofac!Autofac.Core.Lifetime.LifetimeScope	0.0	247	5	73.7	106,276,800.0	2,777,605	0	0
LIB <<mscorlib!Dictionary<Guid,Object>>>	2.7	3,917,683	84,396	73.7	106,276,300.0	2,777,582	3,917,455	84,391
Microsoft.Extensions.Caching.Memory!Microsoft.Extensions.Caching.Memory.MemoryCache	0.0	64	1	48.4	69,808,310.0	1,963,239	0	0
LIB <<mscorlib!ConcurrentDictionary<Object,Microsoft.Extensions.Caching.Memory.CacheEntry>>>	10.7	15,486,060	203,556	48.4	69,808,250.0	1,963,238	15,485,970	203,553
Microsoft.Extensions.Caching.Memory!Microsoft.Extensions.Caching.Memory.CacheEntry	11.1	16,030,910	222,660	38.0	54,777,400.0	1,772,371	1,725,223	114,307
Nop.Core!Nop.Core.Caching.MemoryCacheManager	0.0	16	1	21.3	30,773,610.0	703,282	0	0
LIB <<mscorlib!CancellationTokenSource>>	18.1	26,143,850	477,556	21.3	30,773,590.0	703,281	26,143,160	477,539

Рис. 8.3 ❖ Представление By Name снимка кучи в PerfView (отсортировано по столбцу Inc в порядке убывания)

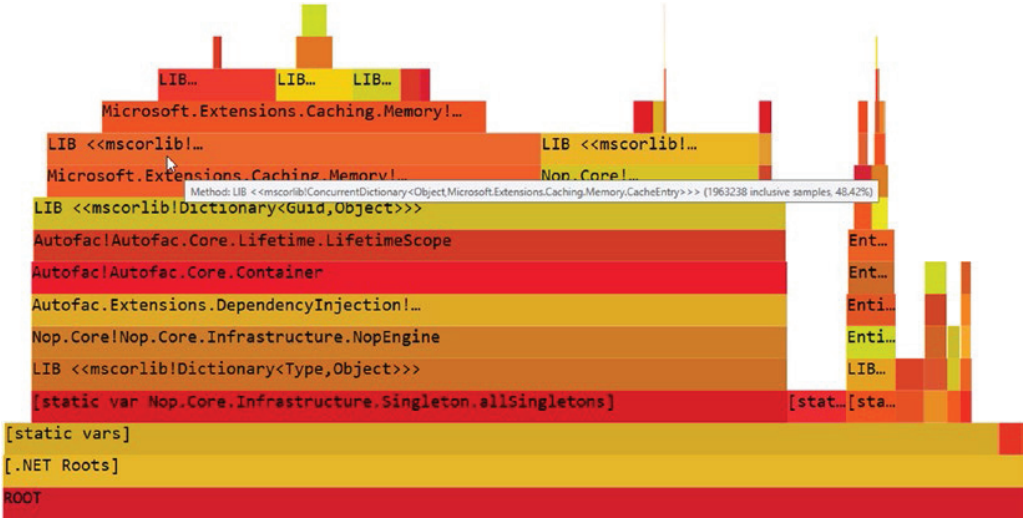


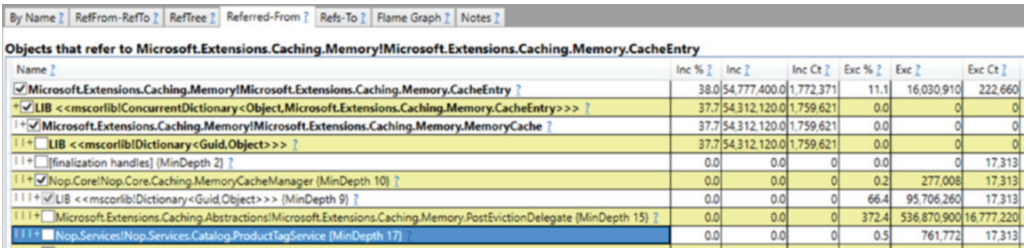
Рис. 8.4 ❖ Пламенная диаграмма снимка кучи в PerfView (подсказка, вспывающая при наведении мыши, оставлена намеренно)

Но все еще остается шанс, что это ожидаемое поведение, если приложение предназначено для кеширования большого объема данных. Настораживает только постоянный рост потребления памяти – быть может, мы все время кешируем, но ничего не освобождаем? На этом этапе, несомненно, имеет смысл взять исходный код и еще раз проверить механизмы кеширования. Но мы можем немного помочь себе, взглянув на объекты приложения, ссылающиеся на объекты класса `CacheEntry`.

Перейдя на вкладку **Referred-From** для объектов `CacheEntry`, мы найдем кое-какие ключи. Посмотрев на объекты, относящиеся к nopCommerce, мы быстро обнаружим экземпляры класса `Nop.Core.Caching.MemoryCacheManager`, удерживаемые экземплярами класса `Nop.Services.Catalog.ProductTagService` (рис. 8.5). Их не так много, но сам факт открывает новые направления поиска.

Теперь мы можем посмотреть в исходном коде, как использует кеш служба `ProductTagService`, и обнаружить ту самую истинную причину, которая уже была описана в сценарии 5.2, поэтому не будем повторяться. Понятно, что проблема

не в nopCommerce, а в том, что мы плохо подготовили нагрузочный тест. Имейте в виду, что это не искусственный пример, а самый что ни на есть реальный, с которым я лично сталкивался.



By Name | RefFrom-RefTo | RefTree | Referred-From | Refs-To | Flame Graph | Notes

Objects that refer to Microsoft.Extensions.Caching.Memory!Microsoft.Extensions.Caching.Memory.CacheEntry

Name	Inc %	Inc	Inc Ct	Exc %	Exc	Exc Ct
Microsoft.Extensions.Caching.Memory!Microsoft.Extensions.Caching.Memory.CacheEntry	38.0	54,777,400.0	1,772,371	11.1	16,030,910	222,660
LIB <<mscorlib!ConcurrentDictionary<Object,Microsoft.Extensions.Caching.Memory.CacheEntry>>>	37.7	54,312,120.0	1,759,621	0.0	0	0
Microsoft.Extensions.Caching.Memory!Microsoft.Extensions.Caching.Memory.MemoryCache	37.7	54,312,120.0	1,759,621	0.0	0	0
LIB <<mscorlib!Dictionary<Guid,Object>>>	37.7	54,312,120.0	1,759,621	0.0	0	0
[finalization handles] (MinDepth 2)	0.0	0.0	0	0.0	0	17,313
Nop.Core!Nop.Core.Caching.MemoryCacheManager (MinDepth 10)	0.0	0.0	0	0.2	277,008	17,313
LIB <<mscorlib!Dictionary<Guid,Object>>> (MinDepth 9)	0.0	0.0	0	66.4	95,706,260	17,313
Microsoft.Extensions.Caching.Abstractions!Microsoft.Extensions.Caching.Memory.PostEvictionDelegate (MinDepth 15)	0.0	0.0	0	372.4	536,870,900	16,777,220
Nop.Services!Nop.Services.Catalog.ProductTagService (MinDepth 17)	0.0	0.0	0	0.5	761,772	17,313

Рис. 8.5 ❖ Представление Referred-From снимка кучи в PerfView (для типа Microsoft.Extensions.Caching.Memory.CacheEntry)

При таком подходе к исследованию проблемы бывает трудно отыскать истинную причину. Это связано с очень кратковременными, но при этом исключительно важными связями между объектами. Например, в нашем случае `Nop.Services.Catalog.ProductTagService` действительно хранит ссылку на `Nop.Core.Caching.MemoryCacheManager` на протяжении запроса, но она быстро исчезает, и оказывается, что записи `CacheEntry` удерживает сам механизм кеширования.

ПОДХОД 2 – СРАВНЕНИЕ СНИМКОВ ПАМЯТИ В этом случае мы создаем два снимка памяти в PerfView (**Memory ➤ Take Heap Snapshot option**), второй и третий из тех, что показаны на рис. 8.2. Их разделяет три минуты, поскольку анализируемая утечка памяти весьма внушительна. Иногда нужно сравнивать снимки, отстоящие друг от друга на несколько десятков минут. Имея оба снимка, мы можем сравнить их с помощью меню **Diff**. Результаты, показанные в представлении By Name, говорят сами за себя (рис. 8.6). Подавляющее большинство новых объектов имеют тип `CacheEntry`, а остальные как-то связаны с кешированием – на это указывают положительные значения в столбцах **Exc** (суммарный размер объектов данного типа без учета дочерних) и **Inc** (с учетом дочерних), означающие, что за время между двумя снимками суммарный размер таких объектов увеличился.

Name	Exc %	Exc	Exc Ct	Inc %	Inc	Inc Ct	Fold	Fold Ct
ROOT	0.0	0	0	100.0	68,687,080.0	1,700,950	0	0
[.NET Roots]	0.0	-18,380	3,929	100.0	68,687,080.0	1,700,950	-18,380	3,929
[static vars]	1.5	997,758	33,546	93.2	64,034,470.0	1,652,405	997,758	33,546
Microsoft.Extensions.Caching.Memory!Microsoft.Extensions.Caching.Memory.CacheEntry	19.4	13,345,800	102,257	82.4	56,583,260.0	1,518,337	5,019,994	39,201
LIB <<mscorlib!List<IDisposable>>>	43.0	29,524,780	645,303	71.0	48,765,210.0	1,238,704	28,007,190	582,108
LIB <<mscorlib!Action<Microsoft.Extensions.Caching.Memory.CacheEntry>>>	0.0	128	4	15.1	10,366,680.0	386,366	0	0
LIB <<mscorlib!ConcurrentDictionary<Object,Microsoft.Extensions.Caching.Memory.Cach	4.5	3,077,782	74,918	15.1	10,366,560.0	386,362	3,077,781	74,918
Microsoft.Extensions.Caching.Memory!Microsoft.Extensions.Caching.Memory.MemoryCac	0.0	0	0	15.1	10,366,560.0	386,362	0	0
LIB <<mscorlib!List<Microsoft.Extensions.Caching.Memory.PostEvictionCallbackRegistrati	4.8	3,280,897	126,321	9.2	6,308,843.0	252,470	1,768,929	63,187

Рис. 8.6 ❖ Представление By Name различий двух снимков кучи в PerfView (отсортировано по столбцу Inc в порядке убывания)

В правильно функционирующей системе количество новых записей в кеше было бы примерно равно количеству уже вытесненных из кеша (в предположе-

нии, что число обращений к странице стабильно). Таким образом, суммарный размер экземпляров CacheEntry с учетом дочерних и других связанных с кешем типов должен быть близок к нулю[1]. Это прямо указывает на проблемы с механизмом кеширования. Теперь можно внимательнее проанализировать экземпляры класса CacheEntry в одном или обоих снимках, как в первом подходе.

Сценарий 8.2. Нахождение самых популярных корней

Описание. Мы хотели бы проанализировать самые популярные типы корней в приложении. Это может стать полезным дополнительным ключом для анализа утечки памяти. Если самые популярные корни изменяются со временем, то можно обнаружить интересные закономерности и прийти к определенным выводам. Напрасно было бы ожидать, что такой анализ приведет нас прямиком к причине проблемы. Но в кропотливом процессе поиска истины чем больше различных подсказок, тем лучше.

Анализ. События, генерируемые средой выполнения, – богатейший источник знаний. Это относится и к статистическим сведениям о корнях. Существует событие ETW/LTTng MarkWithType, которое дает информацию о том, сколько было помечено байтов, относящихся к корням различных видов (и, стало быть, достижимых), во время конкретной сборки мусора. Для каждого вида корней генерируется одно событие, поэтому чаще всего одной сборке мусора соответствует несколько событий. Виды корней представлены числами, определенными в перечислении _GC_ROOT_KIND (листинг 8.34).

Листинг 8.34 ❖ Перечисление, описывающее виды корней

```
namespace ETW
{
   typedef enum _GC_ROOT_KIND {
      GC_ROOT_STACK = 0,
      GC_ROOT_FQ = 1,
      GC_ROOT_HANDLES = 2,
      GC_ROOT_OLDER = 3,
      GC_ROOT_SIZEDREF = 4,
      GC_ROOT_OVERFLOW = 5
      } GC_ROOT_KIND;
   };
}
```

Событие MarkWithType генерируется, если в диалоговом окне **Collect в PerfView** выбран режим **GC Collect Only**. В результате мы сможем увидеть все события в представлении Events и отфильтровать те, которые относятся к интересующему нас процессу (рис. 8.7). К сожалению, в PerfView нет ни сводной таблицы, ни графического представления таких данных, что сильно затрудняет анализ подобных событий.

[1] Очевидно, что на самом деле трафик колеблется, так что это число не равно в точности нулю.

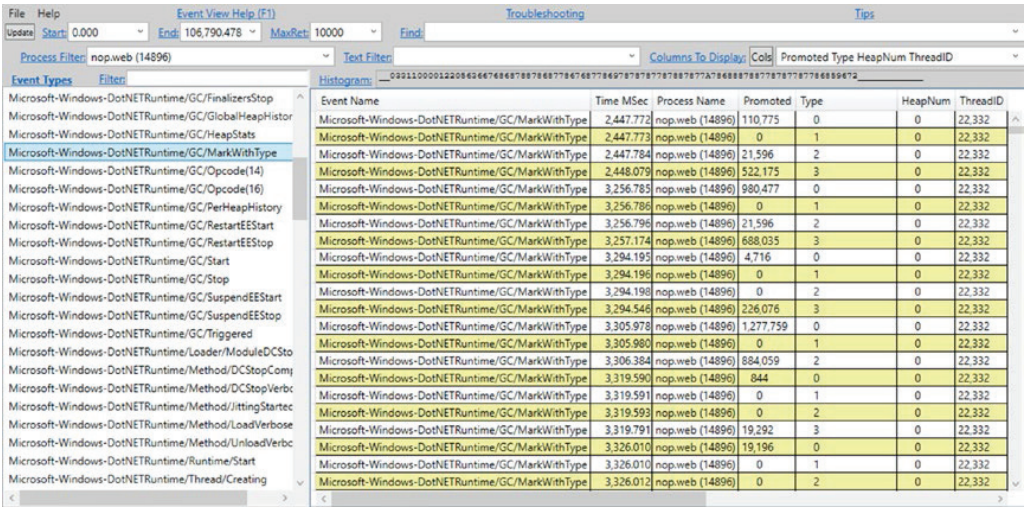


Рис. 8.7 ❖ Событие MarkWithType для демонстрационного процесса (отображаются столбцы Promoted, Type, HeapNum и ThreadID)

Однако мы можем экспортировать отфильтрованные события в формате CSV-файла (команда **Open View in Excel** в контекстном меню) и проанализировать его в любой понимающей этот формат программе, например MS Excel или еще какой-нибудь электронной таблице. Вот на рис. 8.8 показана зависимость от времени суммарного размера переведенных объектов для корней различных видов (графики подготовлены в MS Excel). Заметим, что шкала по вертикальной оси логарифмическая.

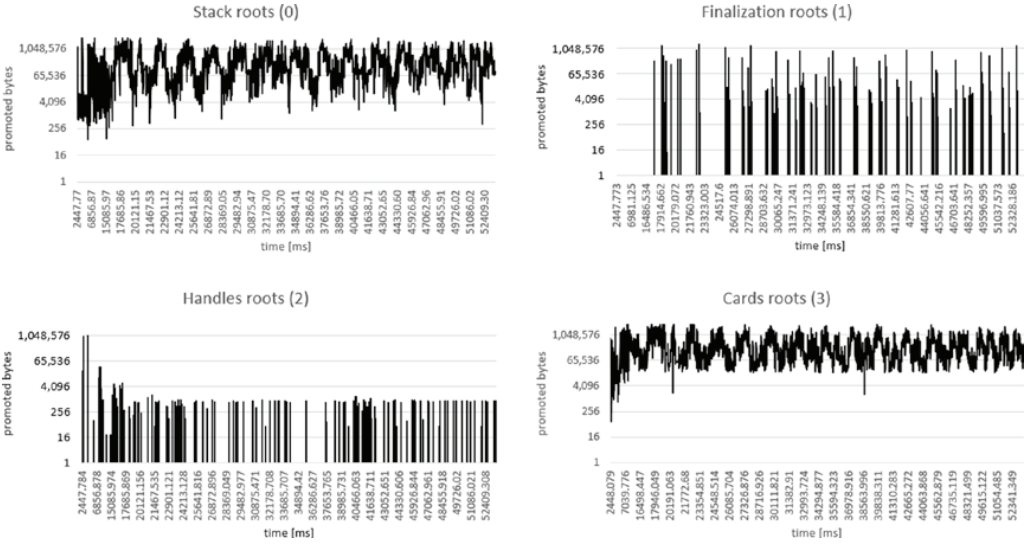


Рис. 8.8 ❖ Зависимость суммарного размера переведенных объектов от времени для корней различных видов

Примите во внимание, что значения в этих событиях по большей части инкрементные (исключение составляет редко встречающийся `GC_ROOT_SIZEDREF`), порядок определен в листинге 8.34. Каждое последующее событие `MarkWithType` показывает, сколько байтов было дополнительно переведено в следующее поколение в связи с корнями данного типа. Например, количество байтов, переведенных в связи с корнями финализации, не учитывается в объектах, уже помеченных в связи со стековыми корнями. События для корней-описателей не учитываются в объектах, переведенных в следующее поколение из-за стековых корней и корней финализации, и т. д.

Резюме

В этой главе мы подробно рассмотрели первую часть сборки мусора – этап пометки. Без его понимания невозможно сказать, какие объекты и почему умирают или остаются в живых. Так что это один из самых практически важных аспектов знания о памяти в .NET вообще.

Процедура пометки начинается с корней различных видов и постепенно строит граф достижимых объектов. Поскольку флаг пометки, хранящийся в объекте, учитывается при рассмотрении корней следующего вида, объекты, на которые ссылаются уже помеченные объекты (т. е. целые подграфы полного графа объектов), повторно не посещаются. Корни следующего типа просто дополняют уже построенный граф.

Надеюсь, что приведенное в этой главе полное описание механизма пометки поможет вам лучше понять, как он работает. Особенно удивительным может показаться тот факт, что интернированные строки и статические ссылочные данные обрабатываются тем же механизмом, что и все остальные объекты!

Ну а теперь пора перейти к следующему важному шагу процедуры сборки мусора – этапу планирования.

Глава 9

Сборка мусора – этап планирования

По завершении этапа пометки мы знаем про каждый объект, достижим он или нет. Достижимые объекты помечены специальным битом. Некоторые из них могут быть дополнительно помечены битом закрепления. В этот момент у сборщика мусора есть вся информация, необходимая для работы. Но возникает вопрос: какую стратегию выбрать – очистку или уплотнение?

Ответ на этот вопрос можно получить двумя способами. Можно высказать обоснованную гипотезу, например исходя из шаблонов предшествующего использования памяти или предыдущих результатов очистки и уплотнения. Но это все же будет только гипотезой. А в таких динамичных обстоятельствах, как непрерывное создание и удаление объектов, трудно ожидать, что гипотеза окажется чем-то большим, чем выигрыш в лотерею.

Вместо попыток угадать мы можем каким-то образом рассчитать, окупится ли уплотнение при текущих условиях или фрагментация не слишком велика, так что можно обойтись и очисткой. Это гораздо более многообещающий подход. В зависимости от точности вычислений мы можем приблизиться к оптимальному решению. Но, как мы вскоре увидим, точно предсказать итоговую фрагментацию нелегко (в основном из-за закрепления). Налицо парадокс: чтобы узнать, есть ли смысл производить уплотнение, мы должны его произвести и посмотреть, что получилось.

Но как уплотнить, не уплотняя? Именно в этом и состоит задача этапа планирования. Он производит вычисления так, что результаты прямо соотносятся с информацией о результате процесса уплотнения. Эти данные готовятся «в стороне» – без реального перемещения объектов. А в итоге мы будем точно знать, чем закончится возможное уплотнение.

Более того, информация подготавливается таким образом, что позже ей смогут воспользоваться как процедура уплотнения, так и процедура очистки. Если уплотнение обещает хорошие результаты (а как принимается решение, мы рассмотрим дальше), то GC выполняет его, пользуясь ранее подготовленной информацией. А если достаточно очистки, то та же информация используется для очистки. Поскольку уплотнение производится гораздо чаще очистки, особенно в эфемерных поколениях, то результаты имитированного уплотнения редко оказываются невостребованными.

Итак, этап планирования можно рассматривать как главный двигатель всего процесса GC. Именно здесь производятся все сложные, но необходимые вычис-

ления. Этапы очистки и уплотнения всего лишь пользуются результатами этих вычислений более или менее сложным, но все же понятным образом.

Так как же творится это волшебство, как этап планирования ухитряется «выполнить» одновременно очистку и уплотнение, не изменяя объектов в управляемой куче? Ответ очень интересен, так что приглашаю вас к дальнейшему чтению. Описание будет довольно подробным, ведь мы находимся в самом сердце GC. Разобравшись в этапе планирования, вы поймете, как на самом деле работает GC. Я верю, что эти усилия того стоят!

Описываемые в данной главе процессы для SOH и для LOH немного различаются.

Куча малых объектов

Начнем с описания планирования в SOH. Оно несколько сложнее, чем в LOH, так что, поняв, как оно устроено, мы уже легко разберемся и с LOH.

Заполненные и пустые блоки

Представьте себе фрагмент управляемой кучи (точнее, кучи малых объектов) в самом начале процесса GC (рис. 9.1). Несколько объектов расположены вплотную друг к другу. В каждом объекте есть заголовок, указатель на таблицу методов и хотя бы одно поле размером с указатель (даже если оно не используется, см. главу 4). Одни объекты больше, другие меньше.



Рис. 9.1 ❖ Фрагмент управляемой кучи (внутри кучи малых объектов) в самом начале процесса GC (H означает заголовок, MT – указатель на таблицу методов, объекты заштрихованы светло-серым цветом)

После этапа пометки, описанного в предыдущей главе, все достижимые объекты помечены (рис. 9.2). Теперь в игру вступает этап планирования.



Рис. 9.2 ❖ Фрагмент управляемой кучи сразу после этапа пометки (умеренно-серым цветом изображены помеченные объекты)

На этапе планирования объект за объектом просматривается выбранное и более молодые поколения. Это легко, потому что размер текущего объекта можно вычислить по хранящейся внутри него «горячей» информации. Для массивов это базовый размер объекта плюс количество элементов, умноженное на размер одного элемента. В процессе просмотра выделенный указатель просто сдвигается на размер текущего объекта (выровненного в памяти).

Главный принцип этапа планирования – объединить все помеченные и непомеченные объекты в группы во время этого просмотра (рис. 9.3). Таким образом, создаются группы двух видов:

- *заполненный блок* (plug) – представляет сплошную группу помеченных (достижимых) объектов;
- *пустой блок* (gap) – представляет сплошную группу непомеченных (недостижимых) объектов.

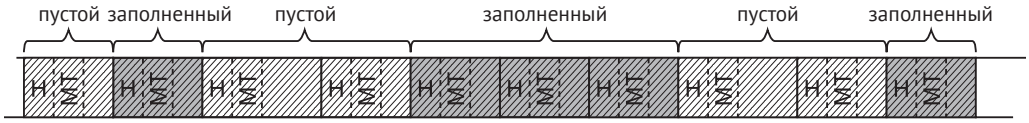


Рис. 9.3 ❖ Заполненные и пустые блоки в управляемой куче

Разбив всю управляемую кучу на последовательность заполненных и пустых блоков, мы легко можем вычислить нужную информацию (рис. 9.4).

- Вместе с каждым пустым блоком можно запоминать его размер и положение. Если будет выбрана очистка, то большинство пустых блоков станут свободным местом, управляемым с помощью списка свободных блоков.
- Вместе с каждым заполненным блоком запоминается его положение и смещение переноса (relocation offset). Если будет выбрано уплотнение, то достаточно будет перенести каждый заполненный блок на эту величину.

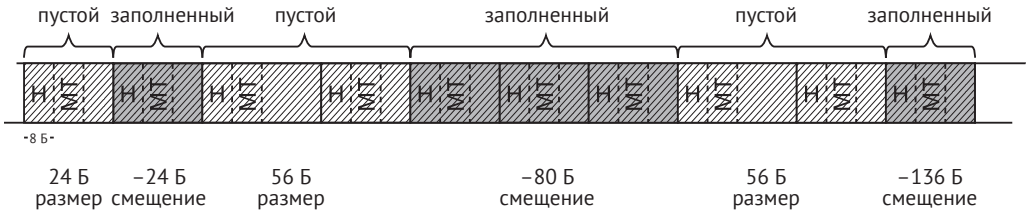


Рис. 9.4 ❖ Размеры и смещения, запоминаемые для заполненных и пустых блоков. Числа на рисунке вычислены в предположении, что длина каждого элемента (например, заголовка) равна 8 байтам

Как вычислить смещение переноса? В простейшем случае – как сумму размеров всех предшествующих пустых блоков (как на рис. 9.4). Но настоящая реализация гораздо сложнее. В ней используется собственный внутренний распределитель, который находит подходящий новый адрес для каждого заполненного блока, но не перемещает туда блок, а запоминает этот адрес.

> Если вас интересуют детали и вы готовы изучать код CoreCLR, то обратитесь к методу `gc_heap::plan_phase`. Этот метод последовательно просматривает объекты и находит все заполненные и пустые блоки. Для вычисления нового положения каждого заполненного блока вызывается один из методов – `allocate_in_condemned_generations` или `allocate_in_older_generations`. Из этой точки можете двигаться дальше.

В простом случае, когда заполненный блок можно перемещать, т. е. он не закреплен, распределитель методом сдвига указателя расположит заполненные

блоки рядом. На рис. 9.5 показано некое «виртуальное пространство» – представление управляемой кучи с точки зрения внутреннего распределителя (показывает, как куча выглядела бы после уплотнения). Эта иллюстрация приведена только для удобства – на самом деле распределитель оперирует указателями, обновляя их соответствующим образом. Этап планирования для нашего небольшого фрагмента кучи состоял бы из следующих шагов:

- сначала указатель распределения памяти устанавливается на начало поколения (рис. 9.5a);
- встретив первый заполненный блок (состоящий из одного объекта), распределитель определяет для него место там, где находится его указатель (рис. 9.5b), после чего сдвигает указатель. Разность между новым и старым положениями блока запоминается как его смещение переноса;
- встретив следующий заполненный блок (состоящий из трех объектов), распределитель определяет для него место сразу после предыдущего заполненного блока. И снова запоминает смещение переноса – разности между новым и старым положениями блока;
- при обработке последнего заполненного блока все повторяется.

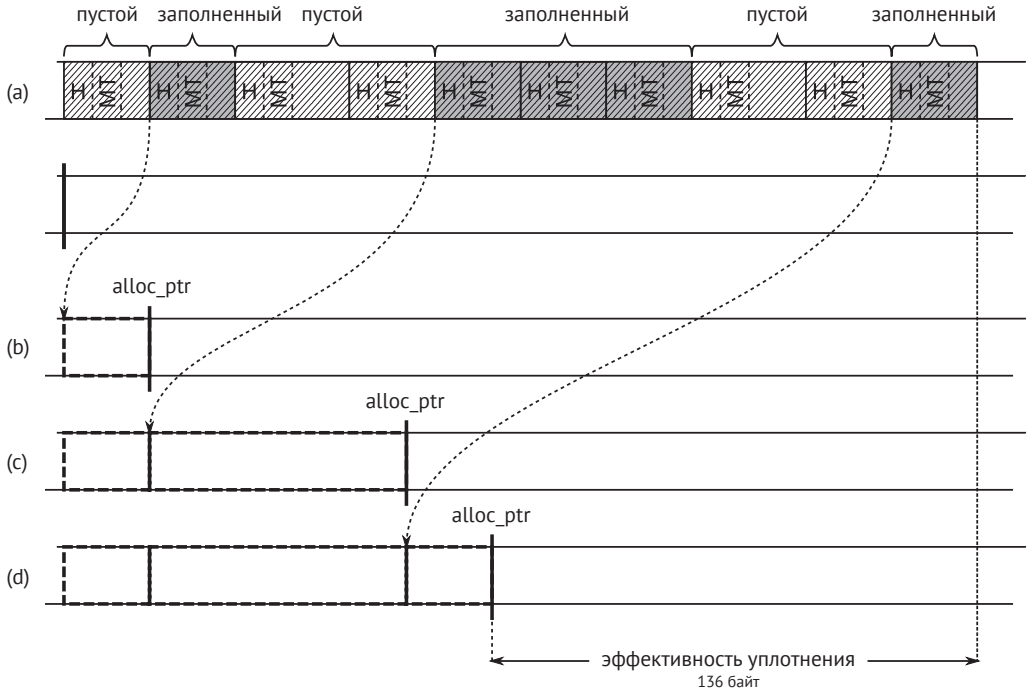


Рис. 9.5 ❖ Чтобы вычислить смещения переноса заполненных блоков, внутренний распределитель вычисляет новый адрес каждого такого блока: (a) размещение объектов, как на рис. 9.4, и взгляд распределителя на управляемую кучу; (b) внутренний распределитель нашел место для первого заполненного блока; (c) внутренний распределитель нашел место для второго заполненного блока; (d) внутренний распределитель нашел место для последнего заполненного блока

В результате вычислены все смещения переноса, так что GC точно знает, куда перемещать каждый заполненный блок, если возникнет нужда в уплотнении. И мы получили информацию об эффективности уплотнения, которой воспользуемся позже, когда будем принимать решение о стратегии сборки мусора.

> В примере на рис. 9.5 мы знаем, что после уплотнения место, занятое объектами, уменьшится на 136 байт, потому что это разность между текущим и будущим положениями указателя распределения памяти.

Из этого упрощенного примера не понятно, зачем может понадобиться более сложный распределитель. Но все прояснится, когда мы перейдем к обсуждению закрепленных объектов.

Подведем итоги. Благодаря группировке объектов в заполненные и пустые блоки можно весьма эффективно получить всю необходимую информацию:

- чему равна эффективность уплотнения;
- где нужно создавать элементы списка свободных блоков, если будет выбрана сборка очисткой;
- куда перемещать достижимые объекты, если будет выбрана сборка с уплотнением.

Возникает вопрос: где хранить данные о заполненных и пустых блоках? GC мог бы использовать для этой цели специальную область памяти, которой сам бы и управлял. Но если есть много небольших чередующихся заполненных и пустых блоков, то памяти потребовалось бы очень много. Кроме того, интенсивный доступ то к этой области, то к области, занятой управляемой кучей, был бы неэффективен с точки зрения использования кеша процессора. Но раз GC и так уже активно работает с управляемой кучей, почему бы не хранить в ней же информацию о блоках? Так и поступили разработчики Microsoft .NET.

Если правильно строить заполненные и пустые блоки, то каждому заполненному блоку будет предшествовать пустой[1]. Поэтому GC хранит интересную информацию только для заполненных блоков – прямо перед началом блока, в конце предшествующего пустого блока (рис. 9.6). Содержимое пустого блока можно затереть – все равно там находятся только недостижимые объекты, которые больше никому не нужны. Информация о заполненном блоке занимает ровно 24 байта (в 64-разрядной среде) или 12 байт (в 32-разрядной среде): размер соответствующего пустого блока, смещение переноса заполненного блока и некоторые дополнительные данные, о которых будет рассказано ниже (два бита, являющихся частью смещения переноса, и два дополнительных смещения – левого и правого).

Хранение информации о заполненном блоке в управляемой куче непосредственно перед блоком – главная причина того, что даже пустой объект должен занимать 24 байта (в 64-разрядной среде). Пустой блок перед заполненным содержит как минимум один объект, и он будет занимать не менее 24 байт. Таким образом, мы получаем гарантию, что места для информации о заполненном бло-

---

[1] Единственным исключением может быть первый заполненный блок, перед которым нет пустого, но в своих рассуждениях мы можем его опустить. И как мы скоро увидим, на самом деле в начале каждого поколения находится один пустой объект, так что даже первому заполненному блоку предшествует пустой.

ке хватит – просто и элегантно! Так хранится информация о каждой паре пустой–заполненный блоки в управляемой куче (рис. 9.7). Впоследствии эта информация будет использована во время очистки или уплотнения.

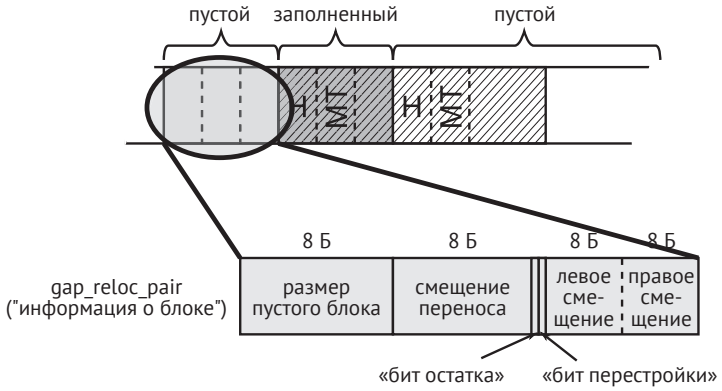


Рис. 9.6 ❖ Размещение информации о заполненном блоке в управляемой куче

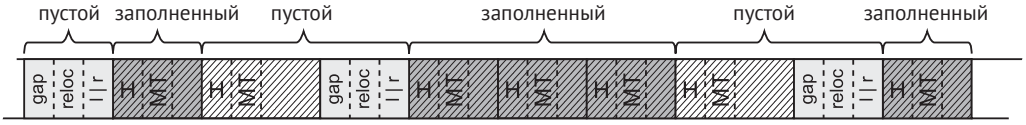


Рис. 9.7 ❖ Информация о размере и смещении, связанная с пустыми и заполненными блоками, хранится в самой управляемой куче (для ситуации на рис. 9.4)

Если GC решит произвести уплотнение, то информация о заполненных блоках будет использоваться очень часто. Заметим, что с ее помощью он может ответить на самый частый вопрос (возникающий при трансляции адресов): каким будет новый адрес объекта, размещенного по адресу X? Нам нужно только проверить, принадлежит ли адрес X какому-нибудь заполненному блоку, и если да, то вычесть из X смещение переноса этого блока. Поскольку данный вопрос задается очень, очень часто, то нужно сделать все возможное, чтобы ответ можно было получить эффективно. Поэтому заполненные блоки организованы в двоичное дерево поиска (binary search tree – BST).

Информация о заполненном блоке содержит смещение по отношению к левому и правому потомкам (в контексте BST) относительно начала данного блока (мы видели их на рис. 9.6) или 0, если соответствующего потомка нет.

Таким образом строится двоичное дерево, содержащее адреса всех заполненных блоков (рис. 9.8). Это дерево сбалансировано, так что для каждого узла все левые потомки находятся по младшим адресам, а все правые – по старшим.

> Адрес в дереве заполненных блоков указывает на первый объект блока (на его поле MT, как принято в CLR). GC знает, где найти информацию о блоке по постоянному смещению, связанному с ним.

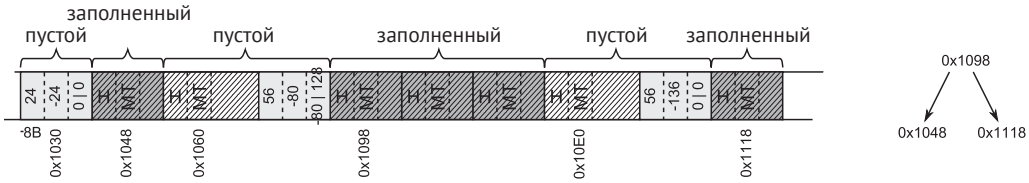


Рис. 9.8 ❖ Заполненные блоки организованы в двоичное дерево поиска

Сценарий 9.1. Дамп памяти с поврежденными структурами

Описание. Во время исследования какой-то проблемы был создан полный дамп памяти приложения .NET. Однако его, похоже, нельзя использовать, потому что структуры данных повреждены. При вызове большинства команд SOS появляется следующее сообщение:

```
> !dumpheap -stat

The garbage collector data structures are not in a valid state for traversal.
It is either in the "plan phase," where objects are being moved around, or
we are at the initialization or shutdown of the gc heap. Commands related to
displaying, finding or traversing objects as well as gc heap segments may not
work properly. !dumpheap and !verifyheap may incorrectly complain of heap
consistency errors.1
```

Анализ. Дамп памяти действительно мог быть создан в момент, когда GC выполнял этап планирования, а в этом случае нет гарантии, что объекты находятся в «нормальном состоянии», потому что кучу невозможно обойти обычным способом (т. е. начать с начала сегмента и продвигать указатель на размер объекта, как было сказано в начале главы). Заглянув в код CoreCLR, мы увидим код защиты этапа планирования:

```
GCScan::GcRuntimeStructuresValid (FALSE);
plan_phase (n);
GCScan::GcRuntimeStructuresValid (TRUE);
```

Это единственное место, где встречается такая защита. Поэтому мы легко можем проверить, действительно ли дамп памяти создан в такой неподходящий момент, – достаточно поискать потоки, исполняющие относящийся к GC код. В зависимости от среды существует четыре возможные комбинации библиотеки и пространства имен:

- `coreclr!wks` – .NET Core, GC в режиме рабочей станции;
- `coreclr!srv` – .NET Core, GC в серверном режиме;
- `clr!wks` – .NET Framework, GC в режиме рабочей станции;
- `clr!srv` – .NET Framework, GC в серверном режиме.

---

[1] Состояние структур данных сборщика мусора непригодно для обхода. Он либо находился на «этапе планирования», когда объекты перемещались, либо на этапе инициализации или уничтожения управляемой кучи. Команды отображения, поиска и обхода объектов или сегментов кучи GC могут работать некорректно. Команды `!dumpheap` и `!verifyheap` могут давать неверную информацию об ошибках несогласованности кучи. – *Прим. перев.*

Например, если есть дамп приложения для .NET Core с GC, работающим в режиме рабочей станции, то поиск производится следующим образом:

```
> !findstack coreclr!wks
Thread 000, 6 frame(s) match
    * 00 000000a963b7cd30 00007ff903bb0b48 CoreCLR!WKS::gc_heap::plan_phase+0xa9
    * 01 000000a963b7ce40 00007ff903bb095a CoreCLR!WKS::gc_heap::gc1+0x178
    * 02 000000a963b7ceb0 00007ff903b90d21 CoreCLR!WKS::gc_heap::garbage_collect+0x5ca
    * 03 000000a963b7cf20 00007ff903b90e98 CoreCLR!WKS::GCHeap::GarbageCollectGeneration+0x191
    * 04 000000a963b7cf60 00007ff903b90b15 CoreCLR!WKS::GCHeap::GarbageCollectTry+0xe8
    * 05 000000a963b7cff0 00007ff903670613 CoreCLR!WKS::GCHeap::GarbageCollect+0x2a5
```

Очевидно, что мы действительно попали на этап планирования, потому что есть исполняющий его поток.

Однако мой опыт показывает, что это сообщение отображается также в случае невозможности получить данные GC, потому что загружена неправильная версия SOS (например, для среды выполнения .NET 2.0 вместо .NET 4.0 или наоборот).

Таблица кирпичей

Корень дерева заполненных блоков нужно где-то сохранить. Создавать одно гигантское дерево заполненных блоков для всей управляемой кучи было бы непрактично. Когда в процессе просмотра пустых и заполненных блоков в дерево добавляется новый узел, может возникнуть необходимость перебалансировки дерева. Для огромного дерева, включающего все заполненные блоки, такая операция обошлась бы очень дорого. Да и обходить такое дерево для поиска нужного блока было бы накладно, потому что пришлось бы перемещаться по многим уровням.

Гораздо практичнее было бы построить несколько деревьев заполненных блоков для диапазонов соседних адресов. В CLR такой диапазон называется *кирпичом* (brick). Размер кирпича равен 2048 Б для 32-разрядной и 4096 Б для 64-разрядной среды. Иными словами, каждые 2 или 4 КБ управляемой кучи представлены одним кирпичом, который содержит информацию о его дереве заполненных блоков. Кирпичи хранятся в таблице кирпичей, которая покрывает всю управляемую кучу (рис. 9.9). Каждый элемент таблицы кирпичей – 16-разрядное целое число, которое может принимать три логически различных значения:

- 0 – с кирпичом не связано никакой информации о заполненных блоках (в соответствующем диапазоне адресов нет заполненных блоков);
- >0 – представляет смещение корня дерева заполненных блоков (это значение больше настоящего на 1, поскольку 0 означал бы отсутствие информации) от начала соответствующей области памяти;
- <0 – означает, что этот кирпич является продолжением предыдущих (есть большой заполненный блок, охватывающий несколько кирпичей), и мы должны вернуться на указанное количество кирпичей к начальному.

Путем объединения записей таблицы кирпичей с левым и правым смещением в информации о блоке мы получим эффективное представление дерева (рис. 9.10). В примере запись таблицы кирпичей содержит значение 0x6f1 – оно представляет смещение корня дерева заполненных блоков от начала соответствующей области памяти. Поскольку это вторая запись в таблице кирпичей, она представляет диапазон адресов 0x1000–0x2000. Следовательно, корень дерева находится по адресу

0x6f0 (положительные значения нужно уменьшить на 1, как отмечено выше) плюс 0x1000, что дает адрес 0x16f0 в управляемой куче. Начиная с этого адреса мы можем получить доступ ко всему дереву заполненных блоков, пользуясь смещениями, хранящимися в информации о блоке.

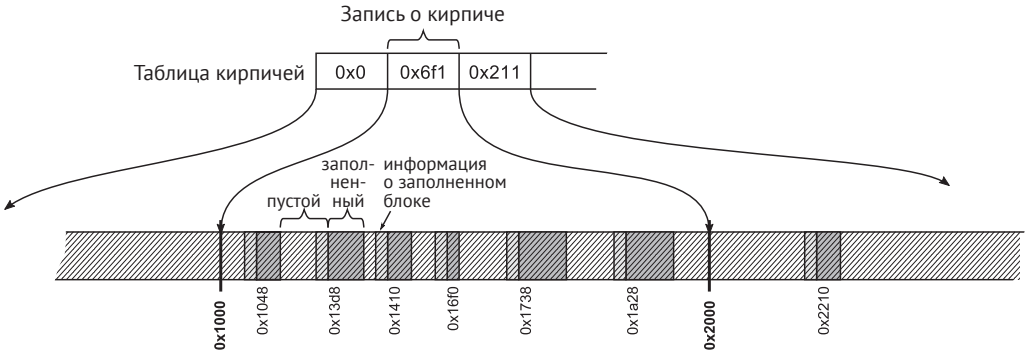


Рис. 9.9 ❖ Кирпичи и таблица кирпичей

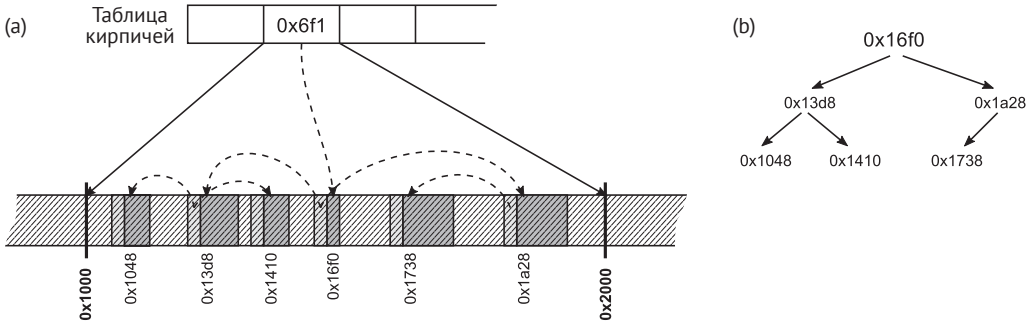


Рис. 9.10 ❖ Пример таблицы кирпичей:
(a) запись о кирпиче, содержащая смещение корня дерева заполненных блоков, и информация о блоках в дочерних узлах дерева; (b) логическое представление дерева заполненных блоков

Число записей в таблице кирпичей, а также левое и правое смещения – короткие (16-разрядные) целые, т. е. позволяют хранить значения от –32 767 до 32 767, но этого достаточно для представления смещений в диапазонах адресов шириной не более 4 КБ.

Чтобы ответить на вопрос «каким будет новый адрес объекта по адресу X?», нужно выполнить следующие простые действия:

- вычислить номер записи в таблице кирпичей, зная адрес X, – для этого достаточно разделить X на размер кирпича;
- если значение в этой записи меньше 0, то перейти к нужной записи таблицы кирпичей и повторить;
- если значение в этой записи больше 0, то начать обход дерева заполненных блоков для поиска нужного блока;
- получить смещение переноса из блока и вычесть его из X.

На этом мы могли бы закончить описание работы GC на этапе планирования. Вся необходимая информация собрана, и GC может двигаться дальше. Эффективность уплотнения можно получить из смещения переноса последнего заполненного блока. Однако в этой головоломке остался один очень важный кусочек, который существенно усложняет все дело.

Закрепление

Если объект закреплен, то, скорее всего, потому, что мы хотим передать его адрес неуправляемому коду (рис. 9.11).

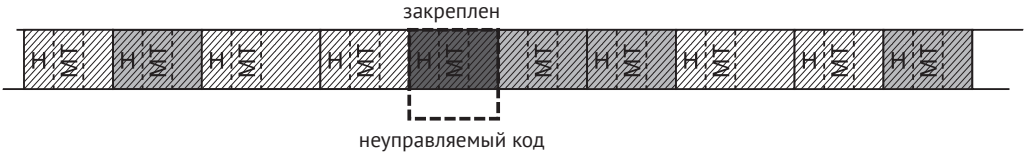


Рис. 9.11 ❖ Пример закрепления.
Закрепленные объекты изображены темно-серым цветом

Мы не можем переместить закрепленный объект во время уплотнения, потому что неуправляемый код об этом никогда не узнал бы. Он будет обращаться по прежнему адресу, где теперь находятся совсем другие данные (рис. 9.12).



Рис. 9.12 ❖ Пример закрепления.
Неуправляемый код обращается к неопределенным данным,
после того как закрепленный объект перемещен

Закрепление довольно сильно осложняет описанный в предыдущем разделе простой алгоритм. Закрепленные объекты следует специально обрабатывать во внутреннем распределителе и при построении дерева заполненных блоков. В этом разделе объясняется, как это реализовано.

Из-за закрепления существует не две, а три группы объектов:

- заполненный блок – представляет группу помеченных (достижимых) объектов;
- закрепленный заполненный блок – представляет группу закрепленных (и, следовательно, помеченных) объектов;
- пустой блок – представляет группу непомеченных (недостижимых) объектов.

Сначала рассмотрим самый простой случай – закрепленный заполненный блок находится после пустого (рис. 9.13а). Тогда мало что изменяется. Мы можем сохранить информацию о заполненном блоке как обычно, т. е. в конце соответ-

ствующего пустого. А при построении дерева заполненных блоков мы сохраняем левое и правое смещения. Основное отличие заключается в том, что для такого блока смещение переноса равно нулю.

Дополнительно для каждого закрепленного заполненного блока хранится размер свободного места перед ним (на случай, если будет выбрана стратегия уплотнения) (рис. 9.13b).

Теперь во время уплотнения обычные заполненные блоки будут перемещаться, а закрепленные – нет (рис. 9.13c), потому что описанный ранее внутренний распределитель не перемещает закрепленные заполненные блоки (он «выделяет» место для них точно там, где они уже находятся).

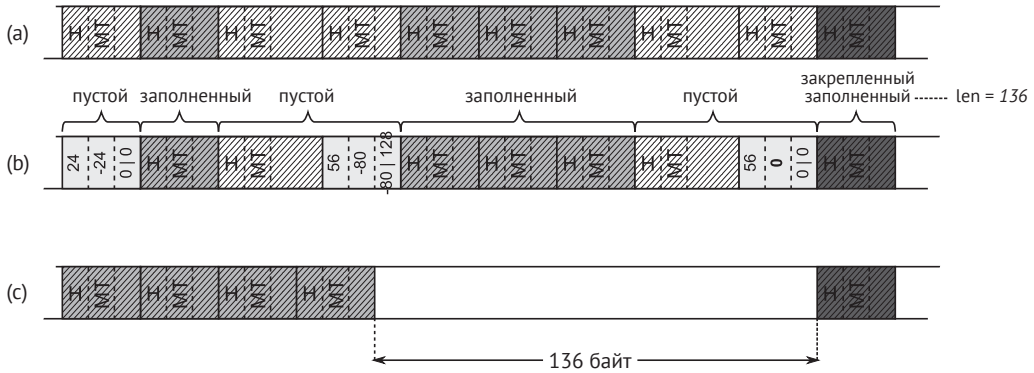


Рис. 9.13 ❖ Случай, когда закрепленный заполненный блок находится после пустого: (a) пример с одним закрепленным блоком; (b) организация информации о блоке; (c) результат уплотнения

Заметим, что в случае, изображенном на рис. 9.13c, может образоваться большой участок свободного места между обычным и закрепленным заполненными блоками. Эту ситуацию мы обсудим ниже в разделе «Оставление», чтобы не перегружать читателя деталями.

Данные обо всех закрепленных заполненных блоках запоминаются также в *очереди закрепленных заполненных блоков*. Как мы скоро увидим, у GC часто возникает необходимость хранить о закрепленном заполненном блоке такую информацию, которая не помещается в стандартную структуру для обычного блока, отсюда и необходимость в отдельной очереди.

Интересно, что для хранения данных о закрепленных заполненных блоках используется уже известный нам массив `mark_stack_array`. Но теперь в нем хранятся указатели на экземпляры специального класса пометки, а не адреса объектов. Поэтому упоминания `mark_stack_array` (а также соответствующие указатели `mark_stack_tos` и `mark_stack_bos`) часто встречаются не только в коде CoreCLR, относящемся к стеку помеченных объектов, но и в коде обработки закрепленных заполненных блоков.

Рассмотрим более сложный сценарий – закрепленный заполненный блок расположен сразу после обычного (рис. 9.14a). Теперь у нас проблема – мы хотели бы сохранить информацию о блоке прямо перед его началом, как обычно, но там находится достижимый объект! GC мог бы сделать исключение и сохранить ин-

формацию о закрепленном блоке где-то в другом месте, но ... на самом деле GC перезаписывает предшествующий ему объект (рис. 9.14b). Это допустимо, потому что во время этапа планирования все потоки гарантированно приостановлены, поэтому не может быть такого, что какой-то код попытается обратиться к таким «испорченным» объектам до того, как мы их «восстановим».

Обрезанный конец этого объекта (на 64-разрядной платформе его размер составляет 24 байта – длина трех указателей) сохраняется вместе с другими данными в новой записи очереди закрепленных заполненных блоков. Этот конец объекта называется *предблоком* (pre plug), потому что он предшествует закрепленному блоку. Позже GC воспользуется им во время очистки или уплотнения.

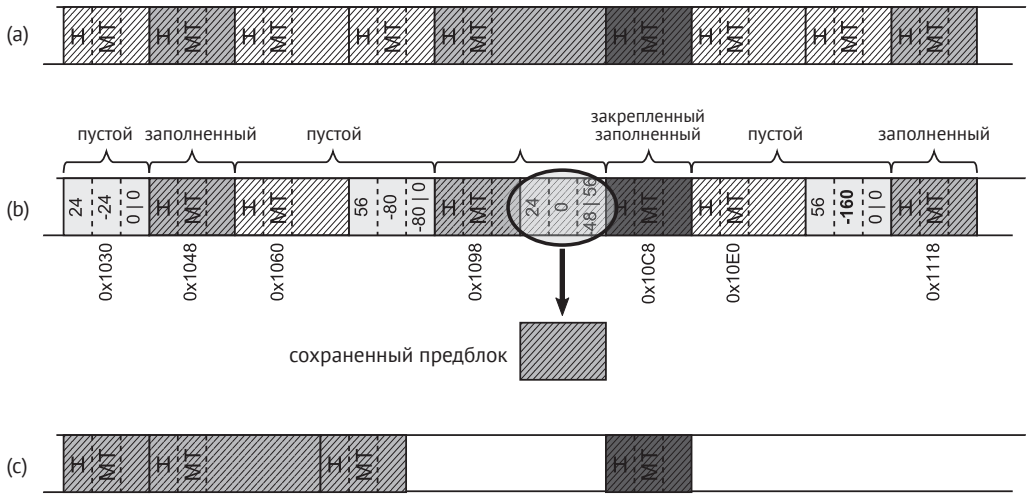


Рис. 9.14 ❖ Случай, когда закрепленный заполненный блок находится после обычного:
(a) пример с одним закрепленным блоком после обычного;
(b) организация информации о блоке, когда конец объекта хранится как предблок;
(c) результат уплотнения

> Снова заметим, как нам помогает тот факт, что размер объекта должен быть не менее 24 байтов, – гарантируется, что для информации о блоке хватит места, даже если ему предшествует минимально возможный объект

При таком подходе закрепленные заполненные блоки можно обрабатывать как обычные. Смещение переноса будет равно 0, размер пустого блока искусственно установлен равным 24 байтам[1], и такая информация о блоке естественным образом включается в дерево заполненных блоков (рис. 9.15).

Однако на этом приключения, связанные с закреплением объектов, не заканчиваются. Рассмотрим случай, когда закрепленный блок расположен прямо перед обычным (рис. 9.16a). Тогда возникает еще одна проблема – информацию об обычном заполненном блоке надо бы сохранить точно перед ним, там, где кончается закрепленный объект. Но к закрепленным объектам могут обращаться неуправ-

[1] Хотя никакого пустого блока на самом деле нет, GC необходимо учитывать его для статистики.

ляемые потоки, которые не приостанавливаются даже на время сборки мусора (рис. 9.16b). Поэтому закрепленные объекты никогда нельзя модифицировать. Решение простое – вместо создания отдельного закрепленного блока объект, находящийся сразу после закрепленного блока, включается в него же (рис. 9.16c). И запись о закрепленном заполненном блоке соответственно модифицируется. В следующем разделе мы увидим, как эта информация используется при уплотнении.

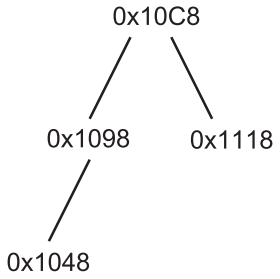


Рис. 9.15 ❖ Логическое представление дерева заполненных блоков для блоков на рис. 9.14

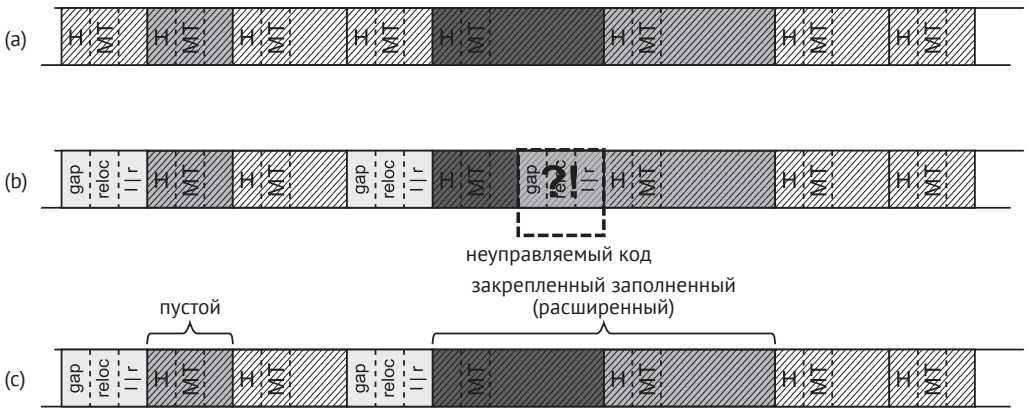


Рис. 9.16 ❖ Случай, когда закрепленный заполненный блок находится перед обычным: (a) пример с одним закрепленным блоком; (b) информация о блоке, которую нужно организовать надлежащим образом; (c) организация информации о блоке

Это своего рода компромисс. Начиная с этого момента закрепленные и обычные объекты обрабатываются как расширенный закрепленный блок со всеми связанными с закреплением недостатками. Закрепления следует всячески избегать, но здесь мы сделали прямо противоположное – взяли и закрепили вполне нормальный объект. Однако преимущества такого общего подхода к заполненным блокам все же весомее недостатков. А если обычный объект, находящийся после закрепленного, невелик, то минусами можно и вовсе пренебречь.

Но проблема может встать во весь рост, если после закрепленного объекта находится большой блок помеченных объектов. Включать ли их все в расширенный закрепленный блок (теоретически увеличив размер закрепленных объектов на

несколько мегабайтов или гигабайтов)? Нет, конечно. Закрепленный блок расширяется только за счет первого объекта.

Рассмотрим случай, когда после закрепленного объекта расположено два помеченных (рис. 9.17a). Закрепленный заполненный блок расширяется, как описано выше. Это позволяет создать обычный заполненный блок из следующих объектов, поскольку обычный объект можно безопасно перезаписать (рис. 9.17b). Понятно, что конец такого «поврежденного» объекта нужно где-то сохранить, как и в случае предблока. Этот конец объекта называется *постблоком* (post plug). Впоследствии он будет использован во время очистки или уплотнения.

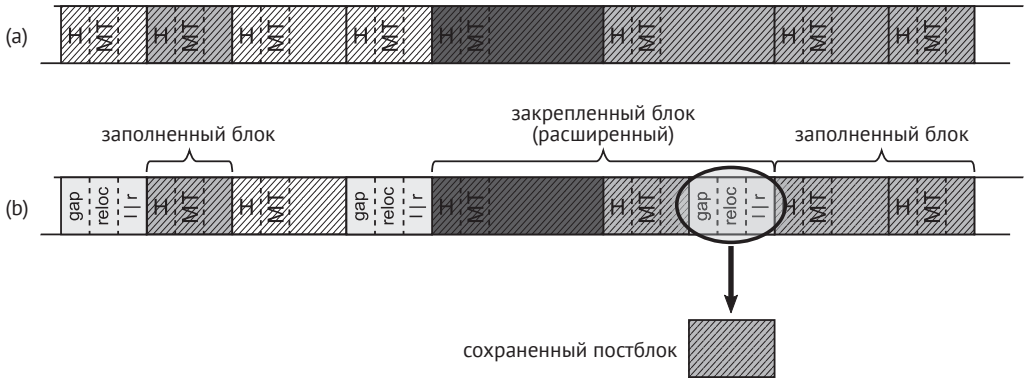


Рис. 9.17 ❖ Случай, когда закрепленный блок находится перед по меньшей мере двумя помеченными объектами: (a) пример размещения объекта в случае одного закрепленного блока; (b) организация информации о блоке

Подведем итоги. Самая типичная ситуация – когда закрепленный объект находится внутри блока помеченных объектов (рис. 9.18a). В таком случае необходимо сохранить предблок и постблок и создать три отдельных заполненных блока (включая один закрепленный и расширенный) (рис. 9.18b).

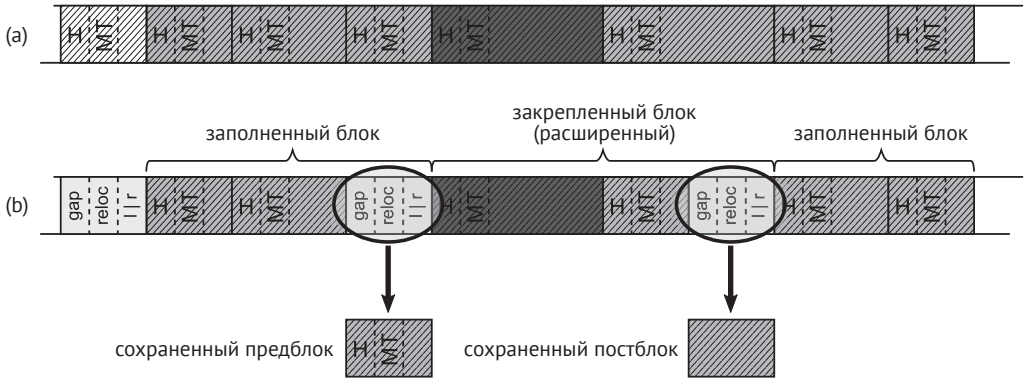


Рис. 9.18 ❖ Случай, когда закрепленный блок находится внутри блока помеченных объектов: (a) пример размещения объекта в случае одного закрепленного блока; (b) организация информации о заполненном блоке

Отсюда вытекает несколько следствий:

- копирование пред- и постблоков увеличивает количество операций в памяти – чем больше закрепленных объектов, тем больше неудобств;
- закрепленный блок может быть расширен на один объект, поэтому закрепляется больше памяти, чем необходимо, а если этот добавленный объект оказывается большим, мы замораживаем большую область памяти, что не дает уменьшить фрагментацию;
- на этапе планирования некоторые объекты в управляемой куче «повреждаются», что не дает возможности обойти их обычным способом. С этой проблемой можно столкнуться во время анализа дампа памяти (см. сценарий 9.1).

Сценарий 9.2. Исследование закрепления

Описание. Счетчик `\.NET CLR Memory\# of Pinned Objects` показывает, что в нашем приложении, работающем в производственной среде, много закрепленных объектов (рис. 9.19). Мы хотим исследовать, является эта ситуация естественной или нет.

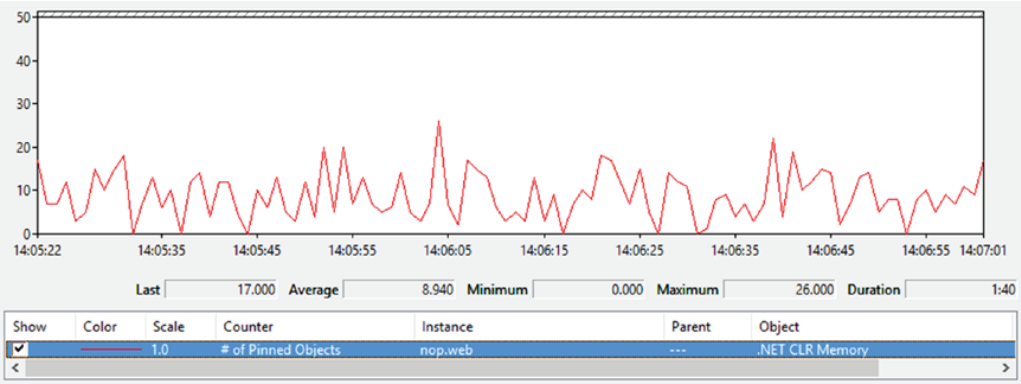


Рис. 9.19 ❖ Счетчик `\.NET CLR Memory\# of Pinned Objects`

Анализ. Выше мы узнали о том, что есть два источника закрепления:

- локальные закрепленные переменные – закрепленные объекты, на которые ссылаются локальные переменные, часто создаются неявно из-за использования ключевого слова `fixed`. Их жизнь ограничена временем жизни содержащего их метода. Поэтому дамп памяти или снимок кучи (Heap Snapshot), созданный в PerfView, покажет только небольшую их часть, объявленную в методах, выполняющихся в текущий момент. Но для каждого такого объекта генерируется событие ETW `PinObjectAtGCTime`;
- закрепленные описатели – объекты, закрепленные явно с помощью ссылки из закрепленного описателя. К таковым относятся некоторые внутренние объекты, удерживаемые самой CLR, а также созданные явно методом `GCHandle.Allocate`. Таблица описателей находится в памяти на протяжении всего времени работы приложения, поэтому ее легко проанализировать, имея

дамп или снимок кучи. В сеансах ETW такая информация тоже запоминается в событии `PinObjectAtGCTime`, но только для поколений, в которых производится сборка мусора (поскольку таблица описателей знает о поколениях).

Счетчик производительности `\.NET CLR Memory\# of Pinned Objects` также учитывает оба типа. В начале работы мы не знаем, какой тип закрепления дает больший вклад.

Анализ можно начать с записи сеансов ETW в периоды, когда количество закрепленных объектов велико. В PerfView достаточно задать режим .NET (без GC Collect Only). Открыв отчет GCStats в группе Memory Group, мы, вероятно, увидим, что количество закрепленных объектов действительно большое (рис. 9.20). В последнем столбце, Pinned Obj, показано, сколько закрепленных объектов GC переместил в следующее поколение. Эта величина должна совпадать со значением счетчика. Если счетчики производительности недоступны (в случае среды .NET Core), то диагностировать заметное количество закрепленных объектов можно, начиная отсюда.

Очевидно, что в нашем случае закрепленные объекты происходят в основном из локальных закрепленных переменных, регистрируемых в событии `PinObjectAtGCTime`.

All GC Events for Process 20700: nop.web

GC Events by Time

All times are in msec. Hover over columns for help.

GC Index	Pause Start	Trigger Reason	Gen	Suspend Msec	Pause MSec	% Pause Time	% GC	Gen0 Alloc MB	Gen0 Alloc Rate MB/sec	Peak MB	After MB	Ratio Peak/After	Promoted MB	Gen0 MB	Gen0 Survival Rate %	Gen0 Frag %	Gen1 MB	Gen1 Survival Rate %	Gen1 Frag %	Gen2 MB	Gen2 Survival Rate %	Gen2 Frag %	LOH MB	LOH Survival Rate %	LOH Frag %	Finalizable Surv MB	Pinned Obj
1428 Beginning entries truncated, use View in Excel to view all...																											
2326	30,355.591	AllocSmall	0N	0.009	5.089	65.7	51.6	4.082	1,405.13	125.733	125.733	1.00	0.741	5.914	17	99.90	1.819	NaN	0.00	117.470	NaN	0.00	5.530	NaN	0.00	0.00	11
2327	30,367.613	AllocSmall	0N	0.011	10.160	59.4	20.0	4.105	591.57	126.024	125.727	1.00	1.187	2.705	28	99.95	2.222	NaN	0.44	117.470	NaN	0.00	3.330	NaN	0.00	0.00	11
2328	30,387.252	AllocSmall	1N	0.091	9.470	50.0	11.9	6.216	655.31	129.315	128.512	1.01	1.427	7.234	7	99.99	0.477	43	0.00	117.470	NaN	53.97	3.330	NaN	0.13	0.03	8
2329	30,401.142	AllocSmall	1N	0.080	7.653	63.4	17.4	4.138	934.79	128.512	125.504	1.02	0.728	4.403	7	99.99	0.301	89	0.00	117.470	NaN	53.60	3.330	NaN	0.13	0.01	5
2330	30,414.400	AllocSmall	0N	0.055	5.680	50.3	12.8	4.141	737.74	125.504	125.504	1.00	0.213	4.403	5	95.11	0.301	NaN	0.00	117.470	NaN	0.00	3.330	NaN	0.00	0.00	6
2331	30,426.190	AllocSmall	0N	0.057	5.462	47.2	13.3	4.149	678.37	125.511	125.504	1.00	0.494	3.906	11	99.99	0.798	NaN	0.02	117.470	NaN	0.00	3.330	NaN	0.00	0.00	6
2332	30,506.055	AllocSmall	0N	0.009	8.202	9.9	4.1	6.226	83.68	127.891	125.021	1.02	0.698	2.721	11	99.88	1.499	NaN	0.00	117.470	NaN	0.00	3.330	NaN	0.00	0.03	10
2333	30,522.738	AllocSmall	1N	0.010	6.844	44.6	15.2	6.274	739.04	128.594	128.230	1.00	0.904	7.595	0	99.98	0.035	57	0.00	117.470	NaN	52.86	3.330	NaN	0.13	0.00	12
2334	30,534.114	AllocSmall	0N	0.056	5.325	54.0	14.7	4.087	900.15	128.230	124.829	1.03	0.390	3.603	9	99.97	0.426	NaN	0.00	117.470	NaN	0.00	3.330	NaN	0.00	0.00	6

Рис. 9.20 ❖ Столбец Pinned Obj в таблице GC Events by Time

Как уже было сказано, событие `PinObjectAtGCTime` генерируется для каждого закрепленного объекта на этапе пометки. Мы можем просто посмотреть на сами эти события в представлении Events – особенно интересно поле `TypeName` (рис. 9.21). Иногда достаточно взглянуть на него, чтобы идентифицировать источник закрепления – если закрепленный тип достаточно уникален.

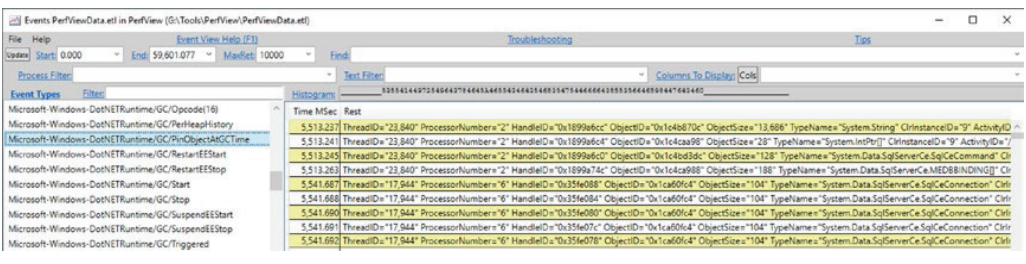
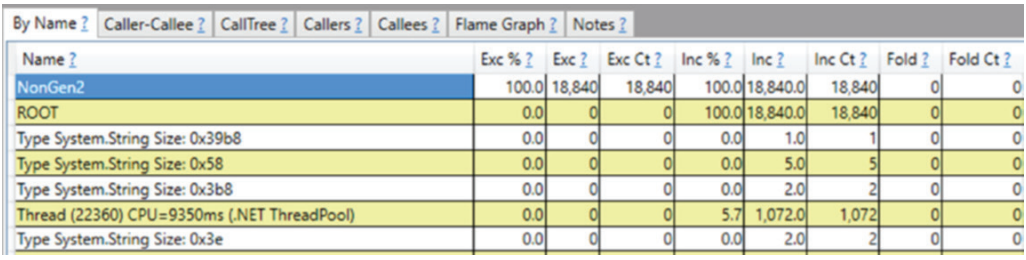


Рис. 9.21 ❖ ETW Microsoft-Windows-DotNETRuntime/GC/PinObjectAtGCTime

> Заметим, что к событию `PinObjectAtGCTime` не подключены стеки вызова. Мы могли бы добавить их, установив параметр `@StacksEnabled=true` для поставщика .NET ETW, но это ничего бы не дало. Стек вызовов для таких событий всегда указывает на методы в коде GC, а не на то место, где используется закрепленный объект.

Но существует более хороший способ проанализировать этот источник закрепления – специально предназначенное для данной цели представление Pinning At GC Time Stacks в группе Advanced. Здесь произведен дополнительный анализ и группировка для получения сводных данных. На вкладке по умолчанию, By Name, показаны основные источники закрепленных объектов по типам (рис. 9.22). Мы видим, что все закрепленные объекты попали в группу NonGen2.

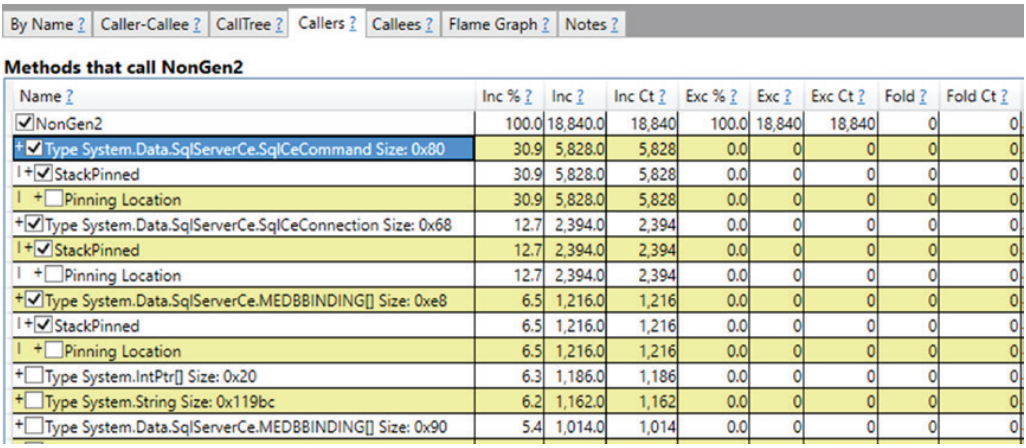


By Name | Caller-Callee | CallTree | Callers | Callees | Flame Graph | Notes

Name	Exc %	Exc	Exc Ct	Inc %	Inc	Inc Ct	Fold	Fold Ct
NonGen2	100.0	18,840	18,840	100.0	18,840.0	18,840	0	0
ROOT	0.0	0	0	100.0	18,840.0	18,840	0	0
Type System.String Size: 0x39b8	0.0	0	0	0.0	1.0	1	0	0
Type System.String Size: 0x58	0.0	0	0	0.0	5.0	5	0	0
Type System.String Size: 0x3b8	0.0	0	0	0.0	2.0	2	0	0
Thread (22360) CPU=9350ms (.NET ThreadPool)	0.0	0	0	5.7	1,072.0	1,072	0	0
Type System.String Size: 0x3e	0.0	0	0	0.0	2.0	2	0	0

Рис. 9.22 ❖ Представление Pinning At GC Time Stacks – вкладка By Name

Выбрав из ее контекстного меню команду Goto Item in Callers, мы сможем более подробно проанализировать, какие типы являются основными источниками закрепления. Можно заметить, что они в основном относятся к виду «StackPinned» (рис. 9.23). Видно, что в нашем примере наибольший вклад дают типы из пространства имен `System.Data.SqlServerCe` (точнее, `SqlCeCommand`, `SqlCeConnection` и массив `MEDBBINDING[]`).



By Name | Caller-Callee | CallTree | Callers | Callees | Flame Graph | Notes

Methods that call NonGen2

Name	Inc %	Inc	Inc Ct	Exc %	Exc	Exc Ct	Fold	Fold Ct
NonGen2	100.0	18,840.0	18,840	100.0	18,840	18,840	0	0
+ Type System.Data.SqlServerCe.SqlCeCommand Size: 0x80	30.9	5,828.0	5,828	0.0	0	0	0	0
+ StackPinned	30.9	5,828.0	5,828	0.0	0	0	0	0
+ Pinning Location	30.9	5,828.0	5,828	0.0	0	0	0	0
+ Type System.Data.SqlServerCe.SqlCeConnection Size: 0x68	12.7	2,394.0	2,394	0.0	0	0	0	0
+ StackPinned	12.7	2,394.0	2,394	0.0	0	0	0	0
+ Pinning Location	12.7	2,394.0	2,394	0.0	0	0	0	0
+ Type System.Data.SqlServerCe.MEDBBINDING[] Size: 0xe8	6.5	1,216.0	1,216	0.0	0	0	0	0
+ StackPinned	6.5	1,216.0	1,216	0.0	0	0	0	0
+ Pinning Location	6.5	1,216.0	1,216	0.0	0	0	0	0
+ Type System.IntPtr[] Size: 0x20	6.3	1,186.0	1,186	0.0	0	0	0	0
+ Type System.String Size: 0x119bc	6.2	1,162.0	1,162	0.0	0	0	0	0
+ Type System.Data.SqlServerCe.MEDBBINDING[] Size: 0x90	5.4	1,014.0	1,014	0.0	0	0	0	0

Рис. 9.23 ❖ Представление Pinning At GC Time Stacks – вызывающие стороны NonGen2

Теперь достаточно поискать в исходном коде, где эти типы используются с ключевым словом fixed, чтобы однозначно идентифицировать источник такого закрепления. Например, метод System.Data.SqlServerCe.SqlCeCommand.ExecuteCommandText содержит показанный в листинге 9.1 код, в котором у поля DbBinding тип MEDBBINDING[].

Листинг 9.1 ❖ Пример локальной закрепленной переменной из библиотеки System.Data.SqlServerCe.dll (декомпилированной с помощью dnSpy)

```
fixed (IntPtr* ptr = this.accessor.DbBinding)
{
   // ...
}
```

Существует еще один способ проанализировать объекты, закрепленные описателями, а именно команда SOS !GCHandles в WinDbg. Создадим дамп памяти в момент, когда счетчик \.NET CLR Memory\# of Pinned Objects принимает большое значение. Открыв его в WinDbg и загрузив расширение SOS, мы сможем получить список всех закрепленных описателей с помощью команды !GCHandles (листинг 9.2). Мы увидим список объектов – закрепленных, потому что закреплен описатель, и среди них внутренние массивы CLR (вспомните пул интернированных строк и массив статических объектов), различные буферы, используемые сервером Kestrel, и т. д. В настоящее время не существует расширения WinDbg, которое позволило бы перечислить источники закрепления, находящиеся в стеке.

Листинг 9.2 ❖ Получение списка всех закрепленных описателей командой !GCHandles

```
> !GCHandles -type Pinned
  Handle Type          Object       Size         Data Type
007f1374 Pinned        04988078   131084         System.Byte[]
007f1378 Pinned        04968058   131084         System.Byte[]
007f137c Pinned        04948038   131084         System.Byte[]
007f1398 Pinned        0490f058    32780         System.Object[]
007f13ac Pinned        04928018   131084         System.Byte[]
007f13b4 Pinned        0490b038    16396         System.Object[]
007f13b8 Pinned        048fb028    65532         System.Object[]
007f13bc Pinned        048f9008     8204         System.Object[]
007f13c0 Pinned        0403dbac       12         Bid+BindingCookie
007f13c4 Pinned        048f7fe8     4108         System.Object[]
007f13c8 Pinned        04918008    65532         System.Object[]
007f13cc Pinned        048e7fd8    65532         System.Object[]
007f13d0 Pinned        048e3ff8    16332         System.Object[]
007f13d4 Pinned        048e1ff8     8172         System.Object[]
007f13d8 Pinned        048e17d8     2060         System.Object[]
007f13dc Pinned        048d18b8    65292         System.Object[]
007f13e0 Pinned        048c9918    32652         System.Object[]
007f13e4 Pinned        048c94f8     1036         System.Object[]
007f13e8 Pinned        048c5518    16332         System.Object[]
007f13ec Pinned        048c3518     8172         System.Object[]
007f13f0 Pinned        048c2508     4092         System.Object[]
```

```
007f13f4 Pinned        048c22e8      524        System.Object[]
007f13f8 Pinned        038c121c       12        System.Object
007f13fc Pinned        048c1020      788        System.Object[]

Statistics:
      MT    Count      TotalSize Class Name
720dff90        1             12 System.Object
57fbb464        1             12 Bid+BindingCookie
720dffe4       18         417536 System.Object[]
720e419c        4         524336 System.Byte[]
Total 24 objects
```

Вывод прост: чтобы составить картину закрепления, следует анализировать события ETW `PinObjectAtGCTime`, которые учитывают оба источника закрепления. Имейте в виду, что расширение SOS умеет выводить только источники закрепления, связанные с описателями.

Напоследок отметим, что способность PerfView анализировать свои снимки кучи здесь может оказаться чуть более полезной. Открыв такой снимок, мы можем найти строку [.NET Roots] и выбрать из ее контекстного меню команду Goto Item in CallTree. Отказавшись от свертывания (очистив поле Fold%), мы сможем перечислить все типы корней, включая закрепленные локальные переменные (рис. 9.24). И увидим, что основным источником такого закрепления является уже знакомый нам тип `MEDBBINDING[]`. Помните, что у нас по-прежнему есть лишь один снимок памяти (snapshot), поэтому полного списка источников закрепления, находящихся в стеке, ожидать не стоит.

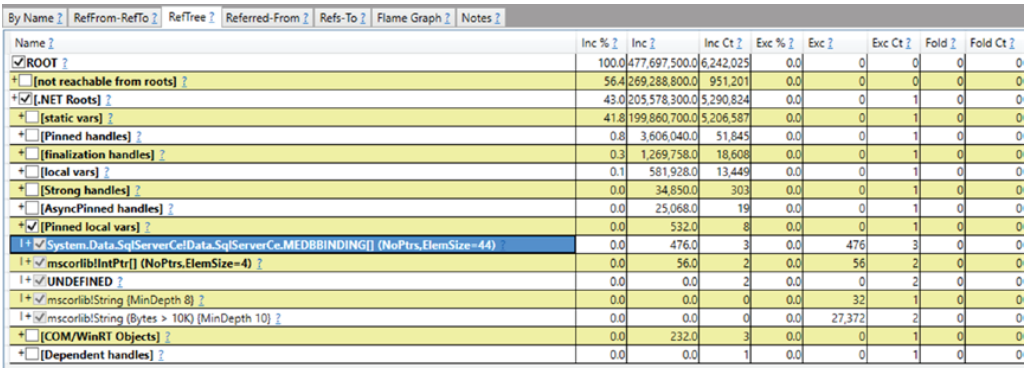


Рис. 9.24 ❖ Строка [.NET Roots] на вкладке **RefTree** в процессе анализа снимка кучи в PerfView

> Иногда также полезно отказаться от группировки в поле **GroupPats** и от свертывания в поле **FoldPats**. Получатся более подробные и более наглядные результаты. Рисунок 9.24 был подготовлен именно так.

Определив источники закрепления, мы можем решить, можно их избежать или нет. Если они не приводят к сильной фрагментации, то, наверное, и делать ничего не надо. Если же проблема серьезна, то надо поискать какое-то решение. Способы избежать избыточного закрепления описаны в главе 13.

Границы поколений

После очистки или уплотнения границы поколений должны быть изменены. В случае когда закрепленных объектов нет, это достаточно просто. Границы выравниваются так, чтобы поколение включало все переведенные в него объекты.

Например, пусть объекты расположены, как показано на рис. 9.25a, и принято решение произвести полную сборку мусора. У нас есть три поколения, и в каждом из них некоторые объекты помечены (достижимы). Как мы знаем, на этапе планирования внутренний распределитель вычисляет новые адреса заполненных блоков (рис. 9.25b). Но вместе с тем вычисляются новые границы поколений. Все это делается виртуально, без перемещения объектов, таким образом, на рис. 9.25b показан взгляд внутреннего распределителя на управляемую кучу как на некую абстракцию.

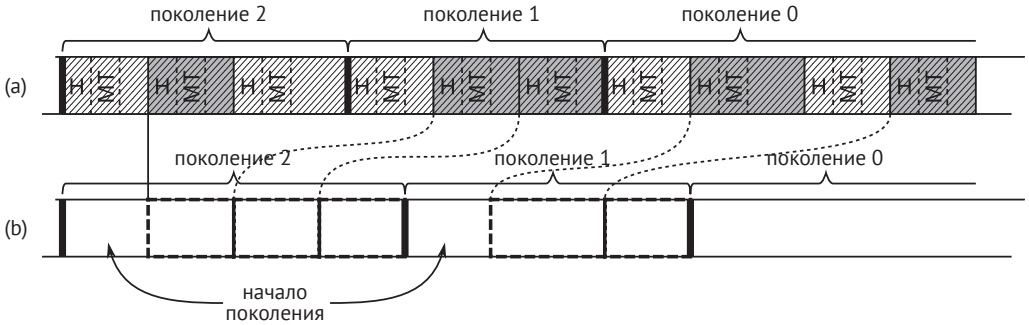


Рис. 9.25 ❖ Вычисление границ поколений: (a) расположение объектов; (b) итоговый взгляд распределителя на управляемую кучу (светло-серым цветом изображены мертвые объекты, умеренно-серым – живые объекты, перемещенные согласно пунктирным линиям)

Новые границы поколений установлены так, что каждое поколение содержит все выжившие объекты. Их легко вычислить на этапе планирования. Но нужно сделать одно замечание. В начале каждого поколения (даже пустого) есть свободное место, достаточное для размещения минимального объекта. Оно необходимо для размещения информации о первом заполненном блоке в поколении, поскольку позволяет обрабатывать их единообразно, не задумываясь о заполненных блоках, пересекающих границы поколений.

Далее мы часто будем опускать это начало поколения, чтобы не загромождать рисунки. Но не удивляйтесь, когда в процессе анализа дампов памяти обнаружите пустое место длиной 24 байта в начале каждого поколения.

Оставление

Выше на рис. 9.13 и 9.14 были показаны возможные результаты уплотнения. Но было не вполне ясно, как внутренний распределитель должен обходить закрепленные заполненные блоки и где должно находиться начало поколения. С точки зрения реализации, самое простое решение – сбрасывать накопленное смещение переноса после каждого закрепленного блока так, чтобы следующий заполнен-

ный блок размещался сразу после него. Тогда начало поколения расположится так, что все выжившие объекты будут охвачены.

Но очевидно, что это решение крайне неэффективно с точки зрения фрагментации, потому что иногда приводит к образованию больших областей свободной памяти. Поэтому внутренний распределитель пытается заполнить все промежутки между закрепленными блоками обычными заполненными блоками и началами поколений (рис. 9.26). Этап планирования для нашего небольшого фрагмента кучи будет состоять из следующих шагов:

- при первом выделении указатель устанавливается на начало поколения (рис. 9.26a);
- распределитель находит место для первого (рис. 9.26b) и второго (рис. 9.26c) заполненных блоков;
- распределитель «выделяет» место для закрепленного блока там, где он сейчас находится (рис. 9.26d);
- распределитель находит место для последнего заполненного блока перед закрепленным – места для него хватает (рис. 9.26e).

Теперь нужно решить, где должны начинаться поколения. В начале нашего примера все объекты принадлежали поколению 0. Если бы мы хотели перевести все выжившие объекты, включая закрепленные, в поколение 1, как и ожидается, то поколение 0 должно было бы начаться сразу после закрепленного заполненного блока – закрепленный объект из поколения 0 следовало бы перевести в поколение 1, как и все остальные объекты. Но тогда в поколении 1 образовалась бы сильная фрагментация. Лучше поступить по-другому – повторно использовать существующую пустую область памяти и закончить поколение 1 раньше. Планировщик поместит начало поколения 0 перед закрепленным объектом (рис. 9.26f)!

Вследствие такого решения закрепленный объект остается в поколении 0, а не переводится в поколение 1, как обычно! В нашем примере так будет со всеми закрепленными блоками, расположенными после нашего закрепленного блока (если таковые имеются и если больше не существует незакрепленных заполненных блоков).

Такая ситуация называется *оставлением* (demotion) и означает, что объект оказывается не в том поколении, в котором должен был бы оказаться. Оставление может означать как то, что объект не был переведен, так и то, что он вернулся в более молодое поколение.

Таким образом, вследствие закрепления возможны все три ситуации с переводом объекта. Проанализируем их с точки зрения закрепленного блока (расширенного на один объект после него) из поколения 1. Для такого закрепленного блока возможны три случая:

- перед ним находится достаточно большой пустой блок, который может вместить обычные блоки и начала поколений 1 и 0, – в таком случае закрепленный блок возвращается из поколения 1 назад в поколение 0 (рис. 9.27);
- перед ним находится достаточно большой пустой блок, который может вместить обычные блоки и начало поколения 1, – в таком случае закрепленный блок остается в поколении 1 (рис. 9.28);
- перед ним недостаточно места для размещения обычных блоков, поэтому и закрепленный, и обычный блоки (вместе с большим промежутком между ними) должны быть переведены в старшее поколение (рис. 9.29).

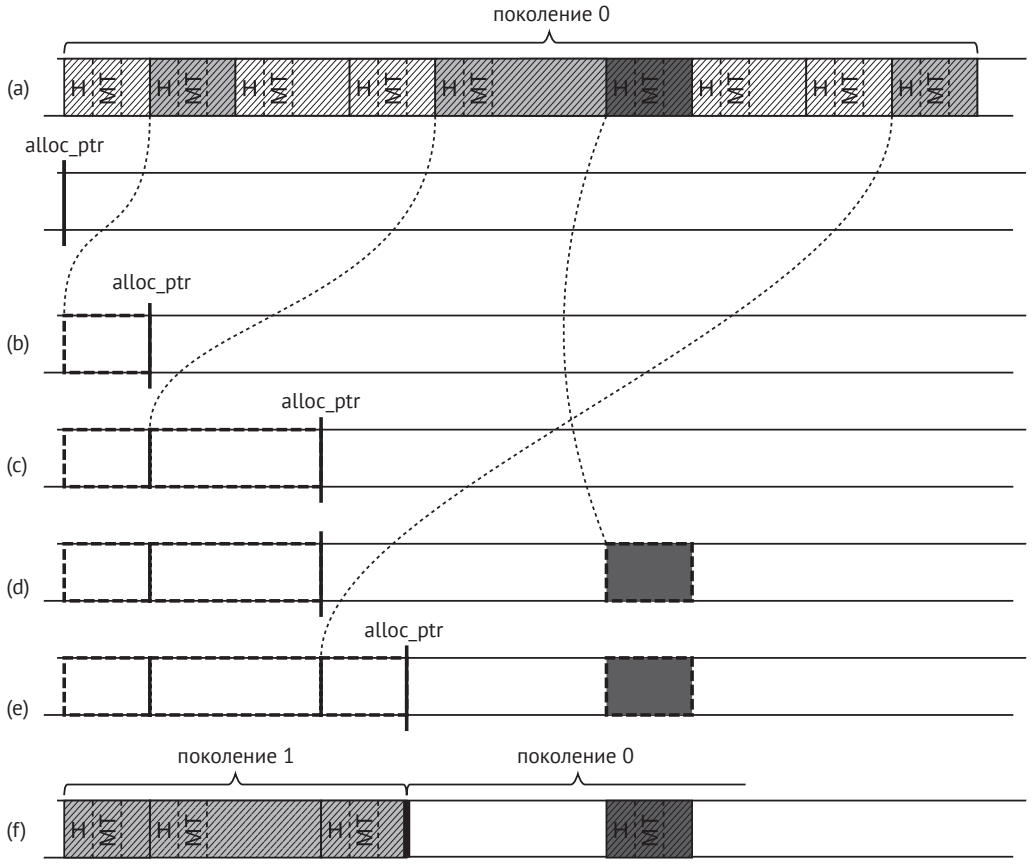


Рис. 9.26 ❖ Внутренний распределитель заполняет пустые пространства, оставшиеся вследствие закрепления: (a) размещение объекта, как на рис. 9.14, и итоговый взгляд распределителя на управляемую кучу; (b) внутренний распределитель нашел место для первого заполненного блока; (c) внутренний распределитель нашел место для второго заполненного блока; (d) закрепленный блок не перемещается; (e) внутренний распределитель нашел место для последнего заполненного блока перед закрепленным (места хватает); (f) поколение 1 начинается перед закрепленным блоком, который теоретически должен быть переведен, но остается в поколении 0

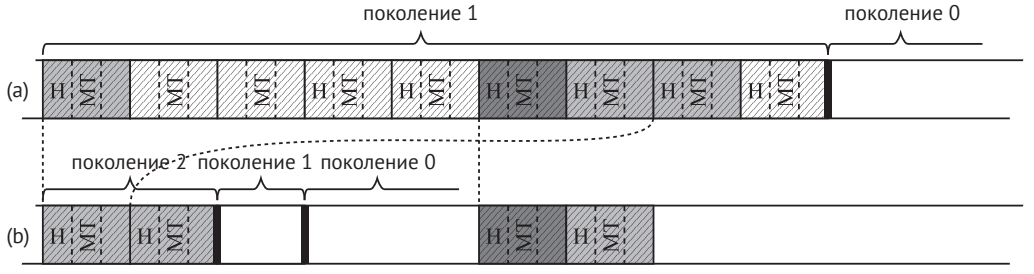


Рис. 9.27 ❖ Возврат из поколения 1 в поколение 0:
(a) размещение объектов; (b) результат уплотнения

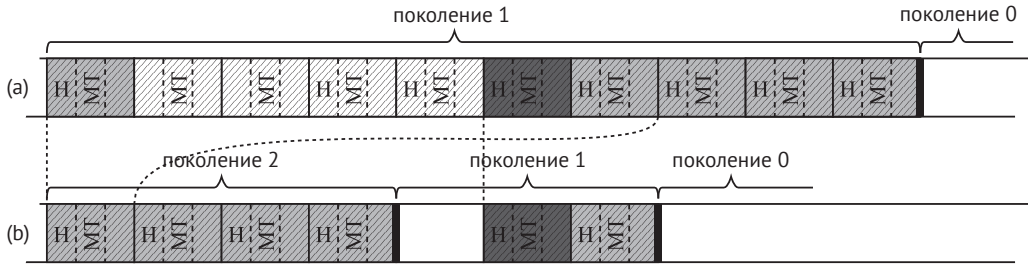


Рис. 9.28 ❖ Оставление в поколении 1:
(a) размещение объектов; (b) результат уплотнения

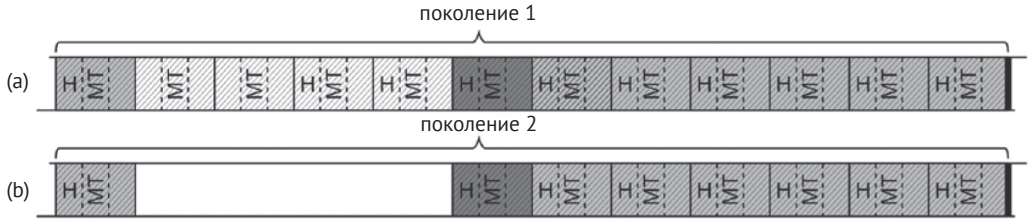


Рис. 9.29 ❖ Нормальный перевод из поколения 1 в поколение 2:
(a) размещение объектов;
(b) результат уплотнения (образуется нежелательная фрагментация)

Внутренний распределитель оперирует заполненными блоками, а не отдельными объектами. Это означает, что даже если бы перед закрепленным объектом на рис. 9.29 было бы достаточно места для нескольких объектов из обычного заполненного блока, он все равно не был бы разделен на блоки меньшего размера, чтобы заполнить пустой блок. Это компромисс между сложностью внутреннего распределителя и создаваемой им фрагментацией. Впрочем, в общем случае этими потерями можно пренебречь. Типичный закрепленный объект живет либо очень недолго, либо, наоборот, долго.

- В первом случае он умирает в поколении 0, которое достаточно мало и динамично, чтобы обработать эти потери без образования фрагментации.
- Во втором случае закрепленный объект оказывается в поколении 2, так что закрепление большую часть времени не имеет значения (пока в поколении 2 не будет произведено уплотнение, сборщику мусора безразлично, закреплен объект или нет).

> **ПРИМЕЧАНИЕ** Заметим, что в текущей реализации оставлять можно только закрепленные блоки (т. е. закрепленный объект, возможно, дополненный одним следующим за ним незакрепленным объектом).

Очевидно, когда закрепленных блоков несколько, может случиться, что оставлены будут не все. Это зависит от текущего расположения заполненных и пустых блоков. Это показано на рис. 9.30. Заполненные блоки размещены в пустых максимально эффективно. В результате первый закрепленный блок переведен, как положено, а второй возвращен из поколения 1 в поколение 0.

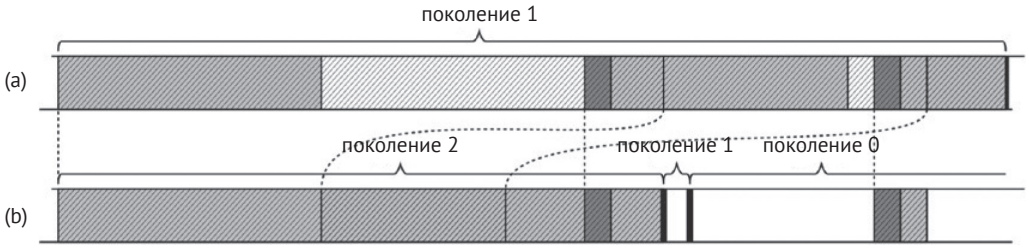


Рис. 9.30 ❖ Пример, когда имеет место и перевод, и возвращение

Оставление – это оптимизация, направленная на то, чтобы занять как можно больше свободного места в пустых блоках. Оставшееся свободное место преобразуется в список свободных блоков, если они достаточно велики, так что у них тоже остается шанс на повторное использование.

Вероятно, именно поэтому нет никаких диагностических данных об оставлении. Его можно увидеть, внимательно изучив дамп памяти, но вряд ли вам это когда-нибудь понадобится. Вас должен больше интересовать уровень фрагментации вследствие закрепленных заполненных блоков. Но механизм оставления – важная часть внутреннего распределителя и этапа планирования, поэтому без его упоминания их описание было бы неполным. Нужно понимать, что закрепленные объекты могут быть как переведены, так и оставлены (или даже возвращены в младшее поколение). Дизайн сборки мусора на основе поколений не налагает в этом отношении никаких ограничений.

> Когда эфемерный сегмент строится путем повторного использования существующего сегмента, содержащего только поколение 2, находящиеся там закрепленные объекты возвращаются из поколения 2 в поколения 1 и 0.

В расширении SOS для WinDBg существует недокументированная команда `!DumpGCData`. Помимо данных, которые можно получить другими способами (например, от ETW) – причины уплотнения, количество различных видов сборки мусора и т. п., – она выводит информацию, которая больше нигде недоступна, а именно «интересные сведения»:

```
Interesting data points
        pre short: 0
       post short: 0
      merged pins: 0
   converted pins: 0
          pre pin: 0
         post pin: 0
 pre and post pin: 0
 pre short padded: 0
post short padded: 0
```

Сюда входят:

- различные типы «pre and post pin» – закрепленные пред- и постблоки с предшествующей и последующей информацией об этих типах;
- различные типы «pre pin» – закрепленные блоки только с предшествующей информацией о данных типах;

- различные типы «post pin» – закрепленные блоки только с последующей информацией об этих типах;
- «converted pin» – объекты, которые были преобразованы в закрепленные из-за расширения закрепленного блока.

Понятно, что эта команда больше полезна разработчикам GC, а обычным пользователям от нее толку мало. Не гарантируется даже, что эта команда сохранится в следующих версиях расширения SOS. Если вам интересно узнать больше, посмотрите метод `gc_heap::record_interesting_data_point` в исходном коде CoreCLR.

Куча больших объектов

Вообще-то этап планирования в LOH почти никогда не нужен, потому что в ней в основном производится сборка очисткой. Однако LOH должна быть организована так, чтобы можно было произвести уплотнение, если мы явно попросим об этом.

Заполненные и пустые блоки

Этап планирования в куче больших объектов нужен только для уплотнения. По умолчанию производится очистка, а для нее ни заполненные, ни пустые блоки не нужны (как будет ясно дальше). Для LOH запрашивать уплотнение нужно явно, по умолчанию оно никогда не производится. Это означает, что в подавляющем большинстве .NET-приложений LOH вообще не уплотняется. Однако же куча больших объектов должна быть к этому готова. Поэтому понятия заполненных и пустых блоков в ней присутствуют, хотя и в упрощенной форме.

Отличительной чертой LOH является тот факт, что в ней находятся только большие объекты. Поэтому возможны некоторые упрощения.

- Нет острой необходимости группировать объекты в блоки, потому что отдельные объекты и так достаточно велики. Поэтому, чтобы упростить этап планирования в LOH, каждый достижимый объект рассматривается как отдельный заполненный блок. Во-первых, этого достаточно для обеспечения приемлемой эффективности трансляции адресов (плотность объектов в LOH гораздо ниже, чем в SOH). Во-вторых, это позволяет избежать фрагментации (было бы намного труднее эффективно перемещать гигантские блоки, состоящие из многих больших объектов).
- Чтобы избежать трудностей с обработкой информации о заполненных блоках (в т. ч. пред- и постблоках вокруг закрепленных блоков), объекты размещаются в LOH с небольшими промежутками (рис. 9.31). В текущей реализации промежуток занимает 4 слова размером с указатель (32 байта на 64-разрядной платформе) и отформатирован как обычный свободный объект.

> Описанные здесь промежутки между объектами в LOH используются во всех текущих вариантах среды выполнения .NET, позволяющих явное уплотнение LOH. Однако среду выполнения .NET можно скомпилировать и без поддержки этого механизма, так что выделение памяти в LOH будет производиться в режиме «без промежутков (without padding)». Поскольку такая среда не поддерживает уплотнения LOH, отпадает нужда в этапе планирования и создании заполненных блоков (хранении информации о них).

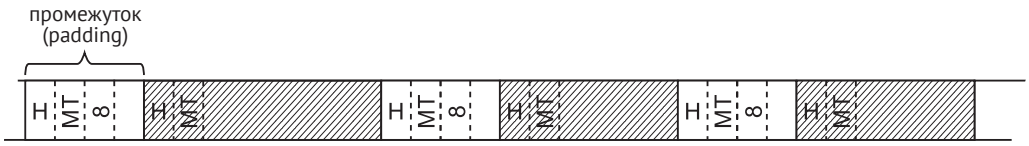


Рис. 9.31 ❖ Размещение объектов в куче больших объектов.
Показаны промежутки между объектами на случай, если будет запрошено уплотнение

На этапе пометки каждый объект может быть идентифицирован либо как помеченный, либо как помеченный и закрепленный. Из каждого такого объекта создается заполненный блок (рис. 9.32).

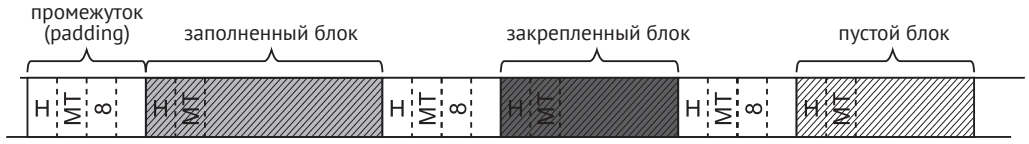


Рис. 9.32 ❖ Размещение объектов в куче больших объектов после этапа пометки

Перед каждым заполненным блоком необходимо сохранить информацию о нем, но благодаря наличию промежутков для этого всегда достаточно места (рис. 9.33). Эта информация состоит только из смещения переноса блока.

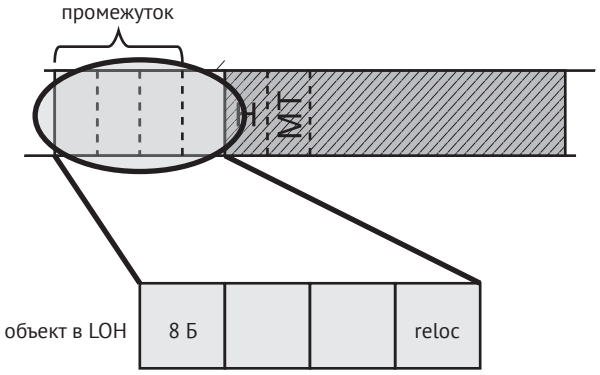


Рис. 9.33 ❖ Информация о блоке,
хранящаяся в куче больших объектов (в предшествующем промежутке)

Смещение переноса вычисляется так же, как в случае кучи малых объектов. Внутренний распределитель находит подходящее место, просматривая заполненные блоки (объекты) один за другим. Как уже было сказано, именно поэтому удобно рассматривать каждый объект как отдельный заполненный блок, а не группировать их в более крупные блоки. Скорее всего, у распределителя возникли бы серьезные проблемы при попытке найти место для таких огромных блоков между закрепленными.

Поскольку информация о заполненном блоке не может перезаписать другой объект в LOH, нет необходимости поддерживать пред- и постблоки.

Поскольку количество объектов относительно мало, а их размеры велики, нет нужды строить дерево заполненных блоков в LOH. Для ответа на вопрос «каким будет новый адрес объекта, находящегося по адресу X?» нужно выполнить одно простое действие: получить смещение переноса из информации о блоке X и вычесть его из X. Поэтому не нужны ни кирпичи, ни таблица кирпичей.

Поскольку внутри LOH нет поколений, отпадает необходимость вычислять их границы. Понятия оставления также не существует.

С учетом всего вышесказанного этап планирования в LOH гораздо проще, чем в SOH. Перед каждым блоком, обычным или закрепленным, сохраняется информация о нем (рис. 9.34). Кроме того, для каждого закрепленного блока хранится размер свободного места перед ним (в соответствующей записи очереди закрепленных блоков) – на случай уплотнения.

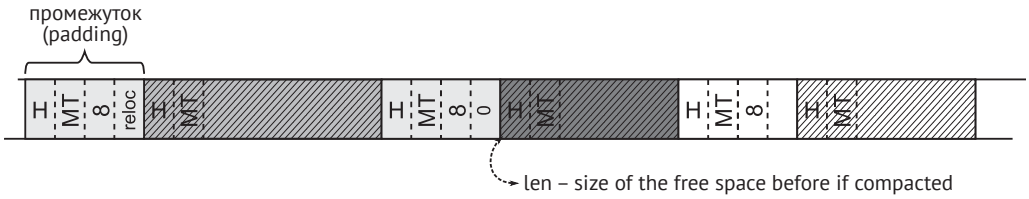


Рис. 9.34 ❖ Результат этапа планирования в куче больших объектов (в последнем промежутке не хранится смещение, потому что он предшествует пустому блоку)

Попутно отметим, что закрепление в LOH, как и в SOH, чревато фрагментацией.

В исходном коде CoreCLR код этапа планирования в куче больших объектов находится в методе `gc_heap::plan_loh`.

Принятие решения об уплотнении

Выполнив сложные вычисления на этапе планирования, GC должен решить, стоит ли производить уплотнение. Иногда в силу ряда объективных причин это может оказаться необходимым. Но в большинстве случаев решение основано на уровне фрагментации.

Приведем список причин, по которым GC может принять решение об уплотнении.

- Это последняя полная сборка перед возбуждением исключения `OutOfMemoryException` – GC должен сделать все возможное, чтобы освободить память.
- Уплотнение запрошено явно, например с помощью параметра при вызове `GC.Collect`.
- Кончилось место в эфемерном сегменте – как было сказано в разделе о выборе поколения, GC агрессивно пытается освободить память, прежде чем расширять существующий или создавать новый эфемерный сегмент.
- Высокий уровень фрагментации поколения – если в каком-то поколении уровень фрагментации высок, то в нем сборка мусора с уплотнением должна быть продуктивной, т. е. можно освободить большой объем памяти.

- Велика потребность системы в физической памяти – если потенциальный объем памяти, возвращенной системе в результате уплотнения, превышает некоторый порог, то GC выбирает уплотнение.

В некоторых из перечисленных выше решений превышение порога фрагментации играет важную роль. Может возникнуть вопрос, чему равен этот порог. В каждом поколении он свой и состоит из двух значений, взятых из статических данных поколения (см. табл. 7.1 и 7.2).

- Полная фрагментация – располагая информацией, собранной на этапе планирования, легко вычислить фрагментацию конкретного поколения. Достаточно рассмотреть запланированные адреса последних выделенных объектов в отдельных сегментах и свободное место, которое будет создано из-за наличия закрепленных объектов. Это значение представлено в столбце «Предельная фрагментация» в табл. 7.1 и 7.2 (см. сводку в табл. 9.1).
- Коэффициент фрагментации – отношение полной фрагментации к размеру всего поколения, в котором производится сборка. Это значение представлено в столбце «Предельная доля фрагментации» в табл. 7.1 и 7.2 (см. сводку в табл. 9.1).

Таблица 9.1. Пороги фрагментации для разных поколений

	Полная фрагментация	**Коэффициент фрагментации**
Gen0	40000	50 %
Gen1	80000	50 %
Gen2	200000	25 %

Например, поколение 2 будет считаться сильно фрагментированным, если полная фрагментация превышает 200 000 байт, и это больше 25 % всего размера поколения.

> В исходном коде CoreCLR код принятия решения об уплотнении находится в методе `gc_heap::decide_on_compacting`.

Резюме

Описанному в этой главе этапу планирования часто не уделяют должного внимания в простых описаниях сборки мусора, сводя ее к этапам «пометка–очистка–уплотнение». Но надеюсь, что, прочитав эту главу, вы понимаете, насколько важен этот этап. Он готовит все необходимые данные, которые последующие этапы просто используют надлежащим образом.

Лично меня удивляет изобретательное сочетание заполненных и пустых блоков и таблиц кирпичей, благодаря которому удается вычислить результаты очистки и уплотнения, не выполняя их. Эта часть очень плохо документирована в материалах, относящихся к GC. И хотя полученные в этой главе знания трудно применить на практике (разве что вы станете лучше понимать, почему закрепление затрудняет реализацию GC), я полагаю, что любознательный читатель найдет эти сведения очень интересными.

Мы почти подошли к концу описания GC. В следующей главе описывается последний этап – уплотнение и очистка.

Глава 10

Сборка мусора – очистка и уплотнение

Это последняя и самая короткая глава о деталях сборки мусора. Хотя в ней описываются такие важные этапы, как очистка или уплотнение, мы уже знаем, как много было сделано на предыдущих этапах. После того как на этапе планирования (см. предыдущую главу) было принято решение о стратегии, GC осталось выполнить один из описанных ниже шагов.

Но имейте в виду, что хотя большая часть вычислений уже выполнена раньше, с точки зрения накладных расходов этапы очистки и уплотнения остаются самыми ресурсоемкими – дороже всего обходится доступ к памяти во время модификации и (или) перемещения заполненных блоков. Поэтому в плане реализации эти этапы проще предыдущих, но для производительности они самые важные!

Также помните, что самая типичная комбинация GC – уплотнение SOH и очистка LOH, при этом очистка LOH производится перед уплотнением SOH.

Этап очистки

Если GC решает не уплотнять (или если это не было запрошено явно в случае LOH), то начинается этап очистки. Как описано в главе 1, сборка очисткой не представляет сложностей. Все недостижимые объекты нужно преобразовать в свободное пространство. Мы уже знаем, что в терминологии сборки мусора в .NET это означает, что все или некоторые пустые блоки нужно добавить в список свободных блоков.

Как уже не раз отмечалось выше и как вы, наверное, сами понимаете, на этапах очистки и уплотнения просто используется информация, собранная на этапе планирования. Читателю, который просматривает книгу по диагонали, может показаться странным, что очистке и уплотнению – которые столь часто упоминаются в литературе по GC – в этой книге уделено так мало места. Но это потому, что все трудные вычисления уже были выполнены на этапе планирования!

Куча малых объектов

Очистка кучи малых объектов состоит из следующих шагов (рис. 10.1):

- создать элементы списка свободных блоков из пустых – из каждого пустого блока, большего, чем два минимальных объекта, создается новый элемент

и вставляется в список свободных блоков (описанный в главе 6). Более узкие пустые блоки считаются непригодным для использования свободным пространством (но учитываются в статистике фрагментации);

- восстановить сохраненные пред- и постблоки – все «поврежденные» объекты восстанавливаются;
- произвести дополнительную сверку, чтобы обновить очередь финализации (отразить новые границы поколений) и состарить (или омолодить) выжившие описатели соответствующего типа;
- реорганизовать сегменты, например удалить ставшие ненужными (или сохранить их в списке повторно используемых в случае придерживания виртуальной памяти).

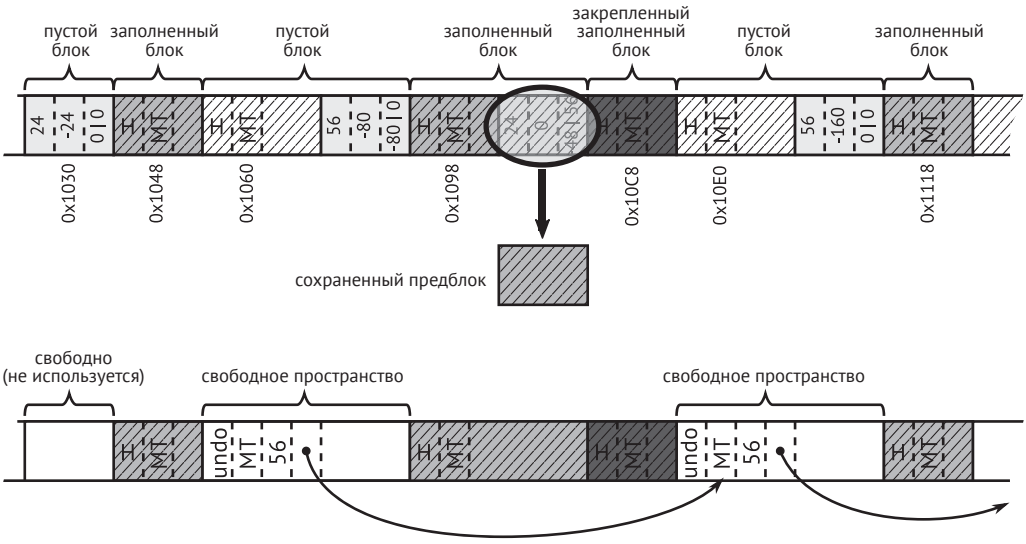


Рис. 10.1 ❖ Пример результатов очистки кучи малых объектов (на основе информации, собранной на этапе планирования)

Если вы хотите самостоятельно изучить код очистки SOH в CoreCLR, то начните с метода `gc_heap::plan_phase`. В части `else` условного оператора, проверяющего переменную `should_compact`, вызываются два наиболее важных метода: `gc_heap::make_free_lists` создает список свободных блоков из пустых блоков, а `gc_heap::recover_saved_pinned_info` восстанавливает объекты, частично затертые пред- и постблоками.

Куча больших объектов

Для очистки кучи больших объектов этап планирования вообще не задействуется. Объекты просматриваются один за другим (как на этапе планирования для SOH), и пустое пространство между помеченными объектами добавляется в список свободных блоков. Кроме того, удаляются ставшие ненужными сегменты LOH (если только не включен режим придерживания виртуальной памяти, а тогда она запоминается в списке сегментов для повторного использования).

Такая реализация очистки LOH проста и эффективна. Но у нее есть один недостаток – фрагментация. Обычно проблем с этим не возникает. Распределение размеров создаваемых объектов, скорее всего, вполне естественное – есть несколько типичных размеров и их вариации. В таком случае у повторного использования элементов из списка свободных блоков должны быть хорошие статистические характеристики. Но если это не так, то следует рассмотреть явное уплотнение LOH.

Этап уплотнения

Если GC решает произвести уплотнение (или ему поступил явный запрос), то начинается этап уплотнения. Как уже отмечалось, на нем используется информация, собранная на этапе планирования. В общем случае уплотнение состоит из двух основных шагов: перемещение (копирование) объектов и обновление всех ссылок на перемещенные объекты, где бы они ни встретились. Поэтому этап уплотнения гораздо сложнее этапа очистки. Ниже приводится подробное описание уплотнения для куч малых и больших объектов, хотя в принципе они похожи.

Куча малых объектов

Уплотнение кучи малых объектов должно быть максимально эффективным. Обычно есть много чередующихся пустых и заполненных блоков, занимающих, быть может, гигабайты памяти. Переместить всю эту память, не испортив при этом адреса, – нетривиальная задача с точки зрения производительности.

> Если вы хотите самостоятельно изучить код уплотнения SOH в CoreCLR, то обратите внимание на методы `relocate_phase` (который обновляет адреса перемещенных объектов) и `compact_phase` (который рекурсивно обходит дерево заполненных блоков кирпич за кирпичом, вызывая методы `compact_plug` и `compact_in_brick`).

Процесс уплотнения состоит из описанных в следующих разделах шагов, на которых используется информация, полученная на этапе планирования.

Получение нового эфемерного сегмента при необходимости

Этот шаг выполняется, если на этапе планирования было обнаружено, что необходимо расширить эфемерный сегмент (после уплотнения не хватит места для поколения 0 и/или 1). В результате текущий эфемерный сегмент расширяется посредством повторного использования другого (как описано в главе 7) или создания нового.

Перемещение ссылок

На этом шаге обновляются все адреса объектов, которые будут перемещены впоследствии. Благодаря данным, собранным на этапе планирования, это можно сделать еще до фактического перемещения объектов. Понятно, что работы здесь много, т. к. по управляемой куче разбросано множество таких ссылок. Во время перемещения ссылок активно используются кирпичи и деревья заполненных

блоков, чтобы быстро транслировать старый адрес в новый. На этом шаге в поисках адресов просматриваются различные области памяти, а именно:

- ссылки в стеке – при поддержке среды выполнения просматриваются кадры стеков во всех управляемых потоках, чтобы найти ссылки на управляемые объекты;
- ссылки внутри объектов, хранящиеся в межпоколенческом запомненном наборе, – в случае неполной сборки мусора все межпоколенческие ссылки, хранящиеся в картах (см. главу 5), должны быть обновлены (сюда входят межпоколенческие ссылки в SOH и в LOH);
- ссылки внутри объектов в куче малых и больших объектов – ссылки на другие объекты, находящиеся в выживших объектах, должны быть обновлены. В случае SOH для быстрого поиска выживших объектов (как мы знаем, они собраны в заполненные блоки) используются кирпичи и деревья заполненных блоков. При полной сборке в LOH на этом шаге, как правило, присутствуют только выжившие объекты, потому что уплотнению предшествует очистка LOH. Это дает возможность эффективно просматривать выжившие объекты в LOH и без поддержки со стороны кирпичей;
- ссылки внутри пред- и постблоков – как мы знаем, концы некоторых объектов могли быть затерты информацией о заполненном блоке. Предыдущее их содержимое хранится в записях очереди закрепленных заполненных блоков. Если там есть ссылки, то их тоже надо обновить;
- ссылки внутри объектов в очереди готовых к финализации – адреса объектов, оставшихся в этой очереди (см. главу 12), необходимо обновить;
- ссылки в таблицах описателей – указатели, хранящиеся в описателях, необходимо обновить.

Чем больше ссылок хранится в объектах, тем больше работы у сборщика мусора на этом этапе. В типичном приложении это, возможно, и не страшно. Но при наличии очень сложных структур данных, да еще на критическом с точки зрения производительности участке кода, стоит подумать о том, как обойтись без прямых ссылок на объекты.

> Если вы хотите изучить код перемещения ссылок в CoreCLR, то начните с метода `gc_heap::relocate_phase`. Самый важный из вызываемых им внутренних методов, `gc_heap::relocate_address`, пользуется кирпичами и деревом заполненных блоков для трансляции адресов. К нему обращаются также методы `GCScan::GcScanRoots`, `gc_heap::relocate_in_large_objects` и `gc_heap::relocate_survivors`.

Уплотнение объектов

После того как на предыдущем шаге все ссылки обновлены, GC наконец может приступить к перемещению объектов. Этот процесс состоит из следующих шагов (рис. 10.2):

- копирование объектов – заполненные блоки копируются последовательно, при этом используются ранее вычисленные смещения переноса;
- восстановление информации, затертой пред- и постблоками, – поврежденные части объектов копируются из записей очереди закрепленных объектов.

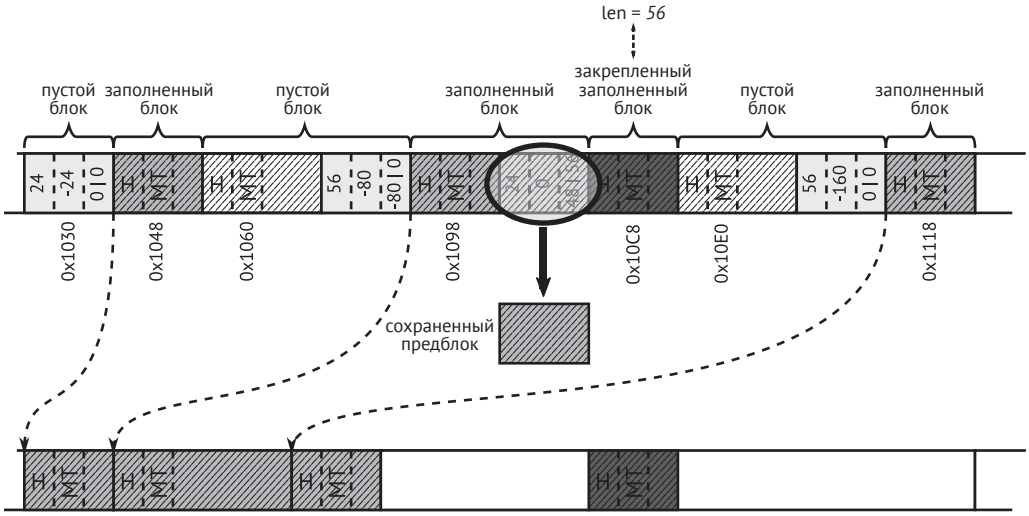


Рис. 10.2 ❖ Уплотнение кучи малых объектов
с использованием информации, вычисленной на этапе планирования

Хотя описание этого шага короткое и простое, надо понимать, что именно на нем производится много тяжелой работы. В случае полной сборки копирование всех заполненных блоков в управляемой куче может создать интенсивный трафик в памяти, поэтому большую часть всего времени уплотнения сборщик мусора проводит здесь.

Может возникнуть вопрос, как реализовано копирование объектов. Раз они копируются один за другим на месте в составе заполненных блоков, которые теоретически могут быть очень длинными, то почему они не затирают друг друга (в т. ч. и самих себя, см. рис. 10.3)?

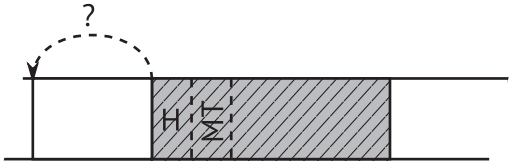


Рис. 10.3 ❖ Теоретическая проблема при копировании объектов:
в результате копирования на месте часть объекта может быть затерта

Первое, что приходит на ум, – воспользоваться промежуточным буфером (рис. 10.4). Но это означало бы удвоение трафика в памяти – каждый объект пришлось бы копировать дважды. Очевидно, что такое решение не годится.

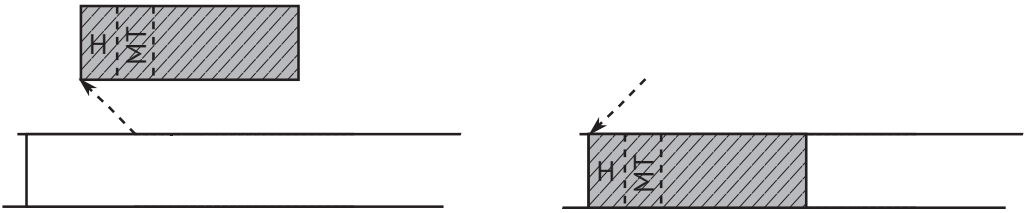


Рис. 10.4 ❖ Возможное решение проблемы копирования объектов – использование временного буфера

Но, поразмыслив, мы приходим к выводу, что никакой проблемы-то и нет. Мы без всякой необходимости рассматриваем объекты как детали конструктора Lego, которые нужно копировать целиком. Однако это всего лишь непрерывные области памяти, и копировать их можно по частям. CLR так и поступает. Идея скользящего уплотнения состоит в том, что копирование всегда начинается с младших адресов и такими кусочками, чтобы перекрытие было невозможно (в .NET расстояние между адресами перемещения ссылок не меньше размера указателя). Таким образом, копирование объектов производится функцией `memcopy`, которая копирует память блоками размером в четыре указателя, а затем копирует оставшуюся часть, размер которой составляет два или один указатель (листинг 10.1).

Листинг 10.1 ❖ Основная часть метода memcopy, применяемого для копирования объектов

```
void memcopy (uint8_t* dmem, uint8_t* smem, size_t size)
{
   const size_t sz4ptr = sizeof(PTR_PTR)*4;
   // ...
   // копировать блоками размером с четыре указателя
   if (size >= sz4ptr)
   {
      do
      {
         ((PTR_PTR)dmem)[0] = ((PTR_PTR)smem)[0];
         ((PTR_PTR)dmem)[1] = ((PTR_PTR)smem)[1];
         ((PTR_PTR)dmem)[2] = ((PTR_PTR)smem)[2];
         ((PTR_PTR)dmem)[3] = ((PTR_PTR)smem)[3];
         dmem += sz4ptr;
         smem += sz4ptr;
      }
      while  ((size -= sz4ptr) >= sz4ptr);
   }

   // копировать оставшиеся 16 или 8 байт
}
```

Строки листинга 10.1, в которых копируется память, транслируются в несколько ассемблерных команд `mov`, так что операция очень эффективна.

> Если вы хотите изучить код этапа уплотнения в CoreCLR, начните с метода `gc_heap::compact_phase`. Он для каждого активного кирпича вызывает метод `gc_heap::compact_in_brick`, который обращается к методу `gc_heap::compact_plug`.

Изменение границ поколений

Этот шаг, вызываемый после уплотнения, исправляет границы поколений, т. е. переустанавливает внутренние указатели выделения памяти, создает свободное место для запланированного контекста выделения памяти и выполняет другие необходимые корректировки.

Удаление (возврат) сегментов при необходимости

Реорганизовать сегменты, например вернуть системе ненужные (или сохранить их в списке для повторного использования, если включен режим придерживания виртуальной памяти).

Создание списка свободных блоков

Перед каждым закрепленным блоком создается новый свободный объект, и если он достаточно большой, то добавляется в список свободных блоков (как мы помним, длина вычисляется на этапе планирования и сохраняется в очереди закрепленных заполненных блоков) – см. рис. 10.5.

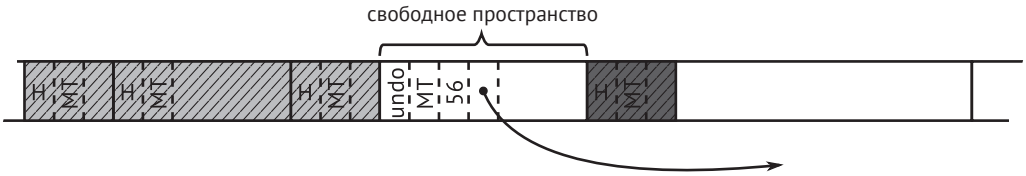


Рис. 10.5 ❖ Создание элементов списка свободных блоков перед закрепленными заполненными блоками (продолжение рис. 10.2)

Состаривание корней

Дополнительное состаривание производится для обновления очереди финализации (чтобы отразить изменение границ поколений) и состаривания (или омоложения) выживших описателей соответствующего типа.

Куча больших объектов

Техника уплотнения кучи больших объектов примерно такая же, как для кучи малых объектов. Но в силу отсутствия поколений, сложных заполненных блоков, кирпичей и дерева заполненных блоков ее реализация гораздо проще.

Если уплотнение LOH необходимо, то оно производится раньше, чем уплотнение SOH. Эта операция состоит из одного цикла просмотра помеченных объектов и их поочередного копирования в место назначения (для чего используется смещение переноса, вычисленное на этапе планирования в LOH). Кроме того, перед каждым закрепленным объектом создается свободное место (рис. 10.6), добавляемое затем в список свободных блоков. Разумеется, промежутки между объектами остаются, потому что это необходимо для последующих сборок мусора.

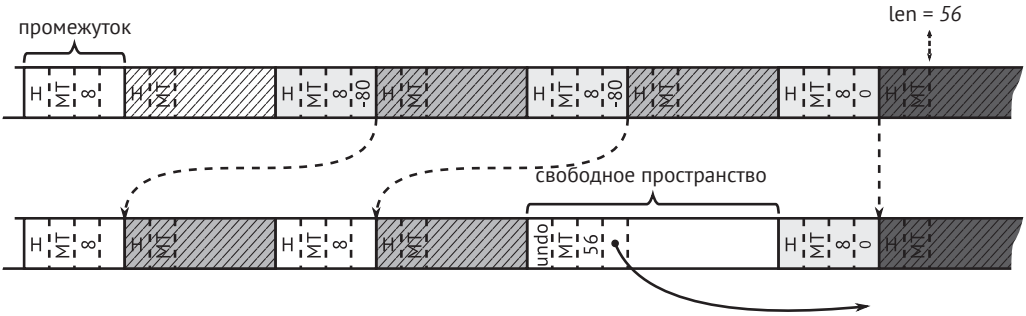


Рис. 10.6 ❖ Уплотнение кучи больших объектов
с использованием информации, вычисленной на этапе планирования

Сценарий 10.1. Фрагментация кучи больших объектов

Описание. Разрабатывая приложение, мы обратили внимание, что оно потребляет существенно больше памяти, чем ожидалось. Приложение обрабатывает большие пакеты данных и порождает новые пакеты – будем считать, что речь идет о пакетной обработке изображений. В листинге 10.2 приведен фрагмент кода. Обратите внимание на комментарии, в которых описываются размерные характеристики обрабатываемых данных. Как входные, так и выходные данные будут находиться в LOH, потому что их размер больше 85 000 байт. Мы собираемся сохранять блоки данных размером 100 КБ (`largeBlocks`), так что память для них будет выделяться в LOH.

Листинг 10.2 ❖ Пример, иллюстрирующий фрагментацию LOH

```
void Main()
{
   // ...
   List<byte[]> largeBlocks = new List<byte[]>();
   while (someCondition)
   {
      // ...
      var frame = reader.ReadBytes(size); // входной кадр всегда больше 85 000 байт
      var output = processor.Process(frame); // выход чуть больше входа
      var largeBlock = new byte[102_400];
      // сохранить часть выходных данных в largeBlock
      largeBlocks.Add(largeBlock);
   }
   // ...
}
```

Не думайте, что это искусственный пример, который не может встретиться на практике. Конечно, вы вряд ли будете писать такой наивный код, как в листинге 10.2. Но обработка пакета данных с появлением промежуточных результатов, которые необходимо сохранить, – это звучит куда более реально. Применение массивов (особенно байтовых) также нельзя считать неоправданным. Трудно представить себе фрагментацию LOH без использования массивов и строк, поскольку объекты именно этих типов чаще всего создаются в куче больших объ-

ектов. Весьма затруднительно было бы создать обычный объект, содержащий так много полей, что его необходимо размещать в LOH. Поэтому приведенный выше код вполне реалистично отражает суть проблем, с которыми можно столкнуться на практике.

Анализ. Предположим, что предварительный анализ показал, что именно размер LOH больше, чем ожидалось (см. табл. 10.1). Мы можем убедиться в этом с помощью счетчиков производительности или данных ETW.

Таблица 10.1. Ожидаемый и наблюдаемый размеры кучи больших объектов

Число объектов	Ожидаемый размер (МБ)	Наблюдаемый размер (МБ)
1000	102 400 000	152 769 104
2000	204 800 000	324 972 048
3000	307 200 000	463 287 752
4000	409 600 000	686 795 056

Записав сеанс ETW в PerfView (в стандартном режиме GC Collect Only), мы быстро обнаруживаем, что причина – фрагментация LOH (рис. 10.7). Из столбца LOH Frag % видно, что фрагментация составляет около 48 %. Столько места выброшено на ветер!

GC Events by Time

All times are in msec. Hover over columns for help.

GC Index	Pause Start	Trigger Reason	Gen	Suspend Msec	Pause MSec	% Pause Time	% GC	Gen0 Alloc MB	Gen0 Alloc Rate MB/sec	Peak MB	After MB	Ratio Peak/After	Promoted MB	Gen0 MB	Gen0 Survival Rate %	Gen0 Frag %	Gen1 MB	Gen1 Survival Rate %	Gen1 Frag %	Gen2 MB	Gen2 Survival Rate %	Gen2 Frag %	LOH MB	LOH Survival Rate %	LOH Frag %	Finalizable Surv MB	Pinned Obj
1	3,742.311	AllocLarge	2N	0.004	0.233	0.0	2.7	0.000	0.00	3.264	3.264	1.00	1.271	0.000	64	0.00	0.030	0	35.53	0.000	0	0.00	3.234	38	61.29	0.00	5
2	3,877.645	AllocLarge	2B	0.057	0.335	0.2	0.0	0.000	0.00	5.340	5.533	0.97	2.501	0.008	0	89.04	0.030	0	36.27	0.000	0	0.00	5.495	55	54.85	0.00	0
3	4,013.101	AllocLarge	2B	0.847	1.010	0.7	7.7	0.000	0.00	7.321	7.513	0.97	3.750	0.010	0	7.41	0.030	0	36.27	0.000	0	0.00	7.473	65	50.36	0.00	0
4	4,196.935	AllocLarge	2B	0.692	0.827	0.5	7.1	0.000	0.00	10.342	10.432	0.99	5.368	0.011	0	90.17	0.030	0	36.27	0.000	0	0.00	10.390	65	48.52	0.00	0
5	4,495.480	AllocLarge	2B	0.083	0.215	0.1	4.2	0.000	0.00	15.528	15.721	0.99	8.032	0.019	0	91.30	0.030	0	36.27	0.000	0	0.00	15.672	64	48.88	0.00	0
6	4,883.850	AllocLarge	2B	0.108	0.230	0.1	3.0	0.104	0.27	22.139	22.229	1.00	11.514	0.023	0	93.02	0.030	0	36.27	0.000	0	0.00	22.176	65	48.17	0.00	0
7	5,510.978	AllocLarge	2B	0.007	0.147	0.0	1.8	0.000	0.00	33.560	33.651	1.00	17.238	0.039	0	92.75	0.030	0	36.27	0.000	0	0.00	33.581	63	48.73	0.00	0
8	6,359.557	AllocLarge	2B	0.007	0.190	0.0	3.0	0.109	0.15	47.902	48.001	1.00	24.817	0.058	0	82.05	0.030	0	36.27	0.000	0	0.00	47.912	65	48.24	0.00	0
9	7,726.891	AllocLarge	2B	0.010	0.255	0.0	2.3	0.000	0.00	71.823	71.921	1.00	36.801	0.099	0	87.37	0.030	0	36.27	0.000	0	0.00	71.793	63	48.77	0.00	0
10	9,532.924	AllocLarge	2B	0.257	0.577	0.0	2.9	0.208	0.12	102.471	102.661	1.00	52.984	0.129	0	86.80	0.030	0	36.27	0.000	0	0.00	102.502	65	48.33	0.00	0
11	12,415.028	AllocLarge	2B	0.010	0.622	0.0	2.8	0.106	0.04	153.704	153.810	1.00	78.591	0.170	0	95.10	0.030	0	36.27	0.000	0	0.00	153.618	63	48.85	0.00	0
12	18,015.551	AllocLarge	2B	0.008	0.651	0.0	1.7	0.212	0.04	218.477	218.579	1.00	112.001	0.251	0	88.80	0.030	0	36.27	0.000	0	0.00	218.397	65	48.36	0.00	0
13	24,133.417	AllocLarge	2B	0.013	1.114	0.0	1.9	0.422	0.07	327.169	327.065	1.00	166.779	0.367	0	90.70	0.030	0	36.27	0.000	0	0.00	326.687	63	48.95	0.00	0
14	34,280.586	AllocLarge	2B	0.010	1.516	0.0	1.8	0.449	0.04	463.820	463.931	1.00	238.498	0.510	0	91.31	0.030	0	36.27	0.000	0	0.00	463.391	64	48.54	0.00	0
15	52,840.287	AllocLarge	2B	0.012	1.551	0.0	1.4	0.747	0.04	690.833	691.048	1.00	350.839	0.744	0	94.09	0.030	0	36.27	0.000	0	0.00	690.273	63	49.18	0.00	0

Рис. 10.7 ❖ Таблица GC Events by Time из отчета GCStats в Perf View для исследуемого процесса

Конечно, мы можем просто проанализировать свой код и найти, где создаются объекты в LOH. Но можно ли как-то облегчить эту задачу? На помощь, как часто бывает, приходит PerfView! Причиной фрагментации LOH являются мертвые объекты. Поэтому надо бы проверить, какие объекты чаще всего умирают в LOH. Учитывая такую заметную фрагментацию, вполне вероятно, что именно они являются причиной проблемы. К счастью, PerfView может предоставить такую статистику, если записать сеанс ETW в режиме .NET (а не GC Collect Only или GC Only). По завершении такого сеанса мы сможем открыть отчет **Gen 2 Object Deaths (Coarse Sampling) Stacks** (Смерть объектов в поколении 2 (приблизительная выборка) со стеками) в группе **Memory** (рис. 10.8). Помимо того что указано в названии, отчет содержит объекты в LOH. Как видим, умирает очень много массивов типа `System.Byte[]`. Это может оказаться полезно само по себе (если поможет однозначно идентифицировать источник такого выделения памяти). Но можно пойти дальше.

Name	Exc %	Exc	Exc Ct	Inc %	Inc
Type System.Byte[]	100.0	627,998,500	3,422	100.0	627,998,500.0
Type System.Char[]	0.0	800	0	0.0	800.0
Type System.String	0.0	800	0	0.0	800.0
Type System.Int32	0.0	200	0	0.0	200.0
Type System.Double	0.0	100	0	0.0	100.0
Type System.Byte[][]	0.0	100	0	0.0	100.0
Type CoreCLR.LOHFragmentation.DataFrame	0.0	100	0	0.0	100.0
GC Occured Gen(2)	0.0	0	15	0.0	0.0
Process64 CoreCLR.LOHFragmentation (32064) Args: 102400	0.0	0	0	100.0	628,000,600.0
Thread (30188) CPU=4528ms (Startup Thread)	0.0	0	0	100.0	628,000,600.0
OTHER <<ntdll!RtlUserThreadStart>>	0.0	0	0	100.0	628,000,600.0
coreclr.lohfragmentation!CoreCLR.LOHFragmentation.Program.Main(class System.String[])	0.0	0	0	100.0	628,000,600.0
coreclr.lohfragmentation!CoreCLR.LOHFragmentation.Processor.Process(class CoreCLR.LOHFragmentation.DataFrame)	0.0	0	0	100.0	627,873,200.0

Рис. 10.8 ❖ Отчет Gen 2 Object Deaths (Coarse Sampling) – представление By Name в PerfView, показывающее, какие объекты умирают в поколении 2

Выбрав из контекстного меню строки `System.Byte[]` команду **Goto Item in Callers** в группе **Goto**, мы увидим стеки вызовов выделения памяти для таких умирающих объектов (рис. 10.9). Вот это действительно полезная информация!

> Напомним, что это информация на основе выборок по событиям ETW `GCAllocationTick`. Но для объектов LOH ее достаточно, потому что такие события генерируются после каждых 100 КБ выделенной памяти. В LOH в 100 КБ не может поместиться два объекта, поскольку, по определению, размер каждого не менее 85 000 байт. При анализе фрагментации SOH можно получить более точные результаты, выбрав в PerfView режим сбора данных .NET Alloc или .NET SampAlloc.

Methods that call Type System.Byte[]

Name	Inc %	Inc	Inc Ct	Exc %
Type System.Byte[]	100.0	627,998,500.0	3,422	100.0
+ OTHER <<clr!JIT_NewArr1>>	100.0	627,998,500.0	3,422	0.0
+ coreclr.lohfragmentation!CoreCLR.LOHFragmentation.Processor.Process(class CoreCLR.LOHFragmentation.DataFrame)	100.0	627,873,200.0	3,421	0.0
+ coreclr.lohfragmentation!CoreCLR.LOHFragmentation.Program.Main(class System.String[])	100.0	627,873,200.0	3,421	0.0
+ OTHER <<ntdll!RtlUserThreadStart>>	100.0	627,873,200.0	3,421	0.0
+ Thread (30188) CPU=4528ms (Startup Thread)	100.0	627,873,200.0	3,421	0.0
+ Process64 CoreCLR.LOHFragmentation (32064) Args: 102400	100.0	627,873,200.0	3,421	0.0
+ ROOT	100.0	627,873,200.0	3,421	0.0
+ coreclr.lohfragmentation!CoreCLR.LOHFragmentation.Reader.ReadBytes(int32)	0.0	125,296.0	1	0.0
+ coreclr.lohfragmentation!CoreCLR.LOHFragmentation.Program.Main(class System.String[])	0.0	125,296.0	1	0.0
+ OTHER <<ntdll!RtlUserThreadStart>>	0.0	125,296.0	1	0.0
+ Thread (30188) CPU=4528ms (Startup Thread)	0.0	125,296.0	1	0.0
+ Process64 CoreCLR.LOHFragmentation (32064) Args: 102400	0.0	125,296.0	1	0.0
+ ROOT	0.0	125,296.0	1	0.0

Рис. 10.9 ❖ Отчет Gen 2 Object Deaths (Coarse Sampling) – представление Callers в PerfView, показывающее, какие методы создавали массивы System.Byte[]

Представление Callers ясно показывает, что есть два источника выделения памяти для умирающих массивов `byte[]`. Но метод `Reader.ReadBytes()` создает всего один умирающий массив, тогда как метод `Processor.Process` создает их тысячами. Конечно, в приложении может быть много разных типов «часто умирающих» объектов. В общем случае начинать поиск причин проблем лучше с объектов в начале списка. Так что нам стоит с подозрением отнестись к методу `Processor.Process`, создающему уж очень много умирающих байтовых массивов.

Еще один подход к диагностике этой проблемы – использование WinDbg с расширением SOS; при этом можно анализировать дамп памяти или присоединиться

к работающему процессу. Команда `!heapstat` даст общую картину всей управляемой кучи (листинг 10.3). Мы действительно видим, что LOH сильно фрагментирована (22 %). Кроме того, существует много еще не убранных в мусор, но уже недостижимых объектов (25 %). В совокупности это дает ожидаемую фрагментацию 47 %, что подтверждает полученные ранее результаты.

Листинг 10.3 ❖ Анализ фрагментации – команда !heapstat дает общую картину управляемой кучи

```
> !heapstat -inclUnrooted
Heap     Gen0     Gen1     Gen2          LOH
Heap0 1579192    96024       24   1907001192

Free space:                                  Percentage
Heap0    7816    11160        0    434527752 SOH:  1% LOH: 22%

Unrooted objects:                            Percentage
Heap0 1567816    65560        0    488427824 SOH: 97% LOH: 25%
```

Однако мы можем воспользоваться знаниями о том, как организована и выделяется память в куче больших объектов. Команда `!eeheap` выводит список всех сегментов LOH (листинг 10.4). Поскольку память растет, сегментов LOH много, как и следовало ожидать (согласно табл. 5.3, размер каждого равен 128 МБ, потому что процесс выполняется на 64-разрядной платформе, а GC работает в режиме рабочей станции). Мы знаем, что сегменты обычно создаются по одному, когда заканчивается память в текущем сегменте. И еще мы знаем, что распределитель выделяет память внутри сегментов линейно. То есть, немного упрощая, можно сказать, что чем больше адрес, тем новее хранящиеся по нему данные.

Листинг 10.4 ❖ Анализ фрагментации – команда !eeheap выводит список сегментов LOH

```
> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x0000013acb3c8730
generation 1 starts at 0x0000013acb3b1018
generation 2 starts at 0x0000013acb3b1000
ephemeral segment allocation context: none
         segment             begin         allocated               size
0000013acb3b0000  0000013acb3b1000  0000013acb549fe8  0x198fe8(1675240)
Large object heap starts at 0x0000013adb3b1000
         segment             begin         allocated               size
0000013adb3b0000  0000013adb3b1000  0000013ae33af528  0x7ffe528(134210856)
0000013ae4a60000  0000013ae4a61000  0000013aeca5fdb0  0x7ffedb0(134213040)
0000013aed130000  0000013aed131000  0000013af512f300  0x7ffe300(134210304)
0000013af5130000  0000013af5131000  0000013afd11c870  0x7feb870(134133872)
0000013a80000000  0000013a80001000  0000013a87fecf10  0x7febf10(134135568)
0000013a8a890000  0000013a8a891000  0000013a9287d0d0  0x7fec0d0(134136016)
0000013a92890000  0000013a92891000  0000013a9a8811c8  0x7ff01c8(134152648)
0000013a9a890000  0000013a9a891000  0000013aa28881a0  0x7ff71a0(134181280)
0000013aa2890000  0000013aa2891000  0000013aaa879090  0x7fe8090(134119568)
0000013aaa890000  0000013aaa891000  0000013ab287d060  0x7fec060(134135904)
0000013ab2890000  0000013ab2891000  0000013aba87bb20  0x7feab20(134130464)
```

```
0000013aba890000  0000013aba891000  0000013ac2880680  0x7fef680(134149760)
0000013afd130000  0000013afd131000  0000013b05117f28  0x7fe6f28(134115112)
0000013b05130000  0000013b05131000  0000013b0d118458  0x7fe7458(134116440)
0000013b0d130000  0000013b0d131000  0000013b0ecb6fc8  0x1b85fc8(28860360)
Total Size:               Size: 0x71c41750 (1908676432) bytes.
------------------------------
GC Heap Size:             Size: 0x71c41750 (1908676432) bytes.
```

Распечатав содержимое самого старого сегмента (первого в листинге 10.4), мы получим представление о том, как выглядит его фрагментация (листинг 10.5). Фрагментация отчетливо видна – области свободной памяти размером 78 974 байта чередуются с объектами длиной 102 424 байта. Их легко идентифицировать с помощью команды `!gcroot` (листинг 10.5). Например, единственным корнем последнего объекта (массива байтов) является локальная переменная типа `List<byte[]>` в методе `Main`, а именно `largeBlocks`. Это типичная картина фрагментации – большое количество живых объектов (главным образом массивов) чередуется со свободными блоками памяти.

Листинг 10.5 ❖ Анализ фрагментации – команда `!dumpheap` выводит список объектов в первом сегменте LOH (показано только несколько последних строк), а команда `!gcroot` находит корни выбранного для примера объекта

```
> !dumpheap 0000013adb3b1000 0000013ae33af528
...
0000013ae22b4cd8 00007fff857ebe10   102424
0000013ae22cdcf0 0000013ac914e200    78974 Free
0000013ae22e1170 00007fff857ebe10   102424
0000013ae22fa188 0000013ac914e200       30 Free
0000013ae22fa1a8 00007fff857ebe10   102424
0000013ae23131c0 0000013ac914e200    78974 Free
0000013ae2326640 00007fff857ebe10   102424
0000013ae233f658 0000013ac914e200       30 Free
0000013ae233f678 00007fff857ebe10   102424
0000013ae2358690 0000013ac914e200    78974 Free
0000013ae236bb10 00007fff857ebe10   102424
0000013ae2384b28 0000013ac914e200       30 Free
0000013ae2384b48 00007fff857ebe10   102424
0000013ae239db60 0000013ac914e200    78974 Free
0000013ae23b0fe0 00007fff857ebe10   102424
0000013ae23c9ff8 0000013ac914e200       30 Free
0000013ae23ca018 00007fff857ebe10   102424
> !gcroot 0000013ae23ca018
Thread 811c:
   000000233e9feeb0 00007fff28fc0645 CoreCLR.LOHFragmentation.Program.
   Main(System.String[])
      rbp-80: 000000233e9fef20
      -> 0000013acb3b68d0 System.Collections.Generic.List`1
      [[System.Byte[], mscorlib]]
      -> 0000013abaf50a68 System.Byte[][]
      -> 0000013ae23ca018 System.Byte[]

Found 1 unique roots (run '!GCRoot -all' to see all roots).
```

Однако мало знать, что между живыми объектами есть дырки. Настоящий вопрос в том, на месте каких объектов эти дырки образовались! Поискать ответ можно путем анализа недавно созданных данных. Распечатав содержимое самого нового сегмента (последнего в листинге 10.4), мы увидим, как выглядит в нем фрагментация (листинг 10.6). Если нам повезет, то на месте будущих дырок мы обнаружим какие-то объекты. Так оно и есть. В самой новой области LOH есть небольшие свободные участки-промежутки, объекты длиной 102 424 байта, которые мы уже видели раньше, а еще мы видим какие-то объекты между ними!

Листинг 10.6 ❖ Анализ фрагментации – команда `!dumpheap` выводит список объектов в последнем сегменте LOH (оставлено только несколько последних строк)

```
> !dumpheap 0000013b0d131000 0000013b0ecb6fc8
0000013b0ec0b4b0 0000013ac914e200      30 Free
0000013b0ec0b4d0 00007fff857ebe10   99634
0000013b0ec23a08 0000013ac914e200      30 Free
0000013b0ec23a28 00007fff857ebe10  102424
0000013b0ec3ca40 0000013ac914e200      30 Free
0000013b0ec3ca60 00007fff857ebe10   99627
0000013b0ec54f90 0000013ac914e200      30 Free
0000013b0ec54fb0 00007fff857ebe10   99635
0000013b0ec6d4e8 0000013ac914e200      30 Free
0000013b0ec6d508 00007fff857ebe10  102424
0000013b0ec86520 0000013ac914e200      30 Free
0000013b0ec86540 00007fff857ebe10   99628
0000013b0ec9ea70 0000013ac914e200      30 Free
0000013b0ec9ea90 00007fff857ebe10   99636
```

Проанализировав корни этих объектов, мы поймем, в чем истинная причина фрагментации (листинг 10.7). Очевидно, это байтовые массивы внутри экземпляров класса `DataFrame`, которые создаются методами `Program.Main` и `Processor.Process`.

Листинг 10.7 ❖ Анализ фрагментации – команда `!gcroot` находит корни объектов, вызывающих фрагментацию

```
0:000> !gcroot 0000013b0ec3ca60
Found 0 unique roots (run '!GCRoot -all' to see all roots).
0:000> !gcroot 0000013b0ec54fb0
Found 0 unique roots (run '!GCRoot -all' to see all roots).
0:000> !gcroot 0000013b0ec86540
Thread 811c:
    000000233e9feeb0 00007fff28fc0645 CoreCLR.LOHFragmentation.Program.
    Main(System.String[])
        r15:
            -> 0000013acb549228 CoreCLR.LOHFragmentation.DataFrame
            -> 0000013b0ec86540 System.Byte[]
Found 1 unique roots (run '!GCRoot -all' to see all roots).
0:000> !gcroot 0000013b0ec9ea90
Thread 811c:
    000000233e9fee50 00007fff28fc0aad CoreCLR.LOHFragmentation.Processor.
    Process(CoreCLR.LOHFragmentation.DataFrame)
        rbx:
            -> 0000013acb549240 CoreCLR.LOHFragmentation.DataFrame
```

```
          -> 0000013b0ec9ea90 System.Byte[]
Found 1 unique roots (run '!GCRoot -all' to see all roots).
```

На этом расследование завершается. Это простой пример, поскольку в LOH создаются объекты всего нескольких типов, а порядок выделения памяти для больших объектов специально выбран неудачно (каждый последующий входной пакет немного больше предыдущего). В результате образуются свободные участки, которые очень редко можно использовать повторно. В таком сценарии самые новые объекты скапливаются в конце, поэтому легко найти место, где еще могут остаться живые объекты перед сборкой мусора.

Но в сложных приложениях в LOH может присутствовать много объектов разного размера. И тогда анализ происхождения объектов, которые в итоге стали непригодными для использования дырками, будет куда утомительнее. Не существует общего золотого правила для исследования проблем с фрагментацией. Вообще, из всех связанных с памятью проблем эта самая трудная. И объясняется это зависимостью от времени. Дырки налицо, но что там было раньше, понять нелегко. В большинстве случаев дырки используются повторно благодаря выделению памяти из списка свободных блоков. Это еще сильнее осложняет расследование, т. к. новые объекты разбросаны по всему поколению 2 или LOH на месте прежних дырок. Увы, не существует события, которое позволило бы сказать «вот дырка, которая раньше была занята объектом X, но уже долго не используется». Мы располагаем только косвенными уликами.

> Помните, что в куче больших объектов хранятся также массивы, используемые самой CLR. Массивы ссылок на статические объекты, создаваемые в момент загрузки сборок, не должны составлять проблемы. Но есть также массивы, необходимые для интернирования строк (см. рис. 8.1 в главе 8 и раздел «Интернирование строк» в главе 4). И если вы злоупотребляете явным интернированием строк, то эти таблицы также могут стать причиной фрагментации LOH!

Итак, мы знаем, что фрагментация LOH – это проблема, но что с этим можно сделать? Начиная с версии .NET Framework 4.5.1 (и версии .NET Core 1.0) существует возможность установить режим принудительного уплотнения кучи больших объектов. Для этого нужно присвоить статическому свойству `GCSettings.LargeObjectHeapCompactionMode` значение `GCLargeObjectHeapCompactionMode.CompactOnce`. Уплотнение будет произведено только один раз, при первой блокирующей сборке мусора. Отметим, что это свойство влияет лишь на блокирующие сборки мусора, а для типичной неблокирующей (фоновой) GC оно не принимается во внимание. Поэтому чаще всего после установки этого свойства полная блокирующая сборка мусора запускается явно.

Таким образом, чтобы решить нашу проблему, мы можем запускать уплотнение LOH явно. Это можно делать периодически или только когда потребление памяти превысит некоторый порог (как в примере в листинге 10.8). Оба решения не идеальны и нуждаются в тщательной проработке. Они несут с собой все те проблемы, которые мы обсуждали при описании явных вызовов GC.

Листинг 10.8 ❖ Пример, иллюстрирующий фрагментацию LOH

```
if (GC.GetTotalMemory() > LOH_COMPACTION_THRESHOLD)
{
```

```
    GCSettings.LargeObjectHeapCompactionMode = GCLargeObjectHeapCompactionMode.CompactOnce;
    GC.Collect();
}
```

Кроме того что уплотнение LOH блокирующее, оно к тому же просто медленное. Время приостановки линейно зависит от суммарного размера выживших объектов. Даже если LOH мала и выжило лишь несколько сотен мегабайтов, работа приложения будет приостановлена на время от 100 до 200 миллисекунд. Чем больше размер выживших объектов, тем хуже. Когда это значение достигает нескольких гигабайтов, мы начинаем замечать, что приложение зависает больше, чем на секунду! На рис. 10.10 показан график зависимости времени GC от размера выживших объектов для обоих режимов: рабочей станции и серверного (точные величины могут зависеть от производительности оборудования).

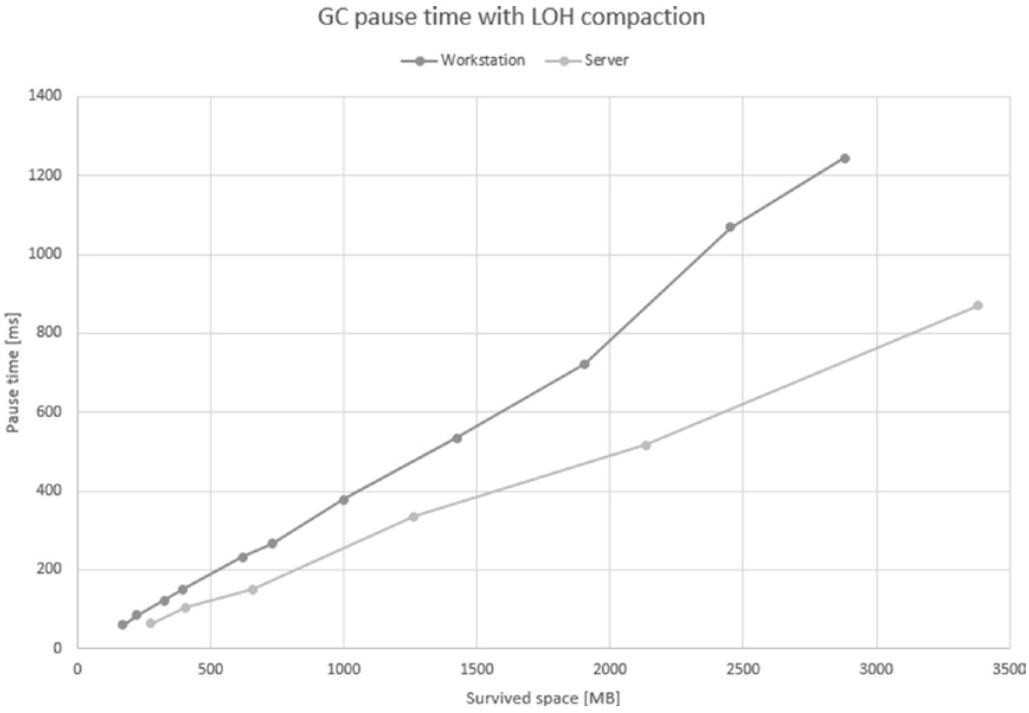


Рис. 10.10 ❖ Зависимость времени приостановки GC от фрагментации LOH для GC в режиме рабочей станции и в серверном режиме с 8 управляемыми кучами (график построен для компьютера Intel i7-4770K с 16 ГБ памяти типа DDR3-1600)

В серверном режиме уплотнение кучи больших объектов немного быстрее, потому что она разбита на несколько сегментов, которые можно уплотнять одновременно.

Бывает много ситуаций, когда иначе, как уплотнением LOH, проблему не решить, например если проблемный код не ваш и вы не можете его переделать, улучшив управление большими объектами. Но если вы пишете код сами, то есть гораздо лучший выход – организовать пул больших объектов или массивов (см.

разделы «Создание массивов – используйте ArrayPool» и «Создание большого числа объектов – используйте пул объектов» в главе 6).

> Есть планы когда-нибудь реализовать автоматический запуск уплотнения LOH при некоторых условиях. Это хорошо объяснено в комментарии на GitHub: «Можно смело предполагать, что в ближайшем будущем LOH по-прежнему не будет уплотняться автоматически, за исключением одного случая, когда мы, возможно, решимся это сделать, – если количество выживших объектов в LOH очень мало по сравнению с поколением 2, и коэффициент фрагментации LOH велик (скажем, не менее 75 %) и (или) LOH заполнена объектами, в которых нет ссылок (поскольку перемещение обходится очень дорого)».

Резюме

Как вы могли заметить, очистка производится очень быстро, потому что количество операций в памяти невелико. Для создания списка свободных блоков и восстановления памяти, затертой информацией о заполненных блоках, нужны лишь небольшие изменения. С другой стороны, уплотнение – весьма сложная процедура, которая может сопровождаться высоким трафиком памяти. Какую стратегию выбрать – дело описанного выше этапа планирования.

Эта глава завершает подробный рассказ о сердце управления памятью в .NET – самом сборщике мусора, который был описан в главах 7–10. Все сказанное в более ранних главах было только введением. А все, что последует дальше, – дополнения.

Подводя итог этим главам, перечислим основные аспекты сборки мусора, которые были в них описаны:

- механизмы, запускающие сборку мусора (глава 7);
- кооперация со средой выполнения для приостановки всех управляемых потоков на время выполнения GC (глава 7);
- как GC выбирает поколение, в котором произвести сборку мусора (глава 7);
- как с помощью обхода графа, начинающегося из различных корней, GC находит достижимые объекты (глава 8);
- как GC одновременно планирует сбоку очисткой и с уплотнением, а затем решает, какая окажется более продуктивной (глава 9);
- как выполняются очистка и уплотнение (эта глава).

Изложение этих вопросов перемежалось теоретическими отступлениями (как и почему что-то работает) и практическими сценариями (как использовать полученные знания при анализе проблем и разработке кода). Если вы читали все главы подряд, то теперь хорошо понимаете, что в действительности представляет собой сборка мусора в .NET. Практические сценарии помогут вам расследовать типичные проблемы и избежать типичных ошибок.

Поскольку эти главы тесно связаны друг с другом, относящиеся к ним правила собраны здесь, в конце главы 10.

Однако это не все. В сборщике мусора есть еще много потаенных уголков. Начиная с этого момента книга будет становиться все более и более практически ориентированной. И конечно, еще многое стоит сказать о работе внутренних механизмов – различных режимах GC (глава 11) и финализации (глава 12). Приглашаю вас продолжить путешествие!

ПРИМЕЧАНИЕ Заметим, что в главе, посвященной сборке мусора, ни разу не упоминался интерфейс `IDisposable`. Программисты с небольшим опытом иногда считают, что он как-то связан с механизмом сборки мусора, что `IDisposable` каким-то образом «запускает» сборку объекта в мусор. Но это совсем не так. `IDisposable` – всего лишь интерфейс, контракт между классом и разработчиком, который говорит, что время жизни экземпляра следует внимательно отслеживать и предпринимать определенные действия, когда объект перестает быть нужным. Чтобы не усугублять это недопонимание и не загромождать данную главу, описание механизма `IDisposable` помещено в главу 12.

Правило 17 : следите за приостановкой среды выполнения

Применимость. Носит общий характер, но применяется редко.

Обоснование. Приостановка среды выполнения – это служба, которой сборщик мусора пользуется, чтобы приостановить все управляемые потоки и создать безопасную обстановку для своей работы. Иными словами, во время неконкурентной сборки мусора пользовательские потоки не должны читать и модифицировать память, которой манипулирует GC. Этот процесс должен быть в высшей степени оптимизирован. Следует предпринять все возможные усилия, чтобы приостановка (и возобновление) потоков происходила максимально быстро. И эти усилия предприняты – для приостановки всех потоков требуются доли миллисекунды! В редких случаях, когда приостановка занимает больше времени, следует заподозрить неладное и выяснить, есть ли какая-то закономерность.

Как применять. Прежде всего мы можем измерить время приостановки движка выполнения в своем приложении. Самый удобный механизм для этого – события ETW. А самый простой способ проанализировать это время – заглянуть в таблицу GC Events by the Timetable из отчета GCStats в PerfView. Время, близкое к одной миллисекунде и более, должно настораживать.

В таком случае можно приступить к расследованию посредством продуманной отладки или выборки данных о загруженности ЦП во время приостановки – возможно, мы можем заметить, что наш код медлит с передачей управления среде выполнения (исполняя высокоприоритетные потоки или длинную синхронную операцию ввода-вывода).

Иллюстрирующий сценарий: 7.4.

Правило 18: избегайте кризиса среднего возраста

Применимость. Носит общий характер и очень популярно.

Обоснование. В основе дизайна сборщика мусора в .NET лежат гипотезы о поколениях, т. е. предположение о том, что объект либо умирает молодым, либо живет очень долго. Вы уже должны ясно понимать, почему сборка мусора в эфемерных поколениях сопровождается меньшими накладными расходами, чем в старших. Кризис среднего возраста – это несоблюдение гипотез о поколениях: большое количество объектов живет достаточно долго для перевода в поколение 2 только для того, чтобы там быстро умереть. Поколение 2 не предназначено для такого режима работы!

Как применять. Мы знаем, что создается много объектов и что многие из них рано или поздно попадают в поколение 2, где умирают. Поэтому следует уделять

больше внимания времени жизни своих объектов. Создание блока временных данных и хранение его в течение длительного времени – прямой путь к кризису среднего возраста. Однако в сложных приложениях часто бывает трудно судить о времени жизни создаваемых объектов. Поэтому общий способ применения этого правила – реакция на поведение программы: лишь после измерений мы замечаем, что процент времени, проведенного в GC, велик. Вот тогда мы приступаем к диагностике и расследованию.

На этом этапе нужно посмотреть:

- каково содержание старшего поколения – используя свой любимый инструмент для анализа дампов памяти;
- что умирает в старшем поколении – например, с помощью представления Gen 2 Object Deaths в PerfView (см. сценарий 10.1);
- для чего чаще всего выделяется память – поскольку кризис среднего возраста проявляется, когда создается много объектов, которые в конечном итоге оказываются в самом старом поколении (см. сценарий 6.2);
- по каким причинам выбирается самое старое поколение (см. сценарий 7.5).

Иллюстрирующие сценарии: 5.1, 6.2, 7.5, 10.1.

Правило 19: избегайте фрагментации старого поколения и LOH

Применимость. Носит общий характер и очень популярно.

Обоснование. Фрагментация, если удается использовать повторно свободные участки памяти, вовсе не так плоха – распределитель размещает в свободном пространстве новые объекты. Но бесконтрольная фрагментация может обернуться злом – если мы наблюдаем, что, несмотря на неоднократные сборки мусора, в данном поколении фрагментация не снижается. Потребление памяти программой может расти непредсказуемо, хотя количество используемых объектов невелико. Сильная фрагментация кучи малых объектов приводит к частым дорогостоящим уплотнениям. А в куче больших объектов бороться с фрагментацией еще труднее. Запускать ее приходится явно, и, будьте уверены, она занимает заметное время.

Как применять. Фрагментация SOH обычно не очень болезненна, если случается только в эфемерных поколениях. Для них уплотнение производится очень быстро, так что особо тревожиться по этому поводу не стоит. Хуже, если от фрагментации страдает поколение 2, и тому есть, по крайней мере, две причины:

- уплотнение поколения 2 обходится гораздо дороже, чем для эфемерных поколений, потому что оно часто занимает много сегментов. А значит, требуется гораздо больше операций с памятью;
- в результате фрагментации поколения 2 могут создаваться дополнительные сегменты. А чем больше сегментов, тем дороже сборка мусора в них.

По тем же причинам следует внимательно относиться к фрагментации кучи больших объектов. Но главная проблема в том, что LOH автоматически никогда не уплотняется, поэтому в гораздо большей степени уязвима для фрагментации.

Безусловно, необходимо следить за коэффициентами фрагментации в приложениях – например, с помощью сеансов ETW/LTTng. Но первый шаг – обнаружение сильной фрагментации. И лишь после этого нужно решить, действительно ли это составляет проблему – приводит к большим накладным расходам на сборку мусо-

ра или чрезмерному потреблению памяти? Если да, то пора переходить к самому трудному шагу – выявлению источников фрагментации. Здесь нет какого-то единого «золотого правила». Наиболее типичные подходы описаны в сценарии 10.1.

У проблемы фрагментации нет общего решения. Часто ее можно смягчить путем организации пула объектов, вызывающих фрагментацию, а именно массивов разных типов.

Иллюстрирующий сценарий: 10.1.

Правило 20: избегайте явной сборки мусора

Применимость. Носит общий характер и очень популярно.

Обоснование. Явные запуски сборки мусора нарушают работу этого механизма. Мы заставляем GC позабыть обо всех его внутренних самонастройках и приступить к сборке в конкретный момент. Да, есть несколько ситуаций, когда явный запуск оправдан, но в большинстве случаев – нет.

Как применять. Изучите GC – почему он работает, как и когда (например, прочитав эту книгу!). Тогда вы поймете, что чаще всего явный запуск GC – не решение возникшей проблемы. Нужно семь раз подумать, прежде чем прибегать к этому средству в своей программе. Существует очень немного ситуаций, в которых оно оправдано (они перечислены в разделе «Явный запуск» главы 7).

Иллюстрирующий сценарий: 7.3.

Правило 21: избегайте утечек памяти

Применимость. Носит общий характер и очень популярно.

Обоснование. Тут все понятно. Утечки памяти – это плохо. Точка. Из-за них программа становится непригодной для использования или со временем начинает работать настолько медленно, что ее приходится перезапускать. В худшем случае она просто «падает». Полагаю, никого не нужно убеждать в том, что утечка памяти нежелательна. Но вместе с тем встречаются небольшие и неизбежные утечки, с которыми можно смириться – если память растет так медленно, что никакого практического влияния это не оказывает, и лучше направить усилия на устранение более серьезных проблем. Например, если нам и так приходится перезапускать приложение раз в несколько дней, чтобы развернуть новую сборку, и при этом мы знаем о небольшой утечке памяти, возможно, нам стоит потратить время на более существенные проблемы производительности. Чаще всего такие «приемлемые» источники утечек находятся в стороннем коде, и устранить их мы попросту не можем.

Как применять. В .NET утечка означает неконтролируемый рост потребления памяти вследствие увеличения количества достижимых объектов. Проще говоря, что-то удерживает ссылки на объекты, которые, по идее, уже давно не используются и должны были бы умереть.

Это одна из самых распространенных проблем. Есть разные типы таких «скрытых» корней: статические переменные, события, неправильно сконфигурированные IoC-контейнеры и т. д.

В этой книге ряд примеров диагностики утечек памяти был представлен в форме сценариев. Среди них нет утечек, специфичных для конкретной технологии (например, встречающихся в WCF или WPF). Какие бы технологии ни использова-

лись в .NET сейчас и в будущем, сборщик мусора меняется гораздо медленнее – как и важнейшие инструменты: WinDbg, SOS и PerfView. Если налицо утечка памяти, знания, приобретенные после прочтения этой книги, помогут с ней разобраться!

Иллюстрирующие сценарии: 5.2, 8.1, 8.2, 9.1 и с 1.1 по 1.5 (чтобы различить утечки в управляемом и неуправляемом кодах).

Правило 22: избегайте закрепления

Применимость. Носит общий характер, умеренно популярно. Важно в коде, требующем высокой производительности.

Обоснование. Закрепление – плохо, поскольку зачастую ведет к фрагментации (см. правило 21). Кроме того, оно замедляет саму сборку мусора, т. к. усложняет работу внутреннего распределителя.

Как отмечалось в главе 9, закрепление бывает краткосрочным и долгосрочным, а проблематично среднесрочное. Если используется самая распространенная конкурентная сборка мусора и закрепленный объект находится в поколении 2, то обычно никаких проблем, в частности фрагментации, не возникает, поскольку сборка мусора в поколении 2 чаще всего фоновая (без уплотнения, так что есть закрепление или нет, не важно). Короткоживущие закрепленные объекты также, скорее всего, умрут в поколении 0 и не успеют привести к фрагментации.

Поэтому самыми неприятными являются те закрепленные объекты, которые живут достаточно долго, чтобы перейти в старшее поколение, где вызывают нежелательные побочные эффекты, например ограничивают свободу планирования поколения и приводят к необходимости реорганизовать сегменты (если в эфемерном сегменте закрепленных элементов так много, что он становится практически непригодным для использования).

Как применять. Вообще говоря, лучше всего не закреплять объекты вовсе, но иногда без этого не обойтись. В таком случае важно помнить, что наибольшие проблемы связаны со средним временем жизни. Поэтому если уж закрепления не избежать, то старайтесь соблюдать следующие правила:

- закрепляйте ненадолго, т. е. используйте ключевое слово `fixed` в очень коротком участке кода. В главе 8 отмечалось, что это влияет только на информацию для сборки одного метода, делая его специальным корнем. Поэтому если во время работы этого метода сборка мусора не начиналась, то со словом `fixed` вообще не будет связано никаких накладных расходов;
- создавайте долгоживущие закрепленные буферы. Тут сразу два преимущества: продлевается время жизни таких повторно используемых закрепленных объектов (так что они оказываются в поколении 2, где накладные расходы меньше), и улучшается локальность (они располагаются рядом, а не разбросаны по всей управляемой куче).

Следить нужно не только за фрагментацией, но и за количеством закрепленных объектов. Это не значит, что от закрепления нужно избавляться сразу после обнаружения. В типичном приложении можно не беспокоиться, пока фрагментация не слишком велика. С другой стороны, если от программы требуется высочайшая производительность и роль играет каждая миллисекунда, то отслеживать нужно все закрепленные объекты. Вам решать, какая ситуация характерна для вашей программы.

Иллюстрирующий сценарий: 9.2.

Глава 11

Варианты сборки мусора

В четырех предыдущих главах содержится очень подробное описание сборщика мусора в .NET – как правило, простейшего варианта. В этой главе мы рассмотрим все многообразие вариантов GC. Помимо знаний о том, как они спроектированы, мы поговорим о «плюсах» и «минусах» каждого варианта. Мы познакомимся с режимами работы и параметрами задержки GC.

Что касается различных вариантов GC в .NET, то чаще всего задают вопрос, какой использовать. Изучив различия, мы попытаемся ответить на него. Дополнительный интерес в этом плане могут представлять сценарии, приведенные в этой главе, поскольку в них рассматривается влияние выбранного режима на производительность и поведение приложения.

Обзор режимов

Краткие сведения о различных режимах работы GC в .NET уже были приведены в начале главы 7, в разделе «Общее описание». Теперь же можно копнуть поглубже и разобраться, чем и почему эти режимы отличаются.

Режим рабочей станции и серверный режим

Первая разделительная линия проходит между режимом рабочей станции и серверным режимом. Она существовала с момента создания среды выполнения .NET. Названия режимов связаны с тем, на какие приложения они рассчитаны. Впрочем, не стоит относиться к ним слишком серьезно. Хотя они действительно описывают типичное применение, ничто не мешает запускать настольное приложение в серверном режиме или веб-приложение в режиме рабочей станции – все зависит от потребностей. Разумнее считать, что эти режимы – два сильно отличающихся набора конфигурационных параметров.

Режим рабочей станции

Режим рабочей станции предназначен в основном для отзывчивости, необходимой в интерактивных приложениях с графическим интерфейсом. Интерактивность подразумевает, что видимые паузы приложения должны быть как можно короче. Мы не хотим, чтобы интерфейс надолго зависал из-за запуска сборщика мусора. Поэтому:

- сборка мусора будет производиться часто, зато при каждом запуске у сборщика будет меньше работы (было создано меньше объектов, а значит, образовалось меньше мусора);
- побочным эффектом будет пониженное потребление памяти – чем чаще производится сборка, тем агрессивнее освобождается память, так что остается меньше «зависшего» мусора;
- существует единственная управляемая куча – поскольку настольные приложения обычно выполняют одно главное действие, связанное с действием пользователя, и нет нужды специально распараллеливать их работу. Кроме того, этот режим предполагает, что на компьютере работает много приложений, каждое из которых занимает процессорные ядра и потребляет память. Поэтому необязательно и даже нежелательно умножать количество потоков GC для параллельной обработки нескольких куч одновременно. С самого начала режим рабочей станции проектировался в предположении, что есть одна управляемая куча, обрабатываемая одним потоком;
- сегменты меньше – чтобы приходилось работать с меньшими областями памяти.

Имейте в виду, что хотя таких решений достаточно для большинства интерактивных приложений, бывают и исключения. Вполне можно представить себе настольное приложение, которое производит параллельную обработку в фоновом режиме.

Серверный режим

Серверный режим предназначен для приложений, параллельно обрабатывающих поступающие запросы. Следовательно, необходима высокая пропускная способность – обработка максимально возможного количества запросов в единицу времени. В предположении, что запросы обрабатываются сравнительно быстро, спорадические приостановки приложения не окажут существенного влияния на них, поскольку статистически сборка мусора захватывает всего несколько таких запросов. Поэтому:

- сборка мусора производится реже – но при этом пауза может оказаться дольше, т. к. в промежутке между сборками создано больше объектов[1]. Однако это позволит повысить пропускную способность, поскольку, в то время когда сборщик не работает, можно обработать больше параллельных запросов;
- побочным эффектом является повышенное потребление памяти – раз сборки производятся реже, в промежутках между ними образуется больше мусора. Поэтому рабочий набор больше, чем в режиме рабочей станции. Но обычно «серверы» оснащаются большим объемом памяти, так что эта проблема не слишком серьезна;
- есть несколько управляемых куч – это означает, что механизм масштабируется вместе с ростом производительности оборудования. Если сборка мусора началась, то мы хотим, чтобы она закончилась как можно скорее. Параллельная обработка нескольких куч быстрее, чем обработка одной большой

[1] Но поскольку сборка производится параллельно на нескольких ядрах процессора, паузы могут оказаться даже короче, чем в режиме рабочей станции.

кучи[1]. Кроме того, каждое серверное приложение обычно размещается на своем сервере, так что к его услугам все имеющиеся ядра;

- размер сегмента по умолчанию больше, особенно в 64-разрядных системах, так что он может вместить гораздо больше объектов, прежде чем понадобится запустить GC;
- с учетом всего вышесказанного неудивительно, что в серверном режиме часто потребляется больше памяти, но процент времени, проведенного в GC, меньше.

Может возникнуть вопрос, как эти два режима организованы в исходном коде .NET и много ли у них общего кода. В случае CoreCLR (а как было сказано в главе 4, во всех версиях .NET используется один и тот же сборщик мусора) большая часть кода находится в одном файле `.\src\gc\gc.cpp`, в котором встречается немало директив препроцессора `#if`. Затем этот файл компилируется дважды с разными пространствами имен и значениями констант препроцессора. В файле `.\src\gc\gcsvr.cpp` определены константа `SERVER_GC` и пространство имен `SVR`:

```
#define SERVER_GC 1
namespace SVR {
  #include "gcimpl.h"
  #include "gc.cpp"
}
```

а в файле `.\src\gc\gcwks.cpp` – пространство имен `WKS`:

```
namespace WKS {
  #include "gcimpl.h"
  #include "gc.cpp"
}
```

Таким образом, различные типы и методы, относящиеся к сборке мусора, находятся в одном из двух пространств имен: `WKS::` или `SRV::`. От определения `SERVER_GC` зависят другие важные константы, особенно `MULTIPLE_HEAPS`, от которой, в свою очередь, зависят очень многие участки кода внутри `gc.cpp`.

Неконкурентный и конкурентный режим

Этот режим не зависит от того, в каком режиме – серверном или рабочей станции – работает GC, и имеет отношение к пользовательским потокам. Вообще говоря, под «неконкурентным» мы понимаем «не одновременный с чем-то еще», а «конкурентный» – прямо противоположное.

Неконкурентный режим

Неконкурентная версия GC существовала в .NET с самого начала – и в серверном режиме, и в режиме рабочей станции. На время GC приостанавливаются все пользовательские потоки. Концептуально это очень просто – нужно остановить все пользовательские потоки, выполнить GC и возобновить потоки.

---

[1] Напомним, что узким местом является доступ к памяти. Параллельная обработка кучи четырьмя ядрами не будет в четыре раза быстрее, чем обработка кучи такого же размера одним ядром. Но все-таки она, несомненно, будет быстрее.

Конкурентный режим

В конкурентном режиме, как можно ожидать, пользовательские потоки работают одновременно со сборкой мусора. Это усложняет как концепцию, так и реализацию. Необходима дополнительная синхронизация пользовательских потоков и сборщика мусора, чтобы у всех было одинаковое представление о реальности и не возникало серьезных проблем (например, проблем модификации уже убранных объектов или сборки еще живых). Очевидно, что реализовать такую синхронизацию нелегко, особенно так, чтобы сохранить требуемый уровень производительности. Вскоре мы увидим, как это сделано в .NET.

Конкурентная версия GC по-разному называется в разных версиях .NET, а именно:

- в режиме рабочей станции конкурентная версия существовала со времен .NET 1.0 и называлась Concurrent Workstation GC (конкурентная сборка мусора в режиме рабочей станции). В .NET 4.0 после существенных усовершенствований этот режим был переименован в Background Workstation GC (фоновая сборка мусора в режиме рабочей станции);
- в серверном режиме конкурентная версия появилась только в .NET 4.5 и называется Background Server GC (фоновая сборка мусора в серверном режиме).

> Если говорить об организации исходного кода, то оба режима реализованы в одном и том же файле `.\src\gc\gc.cpp`. Конкурентная версия выделяется директивой препроцессора `#if BACKGROUND_GC`. Но константа `BACKGROUND_GC` всегда определена в обеих версиях (`SVR` и `WKS`), которые содержат конкурентный и неконкурентный коды, а какой вариант выбрать, определяется на этапе запуска среды выполнения.

Конфигурирование режимов

Из предыдущих разделов ясно, что есть два независимых друг от друга параметра, каждый из которых может принимать два значения. Таким образом, получаем четыре режима работы GC. И этим настройки GC в общем-то исчерпываются. Читатели, привыкшие к очень детальным настройкам JVM, наверное, будут удивлены. Понятно, что это осознанное проектное решение. Подход JVM GC-центричен – мы можем настроить чуть ли не любой аспект работы GC, но должны очень хорошо понимать, что и почему делаем. А компания Майкрософт выбрала подход, ориентированный на приложения. Зная характер приложения, мы выбираем один из режимов работы сборщика мусора, а за подстройку под вид нагрузки он уже отвечает сам.

В следующих разделах кратко описано, как изменить режим работы GC в .NET Framework и более новом .NET Core.

> Эти режимы можно задать также при размещении CLR в своем процессе, установив соответствующие флаги с помощью интерфейса `ICLRRuntimeHost` (это относится и к .NET Framework, и к .NET Core). Именно это и делает простое приложение-хост CoreRun, которое строится в процессе сборки CoreCLR из исходного кода. В CoreRun используется очень упрощенный поставщик конфигурации, который игнорирует опи-

санные ниже параметры и учитывает только две переменные окружения: CORECLR_SERVER_GC и CORECLR_CONCURRENT_GC (обе могут принимать значения 0 или 1). Пользуйтесь ими, если захотите поэкспериментировать с самостоятельно собранной средой CoreCLR, размещенной в приложении CoreRun.

Вы, наверное, обратили внимание, что мы здесь не описываем, как эти параметры представлены на уровне файла проекта, например в Visual Studio. В экосистеме .NET есть много инструментов и форматов проекта. Нас будет интересовать лишь то, как параметры используются самой средой выполнения, а это вряд ли изменится в обозримом будущем.

Имейте в виду, что на машине с одним процессорным ядром GC всегда работает в режиме рабочей станции, вне зависимости от параметра gcServer.

.NET Framework

В приложениях для .NET Framework основным способом изменить режим GC является стандартный конфигурационный файл (листинг 11.1):

- в веб-приложениях ASP.NET, размещаемых в IIS, используется файл web.config. При этом ASP.NET по умолчанию включает серверный режим (а в версиях .NET 4.5+ еще и фоновый режим);
- в консольных приложениях и службах Windows по умолчанию используется файл [appName].exe.config. Если в нем не заданы соответствующие параметры, то по умолчанию включается конкурентный режим рабочей станции. Это может быть очень важно, особенно когда служба Windows обрабатывает много данных в ответ на запросы! Такая служба больше похожа на серверное, а не на интерактивное приложение. Поэтому перевод в серверный режим может заметно улучшить ее производительность.

Листинг 11.1 ❖ Относящаяся к GC часть конфигурационного файла для приложений .NET Framework (файл ([appName].exe.config или Web.config)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <startup>
      <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7" />
   </startup>
   <runtime>
      <gcServer enabled="true"/>
      <gcConcurrent enabled="true"/>
   </runtime>
</configuration>
```

.NET Core

В .NET Core возможностей настройки несколько больше. Параметры по-прежнему задаются в файле, но есть два дополнительных.

Конфигурационный файл очень похож на используемый в .NET Framework, только его формат сменился с XML на JSON (листинг 11.2).

Листинг 11.2 ❖ Относящаяся к GC часть конфигурационного файла SomeApplication.runtimeconfig.json для приложений .NET Core

```
{
  "runtimeOptions": {
    "tfm": "netcoreapp2.0",
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "2.0.0"
    },
    "configProperties": {
      "System.GC.Server": false,
      "System.GC.Concurrent": false
    }
  }
}
```

В CoreCLR введено понятие *конфигурационной рукоятки* (Configuration Knob). Задать ее значение можно разными способами, один из которых наиболее интересен – задание значения через переменную окружения (и реестр в случае Windows). Особенно это полезно в строго изолированных средах вроде образов Docker. Полный перечень конфигурационных рукояток можно найти на странице документации по CoreCLR.

Чтобы задать конфигурационную рукоятку с именем `X`, нужно завести переменную окружения `COMPlus_X` с требуемым значением или раздел реестра `HKCU\Software\Microsoft\.NETFramework`, в котором поле `Value` называется `X`. Так, в случае режима работы GC это будет:

- переменная окружения `COMPlus_gcServer=0` или `1` или раздел реестра `gcServer` со значением 0 или 1;
- переменная окружения `COMPlus_gcConcurrent=0` или `1` или раздел реестра `gcConcurrent` со значением 0 или 1.

> **ПРИМЕЧАНИЕ** У параметра, заданного в переменной окружения `COMPlus`, есть приоритет над таким же параметром из конфигурационного JSON-файла.

Приостановка и накладные расходы GC

Говоря об автоматическом управлении памятью, нельзя обойти стороной сопутствующие накладные расходы. В конце концов, GC – это код, работающий как часть нашего приложения. Он занимает процессор и может стать причиной пауз, когда в приложении больше ничего не работает. Мы пока не проявляли особого интереса к вопросу о накладных расходах, сопутствующих GC. Пора бы этим и заняться. Поскольку накладные расходы зависят от режима работы GC, сейчас для этого самое подходящее время.

Но как измерить эти накладные расходы? О каких расходах мы вообще говорим? В общем контексте производительности .NET-приложений у этого вопроса есть две стороны.

- Сторона GC – как уже отмечалось, у GC есть два очень важных и нежелательных побочных эффекта:

  - паузы GC – в настоящее время не существует GC вообще без пауз[1]. Приостановка потоков приложения, инициированная сборщиком мусора, очевидно, нежелательна, особенно в интерактивных приложениях. Нас может интересовать измерение длительности паузы на сборку мусора (суммарное, среднее, перцентили и т. д.). Величина приемлемого порога паузы зависит от характеристик конкретного приложения. На мой взгляд, есть повод беспокоиться, если время одной паузы превышает десятки миллисекунд, и это происходит довольно часто;
  - процессорные накладные расходы – выполнение кода GC, как и любого другого кода, потребляет ресурсы ЦП. Чем дольше работает GC или чем больше ядер он использует, тем меньше тактов процессора остается на выполнение обычного кода вашего и других приложений. Это важно как для конкурентного, так и для неконкурентного режима. И снова приемлемая величина порога зависит от характеристик приложения. В обычных веб-приложениях мне доводилось видеть постоянные накладные расходы на уровне 10–20 %, что довольно тревожно.
- Сторона приложения – вопросу об измерении производительности приложения можно посвятить целую книгу. Но к наиболее очевидным можно отнести следующие показатели:
  - пропускная способность – насколько быстро работает приложение. Например, сколько времени ему требуется для обработки одного HTTP-запроса или определенного пользовательского действия;
  - задержка – часто обращают внимание на распределение задержек, например сколько времени приходится на *x* % самых продолжительных действий;
  - потребление памяти – сколько потребляется памяти, особенно в пиковые периоды.

На рис. 11.1 показаны два самых популярных показателя накладных расходов на GC в .NET. Представлено два пользовательских потока (T1 и T2) и один поток GC (GC1). На рисунке показано изменение состояния потоков во времени. Поток, не занимающий процессор (ждущий чего-то), изображен штриховой линией, поток, исполняющий код GC, – стрелкой, а поток, исполняющий код программы, – светло-серым прямоугольником. Темно-серая область соответствует моменту приостановки и возобновления потоков. Далее мы будем придерживаться этого соглашения об изображении потоков.

При таком подходе легко проиллюстрировать оба самых популярных показателя:

- время пауз на GC – сюда входят неконкурентные этапы GC, включая приостановку и возобновление пользовательских потоков. Обычно эти данные можно получить из событий ETW/LTTNg, а конкретно – время между собы-

[1] В мире JVM можно встретить коммерческий GC под названием Azul Pauseless GC, но нельзя сказать, что в нем действительно нет пауз, потому что иногда потоки должны приостанавливать выделение памяти, чтобы «подождать отставших» (например, если GC не успевает предоставлять свободное место достаточно быстро). Такой GC принято относить к классу конкурентных сборщиков мусора с непрерывным уплотнением (Continuously Concurrent Compacting Collector – C4), и это название, пожалуй, вносит меньше путаницы.

тиями SuspendEEStart и RestartEEStop. Увидеть их можно в таблице GC Events by Time из отчета GCStats в PerfView (столбец Pause MSec);

- доля времени процессора, затраченная на сборку мусора, – равна отношению полного времени, проведенного в GC (включая его конкурентную часть), ко времени с момента предыдущей сборки мусора. Увидеть эту величину можно в таблице GC Events by Time из отчета GCStats в PerfView (столбец % GC).

Для измерения накладных расходов на GC можно также использовать популярный счетчик производительности % времени в GC (% Time in the GC). Однако он менее точный, и разработчики .NET рекомендуют измерения, основанные на событиях ETW (после появления фоновой GC они вообще тратят больше времени на поддержку ETW, чем на счетчики производительности). Имейте в виду, что если сборки мусора не было, то этот счетчик не обновляется и содержит предыдущее значение. Так что пусть вас не смущает, если в системном мониторе постоянно будет значение 99 % – это просто значит, что последнее значение не обновлялось, поскольку сборщик мусора не запускался! Всегда проверяйте, работает ли GC, например глядя на счетчик Сборок в поколении 0.

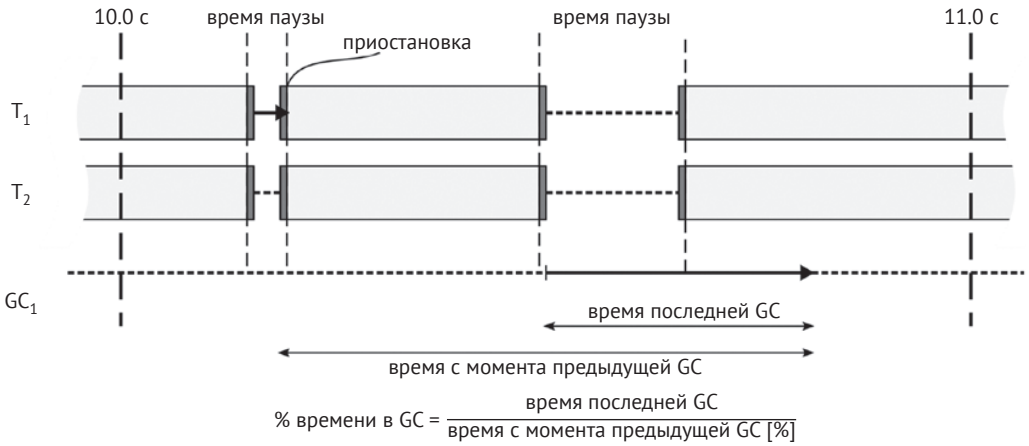


Рис. 11.1 ❖ Типичные показатели накладных расходов на GC в .NET: время пауз и % времени в GC

Понятно, что есть много других бесплатных и коммерческих инструментов, которые измеряют эти показатели по-своему. Как именно они измеряются – детали реализации, узнать об этом можно в документации.

Мы вернемся к этим измерениям при рассмотрении различных режимов работы GC. А пока перейдем к подробному описанию всех четырех вариантов сборки мусора в .NET.

Описание режимов

В следующих четырех подразделах описывается, как работают все четыре режима GC, имеющихся в .NET. Мы будем использовать рисунки, аналогичные рис. 11.1. Для простоты на большинстве рисунков удалены участки, соответствующие при-

остановке. Кроме того, везде предполагается, что в какой-то момент распределитель решил, что необходимо произвести сборку мусора. Длины отрезков на диаграммах носят иллюстративный характер. Сколько времени занимает GC, а сколько – пользовательские потоки, следует измерять с помощью подходящего инструмента.

Помимо описания работы для каждого режима приводится перечень типичных ситуаций, в которых его имеет смысл использовать.

Неконкурентный режим рабочей станции

Простейший режим работы GC был описан во всех подробностях в главах 7–10. Это база работы GC в .NET. Теперь рассмотрим его под тем же углом зрения, что и остальные режимы в этой главе.

Неконкурентный режим рабочей станции, который мы далее будем называть просто неконкурентным GC (без уточнения: рабочей станции или серверный), обладает следующими характеристиками (рис. 11.2):

- все управляемые потоки приостанавливаются на все время сборки мусора независимо от того, производится она в поколении 0, 1 или 2 (полная сборка мусора) – одна сборка мусора в эфемерном поколении должна занимать мало времени, так что ее неконкурентность – не проблема. Но, как показано на рисунке, полная блокирующая сборка мусора (сборка, производимая в неконкурентном режиме, называется блокирующей) может занимать гораздо больше времени. Поэтому полные блокирующие сборки нежелательны;
- код GC выполняется в пользовательском потоке, инициировавшем сборку (из распределителя). При этом приоритет потока (как правило, нормальный) не изменяется, а это значит, что поток конкурирует с другими потоками в других приложениях;
- GC всегда выполняется на этапе «остановки мира»; при необходимости может выполняться уплотнение.

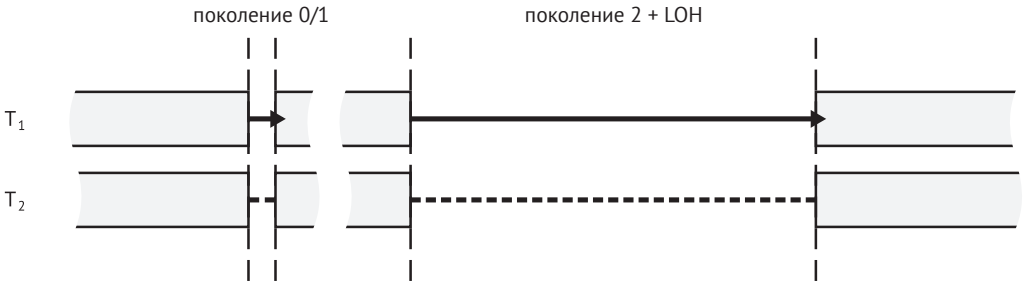


Рис. 11.2 ❖ Иллюстрация неконкурентной сборки мусора в режиме рабочей станции

Если бы мы захотели отслеживать такую GC в терминах событий ETW/LTTNg, то получили бы картину, показанную на рис. 11.3.

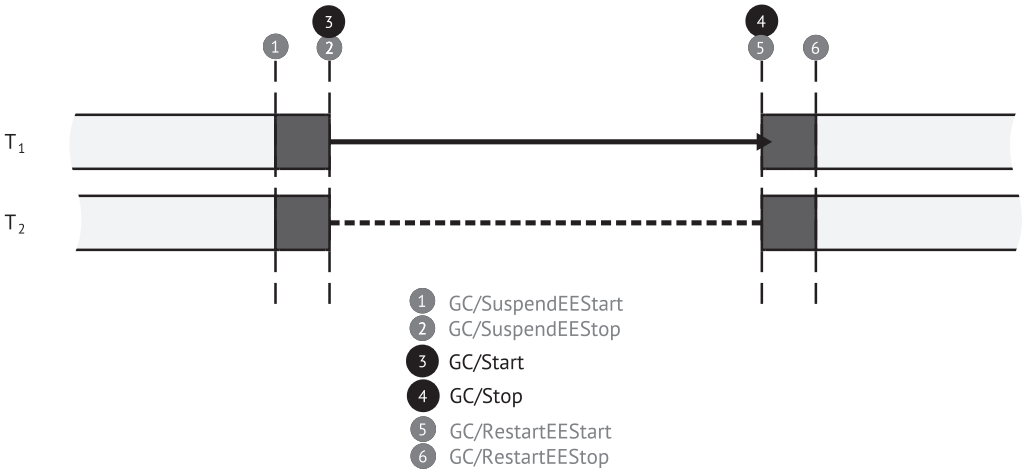


Рис. 11.3 ❖ События ETW/LTTNg,
генерируемые в процессе неконкурентной сборки мусора в режиме рабочей станции

Перечислим основные сценарии, в которых используется этот режим:

- в интенсивной среде, где ресурсов ЦП не хватает для обслуживания всех приложений, – так как нет специальных потоков GC, сборка мусора не увеличивает нагрузку на дефицитные процессорные ядра;
- в среде, где работает много нетребовательных к ресурсам веб-приложений (например, микросервисов в контейнере Docker). Если они потребляют мало памяти, то неконкурентной сборки мусора может оказаться достаточно. Также при этом немного уменьшается общее количество потоков, что может быть ценно с точки зрения распределения процессорных ядер между большим количеством одновременно работающих приложений.

Конкурентный режим рабочей станции (до версии 4.0)

Как уже отмечалось, этот режим назывался «конкурентный GC» и начиная с версии 4.0 замещен «фоновым GC» (Background GC). Поэтому мы не станем уделять ему много внимания (в частности, опустим весь раздел о реализации конкурентного GC). В описании пришедшего ему на смену преемника (см. следующий раздел) неявно присутствует и описание основ этого режима.

Конкурентный режим рабочей станции обладает следующими характеристиками (рис. 11.4):

- существует дополнительный поток, занимающийся исключительно сборкой мусора, – большую часть времени он приостановлен в ожидании работы;
- сборка мусора в эфемерных поколениях всегда неконкурентная – она производится достаточно быстро. Это также позволяет выполнить уплотнение при необходимости;

- полная сборка мусора может быть выполнена в двух режимах:
  - неконкурентная GC – поскольку все остальные потоки приостановлены («остановка мира»), полная сборка может производиться с уплотнением;
  - конкурентная GC – она выполняет большую часть работы, пока остальные управляемые потоки работают нормально. Поскольку уплотнение сильно усложнило бы реализацию, в этом режиме оно не производится;
- у конкурентной полной GC есть дополнительные характеристики:
  - управляемые пользовательские потоки могут создавать объекты во время сборки мусора, однако они ограничены размером эфемерного сегмента, потому что если место в нем кончится, то выделить дополнительное будет невозможно (во время конкурентной сборки нельзя запустить еще одну). Если такая ситуация возникнет, то пользовательские потоки приостанавливаются до завершения полной сборки мусора;
  - есть два коротких промежутка, когда все потоки приостанавливаются («остановка мира»), – в начале и в середине;
  - объекты, созданные в промежуток времени между началом GC и вторым промежутком «остановки мира», будут переведены в следующее поколение;
  - все объекты, созданные после второго промежутка «остановки мира», будут переведены в следующее поколение.

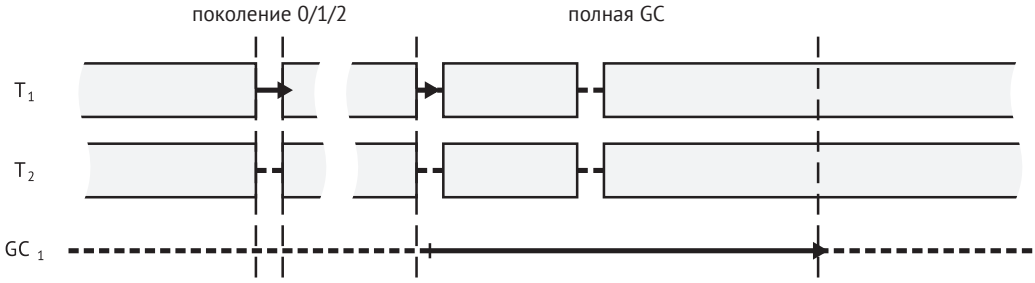


Рис. 11.4 ❖ Иллюстрация конкурентной сборки мусора в режиме рабочей станции (до версии .NET Framework 4.0)

Перечислим типичные сценарии:

- для большинства настольных приложений с графическим интерфейсом, созданных до .NET 4.0, конкурентная GC стала большим шагом вперед, позволившим уменьшить время пауз, что очень важно в интерактивных приложениях. Как правило, приложения не зависали надолго по вине GC. Понятно, что конкурентная GC была неуплотняющей, поэтому иногда приходилось запускать неконкурентную полную GC для устранения фрагментации. Однако необходимость блокировать выделяющие память потоки, когда кончается место в эфемерном сегменте, – серьезное ограничение. Размер сегментов в режиме рабочей станции невелик (а в 32-разрядном режиме всего-то 16 МБ!), так что даже конкурентная GC может привести к более длительной приостановке, чем хотелось бы. Преодоление этого ограничения в результате реализации фоновой GC в режиме рабочей станции стало серьезным улучшением.

Фоновый режим рабочей станции

Фоновый режим рабочей станции пришел на смену конкурентному режиму рабочей станции начиная с версии .NET Framework 4.0; он реализован также в .NET Core. Основные усовершенствования связаны с тем, что даже во время конкурентной GC можно при необходимости запускать GC в эфемерном сегменте. Тем самым снимается ограничение на создание объектов обычными потоками, что делает их практически независимыми от сборки мусора, работающей в фоне.

Фоновый режим рабочей станции обладает следующими характеристиками, по большей части такими же, как конкурентный режим рабочей станции (рис. 11.5):

- существует дополнительный поток, занимающийся исключительно сборкой мусора, – большую часть времени он приостановлен в ожидании работы;
- сборка мусора в эфемерных поколениях неконкурентная – она производится достаточно быстро. Это также позволяет выполнить уплотнение при необходимости;
- полная сборка мусора может выполняться в двух режимах:
  - неконкурентная GC – поскольку все остальные потоки приостановлены («остановка мира»), полная сборка может производиться с уплотнением;
  - фоновая GC – она выполняет большую часть работы, пока остальные управляемые потоки работают нормально. Как и в случае неконкурентной GC, уплотнение в этом режиме не производится;
- у фоновой полной GC имеются дополнительные характеристики:
  - управляемые пользовательские потоки могут создавать объекты во время сборки мусора, причем выделение памяти может запускать сборку в эфемерных поколениях (которая называется *приоритетной GC* (*Foreground GC*), в отличие от фоновой);
  - приоритетная GC может запускаться во время фоновой несколько раз. В документации по .NET написано: «Выделенный поток для фоновой сборки мусора часто проверяет в безопасных точках, есть ли запрос на приоритетную сборку мусора. Приоритетная GC – это обычная неконкурентная GC, на время которой фоновая GC временно приостанавливается. Она может быть уплотняющей (поскольку все потоки приостановлены) и даже может расширять кучу путем создания дополнительных сегментов»;

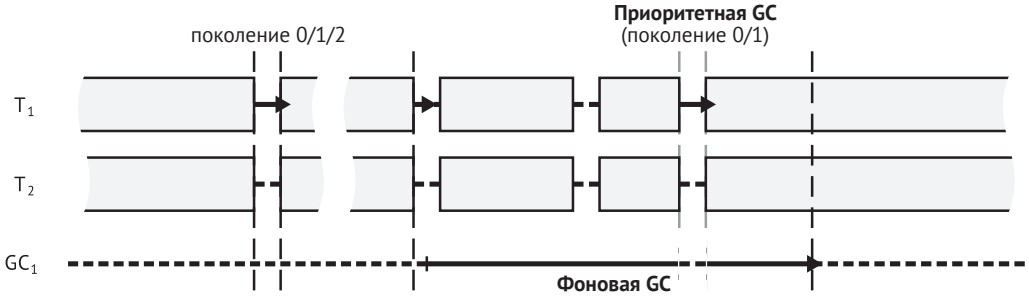


Рис. 11.5 ❖ Иллюстрация фоновой сборки мусора в режиме рабочей станции (начиная с версии .NET Framework 4.0)

– есть два коротких промежутка, когда все потоки приостанавливаются («остановка мира»), – в начале и в середине; оба будут подробно описаны ниже.

Займемся теперь «анатомией» фонового режима рабочей станции. Входящие в него неконкурентные этапы GC в поколениях 0, 1 и 2 тривиальны. Но как работает фоновая GC и в какие моменты может быть запущена приоритетная GC? Фоновую GC можно разделить на несколько стадий (рис. 11.6):

- начальная стадия «остановки мира» (**A**) – когда распределитель запустил обычный код GC, а тот решил начать фоновую GC. Кроме того, вполне вероятно, что на этой стадии возникнет необходимость выполнить обычную GC в эфемерных поколениях (например, если превышен какой-то бюджет выделения памяти). На этой стадии производится начальная пометка объектов, которая затем будет использована в процессе фоновой GC;
- конкурентная стадия пометки (**B**) – в то время как пользовательские потоки запущены, фоновый сборщик мусора продолжает определять достижимость объектов. Как именно это делается в условиях, когда пользовательские потоки одновременно с ним выполняют операции, будет описано ниже в этой главе. Дополнительно на этой стадии может быть запущено несколько приоритетных GC из-за создания новых объектов;
- стадия «остановка мира» для окончательной пометки (**C**) – пока пользовательские потоки приостановлены, фоновая GC окончательно определяет недостижимые объекты, которые уберет на следующей стадии;
- стадия конкурентной очистки (**D**) – пользовательские потоки работают, а GC в это время безопасно очищает уже неиспользуемые объекты, которые нашел раньше. На этой стадии могут быть запущены дополнительные приоритетные сборки мусора.

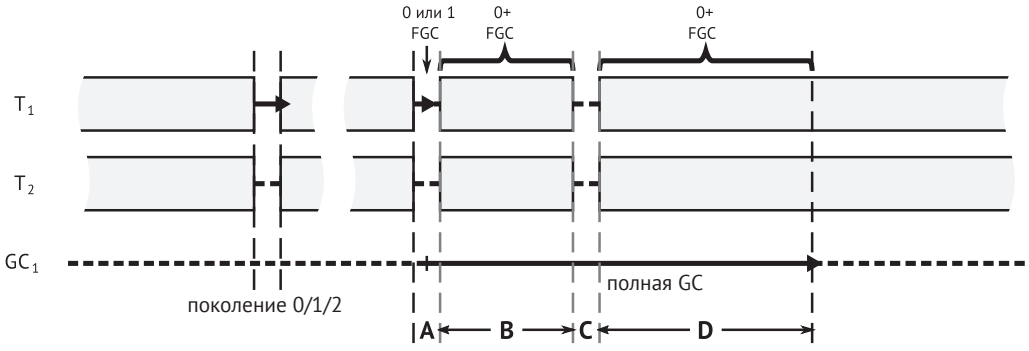


Рис. 11.6 ❖ Детальная иллюстрация фоновой сборки мусора в режиме рабочей станции

На рис. 11.7 показаны события ETW/LTTNg, полезные для отслеживания фоновой и приоритетной GC. Их гораздо больше, чем в случае простой неконкурентной GC (см. рис. 11.3). Как видим, помимо типичных событий GC, есть ряд событий, помеченных как BGC и описывающих детали фоновой GC. Два из них – `BGCRevisit` и `BGCDrainMark` – будут объяснены ниже, а остальные не нуждаются в пояснениях. Отметим, что на рис. 11.7 показан случай, когда в процессе фоновой GC случилась только одна приоритетная.

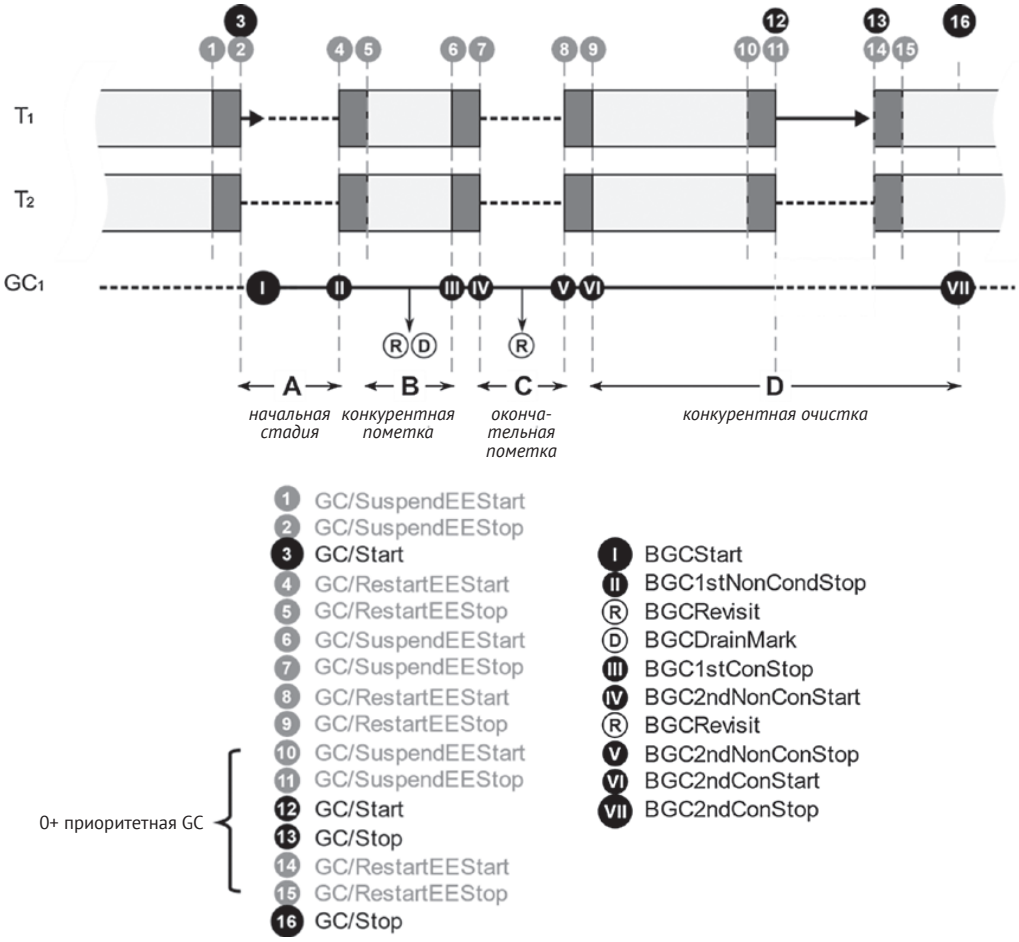


Рис. 11.7 ❖ События ETW/LTTNg, генерируемые в процессе одной фоновой сборки мусора в режиме рабочей станции

Код фонового GC для режима рабочей станции и серверного режима в основном общий (основное различие – количество потоков, исполняющих его). Но он компилируется дважды – в пространствах имен `SVR` и `WKS`. Если вы хотите изучить, как он устроен в CoreCLR, начните с метода `gc_heap::garbage_collect` и посмотрите, как используется флаг `do_concurrent_p`. Если требуется выполнить фоновую сборку мусора, то вызывается метод `gc_heap::do_background_gc`, который будит потоки фоновой GC. Интересно, что приоритетной и фоновой GC соответствует один и тот же метод `gc_heap::gc1`, различие лишь в значении глобального флага `settings.concurrent`. Таким образом:

- в случае приоритетной GC метод `gc_heap::gc1` выполняется со сброшенным флагом `concurrent` (это тот вариант, который описан в главах 7–10);
- в случае фоновой GC метод `gc_heap::gc1` выполняется с выставленным флагом `concurrent`. В результате вызываются методы `gc_heap::background_mark_phase` и `gc_heap::background_sweep`, которые кратко описываются в двух следующих разделах.

Перечислим основные сценарии использования.

- В большинстве приложений с пользовательским интерфейсом. Много сил потрачено на то, чтобы сделать паузы в фоновом режиме рабочей станции как можно короче. Поэтому такой режим идеален для всех видов интерактивных приложений (большинство из которых составляют приложения с графическим интерфейсом). Поскольку в фоновой GC уплотнение по-прежнему не производится, фрагментация может стать проблемой, поэтому для борьбы с ней иногда может запускаться блокирующая полная сборка мусора, которая сводит на нет все усилия по достижению низкой задержки.

Конкурентная пометка

Может возникнуть вопрос: как можно установить достижимость объекта, если пользовательские потоки работают? Ведь они постоянно модифицируют объекты, создают и удаляют ссылки между ними. Как определяется достижимость объектов в таких меняющихся условиях?

Как мы знаем, отслеживающий сборщик мусора, реализованный в .NET, определяет достижимость объектов, начиная обход графа объектов с различных корней (см. описание этапа пометки в главе 1 и на рис. 1.15). Уже посещенные объекты помечаются. В конце этого процесса живыми считаются только помеченные объекты, а остальные – мусор, который можно убрать. Если эта деятельность происходит одновременно с работой пользовательских потоков, то возникают два вопроса:

- как пометить объекты так, чтобы это не мешало работе пользовательских потоков?
- как обеспечить одинаковый взгляд на отношения между объектами со стороны пользовательских потоков и сборщика?

Сначала рассмотрим вопрос о пометке. В главе 9 было сказано, что для пометки объекта устанавливается один бит в его таблице методов. Когда «мир останавливался», подобный подход был возможен. Но модифицировать такой важный указатель в момент, когда его могут использовать другие потоки, нельзя из соображений безопасности и производительности (включая недействительность кеша).

Поэтому во время конкурентной пометки информация о ней хранится в специальном *массиве пометок*. Он организован как таблицы карт, описанные в главе 5. Один бит массива соответствует 16-байтовой области управляемой кучи (8-байтовой в случае 32-разрядной среды выполнения), как показано на рис. 11.8. Элементами массива пометок являются 4-байтовые *слова пометок*. Если сборщик мусора посетил объект и хочет пометить его, то устанавливается соответствующий бит в массиве пометок. Поскольку сборщик мусора – единственный владелец массива пометок, синхронизировать доступ к нему не нужно. Более того, в процессе конкурентной пометки биты могут только устанавливаться, но не сбрасываться. Это существенно упрощает синхронизацию в случае, когда конкурентную пометку выполняют несколько параллельных потоков (как в случае фоновой GC в серверном режиме, описанной ниже).

Отметим, что гранулярности 16 байт достаточно, потому что в области такого размера может находиться только один объект (напомним, что минимальный размер объекта равен 24 байтам). Впоследствии, просматривая установленные биты в массиве пометок, мы получим информацию о достижимости соответствующих объектов. Это простое решение первой проблемы конкурентной пометки.

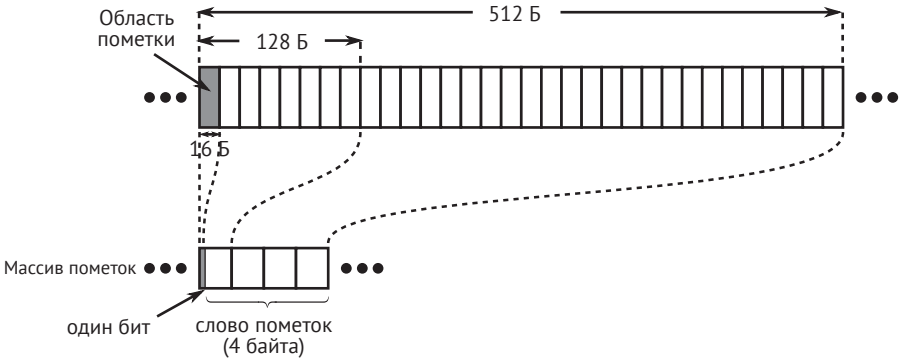


Рис. 11.8 ❖ Организация массива пометок
(в случае 64-разрядной среды выполнения)

Вторая проблема требует некоторого размышления. Что может случиться, если мы будем модифицировать ссылки между объектами, когда сборщик обходит граф объектов? Возможны следующие ситуации:

- в еще не посещенном объекте модифицированы (добавлены, удалены или и то, и другое) ссылки на другие объекты. Ничего страшного – этот объект еще не посещался, так что изменения будут учтены, когда сборщик мусора дойдет до него;
- в уже посещенном объекте удалена ссылка на объект, который без нее был бы недостижимым (рис. 11.9a). Тоже нормально – в течение некоторого времени будет существовать так называемый *плавающий мусор*. Позже GC увидит, что этот объект недостижим, и уберет его;
- в уже посещенном объекте добавлена ссылка на объект, который без нее был бы недостижимым (рис. 11.9b), например путем создания новой ссылки или перенаправления ссылки, которая указывала на другой объект. Вот это опасно. У нас может не оказаться шанса посетить (пометить) объект, который после такого изменения стал достижимым из другого объекта. Он будет считаться мусором, и сборщик уберет его, хотя объект, возможно, используется! Это так называемая проблема «потерянного объекта». При правильной реализации конкурентной пометки таких ситуаций не должно возникать;
- в уже посещенном объекте модифицирована ссылка на объект, который с ней был бы достижим. Чтобы понять, столкнулись мы с проблемой «потерянного объекта» или нет, нужно проверить, будет ли у нас еще шанс посетить этот объект;
- посещаемый в настоящий момент объект модифицировал свои ссылки. Необходимо проверить, был ли уже посещен объект, на который указывает ссылка. Если нет, то у нас первая ситуация. Если да – одна из трех оставшихся.

Решение обеих проблем представляется очевидным – проблематичные объекты необходимо посетить повторно! Существуют различные техники конкурентной пометки с разными компромиссами между количеством «плавающего мусора», количеством повторно посещаемых объектов и общими затратами на синхронизацию между пользовательскими потоками и сборщиком мусора.

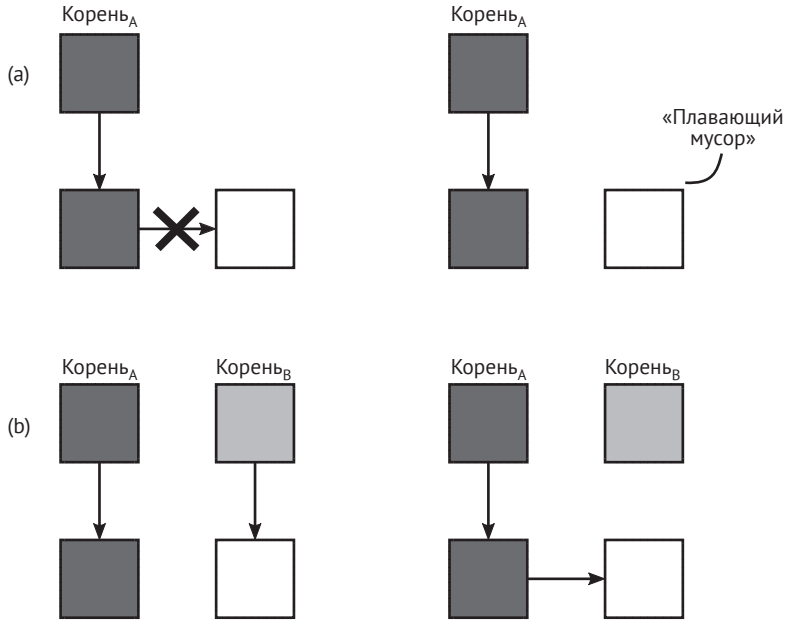


Рис. 11.9 ❖ Потенциальные проблемы в процессе конкурентной пометки: (a) создание плавающего мусора; (b) проблема потерянного объекта

В .NET была выбрана простая, но эффективная техника барьеров записи. Всякий раз как уже посещенный (или посещаемый в данный момент) объект модифицируется, его следует рассматривать как «подлежащий повторному посещению». Однако для простоты любая модификация считается основанием для повторного посещения. В Windows к ведению списка модификаций подключается операционная система с ее механизмом наблюдения за записью WriteWatch (он используется также в таблицах карт, рассмотренных в главе 5). Гранулярность этого механизма составляет одну страницу, поэтому любой модифицированный объект делает недействительной целую страницу размером 4 КБ. В других операционных системах CLR реализует собственный механизм WriteWatch с помощью специальным образом подготовленных барьеров записи, вставляемых JIT-компилятором, которые изменяют соответствующие байты в предназначенных для этой цели массивах. В определенные моменты сборщик мусора просматривает этот список модификаций (назовем его *списком наблюдения за записью*) и повторно посещает внесенные в него объекты (трактуя их как дополнительные корни). Это простое решение второй проблемы конкурентной пометки.

Вернемся к стадиям фоновой GC, показанным на рис. 11.6 и 11.7, и подробно опишем, что они делают.

- Начальная стадия «остановка мира» (A) – пока потоки приостановлены, подготавливается начальный список. Для заполнения «рабочего списка», который понадобится во время будущей конкурентной пометки, просматриваются только стек и очереди финализации. Рабочий список содержит лишь найденные объекты, проход по исходящим из них ссылкам пока не производится.

- Стадия конкурентной пометки (B) – пользовательские потоки работают, и вместе с ними выполняется главная часть конкурентной пометки. Граф объектов обходится из следующих корней (при этом помечаются объекты в массиве пометок):
  - описатели;
  - рабочий список, подготовленный на предыдущем шаге, – это означает, что обходится большой граф объектов, достижимых из корней в стеке. На этом шаге генерируется событие ETW/LTTNg `BGCDrainMark`, содержащее информацию о количестве объектов в рабочем списке;
  - список наблюдения за записью – в конце конкурентной пометки рассматриваются все объекты, модифицированные на этой стадии. На этом шаге генерируется событие ETW/LTTNg `BGCRevisit`, в котором сообщается, сколько обнаружилось «грязных» страниц и сколько объектов из-за этого было помечено;
- стадия окончательной пометки, или «остановка мира» (C), – это истина в последней инстанции. Все потоки приостанавливаются, и у GC есть возможность «подобрать остатки». В этот момент массив пометок должен довольно хорошо отражать фактическое состояние достижимости объектов. Но для пущей уверенности они проверяются еще раз. Заметим, что это делается инкрементно. При обходе графа объектов учитываются флаги в массиве пометок, так что многие объекты повторно не посещаются. Повторное посещение корней нужно только для того, чтобы не пропустить новых достижимых объектов. Разумеется, при этом образуется какое-то количество плавающего мусора (с уже помеченных объектов пометка не снимается), но, как уже было сказано, с точки зрения правильности результата это не проблема. В процессе окончательной пометки рассматриваются следующие корни:
  - стек, очереди финализации и описатели;
  - список наблюдения за записью – чтобы включить все модификации, которые GC не смог увидеть во время предыдущей проверки;
  - кроме того, производятся все обычные связанные с пометкой действия, в т. ч. просмотр зависимых описателей и слабых ссылок.

В CoreCLR код, отвечающий за конкурентную пометку, находится в методе `gc_heap::background_mark_phase`. Две самые важные структуры данных – это `mark_array` (реализует массив, показанный на рис. 11.8) и `c_mark_list` (реализует «рабочий список», заполненный на начальной стадии). Структура `c_mark_list` заполняется методом `gc_heap::background_promote_callback` во время просмотра стека и очереди финализации, а затем используется в методе `gc_heap::background_drain_mark_list`.

В Windows список наблюдения за записью управляется самой системой и используется сборщиком мусора в методе `gc_heap::revisit_written_pages`. Он получает от системы текущий список модифицированных страниц (из области памяти в управляемой куче) и просматривает все объекты в каждой странице с помощью метода `gc_heap::revisit_written_page`. В других операционных системах определены константы `DFEATURE_MANUALLY_MANAGED_CARD_BUNDLES` и `DFEATURE_USE_SOFTWARE_WRITE_WATCH_FOR_GC_HEAP`, которые активируют программный механизм наблюдения за записью. Как все это используется, можно видеть в барьерах записи, например `JIT_WriteBarrier_WriteWatch_PreGrow64`.

Конкурентная пометка целиком реализована в методе `gc_heap::background_promote`,

который вызывает методы gc_heap::background_mark_simple и gc_heap::background_mark_simple1 для обхода графа объектов (поднимая биты в массиве mark_array методом gc_heap::background_mark1).

Резюмируя, мы можем сделать следующие выводы об операции конкурентной пометки:

- она порождает какое-то количество плавающего мусора, т. е. сборка мусора менее агрессивна – мертвые объекты будут занимать место дольше времени, чем в случае блокирующей пометки;
- интенсивная модификация зависимостей между объектами во время фоновой сборки мусора может сделать недействительными много страниц, так что сборщику мусора придется повторно посетить много объектов (напомним, что размер страницы равен 4 KB и она может содержать много небольших объектов).

Конкурентная очистка

Когда наступает время для конкурентной очистки, массив пометок уже содержит информацию обо всех живых объектах. Как и в случае этапов планирования и очистки, описанных в главах 9 и 10, этой информацией можно воспользоваться для очистки мертвых объектов. На этом этапе объекты в куче просматриваются один за другим, каждый ищется в массиве пометок, и создаются элементы списка свободных блоков (точно так же, как было описано в главе 10). Поскольку во время конкурентной очистки может иметь место выделение памяти в SOH, интересно посмотреть на взаимодействие того и другого.

Этот процесс состоит из следующих шагов:

- прежде чем среда выполнения возобновит пользовательские потоки, списки свободных блоков во всех поколениях очищаются – начиная с этого момента распределители в течение короткого времени не будут знать о свободном пространстве (т. е. будут выделять память после уже использованной части сегмента);
- производится конкурентная очистка в эфемерных поколениях – создаются раздельные списки свободных блоков в поколениях 0 и 1, которые делаются доступными распределителю в самом конце (чтобы избежать одновременного доступа к списку из пользовательских потоков, выделяющих память, и потока GC). Таким образом, как только этот короткий шаг закончится, распределители в эфемерных поколениях смогут воспользоваться созданным свободным пространством. Кроме того, на протяжении этого шага приоритетная GC не разрешена, потому что она может быть уплотняющей и, стало быть, вступит в конфликт с продолжающимся последовательным просмотром объектов;
- конкурентная очистка в поколении 2 и в куче больших объектов – создаются списки свободных блоков в поколении 2 и в LOH, которые сразу же делаются доступными соответствующим распределителям. На этом шаге:
  - пользовательские потоки, выделяющие память, могут использовать уже опубликованный список свободных блоков в поколении 0;
  - разрешена приоритетная GC, так что если объекты переводятся из поколения 1 в поколение 2, то будет использоваться уже созданный список

свободных блоков в поколении 2 – это безопасно, потому что приоритетная GC – это обычная неконкурентная сборка мусора, во время которой фоновая GC временно приостанавливается, так что одновременного доступа к списку не будет;

- на протяжении всего процесса выделение памяти в LOH запрещено. Это объясняется тем, что распределителю в LOH понадобился бы доступ к списку свободных блоков памяти одновременно с изменяющим его сборщиком мусора. Если пользовательский поток захочет создать большой объект во время конкурентной очистки, он будет заблокирован до ее окончания. В начале и в конце такого ожидания генерируется пара событий ETW/LTTng `BGCAllocWaitBegin/BGCAllocWaitEnd`, которую можно поискать в трассах и узнать о нежелательных задержках (они также представлены в разделе «LOH Allocation Pause (due to background GC) > 200 Msec»[1] отчета GCStats в PerfView);
- в процессе конкурентной очистки (как и неконкурентной) могут быть удалены опустевшие сегменты (занятая ими память возвращается системе).

В CoreCLR весь код конкурентной очистки находится в методе `gc_heap::background_sweep`. Он вызывает метод `gc_heap::background_ephemeral_sweep`, который просматривает объекты из поколений 0 и 1, а затем просматривает объекты из поколения 2 и LOH (вызывая в определенных безопасных точках метод `gc_heap::allow_fgc` после просмотра каждых 256 объектов). В процессе просмотра объектов уже известные нам методы `gc_heap::thread_gap` и `gc_heap::make_unused_array` вызываются соответственно для создания списка свободных блоков или свободного пространства, непригодного для использования из-за малого размера.

Выделение памяти в LOH блокируется глобальным событием `gc_heap::gc_lh_block_event`, которое ожидает метод `gc_heap::user_thread_wait`, вызываемый из `gc_heap::wait_for_background_planning`. Эта последовательность находится в начале метода `gc_heap::a_fit_free_list_large_p`, который сам знаменует начало пути выделения памяти в LOH (см. описание в главе 6).

Неконкурентный серверный режим

С появления .NET и до версии .NET Framework 4.5 это был единственный и подразумеваемый по умолчанию режим, предназначенный для серверных приложений (в основном веб-приложений). На самом деле это простое расширение неконкурентного режима рабочей станции, описанного выше. Любая сборка мусора в этом режиме является блокирующей вне зависимости от того, в каком поколении производится. Но, как мы помним, с точки зрения управления памятью, есть существенное отличие – по умолчанию управляемых куч столько, сколько логических процессорных ядер.

Неконкурентный серверный режим сборки мусора обладает следующими характеристиками (рис. 11.10):

- существуют дополнительные потоки, предназначенные исключительно для сборки мусора, – по умолчанию их ровно столько, сколько управляемых куч (они называются просто *серверными потоками сборки мусора*). Большую часть времени они приостановлены в ожидании работы. Каждый поток обслуживает одну управляемую кучу;

[1] Пауза при выделении памяти в LOH (из-за фоновой GC) > 200 мс. – *Прим. перев.*

- все сборки мусора неконкурентные – благодаря параллельной сборке мусора в несколько потоков паузы оказываются короче, чем для кучи сравнимого размера в режиме рабочей станции. Режим «остановки мира» позволяет также производить уплотнение;
- пометка производится параллельно несколькими потоками GC – это уменьшает время блокировки. Кроме того, применяется техника *заимствования пометки* (mark stealing), чтобы операции пометки, выполняемые разными потоками, были сбалансированы. Кучи могут быть несбалансированными из-за различного распределения объектов с исходящими ссылками, тогда объем работы по пометке разнится. Поэтому потоки GC иногда «заимствуют» друг у друга пакеты объектов, подлежащих посещению.

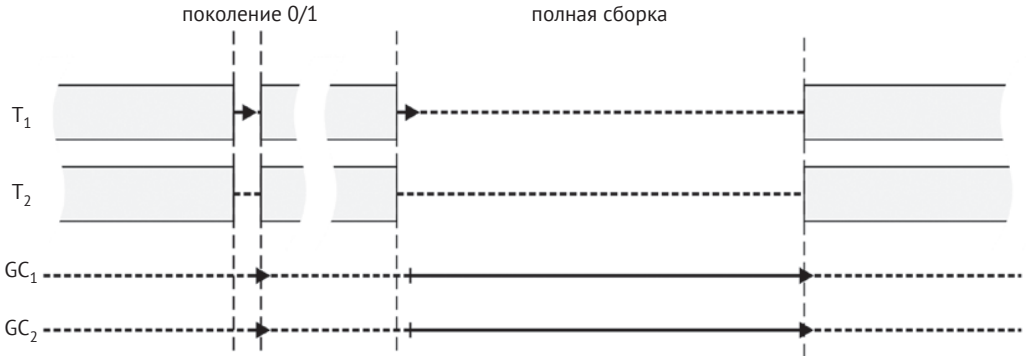


Рис. 11.10 ❖ Неконкурентный серверный режим сборки мусора

В серверном режиме количество управляемых куч и, следовательно, потоков GC необязательно в точности равно количеству процессорных ядер. В .NET Framework начиная с версии 4.6 и в .NET Core существует конфигурационный параметр `GCHeapCount`. Он задает количество потоков сборки мусора и управляемых куч. Задавать его можно только в серверном режиме GC с помощью переменной окружения `COMPlus_GCHeapCount` или в конфигурационном XML/JSON-файле (листинг 11.3). Значение должно быть меньше количества логических ядер, которые разрешено использовать процессу (операционные системы предоставляют различные способы ограничить эту величину), в противном случае оно будет автоматически уменьшено.

Листинг 11.3 ❖ Задание количества потоков GC и управляемых куч

```
<configuration>
   <runtime>
      <gcServer enabled="true"/>
      <GCHeapCount enabled="6"/>
   </runtime>
</configuration>
```

Ранее такие ограничения приходилось задавать на уровне операционной системы – чтобы среда выполнения думала, будто ей доступно меньше логических ядер, чем на самом деле. Но тут есть одна неприятность – ограничение распро-

страняется на всю среду выполнения, а не только на GC. Поэтому нежелательным ограничениям подвергается степень конкурентности всей .NET-программы, тогда как мы хотели ограничить только возможности GC. Поэтому параметр `GCHeapCount` – более предпочтительный способ управления этой стороной GC.

> Существует еще два параметра, связанных с привязкой потоков и куч к процессору: `GCNoAffinitize` и `GCHeapAffinitizeMask`. Они могут пригодиться в случаях, когда есть очень много процессоров, которые не полностью загружены из-за параметров типа `GCHeapCount`. С помощью указанных параметров можно назначить процессоры приложениям, так что у каждого приложения будет свой процессор.

Перечислим типичные сценарии использования неконкурентного серверного режима:

- в высоконагруженных веб-серверах, где присутствует сильная конкуренция за процессорные ядра, поскольку имеется много приложений и много конкурентных потоков, этот режим может оказаться лучшим вариантом, чем ресурсоемкий фоновый серверный режим GC, описанный ниже. Дополнительно можно ограничить потребление потоков с помощью параметра `GCHeapCount`;
- поскольку любая сборка мусора, включая полную, может быть уплотняющей, этот режим лучше борется с фрагментацией, чем конкурентный. Результат – меньший рабочий набор;
- поскольку все сборки блокирующие, не образуется плавающий мусор, как во время конкурентной пометки. Это еще больше сокращает рабочий набор.

Фоновый серверный режим

Начиная с версии .NET Framework 4.5 это режим по умолчанию для серверных приложений. Из всех режимов GC этот самый сложный. Но, зная, как устроены неконкурентный серверный режим и фоновый режим рабочей станции, нетрудно заметить, что он фактически является их комбинацией.

Фоновый серверный режим сборки мусора обладает следующими характеристиками (рис. 11.11), очень похожими на фоновый режим рабочей станции:

- для каждой управляемой кучи существует два потока, занимающихся исключительно сборкой мусора, – большую часть времени они приостановлены в ожидании работы:
  - серверные потоки GC – как и в неконкурентном серверном режиме, они отвечают за выполнение всех блокирующих сборок мусора (включая приоритетные);
  - фоновые потоки GC – дополнительные потоки, по одному для каждой кучи, отвечающие за выполнение фоновых сборок мусора;
- сборка мусора в эфемерных поколениях неконкурентная – она производится достаточно быстро. Это также позволяет выполнить уплотнение при необходимости. Они выполняются потоками приоритетной GC параллельно – каждый поток отвечает за свою управляемую кучу;
- полная сборка мусора может производиться в двух режимах:
  - неконкурентная GC – поскольку все остальные потоки приостановлены («остановка мира»), полная сборка может производиться с уплотнением.

Как и в эфемерных поколениях, все потоки серверной GC выполняются параллельно;
- фоновая GC – на протяжении большей части работы остальные управляемые потоки работают нормально. Уплотнение в этом режиме не производится. Как и в фоновом режиме рабочей станции, эта сборка мусора выполняется выделенными фоновыми потоками GC (работающими параллельно);

❍ у фоновой полной GC имеются дополнительные характеристики:
- управляемые пользовательские потоки могут создавать объекты во время сборки мусора, причем выделение памяти может запускать сборку в эфемерных поколениях (приоритетная GC);
- приоритетная GC может запускаться во время фоновой несколько раз;
- есть два коротких промежутка, когда все потоки приостанавливаются («остановка мира»), – в начале и в середине.

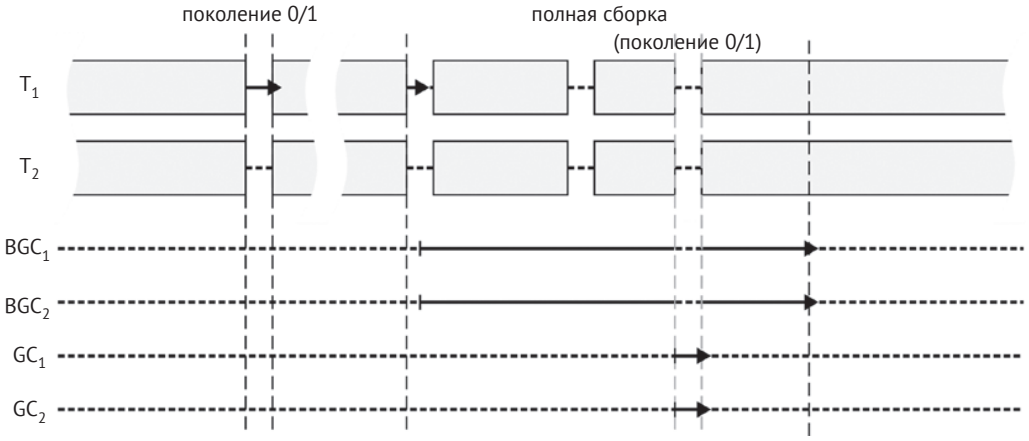


Рис. 11.11 ❖ Фоновый серверный режим GC

Точное описание фонового серверного режима было бы почти полным повторением описания фонового режима рабочей станции. Основное отличие состоит в том, что дополнительных потоков GC не один, а столько, сколько имеется процессорных ядер.

Понятно, что это технически весьма изобретательное решение, сочетающее преимущества фонового режима рабочей станции (короткие паузы, необременительные ограничения на выделение памяти потоками) и неконкурентного серверного режима (масштабируемость благодаря параллельной сборке). С точки зрения количества потоков, это самый ресурсоемкий вариант GC. На 8-ядерной машине только сборкой мусора будут заниматься 16 потоков.

Перечислим типичные сценарии использования этого режима:

❍ режим сборки мусора по умолчанию для большинства серверных приложений. Если на одном сервере размещено несколько десятков .NET-приложений, то вряд ли вам понравится, если все они будут работать в фоновом серверном режиме GC;

- ресурсоемкое настольное приложение, работающее на выделенной машине. Если организована контролируемая среда (как, например, на медицинском мониторе или заводской рабочей станции), в которой работает только одно приложение, то можно рассмотреть этот режим – он должен показывать хорошие результаты, т. к. располагает большим объемом ресурсов.

Режимы задержки

В дополнение к четырем режимам GC есть также независимый параметр, управляющий задержкой (или паузой). Он позволяет контролировать степень вмешательства сборщика мусора – насколько охотно тот будет приостанавливать программу. В отличие от ранее описанных режимов GC, режим задержки может динамически изменяться во время работы программы. Это открывает интересные возможности, о которых мы поговорим ниже.

Режим задержки можно задать с помощью конфигурационных рукояток (с помощью переменной окружения `COMPlus_GCLatencyMode`), но предпочтительнее делать это из программы, устанавливая статическое поле `GCSettings.LatencyMode`. Оно может принимать одно из значений, определенных в перечислении `GCLatencyMode` (листинг 11.4).

Листинг 11.4 ❖ Перечисление, определяющее режимы задержки

```
public enum GCLatencyMode
{
   Batch = 0,
   Interactive = 1,
   LowLatency = 2,
   SustainedLowLatency = 3,
   NoGCRegion = 4
}
```

Как мы увидим, режим задержки позволяет также управлять уровнем конкурентности GC. Это станет ясно из последующих разделов, где кратко описаны все режимы.

Пакетный режим

В пакетном режиме (`Batch`) длительность пауз нам безразлична. Это открывает возможности для оптимизации различных аспектов GC, например пропускной способности или потребления памяти. Пакетный режим по умолчанию подразумевается во всех неконкурентных режимах работы GC (т. е. когда приложение запущено с выключенным параметром `System.GC.Concurrent` или `gcConcurrent`).

На практике это позволяет подавить фоновую GC. Иначе говоря, мы можем использовать этот параметр, чтобы динамически выключить конкурентную сборку мусора, даже если она была включена в момент запуска среды выполнения. Но что в таком случае произойдет с потоками фоновой GC? Ответ зависит от режима работы GC:

- в случае серверного режима они просто останутся приостановленными, пока мы не вернемся в интерактивный режим задержки;

- в случае режима рабочей станции по истечении некоторого времени (в текущей версии это 20 секунд) произойдет тайм-аут, и потоки будут уничтожены, что сопровождается событием ETW/LTTNg `GCTerminateConcurrentThread`.

Интерактивный режим

В интерактивном режиме (`Interactive`) желательно, чтобы паузы были короткими, пусть даже за это придется заплатить повышенным потреблением памяти (например, если речь идет об интерактивном приложении с графическим интерфейсом пользователя). Этот режим является режимом по умолчанию для всех конкурентных сборок мусора, он включает возможность фоновой GC. Таким образом, это режим по умолчанию в .NET, поскольку и серверный режим, и режим рабочей станции по умолчанию конкурентные.

Этот режим можно использовать для динамического включения конкурентной GC – в таком случае будут созданы подходящие потоки фоновой GC, если их еще не существует, что сопровождается генерацией события ETW/LTTNg `GCCreateConcurrentThread`.

Кроме того, если активен режим рабочей станции и интерактивный режим (комбинация по умолчанию), то производится оптимизация по времени, описанная в разделе «Выбранное поколение» главы 7.

Режим низкой задержки

Режим низкой задержки (`LowLatency`) следует использовать, когда любой ценой необходимо уменьшить паузы до минимума. Он доступен только в режиме рабочей станции. В этом режиме отключаются все регулярные, конкурентные и неконкурентные сборки мусора в поколении 2 (полные) – весьма сильное ограничение! Полная сборка возможна только при получении уведомления «мало памяти» от системы или в результате явного запроса (например, вызова метода `GC.Collect`).

Не стоит и говорить, что этот режим оказывает очень сильное влияние на работу приложения:

- общее время паузы действительно очень мало, потому что сборка мусора производится только в эфемерных поколениях;
- потребление памяти заметно возрастает, потому что объекты, попавшие в поколение 2 или кучу больших объектов, не убираются вовсе.

Такой суровый режим следует включать только на короткое время, когда низкая задержка критична, например во время интенсивного взаимодействия с пользователем. Но помните, что после выхода из этого режима рано или поздно придется собрать весь накопившийся мусор, и на это может уйти много времени. Поэтому лучше самостоятельно запустить GC в подходящий момент, и чем раньше, тем лучше.

Устанавливая режим низкой задержки, следует позаботиться о том, чтобы гарантированно выйти из него – и поскорее. Обычной конструкции try/finally может оказаться недостаточно, потому что редко, но случаются ситуации, когда блок `finally` не выполняется. Чтобы подстраховаться, лучше использовать так называемые *области ограниченного выполнения* (Constrained Execution Region –

CER). На этот счет в документации по .NET написано: «Область ограниченного выполнения (CER) является одной из составляющих механизма создания надежного управляемого кода. В этой области общеязыковая среда выполнения (CLR) не может выдавать специализированные исключения, препятствующие полному выполнению заключенного в эту область кода. В этой области не может выполняться пользовательский код, в результате которого могут возникать специализированные исключения». Например, CLR откладывает аварийное завершение потока для кода, выполняемого внутри CER. Мы не будем отвлекаться на реализацию этого механизма, скажем лишь, что для его использования достаточно поместить перед блоком try обращение к методу `PrepareConstrainedRegions` (листинг 11.5).

Листинг 11.5 ❖ Безопасная установка режима LowLatency с помощью области ограниченного выполнения

```
GCLatencyMode oldMode = GCSettings.LatencyMode;
RuntimeHelpers.PrepareConstrainedRegions();
try
{
   GCSettings.LatencyMode = GCLatencyMode.LowLatency;
   // здесь выполняется короткое действие с жесткими временными ограничениями
}
finally
{
   GCSettings.LatencyMode = oldMode;
}
```

Режим длительной низкой задержки

Поскольку в режиме низкой задержки требования очень жесткие и куча может расти слишком быстро, в версии .NET Framework 4.5 появился дополнительный режим задержки, доступный как в режиме рабочей станции, так и в серверном режиме. Режим длительной низкой задержки – это компромисс между короткими паузами и умеренным потреблением памяти; в нем запрещены только неконкурентные полные сборки. Иными словами, разрешены сборки в эфемерных поколениях и фоновая сборка мусора. Этот режим доступен, только если при запуске среды выполнения были разрешены конкурентные режимы (пусть даже впоследствии они были запрещены вследствие перехода в пакетный или интерактивный режим задержки). Как и в предыдущем режиме, полная, блокирующая GC возможна только при получении уведомления «мало памяти» от системы или в результате явного запроса (например, вызова метода `GC.Collect`).

Режим длительной низкой задержки позволяет поддерживать низкую задержку (пусть и не такую низкую, как в режиме `LowLatency`) дольше, не опасаясь чрезмерно быстрого роста кучи. Это может оказаться очень хорошим компромиссом в случаях, когда требуется отреагировать на действие пользователя. Именно так сделано в исходном коде анализатора Roslyn, который используется в Visual Studio. Режим `SustainedLowLatency` включается, когда пользователь что-то печатает в редакторе, но по истечении тайм-аута восстанавливается предыдущий режим задержки (листинг 11.6).

Листинг 11.6 ❖ Пример установки режима SustainedLowLatency в исходном коде Roslyn[1]

```
/// <summary>
/// Этот класс управляет переключением GC в режим задержки SustainedLowLatency.
///
/// Его можно безопасно вызывать из любого потока, но предполагается, что он
/// вызывается из потока UI, когда пользователь нажимает на клавишу или щелкает
/// кнопкой мыши.
/// </summary>
internal static class GCManager
{
   /// <summary>
   /// Этот метод вызывается, чтобы подавить дорогостоящую блокирующую сборку
   /// мусора в поколении 2 в тех случаях, когда режим низкой задержки не
   /// подходит (например, при обработке типичных данных, введенных пользователем).
   ///
   /// Блокирующая сборка мусора автоматически восстанавливается через короткое
   /// время, если только метод UseLowLatencyModeForProcessingUserInput не будет
   /// вызван снова.
   /// </summary>
   internal static void UseLowLatencyModeForProcessingUserInput()
   {
      var currentMode = GCSettings.LatencyMode;
      var currentDelay = s_delay;
      if (currentMode != GCLatencyMode.SustainedLowLatency)
      {
         GCSettings.LatencyMode = GCLatencyMode.SustainedLowLatency;
         // Восстановить LatencyMode спустя короткое время после последнего
         // обращения к UseLowLatencyModeForProcessingUserInput.
         currentDelay = new ResettableDelay(s_delayMilliseconds);
         currentDelay.Task.SafeContinueWith(_ => RestoreGCLatencyMode(currentMode),
                                            TaskScheduler.Default);
         s_delay = currentDelay;
      }
      if (currentDelay != null)
      {
         currentDelay.Reset();
      }
   }
}
```

Регион без сборки мусора (No GC Region)

Это самое сильное из возможных ограничений, добавленное в версии .NET Framework 4.6. Как написано в документации MSDN, этот режим «является попыткой запретить сборку мусора на критическом пути выполнения, если можно получить запрошенную память». Иными словами, это попытка вообще запретить всякую сборку мусора, но понятно, что бесконечно так продолжаться не может. Поэтому

[1] Комментарии переведены на русский язык для удобства читателя. – *Прим. перев.*

мы не можем установить режим NoGCRegion, просто присвоив такое значение полю GCSettings.LatencyMode (установка значения GCLatencyMode.NoGCRegion не возымеет никакого эффекта). Вместо этого имеется специальный метод с несколькими перегруженными вариантами:

- bool GC.TryStartNoGCRegion(long totalSize);
- bool GC.TryStartNoGCRegion(long totalSize, bool disallowFullBlockingGC);
- bool GC.TryStartNoGCRegion(long totalSize, long lohSize);
- bool GC.TryStartNoGCRegion(long totalSize, long lohSize, bool disallowFullBlockingGC).

Как видим, все варианты принимают объем памяти в байтах (totalSize) – то количество памяти в каждой куче, которое мы хотели бы иметь возможность выделить, не активируя сборку мусора (иными словами, столько памяти должно быть доступно в момент вызова метода). Метод TryStartNoGCRegion возвращает true, если GC подтверждает, что столько памяти имеется в наличии и мы вошли в режим без задержки. Дополнительно можно указать, какая часть этой памяти может быть выделена в куче больших объектов (аргумент lohSize). Если lohSize не задан, то totalSize относится и к SOH, и к LOH (т. е. всего мы сможем выделить памяти в два раза больше, чем totalSize).

Если изначально памяти меньше, чем запрошено, то метод TryStartNoGCRegion запускает полную неконкурентную сборку мусора, пытаясь удовлетворить запрос. Но такое поведение можно запретить, задав параметр disallowFullBlockingGC.

Есть важное ограничение – заданный размер должен быть меньше или равен суммарному размеру всех эфемерных сегментов (т. е. подходящему кратному размеру эфемерного сегмента в случае серверной GC):

- если lohSize задан, то разность totalSize – lohSize (размер SOH) должна быть меньше или равна размеру эфемерного сегмента;
- если задан только totalSize, то невозможно сказать, что вы имели в виду: SOH, LOH или какое-то сочетание того и другого, поэтому на всякий случай предполагается, что вся величина totalSize должна быть меньше или равна размеру эфемерного сегмента.

Это связано с тем, что GC может не запускаться, если для удовлетворения запроса не нужно реорганизовывать сегменты из-за нехватки места в эфемерном сегменте. Если заданный размер превосходит суммарный размер эфемерных сегментов, то будет выдано исключение ArgumentOutOfRangeException.

После входа в режим без задержки выполнение программы продолжается как обычно. Пока в SOH и LOH выделено не больше памяти, чем задано, GC не запускается. Но надо не забыть выйти из этого режима, обратившись к методу GC.EndNoGCRegion()! С точки зрения GC это не так уж важно – даже если мы забудем, гарантируется, что после выделения totalSize байтов памяти будет восстановлен предыдущий режим задержки.

Однако с точки зрения API важно, чтобы каждому вызову GC.TryStartNoGCRegion соответствовал вызов GC.EndNoGCRegion, иначе следующий вызов GC.TryStartNoGCRegion выдаст исключение InvalidOperationException с сообщением «The NoGCRegion mode was already in progress». Это произойдет даже в том случае, когда превышен лимит выделения памяти и автоматически восстановлен предыдущий режим задержки! В таком случае мы все равно обязаны вызвать EndNoGCRegion, зная, что при этом бу-

дет выдано исключение InvalidOperationException с сообщением «Allocated memory exceeds specified memory for NoGCRegion mode».

Поскольку режим без GC по определению рассчитан на определенный объем выделения памяти, то его не нужно так тщательно защищать с помощью областей ограниченного выполнения, как режимы с низкой задержкой. В худшем случае будет просто запущена сборка мусора. Но всегда полезно проверять, что предыдущая область без GC закончена, прежде чем снова вызывать TryStartNoGCRegion, – нам ведь ни к чему исключение InvalidOperationException.

Приняв все вышесказанное во внимание, для использования режима без GC нужно выполнить несколько проверок, так что код будет иметь такую структуру, как в листинге 11.7.

Листинг 11.7 ❖ Пример создания области без сборки мусора

```
// на случай, если предыдущий блок finally не был выполнен
if (GCSettings.LatencyMode == GCLatencyMode.NoGCRegion)
   GC.EndNoGCRegion();
if (GC.TryStartNoGCRegion(1024, true))
{
   try
   {
      // Сделать что-то полезное.
   }
   finally
   {
      try
      {
         GC.EndNoGCRegion();
      }
      catch (InvalidOperationException ex)
      {
         // Запротоколировать сообщение
      }
   }
}
```

Заметим, что вызов метода GC.EndNoGCRegion без предшествующего успешного вызова GC.TryStartNoGCRegion приведет к исключению InvalidOperationException с сообщением «NoGCRegion mode must be set»[1]. Поэтому рекомендуется заранее проверять режим задержки, как в коде выше: if (GCSettings.LatencyMode == GCLatencyMode.NoGCRegion) GC.EndNoGCRegion. Но это бессмысленно делать в блоке finally. Как уже было сказано, даже в случае превышения лимита выделения памяти все равно нужно вызывать метод GC.EndNoGCRegion, хотя режим задержки GCSettings.LatencyMode уже вернулся к предыдущему значению (Batch или Interactive).

> Если вы хотите самостоятельно познакомиться с тем, как устроен режим задержки без GC в коде CoreCLR, начните с метода GCHeap::StartNoGCRegion, в котором реализованы все варианты метода GC.TryStartNoGCRegion. Он может вызывать метод GCHeap::GarbageCollect и обращается к методу gc_heap::prepare_for_no_gc_region для

[1] Должен быть установлен режим NoGCRegion. – *Прим. перев.*

проверки условия на размер эфемерного сегмента и задания разрешенного объема памяти, который можно выделить без сборки мусора. Впоследствии в момент, когда во время нормального выполнения программы должна была бы быть запущена сборка мусора, вместо нее вызывается метод `gc_heap::should_proceed_for_no_gc`, который проверяет, не нарушены ли лимиты выделения.

Цели оптимизации задержки

Если вы помните, в разделе «Статические данные» главы 7 был описан дополнительный уровень управления задержкой – *цели (уровни) оптимизации задержки*, – который влияет на значения статических данных. В комментариях к коду CoreCLR написано: «Режимы задержки требуют от пользователя знания специфики GC (например, что такое бюджет выделения памяти, полная блокирующая сборка мусора и т. д.). Мы пытаемся избавить пользователя от этого груза, потому что с его точки зрения гораздо уместнее сообщить нам, какой из аспектов производительности он считает наиболее важным» (а к таким аспектам относятся потребление памяти, пропускная способность и предсказуемость пауз). Поэтому можно ожидать, что в будущих версиях .NET на смену описанным выше режимам задержки придут более аспектно-ориентированные цели задержки. В настоящее время планируется четыре такие цели:

- потребление памяти (уровень 1) – паузы могут быть длительными и более частыми, но размер кучи остается небольшим;
- пропускная способность (уровень 2) – паузы непредсказуемы, но случаются не слишком часто (и могут быть длительными);
- баланс между паузами и пропускной способностью (уровень 3) – паузы более предсказуемы и более частые. Самая длинная пауза короче, чем для уровня 1;
- короткие паузы (уровень 4) – паузы более предсказуемы и более частые. Самая длинная пауза короче, чем для уровня 3.

Как отмечалось в главе 7, в настоящее время (версии .NET Framework 4.7 и .NET Core 2.1) поддерживаются только уровни 1 и 3, да и их использование в среде выполнения и сборщике мусора очень ограничено.

Уровень задержки доступен с помощью конфигурационной рукоятки `GCLatencyLevel`, т. е. задается посредством переменной окружения `COMPlus_GCLatencyLevel`, которая может принимать значения 1 и 3.

Выбор варианта GC

Мы уже многое знаем о различных режимах работы GC и умеем управлять степенью его вмешательства с помощью задания режима задержки. Хотя мы обсуждали плюсы и минусы описанных режимов, четкий ответ на вопрос, какой вариант GC оптимален в конкретном случае, пока еще не был дан.

Простой ответ – используйте режим GC по умолчанию! Во многих случаях этого достаточно, и не нужно забивать голову альтернативами. Но есть у нас и различные рукоятки для тонкой настройки, и бывают ситуации, когда стоит подумать об их использовании. Вот два наиболее типичных исключения:

- веб-приложение, размещенное на сервере, где работает много других приложений, – в таком случае фоновый серверный режим, подразумеваемый по умолчанию, может оказаться чрезмерно ресурсоемким. Это можно исправить, установив параметр `GCHeapCount` или сменив режим работы;
- служба Windows выполняет большой объем работы – в таком случае подразумеваемый по умолчанию режим фоновой рабочей станции может недостаточно хорошо масштабироваться, так что есть смысл заменить его одним из серверных режимов.

Сводка всех доступных режимов с учетом всего изложенного выше приведена в табл. 11.1.

Таблица 11.1. Сводка режимов GC

	Режим рабочей станции		**Серверный режим**	
	Неконкурентный	**Конкурентный**	**Неконкурентный**	**Фоновый**
Использование ЦП	Нет потоков GC	Один поток GC	Количество потоков GC равно количеству видимых логических ядер	Количество потоков GC в два раза больше количества видимых логических ядер
Batch	Да (по умолчанию)	Да (запрещены фоновые GC)	Да (по умолчанию)	Да (запрещены фоновые GC)
Interactive	Да (разрешены фоновые GC)	Да (по умолчанию)	Да (разрешены фоновые GC)	Да (по умолчанию)
LowLatency	Да	Да	Нет	Нет
Sustained-LowLatency	Нет	Да	Нет	Да
GCHeapCount	Нет	Нет	Да	Да
Типичное использование	Много облегченных приложений на одной машине, и все они терпимо относятся к длительным паузам (временем которых при необходимости можно управлять с помощью режима LowLatency)	Интерактивные приложения с жесткими требованиями ко времени реакции (при необходимости временем задержки можно управлять с помощью режимов LowLatency и SustainedLow-Latency)	В настоящее время встречается нечасто. Может рассматриваться как компромисс между ресурсоемким фоновым серверным режимом и фоновым режимом рабочей станции, в котором паузы на GC дольше. О длинных блокирующих GC можно сообщать с помощью уведомлений	Большинство приложений, занимающихся обработкой запросов (веб-приложения, размещенные в IIS, службы Windows)

Сценарий 8.1. Проверка параметров GC

Описание. Мы занимаемся разработкой или сопровождением .NET-приложения. В силу различных причин мы хотим точно знать его текущие параметры в производственной среде – допустим, наблюдаемое поведение дает основания заподозрить неправильную конфигурацию. Конечно, можно было бы заглянуть в конфигурационный файл, но полной уверенности мы при этом не получим. Как мы знаем, параметры, заданные в конфигурационном файле, могут быть переопределены переменными окружения или с помощью реестра. Да и сам файл может

содержать ошибки (например, опечатки). Так почему бы не спросить у самого процесса .NET о его параметрах?

Анализ. Самый простой и быстрый способ узнать параметры процесса с минимальным вмешательством в работу программы – воспользоваться механизмом ETW/LTTNg. Каждый раз, когда начинается и заканчивается сеанс ETW, среда выполнения .NET генерирует диагностические события (которые могут быть интерпретированы с помощью соответствующих инструментов). Нас интересует событие `Microsoft-Windows-DotNETRuntimeRundown/Runtime/Start`. Несмотря на свое название, оно генерируется не только в начале, но и в конце сеанса ETW.

Поэтому нам нужно всего лишь начать и закончить ETW и посмотреть на это событие, которое содержит интересное нам поле `StartupFlags`. Для этого можно воспользоваться, например, программой PerfView – записать очень короткий сеанс и поискать это событие в списке событий (рис. 11.12). Поле `StartupFlags` почти не требует пояснений, а нас интересуют следующие три значения:

- `CONCURRENT_GC` – при запуске среды выполнения была разрешена конкурентная сборка мусора. Если этого значения нет, значит, включена неконкурентная GC;
- `SERVER_GC` – среда выполнения запущена в серверном режиме GC. Если этого значения нет, значит, GC работает в режиме рабочей станции;
- `HOARD_GC_VM` – включено придерживание виртуальной памяти (см. главу 5).

Эти значения могут комбинироваться. Например, если GC работает в фоновом серверном режиме, то будут присутствовать оба флага `CONCURRENT_GC` и `SERVER_GC`, а если в неконкурентном режиме рабочей станции – не будет ни одного.

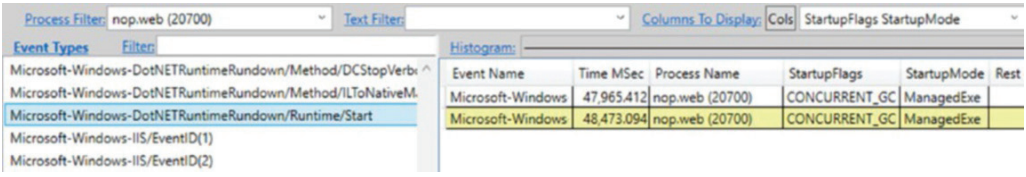


Рис. 11.12 ❖ Параметры CLR в событии Microsoft-Windows-DotNETRuntimeRundown/Runtime/Start

Чтобы еще уменьшить эффект от вмешательства такой проверки, можно воспользоваться замечательной программой *etrace*, написанной Сашей Гольдштейном. Она позволяет управлять сеансами ETW из командной строки и задавать различные фильтры. Нас интересует только одно событие в одном процессе. Поскольку *etrace* начинает сеанс ETW для .NET, будут сгенерированы вышеупомянутые диагностические события, включая `Runtime/Start`. В листинге 11.8 показана соответствующая команда и ее результаты.

Листинг 11.8 ❖ Напечатанный программой etrace список событий ETW от заданных поставщиков с дополнительными фильтрами (в частности, идентификатор процесса)

```
.\etrace.exe --other Microsoft-Windows-DotNETRuntimeRundown --event
Runtime/Start --pid=21316
Processing start time: 30/04/2018 10:21:51
```

```
Runtime/Start [PNAME= PID=21316 TID=14648 TIME=30/04/2018 10:21:51]
ClrInstanceID   = 9
Sku             = 1
BclMajorVersion = 4
BclMinorVersion = 0
BclBuildNumber  = 0
BclQfeNumber    = 0
VMMajorVersion  = 4
VMMinorVersion  = 0
VMBuildNumber   = 30319
VMQfeNumber     = 0
StartupFlags    = 1
StartupMode     = 1
CommandLine     = F:\IIS\nopCommerce\Nop.Web.exe
ComObjectGuid   = 00000000-0000-0000-0000-000000000000
RuntimeDllPath  = C:\Windows\Microsoft.NET\Framework\v4.0.30319\clr.dll
```

Единственное неудобство этого подхода состоит в том, что значение `StartupFlags` печатается в числовом виде, и мы должны сами интерпретировать его, зная значения соответствующего перечисления (листинг 11.9). В примере в листинге 11.8 поле `StartupFlags` равно 1, т. е. выставлен только флаг `CONCURRENT_GC`.

Листинг 11.9 ❖ Перечисление StartupFlags

```
public enum StartupFlags
{
   None = 0,
   CONCURRENT_GC = 0x000001,
   LOADER_OPTIMIZATION_SINGLE_DOMAIN = 0x000002,
   LOADER_OPTIMIZATION_MULTI_DOMAIN = 0x000004,
   LOADER_SAFEMODE = 0x000010,
   LOADER_SETPREFERENCE = 0x000100,
   SERVER_GC = 0x001000,
   HOARD_GC_VM = 0x002000,
   SINGLE_VERSION_HOSTING_INTERFACE = 0x004000,
   LEGACY_IMPERSONATION = 0x010000,
   DISABLE_COMMITTHREADSTACK = 0x020000,
   ALWAYSFLOW_IMPERSONATION = 0x040000,
   TRIM_GC_COMMIT = 0x080000,
   ETW = 0x100000,
   SERVER_BUILD = 0x200000,
   ARM = 0x400000,
}
```

С другой стороны, для веб-приложения ASP.NET, размещенного в IIS, поле `StartupFlags` будет равно `208919` (шестнадцатеричное `33017`), что соответствует таким флагам: `CONCURRENT_GC`, `LOADER_OPTIMIZATION_SINGLE_DOMAIN`, `LOADER_OPTIMIZATION_MULTI_DOMAIN`, `LOADER_SAFEMODE`, `SERVER_GC`, `HOARD_GC_VM`, `LEGACY_IMPERSONATION`, `DISABLE_COMMITTHREADSTACK`.

Сценарий 8.2. Измерение и тестирование производительности различных режимов GC

Описание. Тема режимов работы GC неразрывно связана с вопросом, какой режим оптимален для нашего приложения. С одной стороны, ответ очевиден – режим по умолчанию, скорее всего, достаточно хорош в большинстве случаев. Веб-приложение, размещенное на сервере? Фоновый серверный режим GC. Интерактивное приложение с пользовательским интерфейсом? Фоновый режим рабочей станции. Редко есть смысл выключать конкурентный режим. С другой стороны, не бывает двух одинаковых приложений, и нет никакой гарантии, что режим по умолчанию для данного приложения оптимален. И тогда остается единственный выход – просто измерить эффект различных опций.

Но как его измерить? Какие инструменты использовать? На что смотреть? Именно эти вопросы мы и разберем в данном сценарии. Мы будем анализировать уже известное нам веб-приложение `nopCommerce`. Но не следует придавать слишком большое значение результатам – они значимы только для этого приложения на текущем этапе его развития. Не переносите вытекающие из нашего анализа выводы на свое приложение. Цель этого сценария – показать, как выполняется подобный анализ, а уж применить полученные знания к своему приложению – ваше дело. Мы увидим также типичные подводные камни, на которые можно натолкнуться при интерпретации результатов измерений.

Анализ. Сначала надо понять, как измерить эффект задания различных параметров GC. Этот вопрос уже обсуждался в разделе о приостановке и накладных расходах GC. Тестируемое приложение nopCommerce работает на платформе Windows. Поэтому, чтобы получить полную картину, мы будем измерять следующие аспекты:

- накладные расходы GC с помощью:
  - таблицы GC Rollup By Generation из отчета GCStats в Perf View;
- обработанные данные в формате CSV, экспортированные из таблицы Individual GC Events из отчета GCStats в PerfView, – чтобы вычислить перцентили времени приостановки (столбец Pause MSec) и накладные расходы ЦП (столбец % GC);
- потребление памяти. Для этого будем использовать:
  - обработанные данные в формате CSV, экспортированные из таблицы Individual GC Events из отчета GCStats в PerfView, – чтобы вычислить размер управляемой кучи с помощью столбца After MB (можно было бы также использовать счетчик производительности `/Память .NET CLR/Байт во всех кучах` (`use/.NET CLR Memory/#Bytes in all Heaps`), дающий примерно такую же точность);
  - частный рабочий набор процесса, полученный от диспетчера задач (можно было бы также использовать счетчик производительности `/Процесс/Рабочий набор – частный` (`/Process/Working Set – Private`));
- взгляд со стороны приложения:
  - данные о времени реакции, взятые из сводного отчета JMeter;
  - обработанные данные в формате CSV из Response Times Percentiles (Перцентилей времени ответа), полученные JMeter.

Из-за необходимости обрабатывать все эти данные измерение и тестирование становятся довольно утомительным делом. Процедура в основном ручная, поскольку хороших инструментов, автоматизирующих объединение и обработку результатов, не существует. Если найдете что-то в этом роде, пользуйтесь! Тем не менее я горячо рекомендую провести эти измерения от начала до конца, иначе эксперимент будет неполным, и выводы могут оказаться ложными.

Тестирование включает следующие шаги:

- прогнать с помощью JMeter нагрузочный тест, имитирующий типичную нагрузку, создаваемую пользователями (как всегда, позаботьтесь о повторяемости начальных условий: перезапустите пул приложений, немного прогрейте кеш, закройте посторонние фоновые приложения и т. д.);
- сразу же запустить сеанс ETW из PerfView – очень простого, создающего минимальную дополнительную нагрузку. Достаточно просто выбрать режим .NET;
- дать нагрузочному тесту поработать заданное время;
- остановить все и начать анализ – сюда входит построение таких графиков, как показано ниже. Возможно, для этого придется поработать с CSV-данными в Excel (или другом подобном инструменте), но эти тривиальные аспекты для краткости опущены.

Основное достоинство этого подхода – очень низкая степень вмешательства. Тест можно запустить в любое время, даже в производственной среде. Можно даже обойтись без нагрузочного тестирования, достаточно понаблюдать за периодами с похожей пользовательской активностью (время дня, недели, месяца…), если мы уверены, что условия действительно повторяемые.

У такого рода измерений есть еще один аспект, упомянутый в главе 3, – остерегайтесь средних! Среднее – это статистическая величина, которая дает иллюзию ценной информации, но на самом деле может затушевывать важные факты. Если, к примеру, размер частного рабочего набора изменяется слабо, то среднего может быть достаточно. Но для таких существенных параметров, как время реакции приложения (или, в нашем случае, длительность приостановки на сборку мусора), среднего зачастую не хватает.

Для ключевых метрик по-настоящему ценную информацию дают перцентили. Графики перцентилей создаются для длительности приостановок на сборку мусора и для времени реакции приложения. При этом используются данные, экспортированные в формате CSV. Перцентили напрямую сопоставляются с бизнес-требованиями; например, мы хотим, чтобы для 99 % пользователей наблюдаемое время реакции было меньше 2 секунд, а для 99,99 % – меньше 10 секунд. Такие перцентили вычисляются непосредственно по наблюдаемым данным – выборкам, полученным от ETW и JMeter, – путем ручной обработки в Microsoft Excel. Если мы можем позволить себе большую степень вмешательства, в т. ч. изменение кода приложения, то можем воспользоваться замечательной библиотекой HdrHistogram.NET (https://github.com/HdrHistogram/HdrHistogram.NET), которая вычисляет перцентили внутри приложения.

В этом сценарии мы попытаемся ответить на вопрос, какая из четырех конфигураций GC оптимальна в рассматриваемом приложении:

- неконкурентный режим рабочей станции;
- фоновый режим рабочей станции;

- неконкурентный серверный режим;
- фоновый серверный режим.

Разумеется, «оптимальность» должна определяться бизнес-требованиями. Это может быть время реакции, прописанное в соглашении об уровне обслуживания (SLA), потребление ресурсов (процессора, памяти) или еще какие-то показатели. Отметим, что накладные расходы GC сами по себе не являются бизнес-требованием. Можете ли вы представить себе руководство компании, которое требует, чтобы процент времени, проведенного в GC, был меньше 10? Но в этом сценарии мы увидим, как накладные расходы влияют на приложение в целом.

Перед каждым тестом в конфигурационном файле устанавливаются нужные параметры среды выполнения. Для каждого режима было выполнено несколько тестов, чтобы минимизировать случайные воздействия внешних факторов.

Сначала обсудим накладные расходы, связанные с процессором. На рис. 11.13 показано несколько следствий из полученных результатов:

- сборка мусора в эфемерных поколениях немного быстрее в обоих вариантах серверного режима;
- полная сборка мусора сопряжена с немного меньшими накладными расходами в обоих конкурентных вариантах.

Это подводит к выводу о том, что оптимальным выбором был бы фоновый серверный режим GC. Однако измеренные различия с точки зрения накладных расходов ЦП не потрясают воображение, можно сказать, что поведение во всех режимах примерно одинаково. Но чтобы подтвердить этот вывод, мы должны были произвести детальные измерения. Анализ данных ETW выполнялся с привлечением средства обработки данных (типа Excel), чтобы усреднить результаты (и с помощью гистограммы дополнительно проверялось, что распределение не является многомодальным).

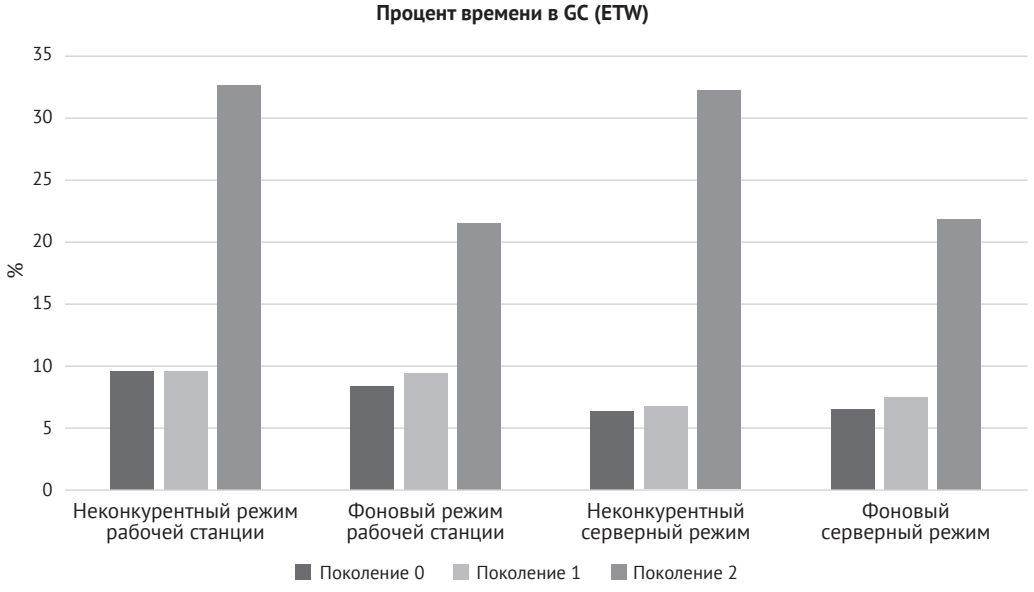


Рис. 11.13 ❖ Процент времени в GC (для каждого поколения)

Если бы мы использовали счетчик производительности % времени в GC, то получили бы сбивающие с толку результаты – значения для обоих режимов рабочей станции были бы гораздо больше, чем для серверных режимов. Обратившись к рис. 11.1, вы вспомните, что % времени в GC – это время GC по сравнению со временем в предыдущей GC. В серверном режиме время, проведенное в GC, мало, но обработка производится параллельно для нескольких управляемых куч (несколькими ядрами). Таким образом, хотя время меньше, общее потребление процессора примерно такое же, но счетчик % времени в GC этого не показывает. Это важное наблюдение. Счетчик % времени в GC следует рассматривать вместе с режимом GC – в режиме рабочей станции высокие значение более терпимы, чем в серверном режиме. Но, как уже было сказано, вообще лучше использовать данные ETW, а не счетчики производительности.

Потребление памяти в разных режимах GC различается (рис. 11.14). В обоих конкурентных (фоновых) режимах управляемая куча заметно больше, чем в неконкурентных, – это подтверждает уже отмеченную выше более сильную фрагментацию из-за частых неуплотняющих фоновых сборок мусора. Да и размеры всего рабочего набора сильно зависят от режима. Этот размер меньше в простых режимах – в неконкурентном режиме рабочей станции часто удается уплотнять и без того небольшие сегменты. На другом конце спектра находится самый сложный вариант – фоновый серверный режим, в котором создаются самые большие сегменты и образуется фрагментация и плавающий мусор. Если потребление памяти для вас – самая важная метрика, то эти данные помогут принять решение.

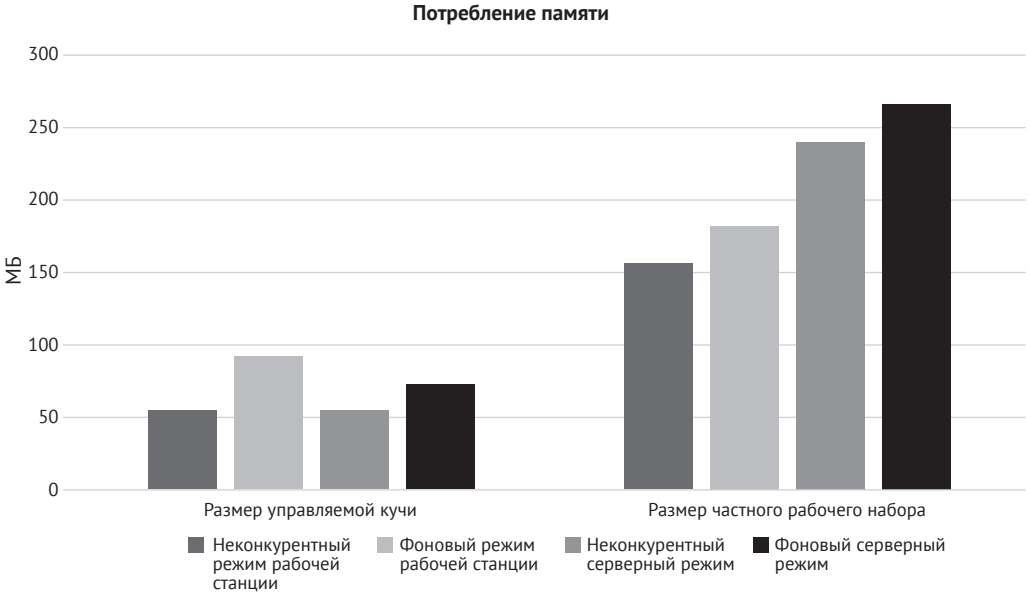


Рис. 11.14 ❖ Потребление памяти

Более интересной может оказаться информация о паузах в каждом режиме GC, предпочтительно для каждого выбранного поколения. Эти данные также не расходятся с ожиданиями (рис. 11.15). В обоих эфемерных поколениях сборка мусора производится очень быстро вне зависимости от режима. Но с полной сборкой все

обстоит иначе. Здесь с большим отрывом проигрывает неконкурентный режим рабочей станции – весь мусор убирается в одном потоке, который блокирует все остальные. Неконкурентный серверный режим быстрее, потому что это делается параллельно в нескольких управляемых кучах. Но все равно заметно медленнее, чем в обоих конкурентных режимах.

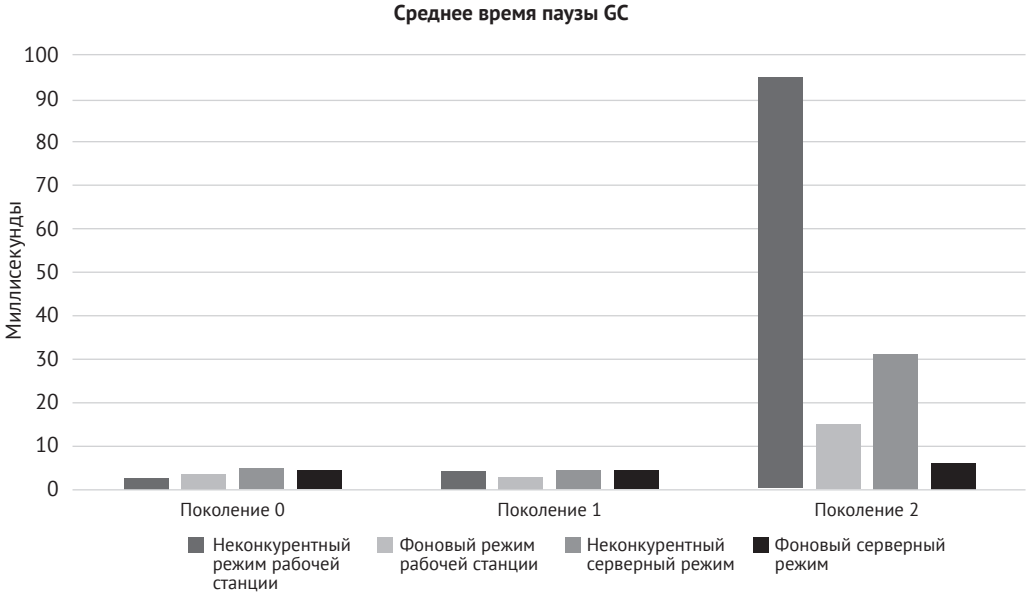


Рис. 11.15 ❖ Среднее время паузы для каждого поколения

Но, как было отмечено выше, среднее – недостаточно точный показатель для столь интересных измерений. Как это ни удивительно, при рассмотрении перцентилей (рис. 11.16) фоновый режим рабочей станции оказывается самым лучшим, а неконкурентный режим рабочей станции – самым худшим (с резким ухудшением в области перцентилей, больших 99). Вот так дотошно следует анализировать длительность паузы в приложениях. Измеряйте – и, возможно, результаты вас удивят!

Но, как уже было сказано, накладные расходы GC (включая и паузы) – лишь основа для куда более ценных бизнес-ориентированных показателей. Как наши тесты выглядят с точки зрения приложения? Удивительно, но среднее время отклика в подготовленном сценарии настолько велико, что почти нивелирует различные настройки (рис. 11.17). В большинстве конфигураций приложение обрабатывало примерно одинаковое количество запросов в единицу времени (но все-таки ориентированный на пропускную способность конкурентный серверный режим показал чуть лучшие результаты). Среднее время отклика тем меньше, чем более «сложный» вариант GC выбран, но различия не очень велики. Это специфика тестируемого приложения. Если бы время отклика было существенно меньше, то влияние GC стало бы более ощутимым.

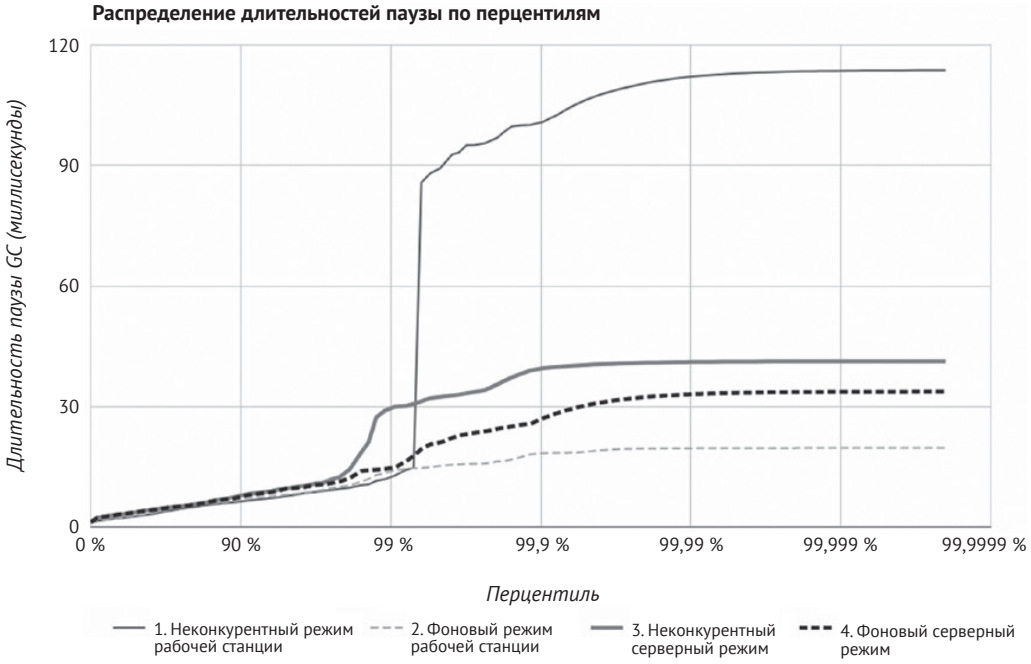


Рис. 11.16 ❖ Перцентили длительности паузы

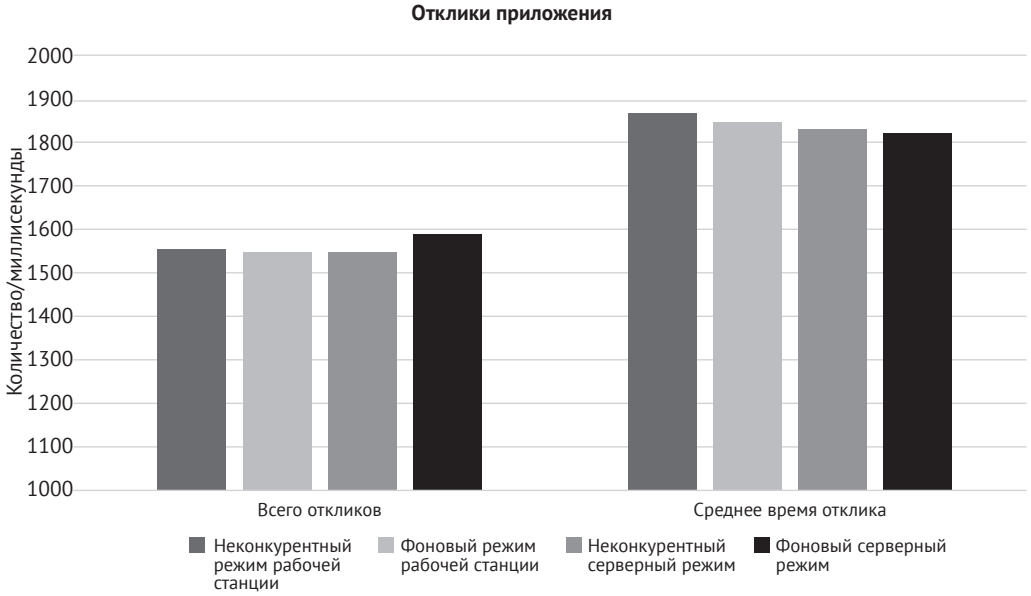


Рис. 11.17 ❖ Количество откликов и среднее время отклика

Но средних величин недостаточно, поэтому рассмотрим перцентили времени отклика (рис. 11.18). Эти данные подтверждают тот факт, что влияние настроек GC пренебрежимо мало. Но это отнюдь не значит, что весь сценарий не имеет смысла. Напротив! Это показывает, как важно измерять не только синтетический показатель `% времени в GC` или длительность паузы, но – и это самое главное – их влияние на работу приложения, на те показатели, которые важны для реальных пользователей.

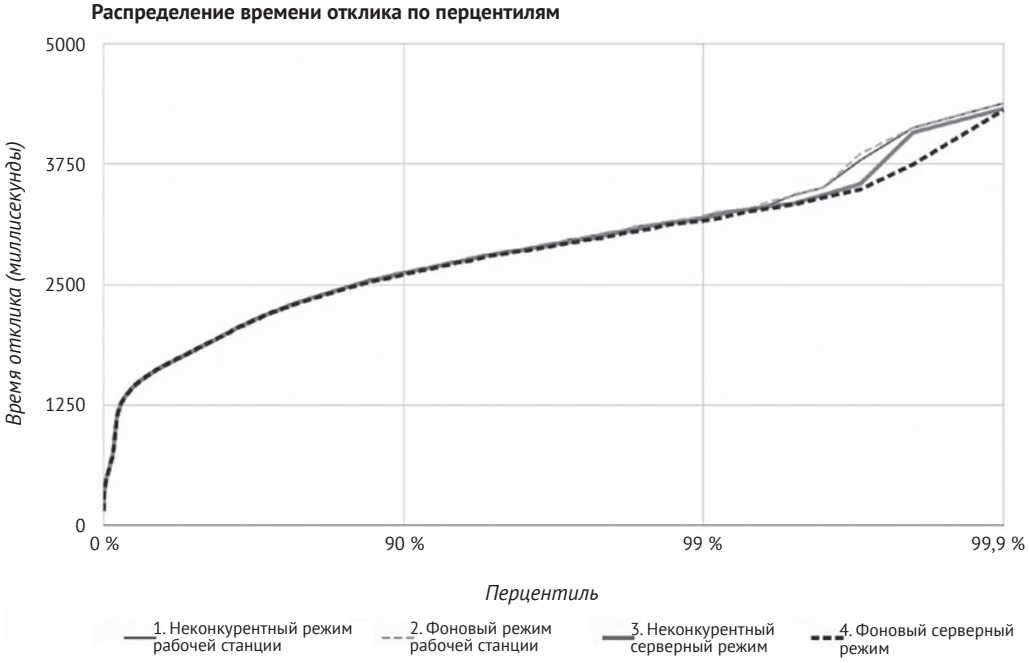


Рис. 11.18 ❖ Перцентили времени отклика

Мы пришли к выводу, что в нашем случае лучше использовать один из двух конкурентных режимов GC. Напомним, что наши выводы верны при определенных предположениях о генерируемой пользователями нагрузке и о среде (количестве процессорных ядер, объеме памяти, других работающих приложениях). Поэтому очень важно выполнять такие тесты в среде, максимально приближенной к производственной, а не на персональном компьютере, на котором ведется разработка.

> В нашем сценарии рассматривается веб-приложение, для которого порядок нагрузочного тестирования понятен. Но автоматизированное тестирование применимо и к настольным, и к мобильным приложениям. Кроме того, если логика приложения удачно распределена по уровням (как, например, при применении технологии MVVM), то можно протестировать только логический уровень, раскрываемый через API. Отказу от тестирования производительности не может быть никаких оправданий!

Для краткости аналогичные тесты различных режимов задержки опущены. Процедура точно такая же. И выводы не должны расходиться с ожиданиями. Но

что касается вашего приложения, то только измерение ответит на вопрос, имеет ли смысл изменять режимы задержки.

Резюме

В этой главе мы узнали о различных способах конфигурирования GC в .NET. Мы разобрались в различиях между режимом рабочей станции и серверным режимом с точки зрения реализации и использования на практике. Мы также узнали о том, что такое неконкурентная и конкурентная сборка мусора, и о том, что последняя теперь называется фоновой. Мы кратко описали реализацию таких интересных механизмов, как конкурентная пометка и очистка.

Мы завершили эту главу рассуждениями о том, какой режим выбрать: серверный или рабочей станции, конкурентный или неконкурентный. С одной стороны, знания о различных доступных режимах, похоже, широко распространены. С другой – мы часто вообще не задумываемся о том, чтобы изменить режим по умолчанию. То, что настройки по умолчанию так хорошо работают, – заслуга команды разработчиков .NET, и обычно менять что-то не возникает необходимости.

Но всегда будут существовать ситуации, когда настроек по умолчанию недостаточно. Поэтому в последнем сценарии подробно описано, как принимать решения о настройках на основе результатов тщательного тестирования производительности.

В двух приведенных ниже правилах подведены итоги этой главе. А следующая глава посвящена важному механизму, связанному со временем жизни объекта, – финализации.

Правило 23: выбирайте режим GC обдуманно

Применимость. Носит общий характер, умеренно популярно. Очень важно в коде, требующем высокой производительности.

Обоснование. В главе 8 мы узнали, что есть различные режимы и параметры сборки мусора. Мы можем управлять важнейшими параметрами GC: количеством куч и потоков GC, агрессивностью и т. д. Как правило, параметров по умолчанию достаточно. Но необходимо знать об имеющихся альтернативах и о том, как правильно выбрать оптимальную.

Как применять. Начинать нужно с главного варианта, который адекватно отражает характеристики приложения: является оно серверным или настольным, важна ли длительность пауз и т. д. Это несложная работа, поскольку вы, надо полагать, знаете общие характеристики своего приложения. С другой стороны, у каждого режима GC есть свои плюсы и минусы с точки зрения потребления процессора и памяти. Они могут изменять характеристики производительности приложения. Без измерений трудно сказать, какой режим оптимален именно для вашего приложения. Поэтому если вы относитесь к производительности серьезно, измеряйте и сравнивайте. Применение правила 5 «Измеряйте GC как можно раньше» и правила 6 «Подвергайте программу измерениям» может помочь в этом, особенно в предпроизводственной среде (и в какой-то мере даже в про-

изводственной). Выполняя тесты, не забывайте о правильной методике, особенно это касается распределения результатов измерений по перцентилям.

Иллюстрирующие сценарии: 8.1, 8.2.

Правило 24: помните о режимах задержки

Применимость. Носит общий характер, довольно редко. Важно в коде, требующем высокой производительности.

Обоснование. На агрессивность GC можно повлиять не только с помощью режимов сборки мусора, но и с помощью режимов задержки. Они управляют тем, насколько сборщик склонен выполнять блокирующую сборку (приводящую к нежелательным паузам). Это позволяет определить нужный баланс между отзывчивостью приложения (блокирующие паузы короткие) и потреблением памяти (сборка по большей части фоновая и неуплотняющая). Поэтому режимы с короткой задержкой чаще всего используются в интерактивных приложениях, когда требуется дополнительный контроль над отзывчивостью пользовательского интерфейса, – как правило, в течение коротких промежутков времени, когда время отклика должно быть минимально (например, во время печати на клавиатуре). В некоторых серверных приложениях, например трейдерских, также используется режим `SustainedLowLatency`, поскольку важно, чтобы во время торговых периодов их работу не прерывала полная блокирующая сборка, если памяти пока достаточно.

Как применять. Режим задержки устанавливается в самой программе. В этой главе были описаны различные способы и связанные с ними паттерны. Режим низкой задержки всегда устанавливается на определенное время – он тем короче, чем жестче требования. На одном конце находится режим `SustainedLowLatency`, в котором программа может работать довольно долго, поскольку запрещены только полные блокирующие сборки мусора. А на другом конце – области без GC, в которых любая сборка мусора запрещена. Кроме того, мы можем динамически переключаться между конкурентными и неконкурентными вариантами. Если мы хорошо понимаем, как пользователи работают с приложением, то сможем лучше оптимизировать потребление памяти и процессора. Но такая точная настройка в обычном приложении не нужна. Интерес к режиму задержки обычно возникает, только когда требования к производительности близки к предельным.

Иллюстрирующие сценарии: 8.2 (в части предлагаемой методики тестирования).

Глава 12

Время жизни объекта

В предыдущих главах весьма подробно описан процесс автоматического управления памятью в .NET. В главе 6 мы рассказали о том, как объекты создаются, а в главах 7–11 – о том, как освобождается занятая ими память, когда они перестают быть нужными. Но существуют еще и дополнительные механизмы, без знакомства с которыми наши знания были бы неполными. В этой главе мы рассмотрим три из них. Хотя это разные механизмы, которые можно использовать независимо, концептуально они связаны. Все они относятся к одному аспекту – времени жизни объекта.

Речь идет о финализации, уничтожаемых объектах (и очень популярном паттерне Disposable) и слабых ссылках. Из этой главы вам станет ясно, как и почему эти механизмы реализованы и как их использовать. Будут представлены практические сценарии диагностики связанных с ними проблем. Отметим, что все эти механизмы излагаются прежде всего с точки зрения управления памятью. В других книгах есть гораздо более полные описания, с обсуждением всех плюсов и минусов, а также возможных подводных камней. Но это не учебник C#, поэтому общим рассуждениям на тему языка здесь не место.

Финализация и паттерн Disposable тесно связаны с работой с неуправляемым кодом (и механизмом P/Invoke), поэтому значительная часть главы посвящена этой теме. Однако имейте в виду, что оба механизма, а слабые ссылки в особенности, могут использоваться и в обычном управляемом коде, не имеющем никакого отношения к неуправляемым ресурсам, например для протоколирования или кеширования. Так что даже если неуправляемый код и P/Invoke не входят в сферу ваших обычных интересов, все равно рекомендую прочитать эту главу.

Жизненные циклы объекта и ресурса

В управляемом мире все кажется довольно простым. Мы создаем объекты, используем их, а потом, когда они перестают быть нужными, сборщик мусора их удаляет. Нам не важно, что сборка мусора не детерминирована, т. е. мы не знаем точно, в какой момент она произойдет, лишь бы GC не удалял объекты слишком рано, когда мы еще с ними работаем (а этого не произойдет, если только в GC нет очень грубой ошибки). Такое недетерминированное освобождение памяти характерно для отслеживающих сборщиков мусора, определяющих достижимые объекты, к каковым относится и реализованный в .NET.

Все хорошо до тех пор, пока не возникнет необходимость выполнить некоторое действие, когда объект перестает быть нужным, – это называется *финализацией*. В этой ситуации недетерминированный характер GC внезапно становится проблемой – попросту не существует места, где разработчик мог бы разместить соответствующий код. Ведь с точки зрения структуры кода есть точно определенный момент создания объекта (конструктор), но нет момента его уничтожения.

В управляемых средах выполнения типа .NET предусмотрен специальный механизм финализации, в т. ч. точно определенное место, в котором программист может написать код, исполняемый, когда объект становится мусором. Большая часть этой главы как раз и посвящена финализации. Поскольку она неразрывно связана с недетерминированной природой сборки мусора, ее часто называют *недетерминированной финализацией* – она обязательно произойдет, но неизвестно когда.

Иногда необходима также *детерминированная финализация* – чтобы можно было явно предпринять действие, когда мы знаем, что объект уже не будет использоваться. .NET предлагает контракт в форме интерфейса `IDisposable` как раз на такой случай. В этой главе мы рассмотрим и его тоже.

> Помимо названий недетерминированная и детерминированная финализация, иногда встречаются названия *неявная* и *явная очистка.*

Имейте в виду, что концептуально финализация напрямую не связана с механизмами сборки мусора. Вне всякого сомнения, это НЕ сборка мусора как таковая, хотя есть разработчики, которые так думают. Финализация просто дает побочный эффект – мы можем выполнить некоторое действие, когда объект становится недостижимым или просто больше не нужен. Но ни финализатор (о чем, вероятно, известно программистам на C#), ни интерфейс `IDisposable` не отвечают за освобождение памяти, занятой ненужным объектом! Мне несколько раз доводилось слушать во время собеседований при приеме на работу утверждения о том, что метод `Dispose` освобождает память объекта. Надеюсь, что, прочитав предыдущие главы, вы понимаете, что это не так.

Но почему вообще нужен механизм финализации? В полностью управляемой среде необходимости в нем не возникало бы. Все управляемые объекты ссылались бы друг на друга, а управление получающимся графом объектов брал бы на себя сборщик мусора. Если кто-то удаляет объект (например, присваивая `null` последней ссылке на него), то отслеживающий сборщик мусора позаботится об удалении всех связанных с ним объектов, недостижимых из других мест. Ответственность за удаление объектов, которыми владел удаленный объект, в неуправляемом мире (т. е. в C/C++) обычно возлагается на деструктор.

В управляемом мире финализация чаще всего полезна, когда объект удерживает какие-то ресурсы, неконтролируемые сборщиком мусора и средой выполнения. Это разного рода описатели, дескрипторы и другие связанные с системными ресурсами данные, которые необходимо освобождать явно. Чем сильнее окружение зависит от кооперации с неуправляемыми ресурсами, тем важнее роль финализации. Среда .NET с самого начала проектировалась как «дружественная к неуправляемому коду». Как уже отмечалось, одной из целей проектирования была возможность взять обычный код на C++ и после минимальных изменений

скомпилировать его как .NET-программу (очень напоминает современный язык C++/CLI). Многие популярные API опираются на неуправляемые ресурсы (файлы, сокеты, растровые изображения и т. д.). Поэтому финализация существовала в головах разработчиков .NET изначально – как в форме детерминированного контракта `IDisposable`, так и в форме недетерминированной финализации.

> JVM, чрезвычайно популярная конкурирующая управляемая среда, уделяет куда меньше внимания недетерминированной финализации. Она считается ненадежной, проблематичной и привносящей ненужные накладные расходы на сборку мусора. На самом деле она настолько непопулярна, что начиная с Java 9 объявлена нерекомендуемой. Вместо этого вот уже много лет предпочтительными считаются различные методы детерминированной финализации. Для этого предоставляется явный метод очистки, который разработчик обязан вызвать, когда объект перестает быть нужным (чаще всего использование объекта оборачивается блоком `try-finally`). Это напоминает хорошо знакомый паттерн `IDisposable` в .NET.
>
> Вместо нерекомендуемого метода `java.lang.Object.finalize` для недетерминированной финализации предлагается класс `java.lang.ref.Cleaner`, который управляет ссылками из объекта типа `java.lang.ref.PhantomReference` и производит для них соответствующие действия по очистке. Фантомные ссылки ставятся в очередь, после того как сборщик мусора определит, что объекты, на которые они указывают, можно было бы освободить (так что этот механизм также недетерминированный).

Поскольку оба мира – управляемый и неуправляемый – сосуществуют, нам приходится думать о двух разных вопросах: управление временем жизни объекта и управление ресурсами (неуправляемыми), которыми он владеет. Что касается времени жизни объекта, то управление им лежит на сборщике мусора. С другой стороны, среда выполнения почти ничего не знает о наших неуправляемых ресурсах, так что ответственность за управление ими возлагается на нас, а в помощь предлагаются средства, описанные в этой главе.

Хотя финализация – побочный эффект удаления объекта, мы увидим, что конкретные ее реализации в .NET все-таки влияют на время жизни объекта.

Финализация

То, что обычно называют финализацией в .NET, вообще говоря, является недетерминированной финализацией. Процитируем стандарт ECMA-335: «В определении класса, создающем тип объекта, может присутствовать метод экземпляра (называемый *финализатором*), который вызывается, когда экземпляр класса становится недостижимым». Именно его мы и будем рассматривать в этой части главы – как объявить и использовать финализатор и как он реализован в CLR.

Введение

Для объявления финализатора в C# предусмотрен специальный синтаксис *деструктора* (листинг 12.1). Это код, который вызывается, когда объект перестает быть достижимым и вот-вот будет удален. В нашем примере он служит для того, чтобы закрыть описатель открытого файла (иначе мы рано или поздно достигнем максимально разрешенного количества открытых описателей в системе). В Win-

dows системные ресурсы представляются описателями (handle), которым часто соответствует структура `IntPtr`[1].

Листинг 12.1 ❖ Простой пример использования финализатора в C# (определен в деструкторе)

```
class FileWrapper
{
   private IntPtr handle;
   public FileWrapper(string filename)
   {
      Unmanaged.OFSTRUCT s;
      handle = Unmanaged.OpenFile(filename, out s, 0x00000000);
   }

   // Деструктор
   ~FileWrapper()
   {
      if (handle != IntPtr.Zero)
         Unmanaged.CloseHandle(handle);
   }
```

В C# деструктор – это просто обертка, которая транслируется компилятором в метод, переопределяющий `System.Object.Finalize` (листинг 12.2).

Листинг 12.2 ❖ Определение деструктора на промежуточном языке

```
.method family hidebysig virtual
instance void Finalize () cil managed
{
   .override method instance void [System.Runtime]System.Object::Finalize()
   // ...
}
```

Тот факт, что переопределяется метод `Finalize`, имеет решающее значение. Это контракт между типом и GC – объекты, в которых переопределен метод `Finalize`, называются *финализируемыми* и обрабатываются сборщиком мусора специальным образом.

> Чтобы объявить финализируемый тип в F# или VB.NET, нужно просто переопределить метод `Finalize`. Но в C# это невозможно. При попытке так поступить будет выдана ошибка «Do not override `Object.Finalize`. Instead, provide a destructor» (Не переопределяйте Object.Finalize. Вместо этого реализуйте деструктор). Поэтому синтаксис `~Typename` – единственная возможность. Название «деструктор» неудачно, поскольку, как мы знаем, это понятие не имеет ничего общего с деконструкцией управляемых объектов, а больше связано с управлением ресурсами. Интересно, что поскольку в C++ уже имеется обозначение `~Typename()` для деструктора в смысле C++, то для финализаторов в C++/CLI применяется синтаксис `!Typename()`.
>
> Заметим также, что в MSDN написано: «любая реализация `Finalize` в производном типе обязана вызывать реализацию `Finalize` в базовом типе. Это единственное место в пользовательском коде, где разрешается вызывать `Finalize`». В C# обертка деструктора делает это автоматически, но в других языках нужно об этом помнить.

[1] В Linux ресурсы обычно представляются просто целыми числами.

Мы можем, например, использовать финализаторы для управления дополнительной нагрузкой на память (с помощью методов `GC.AddMemoryPressure` и `GC.RemoveMemoryPressure`), создаваемой потребляемым ресурсом (даже если он сам управляемый, но мы знаем, что реализация пользуется какими-то ресурсами). Типичный пример – использование класса растрового изображения `System.Drawing.Bitmap`, который на самом деле представлен описателем системного ресурса, но, очевидно, потребляет дополнительную память (листинг 12.3).

Листинг 12.3 ❖ Пример использования финализаторов для управления дополнительной нагрузкой на память

```
class MemoryAwareBitmap
{
   private System.Drawing.Bitmap bitmap;
   private long memoryPressure;
   public MemoryAwareBitmap(string file, long size)
   {
      bitmap = new System.Drawing.Bitmap(file);
      if (bitmap != null)
      {
         memoryPressure = size;
         GC.AddMemoryPressure(memoryPressure);
      }
   }

   ~MemoryAwareBitmap()
   {
      if (bitmap != null)
      {
         bitmap.Dispose();
         GC.RemoveMemoryPressure(memoryPressure);
      }
   }
   ...
}
```

Однако с финализаторами связаны некоторые ограничения.

- Как уже отмечалось, они выполняются в недетерминированный момент – финализатор будет вызван (почти наверняка, см. ниже), но когда – неизвестно. Это плохо с точки зрения управления ресурсами. Если объект владеет ограниченным ресурсом, то освободить его следует как можно быстрее. Ждать недетерминированной очистки едва ли оптимально. Если нам очень нужно выполнить финализаторы, можно вызвать метод `GC.WaitForPendingFinalizers`. Мы еще несколько раз вернемся к нему.
- Порядок выполнения финализаторов не определен. Даже если один из финализируемых объектов ссылается на другой, не гарантируется, что их финализаторы будут выполнены в каком-то логически осмысленном порядке (например, «подчиненный» раньше «главного» или наоборот). Поэтому нельзя ссылаться на другие финализируемые объекты внутри финализатора, даже если мы «владеем» ими. Неупорядоченное выполнение – осо-

знанное проектное решение, поскольку иногда определить естественный порядок попросту невозможно (например, что делать, если между финализируемыми объектами есть циклические ссылки?). Однако некоторый порядок финализаторов определить все-таки можно – в форме критических финализаторов, описанных ниже. Но из кода финализатора можно ссылаться на обычные управляемые объекты, поскольку гарантируется, что сборка мусора в графе объектов производится после выполнения финализаторов.

- Не определено также, в каком потоке выполняется финализатор. Хотя мы увидим, как он определяется в текущей реализации, стандарт ECMA-335 не содержит никаких требований в этом отношении. Поэтому полагаться на какой-то контекст потока нельзя (это относится и к синхронизации потоков в виде блокировок – из-за отсутствия каких-либо гарантий это могло бы привести к взаимоблокировкам).
- Не гарантируется, что код финализации вообще будет выполнен. Он может быть также выполнен лишь частично, например если какой-то финализатор по ошибке заблокировал выполнение на неопределенно долгое время или процесс завершился преждевременно, не дав GC возможности выполнить финализаторы. Более того, может даже случиться, что финализатор будет выполнен несколько раз из-за возможности воскрешения объекта, описанной далее[1].
- Бросать исключение из финализатора крайне опасно – по умолчанию оно приводит к аварийному завершению процесса. Поскольку код финализатора считается очень важным (например, он может освобождать общесистемные примитивы синхронизации), невозможность выполнить его считается критической ошибкой. Поэтому нужно очень внимательно следить за тем, чтобы не бросать исключений из финализатора.
- Финализируемые объекты создают дополнительную нагрузку на GC, и это может негативно отразиться на производительности приложения в целом, как мы увидим это в разделе, где описывается реализация финализации. Этот механизм требует дополнительной обработки объектов, что, конечно, обходится не бесплатно.

Все сказанное приводит к заключению, что реализация финализаторов – дело сложное, а использование не всегда надежно, так что в общем случае их лучше избегать. Считайте, что это неявная «страховочная сетка» для случаев, когда разработчик не освободил ресурсы предпочтительным способом с помощью явной очистки (например, паттерна Disposable). Мы рассмотрим пример типичного использования этого паттерна, когда будем обсуждать его ниже.

> Процитируем стандарт ECMA-335: «Разрешается определять финализатор для типа значений. Однако этот финализатор будет выполняться только для упакованных экземпляров типа значений». По крайней мере, для среды выполнения .NET Core это утверждение неверно. Среда просто игнорирует финализаторы, определенные в типах значений.

---

[1] Хуже того, при неблагоприятном стечении обстоятельств воскрешение может привести к нескольким одновременным вызовам одного и того же финализатора.

С точки зрения программиста, важно лишь, что финализатор вызывается «в какой-то момент времени», после того как объект стал недостижимым. Но, хотя это детали реализации, было бы полезно знать, когда именно вызываются финализаторы. Возможно два случая:

- в конце сборки мусора – по какой бы причине ни была запущена сборка, в конце процесса вызываются финализаторы для объектов, которые в этой сборке признаны недостижимыми. Имейте в виду, что вызываются только финализаторы объектов из собираемого и более молодых поколений;
- во время операций внутреннего обслуживания CLR – когда среда выполнения выгружает домен приложения и останавливается.

Как уже отмечалось, финализаторы не обязательно связаны с неуправляемыми ресурсами. Можно представить себе и другие применения, например протоколирование времени жизни, как в листинге 12.4. Если по какой-то причине (например, потому что объект очень важный или ресурсоемкий) мы хотим отражать в журнале факты создания и удаления объекта, то самые подходящие места для этого – его конструктор и финализатор.

Листинг 12.4 ❖ Простой пример использования финализатора в C#

```
class LoggedObject
{
   private ILogger logger;
   public LoggedObject(ILogger logger)
   {
      this.logger = logger;
      // ...
      this.logger.Log("Объект создан.");
   }

   // Деструктор
   ~LoggedObject()
   {
      this.logger.Log("Объект уничтожен.");
   }
}
```

Заметим, что даже в таком «не неуправляемом» мире реализация финализатора нетривиальна. В листинге 12.4 финализатор мог бы использовать регистратор (logger), зависимость от которого внедрена через интерфейс. Это означает, что нет никакой гарантии, что конкретный внедренный экземпляр регистратора не окажется финализируемым, и, стало быть, мы столкнемся с проблемой неупорядоченного выполнения финализаторов – внутри нашего финализатора регистратор уже может быть уничтожен.

Как можно уменьшить такую опасность? Есть решения, основанные на процессе код-ревью или автоматизированном статическом анализе кода, – с целью убедиться, что реализации ILogger нефинализируемые или критически финализируемые (вскоре мы поймем, почему это может помочь). Но предпочтительное решение всегда одно и то же – избегать финализации. Если время жизни такого объекта настолько важно, то, скорее всего, будет лучше включить в него паттерн Disposable – тогда момент очистки будет точно определен, а включать средства протоколирования будет гораздо безопаснее.

Проблема ранней сборки корней

Раздельное управление временем жизни объектов и ресурсов может приводить к странным побочным эффектам. Мы уже встречались с ними в листингах 8.13–8.16 из главы 8. Большая их часть связана с техникой ранней сборки корней. Хотя сама по себе эта оптимизация, основанная на JIT-компиляции, замечательна и гарантирует кратчайшее время жизни объектов, в контексте управления ресурсами она иногда приводит к проблемам.

Самый типичный пример такой проблемы – использование потока (stream) для доступа к файлу (листинг 12.5). Если раскомментировать обращения к GC в методе `ProblematicObject.UseMe` (моделирующие сборку мусора, которая может происходить одновременно с выполнением этого метода), то программа прекратит работу из-за необработанного исключения `System.ObjectDisposedException: Cannot access a closed file` (Невозможно получить доступ к закрытому файлу). Это связано с JIT-оптимизацией: в методе `UseMe` весь экземпляр `ProblematicObject` считается недостижимым сразу после последнего использования this[1]. Таким образом, сразу после присваивания потока переменной `localStream` можно ожидать, что будет выполнен финализатор объекта `ProblematicObject`. Но, как мы видим, этот финализатор закрывает поток, так что следующий вызов `ReadByte` закончится неудачно. В этом простом случае ошибку можно исправить, всегда используя сам экземпляр потока, а не локальную переменную (т. е. в последней строке нужно было вернуть `this.stream.ReadByte()`). В этом случае на экземпляр `ProblematicObject` появилась бы ссылка из последней строки метода `UseMe`, поэтому оптимизация с ранней сборкой корней не использовалась бы.

Листинг 12.5 ❖ Проблема из-за того, что финализатор слишком рано освобождает ресурсы

```
class ProblematicObject
{
   Stream stream;

   public ProblematicObject() => stream = File.OpenRead(@"C:\Temp.txt");

   ~ProblematicObject()
   {
      Console.WriteLine("Финализация ProblemticObject");
      stream.Close();
   }

   public int UseMe()
   {
      var localStream = this.stream;
      // Какой-то код, достаточно сложный, для того чтобы этот метод был невстраиваемым
      // (not inlineable) и полностью или частично прерываемым.
      ...
      // Здесь происходит GC, и у финализаторов достаточно времени для выполнения.
      // Это можно смоделировать с помощью следующих вызовов:
      // GC.Collect();
      // GC.WaitForPendingFinalizers();
```

[1] Смотрите описание ранней сборки корней в главе 8.

```
      return localStream.ReadByte();
   }
}

class Program
{
   static void Main(string[] args)
   {
      var pf = new ProblematicObject();
      Console.WriteLine(pf.UseMe());
      Console.ReadLine();
   }
}
```

При использовании P/Invoke можно столкнуться с такими же проблемами, поэтому было предложено несколько способов исправить положение вещей. Для начала расширим код в листинге 12.1, добавив соответствующий метод `UseMe`, но теперь будем вызвать P/Invoke непосредственно (листинг 12.6). Мы получили точно такую же проблему – ранняя уборка экземпляра `ProblematicFileWrapper` запускает выполнение его финализатора, который закрывает используемый описатель файла, хотя в последующем коде этот описатель используется. Вызов `Unmanaged.ReadFile` закончится ошибкой, и метод `UseMe` вернет –1. В нашем примере ошибку легко исправить, используя `this.handle` вместо локальной переменной `hnd`, но такое возможно не всегда – очень часто `IntPtr` не является частью управляемого объекта (а только статической или локальной переменной).

Листинг 12.6 ❖ Проблема из-за того, что финализатор слишком рано освобождает ресурсы (расширение листинга 12.1)

```
public class ProblematicFileWrapper
{
   private IntPtr handle;
   public ProblematicFileWrapper(string filename)
   {
      Unmanaged.OFSTRUCT s;
      handle = Unmanaged.OpenFile(filename, out s, 0x00000000);
   }
   ~ProblematicFileWrapper()
   {
      Console.WriteLine("Финализация ProblematicFileWrapper");
      if (handle != IntPtr.Zero)
         Unmanaged.CloseHandle(handle);
   }

   public int UseMe()
   {
      var hnd = this.handle;
      // Какой-то код      ...
      // Здесь происходит GC, и у финализаторов достаточно времени для выполнения.
      // Это можно смоделировать с помощью следующих вызовов:
      //GC.Collect();
      //GC.WaitForPendingFinalizers();
      byte[] buffer = new byte[1];
```

```
      if (Unmanaged.ReadFile(hnd, buffer, 1, out uint read, IntPtr.Zero))
      {
         return buffer[0];
      }
      return -1;
   }
}
```

Первое общее решение этой проблемы типично для контроля над ранней сборкой корней – мы можем добавить вызов `GC.KeepAlive(this)` прямо перед оператором `return` в методе `UseMe`. Тем самым мы продлим время жизни объекта, хранящего описатель. Но это решение загромождает код и не слишком изящно.

Как раз для таких задач существует вспомогательная структура `HandleRef`. Это очень простая обертка, которая хранит и описатель, и владеющий им объект. Она обрабатывается маршалером Interop для продления жизни указанного объекта на все протяжении вызова P/Invoke. API таких P/Invoke-вызовов ожидают получить `HandleRef`, а не просто `IntPtr` (листинг 12.7).

Листинг 12.7 ❖ Решение проблемы финализатора с помощью структуры HandleRef
public int UseMe()

```
{
   var hnd = this.handle;
   // Какой-то код ...
   // Здесь происходит GC, и у финализаторов достаточно времени для выполнения.
   // Это можно смоделировать с помощью следующих вызовов:
   //GC.Collect();
   //GC.WaitForPendingFinalizers();
   byte[] buffer = new byte[1];
   if (Unmanaged.ReadFile(new HandleRef(this, hnd), buffer, 1, out uint read, IntPtr.Zero))
   {
      return buffer[0];
   }
   return -1;
}
```

Однако `HandleRef` решает не все проблемы, и особенно это относится к злонамеренной атаке с рециркуляцией ссылок (handle-recycling attack), которую мы скоро обсудим. Так что это довольно старый и нерекомендуемый подход, встречающийся в основном в унаследованном коде (80 % его использования приходится на код Windows Forms и System.Drawing).

Одновременно с `HandleRef` появился класс `HandleCollector`, который реализует семантику подсчета ссылок для описателей – если достигнуто пороговое количество созданных описателей, запускается GC. Он также считается устаревшим и используется очень редко.

> Не используйте классы `HandleRef` и `HandleCollector`. Они описаны здесь только для того, чтобы у вас сложилась полная картина об управлении ресурсами и чтобы вы понимали, в результате чего возник подход на основе класса `SafeHandle`. Даже если вы встретите эти классы в существующем коде, не берите этот код за образец. Безопасные описатели (safe handles), появившиеся в версии .NET Framework 2.0 и описанные в следующем разделе, гораздо лучше.

Критические финализаторы

В силу вышеупомянутых проблем, связанных с финализаторами, в .NET Framework введено несколько более строгое понятие *критических финализаторов*. Это обычные финализаторы с дополнительными гарантиями, и предназначены они для ситуаций, когда код финализатора должен быть выполнен в любом случае, даже при грубой выгрузке домена приложения или аварийном завершении потока. Процитируем MSDN: «Для классов, производных от `CriticalFinalizerObject`, общеязыковая среда выполнения (CLR) гарантирует, что коду критической финализации будет предоставлена возможность выполниться при условии, что финализатор соответствует правилам *CER* (*области ограниченного выполнения*), даже в тех случаях, когда CLR принудительно выгружает домен приложения или аварийно завершает поток».

Для определения критического финализатора содержащий его класс должен наследоваться от класса `CriticalFinalizerObject`. Сам класс `CriticalFinalizerObject` абстрактный, и у него нет реализации (листинг 12.8). Это просто еще один контракт между системой типов и средой выполнения. Среда выполнения принимает меры для того, чтобы выполнение критических финализаторов было возможно при любых обстоятельствах. Например, JIT-компилятор генерирует код критического финализатора заранее, чтобы не попасть в ситуацию, когда для этого не хватит памяти.

Листинг 12.8 ❖ Определение класса критического финализатора (некоторые атрибуты для краткости опущены)

```
public abstract class CriticalFinalizerObject
{
   [ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail)]
   protected CriticalFinalizerObject()
   {
   }
   [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
   ~CriticalFinalizerObject()
   {
   }
}
```

Поскольку неопределенность порядка выполнения финализаторов иногда вызывает проблемы, критические финализаторы предоставляют некоторые дополнительные гарантии и в этой части. Снова процитируем MSDN: «CLR устанавливает слабый порядок на множестве обычных и критических финализаторов: для объектов, освобождаемых сборщиком мусора в одно и то же время, все некритические финализаторы вызываются раньше критических. Например, класс `FileStream`, который хранит данные в объекте типа `SafeHandle` и наследуется от класса `CriticalFinalizerObject`, может выполнить стандартный финализатор для сброса всех буферизованных данных на диск».

Вам редко придется определять типы, которые непосредственно наследуются от класса `CriticalFinalizerObject`. Чаще вы будете наследоваться от типа `SafeHandle` (производного от `CriticalFinalizerObject`). Но поскольку `SafeHandle` тесно связан с финализацией и паттерном Disposable, мы опишем его после знакомства и с тем, и другим.

Внутреннее устройство финализации

Узнав о назначении финализаторов, посмотрим, как они реализованы в текущей версии среды выполнения. Пока что я описывал их в основном с точки зрения семантики – для чего предназначены, какие гарантии дают и каковы их ограничения. Но полезно понимать, что их реализация вносит дополнительные недостатки, которые мы и рассмотрим в этом разделе.

Прежде всего, как уже упоминалось в главе 6, если в типе определен финализатор, то выбирается медленный способ выделения памяти – и это главное препятствие на пути производительности, обязанное своим появлением лишь тому, что метод `Finalize` переопределен.

> Если вы хотите исследовать, как выглядит в исходном коде CoreCLR медленный способ выделения памяти из-за финализации, начните с реакции JIT-компилятора на код операции `CEE_NEWOBJ`, реализованной в импортере JIT (`importer.cpp:Compiler::impImportBlockCode`). В методе `CEEInfo::getNewHelperStatic` проверяется, определен ли в типе финализатор. Если да, выбирается метод `CORINFO_HELP_NEWFAST`, который при запуске среды выполнения присваивается функции `JIT_New`. Этот метод в конечном итоге вызывает метод `GCHeap::Alloc` или `GCHeap::AllocLHeap`, который заканчивается макросом `CHECK_ALLOC_AND_POSSIBLY_REGISTER_FOR_FINALIZATION`. Макрос вызывает метод `CFinalize::RegisterForFinalization`, отвечающий за описанный ниже учет финализируемых объектов. Как уже отмечалось, хотя ECMA-335 говорит, что для упакованных типов значений финализаторы вызываются, на самом деле это уже не так. Когда JIT-компилятор решает, какая функция будет представлять помощника `CORINFO_HELP_BOX`, существование финализатора не принимается во внимание и чаще всего используется быстрый, написанный на ассемблере метод `JIT_BoxFastMP_InlineGetThread`, который реализует выделение памяти простым сдвигом указателя.

GC должен знать обо всех финализируемых объектах, чтобы вызывать их финализаторы, когда они перестают быть доступными. Эти объекты хранятся в очереди финализации. Иными словами, в каждый момент времени очередь финализации содержит все еще живые финализируемые объекты. Большое количество объектов в очереди само по себе не означает ничего плохого – просто в настоящий момент существует много объектов с финализаторами.

На этапе пометки сборщик мусора смотрит, есть ли среди объектов в очереди финализации мертвые. Если да, их еще нельзя удалять, поскольку необходимо сначала выполнить финализаторы. Поэтому такие объекты перемещаются в другую очередь – *fReachable*. Название означает, что это очередь объектов, достижимых в силу финализации[1], т. е. таких, которые остаются достижимыми только потому, что ожидают финализации. Если такие объекты найдены, то GC сообщает выделенному потоку финализации о том, что для него есть работа.

Поток финализации – это еще один поток, созданный средой выполнения .NET. Он удаляет объекты из очереди fReachable по одному и вызывает их финализаторы. Это происходит после того, как GC возобновил управляемый поток, потому что коду финализатора может понадобиться создавать объекты. Поскольку единственный корень такого объекта удален из очереди fReachable, при следующем

[1] В литературе также встречается пояснение «достижимых для финализатора», но приведенная трактовка лучше согласуется с принятым в .NET соглашением об именовании.

запуске сборки мусора в поколении, содержащем этот объект, сборщик увидит, что он недостижим, и освободит занятую им память.

Обратите внимание на один из самых крупных накладных расходов, связанных с финализацией: по умолчанию финализируемый объект переживает по крайней мере одну сборку мусора. А если он переходит в поколение 2, то будет удален при сборке в поколении 2, а не 1.

Кроме того, очередь fReachable рассматривается как корень на этапе пометки (о чем упоминалось в главе 8), поскольку потоку финализации может не хватить времени для обработки всех объектов между сборками мусора. В результате финализируемые объекты оказываются подвержены «кризису среднего возраста» – они могут некоторое время оставаться в очереди fReachable, занимая место в поколении 2 только потому, что ожидают финализации.

Для управления асинхронной природой обработки финализации служит метод `GC.WaitForPendingFinalizers`. Он блокирует вызывающий поток до тех пор, пока не будут обработаны все объекты из очереди fReachable (т. е. пока не будут вызваны все финализаторы). В качестве побочного эффекта после его вызова все «достижимые в силу финализации» объекты становятся недостижимыми, так что следующий запуск GC уберет их.

Это подводит нас к очень популярному паттерну «полной явной сборки мусора» (листинг 12.9), который используется, если мы хотим освободить максимальное количество памяти. Будучи на первый взгляд бессмысленным, он на самом деле вполне разумен:

- сначала явная полная блокирующая сборка мусора находит набор объектов из очереди fReachable;
- поток ждет, пока GC обработает все объекты в очереди fReachable и сделает их недостижимыми;
- вторая явная полная блокирующая сборка мусора освобождает занятую ими память.

Листинг 12.9 ❖ Паттерн явной сборки мусора, учитывающий корни финализации

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

Очевидно, что если другие потоки создают финализируемые объекты, пока выполняется метод `WaitForPendingFinalizers`, то при втором вызове `GC.Collect` могут быть найдены новые fReachable-объекты. Это приводит к парадоксу – похоже, мы никогда не сможем освободить память полностью (по крайней мере, не прибегая к агрессивной блокировке всех потоков, которые могут выделять память). Это хорошо видно в реализации метода `GC.GetTotalMemory`, который возвращает полное число байтов, занятых живыми объектами (листинг 12.10). Если мы хотим получить точное значение, то должны передать `true` в качестве его аргумента `forceFullCollection`. Тогда он попытается получить истинное множество объектов, которые живы в данный момент, вызвав полную сборку мусора и ожидание финализации несколько раз – пока разница между новым и предыдущим результатами не станет менее 5 % (при этом еще задается максимальное число итераций, чтобы не зациклиться).

Листинг 12.10 ❖ Реализация метода GC.GetTotalMemory

```
[System.Security.SecuritySafeCritical] // auto-generated
public static long GetTotalMemory(bool forceFullCollection) {
   long size = GetTotalMemory();
   if (!forceFullCollection)
      return size;
   // Если принудительно запускается полная сборка, то мы выполняем
   // финализаторы всех существующих объектов и производим сборку, пока
   // значение не стабилизируется. Значение считается "стабильным", если либо
   // отличается от предыдущего не более чем на 5%, либо цикл уже выполнен
   // x раз (мы не хотим оставаться в этом цикле вечно).
   int reps = 20; // количество итераций
   long newSize = size;
   float diff;
   do {
      GC.WaitForPendingFinalizers();
      GC.Collect();
      size = newSize;
      newSize = GetTotalMemory();
      diff = ((float)(newSize - size)) / size;
   } while (reps-- > 0 && !(-.05 < diff && diff < .05));
   return newSize;
}
```

Можете позаимствовать этот код, чтобы добиться «еще более полной явной сборки мусора» (или просто вызвать `GC.GetTotalMemory(true)`, при условии что его реализация не изменится). Можно еще повысить агрессивность, если присваивать параметру `GCSettings.LargeObjectHeapCompactionMode` значение `GCLargeObjectHeapCompactionMode.CompactOnce` перед первыми или даже перед каждым обращением к `GC.Collect`.

Стоит иметь в виду, насколько дорогостоящим может оказаться обращение к методу `GC.GetTotalMemory` с аргументом `forceFullCollection = true`. Если память выделяется очень динамично, он может запускать полную блокирующую сборку мусора 20 раз! Так что в большом динамичном приложении будьте готовы к тому, что ждать результата придется больше секунды.

Есть еще одна деталь, которую мы пока не объяснили, – как уже было сказано, во время сборки мусора рассматриваются только финализируемые объекты в выбранном и более молодых поколениях. Например, если сборка мусора производится в поколении 1, то в очереди финализации будут рассмотрены только объекты из поколений 0 и 1 и в очередь fReachable будут перемещены те из них, что оказались недостижимыми.

Поэтому очередь финализации должна знать, какому поколению принадлежит рассматриваемый в данный момент объект. Можно было бы подумать, что в процессе обработки очереди финализации для каждого объекта проверяется граница адресов поколения. Но напомним, что поколение 2 и LOH могут занимать несколько сегментов, поэтому такая проверка может обойтись дорого, занимая драгоценное время. Поэтому сама очередь финализации организована по поколениям – адреса объектов из разных поколений хранятся в разных сегментах. Поэтому во время конкретной сборки мусора проверяются только относящиеся

к ней сегменты. И да – приходится перемещать адреса объектов между сегментами, если сами объекты переводятся в старшее поколение или остаются/возвращаются в младшее! Не кажется ли вам, что это еще один вид накладных расходов, связанный с финализацией?

И очередь финализации, и очередь fReachable в настоящее время реализованы в виде одного простого массива адресов объектов (рис. 12.1) – я буду называть его «массивом финализации». Логически он разделен на три области:

- часть финализации – дополнительно разделена на четыре сегмента, по одному для каждого из трех поколений и для LOH;
- часть fReachable – дополнительно разделена на сегменты, содержащие адреса объектов с критическими и обычными финализаторами;
- свободная часть – для расширения вышеупомянутых сегментов по мере их роста.

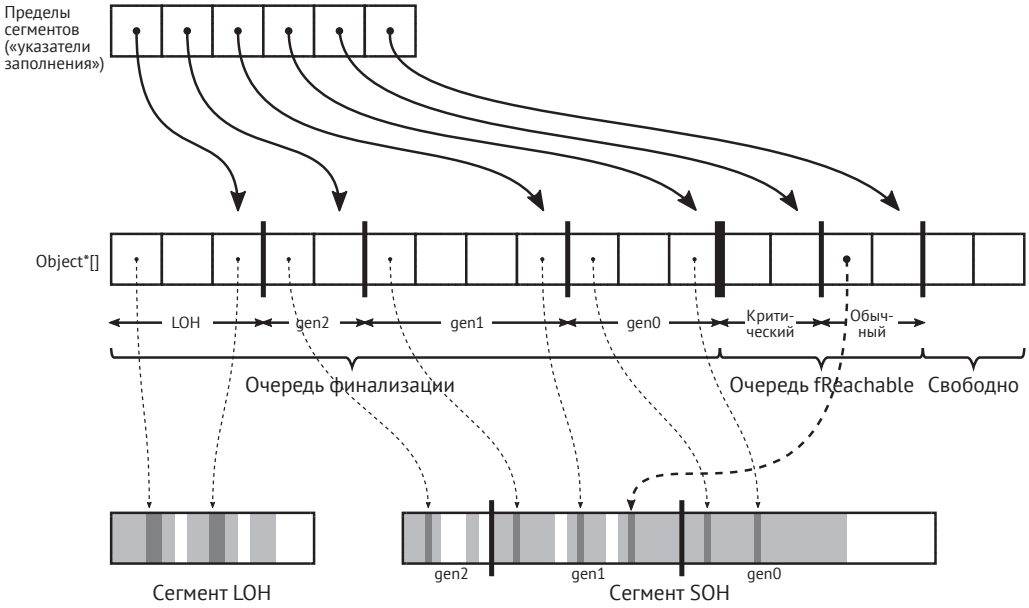


Рис. 12.1 ❖ Очереди финализации и fReachable.
Чтобы не загромождать рисунок, показано только несколько ссылок на объекты.
В действительности все элементы массива финализации (кроме свободной части)
содержат адреса каких-то объектов

Границы между сегментами запоминаются еще в одном коротком массиве адресов и называются *указателями заполнения*. Поэтому, чтобы просмотреть очередь финализации для данного поколения, нужно просто перебрать последовательные элементы массива в границах, хранящихся в соответствующих указателях заполнения. Перемещение из очереди финализации в очередь fReachable сводится к копированию адреса из одного сегмента в другой (в критическую или обычную часть области fReachable в зависимости от вида финализатора). Точно так же перевод или возврат объекта из поколения в поколение влечет за собой копирование адреса из исходного сегмента в конечный. А поскольку в массиве

финализации не должно быть пустых промежутков, копирование сопровождается сдвигом всех элементов в исходном и конечном сегментах массива (с соответствующим обновлением указателей заполнения).

Как уже сказано, только что созданный объект с финализатором необходимо добавить в очередь финализации – это называется *регистрацией для финализации*. С точки зрения реализации, такой объект должен быть добавлен в сегмент поколения 0 в массиве финализации (и да – при этом необходимо сдвинуть на одну позицию все последующие элементы в критическом и обычном сегментах fReachable). Из-за этого доступ к очереди финализации защищен блокировкой, поскольку ее могут одновременно модифицировать несколько потоков. Кроме того, если массив финализации заполнен, создается новая копия, которая на 20 % больше предыдущей. Понятно, что это еще один вид накладных расходов финализации, который напрямую отражается на пользовательских потоках и может замедлять выделение памяти – вследствие блокировки и копирования элементов массива.

Существует два важных метода API финализации в классе GC. Во-первых, метод `GC.ReRegisterForFinalize(object)`, который позволяет *заново зарегистрировать для финализации* ранее зарегистрированный объект. Зачем это нужно, мы увидим дальше. Метод `GC.ReRegisterForFinalize(object)` вызывает те же самые методы среды выполнения, что и при обычной регистрации для финализации во время выделения памяти, а значит, привносит те же самые накладные расходы. Но чаще всего он вызывается из финализатора, так что накладные расходы не слишком важны.

Во-вторых, в некоторых описанных ниже ситуациях может быть полезно явно запретить выполнение финализатора объекта – этот процесс называется *подавлением финализации*, и для него предназначен метод `GC.SuppressFinalize(object)`. Поскольку этот метод часто вызывается из пользовательских потоков (как часть паттерна Disposable), он очень хорошо оптимизирован. Он вообще не манипулирует массивом финализации, как можно было бы ожидать (например, он не удаляет из массива адрес объекта, для чего потребовалось бы сдвигать последующие элементы). Поскольку не надо синхронизировать доступ к массиву финализации, нет и соответствующих накладных расходов. Вместо этого метод устанавливает один бит в заголовке объекта – очень эффективная операция. А поток финализации просто не вызывает метод `Finalize` для тех объектов, в которых этот бит установлен.

Как мы уже объясняли, в конце этапа пометки GC проверяет, какие объекты в сегментах массива финализации, соответствующих обработанным поколениям, оказались помечены. Если объект не помечен, его адрес перемещается в критический или обычный сегмент очереди fReachable.

Позже поток финализации читает элементы из этих сегментов и обновляет для них указатели заполнения (будучи прочитанным, объект попадает в уже не просматриваемую «свободную часть» и становится по-настоящему недостижимым). В текущей реализации есть только один поток финализациии. Ходили слухи, что таких потоков будет несколько, но команда CLR их не подтвердила. С точки зрения реализации, ничто не мешает нескольким потокам финализации одновременно обрабатывать элементы очереди fReachable.

Если вы хотите исследовать исходный код финализации в CoreCLR, начните с класса `CFinalize`, который реализует описанный выше алгоритм. В этом классе есть поля типа `gc_heap::finalize_queue`, так что при наличии нескольких куч (серверный режим GC) массивов финализации на самом деле несколько (но поток финализации все равно один). В классе `CFinalize` массив финализации хранится в поле `m_Array` типа `Object**` (массив указателей на `Object`), а указатели заполнения – в поле `Object**[] m_FillPointers` (массив указателей на элементы массива финализации). В самом начале `m_Array` содержит 100 элементов, но потом он расширяется методом `CFinalize::GrowArray` (который создает массив на 20 % больше текущего и копирует в него все существующие элементы).

Реализация метода `GC.SuppressFinalize` очень проста: он вызывает метод `GCHeap::SetFinalizationRun`, который устанавливает бит `BIT_SBLK_FINALIZER_RUN` в заголовке указанного объекта.

Вышеупомянутый метод `GCHeap::RegisterForFinalization` вызывает метод `CFinalize::RegisterForFinalization`, который реализует описанную логику сдвига элементов (и при необходимости вызывает `GrowArray`), чтобы сохранить адрес объекта в очереди финализации.

На этапе пометки вызывается метод `CFinalize::GcScanRoots`, который начинает пометку с объектов, находящихся в обоих сегментах очереди fReachable (два последних используемых сегмента `m_Array`). В конце этого этапа вызывается метод `CFinalize::ScanForFinalization` для нужных сегментов (соответствующих выбранному и более молодым поколениям), он выполняет перемещение из очереди финализации, вызывая метод `MoveItem` с подходящими параметрами (в зависимости от того, какой финализатор, обычный или критический, определен в объекте). Если в очереди fReachable есть объекты, то генерируется событие `hEventFinalizer`, которое пробуждает поток финализации. И напоследок, в самом конце сборки мусора, вызывается метод `CFinalize::UpdatePromotedGenerations`, который проверяет все объекты в очереди финализации на принадлежность к текущему поколению и перемещает их в нужный сегмент.

Главный цикл потока финализации реализован в методе `FinalizerThread::FinalizerThreadWorker`. Он ждет события `hEventFinalizer` и после его получения начинает обработку, вызывая методы `FinalizerThread::FinalizeAllObjects` и `FinalizerThread::DoOneFinalization` (который вызывает метод финализатора, если не установлен бит `BIT_SBLK_FINALIZER_RUN`).

У проницательного читателя может возникнуть вопрос: зачем нужны все эти очереди и специальный поток, почему нельзя вызывать финализаторы прямо из сборщика мусора? Хороший вопрос. Напомним, что финализатор – это код, написанный пользователем. Программист может поместить туда все, что ему заблагорассудится, в т. ч. вызов `Thread.Sleep` на один час. Если бы GC вызывал финализаторы во время своей работы, то он заснул бы на час! Хуже того, код финализатора может приводить к взаимоблокировке, и тогда заблокируется весь сборщик мусора. Выполнение пользовательского кода финализаторов из GC может привести к непредсказуемым последствиям. Гораздо безопаснее выполнять процесс финализации асинхронно.

Накладные расходы финализации

Каков порядок накладных расходов финализации? В общем случае трудно измерить стоимость дополнительного перемещения объектов и обработки очереди финализации в процессе GC. Но легко измерить накладные расходы на медлен-

ный способ выделения памяти из-за обработки финализации. Это можно сделать с помощью простого теста производительности с помощью BenchmarkDotNet путем создания нескольких финализируемых и нефинализируемых объектов (листинг 12.11).

Листинг 12.11 ❖ Простой тест производительности для измерения накладных расходов на создание финализируемого объекта

```
public class NonFinalizableClass
{
   public int Value1;
   public int Value2;
   public int Value3;
   public int Value4;
}

public class FinalizableClass
{
   public int Value1;
   public int Value2;
   public int Value3;
   public int Value4;
   ~FinalizableClass()
   {
   }
}

[Benchmark]
public void ConsumeNonFinalizableClass()
{
   for (int i = 0; i < N; ++i)
   {
      var obj = new NonFinalizableClass();
      obj.Value1 = Data;
   }
}

[Benchmark]
public void ConsumeFinalizableClass()
{
   for (int i = 0; i < N; ++i)
   {
      var obj = new FinalizableClass();
      obj.Value1 = Data;
   }
}
```

От знакомства с результатами глаза могут полезть на лоб (листинг 12.12). В таком простом сценарии выделение памяти для небольшого финализируемого объекта примерно в 40 раз медленнее, чем для обычного (и есть даже сборки мусора в поколении 1 из-за дополнительного перемещения объектов)! Ассемблерный код, сгенерированный JIT-компилятором, в обоих случаях одинаков (за исключением вызываемой функции распределителя). Может быть, это и не проблема,

если финализируемые объекты создаются редко, но дважды подумайте, прежде чем включать финализатор в объекты, которые создаются часто, да еще на критическом с точки зрения производительности участке кода.

Листинг 12.12 ❖ Результаты теста производительности BenchmarkDotNet из листинга 12.11 (в столбцах Gen 0 и Gen 1 показано среднее количество сборок в поколениях 0 и 1 при одном прогоне теста)

Метод	N	Среднее	Gen 0	Gen 1	Выделено
ConsumeNonFinalizableClass	1	2.777 нс	0.0076	–	32 Б
ConsumeFinalizableClass	1	132.138 нс	0.0074	0.0036	32 Б
ConsumeNonFinalizableClass	10	30.667 нс	0.0762	–	320 Б
ConsumeFinalizableClass	10	1,342.092 нс	0.0744	0.0362	320 Б
ConsumeNonFinalizableClass	100	316.633 нс	0.7625	–	3200 Б
ConsumeFinalizableClass	100	13,607.436 нс	0.7477	0.3662	3200 Б
ConsumeNonFinalizableClass	1000	3,244.837 нс	7.6256	–	32000 Б
ConsumeFinalizableClass	1000	131,725.089 нс	7.5684	3.6621	32000 Б

Зная описанные выше детали реализации, мы можем свести воедино недостатки финализации:

- по умолчанию выбирается более медленный способ выделения памяти, на котором высоки накладные расходы на манипулирование очередью финализации;
- финализируемый объект переводится в другое поколение по меньшей мере один раз, что увеличивает вероятность кризиса среднего возраста;
- появляются накладные расходы на обработку даже тех финализируемых объектов, которые еще живы, – в основном в связи с необходимостью поддерживать соответствие между очередью финализации и поколениями;
- возникает опасность, когда скорость создания финализируемых объектов выше, чем скорость их финализации (см. ниже сценарий 12.1).

Сценарий 12.1. Утечка памяти из-за финализации

Описание. Объем памяти, потребляемой приложением, постоянно растет. Счетчики `\Память CLR .NET\Байт во всех кучах` и `\Память CLR .NET\Размер кучи поколения 2` увеличиваются. Требуется расследовать эту утечку памяти, но никаких очевидных ошибок нет. В этом сценарии моделируется довольно редкая, но возможная причина утечки памяти.

Существует одна трудноуловимая возможность утечки памяти. Поскольку финализаторы объектов из очереди fReachable выполняются последовательно, этот процесс занимает тем больше времени, чем медленнее работает финализатор. Если скорость создания финализируемых объектов выше, чем скорость их финализации, то очередь fReachable будет расти, собирая в себе все новые объекты, ожидающие финализации. И это еще одна причина, по которой код финализации должен быть максимально простым.

Такие «злобные» финализаторы смоделированы в листинге 12.13. Приложение создает финализируемые объекты гораздо быстрее, чем удается выполнить финализаторы. Моделирование высокого трафика в ситуации, когда мы уже

столкнулись с проблемой кризиса среднего возраста, приводит к очень частой сборке мусора[1].

Листинг 12.13 ❖ Экспериментальный код, демонстрирующий утечку памяти из-за финализации

```
public class LeakyApplication
{
   public void Run()
   {
      while (true)
      {
         Thread.Sleep(100);
         var obj = new EvilFinalizableClass(10, 10000);
         GC.KeepAlive(obj); // чтобы компилятор не оптимизировал obj до полного
                            // исчезновения
         GC.Collect();
      }
   }
}

public class EvilFinalizableClass
{
   private readonly int finalizationDelay;
   public EvilFinalizableClass(int allocationDelay, int finalizationDelay)
   {
      this.finalizationDelay = finalizationDelay;
      Thread.Sleep(allocationDelay);
   }

   ~EvilFinalizableClass()
   {
      Thread.Sleep(finalizationDelay);
   }
}
```

Мы уже знаем причину утечки, но посмотрим, как это выглядит с точки зрения диагностики. Кстати, я надеюсь, что решение проблемы вам уже известно – по возможности избегать финализаторов, а если без них никак не обойтись, то делать их максимально простыми и быстрыми.

Анализ. Начнем с самого простого инструмента с минимальной степенью вмешательства – счетчиков производительности. Понаблюдав за приложением некоторое время, мы действительно замечаем, что поколение 2 постоянно растет (тонкая линия на рис. 12.2). Есть еще два относящихся к финализации счетчика производительности:

- `\Память CLR .NET\Объектов, оставшихся после сборки мусора` – количество объектов, переживших последнюю сборку мусора из-за финализации (точнее, количество объектов, перемещенных из очереди финализации в очередь fReachable во время последней сборки мусора);

[1] Для простоты мы вызываем ее на каждой итерации, хотя могли бы вызывать, например, периодически. Но такой искусственный пример позволяет лучше проиллюстрировать некоторые проблемы диагностики.

- `\Память CLR .NET\Ожидающая выполнения операции Finalize память, унаследованная из поколения 0` – суммарный размер объектов, переживших последнюю сборку памяти из-за финализации (как и выше, суммарный размер объектов, перемещенных из очереди финализации в очередь fReachable). Отметим важный факт – несмотря на сбивающее с толку название, этот счетчик учитывает объекты из всех «собранных» поколений, а не только из поколения 0.

Напомним, что если нет счетчиков производительности, эту информацию можно получить из событий ETW/LTTng: событие `GCHeapStats_V1` содержит в точности те же значения, что и поля `FinalizationPromotedCount` и `FinalizationPromotedSize` соответственно.

Но оба эти счетчика на рис. 12.2 стабильны и свидетельствуют о переводе одного объекта размером 24 байта. Это не повод для тревоги. Объясняется это тем, что GC происходит после каждого создания объекта, а значит, переводится ровно один финализируемый объект. Заметим, что эти счетчики связаны со скоростью создания финализируемых объектов – чем больше таких объектов создано, тем больше будет переведено (из-за финализации). Они ничего не говорят о том, что происходит с переведенными объектами.

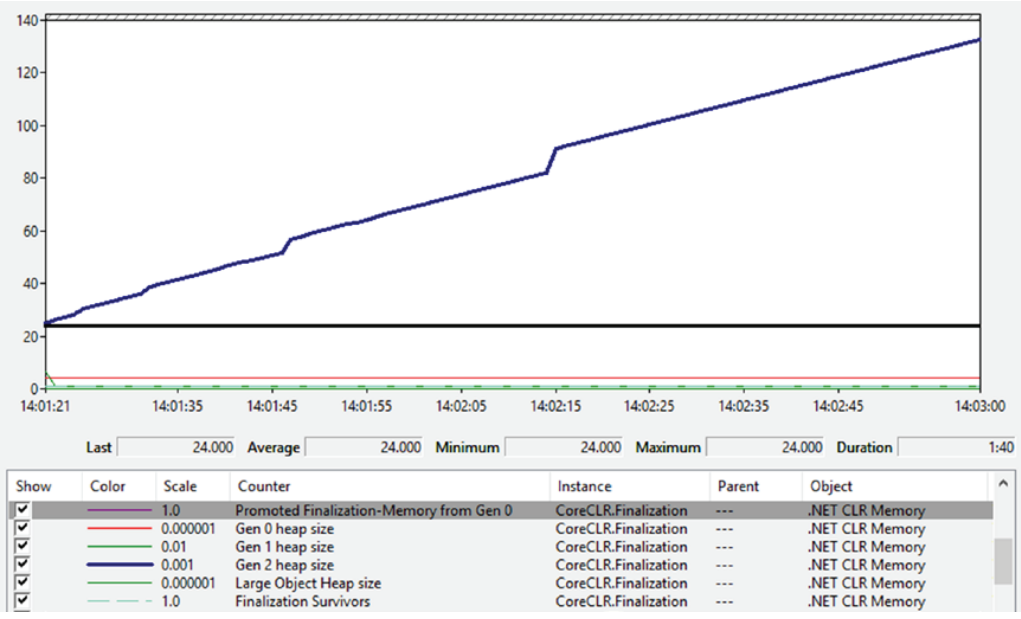


Рис. 12.2 ❖ Относящиеся к финализации счетчики производительности

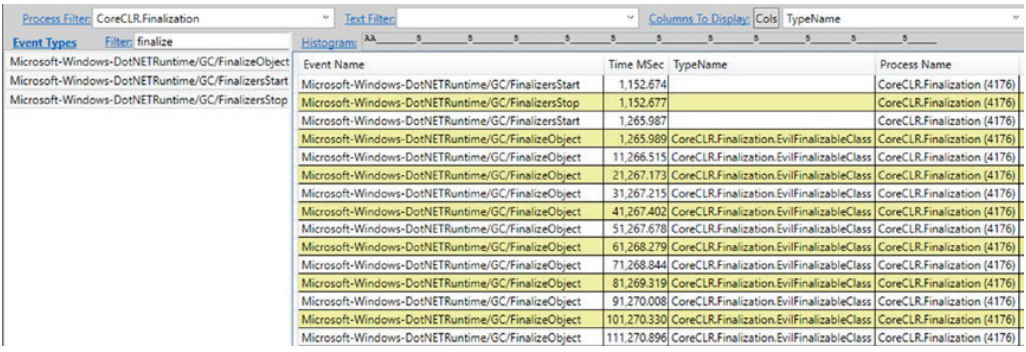
К сожалению, не существует счетчиков ни для скорости финализации, ни для размера очереди fReachable. Анализировать подобную проблему крайне неприятно, потому что многие инструменты вообще не принимают во внимание очередь fReachable – находящиеся в ней объекты, с точки зрения пользовательского кода, уже мертвы, а единственный корень, удерживающий ссылки на них, – очередь fReachable. Некоторые инструменты показывают экземпляры класса `EvilFinalizableClass` как недостижимые (если вообще показывают), что не вызывает тревоги и не побуж-

дает искать здесь источник утечки памяти (ведь они же недостижимы и, стало быть, не могут быть причиной проблемы, правда?). Поэтому мы переходим прямо к анализу проблемы финализации. Однако типичный подход к проблеме «постоянного роста размера поколения 2» – выяснить, что удерживает объекты в этом поколении (например, проанализировав снимок кучи в PerfView или память в WinDbg с расширением SOS). Чуть ниже мы покажем, как выполняется такой анализ.

В мониторинге финализации могут помочь относящиеся к финализации события ETW из группы `Microsoft-Windows-DotNETRuntime/GC` (они записываются, если в диалоговом окне Collect в PerfViews выбран стандартный режим .NET):

- `FinalizersStart` – генерируется, когда после GC пробуждается поток финализации;
- `FinalizeObject` – генерируется для каждого финализируемого объекта, обработанного потоком финализации;
- `FinalizersStop` – генерируется в конце финализации текущей порции объектов (когда обработаны все объекты в очереди fReachable).

Просмотр этих событий в записанном сеансе PerfView сразу же указывает на источник проблемы (рис. 12.3). Если в начале работы приложения произошла одна быстро завершившаяся финализация, то все последующие, очевидно, ведут себя аномально – промежуток времени между выполнениями соседних финализаторов составляет 10 секунд! И пока создаются новые экземпляры `EvilFinalizableClass`, поток финализации никогда не сможет догнать приложение (мы так никогда и не увидим события `FinalizersStop`).



Event Name	Time MSec	TypeName	Process Name
Microsoft-Windows-DotNETRuntime/GC/FinalizersStart	1,152.674		CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizersStop	1,152.677		CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizersStart	1,265.987		CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject	1,265.989	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject	11,266.515	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject	21,267.173	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject	31,267.215	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject	41,267.402	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject	51,267.678	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject	61,268.279	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject	71,268.844	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject	81,269.319	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject	91,270.008	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject	101,270.330	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject	111,270.896	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)

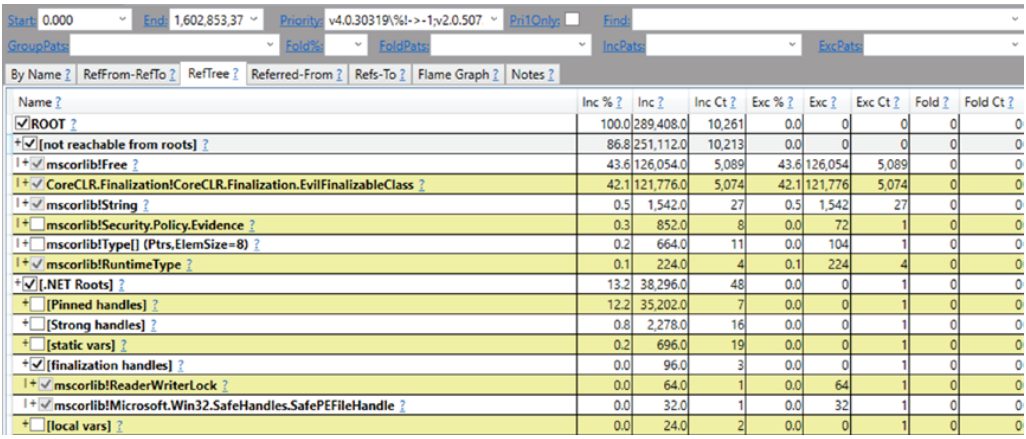
Рис. 12.3 ❖ События ETW, относящиеся к финализации

Очевидно, что в реальном приложении такая проблема проявлялась бы едва заметно. Но и в общем случае долго работающую финализацию можно диагностировать подобным образом.

Хотя наблюдение за аномально большим временем финализации – полезный ключ к разгадке, гораздо лучше было бы понаблюдать за истинной причиной – растущей очередью fReachable. К сожалению, в текущей версии PerfView на снимках кучи не показываются корни из очереди fReachable[1], а только из очереди фи-

[1] Существует запрос https://github.com/Microsoft/perfview/issues/722 на устранение этой ошибки, можете проверить, исправлена ли она сейчас.

нализации (рис. 12.4). И другие инструменты тоже чаще всего показывают такие объекты как недостижимые, не давая возможности исследовать очередь fReachable напрямую.


Start: 0.000 End: 1,602,853,37 Priority: v4.0.30319\%!->-1;v2.0.507 Pri1Only: Find:
GroupPats: Fold%: FoldPats: IncPats: ExcPats:
By Name | RefFrom-RefTo | RefTree | Referred-From | Refs-To | Flame Graph | Notes

Name	Inc %	Inc	Inc Ct	Exc %	Exc	Exc Ct	Fold	Fold Ct
ROOT	100.0	289,408.0	10,261	0.0	0	0	0	0
[not reachable from roots]	86.8	251,112.0	10,213	0.0	0	0	0	0
mscorlib!Free	43.6	126,054.0	5,089	43.6	126,054	5,089	0	0
CoreCLR.Finalization!CoreCLR.Finalization.EvilFinalizableClass	42.1	121,776.0	5,074	42.1	121,776	5,074	0	0
mscorlib!String	0.5	1,542.0	27	0.5	1,542	27	0	0
mscorlib!Security.Policy.Evidence	0.3	852.0	8	0.0	72	1	0	0
mscorlib!Type[] (Ptrs,ElemSize=8)	0.2	664.0	11	0.0	104	1	0	0
mscorlib!RuntimeType	0.1	224.0	4	0.1	224	4	0	0
[.NET Roots]	13.2	38,296.0	48	0.0	0	1	0	0
[Pinned handles]	12.2	35,202.0	7	0.0	0	1	0	0
[Strong handles]	0.8	2,278.0	16	0.0	0	1	0	0
[static vars]	0.2	696.0	19	0.0	0	1	0	0
[finalization handles]	0.0	96.0	3	0.0	0	1	0	0
mscorlib!ReaderWriterLock	0.0	64.0	1	0.0	64	1	0	0
mscorlib!Microsoft.Win32.SafeHandles.SafePEFileHandle	0.0	32.0	1	0.0	32	1	0	0
[local vars]	0.0	24.0	2	0.0	0	1	0	0

Рис. 12.4 ❖ Среди корней на снимках кучи нет адресов в очереди fReachable

Однако есть возможность более пристально взглянуть на обе очереди в программе WinDbg. И в сеансе отладки работающей программы, и при анализе дампа памяти можно выполнить команду SOS `!finalizequeue` (листинг 12.14), которая выдаст очень подробную информацию. Как видим, она сообщает и о «финализируемых объектах» (finalizable objects) (в каждом поколении, поскольку очередь финализации разделена на части по поколениям), и об объектах, «готовых к финализации» (ready for finalization), а это не что иное, как объекты в очереди fReachable. Налицо очевидная проблема – в очереди fReachable находится 5175 объектов!

Листинг 12.14 ❖ Применение команды SOS `finalizequeue` для изучения очередей финализации

```
> !finalizequeue
SyncBlocks to be cleaned up: 0
Free-Threaded Interfaces to be released: 0
MTA Interfaces to be released: 0
STA Interfaces to be released: 0
---------------------------------
generation 0 has 1 finalizable objects (000001751fe7e700->000001751fe7e708)
generation 1 has 0 finalizable objects (000001751fe7e700->000001751fe7e700)
generation 2 has 2 finalizable objects (000001751fe7e6f0->000001751fe7e700)
Ready for finalization 5175 objects (000001751fe7e708->000001751fe888c0)
Statistics for all finalizable objects (including all objects ready for
finalization):
              MT    Count  TotalSize Class Name
00007ffcee93c3e0        1         32 Microsoft.Win32.SafeHandles.
SafePEFileHandle
00007ffcee93d680        1         64 System.Threading.ReaderWriterLock
00007ffc93a35c98     5176     124224 CoreCLR.Finalization.EvilFinalizableClass
Total 5178 objects
```

Мы можем и дальше исследовать очередь fReachable, выполнив эту команду с параметром `-allReady` (листинг 12.15). Вот теперь все ясно и соответствует ожиданиям – имеется 5175 экземпляров класса `EvilFinalizableClass`. Такое большое количество объектов в очереди fReachable – тревожный признак. Можно дополнительно подтвердить, что имеет место проблема, создав еще несколько дампов и посмотрев, растет ли это число.

Листинг 12.15 ❖ Применение команды SOS `finalizequeue` для изучения только очереди fReachable

```
> !finalizequeue -allReady
SyncBlocks to be cleaned up: 0
Free-Threaded Interfaces to be released: 0
MTA Interfaces to be released: 0
STA Interfaces to be released: 0
---------------------------------
generation 0 has 1 finalizable objects (000001751fe7e700->000001751fe7e708)
generation 1 has 0 finalizable objects (000001751fe7e700->000001751fe7e700)
generation 2 has 2 finalizable objects (000001751fe7e6f0->000001751fe7e700)
Finalizable but not rooted:
Ready for finalization 5175 objects (000001751fe7e708->000001751fe888c0)
Statistics for all finalizable objects that are no longer rooted:
              MT   Count   TotalSize Class Name
00007ffc93a35c98    5175      124200 CoreCLR.Finalization.EvilFinalizableClass
Total 5175 objects
```

Обратите также внимание на диапазоны адресов сегментов массива финализации (приведены в скобках). Мы можем распечатать содержимое этих сегментов массива и получить ссылки на конкретные финализируемые объекты (в листинге 12.16 показан диапазон, соответствующий очереди fReachable).

Листинг 12.16. Просмотр содержимого очереди fReachable

```
> dq 000001751fe7e708 000001751fe888c0
...
00000175`1fe88888  00000175`21850358 00000175`21850388
00000175`1fe88898  00000175`218503b8 00000175`218503e8
00000175`1fe888a8  00000175`21850418 00000175`21850448
00000175`1fe888b8  00000175`21850478 00000175`2182ae28
> !do 00000175`2182ae28
Name:        CoreCLR.Finalization.EvilFinalizableClass
MethodTable: 00007ffc93a35c98
EEClass:     00007ffc93b41208
Size:        24(0x18) bytes
...
```

Аналогичный анализ можно провести с помощью расширения SOSEX, выполнив команды `finq` и `frq` для исследования очереди финализации и очереди fReachable соответственно (листинг 12.17). Я упомянул об этих командах, потому что они выводят результат в более удобном виде, чем команды SOS.

Листинг 12.17 ❖ Применение команд `finq` и `frq` из расширения SOSEX для изучения обеих очередей финализации

```
> .load g:\Tools\Sosex\64bit\sosex.dll
> !finq -stat
Generation 0:
       Count     Total Size  Type
-----------------------------------------------------------
           1             24  CoreCLR.Finalization.EvilFinalizableClass
1 object, 24 bytes

Generation 1:
0 objects, 0 bytes
Generation 2:
       Count     Total Size  Type
-----------------------------------------------------------
           1             32  Microsoft.Win32.SafeHandles.SafePEFileHandle
           1             64  System.Threading.ReaderWriterLock
2 objects, 96 bytes

TOTAL: 3 objects, 120 bytes

> !frq -stat
Freachable Queue:
       Count     Total Size  Type
-----------------------------------------------------------
        5175         124200  CoreCLR.Finalization.EvilFinalizableClass
5,175 objects, 124,200 bytes
```

В настоящее время команда `!mex.finalizable` из расширения WinDbg MEX, похоже, неправильно распечатывает объекты из очереди fReachable для приложений .NET Core.

Воскрешение

С финализацией связан очень интересный вопрос. Как мы уже знаем, финализатор вызывается, когда единственным корнем объекта является очередь fReachable. Поток финализации вызывает метод `Finalize`, после чего ссылка на объект удаляется из очереди. Таким образом, объект становится недостижимым и будет убран, когда в этом поколении в следующий раз запустится сборка мусора.

Но метод `Finalize` является методом экземпляра (у него есть доступ к `this`), и в нем допустим любой пользовательский код. Следовательно, ничто не мешает нам присвоить ссылку на сам объект (`this`) какому-то глобально доступному корню (например, статической переменной). И вот – наш объект внезапно снова стал доступным (листинг 12.18)! Это явление, называемое *воскрешением*, неразрывно связано с тем фактом, что финализатор может содержать неконтролируемый пользовательский код.

Листинг 12.18 ❖ Пример (не вполне корректный) воскрешения объекта

```
class FinalizableObject
{
   ~FinalizableObject()
```

```
   {
      Program.GlobalFinalizableObject = this;
   }
}
```

Объект, который нужно было просто убрать в мусор, снова стал достижимым. Теперь единственным корнем является глобальная ссылка на него (в нашем примере `Program.GlobalFinalizableObject`), но, конечно, при желании могут появиться и другие корни.

Но что, если воскрешенный объект снова станет недостижимым? Он будет убран в мусор или снова воскреснет? Чтобы ответить на этот вопрос, вспомним, что регистрация для финализации происходит во время выделения памяти. После выполнения финализатора ссылка на объект пропадает из очереди fReachable. Воскрешение не помещает ее в очередь финализации, поэтому когда экземпляр класса `FinalizableObject` в листинге 12.18 станет недостижимым во второй раз, его финализатор не будет вызван – его просто нет в очереди финализации!

Однако если уж мы используем воскрешение, то чаще всего хотим, чтобы объект воскресал всегда, а не всего один раз. Поэтому следует использовать упоминавшийся выше метод `GC.ReRegisterForFinalize`, чтобы повторно зарегистрировать объект для финализации (листинг 12.19). Сделав это, мы создадим бессмертный объект – он никогда не будет убран в мусор. Заметим, что в упрощенном примере из листинга 12.19 это не совсем так – мы можем создать несколько экземпляров класса `FinalizableObject`, и возникает состояние гонки – только последний финализированный объект будет повторно зарегистрирован для финализации и, стало быть, надлежащим образом воскрешен!

Листинг 12.19 ❖ Пример воскрешения объекта (исправление кода в листинге 12.18)

```
class FinalizableObject
{
   ~FinalizableObject()
   {
      Program.GlobalFinalizableObject = this;
      GC.ReRegisterForFinalize(this);
   }
}
```

Воскрешение – не слишком популярная техника. Она редко используется даже в коде, написанном самой компанией Майкрософт. Дело в том, что мы скрытно манипулируем временем жизни объекта. Это финализация в квадрате – со всеми ее недостатками, но проявляющимися с еще большей силой.

Можно представить себе организацию пула объектов на основе воскрешения – финализатор отвечает за возвращение объекта в некоторый разделяемый пул (его воскрешение), как в листинге 12.20. Однако имя `EvilPool` и отсутствующие детали реализации – не случайность. Есть куда лучшие способы реализации пула объектов, основанные на явном управлении (например, класс `ArrayPool<T>`, продемонстрированный в главе 6). Нет никаких особых преимуществ в том, чтобы делать управление пулом неявным. Памятуя обо всех подводных камнях финализаторов, лучшее решение – не использовать их вовсе (особенно если существуют более простые альтернативы). Тем не менее предлагаю вам реализовать `EvilPool`

в качестве самостоятельного упражнения, пусть даже практически бесполезного, – получите массу удовольствия!

Листинг 12.20 ❖ Пример практического применения воскрешения

```
public class EvilPool<T> where T : class
{
   static private List<T> items = new List<T>();
   static public void ReturnToPool(T obj)
   {
      // ...
      // Добавить obj в список items
      GC.ReRegisterForFinalize(obj);
   }

   static public T GetFromPool() { ... }
}

public class SomeClass
{
   ~SomeClass()
   {
      EvilPool<SomeClass>.ReturnToPool(this);
   }
}
```

Для любого объекта, в котором определен финализатор, можно вызвать методы `GC.ReRegisterForFinalize` или `GC.SuppressFinalize`, потому что эти методы ожидают простой объект в качестве аргумента (они проверяют, действительно ли финализатор определен). Это означает, что управление ресурсами, принадлежащими объекту, можно видоизменять, контролируя, как именно вызывается его финализатор. Для некоторых объектов это нежелательно. Примером может служить тип `System.Threading.Timer`, предоставляющий механизм для периодического выполнения некоторого метода в пуле потоков с заданным интервалом. Финализатор `Timer` говорит пулу потоков, что таймер нужно отменить. Тогда, например, вызывая для такого объекта метод `GC.SuppressFinalize`, мы управляем поведением таймера необычным образом – он никогда не останавливается. Хорошее это решение или плохое, зависит от обстоятельств. Но в большинстве подобных сценариев управление внутренним поведением объекта подобным способом неожиданно.

Если мы все-таки хотим полагаться на финализацию, но при этом оградить себя от таких проблем, то должны исключить саму возможность манипулирования нашим финализатором. Первый шаг на этом пути – сделать класс запечатанным (`sealed`), запретив тем самым переопределение метода `Finalize` в производных классах. Второй шаг – ввести некий помощник, т. е. финализируемый объект, отвечающий за финализацию главного объекта. Именно такой подход был принят при реализации типа `System.Threading.Timer`. В упрощенном виде он представлен в листинге 12.21. Внутренний закрытый класс `TimerHolder` хранит ссылку на наш главный объект `Timer`. Когда объект `Timer` становится недостижимым, таковым оказывается и поле `timerHolder`, и это приводит к срабатыванию его финализатора, который отвечает за очистку родительского объекта (обратите внимание, что в этом примере частично используется паттерн Disposable).

Листинг 12.21 ❖ Упрощенная реализация класса `Timer` (с применением вложенного финализируемого объекта)

```
public sealed class Timer : IDisposable
{
   private TimerHolder timerHolder;

   public Timer()
   {
      timerHolder = new TimerHolder(this);
   }

   private sealed class TimerHolder
   {
      internal Timer m_timer;

      public TimerHolder(Timer timer) => m_timer = timer;

      ~TimerHolder() => m_timer?.Close();

      public void Close()
      {
         m_timer.Close();
         GC.SuppressFinalize(this);
      }
   }

   public void Close()
   {
      Console.WriteLine("Финализация таймера!");
   }

   public void Dispose()
   {
      timerHolder.Close();
   }
}
```

Таким образом, мы получаем финализацию, не раскрывая ее миру, – сам класс `Timer` не является финализируемым, так что вызывать для него методы `GC.SuppressFinalize` и `GC.ReRegisterForFinalize` нельзя.

> Имеет ли смысл вызывать `GC.ReRegisterForFinalize` в сценариях с воскрешением, когда воскрешенный объект не присвоен ни одному корню (например, без строки `Program.GlobalFinalizableObject = this` в листинге 12.19)? Имеет, и еще какой! Что в этом случае происходит? Вновь зарегистрированный объект оказывается в очереди финализации и будет обработан при следующей сборке мусора. И весь цикл начнется снова – ссылка будет перемещена в очередь fReachable, и будет вызван ее финализатор… который снова воскресит объект. Таким образом мы можем создать бессмертный объект, на который будет существовать ссылка только из очередей финализации. Пример, когда это может оказаться полезным, приведен в листинге 12.37 ниже. Впрочем, чаще такая повторная регистрация для финализации условна – это позволяет выполнить код финализатора несколько раз (если логика финализации настолько сложна или важна, что это оправдано). Но помните – это ни в коем случае нельзя назвать паттерном проектирования, которому надлежит следовать. Просто знайте, что такая возможность существует.

Уничтожаемые объекты

Мы уже долго говорим о недетерминированной финализации. Давайте же наконец перейдем к более предпочтительному способу очистки ресурсов – явной детерминированной финализации. Концептуально она гораздо проще недетерминированной финализации с ее финализаторами – и это одно из основных преимуществ. Подводных камней и недостатков куда меньше. На самом деле концептуально существует всего два метода:

- инициализация – служит для создания и хранения ресурсов. В .NET это, очевидно, поддерживаемый средой выполнения конструктор, который вызывается при создании объектов;
- очистка – служит для освобождения ресурсов. В .NET для него нет поддержки на уровне среды выполнения. Как его назвать, решайте сами.

Вернувшись к простому классу `FileWrapper` из листинга 12.1, избавившись от финализации и добавив явную очистку, мы придем к коду, показанному в листинге 12.22. Метод очистки – это обычный метод, который мы должны вызывать для освобождения принадлежащих объекту ресурсов. По сравнению с листингом 12.1 добавлен еще метод `UseMe`, который понадобится в последующих примерах.

Листинг 12.22 ❖ Простой пример явной очистки

```
class FileWrapper
{
   private IntPtr handle;
   public FileWrapper(string filename)
   {
      Unmanaged.OFSTRUCT s;
      handle = Unmanaged.OpenFile(filename, out s, 0x00000000);
   }

   // Очистка
   public void Close()
   {
      if (handle != IntPtr.Zero)
         Unmanaged.CloseHandle(handle);
   }

   public int UseMe()
   {
      byte[] buffer = new byte[1];
      if (Unmanaged.ReadFile(this.handle, buffer, 1, out uint read, IntPtr.Zero))
      {
         return buffer[0];
      }
      return -1;
   }
}
```

Каким же простым становится мир, в котором очистка производится явно (листинг 12.23). Все выполняется в предсказуемом порядке, никаких сюрпризов. Использование объекта заключается в его методах инициализации (конструктор)

и очистки, поэтому ранняя сборка корней нам не помешает. Мы точно знаем, где захватывается ресурс и где он освобождается.

Листинг 12.23 ❖ Использование класса `FileWrapper` из листинга 12.22

```
var file = new FileWrapper(@"C:\temp.txt");
Console.WriteLine(file.UseMe());
file.Close();
```

Но если этот подход настолько идеален, то почему кто-то озаботился придумыванием альтернативы? Увы, у этого подхода есть один гигантский недостаток – программист должен помнить о необходимости вызвать метод очистки. А если он забудет, мы получим утечку нашего (быть может, ограниченного) ресурса.

Чтобы помочь в решении этой проблемы, в языке C# явная очистка стандартизована посредством интерфейса `IDisposable`. Его определение тривиально (листинг 12.24). Это контракт, который говорит: «У меня есть нечто такое, что нужно почистить, когда я закончу работу».

Листинг 12.24 ❖ Объявление интерфейса `IDisposable`

```
namespace System {
   public interface IDisposable {
      void Dispose();
   }
}
```

Таким образом, если следовать этому паттерну, класс `FileWrapper` из листинга 12.22 должен реализовать интерфейс `IDisposable`, и его метод `Dispose` должен вызвать метод `Close` (или заменить его, как показано в листинге 12.25).

Листинг 12.25 ❖ Простой пример явной очистки с применением интерфейса `IDisposable`

```
class FileWrapper : IDisposable
{
   private IntPtr handle;
   public FileWrapper(string filename)
   {
      Unmanaged.OFSTRUCT s;
      handle = Unmanaged.OpenFile(filename, out s, 0x00000000);
   }

   // Очистка
   public void Dispose()
   {
      if (handle != IntPtr.Zero)
         Unmanaged.CloseHandle(handle);
   }

   public int UseMe()
   {
      byte[] buffer = new byte[1];
      if (Unmanaged.ReadFile(this.handle, buffer, 1, out uint read, IntPtr.Zero))
      {
         return buffer[0];
```

```
      }
      return -1;
   }
}
```

Наличие четкого контракта помогает при ручном и автоматическом анализе кода. Если создан экземпляр типа, реализующего интерфейс `IDisposable` (далее будем называть такие объекты *уничтожаемыми (disposable)*), но нигде не вызывается его метод `Dispose`, налицо кандидат на удаление из программы. Особенно полезны в этом отношении различные инструменты автоматизированного анализа (например, ReSharper).

> В комментариях к интерфейсу `IDisposable` читаем: «Теоретически компилятор мог бы использовать этот интерфейс как маркер и гарантировать, что уничтожаемый объект, созданный в данном методе, очищается на любом пути выполнения, но на практике такие драконовские меры отпугнули бы многих пользователей».

Поскольку язык, опирающийся на внешние инструменты, не внушает доверия, стандартизация явной очистки в C# не остановилась на этом, и был введен *оператор using*. Это еще одна простая конструкция, освобождающая нас от необходимости явно вызывать `Dispose` (листинг 12.26).

Листинг 12.26 ❖ Пример использования оператора using

```
public static void Main()
{
   using (var file = new FileWrapper())
   {
      Console.WriteLine(file.UseMe());
   }
}
```

Компилятор C# транслирует оператор using в блок try-finally, так что метод `Dispose` вызывается в блоке finally (листинг 12.27). Отметим, что это заодно дает уверенность в том, что ранняя сборка корней не затронет объект слишком рано, поскольку его метод `Dispose` вызывается в конце.

Листинг 12.27 ❖ Результат трансляции оператора using из листинга 12.26

```
public static void Main()
{
   FileWrapper fileWrapper = new FileWrapper();
   try
   {
      Console.WriteLine(file.UseMe());
   }
   finally
   {
      if (fileWrapper != null)
      {
         ((IDisposable)fileWrapper).Dispose();
      }
   }
}
```

Но даже наличие оператора using не гарантирует, что программист будет им пользоваться. Иными словами, ничто не помешает ему просто создать экземпляр уничтожаемого объекта и забыть вызвать его метод Dispose. Оператор using – не более чем рекомендуемая практика.

Если с точки зрения управления ресурсами код очистки критически важен (а в большинстве случаев так оно и есть), то есть два подхода:

- будьте вежливы и попросите программистов всегда вызывать метод Dispose уничтожаемых объектов. Звучит немного странно, но на самом деле это предпочтительный способ. Вышеупомянутые инструменты могут помочь, особенно если требование еще строже – уничтожаемые объекты должны использоваться только внутри оператора using. Нарушение этого правила легко обнаруживается, и, например, запрос на запись в систему контроля версий может быть отклонен, если оно встречается в коде;
- подстраховаться, воспользовавшись финализатором для вызова Dispose, – и это очень популярный подход. Если метод Dispose не был вызван явно, то финализатор вызовет его вместо нас. Тут есть только один недостаток – мы прибегаем к финализаторам, хотя лучше бы их избегать. Конечно, такой защитный финализатор может быть очень простым, поэтому можно надеяться, что он не вызовет серьезных проблем. Но все равно мы немного замедляем выделение памяти и должны обрабатывать лишний объект в очереди финализации. Так что применяйте этот подход только в действительно важных случаях.

Если мы используем второй подход и единственная обязанность финализатора – освободить ресурсы, вызвав метод Dispose, то его не нужно вызывать, если ответственный программист уже вызвал Dispose явно. Именно для этой цели предназначен упоминавшийся ранее метод GC.SuppressFinalize – он подавляет вызов финализатора объекта. В результате мы пришли к очень популярному паттерну – метод Dispose вызывает GC.SuppressFinalize, поскольку финализатор больше не нужен. Очень лаконичный пример такого подхода можно найти в библиотеке System.Reflection в виде абстрактного класса CriticalDisposableObject (листинг 12.28). Он реализует критический финализируемый объект, в котором финализатор служит для подстраховки.

Листинг 12.28 ❖ Тип CriticalDisposableObject в пространстве имен System.Reflection.Internal

```
namespace System.Reflection.Internal
{
   internal abstract class CriticalDisposableObject :
   CriticalFinalizerObject, IDisposable
   {
      protected abstract void Release();

      public void Dispose()
      {
         Release();
         GC.SuppressFinalize(this);
      }
```

```
    ~CriticalDisposableObject()
    {
       Release();
    }
  }
}
```

Вообще, использование явной очистки в виде `IDisposable` и защитной неявной очистки в виде финализации эволюционировало в *паттерн Disposable* (или *паттерн IDisposable*). Это несколько более структурированный способ сочетания обоих подходов (листинг 12.29). Паттерн Disposable можно рассматривать как едва ли не стандарт в мире .NET. Основное отличие – добавление виртуального метода `Dispose`, который вызывается как из финализатора (с аргументом `disposing`, равным `false`), так и из явного метода `Dispose` (с аргументом `disposing`, равным `true`). Тогда производные классы могут добавлять собственный код очистки, не изменяя общей логики финализации. Кроме того, специальное поле `disposed` предотвращает многократное уничтожение объекта. Каждый открытый метод должен проверять этот флаг и (как правило) бросать исключение `ObjectDisposedException`, информирующее о том, что объект уже нельзя использовать.

Листинг 12.29 ❖ Пример использования явной и неявной очистки в паттерне IDisposable

```
class FileWrapper : IDisposable
{
   private bool disposed = false;

   private IntPtr handle;
   public FileWrapper(string filename)
   {
      Unmanaged.OFSTRUCT s;
      handle = Unmanaged.OpenFile(filename, out s, 0x00000000);
   }

   // Очистка
   protected virtual void Dispose(bool disposing)
   {
      if (!disposed)
      {
         if (disposing)
         {
            // Здесь должен быть код, который работает только при явном вызове Dispose
         }

         // Общая очистка - включая неуправляемые ресурсы
         if (handle != IntPtr.Zero)
            Unmanaged.CloseHandle(handle);
         disposed = true;
      }
   }

   ~FileWrapper()
   {
```

```
      Dispose(false);
   }

   public void Dispose()
   {
      Dispose(true);
      GC.SuppressFinalize(this);
   }

   public int UseMe()
   {
      if (this.disposed) throw new ObjectDisposedException("...");
      byte[] buffer = new byte[1];
      if (Unmanaged.ReadFile(this.handle, buffer, 1, out uint read, IntPtr.Zero))
      {
         return buffer[0];
      }
      return -1;
   }
}
```

> Уничтожаемые объекты и оператор using можно использовать также для реализации простой техники подсчета ссылок, как в листинге 7.3 в главе 7. Вводится специальный вспомогательный класс, который используется совместно с оператором using. Его конструктор прибавляет 1 к счетчику ссылок, а метод `Dispose` вычитает 1. Как только счетчик обращается в ноль, инициируется очистка целевого объекта. Конечно, мы можем подстраховаться, включив в класс паттерн Disposable; это гарантирует, что очистка произойдет, даже если логика подсчета ссылок некорректна.

В общем случае, согласно комментариям к интерфейсу `IDisposable` в исходном коде .NET, реализация метода `Dispose` должна отвечать нескольким критериям:

- ее можно безопасно вызывать несколько раз;
- она освобождает все ресурсы, связанные с экземпляром;
- при необходимости она вызывает метод `Dispose` базового класса;
- она подавляет финализацию экземпляра, чтобы помочь GC уменьшить количество объектов в очереди финализации[1];
- `Dispose` не должен вызывать исключений, кроме очень серьезных, совершенно неожиданных ошибок (например, `OutOfMemoryException`).

Но не забывайте – ни уничтожаемые объекты, ни паттерн Disposable не имеют прямого отношения к сборке мусора! Метод `Dispose` не освобождает память объекта, не удаляет его, не делает ничего подобного. Если бы нужно было назвать одну, самую главную вещь, о которой рассказано в этой части главы, то это она и есть. Как вы могли заметить, мы здесь почти не упоминали о среде выполнения (если не считать финализации). Уничтожаемые объекты реализованы целиком на уровне языка.

---

[1] Как уже отмечалось, подавление финализации реализовано тривиально – установкой одного бита в заголовке объекта. Поэтому можно не беспокоиться о накладных расходах (например, можно вызывать метод подавления дважды для одного и того же объекта как в производном, так и в базовом классе).

Безопасные описатели

В реализации финализаторов много подводных камней. А неуправляемые ресурсы чаще всего представлены каким-то описателем или указателем, отсюда и тип `IntPtr`. В сочетании эти два факта привели к появлению нового типа, помогающего обращаться с неуправляемыми ресурсами. В версии .NET Framework 2.0, помимо критических финализаторов, был введен построенный на их основе тип `SafeHandle`. Это стало гораздо лучшей альтернативой прежним подходам к управлению системными ресурсами (в т. ч. финализаторам, необернутому типу `IntPtr` и типу `HandleRef`). Она основана на уже отмеченном наблюдении, что почти все описатели можно представить в виде `IntPtr`, а значит, можно наделить их дополнительным поведением по умолчанию и поддержать со стороны среды выполнения.

Поэтому предпочтительной и рекомендуемой альтернативой реализации финализатора является создание типа, производного от абстрактного класса `System.Runtime.InteropServices.SafeHandle` (листинг 12.30), и использование его в качестве обертки описателя. Поскольку основная часть логики уже реализована, мы в большей степени защищены от проблем, которые могли бы возникнуть при самостоятельной реализации логики финализации. Как видим, класс `SafeHandle` является критически финализируемым и реализует паттерн `Disposable`. Логика его методов `Dispose` и `Finalize` на самом деле внутренняя (реализована в самой среде выполнения).

Листинг 12.30 ❖ Фрагменты класса `SafeHandle` (значительная часть кода, в т. ч. атрибуты членов, для краткости опущены)

```
public abstract class SafeHandle : CriticalFinalizerObject, IDisposable
{
   protected IntPtr handle; // этот член должен быть защищенным, чтобы производные
                            // классы могли использовать out-параметры
   private int _state; // комбинация счетчика ссылок и флагов closed/disposed
                       // (чтобы ее можно было атомарно модифицировать)

   ~SafeHandle()
   {
      Dispose(false);
   }

   public void Dispose() {
      Dispose(true);
   }

   protected virtual void Dispose(bool disposing)
   {
      if (disposing)
         InternalDispose();
      else
         InternalFinalize();
   }

   [MethodImplAttribute(MethodImplOptions.InternalCall)]
   extern void InternalFinalize();

   [MethodImplAttribute(MethodImplOptions.InternalCall)]
```

```
    private extern void InternalDispose();

    public abstract bool IsInvalid { get; }

    protected abstract bool ReleaseHandle();
}
```

> В CoreCLR методы `InternalDispose` и `InternalFinalize` реализованы в `SafeHandle::DisposeNative` и `SafeHandle::Finalize` соответственно. Тот и другой вызывают метод `SafeHandle::Dispose`, который, в свою очередь, вызывает `SafeHandle::Release` – главную рабочую лошадку. Он вызывает управляемый метод `IsInvalidHandle`, и если тот возвращает `true`, то вызывается управляемый метод `ReleaseHandle` (опосредованно с помощью метода `SafeHandle::RunReleaseMethod`).

Специальная обработка со стороны среды выполнения делает `SafeHandle` более чем просто рекомендуемой практикой. И самое главное – CLR особым образом относится к экземплярам этого класса при обращении к P/Invoke – они защищены от сборщика мусора (как `HandleRef`) и из соображений безопасности реализуют семантику подсчета ссылок. Это означает, что JIT-компилятор вставляет перед каждым таким вызовом P/Invoke увеличение внутреннего счетчика ссылок, а по завершении – уменьшение. И лишь когда счетчик ссылок обращается в ноль, экземпляр освобождает ресурс. Это предотвращает атаку посредством рециркуляции описателей (см. примечание ниже).

> *Атака посредством рециркуляции описателей*
>
> Существует трудноуловимая проблема безопасности при использовании необернутых системных описателей (в частности, весьма популярного представления в виде `IntPtr`, использованного в приведенных выше примерах класса `FileWrapper`). В Windows системные описатели практически сразу же после освобождения используются повторно (рециркулируют), поскольку считаются ограниченным общесистемным ресурсом. Атака посредством рециркуляции описателей может быть использована внутри одного процесса .NET, чтобы получить повышенный уровень привилегий из недоверенного потока (с очень ограниченными правами) к описателю, который, по идее, должен быть доступен только потоку, наделенному всеми правами. Такую атаку можно организовать, когда управляемый объект, хранящий описатель, предоставляет какой-то метод явного завершения, как в популярном паттерне Disposable. Атакующий недоверенный поток может явно очистить такой ресурс (закрыв соответствующий описатель, но запомнив его значение), хотя он еще используется другими потоками. Другие потоки, скорее всего, столкнутся с ошибками, потому что описатель внезапно закрылся. При этом может случиться так, что какой-то другой, доверенный, поток открывает новый ресурс, который благодаря рециркуляции получает тот же самый описатель. И теперь у атакующего потока есть описатель, указывающий на новый ресурс, который вообще-то должен быть ему недоступен.

Таким образом, безопасные описатели обладают многими преимуществами по сравнению с альтернативами:

- они критически финализируемые, что делает их более надежными, чем обычные финализаторы. При этом нет необходимости писать собственные финализаторы, что снимает с программиста ответственность по преодолению различных проблем, связанных с финализацией;
- они представляют собой минимальные обертки над неуправляемыми ресурсами (описателями), чем снижают риск создания больших объектов

с многочисленными зависимостями, которые будут переведены в старшее поколение из-за финализации;
- наш объект вообще не обязан быть финализируемым – когда объект, в котором хранится объект, производный от SafeHandle, становится недостижимым, такой же становится и эта обертка. Так что в конечном итоге будет вызван финализатор, освобождающий обертку;
- улучшенное управление временем жизни – специальная обработка сборщиком мусора во время вызовов P/Invoke оставляет объекты живыми без необходимости вызывать GC.KeepAlive или пользоваться типом HandleRef;
- строгая типизация вместо использования чистых IntPtr, поскольку разным ресурсам соответствуют разные типы, производные от SafeHandle. Поэтому в вызовах P/Invoke не фигурируют ничего не значащие описатели типа IntPtr. Например, мы не сможем передать описатель файла функции, предназначенной для работы с мьютексом;
- повышенная безопасность благодаря предотвращению атак посредством рециркуляции описателей.

К сожалению, несмотря на давнее присутствие в экосистеме .NET, безопасные описатели все еще непопулярны в обычном коде (хотя в самой .NET встречаются сплошь и рядом). Чаще всего программисты предпочитают простую логику финализации, даже если приходится оборачивать простые описатели IntPtr.

> Если вас интересует, как JIT-компилятор обрабатывает объекты типа SafeHandle, начните с метода ILSafeHandleMarshaler::ArgumentOverride. Он вызывает SafeHandle::AddRef и SafeHandle::Release до и после вызова P/Invoke.

А ведь определить тип, производный от SafeHandle, очень просто. Нужно лишь переопределить два члена: IsInvalid и ReleaseHandle. Для удобства есть даже два более специализированных абстрактных класса[1]: SafeHandleMinusOneIsInvalid и SafeHandleZeroOrMinusOneIsInvalid, предлагающих тривиальные реализации свойства IsInvalid (характер проверок следует из самого названия).

В производном классе у нас есть доступ к защищенному описателю IntPtr, и, кроме того, мы можем установить его с помощью метода SetHandle. Чтобы усовершенствовать класс FileWrapper, мы сначала создадим безопасный описатель файла (листинг 12.31). Основная логика класса, производного от SafeHandle, сосредоточена в конструкторе (получении описателя) и в реализации метода ReleaseHandle.

Листинг 12.31 ❖ Пример реализации класса, производного от SafeHandle

```
class CustomFileSafeHandle : SafeHandleZeroOrMinusOneIsInvalid {
   // P/Invoke вызывает этот конструктор при возврате SafeHandle. Затем будет
   // установлено корректное значение описателя
   private CustomFileSafeHandle() : base(true)
   {
   }
```

[1] Они включены, чтобы предоставить стандартизованный способ использования описателей, поскольку чаще всего значения 0 и –1 действительно считаются недопустимыми описателями.

```
   // Если требуется поддержать предоставление описателя пользователем
   internal CustomFileSafeHandle (IntPtr preexistingHandle, bool ownsHandle) :
      base(ownsHandle)
   {
      SetHandle(preexistingHandle);
   }

   internal CustomFileSafeHandle(string filename) : base(true)
   {
      Unmanaged.OFSTRUCT s;
      IntPtr handle = Unmanaged.OpenFile(filename, out s, 0x00000000);
      SetHandle(handle);
   }

   override protected bool ReleaseHandle()
   {
      return Unmanaged.CloseHandle(handle);
   }
}
```

Такой описатель можно затем использовать как поле улучшенного класса `FileWrapper` (листинг 12.32). Он по-прежнему реализует паттерн Disposable, как в листинге 12.29. Но поскольку теперь он не содержит неуправляемых ресурсов (ведь неуправляемый описатель файла скрыт внутри поля типа `CustomFileSafeHandle`), то финализатор необязателен. Явная очистка освободит наш описатель, ну а если мы забудем это сделать, об этом позаботится финализатор класса `CustomFileSafeHandle`.

Листинг 12.32 ❖ Простой пример использования ресурсов, обернутых классом `SafeHandle`

```
public class FileWrapper : IDisposable
{
   private bool disposed = false;
   private CustomFileSafeHandle handle;
   public FileWrapper(string filename)
   {
      Unmanaged.OFSTRUCT s;
      handle = Unmanaged.OpenFile(filename, out s, 0x00000000);
   }

   public void Dispose()
   {
      if (!disposed)
      {
         handle?.Dispose();
         disposed = true;
      }
   }

   public int UseMe()
   {
      byte[] buffer = new byte[1];
      if (Unmanaged.ReadFile(handle, buffer, 1, out uint read, IntPtr.Zero))
      {
         return buffer[0];
```

```
      }
      return -1;
   }
}
```

Отметим, что вызовы P/Invoke `OpenFile` и `ReadFile`, присутствующие в листинге 12.32, возвращают и принимают объекты типа `CustomFileSafeHandle` (листинг 12.33). Это возможно благодаря тому, что механизм маршалинга P/Invoke умеет трактовать производные от `SafeHandle` классы как скрытый внутри них `IntPtr`. Но как бы то ни было, это дает нам вышеупомянутую типобезопасность при работе с описателями.

Листинг 12.33 ❖ Методы P/Invoke, работающие с описателями, производными от `SafeHandle`

```
public static class Unmanaged
{
   [DllImport("kernel32.dll", BestFitMapping = false, ThrowOnUnmappableChar = true)]
   public static extern CustomFileSafeHandle OpenFile2([MarshalAs(Unmanaged Type.LPStr)]
       string lpFileName,
       out OFSTRUCT lpReOpenBuff,
       long uStyle);

   [DllImport("kernel32.dll", SetLastError = true)]
   public static extern bool ReadFile(CustomFileSafeHandle hFile,
       [Out] byte[] lpBuffer, uint nNumberOfBytesToRead,
       out uint lpNumberOfBytesRead, IntPtr lpOverlapped);
   ...
}
```

В нашем примере даже необязательно определять собственный подкласс `SafeHandle` для описателей файлов. Для часто используемых ресурсов уже есть готовые безопасные описатели:

- `SafeFileHandle` – безопасный описатель файла;
- `SafeMemoryMappedFileHandle` и `SafeMemoryMappedViewHandle` – безопасные описатели файлов, спроецированных на память;
- `SafeNCryptKeyHandle`, `SafeNCryptProviderHandle` и `SafeNCryptSecretHandle` – безопасные описатели криптографических ресурсов;
- `SafePipeHandle` – безопасный описатель канала;
- `SafeProcessHandle` – безопасный описатель процесса;
- `SafeRegistryHandle` – безопасный описатель раздела реестра;
- `SafeWaitHandle` – безопасный описатель ожидания (для синхронизации).

> Если вам интересна часть реализации `SafeHandle` внутри самой среды выполнения CoreCLR, то загляните в файл `.\src\vm\safehandle.cpp`.

Если в какой-то части вашего кода, относящейся к неуправляемым ресурсам, необходимо использовать `IntPtr` вместо `SafeHandle`, то можно получить исходный описатель, вызвав метод `DangerousGetHandle`. Но теперь возникает опасность утечки, поскольку простой необернутый `IntPtr` никак не отслеживается. Поэтому следует защищать простой описатель, полученный из `SafeHandle`, с помощью подсчета ссылок, который обеспечивают методы `DangerousAddRef` и `DangerousRelease`.

Широкое распространение знаний о существовании финализаторов на самом деле не так уж и желательно. В хорошо спроектированном коде редко приходится видеть, а еще реже писать собственные финализаторы. А в типичных ситуациях обычно хватает безопасных описателей, производных от `SafeHandle`.

Слабые ссылки

Есть еще один, пока не рассмотренный тип описателей, который реализует очень интересный тип корней, – *слабый описатель (weak handle)*. Концептуально слабый описатель очень прост – он хранит ссылку на объект, но не рассматривается как корень (не делает объект достижимым). Иначе говоря, на этапе пометки GC не просматривает слабые описатели, принимая решение о времени жизни объекта. Слабый описатель «жив», пока целевой объект достижим, но обнуляется, как только он становится недостижимым.

На самом деле есть два типа слабых описателей:

- *короткие слабые описатели* – они обнуляются перед выполнением финализаторов, когда GC решает, что объект мертв. Даже если финализатор воскресит объект, такой описатель останется равным нулю;
- *длинные слабые описатели* – их целевой объект остается действительным, когда переводится в старшее поколение вследствие финализации. Если финализатор воскресит объект, описатель останется действительным (будет указывать на тот же объект). Поэтому говорят, что такие описатели *отслеживают воскрешение*.

Создадим очень простой класс с опциональной реализацией воскрешения, который будет использоваться в последующих примерах (листинг 12.34).

Листинг 12.34 ❖ Класс, реализующий воскрешение в финализаторе

```
public class LargeClass
{
   private readonly bool ressurect;
   public LargeClass(bool ressurect) => this.ressurect = ressurect;
   ~LargeClass()
   {
      if (ressurect)
      {
         GC.ReRegisterForFinalize(this);
      }
   }
}
```

Для создания слабого описателя применяется метод `GCHandle.Alloc` с аргументом `GCHandleType.Weak` или `GCHandleType.WeakTrackResurrection` (листинги 12.35 и 12.36). Свойство `Target` указывает на целевой объект или равно `null`, если целевой объект уже убран сборщиком мусора (с учетом или без учета воскрешения).

Листинг 12.35 ❖ Пример короткого слабого описателя

```
var obj = new LargeClass(ressurect: true);
GCHandle weakHandle = GCHandle.Alloc(obj, GCHandleType.Weak);
GC.Collect();
```

```
GC.WaitForPendingFinalizers();
GC.Collect();
Console.WriteLine(weakHandle.Target ?? "<null>"); // печатается <null>
```

Листинг 12.36 ❖ Пример длинного слабого описателя

```
var obj = new LargeClass(ressurect: true);
GCHandle weakHandle = GCHandle.Alloc(obj, GCHandleType.WeakTrackResurrection);
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
Console.WriteLine(weakHandle.Target ?? "<null>"); // печатается CoreCLR.
                                                  // Finalization.LargeClass
```

Можно сказать, что короткий слабый описатель обнуляется в первый раз, как объект подлежит уборке (хотя потом он может и воскреснуть), тогда как длинный слабый описатель обнуляется, когда объект окончательно и бесповоротно убирается в мусор.

Кому может понадобиться такое странное создание, как слабая ссылка? Есть две ситуации, когда они бывают полезны:

- разного рода наблюдатели и прослушиватели (например, событий) – мы хотим хранить ссылку на объект, пока она используется кем-то еще, но не хотим, чтобы такое наблюдение влияло на состояние объекта;
- кеширование – мы можем создать кеш, в котором поначалу хранятся нормальные ссылки, но после некоторого периода неиспользования преобразуются в слабые. Поэтому вместо агрессивного вытеснения из кеша мы просто оставляем их до следующей сборки мусора в данном поколении (скорее всего, это будет поколение 2, поскольку объекты, которые были кешированы некоторое время, в конечном итоге там и окажутся). Контролируя время такого «слабого вытеснения из кеша», мы обеспечиваем компромисс между потреблением памяти (поскольку можем хранить объекты в кеше дольше) и накладными расходами на создание объектов (так как при попытке обратиться к вытесненному объекту его придется создать заново).

Существует очень интересный пример, иллюстрирующий «наблюдательную природу» слабых ссылок, – класс `Gen2GcCallback` в системной библиотеке .NET (листинг 12.37). После прочтения данной главы мы уже понимаем, что это критически финализируемый объект с опциональным воскрешением. Он наблюдает за данным целевым объектом, удерживая короткую слабую ссылку на него. При каждой финализации, т. е. при каждой сборке мусора в поколении, где находится целевой объект, выполняется заданный обратный вызов. После двух сборок мусора он окажется в поколении 2, т. е. является «обратным вызовом преимущественно в поколении 2» – он выполняется при каждой сборке в поколении 2 и при двух первых сборках в эфемерных поколениях (см. комментарий в начале листинга 12.37, где описаны возможные исправления[1]). Воскрешение прекращается, когда слабый

[1] Здесь мы ходим по тонкому люду, зависящему от деталей реализации – как именно объекты переводятся из поколения в поколение. Например, в текущей реализации, если целевой объект закреплен (или стал частью расширенного закрепленного блока), он может быть оставлен/возвращен в младшее поколение, и тогда наш обратный вызов будет снова выполнен при сборке мусора в эфемерном поколении.

описатель обнуляется, – следовательно, обратные вызовы для целевого объекта прекратятся после его смерти. Не будь ссылка слабой, этого никогда не случилось бы, потому что наш объект обратного вызова удерживал бы живую ссылку на целевой объект.

Класс `Gen2GcCallback` используется внутри класса `PinnableBufferCache`, чтобы при каждой сборке мусора в поколении 2 вызывался метод `TrimFreeListIfNeeded`.

Листинг 12.37 ❖ Интересный пример использования слабых ссылок и воскрешения из системной библиотеки

```
/// <summary>
/// Планирует обратный вызов приблизительно при каждой сборке мусора в поколении 2 (может
/// встретиться также поколение 0 или 1, но только один раз).
/// (Это можно исправить, запомнив счетчик сборок в поколении 2 при запуске и проверяя его,
/// но большой роли это не играет).
/// </summary>
internal sealed class Gen2GcCallback : CriticalFinalizerObject
{
   private Gen2GcCallback()
   {
   }

   public static void Register(Func<object, bool> callback, object targetObj)
   {
      // Создать недостижимый объект, который запоминает функцию обратного вызова
      // и целевой объект.
      Gen2GcCallback gcCallback = new Gen2GcCallback();
      gcCallback.Setup(callback, targetObj);
   }

   private Func<object, bool> _callback;
   private GCHandle _weakTargetObj;

   private void Setup(Func<object, bool> callback, object targetObj)
   {
      _callback = callback;
      _weakTargetObj = GCHandle.Alloc(targetObj, GCHandleType.Weak);
   }

   ~Gen2GcCallback()
   {
      // Проверить, жив ли еще целевой объект
      object targetObj = _weakTargetObj.Target;
      if (targetObj == null)
      {
         // Целевой объект мертв, поэтому данный объект обратного вызова больше не нужен
         _weakTargetObj.Free();
         return;
      }

      // Выполнить функцию обратного вызова.
      try
      {
         if (!_callback(targetObj))
```

```
            {
               // Если обратный вызов возвращает false, данный объект обратного вызова
               // больше не нужен
               return;
            }
         }
         catch
         {
            // Гарантируется шанс воскресить данный объект, даже
            // если функция обратного вызова вызовет исключение
         }

         // Воскресаем, заново зарегистрировавшись для финализации
         if (!Environment.HasShutdownStarted)
         {
            GC.ReRegisterForFinalize(this);
         }
      }
   }
```

Чтобы не создавать вручную слабый описатель GCHandle, были введены типы `WeakReference` и `WeakReference<T>` (листинги 12.38 и 12.39). Логика точно такая же, но поскольку это строго типизированное представление слабых описателей, лучше использовать их. Обратите внимание на смену названия – поскольку слабые описатели реализуют семантику слабых ссылок, было выбрано именно такое имя, чтобы скрыть детали реализации (не всем интересно, что на внутреннем уровне слабая ссылка представлена слабым описателем).

Класс `WeakReference` принимает целевой объект и предоставляет три важных члена:

- `IsAlive` – проверить, жив ли еще целевой объект;
- `Target` – получить ссылку на целевой объект;
- `TrackResurrection` – проверить, должна ли слабая ссылка оставаться действительной после воскрешения.

Но у этого API есть одна небольшая проблема, продемонстрированная в листинге 12.38. Между вызовами `weakReference.IsAlive` и `weakReference.Target` может произойти сборка мусора, которая уберет целевой объект и сделает результат проверки бесполезным. Кроме того, потеря информации о типе (мы храним ссылку на тип `object`) – плохая практика проектирования, поскольку для использования целевого объекта приходится выполнять приведение типа.

Листинг 12.38 ❖ Пример использования типа `WeakReference`

```
var obj = new LargeClass(ressurect: true);
WeakReference weakReference = new WeakReference(obj, trackResurrection: false);
if (weakReference.IsAlive)
   Console.WriteLine(weakReference.Target ?? "<null>"); // печатается <null>
```

Поэтому в версии .NET Framework 4.5 появился обобщенный вариант этого типа, в котором также был пересмотрен API. Теперь существует только метод `TryGetTarget`, который атомарно возвращает информацию о том, жив ли целевой объект (листинг 12.39).

Листинг 12.39 ❖ Пример использования типа `WeakReference<T>`

```
var obj = new LargeClass(ressurect: true);
WeakReference<LargeClass> weakReference = new
WeakReference<LargeClass>(obj, trackResurrection: false);
if (weakReference.TryGetTarget(out var target))
   Console.WriteLine(target);
```

Заметим, что слабую ссылку легко преобразовать в сильную, присвоив ее целевой объект какому-нибудь достижимому корню. Именно такой подход используется во внутреннем классе `System.StrongToWeakReference<T>` (листинг 12.40). Это слабая ссылка, которая опционально хранит сильную ссылку на целевой объект. Сделать такую пару слабой ссылкой так же просто, как присвоить `null` сильной ссылке. Можно еще попытаться вернуться к сильной ссылке, если целевой объект слабой ссылки еще жив. Конечно, это может не получиться, если целевой объект уже убран в мусор (поэтому я бы предпочел назвать метод `bool TryMakeStrong()`, а не `MakeStrong`, как во внутреннем классе).

Листинг 12.40 ❖ Класс `StrongToWeakReference` как пример преобразования между слабой и сильной ссылками

```
internal sealed class StrongToWeakReference<T> : WeakReference where T : class
{
   private T _strongRef;

   public StrongToWeakReference(T obj) : base(obj)
   {
      _strongRef = obj;
   }

   public void MakeWeak() => _strongRef = null;

   public void MakeStrong()
   {
      _strongRef = WeakTarget;
   }

   public new T Target => _strongRef ?? WeakTarget;
   private T WeakTarget => base.Target as T;
}
```

Теперь кратко рассмотрим два самых типичных применения слабых ссылок: кеширование и прослушиватели событий.

Кеширование

Когда слышишь или читаешь о слабых ссылках, на ум сразу приходит кеширование. Соблазнительно удерживать объекты в памяти с помощью такого «слабого кеша». Объект используется как обычно, но существует дополнительная слабая ссылка, поэтому мы можем кешировать его, не продлевая жизнь из-за нахождения в кеше. Пока целевой объект жив, слабая ссылка в кеше тоже жива, но поскольку эта ссылка слабая, объект умирает как обычно, как только становится не нужен приложению. Таким образом, мы кешируем объекты, которые в данный момент используются приложением (например, чтобы не создавать дубликаты,

если объект понадобится в другом месте программы). Это может быть полезно само по себе.

Однако чаще всего недавно использованные ресурсы остаются в кеше в течение некоторого времени даже после того, как перестали быть нужными. Очевидно, что с помощью слабых ссылок этого не добиться. В таком случае вы, наверное, захотите реализовать обычный кеш, в котором сильные ссылки хранятся в течение некоторого времени – абсолютного или зависящего от момента последнего использования. По истечении этого времени ссылки удаляются (*вытесняются* из кеша).

Но можно вместо этого представить себе нечто вроде *кеша со слабым вытеснением*, когда после истечения заданного времени сильные ссылки преобразуются в слабые. Это смягчает политику кеширования – мы безусловно храним объект в кеше в течение указанного времени, а сверх того – если он еще используется. Иными словами, при такой политике живые объекты не вытесняются раньше времени – вместо принудительного удаления из кеша по тайм-ауту объект остается там до тех пор, пока используется. При работе с обычным кешем объекты безусловно удаляются по истечении заданного времени, поскольку без слабых ссылок невозможно проверить, жив еще объект или уже нет (если, конечно, не предусмотрен API, информирующий кеш о том, что объект еще используется, но для обсуждаемого здесь кеша объектов общего вида это маловероятно).

Предположим, что у нас есть обобщение класса `StrongToWeakReference` из листинга 12.40, в котором хранится момент времени, когда ссылка стала сильной (в поле `StrongTime`). Имея такой вспомогательный класс, мы можем предложить сильно упрощенный дизайн кеша со слабым вытеснением, показанный в листинге 12.41. Мы просто храним словарь кешированных объектов в виде наших гибридных объектов, содержащих сильную и слабую ссылки. Вначале объекты сохраняются в виде сильных ссылок. Периодически должен вызываться метод `DoWeakEviction`, который преобразует сильные ссылки в слабые (и удаляет из кеша уже мертвые объекты).

Листинг 12.41 ❖ Кеш со слабым вытеснением, в котором по истечении заданного времени хранятся слабые ссылки

```
public class WeakEvictionCache<TKey, TValue> where TValue : class
{
   private readonly TimeSpan weakEvictionThreshold;
   private Dictionary<TKey, StrongToWeakReference<TValue>> items;

   WeakEvictionCache(TimeSpan weakEvictionThreshold)
   {
      this.weakEvictionThreshold = weakEvictionThreshold;
      this.items = new Dictionary<TKey, StrongToWeakReference<TValue>>();
   }

   public void Add(TKey key, TValue value)
   {
      items.Add(key, new StrongToWeakReference<TValue>(value));
   }

   public bool TryGet(TKey key, out TValue result)
   {
      result = null;
```

```
      if (items.TryGetValue(key, out var value))
      {
         result = value.Target;
         if (result != null)
         {
            // Объектом воспользовались, пытаемся сделать ссылку снова сильной
            value.MakeStrong();
            return true;
         }
      }
      return false;
   }

   public void DoWeakEviction()
   {
      List<TKey> toRemove = new List<TKey>();
      foreach (var strongToWeakReference in items)
      {
         var reference = strongToWeakReference.Value;
         var target = reference.Target;
         if (target != null)
         {
            if (DateTime.Now.Subtract(reference.StrongTime) >= weakEvictionThreshold)
            {
               reference.MakeWeak();
            }
         }
         else
         {
            // Удалить уже обнуленные слабые ссылки
            toRemove.Add(strongToWeakReference.Key);
         }
      }
      foreach (var key in toRemove)
      {
         items.Remove(key);
      }
   }
}
```

Имейте в виду, что класс `WeakEvictionCache` очень простой, и чтобы использовать его в реальной программе, нужны серьезные доработки (в частности, предоставить улучшенный API и обеспечить потокобезопасность).

Паттерн слабых событий

Еще один типичный сценарий применения слабых ссылок – *слабые события*. Пользоваться событиями в .NET нетрудно, но они могут стать одним из самых распространенных источников утечки памяти. Прежде чем описывать решение с помощью слабых событий, рассмотрим подробно, как это может случиться.

Сначала напишем два тривиальных класса, моделирующих оконную библиотеку (будь то Windows Forms, WPF или что-то еще) (см. листинг 12.42). Они ил-

люстрируют весьма популярный иерархический подход к построению таких библиотек – почти каждый элемент находится в отношении родитель–потомок с каким-то другим. Кроме того, очень часто реализуется подписка на события, генерируемые элементами. Для экспериментов мы подготовили событие SettingsChanged и метод RegisterEvents для подписки на него.

Листинг 12.42 ❖ Два простых класса, моделирующих библиотеку пользовательского интерфейса

```
public class MainWindow
{
   public delegate void SettingsChangedEventHandler(string message);
   public event SettingsChangedEventHandler SettingsChanged;
}

public class ChildWindow
{
   private MainWindow parent;

   public ChildWindow(MainWindow parent)
   {
      this.parent = parent;
   }

   public void RegisterEvents(MainWindow parent)
   {
      // ChildWindow - цель, MainWindow - источник
      parent.SettingsChanged += OnParentSettingsChanged;
   }

   private void OnParentSettingsChanged(string message)
   {
      Console.WriteLine(message);
   }
}
```

В коде в листинге 12.43 эти типы используются. Моделируется работа типичного приложения с пользовательским интерфейсом – есть одно главное окно, и время от времени создаются дополнительные дочерние окна, выполняющие какую-то работу. Дочерние окна подписываются на некоторые события родительского окна. На каждой итерации мы запускаем GC, чтобы агрессивно собирать весь накопившийся мусор. Кроме того, для целей диагностики мы ведем список слабых ссылок, в котором будут отслеживаться все созданные дочерние окна (обратите внимание, как хорошо класс WeakReference приспособлен для этой задачи).

Листинг 12.43 ❖ Эксперимент, демонстрирующий утечку памяти из-за событий, от которых никто не отписался

```
public void Run()
{
   List<WeakReference> observer = new List<WeakReference>();

   MainWindow mainWindow = new MainWindow();
   while (true)
   {
```

```
      Thread.Sleep(1000);
      ChildWindow childWindow = new ChildWindow(mainWindow);
      observer.Add(new WeakReference(childWindow));

      childWindow.RegisterEvents(mainWindow); // Оставить эту строку незакоментированной,
                                              // чтобы организовать утечку дочерних окон
      childWindow.Show();

      GC.Collect();
      foreach (var weakReference in observer)
      {
         Console.Write(weakReference.IsAlive ? "1" : "0");
      }
      Console.WriteLine();
   }
}
```

Понятно, что если бы вызов `RegisterEvents` был закомментирован, то экземпляр дочернего окна стал бы недостижимым еще до вызова `GC.Collect` благодаря технике ранней сборки корней. Поэтому результат соответствовал бы ожиданиям (листинг 12.44). Каждое дочернее окно умирает после очередной итерации.

Листинг 12.44 ❖ Результат работы программы из листинга 12.43 (если бы вызов `RegisterEvents` был закомментирован)

```
Показаны ChildWindow
0
Показаны ChildWindow
00
Показаны ChildWindow
000
Показаны ChildWindow
0000
Показаны ChildWindow
00000
```

Однако подписка на событие вносит очевидную утечку памяти (листинг 12.45). В памяти накапливается все больше и больше живых дочерних окон.

Листинг 12.45 ❖ Результат работы программы из листинга 12.43 (когда вызов `RegisterEvents` выполнен)

```
Показаны ChildWindow
1
Показаны ChildWindow
11
Показаны ChildWindow
111
Показаны ChildWindow
1111
Показаны ChildWindow
11111
```

Очевидно, что у этой проблемы есть очень простое решение – в какой-то момент должен быть вызван метод `UnregisterEvents`, в котором используется опера-

тор -= для отписки от событий родительского окна. Это легко, но требуется, чтобы программист не забывал явно отписываться от каждого события. Мы еще вернемся к этому вопросу чуть позже. А пока разберемся в причинах данной утечки памяти.

Подписка на событие – умеренно сложный процесс. Определенный в классе делегат представлен вложенным классом, производным от типа `System.MulticastDelegate` (листинг 12.46). Как видим, его конструктор принимает объект и метод, поскольку делегату нужна информация о том, что вызывать (метод) и от чьего имени (объект).

Листинг 12.46 ❖ Внутренняя реализация `SettingsChangedEventHandler`

```
.class public auto ansi beforefieldinit CoreCLR.Finalization.MainWindow
   extends [System.Runtime]System.Object
{
   // Вложенные типы
   .class nested public auto ansi sealed SettingsChangedEventHandler
      extends [System.Runtime]System.MulticastDelegate
   {
      // Методы
      .method public hidebysig specialname rtspecialname
         instance void .ctor (
            object 'object',
            native int 'method'
         ) runtime managed
      {
      } // конец метода SettingsChangedEventHandler::.ctor

      ...

      .method public hidebysig newslot virtual
         instance void Invoke (
            string message
         ) runtime managed
   } // конец метода SettingsChangedEventHandler::Invoke
} // конец класса SettingsChangedEventHandle
```

Именно это происходит внутри метода `RegisterEvents` (листинг 12.47). Конструктору `SettingsChangedEventHandler` передается поле `this` (ссылка на `ChildWindow`), и вызывается метод `add_SettingsChanged`, который включает делегат в текущий список вызова делегатов (листинг 12.48).

Листинг 12.47 ❖ Представление `RegisterEvents` на языке CIL

```
.method public hidebysig
   instance void RegisterEvents (
      class CoreCLR.Finalization.MainWindow parent
   ) cil managed
   {
      .maxstack 8

      IL_0000: ldarg.1 // parent
      IL_0001: ldarg.0 // this
      IL_0002: ldftn instance void ChildWindow::OnParentSettingsChanged(string)
```

```
      IL_0008: newobj instance void MainWindow/
               SettingsChangedEventHandler::.ctor(object, native int)
      IL_000D: callvirt instance void MainWindow::add_SettingsChanged
               (class CoreCLR.Finalization.MainWindow/SettingsChangedEventHandler)
      IL_0012: ret
} // конец метода ChildWindow::RegisterEvents
```

Листинг 12.48 ❖ Внутренняя реализация события `SettingsChanged` (сильно упрощенная, обеспечение потокобезопасности для краткости опущено)

```
public event MainWindow.SettingsChangedEventHandler SettingsChanged
{
   [CompilerGenerated]
   add
   {
      // value имеет тип SettingsChangedEventHandler (и в нашем примере
      // содержит ссылку на ChildWindow)
      this.SettingsChanged = (MainWindow.SettingsChangedEventHandler)
      Delegate.Combine(this.SettingsChanged, value);
   }
   remove
   {
      ...
   }
}
```

Таким образом, экземпляры `ChildWindow` накапливаются в списке вызова делегатов, представляющем событие `SettingsChanged`. Иными словами, событие становится их единственным корнем и продолжает удерживать их живыми, хоть большая их часть, вероятно, должна уже умереть. А еще вероятнее, что экземпляры `ChildWindow` больше не интересуются событием `SettingsChanged`. Это попросту ошибка, ведущая к утечке памяти, а насколько она серьезна, зависит от того, на какое время источник событий переживет целевые объекты. Худший случай – статические события (или события в статических классах). Они живут столько же, сколько их домен приложения (как правило, на протяжении всего времени работы программы), поэтому времени для образования значительной утечки памяти более чем достаточно.

Чем дольше живет источник по сравнению с целевыми объектами и чем больше памяти занимают целевые объекты, тем серьезнее такая утечка памяти. Мне встречались приложения, работающие много дней, в которых из-за неотмененной подписки на статическое событие к утечке памяти приводили очень маленькие объекты. Но я видел также, как очень большие объекты убивали приложение за несколько часов по той же причине.

> Обратите внимание, что в нашем примере событие намеренно определено не вполне типичным образом. Обычно первый аргумент представляет источник события (и называется `sender`):
>
> ```
> public delegate void SettingsChangedEventHandler(object sender, string message);
> ```
>
> Но то, что отправитель берется из экземпляра `MulticastDelegate`, никак не влияет на утечку памяти. Я отметил этот факт, просто чтобы убедить вас в том, что не этот аргумент, сильно связывающий источник и целевой объект, приводит к утечке памяти.

Так как же решить проблему? Теперь, когда вы знаете о слабых ссылках, решение должно показаться очевидным. Связь между исходным и целевым объектами должна описываться слабой ссылкой – нет никакой необходимости хранить ее, если последний умирает, и наоборот.

Однако полная и корректная реализация такого «слабого события» нетривиальна. Исчерпывающее описание заняло бы слишком много места. Вместо этого мы кратко рассмотрим, как этот паттерн реализован в Windows Presentation Foundation, где слабые события можно определять явно.

К сожалению, изящный и лаконичный синтаксис обработки событий, принятый в C# (с помощью операторов += и -=), невозможно распространить на синтаксис слабых событий. Поэтому аналогичный по существу API основан на вызовах методов. Так, если бы наше приложение с графическим интерфейсом было написано на WPF, то мы могли бы подписаться на слабое событие в методе `RegisterEvents`, как показано в листинге 12.49. В WPF есть разные способы это сделать, но самый предпочтительный – использовать статический метод `AddHandler` обобщенного класса `WeakEventManager`, который связывает все воедино: определяет, что нас интересует событие `SettingsChanged` родительского объекта и что в ответ на него нужно вызвать обработчик `OnParentSettingsChanged` (целевой объект неявно берется из скрытого внутри делегата).

Листинг 12.49 ❖ Использование слабых событий в WPF

```
public void RegisterEvents(MainWindow parent)
{
   // ChildWindow - целевой объект
   // MainWindow - источник
   WeakEventManager<MainWindow, string>.AddHandler(parent,
      "SettingsChanged", OnParentSettingsChanged);
}
```

Изучение реализации класса `WeakEventManager` может оказаться очень полезным. Даже начальный комментарий к нему содержит интереснейшие детали (листинг 12.50).

Листинг 12.50 ❖ Начальный комментарий в исходном файле WeakEventManager.cs

```
// Обычно A начинает прослушивание, добавляя обработчик события к событию Foo объекта B:
// B.Foo += new FooEventHandler(OnFoo);
// но обработчик содержит сильную ссылку на A, поэтому B теперь также хранит сильную
// ссылку на A. (...)
// Образуется утечка памяти, и решение заключается в том, чтобы ввести промежуточный
// прокси-объект P, обладающий следующими свойствами:
// 1. Именно P прослушивает B.
// 2. P хранит список "настоящих прослушивателей", например A, с помощью слабых ссылок.
// 3. Получив событие, P переправляет его тем настоящим прослушивателям, которые еще живы.
// 4. Ожидается, что время жизни P совпадает со временем жизни приложения (или диспетчера).
```

> Если вы хотите попрактиковаться в работе со слабыми ссылками, настоятельно рекомендую изучить паттерн слабых событий в WPF. Одна из основных частей класса `WeakEventManager` – таблица `WeakEventTable`. Посмотрите на структуру `Listener`, которая содержит слабую ссылку на целевой объект, и на структуру `EventKey`, содержащую слабую ссылку на источник.

Почему же в реализации событий в .NET по умолчанию не используется паттерн слабых событий? Разве этот отказ от явной очистки событий не был бы в духе автоматического управления памятью? Основная причина – соотношение между потенциальным снижением производительности и получаемым в обмен удобством API. Для слабых событий необходимы слабые описатели, а с ними связаны накладные расходы, уменьшающие производительность и повышающие потребление памяти. Использование событий ничем не ограничено – даже если в типичном случае мы ожидаем всего десятка событий, связанных с графическим интерфейсом, проектировать надо в расчете на сотни экземпляров. Поэтому гораздо безопаснее использовать обычные члены экземпляра (а именно таковыми события, по существу, и являются), чем идти на накладные расходы, сопряженные с описателями.

Да и вообще, все это нужно было бы делать только для того, чтобы освободить ленивого программиста от необходимости думать об отписке от событий. В большинстве случаев момент, когда следует отписаться от события, четко определен. В MSDN о слабых событиях в WPF сказано следующее: «Паттерн слабых событий обычно имеет смысл использовать, когда время жизни источника события не зависит от прослушивателей события. Применение централизованного механизма диспетчеризации событий в виде класса `WeakEventManager` позволяет убирать обработчики событий в мусор, даже если источник существует долго». Случаи независимого существования источника и прослушивателей встречаются довольно редко, поэтому явная очистка была сочтена гораздо лучшим решением. Но все-таки было бы хорошо в качестве дополнительного бонуса иметь возможность использовать лаконичный синтаксис событий в C#.

> Если вы хотите изучить реализацию слабых ссылок в исходном коде CoreCLR, начните с метода `WeakReferenceNative::Create`, который создает описатели типа `HNDTYPE_WEAK_LONG` или `HNDTYPE_WEAK_SHORT` в обычном хранилище описателей. На этапе пометки метод `GCScan::GcShortWeakPtrScan` обнуляет целевые объекты коротких слабых ссылок, которые не были переведены в старшее поколение. Затем после просмотра корней финализации обнуляются также целевые объекты долгих слабых ссылок, для чего вызывается метод `GCScan::GcWeakPtrScan`.

Сценарий 9.2. Утечка памяти из-за событий

Описание. Наше приложение со временем потребляет все больше памяти. Проверка, например, с помощью счетчиков производительности показала, что растет управляемая куча. В поколении 2 скапливается все больше объектов, но фрагментация в нем стабильна (проверено с помощью сеансов PerfView). Очевидно, имеет место утечка памяти, т. к. какие-то объекты остаются достижимыми из-за некоторого пока не идентифицированного корня.

Воспользуемся кодом из листинга 9.43 для моделирования такой ситуации. Конечно, мы уже знаем, в чем источник проблемы. Но притворимся, что это не так, и продемонстрируем, как можно было бы провести диагностику.

Анализ. В процессе анализа утечки памяти есть два основных пути.

- Создать один дамп памяти в момент, когда потребление очень велико. Можно предполагать, что объекты, которые привели к утечке памяти, чем-то выделяются: количеством, суммарным размером, часто встречаются в очереди финализации (если нам повезло и такие объекты финализируемые)

и т. д. Иногда это единственный возможный подход – например, если утечка происходит очень редко и у нас был всего один шанс создать дамп в производственной среде. Но анализировать такие дампы утомительно – в основном потому, что характеристики утечки могут быть сложнее, чем одиночная утечка большого объекта. Возможно, существует запутанный подграф мелких объектов, связанных друг с другом и удерживаемых непонятно какими корнями, которые прячутся в огромном графе всех объектов. Поэтому для анализа такого единственного дампа требуется хорошо развитая интуиция, хоть какое-то знание внутреннего устройства приложения (чтобы быстро выявить ожидаемые подграфы объектов), ну и, конечно, немного удачи.

- Создать два или более последовательных дампов памяти и проанализировать их отличия (желательно автоматически). Если возможно, следует предпочесть этот подход. Сравнение состояний приложения в близкие моменты времени устраняет из анализа шум – объекты, из-за которых произошла утечка памяти, должны чем-то отличаться от остальных, память для которых выделяется и освобождается стабильно. В этой книге описывались различные инструменты для этой цели. Я предпочитаю проводить низкоуровневый анализ в WinDbg, но тогда сравнивать приходится вручную. Гораздо лучше сравнивать снимки кучи, созданные в PerfView, поскольку накладные расходы при этом невелики и есть хорошая поддержка анализа отличий. Разумеется, все коммерческие инструменты поддерживают такой подход, поскольку нет лучшего способа найти источник утечки памяти.

Воспользуемся сравнением снимков куч, предлагаемых PerfView. Во время работы проблематичного приложения создайте два последовательных снимка кучи (с помощью команды **Memory ➤ Take Heap Snapshot**), в то время когда память процесса заметно растет (чтобы можно было увидеть объекты, вызвавшие утечку). Я всегда предпочитаю делать такие снимки, после того как приложение некоторое время проработало, чтобы дать ему шанс прогреться и освободить память, занятую на стадии инициализации, что бывает довольно часто.

Открыв оба снимка, сравните их командой **Diff ➤ With baseline...**. Решайте сами, как анализировать результат сравнения: в представлении ByName (отсортировав по столбцу Inc или Exc), RefTree или визуально на пламенной диаграмме. Иногда один лишь взгляд на пламенные диаграммы дает достаточно информации. В нашем примере сразу видно, что наибольший вклад в различие снимков дает тип `MainWindow`, который хранит ссылки на `SettingsChangedEventHandler`, который, в свою очередь, удерживает экземпляр `ChildWindow` (см. рис. 12.5). Мы нашли очень серьезного подозреваемого!

Изучение представления RefTree подтверждает наше предположение – за время между двумя снимками образовалось свыше трех сотен новых экземпляров `ChildWindow` и `SettingsChangedEventHandler` (рис. 12.6).

Есть надежда, что этот анализ прямо укажет на проблематичный обработчик события в приложении. Обратите внимание на дополнительный массив `Object[]`, используемый в объектах `SettingsChangedEventHandler`. Это не что иное, как вышеупомянутый список вызовов – поскольку у делегата `SettingsChangedEventHandler` тип `MulticastDelegate` (да, делегат такого вида хранит внутри себя массив прослушивающих объектов, которые сами являются делегатами; загляните в исходный код класса `MulticastDelegate`, если вас интересуют детали).

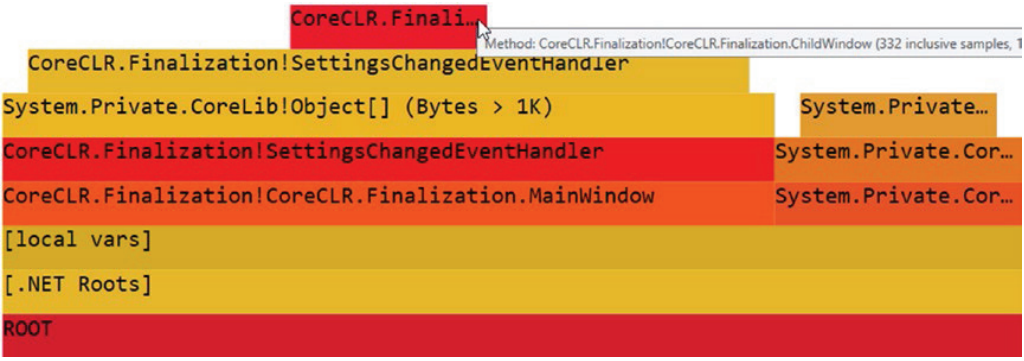


Рис. 12.5 ❖ Пламенная диаграмма различия двух снимков кучи, сделанных в PerfView

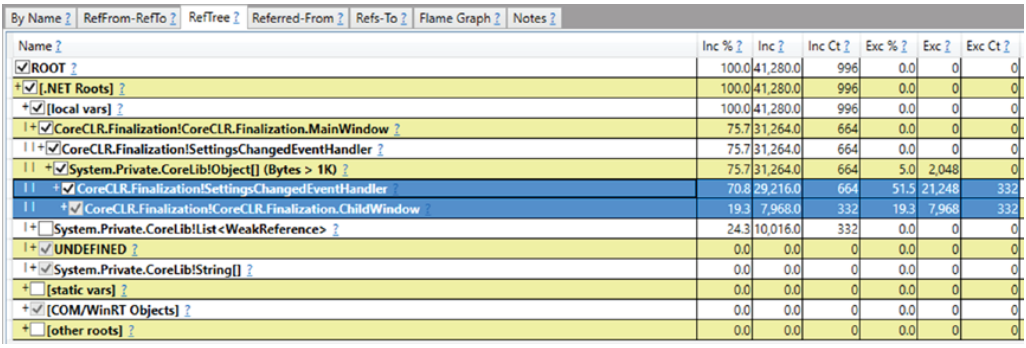


Рис. 12.6 ❖ Представление RefTree двух снимков кучи, сделанных в PerfView

В качестве примера того, как подобная информация представлена в коммерческих инструментах, на рис. 12.7 продемонстрировано сравнение снимков кучи в .NET Memory Profiler (два последовательных снимка были созданы во время сеанса профилирования работающего приложения). Очевидно, что результаты такие же – указывают на те же проблематичные обработчики событий. Мы также видим увеличение количества описателей `GCHandle`, удерживаемых слабыми ссылками, но это ожидаемо, поскольку они накапливаются в нашем списке наблюдателей (см. листинг 12.43).

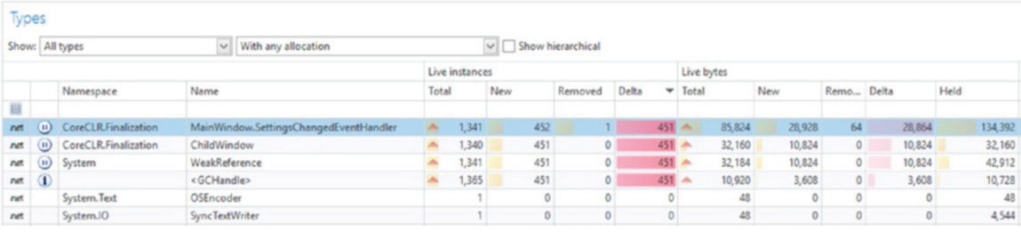


Рис. 12.7 ❖ Отличия двух снимков кучи в .NET Memory Profiler

Как уже было сказано, аналогичные отчеты имеются во всех доступных коммерческих инструментах (тем самым я отвожу от себя обвинения в рекламе конкретной программы).

Резюме

Финализация и уничтожаемые объекты тесно связаны с неуправляемым миром. Они в большей степени относятся к управлению ресурсами, чем к управлению временем жизни объектов. Однако между этими вопросами – и слабыми ссылками – есть трудноуловимые связи.

Уничтожаемые объекты введены с помощью стандартизации явной очистки ресурсов в виде интерфейса `IDisposable` и поддержки со стороны оператора using в языке C#. Тем самым это своего рода замена отсутствующего подхода RAII (захват ресурса есть инициализация), применяемого в неуправляемом мире, смысл которого в том, что локальная переменная в своей лексической области видимости является владельцем некоторого ресурса: ресурс захватывается в момент ее создания (в конструкторе) и освобождается, когда она покидает область видимости (в деструкторе). Хотя интерфейс `IDisposable` с самого начала задумывался именно для этой цели, он обрел популярность и в других случаях. Протоколирование, трассировка, профилирование – вот лишь несколько примеров, никак не связанных с неуправляемыми ресурсами. Этот интерфейс полезен всякий раз, как требуется явная область управления. Ну и конечно, явная очистка остается предпочтительным способом освобождения неуправляемых ресурсов.

С другой стороны, финализация все еще весьма популярна, особенно в контексте полной реализации паттерна Disposable, когда она нужна для страховки на случай, если явная очистка отсутствует. Но необходимо ясно понимать все подводные камни и накладные расходы финализации. Надеюсь, что детали реализации, проиллюстрированные в тестах производительности и в сценарии 9.1, хотя бы немного убедили вас в этом. Общее правило – по возможности избегать финализации. Не нужно рассматривать ее как способ добавить наворотов в протоколирование или что-то еще такое, чтобы ваш код казался хитроумнее!

Слабые ссылки – пожалуй, наименее популярный тип из описанных в этой главе. Они применяются лишь в нескольких ситуациях, и в своем коде вы их, наверное, использовать не будете. Однако знать о них стоит, особенно в связи с популярным паттерном слабого события. Также они полезны для необычных экспериментов с кодом, поскольку предоставляют единственный простой способ проверить достижимость объекта (если вам нужна такая возможность).

Эта глава завершает наиболее важную часть книги, посвященную внутренним механизмам управления памятью в .NET. Мы проделали долгий путь. Следующие две главы имеют более практическую направленность и основаны на полученных знаниях. Очень рекомендую прочитать их!

Правило 25: избегайте финализаторов

Применимость. Носит общий характер, популярно. Важно в коде, требующем высокой производительности.

Обоснование. Финализаторы проектировались для вполне определенной цели – обеспечить неявную очистку неуправляемых ресурсов на случай, когда явная невозможна. Однако я могу себе представить не так много случаев, когда явная очистка невозможна. Используя финализатор, мы создаем себе кучу проблем. Корректно реализовать финализатор совсем не просто, если принимать во внимание разного рода граничные случаи (реентерабельность, многопоточность, возможность частичного выполнения или невыполнения вовсе – и это лишь небольшая часть возможных неприятностей). Кроме того, из-за неизбежных особенностей реализации имеются значительные накладные расходы, в основном в части производительности и потребления памяти.

Как применять. Просто попробуйте альтернативы, а именно:

- `SafeHandle` – хорошо спроектированное финализируемое представление описателя, поддержанное средой выполнения;
- паттерн Disposable – вполне вероятно, что можно избавиться от финализации и управлять ресурсом явно;
- критическая финализация – если освобождение ресурса абсолютно необходимо.

В тех случаях, когда вы не видите возможности обойтись без финализатора, помните о следующих рекомендациях:

- пишите простые обертки, инкапсулирующие неуправляемые ресурсы и не хранящие ссылок на другие управляемые ресурсы – чтобы не слишком много объектов переводилось в старшее поколение из-за финализации;
- избегайте выделения памяти в финализаторе и критическом финализаторе – выброс исключения `OutOfMemoryException` внутри финализатора может привести к очень серьезным проблемам;
- всегда проверяйте, действительно ли вы владеете ресурсом – типична ситуация, когда в конструкторе возникает исключение, и в результате финализатор выполняется в состоянии, когда объект инициализирован не полностью;
- избегайте зависимости от контекста потока – не делайте никаких предположений о том, в каком потоке выполняется финализатор. Сюда же относится рекомендация избегать блокировки выполнения финализатора при помощи любых способов синхронизации;
- не бросайте никаких исключений в финализаторах и не позволяйте бросать их стороннему коду. Код финализатора всегда следует заключать в блок try-finally;
- избегайте вызова виртуальных функций из финализатора, поскольку это может привести ко всем описанным выше видам нежелательного поведения.

Иллюстрирующие сценарии: 12.1.

Правило 26: отдавайте предпочтение явной очистке

Применимость. Носит общий характер, популярно. Важно в коде, требующем высокой производительности.

Обоснование. Детерминированная очистка – предпочтительный способ управления ресурсами. Момент очистки точно определен, и (если проект разра-

ботан правильно) очистка производится максимально быстро – а это хорошо для управления ограниченными ресурсами. Понятно, что к программисту предъявляются повышенные требования. Больше нельзя создать ресурс и забыть о нем. Приходится думать об освобождении всех захваченных ресурсов. Да, мы знаем, что это несколько расходится с обещаниями, которые давала управляемая среда, и прежде всего с обещанием автоматически управлять памятью. Но неуправляемые ресурсы … неуправляемы. И стоит приложить немного усилий, чтобы помнить об этом.

Как применять. Пользуйтесь средствами, предлагаемыми экосистемой .NET, – интерфейсом `IDisposable` и уничтожаемыми объектами. Очистить ресурс чаще всего можно в методе `Dispose`, а не в специальном обремененном накладными расходами финализаторе. Потребуются дополнительные усилия от программиста, но на помощь ему готовы прийти оператор using в C#, а также инструменты вроде ReSharper или правил Visual Studio.

Глава 13

Разное

До сих пор нас интересовали различные аспекты управления памятью в .NET (т. е. по большей части работа сборщика мусора). И сейчас наших знаний достаточно для глубокого понимания работы большинства внутренних механизмов. Я говорю «большинства», потому что есть менее значимые вопросы, которых мы не касались из-за ограниченного объема книги. Надеюсь, однако, что вы уже уверенно можете говорить о разделении (на поколения и сегменты), о выделении и освобождении памяти, о том, как производится сборка мусора и т. д.

Все эти сведения перемежались практическими советами и сценариями (обычно связанными с диагностикой). Но чтобы изложение оставалось ясным, а главы сохраняли разумный размер, не все сложные темы были упомянуты. Вот именно им и посвящены эта и следующая главы. Будем считать их «сливками» управления памятью в .NET – чисто практическими (с отсылкой к знаниям о внутреннем устройстве) вопросами, касающимися более продвинутых тем. Это не значит, что обсуждаемые здесь темы бесполезны в повседневной работе. Совсем напротив, возможно, мы станем свидетелями все более широкого применения описываемых приемов, потому что на .NET пишется все больше высокопроизводительного кода. Особенно это относится к классу `Span<T>` и всему, что его окружает.

Поскольку эта глава дополнительная и не посвящена какой-то конкретной теме, она носит конспективный характер, а отдельные ее разделы слабо связаны между собой. Выбирайте те, что вам наиболее интересны, или читайте подряд (я рекомендую поступить именно так).

Зависимые описатели

Помимо уже известных нам описателей, существует еще один, который мы до сих пор не упоминали, – зависимый описатель, добавленный в версии .NET Framework 4.0 (также доступный и в .NET Core). Он позволяет нам связать время жизни двух объектов. Зависимый описатель указывает на целевой объект – точно так же, как другие описатели GC. И ведет он себя как слабая ссылка, т. е. не удерживает объект живым. Это первичный объект зависимого описателя, но, кроме того, он еще хранит вторичный объект. Поведение зависимого описателя подчиняется следующим правилам:

- он является «слабой» ссылкой по отношению к первичному и вторичному объектам (сам по себе он не влияет на их время жизни);
- существует сильная ссылка между первичным и вторичным объектами (вторичный объект остается живым, пока жив первичный).

В результате получаем очень гибкое средство, позволяющее, например, динамически «добавлять» поля в объекты. Собственно говоря, именно «добавление полей» и является целью зависимого описателя, как мы скоро увидим.

Зависимые описатели, в отличие от других типов описателей, недоступны с помощью API GCHandle. На самом деле для работы с ними нет вообще никакого открытого API, а единственный способ воспользоваться ими дает класс-обертка `ConditionalWeakTable`. В комментарии к его исходному коду говорится, что он «предоставляет поддержку полей объекта, генерируемых во время выполнения, со стороны компилятора», и «дает возможность DLR и компиляторам других языков добавлять произвольные "свойства" к управляемым объектам, созданным на этапе выполнения программы».

> Существует внутренняя структура `DependentHandle` (в пространстве имен `System.Runtime.CompilerServices`), которая оборачивает зависимый описатель на уровне среды выполнения. У нее есть простой конструктор `DependentHandle(object primary, object secondary)` и методы `GetPrimary` и `GetPrimaryAndSecondary`. Но эта структура является внутренней, и было принято решение не давать к ней доступа снаружи. Она используется в вышеупомянутом классе `ConditionalWeakTable`.
>
> Интересно также, что тип зависимого описателя используется средой выполнения для поддержки добавления полей при выполнении таких операций отладчика, как **Edit** и **Continue** (Редактировать и Продолжить). Поскольку экземпляры модифицированного типа могут уже находиться в куче, операция не может просто изменить размещение объекта в памяти, включив новое поле. Поэтому зависимый описатель связывает время жизни старого и нового объектов.

Класс `ConditionalWeakTable` организован как словарь, в котором ключ – первичный объект, а значение – добавленное «свойство» (вторичный объект). Отметим, что эти ключи словаря – слабые ссылки, которые не удерживают объекты живыми (в отличие от ключей обычного словаря). После того как ключ умирает, словарь автоматически удаляет соответствующую запись.

API класса `ConditionalWeakTable` интуитивно понятен и похож на API обычного обобщенного типа `Dictionary<TKey, TValue>` (листинг 13.1). С помощью метода `Add` мы создаем новый зависимый описатель и «добавляем» экземпляр значения к экземпляру ключа. Отметим, что класс `ConditionalWeakTable` обобщенный, т. е. обеспечивает строгую типизацию. Поскольку ключ должен быть уникальным (ключи сравниваются с помощью метода `Object.ReferenceEquals`), этот класс поддерживает добавление только одного значения к каждому управляемому объекту (для имитации добавления нескольких свойств нужно добавить другой объект вроде словаря в качестве значения). Чтобы получить значение, связанное с ключом, воспользуйтесь методом `TryGetValue`, как показано в листинге 13.1.

Листинг 13.1 ❖ Пример использования класса `ConditionalWeakTable`

```
class SomeClass
{
   public int Field;
}

class SomeData
{
   public int Data;
}
```

```
public static void SimpleConditionalWeakTableUsage()
{
   // Зависимые описатели между объектами SomeClass (первичный) и SomeData (вторичный)
   ConditionalWeakTable<SomeClass, SomeData> weakTable = new
   ConditionalWeakTable<SomeClass, SomeData>();

   var obj1 = new SomeClass();
   var data1 = new SomeData();
   var obj1weakRef = new WeakReference(obj1);
   var data1weakRef = new WeakReference(data1);
   weakTable.Add(obj1, data1); // вызывает исключение, если значение с таким ключом
                               // уже добавлено
   weakTable.AddOrUpdate(obj1, data1);

   GC.Collect();
   Console.WriteLine($"{obj1weakRef.IsAlive} {data1weakRef.IsAlive}");
   // печатается True True
   if (weakTable.TryGetValue(obj1, out var value))
   {
      Console.WriteLine(value.Data);
   }
   GC.KeepAlive(obj1);
   GC.Collect();
   Console.WriteLine($"{obj1weakRef.IsAlive} {data1weakRef.IsAlive}");
   // печатается False False
}
```

Не будь в листинге 13.1 обращения к `GC.KeepAlive`, оба объекта `obj1` и `data1` могли быть собраны GC сразу после первого вызова `GC.Collect` (если бы JIT-компилятор использовал раннюю сборку корней, описанную в главе 8). С другой стороны, если бы мы вызвали `GC.KeepAlive(data1)`, чтобы оставить в живых вторичный объект (значение), а не первичный (ключ), то при первом обращении к `Console.WriteLine`, скорее всего, было бы напечатано `False True`. В этот момент ключ был бы собран GC, потому что никакой объект не удерживает ссылку на него.

> Отметим, что `ConditionalWeakTable` – на самом деле контейнер, в котором хранится коллекция зависимых описателей, являющихся неуправляемыми ресурсами (похожих на выделенные с помощью `GCHandle`). Мы создаем их неявно методом `Add` или `AddOrUpdate`, но когда они освобождаются? В текущей реализации это неявно делает финализатор внутреннего контейнера (т. е. после того, как экземпляр `ConditionalWeakTable` становится недостижимым). Однако мы можем произвести явную очистку, вызвав метод `Clear` (добавленный в .NET Core 2.0). Даже вызов метода `Remove` в текущей версии не освобождает описатели (из-за возможных проблем с многопоточностью).

Разумеется, мы можем отказаться от строгой типизации `ConditionalWeakTable`, указав в качестве параметра обобщенных типов тип `object` (листинг 13.2). Тогда можно будет добавлять любой объект к любому другому.

Листинг 13.2 ❖ Пример использования класса `ConditionalWeakTable`

```
ConditionalWeakTable<object, object> weakTable = new
ConditionalWeakTable<object, object>();
var obj1 = new SomeClass();
var data1 = new SomeData();
weakTable.Add(obj1, data1);
```

Следует также иметь в виду, что ограничение «одно значение на один управляемый объект (ключ)» налагает класс `ConditionalWeakTable`, а не сам зависимый описатель. Поэтому ничто не помешает нам добавить несколько «значений» в один и тот же объект, воспользовавшись несколькими экземплярами `ConditionalWeakTable` (листинг 13.3).

Листинг 13.3 ❖ Пример использования класса `ConditionalWeakTable`

```
var obj1 = new SomeClass();
var weakTable1 = new ConditionalWeakTable<object, object>();
var weakTable2 = new ConditionalWeakTable<object, object>();
var data1 = new SomeData();
var data2 = new SomeData();
weakTable1.Add(obj1, data1);
weakTable2.Add(obj1, data2);
```

Слабые ссылки, стоящие за зависимым описателем, ведут себя как длинные слабые ссылки (long weak reference), т. е. поддерживают связь между первичным и вторичным объектами даже в процессе финализации первичного объекта (листинг 13.4). Это позволяет нам корректно обрабатывать случаи воскрешения объекта.

Листинг 13.4 ❖ Поведение зависимых описателей в процессе финализации

```
class FinalizableClass : SomeClass
{
   ~FinalizableClass()
   {
   }
}

public static void FinalizationUsage()
{
   ConditionalWeakTable<SomeClass, SomeData> weakTable = new
      ConditionalWeakTable<SomeClass, SomeData>();
   var obj1 = new FinalizableClass();
   var data1 = new SomeData();
   var obj1weakRef = new WeakReference(obj1, trackResurrection: true);
   var data1weakRef = new WeakReference(data1, trackResurrection: true);
   weakTable.Add(obj1, data1);

   GC.Collect();
   Console.WriteLine($"{obj1weakRef.IsAlive} {data1weakRef.IsAlive}");
   // печатается True True
   GC.KeepAlive(obj1);
   GC.Collect();
   Console.WriteLine($"{obj1weakRef.IsAlive} {data1weakRef.IsAlive}");
   // печатается True True
   GC.WaitForPendingFinalizers();
   GC.Collect();
   Console.WriteLine($"{obj1weakRef.IsAlive} {data1weakRef.IsAlive}");
   // печатается False False
}
```

В WinDbg зависимые описатели рассматриваются как один из типов описателей, поэтому для их изучения мы можем использовать команду SOS `!gchandles` (листинг 13.5). Поскольку внутренний контейнер `ConditionalWeakTable` финализируемый, мы часто видим его в очередях финализации (листинг 13.6).

Листинг 13.5 ❖ Результат команды `!gchandles`, входящей в расширение SOS (для кода, похожего на приведенный в листинге 13.3)

```
> !gchandles -stat
...
Handles:
   Strong Handles:      10
   Pinned Handles:       4
   Weak Long Handles:    1
   Weak Short Handles:   1
   Dependent Handles:    2

> !gchandles -type Dependent
     Handle Type                      Object   Size              Data
Type
00000292abfe1bf0 Dependent  00000292b034d188     24  00000292b034d448
CoreCLR.DependentHandles.SomeClass
00000292abfe1bf8 Dependent  00000292b034d188     24  00000292b034d430
CoreCLR.DependentHandles.SomeClass

Statistics:
              MT   Count   TotalSize  Class Name
00007fff033166b8       2          48  CoreCLR.DependentHandles.SomeClass
Total 2 objects
```

Листинг 13.6 ❖ Результат команды `!finalizequeue`, входящей в расширение SOS (для кода, похожего на приведенный в листинге 13.3)

```
> !finalizequeue
...
Statistics for all finalizable objects (including all objects ready for
finalization):
              MT   Count   TotalSize  Class Name
...
00007fff03429678       2         112  System.Runtime.CompilerServices.
ConditionalWeakTable`2+Container[[System.Object,System.Private.CoreLib],[System.Object,
System.Private.CoreLib]]
Total 5 objects
```

Класс `ConditionalWeakTable` полезен для реализации кеширования или паттернов слабых событий (weak event). В первом случае мы можем кешировать данные, связанные с объектом, до тех пор, пока объект жив, а во втором – связать время жизни обработчика (делегата) со временем жизни целевого объекта (более полное описание паттерна слабого события см. в главе 12). В листинге 13.7 показаны фрагменты класса `WeakEventManager`, используемого в Windows Presentation Foundation. Для связывания времен жизни делегатов и их целевых объектов применяется контейнер `ConditionalWeakTable` (представленный полем `_cwt`). Таким образом, список делегатов жив, пока жив целевой объект.

Листинг 13.7 ❖ Методы класса `ListenerList` (часть класса `WeakEventManager` из WPF)

```
public void AddHandler(Delegate handler)
{
   object target = handler.Target;
   ...
   // добавить запись в главный список
   _list.Add(new Listener(target, handler));
   AddHandlerToCWT(target, handler);
}

void AddHandlerToCWT(object target, Delegate handler)
{
   // добавление обработчика в CWT - гарантия, что объект обработчика будет жив,
   // пока жив целевой объект, но не продление времени жизни целевого объекта
   object value;
   if (!_cwt.TryGetValue(target, out value))
   {
      // верно в 99 % случаев - целевой объект прослушивает только одно событие
      _cwt.Add(target, handler);
   }
   else
   {
      // оставшийся 1 % - целевой объект прослушивает несколько событий,
      // сохраняем делегаты в списке
      List<Delegate> list = value as List<Delegate>;
      if (list == null)
      {
         // создаем список лениво и помещаем в него уже существующий обработчик
         Delegate oldHandler = value as Delegate;
         list = new List<Delegate>();
         list.Add(oldHandler);

         // делаем список новым значением CWT
         _cwt.Remove(target);
         _cwt.Add(target, list);
      }
      // добавляем в список новый обработчик
      list.Add(handler);
   }
}
```

На этапе пометки зависимые описатели просматриваются специальным образом, потому что они могут создавать сложные зависимости и одного просмотра просто недостаточно. Представьте себе три зависимых описателя, хранящихся в таблице описателей в следующем порядке: объект C ссылается на объект A, B ссылается на C, A ссылается на B. Если допустить, что достижимость объекта A уже установлена (он помечен как достижимый), при первом просмотре таких описателей только объект B будет помечен как достижимый. При втором просмотре будет помечен C (поскольку теперь GC знает, что B достижим). При третьем просмотре ничего не изменится (т. к. A уже помечен), и на этом процесс анализа завершается. Теоретически такой многопроходный просмотр может привести к дополнительным накладным расходам, если есть миллионы зависимых описателей со сложными зависимостями между ними; но предполагается, что обычно их не настолько много.

Если вы хотите более глубоко исследовать этот механизм в коде CoreCLR, начните с методов `gc_heap::background_scan_dependent_handles` и `gc_heap::scan_dependent_handles`. Оба прекрасно документированы, как и методы, вызываемые из них: `GcDhReScan` и `GcDhUnpromotedHandlesExist`. В начале этапа пометки вызывается метод `GcDhInitialScan`, комментарии к которому проливают свет на реализацию зависимых описателей.

Локальная память потока

Обычные статические переменные можно рассматривать как глобальные в пределах одного домена приложения. К ним есть доступ у всех потоков приложения. Поэтому для обеспечения потокобезопасности необходима синхронизация потоков. Но существует еще один тип «почти» глобальных данных, уникальных в каждом потоке, – *локальная память потока* (thread local storage – TLS). Иными словами, этот тип ведет себя как глобальная переменная в том смысле, что каждый поток обращается к нему по одному и тому же имени или идентификатору, но при этом в каждом потоке хранится его собственное значение переменной этого типа. Это освобождает нас от забот о синхронизации, поскольку данные доступны только одному потоку.

В настоящее время в .NET есть три способа использования локальной памяти потока:

- *статические поля потока*, которые представляют собой обычные статические поля с дополнительным атрибутом `ThreadStatic`;
- вспомогательный класс `ThreadLocal<T>`, оборачивающий статическое поле потока;
- *слоты данных потока*, которые доступны с помощью методов `Thread.SetData` и `Thread.GetData`.

В документации по .NET ясно сказано, что производительность статических полей потока гораздо выше, чем у слотов данных, поэтому им следует отдавать предпочтение везде, где возможно. Мы рассмотрим детали реализации и того, и другого, чтобы понять разницу. Ко всему прочему статические поля потока строго типизированы (у них есть тип, как у любого другого поля в .NET), тогда как слоты данных всегда работают с типом `Object`, а в случае именованных слотов данных строковые идентификаторы могут приводить к проблемам, которые трудно обнаружить на этапе компиляции.

Статические поля потока

Чтобы сделать обычное статическое поле статическим полем потока, нужно лишь пометить его атрибутом `ThreadStatic`. Статическими полями потока могут быть как типы значений, так и ссылочные типы (листинг 13.8). В нашем примере один и тот же экземпляр класса `SomeClass` используется в двух разных потоках, но у его статических полей в каждом потоке свои значения. Поэтому в одном потоке печатается строка `Worker 1:1`, а в другом – `Worker 2:2`. Если бы оба поля были просто статическими, то при записи в них возникло бы состояние гонки (race condition), и в результате могла бы быть напечатана случайная комбинация чисел 1 и 2.

Листинг 13.8 ❖ Пример использования статических полей потока

```
class SomeData
{
   public int Field;
}

class SomeClass
{
   [ThreadStatic]
   private static int threadStaticValueData;
   [ThreadStatic]
   private static SomeData threadStaticReferenceData;

   public void Run(object param)
   {
      int arg = int.Parse(param.ToString());
      threadStaticValueData = arg;
      threadStaticReferenceData = new SomeData() { Field = arg };
      while (true)
      {
         Thread.Sleep(1000);
         Console.WriteLine($"Worker {threadStaticValueData}:
{threadStaticReferenceData.Field}.");
      }
   }
}

static void Main(string[] args)
{
   SomeClass runner = new SomeClass();
   Thread t1 = new Thread(new ParameterizedThreadStart(runner.Run));
   t1.Start(1);
   Thread t2 = new Thread(new ParameterizedThreadStart(runner.Run));
   t2.Start(2);
   Console.ReadLine();
}
```

У статических полей потока есть одно неожиданное неудобство – если у поля есть инициализатор, то он будет вызван только один раз в том потоке, который вызвал статический конструктор. Иными словами, статическое поле потока будет правильно инициализировано только в потоке, который первым получил к нему доступ, а во всех остальных потоках поле получит значение по умолчанию (листинг 13.9). Удивительно, но из-за такого поведения метод `SomeOtherClass.Run` напечатает в одной строке `Worker 100`, а в другой – `Worker 0`.

Листинг 13.9 ❖ Пример странной инициализации статического поля потока

```
class SomeOtherClass
{
   [ThreadStatic]
   private static int threadStaticValueData = 100;

   public void Run()
   {
```

```
      while (true)
      {
         Thread.Sleep(1000);
         Console.WriteLine($"Worker {threadStaticValueData}");
         // печатается Worker 100 или Worker 0.
      }
   }
}

static void Main(string[] args)
{
   SomeOtherClass runner = new SomeOtherClass();
   Thread t1 = new Thread(runner.Run);
   t1.Start();
   Thread t2 = new Thread(runner.Run);
   t2.Start();
}
```

Для решения этой проблемы в версии .NET Framework 4.0 появился класс `ThreadLocal<T>`, который характеризуется более предсказуемым поведением во время инициализации. Мы можем передать его конструктору фабрику значений, которая будет лениво инициализировать экземпляр такого класса при первом обращении к свойству `Value` (листинг 13.10).

Листинг 13.10 ❖ Пример использования класса `ThreadLocal<T>`

```
class SomeOtherClass
{
   private ThreadLocal<int> threadValueLocal = new ThreadLocal<int>(() => 100,
      trackAllValues: true);

   public void Run()
   {
      while (true)
      {
         Thread.Sleep(1000);
         Console.WriteLine($"Worker {threadStaticValueData}:{threadValueLocal.Value}.");
         Console.WriteLine(threadValueLocal.Values.Count);
      }
   }
}
```

> Кроме того, `ThreadLocal<T>` может отслеживать все инициализированные значения, для этого конструктору нужно передать `true` в аргументе `trackAllValues`. Впоследствии мы можем использовать свойство `Values` для перебора всех текущих значений. Но будьте осторожны, потому что это прямая дорога к проблемам: мы можем начать передавать ссылочные экземпляры между потоками, хотя предполагалось, что они будут локальными для потока.

Класс `ThreadLocal<T>` – это просто обертка вокруг статического поля потока. Правда, наличие дополнительных внутренних структур может несколько снизить производительность, но если это вас не слишком тревожит, то классу `ThreadLocal<T>` следует отдать предпочтение перед простыми статическими полями потока.

Листинг 13.11 ❖ Результаты сравнения времени доступа к локальным данным потока примитивного и ссылочного типов с помощью локальных полей потока и класса `ThreadLocal<T>`. Получено с помощью библиотеки BenchmarkDotNet

```
                Метод |   Среднее |  Выделено |
--------------------- |----------:|----------:|
PrimitiveThreadStatic  |  4.072 нс |       0 Б |
ReferenceThreadStatic  |  5.076 нс |       0 Б |
PrimitiveThreadLocal   |  7.866 нс |       0 Б |
ReferenceThreadLocal   | 11.762 нс |       0 Б |
```

Если вам все-таки необходима производительность простых локальных полей потока, но при этом хочется избежать проблем с инициализацией, то можно использовать нехитрый трюк, обернув статическое поле потока ленивой инициализацией с помощью обычного статического поля (листинг 13.12).

Листинг 13.12 ❖ Решение проблемы инициализации статического поля потока

```
[ThreadStatic]
private static int? threadStaticData;
public static int ThreadStaticData
{
   get
   {
      if (threadStaticData == null)
         threadStaticData = 44;
      return threadStaticData.Value;
   }
}
```

Слоты данных потока

Использовать слоты данных потока легко и просто. Существует два вида слотов данных (листинг 13.13):

- *именованный слот данных* потока, который доступен по строковому имени с помощью метода `Thread.GetNamedDataSlot`. Экземпляр `LocalDataStoreSlot`, который возвращает этот метод, можно сохранить и использовать многократно, а можно и дальше получать его таким же способом, когда возникает необходимость;
- *безымянный слот данных* потока, доступный только с помощью экземпляра `LocalDataStoreSlot`, который возвращает метод `Thread.AllocateDataSlot`.

Листинг 13.13 ❖ Пример использования слотов данных потока

```
public void UseDataSlots()
{
   // Именованные слоты данных
   Thread.SetData(Thread.GetNamedDataSlot("SlotName"), new SomeData());
   object data = Thread.GetData(Thread.GetNamedDataSlot("SlotName"));
   Console.WriteLine(data);
   Thread.FreeNamedDataSlot("SlotName");
```

```
    // Безымянные слоты данных
    LocalDataStoreSlot slot = Thread.AllocateDataSlot();
    Thread.SetData(slot, new SomeData());
    object data = Thread.GetData(slot);
    Console.WriteLine(data);
}
```

Как уже отмечалось, мы теряем строгую типизацию, потому что методы `Thread.SetData` и `Thread.GetData`, составляющие API слотов данных потока, ожидают и возвращают значение типа `Object`. Но взамен слоты данных дают нам гибкость: мы можем динамически определять статические переменные потока с заданным именем. Впрочем, на практике такая гибкость редко бывает необходима, так что статические поля потока и класс `ThreadLocal<T>` предпочтительнее.

Простой тест производительности для доступа к значению примитивного типа (целому) и к целому полю ссылочного типа недвусмысленно показывает значительное превосходство простых статических полей потока (листинг 13.14). Полагаю, такие тесты объясняют, почему слоты данных не популярны – например, во всех связанных с .NET библиотеках с открытым исходным кодом (включая WPF и ASP.NET Core) встречается только один пример их использования.

Листинг 13.14 ❖ Результаты сравнения времени доступа к локальным данным потока примитивного и ссылочного типов с помощью локальных полей потока и слотов данных. Получено с помощью библиотеки BenchmarkDotNet

```
                  Метод |   Среднее |  Выделено |
----------------------- |----------:|----------:|
PrimitiveThreadStatic   |  3.938 нс |       0 Б |
ReferenceThreadStatic   |  5.061 нс |       0 Б |
PrimitiveThreadDataSlot | 51.843 нс |       0 Б |
ReferenceThreadDataSlot | 48.616 нс |       0 Б |
```

Короче говоря, лучше раз и навсегда забыть о слотах данных.

Внутреннее устройство локальной памяти потока

Понимать, как реализована локальная память потока, полезно, потому что иначе может возникнуть соблазн воспользоваться ей как каким-то магическим сверхбыстрым хранилищем, привязанным к потоку. Привязка к потоку наводит на мысли о стеке, а стек ведь работает быстро, правда? А значит, такая специальная локальная память потока, которая хранится где-то в тайном месте, связанном с потоком, может работать еще быстрее, разве не так? Но действительность устроена гораздо сложнее, так что знание о том, как локальная память потока работает, поможет вам в понимании плюсов и минусов этой техники.

Начнем с того, что и вправду существует специальная область памяти, предназначенная для нужд потока. Она называется *локальной памятью потока* (TLS) в Windows и *потоковыми данными* (thread-specific data) в Linux. Но эта область довольно маленькая, ее размер обычно не превышает размера одной страницы. Она организована в виде набора *слотов* размером с указатель. Например, в Windows

гарантируется, что каждому процессу доступно только 64 слота, а максимальное количество слотов не превышает 1088. Это довольно тесные рамки, ведь 64 слота – это всего 512 байт в 64-разрядном процессе!

Поэтому было бы безрассудно говорить, что в TLS хранятся данные. В слотах TLS на самом деле хранятся адреса областей памяти. Это стандартная техника, применяемая в компиляторах, включая компиляторы C и C++.

Локальная память потока попросту слишком мала, чтобы хранить там сами данные. Но все равно она дает следующие преимущества в части производительности:

- страница памяти, содержащая TLS, почти наверняка находится в физической памяти, если мы используем ее регулярно;
- доступ к этой странице не нужно синхронизировать, потому что она видна только одному потоку.

Код работы с локальной памятью потока в CLR написан на C++. Существует глобальная статическая переменная потока типа `ThreadLocalInfo` (листинг 13.15). Один слот TLS компилятор C++ занимает под хранение адреса такой структуры (и каждый системный поток хранит адрес собственного экземпляра `ThreadLocalInfo`).

Листинг 13.15 ❖ Определение локальной памяти потока в CoreCLR

```
#ifndef __llvm__
EXTERN_C __declspec(thread) ThreadLocalInfo gCurrentThreadInfo;
#else // !__llvm__
EXTERN_C __thread ThreadLocalInfo gCurrentThreadInfo;
#endif // !__llvm__
```

В структуре `ThreadLocalInfo` хранятся адреса трех элементов данных CLR:

- экземпляра неуправляемого класса `Thread`, представляющего работающий в данный момент управляемый поток, – это важнейший элемент, который сплошь и рядом используется в среде выполнения (например, в методе `GetThread`);
- экземпляра `AppDomain`, в котором выполняется код текущего потока, – он хранится ради повышения эффективности, потому что этот указатель можно было бы получить также из экземпляра `Thread`;
- экземпляра структуры `ClrTlsInfo` – это массив адресов многочисленных внутренних структур CLR, связанных с потоками (в основном для диагностики и профилирования).

Таким образом, когда мы пользуемся любыми механизмами .NET, относящимися к локальной памяти потока, в TLS хранится лишь указатель на структуру `ThreadLocalInfo`. А все остальное находится в частной куче CLR и в куче GC – так же, как обычные статические поля (рис. 13.1). В экземпляре класса `Thread` данные, относящиеся к локальной памяти потока, разнесены по двум дополнительным классам:

- `ThreadLocalBlock` создается для каждого домена приложения (поэтому в приложениях для .NET Core будет только один экземпляр). Он управляет таблицей `ThreadStaticHandleTable`, где хранятся сильные ссылки на управляемые массивы, содержащие ссылки на статические поля потока:

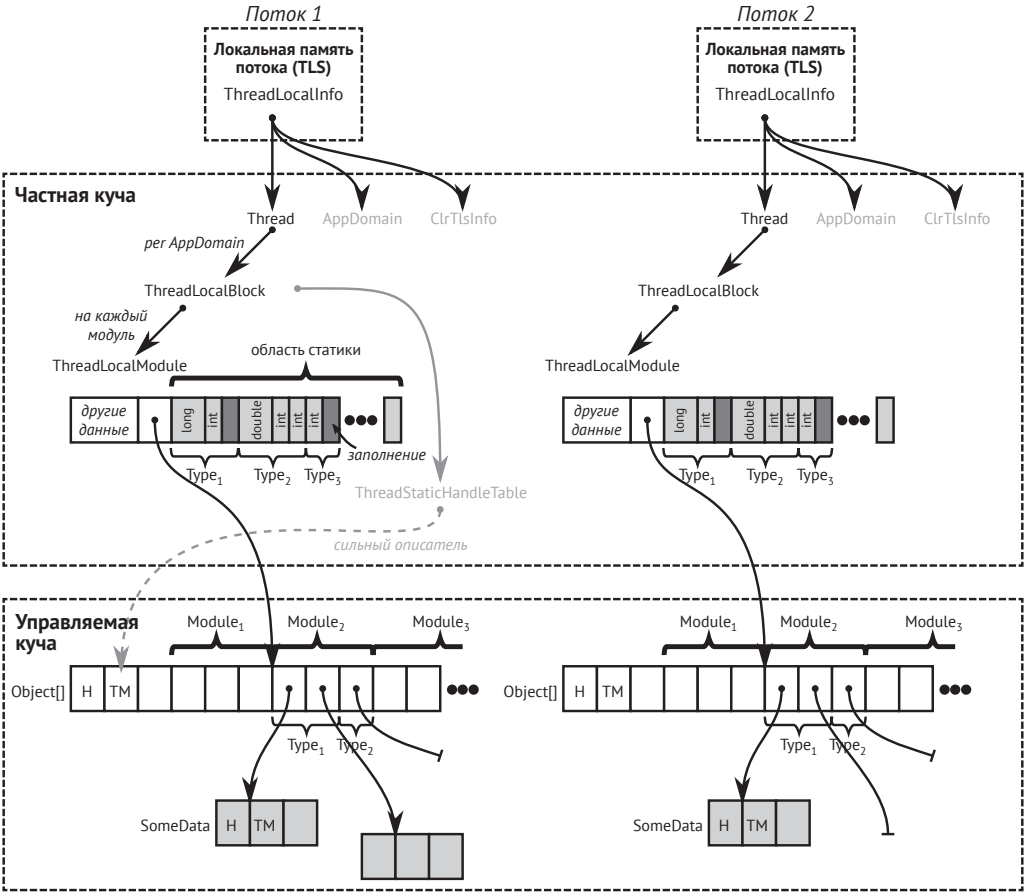


Рис. 13.1 ❖ Внутреннее устройство локальной памяти потока в .NET. Серым цветом отмечены места, где на самом деле хранятся локальные данные потока

- `ThreadLocalModule` создается для каждого модуля в `AppDomain` и содержит два очень важных блока данных:
  - неуправляемая статическая область (statics blob), где хранятся все статические неуправляемые[1] данные потока. Чтобы повысить эффективность доступа к памяти, данные в этих областях хранятся выровненными на границы в памяти;
  - смещение от начала управляемого массива, начиная с которого расположены статические ссылки этого модуля, эти ссылки дополнительно сгруппированы по типам.

Иными словами, статические данные потока хранятся следующим образом:

- память для полей ссылочного типа выделяется в куче, как обычно, а ссылки на них хранятся в специальном массиве `Object[]`, который не собирается GC благодаря сильным описателям в таблице `ThreadStaticHandleTable`. Особо отметим, что:

[1] Имеются в виду примитивные типы или типы значений, не содержащие ссылок.

  - может существовать несколько выделенных в куче объектов одного и того же типа (если эти поля инициализированы, а не оставлены равными null) по одному для каждого работающего управляемого потока;
  - может существовать несколько находящихся в куче массивов Object[] для хранения вышеупомянутых ссылок – по одному для каждого домена приложения и каждого работающего управляемого потока;
- значения полей неуправляемых типов хранятся в статических областях в неуправляемой памяти. Таких областей также может быть несколько: по одной для каждого объекта Thread, для каждого домена приложения и для каждого модуля в нем;
- структуры хранятся в управляемой куче в упакованном виде и обрабатываются так же, как ссылочные типы.

Поскольку количество типов известно на этапе компиляции, размер массивов Object[] и статических областей вычисляется заранее и является постоянным (мы знаем, сколько есть управляемых и неуправляемых статических полей потока).

> Внимательный читатель, вероятно, заметил, что создание потока в .NET может повлечь за собой много операций выделения памяти из-за статических полей потока. Может быть создано несколько новых массивов Object[] для каждого домена приложения (скорее всего, в SOH, поскольку количество управляемых статических полей потока в одном домене приложения невелико) и еще больше объектов ThreadLocalModule в частной куче CLR (содержащие статические области для каждого модуля).

На рис. 13.1 показано, как выглядит один модуль, хотя объектов ThreadLocalModule может быть больше, для краткости они не представлены. В этом модуле определено несколько типов. Мы сосредоточимся на типе Type1, который мог бы выглядеть, как показано в листинге 13.16. Он содержит два примитивных статических поля потока (типа long и int), значения которых хранятся в статической области внутри ThreadLocalModule. Также он содержит два статических поля потока ссылочного типа SomeData. Как и обычные статические поля, эти объекты создаются в куче, а ссылки на них хранятся в обычном массиве Object[]. На рис. 13.1 в потоке Thread 1 уже инициализированы оба поля типа Type1, а в потоке Thread 2 (для иллюстрации) – только первое поле.

Листинг 13.16 ❖ Пример простого типа, показанного на рис. 13.1

```
class Type1
{
   [ThreadStatic] private static int static1;
   [ThreadStatic] private static long static2;
   [ThreadStatic] private static SomeData static3;
   [ThreadStatic] private static SomeData static4;
   ...
}
```

На первый взгляд может показаться неудобным, что объекты, о которых мы думаем как о «статических, принадлежащих одному потоку», на самом деле расположены рядом друг с другом где-то в управляемой куче. Но примите во внимание, что если не случится чего-то ужасного, они невидимы друг для друга с точки зрения управляемых потоков (т. е. все-таки потокобезопасны). С другой стороны,

мы можем случайно получить ложное разделение (false sharing) (см. главу 2) таких объектов, поскольку они могут оказаться в пределах одной строки кеша.

Итак, призываю вас не забывать о рис. 13.1, если вам захочется думать о TLS как о «волшебно быстрой памяти». На самом деле TLS – всего лишь деталь реализации привязки соответствующих структур данных к потокам. В общем случае она ничего не ускоряет.

JIT-компилятор вычисляет смещения статических полей потока относительно начала статической области для неуправляемых типов и относительно начала массива ссылок для ссылочных типов. Эти смещения хранятся в участках памяти, связанных с таблицей методов, так что JIT-компилятор может использовать их для генерации адресов доступа к данным. На самом деле для доступа к данным нужно получить соответствующий объект `ThreadLocalModule` текущего потока. Доступ к статическим данным потока становится причиной дополнительных и вполне ощутимых накладных расходов (см. комментарии к листингам 13.17 и 13.18).

Листинг 13.17 ❖ Присваивание статической неуправляемой переменной потока (такой как `threadStaticValueData` в листинге 13.8)

```
// Предполагается, что регистр esi содержит значение, которое надо сохранить.
// Передаем информацию о модуле и индексе класса (типа) в регистрах rcx и edx
mov    rcx,7FFD3E295690h
mov    edx,2
// Доступ к ThreadLocalModule (по хранящемуся в TLS указателю)
// В результате rax содержит адрес ThreadLocalModule
call CoreCLR!JIT_GetSharedNonGCThreadStaticBase
mov    rdi,rax
// Сохранить значение:
// 1Ch - заранее вычисленное смещение относительно начала статической области,
// в esi хранится значение, подлежащее сохранению
mov    dword ptr [rdi+1Ch],esi
```

Листинг 13.18 ❖ Присваивание статической ссылочной переменной потока (такой как `threadStaticReferenceData` в листинге 13.8)

```
// Предполагается, что регистр rbx содержит значение (ссылку), которое надо сохранить.
// Передаем информацию о модуле и индексе класса (типа) в регистрах rcx и edx
mov    rcx,7FFD3E295690h
mov    edx,2
// Доступ к ThreadLocalModule (по хранящемуся в TLS указателю)
// В результате rax содержит ссылку на элемент массива, с которого начинаются
// ссылки этого типа
call   CoreCLR!JIT_GetSharedGCThreadStaticBase
mov    rcx,rax
// Сохранить ссылку (в rbx) в заданном элементе массива (в rdx), вызвав
// барьер записи (write barrier)
mov    rdx,rbx
call   CoreCLR!JIT_WriteBarrier (00007ffd`9d6c57d0)
```

С другой стороны, если бы поля не были статическими (обычными или принадлежащими конкретному потоку), то доступ к данным был бы на несколько порядков быстрее, поскольку не потребовалось бы обращаться к среде выполнения (достаточно было бы одной или двух простых команд `mov`).

Начать изучение кода CoreCLR, относящегося к локальной памяти потока, лучше всего с методов `JIT_GetSharedNonGCThreadStaticBase` и `JIT_GetSharedGCThreadStaticBase`. Методы, сгенерированные JIT-компилятором, часто содержат макрос `INLINE_GETTHREAD`, который получает `gCurrentThreadInfo` (экземпляр статической переменной потока типа `ThreadLocalInfo`) из TLS-памяти, например в Windows используется `OFFSET__TEB__ThreadLocalStoragePointer` для поиска адреса TLS в текущем *блоке окружения потока* (Thread Environment Block). Как было сказано выше, `ThreadLocalInfo` содержит указатель на неуправляемый объект `Thread`. Указатель на `AppDomain` и массив указателей `m_EETlsData` для нас несущественны. Типы `ThreadLocalModule`, `ThreadLocalBlock` и `ThreadStatics`, определенные в файле `.\src\vm\threadstatics.h`, содержат основную логику работы с локальной памятью потока.

Что касается вычисления смещений статических полей (обычных и статических полей потока), то метод `Module::BuildStaticsOffsets` заполняет вспомогательный массив всех смещений, относящихся к модулю (см. поля `m_pRegularStaticOffsets` и `m_pThreadStaticOffsets`). Затем этот массив используется в методах `MethodTableBulder::PlaceRegularStaticFields` и `MethodTableBulder::PlaceThreadStaticFields`.

Возникает вопрос: как обстоит дело с обобщенными типами, содержащими статические поля потока? Мы говорили, что на этапе компиляции количество статических полей потока известно, но очевидно, что для обобщенных типов это не так – компилятор не знает, сколько в итоге будет обобщенных типов (а для каждого может понадобиться свой набор статических полей потока). Решение такое же, как для обычных статических полей обобщенных типов: в `ThreadLocalModule` хранится дополнительный динамический массив указателей на небольшие структуры данных, похожие на сам класс `ThreadLocalModule` (рис. 13.2 и соответствующий ему листинг 13.19). Каждая такая структура соответствует одному конкретному обобщенному типу и содержит такие же данные, как в `ThreadLocalModule`: смещение, с которого начинаются его поля ссылочного типа в таблице `ThreadStaticHandleTable` (которая может динамически расти) и статической области.

Листинг 13.19 ❖ Простой обобщенный тип `Some<T>`, показанный на рис. 13.2

```
class Some<T>
{
   [ThreadStatic]
   private static T static1;
   [ThreadStatic]
   private static SomeData static2;
   [ThreadStatic]
   private static SomeData static3;
   ...
}
```

С точки зрения сборки мусора, статические данные потока ссылочного типа – это обычные объекты с корнями в вышеупомянутых массивах `Object[]`, которые не собираются GC благодаря сильным описателям, хранящимся в `ThreadLocalBlock`. Таким образом, они живы, пока живы соответствующие поток `Thread` и домен приложения `AppDomain`.

Слоты данных еще медленнее, потому что лежащий в их основе общий механизм построен над внутренней статической памятью потока (листинг 13.20). Поэтому он, очевидно, медленнее, чем простое статическое поле потока. Добавляются расходы на обслуживание внутренних похожих на словарь структур (для

хранения списков слотов «ключ/значение») и на синхронизацию потоков. Для неуправляемых примитивных типов появляются также издержки, связанные с упаковкой и распаковкой. Можете самостоятельно изучить дополнительные типы, показанные в листинге 13.20, чтобы понять, насколько больше работы приходится делать по сравнению с простым доступом к статической переменной потока.

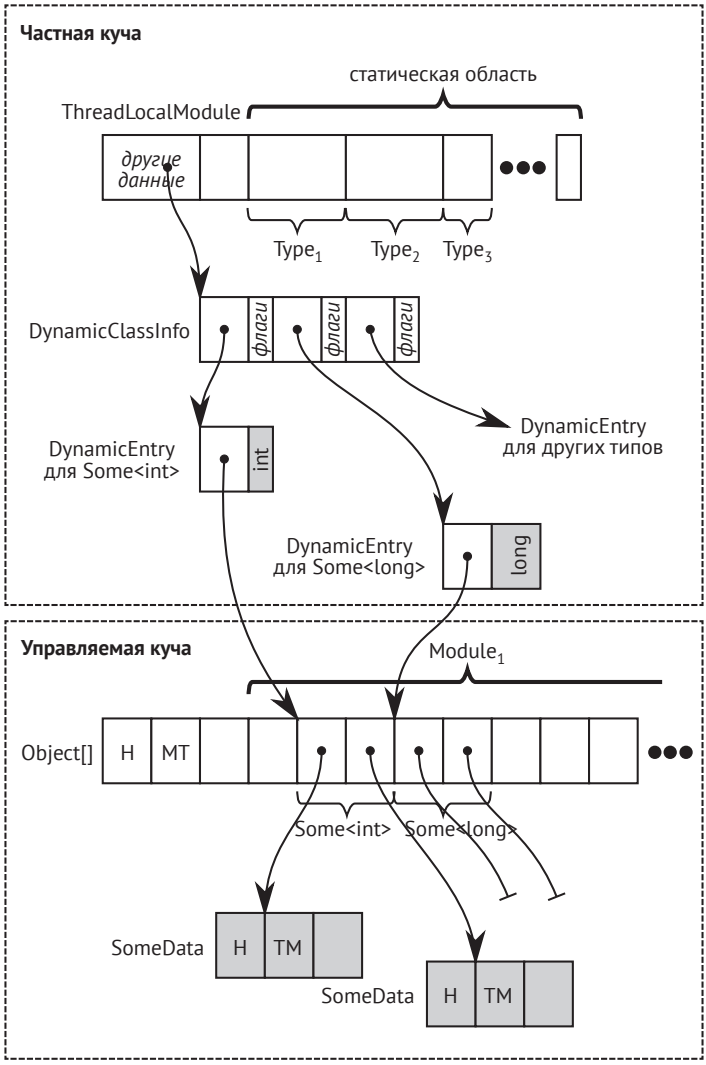


Рис. 13.2 ❖ Внутреннее устройство локальной памяти потока для обобщенных типов

Листинг 13.20 ❖ Часть определения класса `Thread`, относящаяся к локальной памяти потока

```
public sealed class Thread : CriticalFinalizerObject, _Thread
{
```

```
/*=====================================================================
** Thread-local data store
=====================================================================*/
[ThreadStatic]
static private LocalDataStoreHolder s_LocalDataStore; // stores LocalDataStore
sealed internal class LocalDataStore
{
   private LocalDataStoreElement[] m_DataTable;
   private LocalDataStoreMgr m_Manager;
```

Если вы мысленно добавите к рис. 13.1 все управляемые структуры данных, используемые для работы со слотами данных потока, то сможете понять, почему слоты данных настолько медленнее.

Сценарии использования

Хотя приведенное выше описание локальной памяти потока явно показывает наличие дополнительных накладных расходов, с точки зрения производительности у нее есть важное преимущество – можно избавиться от синхронизации потоков. Ну а привязка к потоку – еще одно важное функциональное отличие TLS от других данных.

Локальная память потока может быть полезна в следующих ситуациях.

- Требуется хранить и управлять данными, привязанными к потоку, – например, иногда неуправляемые ресурсы нужно захватывать и освобождать в одном и том же потоке.
- Есть возможность извлечь выгоду из привязки к потоку, например:
  - протоколирование или диагностика: каждый поток может, не нуждаясь в синхронизации, манипулировать какими-то локальными данными в целях диагностики, не мешая остальным потокам (примером может служить пространство имен `System.Diagnostics.Tracing`);
  - кеширование: иногда имеет смысл организовать локальный кеш потока, хотя надо отчетливо понимать, что экземпляров кеша будет столько, сколько работает управляемых потоков. Класс `StringBuilderCache`, показанный в главе 4, может служить идеальным примером такого подхода: в каждом потоке есть кешированный экземпляр небольшого объекта `StringBuilder`, чтобы к нему можно было эффективно обращаться, не неся расходов на синхронизацию доступа к какому-то глобальному пулу. Еще один пример – класс `TlsOverPerCoreLockedStacksArrayPool<T>` из пространства имен `System.Buffers`, реализация `ArrayPool` с помощью многоуровневой схемы кеширования, когда для каждого размера массива есть небольшой локальный кеш потока, а также кеш, разделяемый всеми потоками (разбитый на несколько секций, каждая со своей блокировкой, чтобы минимизировать конкуренцию за доступ со стороны нескольких процессорных ядер). Кстати говоря, именно объект этого класса возвращается, когда мы запрашиваем экземпляр `ArrayPool<T>.Shared`[1].

[1] Это относится к версии .NET Core 2.1, а в .NET Core 2.0 этот класс использовался только для пула массивов `char` и `byte`.

Статические поля потока, очевидно, не годятся для асинхронного программирования, потому что не гарантируется, что выполнение асинхронного метода будет продолжено в том же потоке, в каком начато. Поэтому когда выполнение асинхронного метода продолжится, мы можем потерять локальные данные потока. Для решения этой проблемы, помимо типа `ThreadLocal<T>`, есть тип `AsyncLocal<T>`, который сохраняет данные на всем протяжении выполнения асинхронного метода. Но с точки зрения управления памятью, этот класс не так интересен – его экземпляры (вместе с соответствующим значением) находятся в словаре, который хранится в контексте выполнения (объекте класса `ExecutionContext`).

Управляемые указатели

До сих пор мы старались обходить вопрос об управляемых указателях стороной (хотя внимательный читатель, возможно, вспомнит одно-два упоминания). Большую часть времени типичный .NET-разработчик пользуется ссылками на объекты, и этого достаточно, потому что именно так устроен управляемый мир – объекты указывают друг на друга с помощью ссылок. В главе 4 объяснялось, что ссылка на объект – это на самом деле указатель (адрес), обеспечивающий безопасность типов, который всегда указывает на ссылочное поле объекта MethodTable (часто говорят, что эта ссылка указывает на начало объекта). А значит, их использование может быть весьма эффективным. Таким образом, имея ссылку на объект, мы знаем адрес всего объекта. Например, GC может быстро получить доступ к заголовку объекта с помощью постоянного смещения. Адреса полей также легко вычислить, пользуясь информацией, хранящейся в таблице методов.

Но в CLR есть также *управляемый указатель*. Его можно было бы определить как более общий тип указателя, который не обязательно должен указывать на начало объекта. В ECMA-335 говорится, что управляемый указатель может указывать на:

- локальную переменную, будь то ссылка на объект, находящийся в куче, или просто переменная в стеке;
- параметр – как и выше;
- поле составного типа, то есть поле другого типа значений или ссылочного;
- элемент массива.

Несмотря на такую гибкость, управляемые указатели – это все-таки типы. Существует тип управляемого указателя на объекты `System.Int32` независимо от их местоположения, который в CIL обозначается `System.Int32&`. Или тип `SomeNamespace.SomeClass&`, указывающий на объекты типа `SomeNamespace.SomeClass`. Благодаря строгой типизации они безопаснее, чем неуправляемые указатели, которые могут указывать вообще на все, что угодно. По этой причине для управляемых указателей не определена *арифметика указателей*, как для простых указателей, так как сложение и вычитание представляемых ими адресов лишено смысла.

Однако за гибкость приходится платить. И ценой являются ограничения на то, где разрешено использовать управляемые указатели. Согласно ECMA-335, тип управляемого указателя допустим только:

- для локальных переменных;
- в сигнатурах параметров.

Прямо сказано, что «их не разрешается использовать в сигнатурах полей и в качестве типа элементов массива, запрещается также упаковка типа управляемого

указателя. Использование типа управляемого указателя в качестве типа возвращаемого значения метода не может быть проверено».

Из-за этих ограничений управляемые указатели не используются в языке C# напрямую. Но они уже давно присутствуют в виде хорошо известных ref-параметров. Передача параметра по ссылке – не что иное, как замаскированное использование управляемого указателя. Поэтому управляемые указатели часто называют *типами, передаваемыми по ссылке* (или просто *byref*). Примеры передачи по ссылке мы уже видели в листингах 4.33 и 4.34 из главы 4.

Начиная с версии C# 7.0, сфера применения управляемых указателей расширилась: теперь есть ссылочные локальные переменные (ref locals) и возврат значений по ссылке (ref returns). Следовательно, последнее предложение в приведенной выше цитате из стандарта ECMA, касающееся использования типа управляемого указателя в качестве типа возвращаемого значения, становится не таким строгим.

Ссылочные локальные переменные

Ссылочную локальную переменную (ref local) можно рассматривать как локальную переменную для хранения управляемого указателя. Таким образом, это удобный способ создания вспомогательных переменных, которые впоследствии можно использовать для прямого доступа к полю, элементу массива или другой локальной переменной (листинг 13.21). Заметим, что обе части оператора присваивания должны содержать ключевое слово `ref`, означающее операцию над управляемыми указателями.

Листинг 13.21 ❖ Простое применение ссылочных локальных переменных

```
public static void UsingRefLocal(SomeClass data)
{
   ref int refLocal = ref data.Field;
   refLocal = 2;
}
```

Тривиальный пример в листинге 13.21 носит чисто иллюстративный характер – мы получаем прямой доступ к полю типа `int`, так что выигрыш в производительности пренебрежимо мал. Чаще ссылочная переменная используется, чтобы получить прямой указатель на какой-нибудь тяжеловесный объект и тем самым избежать копирования (листинг 13.22), а затем передать его куда-то по ссылке или использовать локально. В ссылочных локальных переменных также часто хранят результат, возвращаемый методом по ссылке (как мы скоро увидим).

Листинг 13.22 ❖ Возможное использование ссылочных локальных переменных (пример взят из MSDN)

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
// впоследствии с помощью reflocal можно использовать veryLargeStruct без копирования
```

Ссылочной локальной переменной можно присвоить нулевую ссылку (листинг 13.23). На первый взгляд, выглядит странно, но отнюдь не лишено смысла. Ссылочную локальную переменную можно рассматривать как переменную, в которой хранится адрес ссылки, но это не значит, что сама ссылка должна на что-то указывать.

Листинг 13.23 ❖ Присваивание нулевой ссылки ссылочной локальной переменной

```
SomeClass local = null;
ref SomeClass localRef = ref local;
```

Возвращаемые ссылочные значения

Возвращаемое ссылочное значение (ref return) позволяет вернуть управляемый указатель из метода. Очевидно, без некоторых ограничений не обойтись. Процитируем MSDN: «Время жизни возвращаемого значения должно включать в себя время выполнения метода. Другими словами, нельзя возвращать ссылку на локальную переменную вызываемого метода. Это может быть экземпляр статического поля или класса, а также переданный методу аргумент. При попытке возвратить локальную переменную возникает ошибка компилятора CS8168 «Cannot return local 'obj' by reference because it is not a ref local»[1].

Пример описанного ограничения приведен в листинге 13.24. Очевидно, что мы не можем вернуть управляемый указатель на переменную `localInt`, находящуюся в стеке, потому что он станет недействительным, как только метод `ReturnByRefValueTypeInterior` завершится.

Листинг 13.24 ❖ Пример некорректной попытки вернуть локальную переменную по ссылке

```
public static ref int ReturnByRefValueTypeInterior(int index)
{
   int localInt = 7;
   return ref localInt; // Ошибка компиляции: Cannot return local 'localInt' by reference
                        // because it is not a ref local
}
```

Однако никто не запрещает вернуть по ссылке элемент параметра метода, потому что, с точки зрения метода, его аргумент живет дольше, чем сам метод (листинг 13.25). В нашем примере метод `GetArrayElementByRef` возвращает управляемый указатель на запрошенный элемент массива.

Листинг 13.25 ❖ Пример возврата по ссылке

```
public static ref int GetArrayElementByRef(int[] array, int index)
{
   return ref array[index];
}
```

Использовать метод, возвращающий значение по ссылке, можно двумя способами (листинг 13.26):

- воспользоваться возвращенным управляемым указателем – это самый распространенный способ работы с методами, возвращающими значение по ссылке. В таком случае мы должны вызвать метод с ключевым словом `ref` и сохранить результат в локальной ссылочной переменной. В первом вызове метода `GetArrayElementByRef` в листинге 13.26 демонстрируется именно такой подход. Поскольку мы возвращаем управляемый указатель на элемент

[1] Невозможно вернуть по ссылке локальный `'obj'`, так как это не локальная переменная `ref`.

массива, то можем модифицировать его содержимое напрямую (на консоль будет выведена строка 423);

- воспользовавшись значением, на которое указывает возвращенный управляемый указатель, и опустив оба ключевых слова ref, мы вернемся к обычному вызову метода (см. второй вызов метода GetArrayElementByRef в листинге 13.26). В этом случае метод возвращает копию значения, поэтому модификация результата не приведет к модификации исходного значения (на консоль по-прежнему выводится 423, несмотря на попытку сделать первый элемент массива равным 5).

Листинг 13.26 ❖ Использование метода, возвращающего значение по ссылке

```
int[] array = {1, 2, 3};
ref int arrElementRef = ref PassingByref.GetArrayElementByRef(array, 0);
arrElementRef = 4;
Console.WriteLine(string.Join("", array)); // выводится 423

int arrElementVal = PassingByref.GetArrayElementByRef(array, 0);
arrElementVal = 5;
Console.WriteLine(string.Join("", array)); // все равно выводится 423
```

Как и в случае ссылочных локальных переменных, можно вернуть ссылку на нулевую ссылку (листинг 13.27). В этом коде, основанном на демонстрационных примерах из документации по .NET, представлен очень простой тип коллекции книг. Его метод GetBookByTitle возвращает по ссылке книгу с заданным названием, если она существует. А если не существует, то возвращается ссылка на предопределенный член класса nobook, равный null. Далее мы можем проверить, указывает ссылка, возвращенная методом GetBookByTitle, на что-нибудь или нет.

Листинг 13.27 ❖ Возврат нулевой ссылки по ссылке

```
public class BookCollection
{
   private Book[] books =
   {
      new Book { Title = "Call of the Wild, The", Author = "Jack London" },
      new Book { Title = "Tale of Two Cities, A", Author = "Charles Dickens" }
   };
   private Book nobook = null;
   public ref Book GetBookByTitle(string title)
   {
      // Book nobook = null; // не будет работать
      for (int ctr = 0; ctr < books.Length; ctr++)
      {
         if (title == books[ctr].Title)
            return ref books[ctr];
      }
      return ref nobook;
   }
}

static void Main(string[] args)
{
```

```
   var collection = new BookCollection();
   ref var book = ref collection.GetBookByTitle("<Not exists>");
   if (book != null)
   {
      Console.WriteLine(book.Author);
   }
}
```

Отметим, что мы не можем просто использовать ссылку на локальную переменную nobook (как в закомментированной строке в методе GetBookByTitle), потому что время ее жизни не охватывает время выполнения метода.

Постоянные ссылочные переменные и in-параметры

Ссылочные типы – очень мощная вещь, потому что мы можем изменить целевой объект. Так, в версии C# 7.2 были введены *постоянные ссылочные переменные* (readonly ref), чтобы можно было управлять возможностью изменения объекта, на который указывает ссылочная переменная. Отметим в этом контексте тонкое различие между управляемым указателем на тип значений и ссылочным типом:

- для целевого объекта типа значений гарантируется, что значение не будет изменено. Поскольку в роли значения здесь выступает целый объект (область памяти), это означает, что не будет изменено ни одно поле;
- для целевого объекта ссылочного типа гарантируется, что не будет изменено значение ссылки. Поскольку значением здесь является сама ссылка (указывающая на какой-то объект), то гарантируется, что она не станет указывать на другой объект. Однако свойства объекта, на который указывает ссылка, модифицировать можно.

Перепишем пример в листинге 13.27, так чтобы он возвращал постоянное ссылочное значение (листинг 13.28). Код один и тот же, единственная разница – в сигнатуре метода GetBookByTitle.

Листинг 13.28 ❖ Пример, взятый из документации по .NET

```
public class BookCollection
{
   private Book[] books =
   {
      new Book { Title = "Call of the Wild, The", Author = "Jack London" },
      new Book { Title = "Tale of Two Cities, A", Author = "Charles Dickens" }
   };
   private Book nobook = null;
   public ref readonly Book GetBookByTitle(string title)
   {
      // Book nobook = null; // не будет работать
      for (int ctr = 0; ctr < books.Length; ctr++)
      {
         if (title == books[ctr].Title)
            return ref books[ctr];
      }
      return ref nobook;
   }
```

```
}

static void Main(string[] args)
{
   var collection = new BookCollection();
   ref readonly var book = ref collection.GetBookByTitle("<Not exists>");
   if (book != null)
   {
      Console.WriteLine(book.Author);
   }
}
```

Наш класс `BookCollection` иллюстрирует различие между постоянной ссылкой для типа значений и ссылочного типа. Если `Book` – класс, то гарантируется, что мы не сможем изменить значение ссылки, например перенаправить ее на новый объект, как в закомментированной строке в листинге 13.29. Однако ничто не мешает модифицировать поля объекта, на который указывает ссылка (например, изменить автора в том же листинге 13.29).

Листинг 13.29 ❖ Использование класса из листинга 13.28 в случае, когда `Book` – класс

```
static void Main(string[] args)
{
   var collection = new BookCollection();
   ref readonly var book = ref collection.GetBookByTitle("Call of the Wild, The");
   // book = new Book(); // Недопустимо. Было бы допустимо без readonly
   book.Author = "Konrad Kokosa";
}
```

Но если `Book` – структура, то мы гарантированно не сможем изменить ее значение, в частности изменить автора в листинге 13.30 (и по той же причине нельзя присвоить ссылке новое значение, как в предыдущей строке листинга).

Листинг 13.30 ❖ Использование класса из листинга 13.28 в случае, когда `Book` – структура

```
static void Main(string[] args)
{
   var collection = new BookCollection();
   ref readonly var book = ref collection.GetBookByTitle("Call of the Wild, The");
   // book = new Book(); // Недопустимо. Было бы допустимо без readonly
   // book.Author = "Konrad Kokosa"; // Недопустимо. Было бы допустимо без readonly
}
```

Эти на первый взгляд трудные нюансы легко запомнить, если не забывать, что является защищенным значением: весь объект (для типа значений) или ссылка (для ссылочного типа). В этом контексте осталось упомянуть еще один важный аспект. Предположим, что в структуре `Book` есть метод, модифицирующий ее поле (листинг 13.31). Что будет, если вызвать его для возвращенного постоянного ссылочного значения? Даже в таком случае гарантируется, что исходное значение не изменится (листинг 13.32). Это реализовано посредством *защитного копирования* – прежде чем вызывать метод `ModifyAuthor`, создается копия возвращенного типа значений (в нашем случае – структуры `Book`), и метод вызывается для этой копии. Компилятор не анализирует, изменяет ли вызванный метод состояние, по-

скольку это было бы очень трудно сделать (когда в методе много различных условий, часть из которых может зависеть от внешних данных). Это значит, что любой метод, вызываемый для такой структуры, ведет себя подобным образом.

Таким образом, метод `ModifyAuthor` выполняется, но только для временного объекта, который скоро станет неиспользуемым. Никакие изменения, произведенные в защитной копии, не будут видны в исходном значении.

Листинг 13.31 ❖ Простой метод типа значений, модифицирующий его состояние

```
public struct Book
{
   ...
   public void ModifyAuthor()
   {
      this.Author = "XXX";
   }
}
```

Листинг 13.32 ❖ Использование класса из листинга 13.28 в случае, когда Book – структура

```
static void Main(string[] args)
{
   var collection = new BookCollection();
   ref readonly var book = ref collection.GetBookByTitle("Call of the Wild, The");
   book.ModifyAuthor();
   Console.WriteLine(collection.GetBookByTitle("Call of the Wild, The").Author);
   // печатается Jack London
}
```

Такое защитное копирование может быть неожиданным и дорогим. Так, можно предположить, что в случае успешного завершения метода `ModifyAuthor` поле будет модифицировано. Кроме того, создание защитной копии структуры, очевидно, привносит накладные расходы.

> Отметим, что если `Book` – класс, то сюрпризов не возникает, т. е. `ModifyAuthor` изменит состояние объекта, даже если использовалась постоянная ссылка. Напомним, что ключевое слово `readonly` запрещает лишь изменение самой ссылки, но не объекта, на который она указывает.

Постоянные ссылки можно использовать не только в контексте коллекций. В MSDN есть хороший пример, когда постоянная ссылка применяется для возврата статического типа значений, представляющего некоторую глобальную, часто используемую величину (листинг 13.33). Если бы `Origin` не было постоянной ссылкой, то значение можно было бы модифицировать, что, очевидно, неправильно, потому что `Origin` следует рассматривать как константу. До появления возврата по ссылке такое значение пришлось бы использовать как обычный тип значений, но тогда оно каждый раз копировалось бы.

Листинг 13.33 ❖ Пример использования постоянной ссылки для открытого статического значения (на основе примера из MSDN)

```
struct Point3D
{
```

```
   private static Point3D origin = new Point3D();
   public static ref readonly Point3D Origin => ref origin;
   ...
}
```

Постоянные ссылки можно делать и с помощью модификатора параметра *in*. В версии C# 7.2 введено небольшое, но очень важное дополнение к передаче параметров по ссылке. При передаче по ссылке параметра с ключевым словом `ref` метод может изменить аргумент, что приводит к тем же проблемам, что при возврате ссылочного значения. Поэтому был добавлен модификатор параметра `in`, который означает, что аргумент передается по ссылке, но не должен модифицироваться внутри метода (листинг 13.34).

Листинг 13.34 ❖ Пример использования in-параметра

```
public class BookCollection
{
   ...
   public void CheckBook(in Book book)
   {
      book.Title = "XXX"; // Ошибка компиляции: Cannot assign to a member of variable
                          // 'in Book' because it is a readonly variable.
   }
}
```

Здесь действуют те же правила, что для возвращаемых постоянных ссылок, только гарантируется, что не будет изменено значение параметра. То есть если у in-параметра ссылочный тип, то запрещено изменять только значение самой ссылки, а изменять объект, на который она ссылается, можно. Поэтому если бы в листинге 13.34 тип `Book` был классом, то код скомпилировался бы без ошибок, и значение `Title` было бы изменено. Запрещено только присваивание вида `book = new Book()`.

Когда методу передается in-параметр типа значений, тоже применяется описанное выше защитное копирование (листинг 13.35). Помните об этом, чтобы избежать ненужных накладных расходов на копирование, которое в принципе не имеет смысла (поскольку любые модификации будут отброшены).

Листинг 13.35 ❖ Пример использования in-параметра

```
public class BookCollection
{
   ...
   public void CheckBook(in Book book)
   {
      book.ModifyAuthor(); // вызывается для защитной копии book, поле Title
                           // исходного объекта book не изменяется
   }
}
```

Избежать защитного копирования можно, сделав структуру постоянной (`readonly`) (если логика приложения это допускает), как описано в следующем разделе. Поскольку для постоянных структур любые модификации полей запрещены, компилятор может безопасно опустить создание защитных копий и будет вызывать методы непосредственно с переданными аргументами типа значений.

Внутреннее устройство ссылочных типов

У внимательного читателя при взгляде на листинги с 13.21 по 13.33 могло бы возникнуть много вопросов. Например, как передача управляемых указателей связана с GC? Какой код генерирует JIT-компилятор? Какие преимущества в смысле производительности дают эти сложные механизмы? Если вам интересны ответы, читайте дальше. Но если хотите, можете пропустить этот раздел и сразу перейти к следующему, где описывается практическое использование ссылочных типов в C#.

Давайте выделим основные случаи применения управляемых указателей. Это поможет нам разобраться в причинах вышеупомянутых ограничений и научиться лучше их использовать. В следующих примерах мы будем работать с двумя простыми типами из листинга 13.36. Будут рассмотрены все три способа использования управляемых указателей в C# – ссылочные параметры, ссылочные локальные переменные и возвращаемые ссылочные значения.

Листинг 13.36 ❖ Два простых типа, используемых в примерах ниже

```
public class SomeClass
{
   public int Field;
}

public struct SomeStruct
{
   public int Field;
}
```

Начнем со знакомства с некоторыми деталями реализации управляемых указателей. И это приведет нас к рекомендациям по практическому применению.

Управляемый указатель на объект в стеке

Управляемый указатель может указывать на локальную переменную или параметр метода. В главе 8 мы видели, что и локальная переменная, и параметр могут находиться в стеке или в регистре процессора (где именно – решает JIT-компилятор). Как в таком случае работает управляемый указатель? На самом деле совершенно нормально, если он указывает на адрес в стеке! И это одна из причин, по которым управляемый указатель не может быть полем объекта (и не допускает упаковки). Если в результате этого он окажется в управляемой куче, то сможет прожить дольше метода, в котором когда-то находился адрес в стеке. Это было бы очень опасно (указанный адрес в стеке мог бы содержать неопределенные данные, скорее всего, принадлежащие кадру стека совсем другого метода). Ограничив использование управляемого указателя только локальными переменными и параметрами, мы ограничиваем его время жизни временем жизни самых недолговечных данных – тех, что расположены в стеке.

А как насчет локальных переменных и параметров, находящихся в регистрах? Напомним, что размещение в регистре – всего лишь деталь реализации, характеристики времени жизни должны быть такими же, как у данных, размещенных в стеке. Тут многое зависит от JIT-компилятора. Если целевой объект находится в регистре – отлично! Тогда регистр можно просто использовать как управляемый

указатель. Иначе говоря, с точки зрения JIT-компилятора, что регистр, что адрес в стеке – особой роли не играет.

Но как информация об управляемых указателях (точнее, об объектах, на которые они указывают) передается сборщику мусора? Это необходимо, потому что иначе GC не сможет установить достижимость целевого объекта, если окажется, что управляемый указатель в данный момент является единственным корнем.

Проанализируем очень простой случай передачи по ссылке, похожий на пример в листинге 4.34 из главы 4 (см. листинг 13.37). Чтобы устранить эффект встраивания (inlining) метода `Test` и упростить рассмотрение, мы добавили атрибут `NoInlining` (встраиваемый вариант мы обсудим ниже).

Листинг 13.37 ❖ Простой случай передачи по ссылке (всего объекта ссылочного типа)

```
static void Main(string[] args)
{
   SomeClass someClass = new SomeClass();
   PassingByref.Test(ref someClass);
   Console.WriteLine(someClass.Field); // печатается "11"
}

public class PassingByref
{
   [MethodImpl(MethodImplOptions.NoInlining)]
   public static void Test(ref SomeClass data)
   {
      //data = new SomeClass();
      data.Field = 11; // по крайней мере, до этой строки экземпляр класса
                       // SomeClass должен оставаться в живых (не будет убран в мусор)
   }
}
```

В настоящий момент нам интересно, как этот код представлен в CIL и на языке ассемблера после JIT-компиляции. В CIL-коде мы видим строго типизированный управляемый указатель `SomeClass&` (листинг 13.38). В методе `Main` команда `ldloca` загружает адрес локальной переменной по указанному индексу (а индекс 0 соответствует переменной `someClass`) в стек вычислений, который затем передается методу `Test`. Метод `Test` использует команду `ldind.ref`, которая разыменовывает адрес и помещает результирующую ссылку на объект в стек вычислений.

Листинг 13.38 ❖ CIL-код, соответствующий коду в листинге 13.37

```
.method private hidebysig static
   void Main (string[] args) cil managed
{
   .locals init (
      [0] class SomeClass
   )
   IL_0000: newobj instance void SomeClass::.ctor()
   IL_0005: stloc.0
   IL_0006: ldloca.s 0
   IL_0008: call void PassingByref::Test(class SomeClass&)
   IL_000d: ret
}
```

```
.method public hidebysig static
   void Test (class SomeClass& data) cil managed noinlining
{
   IL_0000: ldarg.0
   IL_0001: ldind.ref
   IL_0002: ldc.i4.s 11
   IL_0004: stfld int32 SomeClass::Field
   IL_0009: ret
}
```

Но хотя CIL-код может представлять интерес, мы уже видели примеры, когда только результат JIT-компиляции раскрывает истинную природу происходящего. Глядя на ассемблерный код обоих методов, мы действительно видим, что метод `Test` получает адрес, указывающий на место в стеке, где хранится ссылка на созданный экземпляр `SomeClass` (см. комментарии к листингу 13.39).

Листинг 13.39 ❖ Ассемблерный код методов из листинга 13.38

```
Program.Main(System.String[])
   L0000: sub rsp, 0x28 // задаем размер кадра стека
   L0004: xor eax, eax // обнуляем регистр EAX
   L0006: mov [rsp+0x20], rax // обнуляем позицию в стеке по адресу
                  // rsp+0x20 (где хранится локальная переменная)
   L000b: mov rcx, 0x7ffa69398840 // копируем таблицу методов класса SomeClass
                  // в регистр RCX
   L0015: call 0x7ffac3452520 // вызываем распределитель (в результате RAX будет
                  // содержать адрес нового объекта)
   L001a: mov [rsp+0x20], rax // сохраняем адрес нового объекта в стеке
   L001f: lea rcx, [rsp+0x20] // помещаем адрес локальной переменной в стеке
                  // в регистр RCX (это первый аргумент метода Test)
   L0024: call PassingByref.Test(SomeClass ByRef)
   L0029: nop
   L002a: add rsp, 0x28
   L002e: ret

PassingByref.Test(SomeClass ByRef)
   L0000: mov rax, [rcx] // разыменовываем адрес в RCX, помещая содержимое
                  // в RAX (в результате RAX содержит адрес объекта)
   L0003: mov dword [rax+0x8], 0xb// сохраняем значение 11 (0x0B) в нужном поле
                  // объекта
   L000a: ret
```

Ассемблерный код, похожий на тот, что показан в листинге 13.39, был бы сгенерирован, например, при использовании указателя на указатель в C++. Но как во время выполнения метода `Test` сборщик мусора узнает, что регистр RCX содержит адрес объекта? Ответ интересен: в случае метода `Test` из листинга 13.39 информация для GC пуста. Иными словами, метод `Test` настолько прост, что GC не станет прерывать его работу. Поэтому и сообщать GC ничего не нужно.

> В примере из листинга 13.39 экземпляр `SomeClass` жив из-за метода `Main`. Взглянув на информацию для GC, связанную с методом `Main`, мы увидим, что по адресу в стеке `rsp+0x20` находится живой корень (команда `!u -gcinfo` напечатает `Untracked: +sp+20`). Но это ничего не меняет в части того, как затем передается экземпляр: по ссылке или нет.

Если бы метод Test был более сложным, то JIT-компилятор мог бы сделать его полностью или частично прерываемым (см. главу 8). В последнем случае мы увидели бы различные безопасные точки, в некоторых из которых в составе живых корней указывались бы регистры (или адреса в стеке). В листинге 13.40 приведена выдержка из распечатки команды !u -gcinfo, входящей в расширение SOS программы WinDbg (подробное объяснение см. в главе 8).

Листинг 13.40 ❖ Пример JIT-компилированного кода и соответствующей информации для GC в случае более сложного метода Test (его код на C# не показан, поскольку не имеет отношения к делу)

```
> !u -gcinfo 00007ffc86850d00
Normal JIT generated code
CoreCLR.Unsafe.PassingByref.Test(CoreCLR.Unsafe.SomeClass ByRef)
Begin 00007ffc86850d00, size 44
push   rdi
push   rsi
sub    rsp,28h
mov    rsi,rcx
...
call   00007ffc`86850938
00000029 is a safepoint:
00000028 +rsi(interior)
...
Call   00007ffc`868508a0
00000033 is a safepoint:
00000032 +rsi(interior)
...
add    rsp,28h
pop    rsi
pop    rdi
ret
```

Эти живые корни были бы показаны как *внутренние указатели* (interior pointer), потому что управляемые указатели, вообще говоря, могут указывать внутрь объекта (вскоре мы объясним это). Таким образом, управляемые указатели всегда описываются как внутренние корни, хотя в нашем случае они и указывают на начало объекта. Интерпретация таких указателей возлагается на GC, как будет объяснено ниже.

Выше уже отмечалось, что наш пример несколько искусственный, поскольку встраивание явно запрещено. Если закомментировать атрибут NoInlining в листинге 13.37, то JIT-компилятор сгенерирует такой код:

```
Program.Main(System.String[])
  L0000: sub rsp, 0x28
  L0004: mov rcx, 0x7ffa69398840  // копируем таблицу методов SomeClass в регистр RCX
  L000e: call 0x7ffac3452520      // вызываем распределитель (в результате RAX будет
                                  // содержать адрес нового объекта)
  L0013: mov dword [rax+0x8], 0xb // непосредственно сохраняем значение 11 в нужном
                                  // поле объекта
  L001a: add rsp, 0x28
  L001e: ret
```

И снова обращаем внимание на высокоэффективные оптимизации, выполненные JIT-компилятором. Управляемые указатели сведены к простейшей обработке непосредственных адресов объектов.

Очень похожий код был бы сгенерирован, если бы вместо класса использовалась структура (см. листинг 13.41, аналогичный листингу 4.33 из главы 4). Но вот что особенно интересно: хотя теоретически известно, что метод `Test` из листинга 13.41 работает только с данными в стеке (локальной переменной типа значений SomeStruct), в информации для GC все равно перечисляются живые корни, т. к. используется управляемый указатель. Сборщик мусора может их проигнорировать.

Листинг 13.41 ❖ Простой случай передачи по ссылке (всего объекта типа значений)

```
static void Main(string[] args)
{
   SomeStruct someStruct = new SomeStruct();
   PassingByref.Test(ref someStruct);
   Console.WriteLine(someStruct.Field);
}

[MethodImpl(MethodImplOptions.NoInlining)]
public static void Test(ref SomeStruct data)
{
   data.Field = 11;
}
```

Управляемый указатель на объект, созданный в куче

Хотя управляемые указатели на объекты в стеке интересны, указатели на объекты в управляемой куче еще интереснее. В отличие от ссылки на объект, управляемый указатель может указывать внутрь объекта – на поле типа или элемент массива (рис. 13.3). Поэтому они и называются в литературе «внутренними указателями». И возникает любопытный вопрос: как сообщить сборщику мусора об указателях, указывающих на место внутри управляемого объекта?

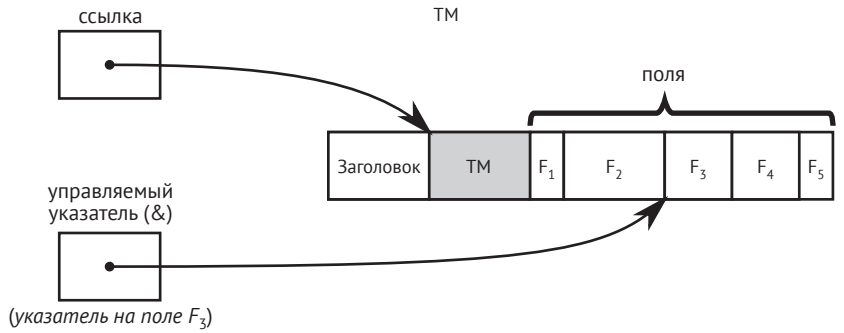


Рис. 13.3. Управляемый (также известный как внутренний, или byref-указатель) указатель и обычная ссылка на объект

Немного модифицируем код из листинга 13.37, чтобы передавать по ссылке только поле находящегося в куче экземпляра класса `SomeClass` (листинг 13.42). Ме-

тод Main не вызывает вопросов. Он создает объект типа SomeClass, передает ссылку на одно из его полей методу Test и печатает результат.

Но вот наш измененный метод Test теперь ожидает получить на вход управляемый указатель System.Int32&. Во время выполнения метод Test оперирует только управляемым указателем на int. Однако это не обычный указатель на int, а указатель на поле находящегося в куче объекта! Откуда GC знает, что нельзя убирать в мусор объект, внутрь которого указывает управляемый указатель? Ведь нет абсолютно никаких намеков на природу указателя int&.

Листинг 13.42 ❖ Простой случай передачи по ссылке (поля объекта)

```
static void Main(string[] args)
{
   SomeClass someClass = new SomeClass();
   PassingByref.Test(ref someClass.Field);
   Console.WriteLine(someClass.Field); // печатается "11"
}

public class PassingByref
{
   [MethodImpl(MethodImplOptions.NoInlining)]
   public static void Test(ref int data)
   {
      data = 11; // это должно сохранить жизнь объекту, которому принадлежит поле!
   }
}
```

Прежде всего отметим, что наш искусственный метод Test будет превращен JIT-компилятором в атомарный (с точки зрения GC) метод, который GC вообще не будет прерывать – по аналогии с кодом в листинге 13.37 (листинг 13.43). Поэтому для такого простого метода вопрос о надлежащем извещении о корнях вообще не стоит.

Листинг 13.43 ❖ Ассемблерный код, сгенерированный JIT-компилятором по коду в листинге 13.42

```
Program.Main(System.String[])
   L0000: sub rsp, 0x28
   L0004: mov rcx, 0x7ffa6d128840
   L000e: call 0x7ffac3452520
   L0013: lea rcx, [rax+0x8]
   L0017: call PassingByref.Test(Int32 ByRef)
   L001c: nop
   L001d: add rsp, 0x28
   L0021: ret

PassingByref.Test(Int32 ByRef)
   L0000: mov dword [rcx], 0xb
   L0006: ret
```

Но предположим, что метод Test настолько сложен, что придется сгенерировать прерываемый код. В листинге 13.44 показано, как в этом случае мог бы выглядеть результат JIT-компиляции. Регистр RSI, в котором хранится значение цело-

численного поля, переданного в качестве аргумента в регистре RCX, помечен как внутренний указатель.

Листинг 13.44 ❖ Фрагменты сгенерированного JIT-компилятором ассемблерного кода, который стал полностью прерываемым

```
> !u -gcinfo 00007ffc86fb0ce0
Normal JIT generated code
CoreCLR.Unsafe.PassingByref.Test(Int32 ByRef)
Begin 00007ffc86fb0ce0, size 41
push  rdi
push  rsi
sub   rsp,28h
mov   rsi,rcx
00000009 interruptible
00000009 +rsi(interior)
...
0000003a not interruptible
0000003a -rsi(interior)
add   rsp,28h
pop   rsi
pop   rdi
ret
```

Если начнется сборка мусора и метод `Test` будет приостановлен, когда в регистре RSI такой внутренний указатель, то GC должен будет интерпретировать его, чтобы найти соответствующий объект. В общем случае это нетривиальная задача. Можно было бы вообразить простой алгоритм, который начинает с адреса, хранящегося в этом указателе, и пытается найти начало объекта, просматривая память слева от него байт за байтом[1]. Очевидно, что такой подход неэффективен, и у него много недостатков:

- внутренний указатель может указывать на поле, находящееся далеко от начала большого объекта (или на далекий элемент очень большого массива), поэтому просматривать пришлось бы большой участок памяти;
- определить, где начинается объект, нелегко – это можно было бы сделать, проверив, что 8 последовательных байт (или четыре в 32-разрядном случае) образуют допустимый адрес таблицы методов, но это только увеличивает сложность алгоритма. Можно было бы представить себе какие-нибудь «маркерные» байты, находящиеся в начале каждого объекта, но это потребовало бы накладных расходов на хранение только для того, чтобы поддержать редкое использование внутреннего указателя (да и трудно было бы определить маркерные байты, так чтобы они однозначно идентифицировали начало объекта);
- обо всех управляемых указателях сообщается как о внутренних, поэтому они могут указывать на стек, и тогда вообще нет смысла искать объект, в который они указывают (поскольку указатель может указывать внутрь структуры, размещенной в стеке).

[1] На каждом шаге сдвигаться пришлось бы только на один байт, потому что нет никаких гарантий того, что внутренние указатели выровнены на начало объекта.

Надеюсь, вы уже поняли, что такой алгоритм непрактичен. Нужно что-то более хитрое, что позволило бы эффективно интерпретировать внутренние указатели.

На самом деле мы уже видели этот механизм. Во время сборки мусора внутренние указатели транслируются в соответствующие объекты благодаря кирпичам и деревьям заполненных блоков, описанным в главе 9. По заданному адресу вычисляется запись в таблице кирпичей и обходится соответствующее ей дерево заполненных блоков, чтобы найти заполненный блок, которому этот адрес принадлежит (см. рис. 9.9 и 9.10 в главе 9). Затем просматриваются все объекты в этом заполненном блоке и находится тот, который содержит нужный адрес[1].

Понятно, что у такого алгоритма тоже есть своя цена. Для обхода дерева заполненных блоков и просмотра блока требуется время. Разыменование внутреннего указателя тоже нетривиально. И это вторая причина, по которой внутренние указатели не могут находиться в куче (особенно в полях объектов), – создание сложного графа объектов, ссылающихся друг на друга с помощью внутренних указателей, оказалось бы слишком дорогим. Предоставление такой гибкости просто не окупило бы сопутствующих накладных расходов.

Отметим также, что при подобной реализации разыменование внутреннего указателя возможно только во время сборки мусора после этапа планирования. Лишь тогда строятся заполненные и пустые блоки, а вместе с ними и дерево заполненных блоков.

> Если вы хотите самостоятельно изучить код CoreCLR, связанный с внутренними указателями, начните с метода `gc_heap::find_object(uint8_t* interior, ...)`. Просмотр блоков производится в методе `gc_heap::find_first_object(uint8_t* start, uint8_t* first_object)`.

Интерпретация внутренних указателей позволяет делать изумительные и, на первый взгляд, опасные вещи. Например, можно вернуть управляемый указатель на локальный экземпляр класса или элемент локального массива (листинг 13.45). Это кажется противоречащим интуиции – как можно вернуть из метода ссылку на один элемент целочисленного массива, хотя сам массив вроде бы уже стал недостижимым? Но на самом деле не стал, потому что после завершения такого метода возвращенный внутренний указатель оказывается единственным корнем массива.

Листинг 13.45 ❖ Пример внутреннего указателя, оставшегося единственным корнем

```
public static ref int ReturnByRefReferenceTypeInterior(int index)
{
   int[] localArray = new[] { 1, 2, 3 };
   return ref localArray[index];
}

static void Main(string[] args)
{
   ref int byRef = ref ReturnByRefReferenceTypeInterior(0);
   // Массив, созданный в приведенном выше классе, уже недоступен из программы,
```

[1] Просмотр заполненного блока возможен, потому что начало любого блока совпадает с началом объекта, а поскольку размер каждого объекта известен, то легко перейти от текущего к следующему.

```
   // но все еще жив
   byRef = 4; // по ссылке, чтобы предотвратить раннюю сборку корней
}
```

Сам массив все еще жив из-за внутреннего указателя, однако ссылку на объект массива мы потеряли (рис. 13.4). Благодаря вышеупомянутому ограничению (доступности кирпичей и дерева заполненных блоков) такой указатель невозможно преобразовать во время выполнения обратно в ссылку на объект, внутрь которого он указывает.

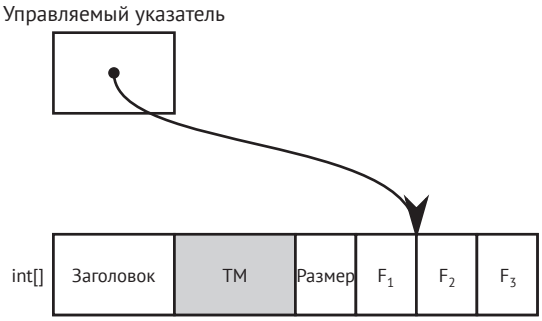


Рис. 13.4 ❖ Управляемый указатель – единственный корень объекта массива (указывающий на один из его элементов)

Мы можем немного поэкспериментировать с типом `WeakReference` с целью понаблюдать за поведением внутреннего указателя (чтобы развлечься или посмотреть интересные модульные тесты). В немного модифицированном коде в листинге 13.46 вместо простого массива используется класс `ArrayWrapper`, который окажется полезен в нашем эксперименте. Метод возвращает целочисленное поле объекта `ArrayWrapper` по ссылке. Кроме того, метод `ObservableReturnByRefReferenceTypeInterior` возвращает слабую ссылку `WeakReference` на созданный объект, чтобы можно было наблюдать за временем его жизни.

Листинг 13.46 ❖ Пример внутреннего указателя, оставшегося единственным корнем

```
public static ref int ObservableReturnByRefReferenceTypeInterior(int index,
      out WeakReference wr)
{
   ArrayWrapper wrapper = new ArrayWrapper() { Array = new[] {1, 2, 3}, Field = 0 };
   wr = new WeakReference(wrapper);
   return ref wrapper.Field;
}

static void Main(string[] args)
{
   ref int byRef = ref ObservableReturnByRefReferenceTypeInterior(2,
      out WeakReference wr);
   byRef = 4;
   for (int i = 0; i < 3; ++i)
   {
      GC.Collect();
```

```
      Console.WriteLine(byRef + " " + wr.IsAlive);
   }
   GC.Collect();
   Console.WriteLine(wr.IsAlive);
}
```

Теперь мы можем наблюдать за экземпляром `ArrayWrapper` в методе `Main` и убедиться, что он жив до тех пор, пока используется возвращенный внутренний указатель, представленный локальной ссылочной переменной `byRef` (листинг 13.47).

Листинг 13.47 ❖ Результаты работы программы в листинге 13.46

```
4 True
4 True
4 True
False
```

Если в WinDbg создать дамп памяти в точке внутри цикла в методе `Main` из листинга 13.46, то мы сможем увидеть, что корнем экземпляра `ArrayWrapper` является внутренний указатель, хранящийся в стеке (листинг 13.48).

Листинг 13.48 ❖ Команды SOS dumpheap и gcroot в WinDbg – внутренний указатель хранится в стеке (регистр RBP адресует стек)

```
> !dumpheap -type ArrayWrapper
         Address               MT Size
0000027b00023d20 00007ffdace07220   32
...
> !gcroot 0000027b00023d20
Thread 3f48:
    000000a65857de60 00007ffdacf60598 CoreCLR.Unsafe.Program.Main
    (System.String[])
        rbp-50: 000000a65857dec0 (interior)
            -> 0000027b00023d20 CoreCLR.Unsafe.ArrayWrapper
Found 1 unique roots (run '!GCRoot -all' to see all roots).
```

Другие инструменты, в частности PerfView, обычно включают такой объект в состав обычных корней в локальных переменных (корень вида [local vars] в случае PerfView). Иногда это сбивает с толку, поскольку в коде нет прямой связи между методом `Main` и типом `ArrayWrapper` (и такая связь могла бы быть еще менее заметной, если бы внутренний указатель указывал на более глубоко вложенный тип).

Еще интереснее, что такое использование внутреннего указателя может приводить к неожиданному (но тем не менее разумному) поведению. Изменим код в листинге 13.46, так чтобы он возвращал по ссылке заданный элемент внутреннего массива `ArrayWrapper`, как в листинге 13.45 (см. листинг 13.49).

Листинг 13.49 ❖ Пример внутреннего указателя, оставшегося единственным корнем

```
public static ref int ObservableReturnByRefReferenceTypeInterior(int index,
   out WeakReference wr)
{
   ArrayWrapper wrapper = new ArrayWrapper() {Array = new[] {1, 2, 3}, Field = 0};
   wr = new WeakReference(wrapper);
   return ref wrapper.Array[index];
}
```

После этого метод `Main` показывает другие результаты (листинг 13.50). Создается впечатление, что возвращенный экземпляр `ArrayWrapper` становится недоступным (и, значит, убирается в мусор) вскоре после завершения метода `ObservableReturnByRefReferenceTypeInterior`. Это может показаться удивительным, поскольку обернутый массив все еще удерживается в живых внутренним указателем, возвращенным по ссылке!

Листинг 13.50 ❖ Результаты работы программы в листинге 13.49

```
4 False
4 False
4 False
False
```

Внимательный читатель, наверное, уже разобрался, в чем дело. Объяснить, что происходит, легко, если показать релевантные связи (рис. 13.5). После того как метод `ObservableReturnByRefReferenceTypeInterior` закончил работу, но перед первым обращением к `GC.Collect`, ситуация выглядит, как показано на рис. 13.5a, – экземпляр `ArrayWrapper` еще жив и ссылается на массив `int[]` с помощью `Array`. И существует ссылочная локальная переменная `byRef`, указывающая внутрь того же массива. Во время сборки мусора массив `int[]` все еще удерживается внутренним указателем. Но на экземпляр `ArrayWrapper` ничто не указывает, поэтому он считается недостижимым и убирается в мусор.

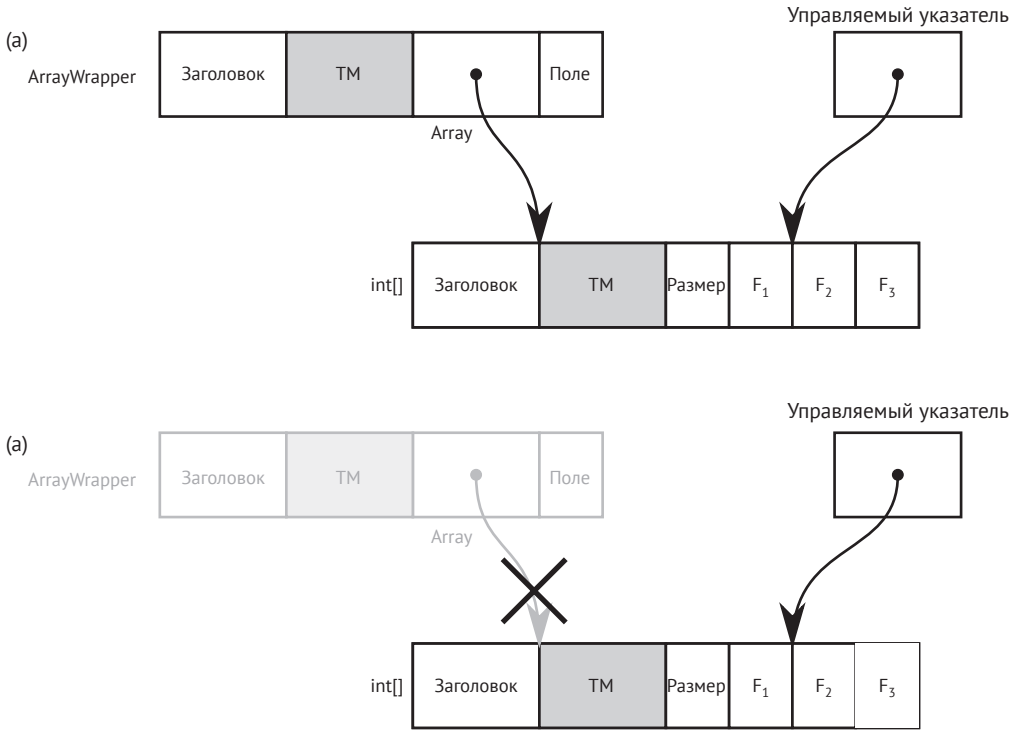


Рис. 13.5 ❖ Связи между объектами в листинге 13.49:
(a) до сборки мусора; (b) после сборки мусора

Надеюсь, вы уже поняли, к чему я веду: нужно избегать возврата ссылок на объекты, для которых корнями являются только внутренние указатели. Это забавно, но может приводить к путанице!

> Конечно, внутренние указатели также учитываются во время перемещения при уплотняющей сборке мусора. Их значение (адрес) изменяется в соответствии со смещением блока, как и для обычных ссылок.

Может показаться неожиданным, что код из листинга 13.45 правильно обрабатывается сборщиком мусора. Не менее удивительным может показаться код в листинге 13.51, хотя мы уже понимаем, почему он работает. Даже если массив `int` на первый взгляд кажется временным, из-за внутреннего указателя на первый элемент он будет оставаться в живых, пока этот указатель используется.

Листинг 13.51 ❖ Ссылочная локальная переменная, содержащая внутренний указатель на временный (но еще живой) управляемый массив

```
ref var local = ref (new int[1])[0];
```

Мы можем использовать такой «магический» синтаксис, чтобы написать обобщенный метод для создания внутренних указателей (листинг 13.52). Использовать его стоит только для тестирования функциональности и производительности (по крайней мере, я не могу себе представить, как его применить в реальной программе).

Листинг 13.52 ❖ Код, который создает внутренний указатель на данный объект

```
public class Helpers {
   public static ref T MakeInterior<T>(T obj) => ref (new T[] { obj })[0];
}
```

> Ради гибкости управляемые указатели могут также указывать на неуправляемые области памяти. Понятно, что сборщик мусора игнорирует их на этапах пометки и уплотнения.

Управляемые указатели в C# – ссылочные переменные

Как уже было сказано, ссылочные переменные с ключевым словом `ref` (ссылочные параметры, ссылочные локальные переменные и возвращаемые ссылочные значения) – это обертки вокруг управляемых указателей. Очевидно, их не следует рассматривать как указатели. Это переменные! Почитайте замечательную статью Владимира Садова «Возвращаемые ссылочные значения – не указатели» в http://mustoverride.com/refs-not-ptrs/.

Конечно, любопытно экспериментировать со всеми этими большими и малыми управляемыми указателями и ссылочными переменными, но зачем они вообще нужны? Для чего их вводили в язык? По одной-единственной, но очень важной причине:

чтобы избежать копирования данных, особенно при работе с большими структурами, но сделать это способом, обеспечивающим безопасность типов!

У типов значений много преимуществ, которые мы уже видели в этой книге: благодаря отказу от выделения памяти в куче и улучшенной локальности дан-

ных код может работать намного быстрее. Но из-за семантики передачи по значению (подробности см. в главе 4) возникают проблемы – JIT-компилятор делает все возможное, чтобы избежать копирования небольших структур, но это детали реализации, управлять которыми мы не можем. Всякий раз, когда тип значений (скорее всего, написанная нами структура) передается в качестве параметра или возвращается из метода, следует предполагать, что имеет место нежелательное копирование памяти.

Ссылочные переменные вводились для того, чтобы преодолеть этот недостаток. Они гарантируют, что тип значений будет передаваться по ссылке, так что мы получим лучшее из обоих миров – избежим выделения памяти в куче и при этом сможем использовать эти типы как ссылочные (поскольку они обладают семантикой ссылок).

Чтобы не быть голословными, напишем простой тест производительности (листинг 13.53). В нем есть методы, которые передают тип значений (структуры) по значению и по ссылке. Чтобы измерить влияние размера передаваемой структуры, мы взяли три разные структуры – из 8, 28 и 48 целых чисел (размером 32, 112 и 192 байта соответственно). Для краткости показано определение лишь самой маленькой структуры. Кроме того, в тесте есть метод, который принимает в качестве аргумента класс аналогичного размера.

Листинг 13.53 ❖ Тест для измерения производительности передачи по значению и по ссылке

```
public unsafe class ByRef
{
   [GlobalSetup]
   public void Setup()
   {
      this.struct32B = new Struct32B();
      // ...
   }

   [Benchmark]
   public int StructAccess()
   {
      int result = 0;
      result = Helper1(struct32B);
      return result;
   }

   [Benchmark]
   public int ByRefStructAccess()
   {
      int result = 0;
      result = Helper1(ref struct32B);
      return result;
   }

   [Benchmark]
   public int ClassAccess()
   {
      int result = 0;
```

```
      result = Helper2(bigClass);
      return result;
   }

   [MethodImpl(MethodImplOptions.NoInlining)]
   private int Helper1(Struct32B data)
   {
      return data.Value1;
   }

   [MethodImpl(MethodImplOptions.NoInlining)]
   private int Helper1(ref Struct32B data)
   {
      return data.Value1;
   }

   [MethodImpl(MethodImplOptions.NoInlining)]
   private int Helper2(BigClass data)
   {
      return data.Value1;
   }

   public struct Struct32B
   {
      public int Value1;
      public int Value2;
      public int Value3;
      public int Value4;
      public int Value5;
      public int Value6;
      public int Value7;
      public int Value8;
   }
}
```

Результаты работы библиотеки BenchmarkDotNet недвусмысленно показывают преимущества передачи по ссылке (листинг 13.54). Производительность передачи по ссылке не зависит от размера структуры (как передача класса по ссылке не зависит от его размера). С другой стороны, при передаче по значению (включающей копирование структуры) операция тем медленнее, чем больше размер структуры. То же самое справедливо и для возврата значения по ссылке, поэтому похожий тест для краткости опущен.

Листинг 13.54 ❖ Результаты теста производительности в листинге 13.53

```
          Метод |  Среднее |  Выделено |
--------------- |---------:|----------:|
      Struct32B | 1.560 нс |       0 Б |
     Struct112B | 5.229 нс |       0 Б |
     Struct192B | 7.457 нс |       0 Б |
 ByRefStruct32B | 1.332 нс |       0 Б |
ByRefStruct112B | 1.343 нс |       0 Б |
ByRefStruct192B | 1.329 нс |       0 Б |
    ClassAccess | 1.098 нс |       0 Б |
```

Таким образом, ссылочные переменные особенно важны при работе с большими типами значений. Располагая ими, мы можем больше не опасаться копирования структур. Мы даже можем управлять их изменяемостью с помощью неизменяемых ссылок и структур (readonly refs and readonly structs), об этом мы поговорим дальше. И все это было сделано для того, чтобы типы значений было удобнее использовать в коде, требующем высокой производительности.

Но ссылочные переменные могут быть полезны даже в тривиальных ситуациях. В листинге 13.55 приведен хороший пример, взятый из документации .NET. Метод, который ищет значение в данной матрице, написан двумя способами: найденный элемент возвращается по значению в виде кортежа (value tuple) или по ссылке. Существенной разницы в производительности не будет (поскольку возвращаемый кортеж, скорее всего, будет помещен в регистры, и копирования структуры не произойдет). Но второй вариант позволяет очень быстро изменить возвращенное значение. В первом случае метод возвращает только индексы элемента, и для изменения потребуется второй доступ к матрице по этим индексам. Понятно, что выбор зависит от того, какой API мы хотели бы предоставить пользователям метода. Конечно, различие в производительности одной операции не так уж велико, но такие различия накапливаются, если метод вызывается очень часто.

Листинг 13.55 ❖ Пример возврата значения по ссылке с целью повысить гибкость и скорость изменения

```
public static (int i, int j) FindValueReturn(int[,] matrix, Func<int, bool> predicate)
{
   for (int i = 0; i < matrix.GetLength(0); i++)
      for (int j = 0; j < matrix.GetLength(1); j++)
         if (predicate(matrix[i, j]))
            return (i, j);
   return (-1, -1); // Не найден
}

public static ref int FindRefReturn(int[,] matrix, Func<int, bool> predicate)
{
   for (int i = 0; i < matrix.GetLength(0); i++)
      for (int j = 0; j < matrix.GetLength(1); j++)
         if (predicate(matrix[i, j]))
            return ref matrix[i, j];
   throw new InvalidOperationException("Не найден");
}
```

> Передачу структур по ссылке мы скоро объясним, но в данном контексте это ничего не меняет.

Благодаря ссылочным переменным могут стать более популярными *коллекции, возвращающие ссылки*. Это особенно полезно, когда в коллекции хранятся массивные типы значений, поскольку открывается возможность обращаться к элементам коллекции без копирования. Пример простой коллекции такого рода показан в листинге 13.56. Здесь определен индексатор, возвращающий указанный элемент по ссылке. Это предоставляет прямой доступ к элементам без копирования, причем такой же, как если бы хранились обычные ссылки (см. метод `Main` в листинге 13.56).

Листинг 13.56 ❖ Простой пример коллекции, возвращающей ссылки

```
public class SomeStructRefList
{
   private SomeStruct[] items;

   public SomeStructRefList(int count)
   {
      this.items = new SomeStruct[count];
   }

   public ref SomeStruct this[int index] => ref items[index];
}

static void Main(string[] args)
{
   SomeStructRefList refList = new SomeStructRefList(3);
   for (var i = 0; i < 3; ++i)
      refList[i].Field = i;
   for (var i = 0; i < 3; ++i)
      Console.Write(refList[i].Field); // печатается 012
}
```

Иногда необходимо, чтобы API не позволял модифицировать возвращенные элементы (т. е. нужно, чтобы коллекция была доступна только для чтения). Это можно сделать с помощью рассмотренных выше неизменяемых ссылок (readonly refs) (листинг 13.57). Но помните о последствиях, особенно о защитном копировании значения, если метод вызывается таким образом (см. метод `Main` в листинге 13.57).

Листинг 13.57 ❖ Простой пример постоянной коллекции, возвращающей ссылки

```
public struct SomeStruct
{
   public int Field;

   public void ModifyMe()
   {
      this.Field = 9;
   }
}

public class SomeStructReadOnlyRefList
{
   private SomeStruct[] items;

   public SomeStructReadOnlyRefList(int count)
   {
      this.items = new SomeStruct[count];
   }

   public ref readonly SomeStruct this[int index] => ref items[index];
}

static void Main(string[] args)
{
   SomeStructReadOnlyRefList readOnlyRefList = new SomeStructReadOnlyRefList(3);
```

```
   for (var i = 0; i < 3; ++i)
      //readOnlyRefList[i].Field = i;
      // Error CS8332: Cannot assign to a member of property 'SomeStructRefList.this[int]'
      // because it is a readonly variable
      readOnlyRefList[i].ModifyMe(); // Вызывается на защитной копии!
                                     // Исходное значение не изменяется.
   for (var i = 0; i < 3; ++i)
      Console.WriteLine(readOnlyRefList[i].Field); // печатается 000, а не 999
}
```

Сравнив релевантные части CIL-кода метода `Main` в листингах 13.56 и 13.57, мы увидим защитное копирование, о котором говорили. Код, возвращающий ссылку, просто вызывает метод `ModifyMe` на элементе, возвращенном индексатором:

```
IL_0008: ldc.i4.0
IL_0009: callvirt instance valuetype SomeStruct& SomeStructRefList::get_Item(int32)
IL_000e: call instance void SomeStruct::ModifyMe()
```

С другой стороны, постоянное ссылочное значение копируется в дополнительную временную локальную переменную:

```
IL_0008: ldc.i4.0
IL_0009: callvirt instance valuetype SomeStruct& modreq(InAttribute)
  SomeStructRefList2::get_Item(int32)
IL_000e: ldobj C/SomeStruct // загрузить объект по возвращенному адресу в стеке
вычислений
IL_0013: stloc.0     // сохранить значение из стека вычислений в локальной переменной
IL_0014: ldloca.s 0  // загрузить адрес локальной переменной
IL_0016: call instance void C/SomeStruct::ModifyMe()
```

После введения более гибких ссылочных переменных в C# 7.2 можно ожидать, что во все большем количестве открытых API коллекций будут появляться методы с семантикой возврата по ссылке. Такие методы получили стандартное имя `ItemRef`. В настоящее время подобные изменения уже внесены в большинство неизменяемых коллекций из пространства имен `System.Collections.Immutable` (например, `ImmutableArray`, `ImmutableList` и т. д.). Логика возврата ссылочных значений может быть и сложнее, чем одиночный доступ к хранилищу. Например, внутреннее хранилище коллекции `ImmutableSortedSet` основано на узлах типа `Node`, образующих двоичное сбалансированное AVL-дерево. Поэтому реализация `ItemRef` содержит обход двоичного дерева (листинг 13.58).

Листинг 13.58 ❖ Пример реализации более сложной коллекции, возвращающей ссылки

```
public sealed partial class ImmutableSortedSet<T>
{
   internal sealed class Node : IBinaryTree<T>, IEnumerable<T>
   {
      ...
      internal ref readonly T ItemRef(int index)
      {
         if (index < _left._count)
         {
            return ref _left.ItemRef(index);
```

```
            }
            if (index > _left._count)
            {
               return ref _right.ItemRef(index - _left._count - 1);
            }
            return ref _key;
         }
         ...
      }
      ...
}
```

Реализация возврата по ссылке не всегда тривиальна, потому что раскрывает сам элемент коллекции. Иногда это нежелательно, поскольку:

- может потребоваться специальная обработка элементов, которая пропускается, если доступ к элементу предоставляется по ссылке. Например, возможно, что каждое изменение элемента коллекции должно протоколироваться или, скажем, версионироваться;
- может потребоваться реорганизация внутреннего хранилища коллекции, в результате чего возвращенная ссылка на значение станет недействительной. Например, в качестве хранилища может использоваться массив, который создается заново по мере роста коллекции.

Именно из-за этих двух проблем добавление `ItemRef` в популярные коллекции `List<T>` и `Dictionary<TKey, TValue>` сопряжено с трудностями:

- используется внутренний счетчик `_version` (нужен для сериализации);
- элементы могут реорганизовываться из-за того, что внутреннее хранилище основано на массиве.

И снова о структурах

Структуры были в .NET с самого начала. Но нельзя не признать, что тогда они не пользовались особой популярностью. И лишь в последние год-два мы наблюдаем улучшение осведомленности о структурах и увеличение частоты их применения на практике. Во времена, когда требуется все более высокая производительность, ограничения, налагаемые на сборку мусора и на работу с памятью вообще, становятся жестче. Поэтому происходит возврат к структурам – так как при аккуратном использовании они не требуют выделения памяти из кучи, а значит, сборщик мусора не привлекается, что сильно повышает производительность. Будучи фанатом производительности, я счастлив видеть это. Есть много мест, где беспечному выделению памяти в куче пришли на смену основанные на структурах типы, позволяющие избежать этого (зачастую полностью отказаться).

Я выступаю всецело за такое направление развития. Помимо растущего интереса к структурам внутри команд в самой компании Майкрософт, занимающихся .NET, в C# появляются все новые средства, ориентированные на структуры. Многие из них уже упоминались: ссылочные локальные переменные и возвращаемые значения дополняют передачу аргументов по ссылке, освобождая типы значений от необходимости копирования. Постоянные ссылки и in-параметры упрощают контроль над изменяемостью значений. А в C# 7.2 добавлено еще два механизма,

которые заслуживают подробного описания, – постоянные (readonly) структуры и ссылочные структуры. Я ожидаю заметного роста их популярности в ближайшие годы, по крайней мере в коде, предъявляющем повышенные требования к производительности. Я, впрочем, не думаю, что бизнес-уровень приложений, основанный на операциях CRUD[1], ни с того ни с сего перейдет на структуры.

Постоянные структуры

Мы уже встречались с постоянными ref- и in-параметрами, которые запрещают изменение аргумента. Иногда очень полезно иметь гарантии, что ссылочные переменные, используемые для типов значений, не могут быть изменены программистом. Но можно пойти еще дальше и создать неизменяемые структуры, которые не могут быть изменены после создания. Надеюсь, вы уже понимаете, какие оптимизации могли бы применить компилятор C# и JIT-компилятор при таких условиях – например, безопасно избавиться от защитного копирования при вызове методов.

Для определения постоянной структуры нужно добавить модификатор `readonly` в ее объявление (листинг 13.59). Тогда компилятор C# позаботится о том, чтобы все поля структуры были постоянными.

Листинг 13.59 ❖ Пример объявления постоянной структуры

```
public readonly struct ReadonlyBook
{
   public readonly string Title;
   public readonly string Author;

   public ReadonlyBook(string title, string author)
   {
      this.Title = title;
      this.Author = author;
   }

   public void ModifyAuthor()
   {
      //this.Author = "XXX"; // Ошибка компиляции: A readonly field cannot be assigned to
                             // (except in a constructor or a variable initializer)
      Console.WriteLine(this.Author);
   }
}
```

Если тип является (или может являться) неизменяемым с точки зрения бизнес-требований или логики, то всегда стоит подумать об использовании постоянной структуры, передаваемой по ссылке (с ключевым словом `in`) в тех частях кода, которые требуют высокой производительности.

Процитируем MSDN: «Модификатор `in` можно использовать всюду, где аргументом является постоянная структура. Кроме того, постоянную структуру можно возвращать по ссылке, если возвращается объект, время жизни которого больше, чем время работы метода, из которого он возвращен». Таким образом, постоян-

[1] Создание, чтение, обновление, удаление. – *Прим. перев.*

ная структура – очень удобный способ безопасного и производительного обращения с неизменяемыми типами.

В качестве примера изменим класс BookCollection в листинге 13.28, так чтобы внутри использовался массив постоянных, а не обычных структур (листинг 13.60). То, что элементы массива создаются в куче, нормально, потому что экземпляры класса ReadOnlyBookCollection – объекты, находящиеся в куче. Однако все гарантии неизменяемости сохраняются, поэтому компилятор может опустить создание защитных копий в методе CheckBook.

Листинг 13.60 ❖ Модификация кода из листинга 13.28 – хранение постоянных структур

```
public class ReadOnlyBookCollection
{
   private ReadonlyBook[] books = {
      new ReadonlyBook("Call of the Wild, The", "Jack London" ),
      new ReadonlyBook("Tale of Two Cities, A", "Charles Dickens")
   };
   private ReadonlyBook nobook = default;
   public ref readonly ReadonlyBook GetBookByTitle(string title)
   {
      for (int ctr = 0; ctr < books.Length; ctr++)
      {
         if (title == books[ctr].Title)
            return ref books[ctr];
      }
      return ref nobook;
   }

   public void CheckBook(in ReadonlyBook book)
   {
      //book.Title = "XXX"; // компилятор выдал бы ошибку
      book.DoSomething();   // гарантируется, что DoSomething не изменяет поля book
   }
}

public static void Main(string[] args)
{
   var coll = new ReadOnlyBookCollection();
   ref readonly var book = ref coll.GetBookByTitle("Call of the Wild, The");
   book.Author = "XXX"; // Ошибка компиляции : A readonly field cannot be assigned to
                        // (except in a constructor or a variable initializer)
}
```

Ссылочные структуры (byref-подобные типы)

Мы уже несколько раз упоминали, что на управляемые указатели налагается ряд обоснованных ограничений – в частности, они не могут находиться в управляемой куче (в виде поля ссылочного типа или просто в результате упаковки). Однако в некоторых ситуациях, которые будут описаны ниже, было бы хорошо, если бы у нас был тип, в котором есть управляемый указатель. У такого типа были бы те же ограничения, что и у управляемого указателя (чтобы не нарушать ограничения на содержащийся внутри управляемый указатель). Поэтому такие структуры

часто называются *byref-подобными типами* (т. к. другое название управляемого указателя – просто byref).

Начиная с версии C# 7.3 разрешено объявлять собственные byref-подобные типы в виде *ссылочных структур*, для чего нужно добавить модификатор `ref` в объявление структуры (листинг 13.61).

Листинг 13.61 ❖ Пример объявления ссылочной структуры

```
public ref struct RefBook
{
   public string Title;
   public string Author;
}
```

Компилятор C# налагает много ограничений на ссылочные структуры, смысл которых – гарантировать, что память для них выделяется только в стеке:

- они не могут быть полем класса или обычной структуры (поскольку такие поля могут упаковываться);
- по той же причине они не могут быть статическими полями;
- их нельзя упаковывать – поэтому запрещается присваивать их объекту, динамическому типу или типу интерфейса (или приводить к такому типу). Также они не могут быть элементами массива, поскольку в массиве хранятся упакованные структуры;
- их нельзя использовать в качестве итератора или аргумента обобщенного типа, и они не могут реализовывать интерфейсы (поскольку в этом случае могли бы стать объектом упаковки);
- их нельзя использовать в качестве локальной переменной в асинхронном методе, поскольку тогда они могли бы быть упакованы как часть асинхронного конечного автомата;
- они не могут захватываться лямбда-выражениями или локальными функциями, так как тогда были бы упакованы соответствующим классом замыкания (см. главу 6).

Попытка использовать ссылочную структуру в любом из описанных выше сценариев закончится ошибкой компиляции. В листинге 13.62 приведено несколько примеров.

Листинг 13.62 ❖ Примеры недопустимого использования ссылочных структур

```
public class RefBookTest
{
   private RefBook book; // ошибка компиляции: Field or auto-implemented
                         // property cannot be of type 'RefBook' unless
                         // it is an instance member of a ref struct
   public void Test()
   {
      RefBook localBook = new RefBook();
      object box = (object) localBook; // ошибка компиляции: Cannot convert type
                                       // 'CoreCLR.UnsafeTests.RefBook' to 'object'
      RefBook[] array = new RefBook[4]; // ошибка компиляции: Array elements cannot
                                        // be of type 'RefBook'
   }
}
```

Как и управляемые указатели, ссылочные структуры можно использовать только как параметры методов и локальные переменные. Ссылочная структура также может быть полем другой ссылочной структуры (листинг 13.63).

Листинг 13.63 ❖ Пример использования ссылочной структуры в качестве поля другой ссылочной структуры

```
public ref struct RefBook
{
   public string Title;
   public string Author;
   public RefPublisher Publisher;
}

public ref struct RefPublisher
{
   public string Name;
}
```

Кроме того, разрешается объявлять «постоянные ссылочные структуры» (`readonly ref struct`), т. е. неизменяемые структуры, которые могут существовать лишь в стеке. Это дает возможность компилятору C# и JIT-компилятору дополнительно оптимизировать их использование (например, не создавать защитные копии).

Хотя мы уже знаем, что дают ссылочные структуры, возникает вопрос: а есть ли от них польза? Понятно, что должна быть, иначе бы их не включили в язык. Благодаря наложенным ограничениям они обладают двумя очень важными свойствами:

- они не могут быть созданы в куче – это позволяет использовать их специальным образом, т. к. время их жизни ограничено только временем работы метода. Как отмечалось в начале этого раздела, главное достоинство заключается в том, что поле ссылочной структуры может содержать управляемый указатель (хотя в текущей версии C# это свойство напрямую не раскрывается, как мы вскоре поясним);
- к ним нельзя обращаться из нескольких потоков – поскольку между потоками нельзя передавать адреса, принадлежащие стеку, гарантируется, что к находящейся в стеке ссылочной структуре будет обращаться только поток, в котором она создана. Это естественным образом устраняет досадные проблемы синхронизации без каких-либо дополнительных затрат.

Эти два свойства уже делают ссылочные структуры интересными. Но главным мотивом для их реализации стала структура `Span<T>`, о которой мы расскажем в следующей главе.

> Можно было бы спросить, почему в объявлениях ссылочных структур не использовать ключевое слово `stackonly` вместо `ref`? Потому что на «ссылочные структуры» налагаются более сильные ограничения, чем просто «выделение памяти только в стеке». Например, их нельзя использовать в качестве аргументов обобщенных типов или в качестве указательных типов. Так что слово `stackonly` только вводило бы в заблуждение.

Буферы фиксированного размера

Если полем структуры является массив, то понятно, что в самой структуре будет храниться только ссылка на массив, находящийся в куче (листинг 13.64 и рис. 13.6а). Это может отвечать или не отвечать вашим требованиям.

Листинг 13.64 ❖ Пример использования массива в поле структуры

```
public struct StructWithArray
{
   public char[] Text;
   public int F2;
   // прочие поля...
}

static void Main(string[] args)
{
   StructWithArray data = new StructWithArray();
   data.Text = new char[128];
   ...
}
```

Однако существует возможность разместить массив целиком в самой структуре, тогда он называется *буфером фиксированного размера*. Единственное ограничение – массив должен иметь предопределенный размер и принадлежать одному из примитивных типов: `bool`, `byte`, `char`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float`, `double`. Кроме того, структура, в которой используется буфер фиксированного размера, должна быть помечена ключевым словом `unsafe` (листинг 13.65 и рис. 13.6b). В классах буферы фиксированного размера запрещены. По рис. 13.6b понятно, что лучше называть их буферами, а не массивами, поскольку это просто последовательно расположенные элементы без какой-либо информации о типе или размере.

Листинг 13.65 ❖ Пример использования буфера фиксированного размера в структуре

```
public unsafe struct StructWithFixedBuffer
{
   public fixed char Text[128];
   public int F2;
   // прочие поля...
}
```

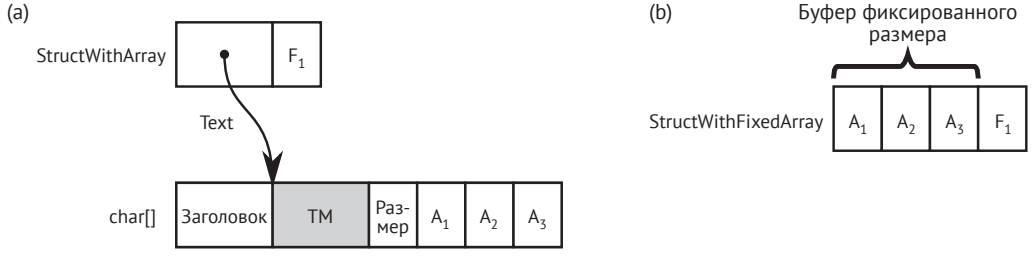


Рис. 13.6 ❖ Различие между двумя видами полей структуры: (a) массив, выделенный в куче; (b) буфер фиксированного размера

Буферы фиксированного размера чаще всего используются в контексте P/Invoke для определения структур маршалинга для Interop (листинг 13.66), которыми обычно представляются массивы символов или целых чисел (например, массив системных описателей).

Листинг 13.66 ❖ Примеры буферов фиксированного размера в репозитории CoreFX

```
public unsafe ref partial struct FileSystemEntry
{
   private const int FileNameBufferSize = 256;
   ...
   private fixed char _fileNameBuffer[FileNameBufferSize];
}

internal unsafe struct WIN32_FIND_DATA
{
   internal uint dwFileAttributes;
   ...
   private fixed char _cFileName[MAX_PATH];
   private fixed char _cAlternateFileName[14];

   internal ReadOnlySpan<char> cFileName
   {
    get { fixed (char* c = _cFileName) return new ReadOnlySpan<char>(c, MAX_PATH); }
   }
}
```

Но можно было бы использовать их и в неспециализированном коде для определения более плотных структур данных. Даже когда такие структуры размещаются в куче, будучи членами обобщенной коллекции, итоговый код обеспечивает лучшую локальность данных. В листинге 13.67 это иллюстрируется для случая обобщенной коллекции `List<T>`.

Листинг 13.67 ❖ Использование упакованных структур в качестве элементов `List<T>`

```
List<StructWithArray> list = new List<StructWithArray>();
List<StructWithFixedBuffer> list = new List<StructWithFixedBuffer>();
```

Получающаяся разница в локальности данных наглядно показана на рис. 13.7. Если в качестве упакованного поля структуры используется обычный размещенный в куче массив, то появляется много объектов, разбросанных по всей управляемой куче (но при этом получаем очевидное преимущество – элементы разных структур могут иметь разные размеры). С другой стороны, при использовании буферов фиксированного размера все элементы массива находятся внутри структуры (но при этом имеется очевидный недостаток – размер буфера в каждой структуре один и тот же). Второй подход обеспечивает гораздо более плотное размещение данных, что может быть полезно в ситуации, когда требуется высокая производительность, т. к. процессорный кеш используется эффективнее[1].

В случае данных, выделенных в стеке, аналогичных результатов можно достичь с помощью оператора `stackalloc`. Так что в описанной ситуации дело вкуса – выбирать ли буфер, выделенный с помощью `stackalloc`, или буфер фиксированного размера в структуре (при этом можно воспользоваться ссылочной структурой, чтобы гарантированно избежать упаковки).

---

[1] Вы можете возразить, что такой подход ничем не отличается от определения структуры с большим числом полей одного типа, – и будете правы. В данном случае так и есть, а различие заключается только в более удобном доступе к данным (по индексу).

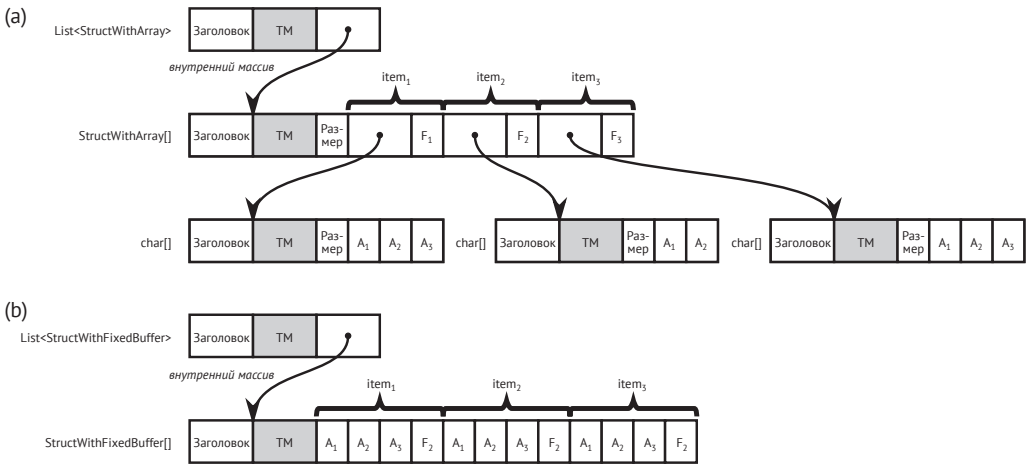


Рис. 13.7 ❖ Различие в локальности данных между списками List<T>, содержащими упакованные структуры двух видов: (a) обычный массив; (b) буфер фиксированного размера

В C# 7.3 добавлена возможность *индексирования перемещаемого фиксированного буфера*. Перемещаемый фиксированный буфер – это просто буфер фиксированного размера, который стал частью объекта, размещенного в куче (как в приведенном выше примере упаковки в обобщенном списке `List<T>`). Он называется «перемещаемым», потому что GC может перемещать его в процессе перемещения всего объекта на этапе уплотнения. Не будь этой возможности, мы должны были бы закрепить весь буфер, перед тем как обращаться к его элементам. Объясним на примере, воспользовавшись дополнительным классом, который оборачивает нашу структуру `StructWithFixedArray` (листинг 13.68).

Листинг 13.68 ❖ Класс, оборачивающий структуру с буфером фиксированного размера

```
public class StructWithFixedArrayWrapper
{
   public StructWithFixedArray Data = new StructWithFixedArray();
}
```

Доступ к буферу фиксированного размера по индексу в случае, когда структура не упакована, очевидно, безопасен, потому что структура, размещенная в стеке, не может перемещаться, так что закреплять вообще ничего не нужно (см. первый блок в листинге 13.69). Но попытка обратиться к буферу по индексу в случае упакованной структуры привела бы к ошибке компилятора «You cannot use fixed size buffers contained in unfixed expressions. Try using the fixed statement»[1]. Таким образом, до версии C# 7.3 необходимо было закреплять весь буфер (см. второй блок в листинге 13.69). Вы можете с полным основанием сказать, что здесь закрепление выглядит странным и ненужным, ведь индексирование определяется

[1] Не разрешается использовать буферы фиксированного размера в незакрепленных выражениях. Попробуйте воспользоваться оператором fixed. – *Прим. перев.*

относительно начала соответствующего поля, и перемещение всего объекта тут ничего не меняет. И в версии C# 7.3 это мелкое неудобство было устранено (см. третий блок в листинге 13.69).

Листинг 13.69 ❖ Изменение в индексировании буфера фиксированного размера в версии C# 7.3

```
static void Main(string[] args)
{
   // Блок 1 - доступ к фиксированному буферу, размещенному в стеке
   StructWithFixedBuffer s1 = new StructWithFixedBuffer();
   Console.WriteLine(s1.text[4]);

   // Блок 2 - доступ к перемещаемому буферу до C# 7.3
   StructWithArrayWrapper wrapper1 = new StructWithArrayWrapper();
   fixed (char* buffer = wrapper1.Data.Text)
   {
      Console.WriteLine(buffer[4]);
   }

   // Блок 3 - доступ к перемещаемому буферу после C# 7.3
   StructWithArrayWrapper wrapper2 = new StructWithArrayWrapper();
   Console.WriteLine(wrapper2.Data.text[4]);
}
```

> Интересно почитать комментарий по поводу этой возможности на сайте C# Language Design: «Одна из причин, по которой мы требуем закрепления целевого объекта, если он перемещаемый, – артефакт нашей стратегии генерации кода: мы всегда выполняем преобразование в неуправляемый указатель и потому требуем, чтобы пользователь закрепил объекты с помощью оператора fixed. Однако в случае индексирования преобразовывать в неуправляемый указатель необязательно. Такая же небезопасная арифметика указателей применима, когда получатель представлен управляемым указателем. Если мы это сделаем, то промежуточная ссылка будет управляемой (отслеживаемой сборщиком мусора), и закрепление необязательно».

И последнее замечание относительно работы с буферами фиксированного размера – имейте в виду, что их можно комбинировать с оператором `stackalloc` для создания размещаемых в стеке массивов элементов, содержащих другие «массивы» (буферы). Для поля, содержащего обычный массив, размещаемый в куче, это было бы невозможно – из-за ограничений, описанных в главе 6 (листинг 13.70).

Листинг 13.70 ❖ Комбинирование stackalloc с буферами фиксированного размера

```
var data = stackalloc StructWithArray[4]; // Ошибка компиляции: Cannot take the address of,
                                          // get the size of, or declare a pointer to
                                          // a managed type ('StructWithArray')[1]
var data = stackalloc StructWithFixedBuffer[4]; // допустимо
```

Размещение объектов и структур в памяти

Вас когда-нибудь интересовало, как размещаются в памяти экземпляры созданных вами классов или структур? Наверное, нет, и это хорошо. При работе с управ-

[1] Невозможно получить адрес, определить размер или объявить указатель на управляемый тип. – *Прим. перев.*

ляемым кодом нам должно быть безразлично, как организованы поля типа. CLR прекрасно справляется с правильным размещением в памяти полей типа. Так что изучение этого вопроса, скорее всего, следует назвать излишним усложнением. Но бывают исключения, когда хотелось бы не только знать, как размещены поля, но даже управлять этим размещением. Из широко известных случаев назовем передачу объектов неуправляемому коду, который ожидает, что поля расположены в памяти определенным образом (например, при вызове операционной системы). С другой стороны, редко, но встречаются ситуации, когда вам очень важно оптимальное использование памяти и эффективность доступа к ней, потому что размещения полей по умолчанию недостаточно.

Поскольку вся книга, и эта глава в особенности, посвящена таким граничным случаям, скажем несколько слов о размещении объектов в памяти. Да и вообще, один из девизов этой книги – знать, как вещи устроены, а не просто что они работают.

Мы уже знаем, что у экземпляров ссылочных типов есть заголовок и ссылка на таблицу методов и что расположены они в начале объекта. А у экземпляров типов значений нет ни того, ни другого, и содержат они только значения полей (рис. 4.17 и 4.18 в главе 4). И как же расположены поля?

Золотое правило, от которого в значительной степени зависит эффективность доступа к памяти и расположение полей, – это выравнивание данных (мы кратко упоминали об этом в главе 2). Для каждого примитивного типа данных (целые, числа с плавающей точкой разного формата и т. д.) определено предпочтительное выравнивание – какому числу должен быть кратен адрес (выраженный в байтах), по которому этот тип хранится. Чаще всего граница выравнивания равна размеру типа. То есть 4-байтовый тип `int32` выравнивается на границу, кратную 4 байтам, 8-байтовый тип `double` – на границу, кратную 8 байтам, и т. д. Для простейших однобайтовых типов `char` и `byte` граница выравнивания равна 1 байту – они считаются выровненными, где бы ни хранились. Современные процессоры особенно эффективно обращаются к выровненным данным. Доступ к невыровненным данным тоже возможен, но требует больше команд и, следовательно, медленнее.

Составные типы, содержащие поля примитивных типов, должны размещать их в памяти в соответствии с требованиями выравнивания. Для этого может потребоваться *заполнение (padding)* между полями – неиспользуемые байты, служащие исключительно для того, чтобы следующее поле начиналось с выровненного адреса (примеры будут приведены ниже). Кроме того, сложные типы и сами должны быть выровнены – чтобы в случае, когда они являются частями еще более сложного типа (или массива), поля по-прежнему оставались выровненными.

В результате мы получаем три правила размещения объекта в памяти, сформулированных в MSDN:

- граница выравнивания типа равна меньшей из двух величин: размеру его наибольшего элемента (1, 2, 4, 8 и т. д. байт) или заданному размеру упаковки[1];
- каждое поле должно быть выровнено на границу, кратную меньшей из двух величин: его собственный размер (1, 2, 4, 8 и т. д. байт) или выравнивание

[1] Упаковка (packing, не путать с boxing) будет описана позже, но к автоматическому размещению полей она отношения не имеет.

всего типа. Поскольку по умолчанию тип выравнивается на границу, кратную размеру его наибольшего элемента, который больше или равен размеру всех прочих полей, обычно это означает, что поля выравниваются на границу, кратную их размеру. Так, даже если наибольшее поле типа – 64-разрядное (8-байтовое) целое или размер упаковки равен 8, то поля типа `Byte` выравниваются на однобайтовую границу, поля типа `Int16` – на 2-байтовую, а поля типа `Int32` – на 4-байтовую;

- для удовлетворения требований выравнивания между полями вставляются байты заполнения.

Памятуя о золотом правиле выравнивания и следующих из него трех правилах, мы также должны знать о проектных решениях, касающихся размещения полей в типах обоих видов:

- структуры – по умолчанию поля размещаются в памяти *последовательно*, т. е. в том порядке, в каком определены. Так сделано главным образом потому, что предполагается, что структура будет передана неуправляемому коду, и порядок полей выбран не случайно. Когда платформа .NET только проектировалась, ожидалось, что структуры будут использоваться в основном в технологиях Interop, поэтому такое поведение по умолчанию казалось разумным. Однако это так только для «неуправляемых типов», как мы уже говорили в главе 6 (мы вскоре снова встретимся с ними в контексте нового ограничения). Но хотя порядок полей явно определен, их размещение в памяти все равно подчиняется требованиям выравнивания. Поэтому может быть добавлено заполнение, и окончательный размер структуры вырастет (это плата за эффективный доступ к выровненным полям);
- классы – по умолчанию поля размещаются в памяти *автоматически*, порядок полей может быть изменен. Поскольку единственным владельцем таких данных является CLR, она и решает, как их разместить. Поля переупорядочиваются так, чтобы обеспечить оптимальное потребление памяти и эффективный доступ (с учетом выравнивания) с точки зрения процессора.

В наши дни, когда типы значений все чаще применяются в неспециализированном коде, принятое по умолчанию последовательное размещение полей структуры уже необязательно является оптимальным, поэтому полезно знать об имеющихся альтернативах.

Посмотрим, как все это выглядит в действии. Для простой структуры в листинге 13.7 размещение полей показано на рис. 13.8а – все три поля располагаются в памяти в порядке объявления. Но из-за требований к выравниванию поля этой структуры начинаются по следующим адресам:

- смещение 0 байт – у первого поля тип `byte`, его граница выравнивания кратна 1 байту, так что оно может размещаться по любому адресу;
- смещение 8 байт – у второго поля тип `double`, его граница выравнивания кратна 8 байтам, так что оно может размещаться по любому адресу, кратному 8. К сожалению, для этого необходимо заполнение длиной 7 байт, которые потрачены зря;
- смещение 16 байт – последним полем является целое, его граница выравнивания кратна 4 байтам, так что оно может начинаться по адресу 16.

Наконец, выравнивание структуры в целом должно быть равно размеру наибольшего элемента – в нашем случае 8 байтам. Иными словами, размер структуры

должен быть кратен 8. Она уже занимает 20 байт, так что придется округлить до 24 байтов, добавив заполнение в конце.

Благодаря выравниванию структуры в целом поля ее экземпляров всегда будут выровнены, например если экземпляры являются элементами массива (рис. 13.8b). Если бы структура была выровнена некорректно (т. е. в конец не были бы добавлены байты заполнения), то в такой ситуации мы получили бы невыровненные данные (рис. 13.8c).

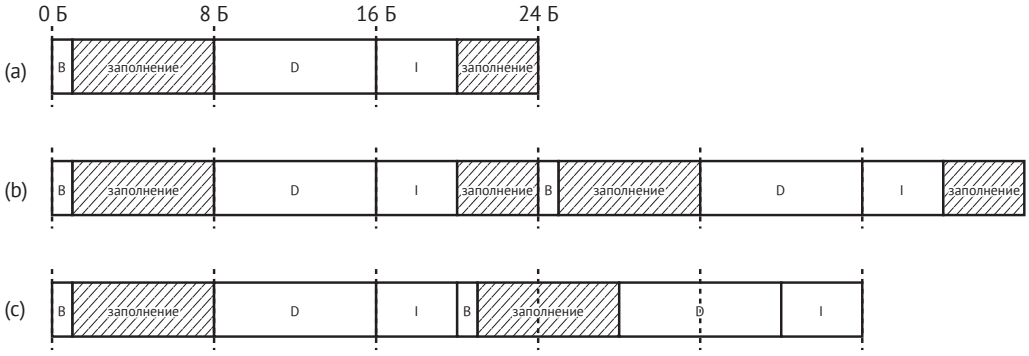


Рис. 13.8 ❖ Размещение полей структуры по умолчанию:
(a) размещение структуры из листинга 13.71;
(b) пример использования структуры AlignedDouble в качестве элемента массива;
(c) пример неправильного использования структуры AlignedDouble
(когда структура в целом выровнена некорректно)

Как видим, последовательное размещение полей структуры в данном случае приводит к большому перерасходу памяти – не используется 11 байт, почти половина всей структуры! Если эта структура используется от случая к случаю, то, вероятно, проблем не будет. Но если программа сильно зависит от типов значений и обрабатывает их миллионами, то могут возникнуть проблемы.

Листинг 13.71 ❖ Пример простой структуры (для изучения размещения полей)

```
public struct AlignedDouble
{
   public byte B;
   public double D;
   public int I;
}
```

.NET предлагает способ, позволяющий контролировать размещение полей. Хотя и он тоже предназначался для сценариев с применением Interop, его можно использовать и в общем случае, чтобы определить размещение в памяти, лучше отвечающее нашим потребностям. Размещением полей управляет атрибут `StructLayout`, который, несмотря на свое название, применим и к структурам, и к классам и может принимать три значения:

- `LayoutKind.Sequential` – уже описанное размещение, при котором гарантируется правильное выравнивание полей, а сами поля хранятся в порядке

определения. Это значение подразумевается по умолчанию для неуправляемых структур (как было описано в главе 6 и скоро мы повторим еще раз);

- `LayoutKind.Auto` – размещение, при котором гарантируется правильное выравнивание полей, но сами поля могут быть переупорядочены (чтобы память использовалась более эффективно). Подразумевается по умолчанию для классов и управляемых структур;
- `LayoutKind.Explicit` – при таком размещении ничего не гарантируется, потому что мы задаем его явно.

Структуру из листинга 13.7 (у которой атрибут по умолчанию `LayoutKind.Sequential`) легко можно изменить, так чтобы использовалось автоматическое размещение (листинг 13.72). Как видно по рис. 13.9, при таком размещении память используется гораздо эффективнее, потому что добавлено только три байта заполнения (при этом все поля правильно выровнены).

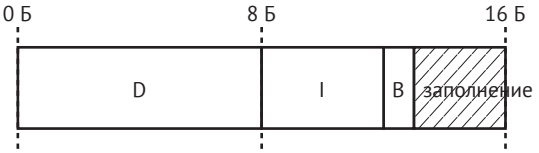


Рис. 13.9 ❖ Автоматическое размещение полей для структуры в листинге 13.72

Листинг 13.72 ❖ Пример простой структуры (для изучения автоматического размещения полей)

```
[StructLayout(LayoutKind.Auto)]
public struct AlignedDoubleAuto
{
   public byte B;
   public double D;
   public int I;
}
```

Основной недостаток автоматического размещения – невозможность использовать такую структуру для Interop. Однако я вижу такие структуры в высокопроизводительном неспециализированном коде, где это ограничение несущественно. Если мы используем типы значений по причине тех преимуществ в плане управления памятью, которые они дают (выделение памяти в стеке, локальность данных, меньшее потребление памяти), то, скорее всего, выберем автоматическое размещение, а не подразумеваемое по умолчанию!

Чем больше полей и чем сильнее они различаются по размеру, тем менее оптимальным оказывается последовательное размещение. В качестве упражнения предлагаю разобраться, почему структура в листинге 13.73 занимает:

- 64 байта при размещении `LayoutKind.Sequential` (когда на заполнение тратится 28 байт);
- 40 байт при размещении `LayoutKind.Auto` (когда впустую расходуется только 4 байта).

Листинг 13.73 ❖ Пример структуры, в которой размещение сильно влияет на размер

```
public struct ManyDoubles
{
   public byte B1;
   public double D1;
   public byte B2;
   public double D2;
   public byte B3;
   public double D3;
   public byte B4;
   public double D5;
}
```

Структуры, представленные до сих пор, были примерами неуправляемых типов. Напомню, что неуправляемым называется тип, который не является ссылочным и не содержит полей ссылочного типа. Однако мы, безусловно, можем создать структуры, не являющиеся неуправляемыми, – достаточно добавить всего одно поле ссылочного типа (листинг 13.74). Как было сказано выше, при этом размещение по умолчанию становится автоматическим, как для ссылочных типов. Как видно по рис. 13.10, поля структуры `AlignedDoubleWithReference` действительно переупорядочены, как в режиме `LayoutKind.Auto`.

Листинг 13.74 ❖ Пример неуправляемой структуры

```
public struct AlignedDoubleWithReference
{
   public byte B;
   public double D;
   public int I;
   public object O;
}
```

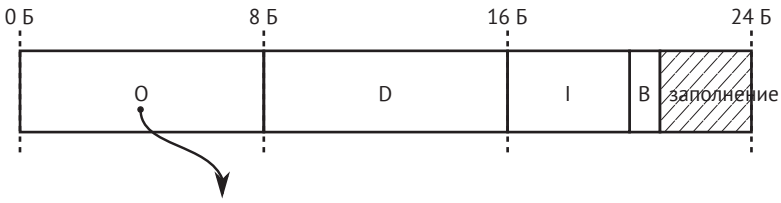


Рис. 13.10 ❖ Подразумеваемое по умолчанию размещение полей для структуры в листинге 13.74

Для управляемых структур поведение по умолчанию изменяется, потому что их не разрешается передавать с помощью механизма P/Invoke, т. к. они могут содержать ссылку на управляемый объект, которая может измениться во время сборки мусора. Поскольку использование таких структур в неуправляемом коде запрещено, для них автоматическое размещение полей безопасно.

> Заметим, что при автоматическом размещении ссылки на объекты помещаются в начало структуры. И вы, наверное, догадались, почему. Это полезно на этапе пометки, поскольку благодаря лучшему использованию строк кеша обеспечивается более эффективный обход графа объектов. Большинство ссылок на объекты попадет в ту же строку кеша, что уже прочитанное поле MT.

Размещение по умолчанию изменяется на автоматическое и тогда, когда структура содержит другие структуры с размещением `LayoutKind.Auto`. Большинство часто используемых встроенных структур (`Decimal`, `Guid`, `Char`, `Boolean`) последовательные, поэтому их использование не приведет к изменению размещения. Но, как ни странно, для структуры `DateTime` размещение автоматическое, поэтому если она используется в качестве поля другой структуры, размещение последней также станет автоматическим (листинг 13.75).

Листинг 13.75 ❖ Различные типы полей и их влияние на размещение

```
public struct StructWithFields
{
   public byte B;
   public double D;
   public int I;
   //public SomeEnum E;      // все еще последовательное
   //public SomeStruct AD;   // все еще последовательное
   //public unsafe void* P;  // все еще последовательное
   //public decimal DE;      // все еще последовательное
   //public Guid G;          // все еще последовательное
   //public char C;          // все еще последовательное
   //public Boolean BL;      // все еще последовательное
   //public object O;        // а теперь автоматическое
   //public DateTime DT;     // автоматическое, потому что для DateTime
                             // размещение автоматическое
}
```

Если вы действительно задумываетесь об оптимальном использовании памяти (а, наверное, так и есть, раз вы решили прибегнуть к структурам), то размещение структур в памяти не должно быть вам безразлично. Подумайте только о драгоценных байтах стека, которые впустую расходуются на заполнение в массиве, созданном оператором `stackallock`! Но потребление памяти – не единственная проблема, иногда о размещении структуры приходится думать в связи с использованием кеша (мы еще обсудим этот вопрос в разделе следующей главы, посвященном проектированию, ориентированному на данные).

> Имейте в виду, что автоматическое размещение для классов и управляемых структур нельзя изменить: явно заданный атрибут `LayoutKind.Sequential` игнорируется.

Еще пока не описанное «явное размещение» особенно полезно при вызове через P/Invoke, поскольку дает полный контроль над размещением структуры в памяти (листинг 13.76). Вы можете создать именно такую структуру, которую ожидает неуправляемый код. Очевидно, следует помнить, что при таком полном контроле забота о выравнивании возлагается на вас, а это значит, что легко могут появиться невыровненные поля (рис. 13.11).

Для P/Invoke это не очень существенно, но будьте осторожны, проектируя структуру для использования в высокопроизводительном коде[1].

Листинг 13.76 ❖ Пример простой структуры (для изучения явного размещения полей)

```
[StructLayout(LayoutKind.Explicit)]
public struct UnalignedDouble
{
   [FieldOffset(0)]
   public byte B;
   [FieldOffset(1)]
   public double D;
   [FieldOffset(9)]
   public int I;
}
```

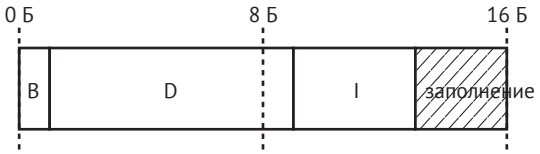


Рис. 13.11 ❖ Явное размещение полей для структуры в листинге 13.76

В частности, компилятор не требует, чтобы явно размещенные поля не перекрывались. Поэтому нужно осторожно задавать смещения, чтобы поля не накладывались друг на друга. Правда, есть ситуация, когда именно это и нужно, – при создании *размеченных объединений (discriminated unions)*. Так называется тип, способный представлять различные наборы данных. Чтобы его создать, мы должны воспользоваться явным размещением и присвоить разным полям одинаковые смещения (листинг 13.77).

Листинг 13.77 ❖ Пример размеченного объединения

```
[StructLayout(LayoutKind.Explicit)]
public struct DiscriminatedUnion
{
   [FieldOffset(0)]
   public bool Bool;
   [FieldOffset(0)]
   public byte Byte;
   [FieldOffset(0)]
   public int Integer;
}
```

---

[1] Честно говоря, выполненные мной тесты производительности не показали значимого изменения производительности при доступе к полю типа `double` в структурах `AlignedDouble` и `UnalignedDouble`. Похоже, что команды набора Intel® Advanced Vector Extensions (Intel® AVX), используемые в моем процессоре Intel, прекрасно справляются с невыровненным доступом к `double`. Но это всего лишь детали реализации, а общая рекомендация – выравнивать поля на границу памяти.

Разумеется, программист должен позаботиться о том, чтобы читать тот тип, который был записан, если только мы не хотим применить эту технику для преобразования типов. Можно представить себе использование буфера фиксированного размера для доступа к одной и той же области памяти через другие типы данных (листинг 13.78).

Листинг 13.78 ❖ Пример размеченного объединения с буфером фиксированного размера

```
[StructLayout(LayoutKind.Explicit)]
public struct DiscriminatedUnion
{
   [FieldOffset(0)]
   public bool Bool;
   [FieldOffset(0)]
   public byte Byte;
   [FieldOffset(0)]
   public int Integer;
   [FieldOffset(0)]
   Public fixed byte Buffer[8];
}
```

Существует еще одна форма контроля размещения объекта в памяти – упаковка (packing). Поле `Pack` атрибута `StructLayout` управляет выравниванием полей типа. Например, можно задать `Pack` равным 1 байту:

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct AlignedDouble
{
  public byte B;
  public double D;
  public int I;
}
```

Как тогда будет выглядеть размещение? Напомним первое правило из MSDN: «граница выравнивания типа равна меньшей из двух величин: размеру его наибольшего элемента (1, 2, 4, 8 и т. д. байт) или заданному размеру упаковки». Поэтому в нашем случае тип выравнивается не на границу, кратную 8 байтам (размер `double`), а просто на 1 байт. Второе правило гласит: «каждое поле должно быть выровнено на границу, кратную меньшей из двух величин: его собственный размер (1, 2, 4, 8 и т. д. байт) или выравнивание всего типа». Таким образом, каждое поле тоже выравнивается на границу, кратную 1 байту. В результате мы получаем очень плотное размещение в 13 байтах без какого-либо заполнения (но выравнивание полей при этом не оптимально).

Для изучения того, как размещен в памяти ваш тип, есть два прекрасных бесплатных инструмента. Первый – библиотека `ObjectLayoutInspector` (есть на GitHub и в виде NuGet-пакета), написанная Сергеем Тепляковым и специально предназначенная для изучения размещения объектов в памяти. Она предлагает очень удобный способ анализа типов с помощью вызова одного-единственного метода (листинг 13.79). Результаты представлены в виде таблицы (листинг 13.80).

Листинг 13.79 ❖ Печать размещения структур из листингов 13.71 и 13.74 с помощью ObjectLayoutInspector

```
static void Main(string[] args)
{
   TypeLayout.PrintLayout<AlignedDouble>();
   TypeLayout.PrintLayout<AlignedDoubleWithReference>();
}
```

Листинг 13.80 ❖ Результат работы программы в листинге 13.79

```
Type layout for 'AlignedDouble'
Size: 24 bytes. Paddings: 11 bytes (%45 of empty space)
|==========================|
|     0: Byte B (1 byte)   |
|--------------------------|
|   1-7: padding (7 bytes) |
|--------------------------|
|  8-15: Double D (8 bytes)|
|--------------------------|
| 16-19: Int32 I (4 bytes) |
|--------------------------|
| 20-23: padding (4 bytes) |
|==========================|

Type layout for 'AlignedDoubleWithReference'
Size: 24 bytes. Paddings: 3 bytes (%12 of empty space)
|==========================|
|   0-7: Object O (8 bytes)|
|--------------------------|
|  8-15: Double D (8 bytes)|
|--------------------------|
| 16-19: Int32 I (4 bytes) |
|--------------------------|
|    20: Byte B (1 byte)   |
|--------------------------|
| 21-23: padding (3 bytes) |
|==========================|
```

Если вы не хотите использовать консольное приложение, то можете получить данные от библиотеки и интерпретировать результат анализа размещения самостоятельно (листинг 13.81).

Листинг 13.81 ❖ Использование ObjectLayoutInspector для ручного анализа размещения структур в памяти

```
static void Main(string[] args)
{
   TypeLayout layout = TypeLayout.GetLayout<AlignedDouble>();
   Console.WriteLine($"Total size {layout.FullSize}B with {layout.Paddings} B padding.");
   foreach (var fieldBase in layout.Fields)
   {
      switch (fieldBase)
      {
         case FieldLayout field: Console.WriteLine($"{field.Offset}
```

```
            {field.Size} {field.FieldInfo.Name}"); break;
          case Padding padding: Console.WriteLine($"{padding.Offset}
            {padding.Size} Padding"); break;
        }
    }
}
```

Такой инструмент с большей вероятностью будет использоваться на этапе сборки или во время разработки, чем во время выполнения готового приложения.

Второй инструмент – веб-страница https://sharplab.io, которая предлагает замечательные средства анализа кода для .NET. В частности, статические методы `Inspect.Heap` и `Inspect.Stack`, которые печатают размещение заданных типов в памяти (листинг 13.82 и рис. 13.12).

Листинг 13.82 ❖ Пример программы для изучения размещения объектов в памяти с помощью Sharplab.io

```
using System;
using System.Runtime.InteropServices;

public class C {
   public static void Main() {
      var o = new AlignedDouble();
      Inspect.Heap(new AlignedDouble());
      Inspect.Stack(in o);
   }
}
```

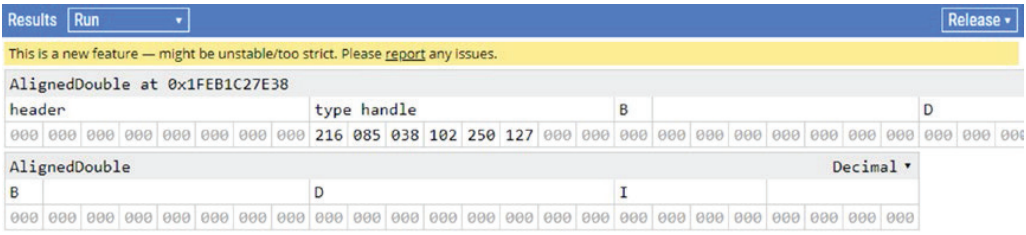


Рис. 13.12 ❖ Результат работы программы из листинга 13.82

Надеюсь, что при наличии двух таких великолепных инструментов вам не понадобится использовать низкоуровневые средства вроде WinDbg для анализа объектов вручную. Но если вы все-таки захотите это сделать, то рекомендую команды SOS `!dumpobject` (для классов) и `!dumpvc` (для типов значений) (листинг 13.83).

Листинг 13.83 ❖ Изучение размещения объекта в памяти с помощью команды SOS dumpvc в WinDbg

```
> !dumpvc 00007ffda2725e18 00007ffda2725e18
Name:        CoreCLR.ObjectLayout.AlignedDouble
MethodTable: 00007ffda2725e18
EEClass:     00007ffda2872110
Size:        40(0x28) bytes
File: (...)\CoreCLR.ObjectLayout.dll
```

```
Fields:
              MT    Field  Offset        Type VT        Attr      Value  Name
00007ffdfd6a8b60  4000001       0    System.Byte  1 instance          0     B
00007ffdfd6b0858  4000002       8  System.Double  1 instance   0.000000     D
00007ffdfd6c66d8  4000003      10   System.Int32  1 instance  -43316160     I
```

Ограничение unmanaged

Неуправляемые типы уже упоминались и в главе 6, когда мы говорили о том, какие типы можно использовать с оператором `stackalloc`, и в этой главе в контексте неуправляемых структур. В версии C# 7.3 появилось новое ограничение – `unmanaged`. Оно позволяет писать универсальный код, оперирующий неуправляемыми типами и указателями на них.

Напомним определение, данное в MSDN: «Неуправляемым называется тип, который не является ссылочным и не содержит полей ссылочных типов ни на каком уровне вложенности». В формулировке ограничений на `stackalloc` эта мысль выражена более точно: «неуправляемый тип может содержать только примитивные типы, перечисления (enum), типы указателей и пользовательские структуры, удовлетворяющие тем же критериям»[1].

В листинге 13.84 приведен пример двух структур, из которых только первая удовлетворяет ограничениям на неуправляемый тип. Напомним, что проверяются все уровни вложенности, т. е. если структура A содержит структуру B, которая содержит структуру C, содержащую структуру D, в которой есть поле ссылочного типа, то вся структура A не считается неуправляемой.

Листинг 13.84 ❖ Пример неуправляемого и управляемого типов

```
public struct UnmanagedStruct
{
   public int Field;
}

public struct NonUnmanagedStruct
{
   public int Field;
   public object O;
}
```

При помощи нового ограничения `unmanaged` на параметр обобщенного типа компилятор проверяет выполнение условий неуправляемости. Если они не выполняются, выдается соответствующее сообщение об ошибке компиляции. Это ограничение можно использовать как в обобщенных методах (листинг 13.85), так и в обобщенных типах структур (листинг 13.86).

Листинг 13.85 ❖ Пример использования ограничения unmanaged в методе

```
public static void UnamanagedContraint<T>(T arg) where T : unmanaged
{
}
```

[1] Строгое определение неуправляемого типа приведено в спецификации языка C# ECMA-334, в параграфе 23.3 «Типы указателей».

```
static void Main(string[] args)
{
   UnamanagedContraint(new UnmanagedStruct());
   UnamanagedContraint(new NonUnmanagedStruct()); // Ошибка компиляции: The type
   // 'NonUnmanagedStruct' должен быть не допускающим null типом значений, равно как
   // и все его поля на любом уровне вложенности, только тогда он может выступать
   // в роли параметра 'T' в обобщенном типе или в методе
   // 'Constraints.UnamanagedContraint<T>(T)'
}
```

Листинг 13.86 ❖ Пример использования ограничения unmanaged в типе

```
public struct UnmanagedStruct<T> where T : unmanaged
{
   ...
}

static void Main(string[] args)
{
   var obj = new UnmanagedGenericStruct<object>(); // Ошибка компиляции:
   // Тип 'object' должен быть не допускающим null типом значений, равно как
   // и все его поля на любом уровне вложенности, только тогда он может выступать
   // в роли параметра 'T' в обобщенном типе или в методе
   // 'UnmanagedGenericStruct<T>'
}
```

Что нам дает ограничение `unmanaged`? Благодаря ему становятся возможны следующие вещи:

- можно использовать указатель на `T` – если тип `T` удовлетворяет ограничению `unmanaged`, то можно использовать также указатель `T*` (преобразование в `void*` тоже возможно);
- допустимо выражение `sizeof(T)`;
- к `T` можно применить оператор `stackalloc`.

Без ограничения `unmanaged` все эти операции были бы недопустимы и приводили бы к ошибке компиляции «Cannot take the address of, get the size of, or declare a pointer to a managed type (‘T’)»[1], даже если на `T` наложено ограничение `struct`. Очевидно, что эти операции требуют небезопасного контекста, но это остается справедливым и при наличии ограничения `unmanaged` (которое само по себе небезопасного контекста не требует).

Конечно, все эти операции низкоуровневые и чаще всего используются, когда требуется низкоуровневое управление памятью, например для быстрой сериализации данных. Вряд ли вам придется столкнуться с ограничением `unmanaged` в коде обычного бизнес-приложения!

В листинге 13.87 приведен пример метода, в котором используются операции, ставшие допустимыми благодаря ограничению `unmanaged`. Отметим интересный факт – в листинге 13.87 можно получить указатель на аргумент без закрепления. Это возможно, потому что из ограничения `unmanaged` вытекает, что `T` – тип значений, т. е. он передается по значению. А раз так, то брать адрес безопасно (поскольку значение не находится в куче).

---

[1] Невозможно получить адрес, получить размер или объявить указатель на управляемый тип (‘T’). – *Прим. перев.*

Листинг 13.87 ❖ Простой пример использования ограничения unmanaged

```
unsafe public static int UseUnmanagedConstraint<T>(T arg) where T : unmanaged
{
   T* ptr = &arg;            // используем указатель T*
   T* sa = stackalloc T[16]; // используем stackalloc
   return sizeof(T);         // используем sizeof
}
```

Похожий код работал бы и без ограничения `unmanaged` для простой структуры (листинг 13.88).

Листинг 13.88 ❖ Использование обычной структуры в коде, похожем на листинг 13.87

```
unsafe static public void UseUnmanagedConstraint2(SomeStruct obj)
{
   SomeStruct* p = &obj;
   ...
}
```

Однако если мы передаем по ссылке объект с ограничением `unmanaged`, то должны явно закрепить его, потому что он может оказаться в куче, например вследствие упаковки (листинг 13.89).

Листинг 13.89 ❖ Пример использования ограничения unmanaged для объекта, передаваемого по ссылке

```
unsafe public int UseUnmanagedRefConstraint<T>(ref T arg) where T : unmanaged
{
   fixed (T* ptr = &arg)
   {
      Console.WriteLine((long) ptr);
      return sizeof(T);
   }
}
```

По той же причине необходимо явно закреплять поля структуры, когда они используются в ее методах экземпляра (листинг 13.90), поскольку метод может быть вызван из метода упакованного экземпляра структуры[1].

Листинг 13.90 ❖ Пример использования ограничения unmanaged в методе структуры

```
public struct StructWithUnmanagedField<T> where T : unmanaged
{
   private T field;
   unsafe public void Use()
   {
      fixed (T* ptr = &field)
      {
```

[1] В версии C# 7.3 замена `StructWithUnmanagedField` ссылочной структурой не изменяет этого поведения, хотя и могла бы, поскольку внутри метода `Use` гарантируется, что поле размещено в стеке.

```
        // ...
      }
   }
}
```

Где на практике применяется ограничение `unmanaged`? Оно предназначено для работы с обобщенными типами в случаях, когда без этого пришлось бы писать много конкретных реализаций. Отличный пример – это различные виды сериализации. Так, наличие `sizeof(T)` позволяет написать обобщенную сериализацию «в массив байтов» (листинг 13.91).

Листинг 13.91 ❖ Пример обобщенной сериализации (взят из MSDN)

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T : unmanaged
{
   var size = sizeof(T);
   var result = new Byte[size];
   Byte* p = (byte*)&argument;
   for (var i = 0; i < size; i++)
      result[i] = *p++;
   return result;
}
```

Можно также представить себе универсальный механизм протоколирования, когда переданный аргумент используется в низкоуровневом коде, как показано в листинге 13.92. Здесь в стеке создана вспомогательная структура, содержащая описание протоколируемого значения (его адрес и размер). Эта структура затем передается основной функции протоколирования. Чтобы такой метод был полезен, необходимо по крайней мере два или три перегруженных варианта, принимающих соответственно два или три аргумента (и в стеке придется создавать массивы большего размера).

Листинг 13.92 ❖ Пример универсального низкоуровневого протоколирования (по образцу протоколирования ETW)

```
public unsafe void LogData<T>(T arg) where T : unmanaged
{
   if (IsEnabled())
   {
      EventData* data = stackalloc EventData[1];
      data[0].DataPointer = (IntPtr)(&arg);
      data[0].Size = sizeof(T);
      WriteEventCore(data);
   }
}
```

Ограничение `unmanaged` также полезно при создании типов, потребляющих неуправляемую память (особенно коллекций). В листинге 13.93 приведен очень простой пример такого типа. Без ограничения `unmanaged` было бы невозможно создать такой обобщенный тип, поскольку был бы недоступен оператор `sizeof` (размер элемента пришлось бы указывать в конструкторе). Но еще важнее, что благодаря ограничению `unmanaged` мы можем использовать указатель `T*`, что делает индексирование тривиальным и позволяет возвращать `T` по ссылке (без этого

ограничения мы были бы вынуждены использовать тип void* и некрасивое приведение указателей, чтобы реализовать методы чтения и установки индексатора).

Листинг 13.93 ❖ Пример типа, скрывающего работу с неуправляемой памятью

```
public unsafe class UnmanagedArray<T> : IDisposable where T : unmanaged
{
   private T* data;
   public UnmanagedArray(int length)
   {
      data = (T*)Marshal.AllocHGlobal(length * sizeof(T));
   }

   public ref T this[int index]
   {
      get { return ref data[index]; }
   }

   public void Dispose()
   {
      Marshal.FreeHGlobal((IntPtr)data);
   }
}

static void Main(string[] args)
{
   using (UnmanagedArray<int> array = new UnmanagedArray<int>(20))
   {
      array[10] = 10;
      for (int i = 0; i < 20; i++)
         Console.WriteLine(array[i]); // печатает мусор, но значение 10 для 10-го элемента
   }
}
```

Непреобразуемые типы

Помимо неуправляемых, существуют также *непреобразуемые (blittable) типы*, представление которых в памяти одинаково для управляемого и неуправляемого кодов. Такие типы чаще всего встречаются в контексте маршалинга Interop и не требуют никаких преобразований при вызовах посредством P/Invoke.

Неуправляемые и непреобразуемые типы – почти одно и то же, но последние определены несколько строже. Это связано с тем, что некоторые типы значений только «иногда могут быть непреобразуемыми», поскольку в некоторых случаях их представления на управляемой и неуправляемой сторонах различаются:

- `decimal` – двоичное представление не вполне стабилизировалось, поэтому нельзя быть до конца уверенным, что на неуправляемой стороне применяется тот же формат;
- `bool` – обычно занимает 1 байт в обоих случаях, но в неуправляемом контексте иногда больше (например, в языке C может занимать 4 байта);
- `char` – обычно занимает 2 байта, но в неуправляемом контексте иногда больше (зависит от кодировки);

- `DateTime` – исторически сложилось, что это структура с автоматическим размещением в памяти, что делает этот тип непреобразуемым;
- `Guid` – внутреннее представление зависит от порядка байтов на конкретной машине.

Таким образом, структура, содержащая поле одного из этих специальных типов, является неуправляемым типом (и будет удовлетворять ограничению `unmanaged`), но не является непреобразуемым в смысле маршалинга Interop.

Усложним ситуацию еще немного: лишь непреобразуемые типы можно закреплять методом `GCHandle.Alloc` (поскольку предполагается, что закрепление нужно для последующего вызова `AddrOfPinnedObject` и передачи адреса всего объекта неуправляемому коду). Следовательно, ограничения `unmanaged` недостаточно, чтобы гарантировать успешность закрепления (листинг 13.94). Структура `WeirdStruct` не является непреобразуемой, потому что содержит поля, которые не являются непреобразуемыми (на самом деле все ее поля такие). Однако это все же неуправляемый тип (поскольку никакие требования к неуправляемому типу не нарушены). Следовательно, к нему применимо ограничение `unmanaged` в методе `UseUnmanagedConstraint`, но при попытке закрепить его методом `GCHandle.Alloc` будет вызвано исключение `ArgumentException`.

Листинг 13.94 ❖ Различие между непреобразуемым и неуправляемым типами при закреплении методом `GCHandle.Alloc`

```
public struct WeirdStruct
{
   public decimal DE;
   public DateTime DT;
   public Guid G;
   public char C;
   public Boolean BL;
}

unsafe public static int UseUnmanagedConstraint<T>(T obj) where T : unmanaged
{
   var handle = GCHandle.Alloc(obj, GCHandleType.Pinned); // исключение System.
   // ArgumentException: Object contains non-primitive or non-blittable data.
   ...
}

static void Main(string[] args)
{
   var s = new WeirdStruct();
   UseUnmanagedConstraint(s);
}
```

Подводя итоги, можно сказать, что:

- неуправляемые типы (и ограничение `unmanaged`) используются в неспециализированном коде для низкоуровневого управления памятью в таких алгоритмах, как сериализация и десериализация, хеширование и т. д. Поскольку в этом контексте они применяются для решения очень общих задач, то были описаны довольно подробно. А так как они работают с памятью

на низком уровне, то чаще всего встречаются в небезопасном контексте, хотя само по себе ограничение `unmanaged` этого не требует;

- непреобразумые типы используются для маршалинга Interop. Поскольку в этой книге Interop подробно не рассматривается, они лишь кратко упомянуты. Единственный аспект, представляющий для нас интерес, – тот факт, что с помощью метода `GCHandle.Alloc` можно закреплять только непреобразумые типы.

Чтобы окончательно вас запутать, стоит сказать, что тип `decimal` – особый случай: он не является непреобразуемым, но содержащие его структуры все равно можно закреплять методом `GCHandle.Alloc`.

Резюме

В этой главе мы коснулись множества интересных и по большей части низкоуровневых вещей. Начав с подробного рассмотрения статических полей потока, мы перешли к управляемым указателям, которые помогают понять механизм передачи по ссылке в .NET. Кроме того, в настоящее время они особенно полезны в связи с растущей популярностью структур.

Вообще, большая часть этой главы так или иначе касается типов значений: ссылочных структур, byref-подобных типов, полей byref-подобных типов и т. д. Мы подробно описали управляемые указатели, поскольку они лежат в основе всех этих механизмов. В наши дни, когда к производительности относятся серьезно и стараются по возможности избавиться от выделения памяти в куче, эти вопросы выходят на первый план в экосистеме .NET. Конечно, эти знания вряд ли понадобятся при написании обычных бизнес-приложений. Но ведь и эта глава посвящена программированию иного рода, поэтому неудивительно, что мы уделили этим типам так много внимания.

Далее мы рассказали о размещении управляемых типов в памяти – вопросе, не столь очевидном, как могло бы показаться. И завершили главу описанием ограничения `unmanaged` на обобщенные типы, которое было добавлено в C# недавно (а заодно затронули тему непреобразуемых типов).

Все изложенное полезно само по себе, но, кроме того, закладывает прочный фундамент для тем, освещаемых в следующей главе, – особенно это относится к использованию и реализации типа `Span<T>`.

Глава 14

Продвинутые приемы

Эта глава в некотором роде является продолжением предыдущей, в ней описываются более продвинутые приемы, используемые в .NET. Поэтому имейте в виду, что знания, полученные в предыдущей главе, окажутся очень полезны для понимания этой (особенно помогут знания о ссылочных типах, возврате по ссылке и ссылочных структурах).

Эта глава отражает современные тенденции в программировании для .NET (по крайней мере, те, которые касаются кода с повышенными требованиями к производительности), – оптимизированное использование процессора и памяти с целью сделать библиотеки и приложения как можно быстрее. Мне это кажется очень интересным. Все больше библиотек переходят на использование эффективного типа `Span<T>` и (или) конвейеров. Надеюсь, что эта глава поможет вам найти свое место в мире современного .NET. И кстати говоря, последний ее раздел посвящен планируемым, но еще не выпущенным (или выпущенным только для предварительного ознакомления) средствам .NET.

Span<T> и Memory<T>

В C# непрерывную область памяти можно выделить разными способами: обычный массив в куче, буфер фиксированного размера (fixed buffer), `stackalloc` или из неуправляемой памяти. Было бы очень удобно иметь единый интерфейс для всех этих случаев, который оставался бы эффективным (как обычный массив). Более того, часто необходимо «вырезать» (slice) кусок такой памяти для обработки различными методами. И в идеале хорошо бы, чтобы все это делалось без участия главного врага производительности в .NET – выделения памяти в куче. Именно благодаря всем этим пожеланиям на свет появился тип `Span<T>`.

> Имейте в виду, что далее в этой главе используется несколько упрощенное разделение стека и кучи. Напомним, что в главе 4 было сказано, что выделение чего-то в стеке или в куче – в значительной степени детали реализации, продиктованные ожидаемыми характеристиками времени жизни данных. Но каждый раз повторять в последующем тексте, что стек или куча – деталь реализации, было бы скучно и утомительно. А поскольку `Span<T>` и `Memory<T>` – в какой-то степени «протекающие абстракции», то это было бы еще менее оправдано.

Span<T>

Обобщенный тип Span<T> был введен в .NET Core 2.1. Это тип значений (ссылочная структура), поэтому сам по себе не влечет никакого выделения памяти. В нем определен индексатор, возвращающий значение по ссылке, поэтому его можно использовать как массив. Кроме того, он спроектирован так, чтобы можно было эффективно использовать поддиапазоны. Поддиапазон представляется другой ссылочной структурой Span<T>, которая также не требует выделения памяти[1].

В листинге 14.1 показано несколько типичных примеров использования Span<T>. Какой бы экземпляр Span<T> мы ни получили в конце метода UseSpan (разные экземпляры представляют разные типы памяти), с ним можно работать как с массивом с помощью свойства Length и индексатора. Заметим, что метод UseSpan помечен как небезопасный, но не потому, что в нем используется Span<T>, а потому, что в нем используется указатель.

Листинг 14.1 ❖ Типичные примеры использования Span<T>

```
unsafe public static void UseSpan()
{
   var array = new int[64];
   Span<int> span1 = new Span<int>(array);
   Span<int> span2 = new Span<int>(array, start: 8, length: 4);
   Span<int> span3 = span1.Slice(0, 4);

   Span<int> span4 = stackalloc[] { 1, 2, 3, 4, 5 };
   Span<int> span5 = span4.Slice(0, 2);

   IntPtr memory = Marshal.AllocHGlobal(64);
   void* ptr = memory.ToPointer();
   Span<byte> span6 = new Span<byte>(ptr, 64);

   var span = span1; // или span2, span3, ...

   for (int i = 0; i < span.Length; i++)
      Console.WriteLine(span[i]);

   Marshal.FreeHGlobal(memory);
}
```

Конечно, не всегда память можно модифицировать. Поэтому есть также тип ReadOnlySpan<T>, представляющий память, которую нельзя изменять. Типичное его применение – представление строковых данных. Строки неизменяемы, а представление их типом Span<char> нарушило бы это ограничение. Поэтому метод расширения AsSpan возвращает значение типа ReadOnlySpan<char>. При желании с помощью этого типа можно сделать доступными только для чтения обычные данные (или экземпляр Span<T>) (листинг 14.2).

Листинг 14.2 ❖ Типичные примеры использования ReadOnlySpan<T>

```
public static void UseReadOnlySpan()
{
   var array = new int[64];
```

[1] Первоначально этот тип даже предполагалось назвать Slice, а не Span.

```
    ReadOnlySpan<int> span1 = new ReadOnlySpan<int>(array);
    ReadOnlySpan<int> span2 = new Span<int>(array);

    string str = "Hello world";
    ReadOnlySpan<char> span3 = str.AsSpan();
    ReadOnlySpan<char> span4 = str.AsSpan(start: 6, length: 5);
}
```

Хотя на первый взгляд все это не кажется чем-то удивительным, но для многих приложений этот тип в корне меняет все. Во-первых, он позволяет существенно упростить некоторые API. Возьмем, к примеру, метод, задача которого – преобразовать строковое представление числа в эквивалентное целое число. Этот метод может получать аргументы разного типа (листинг 14.3). Его API может очень быстро вырасти, если учесть различные варианты использования. С другой стороны, его можно сильно упростить, сведя к единственному методу `Span<char>` (листинг 14.4.)

Листинг 14.3 ❖ Проблематичный API разбора целого числа

```
int Parse(string input);
int Parse(string input, int startIndex, int length);
unsafe int Parse(char* input, int length);
unsafe int Parse(char* input, int startIndex, int length);
```

Листинг 14.4 ❖ Упрощенный API разбора целого числа с применением `Span<T>`

```
int Parse(Span<char> input);
```

Благодаря `Span<T>` открывается возможность представить в различной форме непрерывные последовательности значений (массивы, строки, указатели на неуправляемые массивы и т. д.) и заметно упростить API для работы с ними, избавившись от необходимости писать множество перегруженных вариантов или заставлять пользователя создавать ненужные копии (чтобы адаптировать данные к имеющемуся API).

Во-вторых, `Span<T>` облегчает написание высокопроизводительного кода, например с помощью безопасного использования `stackalloc`, как в листинге 14.1. Важнее, однако, встроенные в этот тип возможности вырезания (slicing), позволяющие работать с меньшими блоками памяти (например, в процессе разбора) и передавать их из одного места программы в другое без сопутствующих накладных расходов. Скоро мы увидим, как реализовано эффективное вырезание. К тому же все это часто можно сделать обобщенным способом, так что становится возможным использование вспомогательных методов или классов.

Компилятор учитывает время жизни данных, обернутых типом `Span<T>`. Поэтому из метода вполне можно вернуть экземпляр `Span<T>`, оборачивающий управляемый массив, поскольку время его жизни больше времени работы метода (см. метод `ReturnArrayAsSpan` в листинге 14.5), но не разрешается возвращать локальные данные в стеке, поскольку они будут отброшены после завершения метода (см. некорректный метод `ReturnStackallocAsSpan` в листинге 14.5). Будьте осторожны с обертыванием неуправляемой памяти, так как впоследствии ее необходимо будет явно освободить (см. метод `ReturnNativeAsSpan` в листинге 14.5, где мы выделили, но не освободили память).

Листинг 14.5 ❖ Три примера возврата Span<T>

```
public Span<int> ReturnArrayAsSpan()
{
   var array = new int[64];
   return new Span<int>(array);
}

public unsafe Span<int> ReturnStackallocAsSpan()
{
   Span<int> span = stackalloc[] { 1, 2, 3, 4, 5 }; // ошибка компиляции CS8352:
   // Cannot use local 'span' in this context because it may expose
   // referenced variables outside of their declaration scope
   return span;
}

public unsafe Span<int> ReturnNativeAsSpan()
{
   IntPtr memory = Marshal.AllocHGlobal(64);
   return new Span<int>(memory.ToPointer(), 8);
}
```

Примеры использования

Рассмотрим несколько примеров использования Span<T>. Имейте в виду, что на момент написания книги тип Span<T> существует в экосистеме .NET не так долго, поэтому еще не сложились общепринятые паттерны его употребления. Но уже есть несколько хороших примеров его использования, особенно в библиотеках для .NET с открытым исходным кодом.

Средства вырезания участков больших данных элегантно используются в сервере Kestrel, который предназначен для запуска веб-приложений ASP.NET Core. В листинге 14.6 приведены соответствующие фрагменты класса HttpParser из репозитория KestrelHttpServer на GitHub. Как видим, построчный разбор входящего HTTP-запроса производится с использованием срезов Span<T>. Сначала каждая строка передается в виде отдельного среза методу ParseRequestLine. Затем вырезаются части этой строки (например, HTTP-путь или запрос) и тоже в виде Span<T> передаются методу OnStartLine. При этом не происходит никакого копирования памяти, как было бы в случае вызова метода string.Substring. Поскольку экземпляр Span<T> создается в стеке, память из кучи вообще не выделяется.

Метод OnStartLine обрабатывает переданный Span<T>; аналогично в том же классе HttpParser анализируются вырезанные HTTP-заголовки.

Листинг 14.6 ❖ Фрагменты класса HttpParser из кода KestrelHttpServer

```
public unsafe bool ParseRequestLine(TRequestHandler handler,
      in ReadOnlySequence<byte> buffer, out SequencePosition consumed,
      out SequencePosition examined)
{
   var span = buffer.First.Span;
   var lineIndex = span.IndexOf(ByteLF);
   if (lineIndex >= 0)
   {
      consumed = buffer.GetPosition(lineIndex + 1, consumed);
```

```
      span = span.Slice(0, lineIndex + 1);
   }
   ...

   // Закрепить и разобрать участок
   fixed (byte* data = &MemoryMarshal.GetReference(span))
   {
      ParseRequestLine(handler, data, span.Length);
   }
}

private unsafe void ParseRequestLine(TRequestHandler handler, byte* data, int length)
{
   int offset;
   // Получить метод и установить смещение
   var method = HttpUtilities.GetKnownMethod(data, length, out offset);

   // Найти индекс pathStart
   var pathBuffer = new Span<byte>(data + pathStart, offset - pathStart);
   ...
   // Найти индекс queryStart
   var targetBuffer = new Span<byte>(data + pathStart, offset - pathStart);
   var query = new Span<byte>(data + queryStart, offset - queryStart);

   handler.OnStartLine(method, httpVersion, targetBuffer, pathBuffer,
                       query, customMethod, pathEncoded);
}
```

Еще один интересный пример использования `Span<T>` – внутренняя ссылочная структура `ValueStringBuilder` в библиотеке .NET CoreFX. Как следует из названия, это вариант `StringBuilder`, являющийся типом значений и предоставляющий функциональность изменяемой строки.

Будучи ссылочной структурой, этот тип всегда размещается в стеке, что избавляет от проблем с многопоточностью (поскольку к стеку может обращаться только текущий поток). Внутреннее хранилище реализовано на базе `Span<char>`, что делает его независимым от источника памяти (листинг 14.7). Память для него может быть получена из стека (`stackalloc`), неуправляемая или массив, находящийся в куче. Индексатор, возвращающий значение по ссылке, предоставляет эффективный доступ к отдельным символам.

Листинг 14.7 ❖ Фрагменты внутренней структуры `ValueStringBuilder`

```
internal ref struct ValueStringBuilder
{
   private char[] _arrayToReturnToPool;
   private Span<char> _chars;
   private int _pos;

   public ValueStringBuilder(Span<char> initialBuffer)
   {
      _arrayToReturnToPool = null;
      _chars = initialBuffer;
      _pos = 0;
   }

   public ref char this[int index]
```

```
   {
      get
      {
         Debug.Assert(index < _pos);
         return ref _chars[index];
      }
   }
   ...
}
```

Как видим, внутреннее поле _pos играет роль курсора, показывающего, сколько символов уже использовано. Текущий результат работы построителя (builder) легко получить с помощью набора методов AsSpan (листинг 14.8) с использованием вырезания (и еще раз повторим, без какого-либо выделения памяти).

Листинг 14.8 ❖ Фрагменты внутренней структуры ValueStringBuilder (возможности вырезания)

```
public ReadOnlySpan<char> AsSpan() => _chars.Slice(0, _pos);
public ReadOnlySpan<char> AsSpan(int start) => _chars.Slice(start, _pos - start);
public ReadOnlySpan<char> AsSpan(int start, int length) => _chars.Slice(start, length);
```

Если вам очень нужна строка, то к вашим услугам метод ToString, выделяющий память в куче (листинг 14.9). Отметим, что в этом случае предполагается, что данный экземпляр построителя уже использован и больше не нужен, поэтому вызывается метод Dispose (объясняется ниже).

Листинг 14.9 ❖ Фрагменты внутренней структуры ValueStringBuilder (возврат строки)

```
public override string ToString()
{
   var s = new string(_chars.Slice(0, _pos));
   Dispose();
   return s;
}
```

Дописывание в конец такого построителя чаще всего сводится к установке символа в позиции курсора (или нескольких символов, если дописывается строка), как показано в листинге 14.10. Конечно, возможно, что в исходном экземпляре Span<char> заканчивается место и требуется увеличить его размер. В таком случае для увеличения хранилища используется массив, полученный из пула ArrayPool<char> (см. метод Grow в листинге 14.10), но также его можно просто присвоить тому же внутреннему экземпляру Span<char> в силу независимости последнего от источника памяти.

Листинг 14.10 ❖ Фрагменты внутренней структуры ValueStringBuilder (логика дописывания в конец)

```
public void Append(char c)
{
   int pos = _pos;
   if (pos < _chars.Length)
   {
      _chars[pos] = c;
```

```
      _pos = pos + 1;
   }
   else
   {
      GrowAndAppend(c);
   }
}

[MethodImpl(MethodImplOptions.NoInlining)]
private void GrowAndAppend(char c)
{
   Grow(1);
   Append(c);
}

[MethodImpl(MethodImplOptions.NoInlining)]
private void Grow(int requiredAdditionalCapacity)
{
   Debug.Assert(requiredAdditionalCapacity > 0);
   char[] poolArray = ArrayPool<char>.Shared.Rent(Math.Max(_pos +
      requiredAdditionalCapacity, _chars.Length * 2));
   _chars.CopyTo(poolArray);
   char[] toReturn = _arrayToReturnToPool;
   _chars = _arrayToReturnToPool = poolArray;
   if (toReturn != null)
   {
      ArrayPool<char>.Shared.Return(toReturn);
   }
}
```

Понятно, что массив, полученный из пула, следует в пул и вернуть. Это делается в методе `Dispose` (листинг 14.11). Отметим, что хотя метод и называется `Dispose`, тип `ValueStringBuilder` не реализует интерфейс `IDisposable`, потому что ссылочным структурам реализация интерфейсов запрещена! Поэтому обязанность явно вызывать `Dispose` для такого экземпляра построителя возлагается на программиста.

Листинг 14.11 ❖ Фрагменты внутренней структуры `ValueStringBuilder` (логика освобождения ресурсов)

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void Dispose()
{
   char[] toReturn = _arrayToReturnToPool;
   this = default; // для безопасности, чтобы избежать использования массива из пула,
                   // в случае когда в этот экземпляр по ошибке дописывают снова
   if (toReturn != null)
   {
      ArrayPool<char>.Shared.Return(toReturn);
   }
}
```

Работать с `ValueStringBuilder` очень просто. Нужно лишь какое-то начальное хранилище (чаще всего для этой цели используют небольшой буфер, полученный с помощью `stackalloc`), которое передается конструктору (листинг 14.12).

Листинг 14.12 ❖ Пример использования `ValueStringBuilder`

```
public string UseValueStringBuilder()
{
   Span<char> initialBuffer = stackalloc char[40];
   var builder = new ValueStringBuilder(initialBuffer);

   // Какой-то код, в котором используется builder.Append(...);

   string result = builder.ToString();
   builder.Dispose();
   return result;
}
```

`ValueStringBuilder` – прекрасный пример типа, в котором применяется много современных приемов программирования: ссылочные структуры (ref structs), возврат по ссылке (ref returns), `Span<T>`, `ArrayPool<T>` и (очень часто) `stackalloc`. Если вы хорошо разберетесь в нем, вы также усвоите эти современные приемы. Рекомендую ознакомиться с исходным кодом `ValueStringBuilder`, который есть в репозитории CoreFX на Github.

> В коде CoreFX также есть очень похожая структура `ValueListBuilder`. Рекомендую ознакомиться и с ней тоже!

Заинтригованные гибкостью `Span<T>`, мы могли бы задуматься о лаконичном решении задачи о получении небольшого локального буфера, показанного в листинге 14.13. Если размер буфера меньше порогового, то мы выделяем буфер в стеке с помощью `stackalloc`, в противном случае получаем его от `ArrayPool`. Выглядит элегантно, код компилируется без ошибок, но у него есть серьезный недостаток. Не существует способа вернуть массив в пул (мы не можем получить исходный массив от экземпляра `Span<T>`)!

Листинг 14.13 ❖ Попытка лаконичного условного получения локального буфера

```
private const int StackAllocSafeThreshold = 128;
public void UseSpanNotWisely(int size)
{
   Span<int> span = size < StackAllocSafeThreshold ? stackalloc int[size] :
                                                     ArrayPool<int>.Shared.Rent(size);
   for (int i = 0; i < size; ++i)
      Console.WriteLine(span[i]);
   //ArrayPool<int>.Shared.Return(??);
}
```

Немного поразмыслив, мы поймем, что описанный выше тип `ValueStringBuilder` решает примерно такую же задачу, как код в листинге 14.14 (с дополнительной возможностью увеличения такого локального буфера).

Пытаясь сделать нечто подобное коду в листинге 14.13, мы упремся в ограничения C# (точнее, текущей версии C# 7.3). Например, невозможно присвоить результат `stackalloc` уже определенной переменной (присваивать можно только в инициализаторе). Поэтому придется написать дополнительный код, так что решение станет не столь лаконичным и элегантным (см. листинг 14.14). Подобный код можно встретить в базовой библиотеке .NET, потому что он делает то, что от

него требуется (правда, требуется ключевое слово unsafe, т. к. в нем используются указатели).

Листинг 14.14 ❖ Попытка лаконичного условного получения локального буфера

```
public unsafe void UseSpanWisely(int size)
{
   int* ptr = default;
   int[] array = null;
   if (size < StackAllocSafeThreshold)
   {
      int* localPtr = stackalloc int[size];
      ptr = localPtr;
   }
   else
   {
      array = ArrayPool<int>.Shared.Rent(size);
   }
   Span<int> span = array ?? new Span<int>(ptr, size);
   for (int i = 0; i < size; ++i)
      Console.WriteLine(span[i]);
   if (array != null) ArrayPool<int>.Shared.Return(array);
}
```

Одно из более типичных применений Span – получение подстроки без выделения памяти с помощью вызовов "some string".AsSpan().Slice(...). Это хороший способ разобрать строку, который не требует накладных вызовов метода string.Substring.

Внутреннее устройство Span<T>

Посмотрев на примеры использования типа Span<T>, обсудим, как он работает. Хотя поначалу этого не скажешь, его реализация отнюдь не тривиальна и раскрывает ряд интересных внутренних особенностей CLR. Поэтому я посвящу много времени тому, чтобы шаг за шагом пояснить различные проектные решения, лежащие в основе внутренних механизмов Span<T>. Если вы торопитесь, можете пропустить этот раздел. Но, как всегда, я призываю внимательно прочитать его! Span<T> – одно из главных изменений в современной экосистеме .NET, поэтому разобраться в нем весьма полезно.

Итак, мы знаем, какие возможности предлагает тип Span<T>. Какие же проектные решения приходят на ум? Начнем по порядку.

- Поскольку этот тип может представлять память, выделенную в стеке (с помощью stackalloc), сам он не может находиться в куче (поскольку тогда он жил бы дольше, чем обернутая им память). Следовательно, мы должны использовать структуру, находящуюся в стеке, и каким-то образом гарантировать, что она никогда не будет упакована (первая трудная проблема).
- Из соображений производительности в любом случае желательно использовать структуру (чтобы не выделять память в куче).
- Поскольку требуется представлять область памяти, понадобятся два элемента информации: указатель (адрес) и размер.
- Если тип Span<T> содержит указатель и размер, то мы столкнемся с проблемами, когда его используют несколько потоков (так называемое *раздирание*

структуры (struct tearing)) – оба поля необходимо изменять атомарно. Но такая обязательная синхронизация должна быть очень эффективна в типе, который проектируется для использования в высокопроизводительном коде (вторая трудная проблема).

- Тип `Span<T>` может представлять участок управляемого массива (например, в результате вырезания), поэтому наш указатель может указывать внутрь управляемого объекта. Если это напоминает вам внутренний указатель – просто отлично! В идеале наш указатель должен быть управляемым (он может указывать внутрь объекта). Но следует помнить, что управляемые указатели допустимы только для локальных переменных, аргументов и возвращаемых значений, но не для полей. Их не должно быть даже в полях структуры, потому что структура может быть упакована (третья трудная проблема).

Из этих наблюдений и появляются идеи того, как лучше всего спроектировать `Span<T>`. Все три трудные проблемы можно было бы решить, если бы:

- у нас был бы тип, который можно создать только в стеке: тогда было бы безопасно хранить в нем относящиеся к стеку адреса, и заодно мы избавились бы от проблем многопоточности, т. к. стек относится только к одному потоку;
- у нас была возможность использовать управляемый указатель в поле `Span<T>`, и тогда мы могли бы безопасно указать на любую интересующую нас область памяти.

Уверен, вы уже все поняли. Действительно, у нас ведь есть типы, допускающие размещение только в стеке, – ссылочные структуры! Эти *byref-подобные типы* (byref-like types) идеально отвечают нашим потребностям (если честно, они и были введены в основном для `Span<T>`). Ко всему прочему, byref-подобные типы не требуют внесения изменений в среду выполнения. Большую часть работы проделывает компилятор C#, при этом сохраняется обратная совместимость на уровне CIL с текущими версиями .NET Core и .NET Framework. Поэтому можно считать, что первое требование выполняется.

Со вторым требованием сложнее. Раз есть byref-подобные типы, то почему бы не быть и *byref-подобным полям экземпляра*: управляемый указатель мог бы являться частью byref-подобного типа, потому что налагаемые на них ограничения взаимосвязаны. Иными словами, управляемый указатель может быть полем находящейся в стеке ссылочной структуры, и это безопасно, поскольку гарантируется, что он никогда не попадет в кучу. К сожалению, ни в языке C#, ни в CIL в настоящее время нет поддержки для таких byref-подобных полей экземпляра, и требуется внести изменения в среду выполнения. Специально для типа `Span<T>` был добавлен новый внутренний (реализованный в среде выполнения) тип для представления byref-подобных полей экземпляра. Поэтому второе требование удовлетворяется только в средах выполнения, поддерживающих это изменение. В настоящее время таковой является лишь .NET Core 2.1 (и более поздние).

Но не все потеряно. Если второе требование не удовлетворяется, то можно найти выход и без поддержки со стороны среды выполнения (скоро мы увидим, какой именно). В результате мы получаем две версии `Span<T>`:

- *«медленный span»* – обратно совместимая версия, существующая в .NET Framework и в версиях .NET Core, предшествующих 2.1, которая не требу-

ет изменений в среде выполнения. Скорее всего, в .NET Framework такие изменения никогда не будут внесены из-за опасения нарушить обратную совместимость;

- *«быстрый span»* – версия, работающая при условии поддержки byref-подобных полей экземпляра, добавленных в .NET Core 2.1.

Не обращайте внимания на слова «медленный» и «быстрый» – обе версии работают достаточно быстро! Медленный вариант лишь немногим медленнее быстрого. Тесты производительности в листинге 14.15 и их результаты в листинге 14.16 ясно показывают, что:

- «быстрый» `Span<T>` в .NET Core 2.1 достигает производительности, сравнимой со стандартным массивом .NET;
- «медленный» Span<T> в .NET Framework действительно медленнее примерно на 25 %.

Но имейте в виду, что этот искусственный тест предназначен только для измерения скорости доступа к данным через индексатор. В более реалистичных примерах различие в производительности составляет 12–15 %.

Листинг 14.15 ❖ Простой тест производительности, сравнивающий производительность Span («медленного» в случае .NET Framework и «быстрого» в случае .NET Core) и стандартного массива

```
public class SpanBenchmark
{
   private byte[] array;

   [GlobalSetup]
   public void Setup()
   {
      array = new byte[128];
      for (int i = 0; i < 128; ++i)
         array[i] = (byte)i;
   }

   [Benchmark]
   public int SpanAccess()
   {
      var span = new Span<byte>(this.array);
      int result = 0;
      for (int i = 0; i < 128; ++i)
      {
         result += span[i];
      }
      return result;
   }

   [Benchmark]
   public int ArrayAccess()
   {
      int result = 0;
      for (int i = 0; i < 128; ++i)
      {
         result += this.array[i];
```

```
      }
      return result;
   }
}
```

Листинг 14.16 ❖ Результаты тестирования производительности из листинга 14.15 с помощью BenchmarkDotNet

```
       Метод |        Задача |  Среднее |     Ошибка| Выделено |
------------ |-------------- |---------:|----------:|---------:|
  SpanAccess |    .NET 4.7.1 | 90.35 нс | 0.1085 нс |      0 Б |
 ArrayAccess |    .NET 4.7.1 | 66.86 нс | 0.7334 нс |      0 Б |
  SpanAccess | .NET Core 2.1 | 65.81 нс | 0.7035 нс |      0 Б |
 ArrayAccess | .NET Core 2.1 | 66.18 нс | 0.0603 нс |      0 Б |
```

Теперь обратимся к деталям реализации обеих версий. Мы рассмотрим только самые интересные аспекты: конструирование как из управляемой, так и из неуправляемой памяти и реализацию индексатора.

> В последующих листингах часто используется класс `Unsafe`. Это обобщенный класс, предоставляющий низкоуровневые операции с памятью и указателями. Он кратко описан далее в этой главе. Использование `Unsafe` в примерах не требует пояснений: он служит для приведения типов и простых арифметических операций над указателями.

Медленный Span

«Медленный Span» должен обходиться без byref-подобных полей. Чтобы имитировать внутренний указатель в поле, мы должны запомнить ссылку на объект и смещение от начала объекта (листинг 14.17). Хранение ссылки на объект позволяет избежать появления дырки при сборке мусора – мы делаем объект достижимым благодаря обертыванию типом `Span<T>`. Понятно, что нужно также хранить длину.

Листинг 14.17 ❖ Объявление «медленного» `Span<T>` в репозитории CoreFX

```
public readonly ref partial struct Span<T>
{
   private readonly Pinnable<T> _pinnable;
   private readonly IntPtr _byteOffset;
   private readonly int _length;
   ...
}

// Единственное назначение этого класса - чтобы произвольные объекты можно
// было привести к нему для получения ссылки на начало пользовательских
// данных
[StructLayout(LayoutKind.Sequential)]
internal sealed class Pinnable<T>
{
   public T Data;
}
```

И как же выглядит конструирование `Span<T>` по данным в управляемой и неуправляемой памяти? Обернуть управляемый массив просто (листинг 14.18). Мы запоминаем полную ссылку на массив (так что сборщик мусора увидит ее и не станет убирать массив в мусор) и сохраняем смещение, с которого начинаются

данные массива (его возвращает свойство ArrayAdjustment), возможно, сдвигая его на нужную величину, если берется срез массива.

Листинг 14.18 ❖ Конструирование «медленного» Span<T> по управляемому массиву

```
public Span(T[] array)
{
   ...
   _length = array.Length;
   _pinnable = Unsafe.As<Pinnable<T>>(array);
   _byteOffset = SpanHelpers.PerTypeValues<T>.ArrayAdjustment;
}

public Span(T[] array, int start, int length)
{
   ...
   _length = length;
   _pinnable = Unsafe.As<Pinnable<T>>(array);
   _byteOffset = SpanHelpers.PerTypeValues<T>.ArrayAdjustment.
   Add<T>(start); // Add method realizes pointer arithmetic
}
```

Обернуть неуправляемую память еще проще, потому что нет никакой ссылки на объект, так что о ней думать не надо (листинг 14.19). Мы сохраняем только длину и адрес.

Листинг 14.19 ❖ Конструирование «медленного» Span<T> по неуправляемой памяти

```
public unsafe Span(void* pointer, int length)
{
   ...
   _length = length;
   _pinnable = null;
   _byteOffset = new IntPtr(pointer);
}
```

Различия в производительности между двумя версиями Span<T> ярче всего проявляются при доступе к элементам. Индексатор «медленного Span» должен проделать больше вычислений: в случае управляемого массива он прибавляет к адресу объекта смещение первого байта данных и смещение (в байтах) элемента по указанному индексу (листинг 14.20).

Листинг 14.20 ❖ Реализация индексатора в «медленном» Span<T>

```
public ref T this[int index]
{
   get
   {
      if (_pinnable == null)
         unsafe { return ref Unsafe.Add<T>(ref Unsafe.AsRef<T>(_byteOffset.ToPointer()),
                  index); }
      else
         return ref Unsafe.Add<T>(ref Unsafe.AddByteOffset<T>
                  (ref _pinnable.Data, _byteOffset), index);
   }
}
```

Если вы хотите изучить исходный код «медленного» Span<T>, загляните в файл .\corefx\src\System.Memory\src\System\Span.Portable.cs.

Быстрый Span

В реализации «быстрого Span» используется поддержка byref-подобных полей самой средой выполнения. На рис. 14.21 показано, как могла бы выглядеть реализация. Но в C# нет синтаксических конструкций для представления byref-подобных полей, поэтому пока она не появится (если это когда-нибудь произойдет), для этой цели используется специальный тип.

Листинг 14.21 ❖ Гипотетический синтаксис byref-подобных полей в объявлении «быстрого» Span<T>

```
public readonly ref partial struct Span<T>
{
   internal readonly ref T _pointer;
   private readonly int _length;
   ...
}
```

Этот тип называется ByReference<T>, поэтому правильное объявление «быстрого» Span<T> выглядит, как показано в листинге 14.22. Внутренний тип ByReference<T> обрабатывается средой выполнения специальным образом, чтобы обернуть его истинную природу управляемого указателя (в настоящее время он используется только в типах Span<T> и ReadOnlySpan<T>).

Листинг 14.22 ❖ Объявление «быстрого» Span<T> (включающее тип ByReference<T>) в репозитории CoreFX

```
// Назначение ByReference<T> - представлять поля вида "ref T". Он нужен, чтобы обойти
// отсутствие полноценной поддержки byref-полей в C# и IL.
// JIT-компилятор и загрузчик типов обрабатывают его специальным образом, превращая в
// обертку над ref T.
[NonVersionable]
internal ref struct ByReference<T>
{
   private IntPtr _value;
   ...
}

public readonly ref partial struct Span<T>
{
   /// <summary>byref или неуправляемый указатель.</summary>
   internal readonly ByReference<T> _pointer;
   /// <summary>Сколько элементов содержит этот Span.</summary>
   private readonly int _length;
   ...
}
```

Благодаря byref-подобному полю реализация этой версии Span<T> проще. И управляемые, и неуправляемые данные хранятся в таком поле (листинг 14.23). Поскольку сборщик мусора знает об управляемых (внутренних) указателях, мож-

но быть уверенным, что соответствующий управляемый объект не будет убран в мусор.

Листинг 14.23 ❖ Конструирование «быстрого» Span<T> по управляемой и неуправляемой памяти

```
public Span(T[] array)
{
   _pointer = new ByReference<T>(ref Unsafe.As<byte, T>(ref array.GetRawSzArrayData()));
   _length = array.Length;
}

public Span(T[] array, int start, int length)
{
   _pointer = new ByReference<T>(ref Unsafe.Add(ref Unsafe.As<byte, T>(ref
       array.GetRawSzArrayData()), start));
   _length = length;
}

public unsafe Span(void* pointer, int length)
{
   _pointer = new ByReference<T>(ref Unsafe.As<byte, T>(ref *(byte*) pointer));
   _length = length;
}
```

А доступ к элементам памяти очень прост и сводится к быстрым арифметическим операциям с указателями (листинг 14.24), что дает производительность, сравнимую с производительностью обычных массивов.

Листинг 14.24 ❖ Реализация индексатора в «быстром» Span<T>

```
public ref T this[int index]
{
   get
   {
      return ref Unsafe.Add(ref _pointer.Value, index);
   }
}
```

Вклад в различие производительности между этими типами дает и другой источник: усовершенствования, внесенные в JIT-компилятор в CoreCLR. В частности, улучшена техника проверки выхода за границы в циклах `for` по «быстрому» Span'у. Кроме того, «быстрый» Span попросту меньше, поэтому его передача по значению обходится дешевле, и это становится заметно в коде, где он передается очень часто.

Интересно, что, с точки зрения затрат на GC, «медленный» и «быстрый» Span<T> в каком-то смысле противоречат своим названиям. «Медленная» версия содержит прямую ссылку на объект (если оборачивается управляемый объект), поэтому обходить ее быстрее. «Быстрая» версия содержит внутренний указатель, разыменование которого производится медленнее (необходим обход и просмотр заполненных блоков). Но это различие пренебрежимо мало, да и трудно представить себе приложение, в котором количество одновременно живых экземпляров Span<T> настолько велико, что разницу можно заметить.

byref-подобные поля в обобщенных типах? Какова вероятность того, что они когда-нибудь появятся в C#? Маловероятно, что это было бы оправдано в классах (поскольку тогда появились бы внутренние указатели из кучи в кучу). Как уже отмечалось, потенциальный выигрыш слишком мал по сравнению с трудностями реализации.

Но почему бы не разрешить byref-подобные поля общего назначения в byref-подобных типах (ссылочных структурах)? Будет ли когда-нибудь возможен код в листинге 14.21? Дискуссии на эту тему продолжаются, и, быть может, вы узнаете ответ спустя год-два после прочтения этой книги. Помимо вырезания частей массивов, которое уже позволяет `Span<T>`, можно представить себе и другие применения таких полей: структуры, связанные друг с другом указателями для ускорения доступа, возврат нескольких результатов по ссылке в одной byref-подобной структуре и т. д. Но, насколько мне известно, такого в планах команды разработчиков CLR нет.

Memory<T>

Тип `Span<T>` хорош и работает быстро. Но, как мы видели, у него много ограничений, и некоторые из них особенно мешают в асинхронном коде. Например, объект типа `Span<T>` не может находиться в куче, а значит, его нельзя упаковывать, т. е. он не может быть полем в асинхронном конечном автомате, который сам может находиться в куче. Поэтому был введен дополнительный тип `Memory<T>`. Он тоже представляет непрерывную область произвольной памяти, но не является byref-подобным типом и не содержит byref-подобных полей экземпляра. Стало быть, в отличие от `Span<T>`, этот тип может находиться в куче (хотя из соображений производительности он тоже является структурой, хоть и не ссылочной). Он может встречаться в полях обычных объектов, его можно использовать в асинхронных конечных автоматах и т. д. Запрещается оборачивать типом `Memory<T>` данные в стеке (например, полученные от `stackalloc`).

Типом `Memory<T>` можно оборачивать следующие данные (листинг 14.25):

- массив `T[]` – играет роль заранее выделенного буфера, повторно используемого на протяжении асинхронного вызова или в API, не допускающих использования `Span<T>`;
- `string` – в этом случае она представлена типом `ReadOnlyMemory<char>`;
- любой тип, реализующий интерфейс `IMemoryOwner<T>`, используется в ситуациях, когда необходим более точный контроль над временем жизни экземпляра `Memory<T>` (такую ситуацию мы рассмотрим ниже).

Листинг 14.25 ❖ Примеры использования `Memory<T>`

```
byte[] array = new byte[] {1, 2, 3};
Memory<byte> memory1 = new Memory<byte>(array);
Memory<byte> memory2 = new Memory<byte>(array, start: 1, length: 2);
ReadOnlyMemory<char> memory3 = "Hello world".AsMemory();
```

Можно рассматривать `Memory<T>` как блок памяти, который можно спокойно передавать методам и получать от методов. Его содержимое по большей части напрямую недоступно. Использовать его можно следующими способами:

- для локального эффективного применения из него можно сгенерировать `Span<T>` (поэтому `Memory<T>` часто называют «фабрикой Span'ов»);

- по Memory<char> можно сгенерировать строку, вызвав метод ToString, в остальных случаях можно использовать метод ToArray (помните, что оба метода выделяют память для нового ссылочного типа!);
- как и для Span<T>, можно вырезать части различными методами Slice.

Вырезание и создание Span<T> – эффективные операции, не выделяющие память, по сути, это просто оборачивание заданной области памяти структурой. И, как мы знаем, иногда всю операцию можно выполнить в регистрах процессора, так что даже не потребуется использовать стек.

Как уже отмечалось, Memory<T> часто используется как замена Span<T> в асинхронном коде (см. листинг 14.26). Внутри асинхронного кода содержимое Memory<T> может быть получено вышеперечисленными способами (в листинге 14.26 используется прямое преобразование методом ToString).

Листинг 14.26 ❖ Пример использования ReadOnlyMemory<T> вместо Span<T> в асинхронном коде

```
public static async Task<string> FetchStringAsync(ReadOnlySpan<char> requestUrl)
   // ошибка CS4012: Parameters or locals of type 'ReadOnlySpan<char>' cannot be declared in
   //  async methods or lambda expressions.
{
   HttpClient client = new HttpClient();
   var task = client.GetStringAsync(requestUrl.ToString());
   return await task;
}

public static async Task<string> FetchStringAsync(ReadOnlyMemory<char> requestUrl)
{
   HttpClient client = new HttpClient();
   var task = client.GetStringAsync(requestUrl.ToString());
   return await task;
}
```

Рассмотрим более сложный пример (листинг 14.27). Класс BufferedWriter реализует буферизованную запись в указанный поток Stream[1]. Внутри него есть небольшой массив байтов (writeBuffer), а в поле writeOffset запоминается, какая его часть уже использована. Единственный открытый метод WriteAsync – асинхронный, поэтому он принимает в качестве источника объект ReadOnlyMemory<byte>. Это позволяет сделать его более общим и гибким, чем различные перегруженные варианты: для массива, строки, неуправляемой памяти и т. д. Наличие зависимости только от ReadOnlyMemory<T> дает возможность писать гораздо более лаконичный код, пока источник совместим с ReadOnlyMemory<T>.

В асинхронном методе WriteAsync тип ReadOnlyMemory<T> нужен для того, чтобы получить от него соответствующий Span и передать закрытому синхронному методу WriteToBuffer, где он и используется. Внутри метода WriteToBuffer еще один объект Span<T> оборачивает writeBuffer, чтобы воспользоваться удобным методом

---

[1] В конкретной реализации Stream буферизация и сброс могут уже присутствовать, но этот пример приведен для демонстрации. На самом деле так устроены классы типа FileStream, в которых поток заменен вызовами ОС.

CopyTo. Кроме того, благодаря средствам вырезания удается написать простой цикл while в методе WriteAsync, который записывает переданные ему данные по частям. Отметим, что класс BufferedWriter не выделяет никакой памяти, кроме массива writeBuffer.

Листинг 14.27 ❖ Пример взаимодействия ReadOnlyMemory<T> и ReadOnlySpan<T>

```
public class BufferedWriter : IDisposable
{
   private const int WriteBufferSize = 32;
   private readonly byte[] writeBuffer = new byte[WriteBufferSize];
   private readonly Stream stream;
   private int writeOffset = 0;

   public BufferedWriter(Stream stream)
   {
      this.stream = stream;
   }

   public async Task WriteAsync(ReadOnlyMemory<byte> source)
   {
      int remaining = writeBuffer.Length - writeOffset;
      if (source.Length <= remaining)
      {
         // Помещается в текущий буфер записи. Просто скопировать и выйти.
         WriteToBuffer(source.Span);
         return;
      }
      while (source.Length > 0)
      {
         // Записать то, что помещается в текущий буфер, и сбросить его.
         remaining = Math.Min(writeBuffer.Length - writeOffset, source.Length);
         WriteToBuffer(source.Slice(0, remaining).Span);
         source = source.Slice(remaining);
         await FlushAsync().ConfigureAwait(false);
      }
   }

   private void WriteToBuffer(ReadOnlySpan<byte> source)
   {
      source.CopyTo(new Span<byte>(writeBuffer, writeOffset, source.Length));
      writeOffset += source.Length;
   }

   private Task FlushAsync()
   {
      if (writeOffset > 0)
      {
         Task task = stream.WriteAsync(writeBuffer, 0, writeOffset);
         writeOffset = 0;
         return task;
      }
      return default;
   }
```

```
   public void Dispose()
   {
      stream?.Dispose();
   }
}
```

IMemoryOwner<T>

С типом Memory<T> связан еще один вопрос – управление временем его жизни. Время жизни Span<T> ограничено временем работы метода, поэтому гарантируется, что обернутая им память не будет использоваться дольше[1]. Напротив, у Memory<T> ограничения на время жизни не такие строгие (поскольку он может оборачивать объекты, находящиеся в куче). Иными словами, связь между Memory<T> и оборачиваемой им памятью не очевидна.

Можно было бы попытаться навязать Memory<T> явное управление ресурсами, ведь обернутую им память можно рассматривать как ресурс. В терминологии .NET, быть может, следует сделать этот тип реализующим IDisposable? Однако экземпляры Memory<T> передаются от одного метода другому, в т. ч. асинхронному. Кто и когда должен нести ответственность за вызов Dispose для такого экземпляра, не понятно. Можно было бы реализовать подсчет ссылок, но у такого решения есть свои проблемы: при попытке написать универсальный код придется думать о синхронизации потоков.

Поэтому было предложено более гибкое решение – дополнительный уровень управления в форме *семантики владения*. Если выдвигается требование управлять временем жизни Memory<T>, то мы должны предоставить владельца как реализацию интерфейса IMemoryOwner<T> (листинг 14.28). Экземпляры Memory<T> можно получить от владельца с помощью свойства Memory. IMemoryOwner<T> наследует интерфейс IDisposable, откуда следует, что владелец сам реализует явное управление ресурсами и управляет временем жизни выданного экземпляра Memory<T>.

Использование объектов типа IMemoryOwner ограничено соглашением (как всегда в случае IDisposable): мы должны помнить о вызове Dispose, например используя объект внутри оператора using. Или можем реализовать свою семантику владения: всегда должен быть только один объект (или метод), который «владеет» экземпляром IMemoryOwner, и понятно, что именно он должен вызвать Dispose по завершении работы.

Листинг 14.28 ❖ Объявление интерфейса IMemoryOwner<T>

```
/// <summary>
/// Владелец экземпляра Memory<typeparamref name="T"/>, отвечающий за освобождение обернутой
/// им памяти.
/// </summary>
public interface IMemoryOwner<T> : IDisposable
{
   Memory<T> Memory { get; }
}
```

[1] Если только мы не передали неуправляемый адрес, см. метод ReturnNativeAsSpan в листинге 14.5.

> Интерфейс IMemoryOwner<T> и семантика владения необязательны в таких простых случаях, как в листинге 14.25. Тогда GC становится единственным неявным «владельцем» обернутой памяти, и он позаботится о том, чтобы убрать ее, когда все использующие ее экземпляры Memory<T> умрут.

Типичный пример, когда явное управление ресурсами необходимо, – оборачивание объекта, полученного из пула, например массива из ArrayPool<T> (листинг 14.29). Если мы арендовали массив у пула и обернули его экземпляром Memory<T>, то когда мы должны его вернуть в пул? В методе Consume из нашего примера? Или, может быть, по завершении await? Но что, если метод Consume где-то сохранил ссылку на переданный экземпляр Memory<T> (это возможно, потому что экземпляр разрешается упаковывать)?

Листинг 14.29 ❖ Проблема с владением памятью, обернутой Memory<T>

```
Memory<int> pooledMemory = new Memory<int>(ArrayPool<int>.Shared.Rent(128));
await Consume(pooledMemory);
```

Интерфейс IMemoryOwner<T> немного помогает навести порядок, так как только метод или класс, хранящий его, должен беспокоиться о явной очистке ресурсов. Экземпляр IMemoryOwner<T> следует передавать очень аккуратно: если какой-то метод или конструктор типа принимает его, то этот метод или тип нужно считать новым владельцем обернутой памяти (т.е. он должен либо вызвать в конце Dispose, либо передать экземпляр дальше). Предполагается, что такой владелец, будь то отдельный метод или целый тип, может безопасно обращаться к свойству Memory.

Чтобы увидеть это в действии, мы можем воспользоваться классом MemoryPool<T>, уже имеющимся в NuGet-пакете System.Memory, который оборачивает экземпляр массива, полученный от ArrayPool<T>.Shared. В листинге 14.30 приведен простой пример, когда управление владением осуществляется с помощью оператора using внутри одного метода, а в листинге 14.31 – пример, когда владельцем обернутой памяти является тип в целом. Во втором случае тип тоже должен реализовывать интерфейс IDisposable, чтобы было понятно, что нужно что-то очистить.

Листинг 14.30 ❖ Пример Memory<T>, в котором явным владельцем является метод

```
using (IMemoryOwner<int> owner = MemoryPool<int>.Shared.Rent(128))
{
   Memory<int> memory = owner.Memory;
   ConsumeMemory(span);
   ConsumeSpan(memory.Span);
}
```

Листинг 14.31 ❖ Пример Memory<T>, в котором явным владельцем является тип

```
public class Worker : IDisposable
{
   private readonly IMemoryOwner<byte> memoryOwner;

   public Worker(IMemoryOwner<byte> memoryOwner)
   {
      this.memoryOwner = memoryOwner;
   }

   public UseMemory()
```

```
   {
      ConsumeMemory(memoryOwner.Memory);
      ConsumeSpan(memoryOwner.Memory.Span);
   }

   public void Dispose()
   {
      this.memoryOwner?.Dispose();
   }
}
```

Свойство `MemoryPool<T>.Shared` использует статический экземпляр `ArrayMemoryPool<T>`, метод `Rent` которого возвращает новый экземпляр `ArrayMemoryPoolBuffer<T>`. Он реализует интерфейс `IMemoryOwner<T>` простым способом: конструктор арендует массив подходящего размера у `ArrayPool<T>.Shared`, а метод `Dispose` возвращает его в пул. Свойство `ArrayMemoryPool<T>.Memory` просто оборачивает арендованный массив новым экземпляром `Memory<T>`. Если вы хотите изучить этот код, обратитесь к файлам `.\corefx\src\System.Memory\src\System\Buffers\ArrayMemoryPool.cs` и `.\corefx\src\System.Memory\src\System\Buffers\ArrayMemoryPool.ArrayMemoryPoolBuffer. cs`.

Например, мы могли бы сделать класс `BufferedWriter` в листинге 14.27 более гибким и позволить ему принимать буфер, вместо того чтобы выделять память самостоятельно (листинг 14.32). Тогда мы сможем использовать в качестве буфера арендованный массив или, например, неуправляемую память.

Листинг 14.32 ❖ Модификация класса `BufferedWriter` из листинга 14.27, в которой буфер передается извне

```
public class FlexibleBufferedWriter : IDisposable
{
   private const int WriteBufferSize = 32;
   private readonly IMemoryOwner<byte> memoryOwner;
   private readonly Stream stream;
   private int writeOffset = 0;

   public FlexibleBufferedWriter(Stream stream, IMemoryOwner<byte> memoryOwner)
   {
      Debug.Assert(memoryOwner.Memory.Length > MinimumWriteBufferSize);
      this.stream = stream;
      this.memoryOwner = memoryOwner;
   }

   ...

   public void Dispose()
   {
      stream?.Dispose();
      memoryOwner?.Dispose();
   }
}
```

Благодаря возможности получить `Span<T>` из `Memory<T>` большая часть реализации нового класса `FlexibleBufferedWriter` очень похожа на предыдущую. Например, метод `WriteToBuffer` теперь вызывает метод `CopyTo` для копирования исходного `Span<T>` в `Span<T>`, представляющий обернутую память (листинг 14.33). В методе

WriteAsync все обращения к writeBuffer.Length можно безопасно заменить обращениями к memoryOwner.Memory.Length.

Листинг 14.33 ❖ Реализация метода FlexibleBufferedWriter.WriteToBuffer

```
private void WriteToBuffer(ReadOnlySpan<byte> source)
{
   source.CopyTo(memoryOwner.Memory.Span.Slice(writeOffset, source.Length));
   writeOffset += source.Length;
}
```

К сожалению, не все API могут работать с типами Span и Memory (но будем надеяться, что вскоре для большинства типов в BCL это будет сделано). Например, до выпуска .NET Core 2.1 метод Stream.WriteAsync принимал только байтовый массив. В таком случае мы должны выполнить подходящее преобразование (листинг 14.34). Если нам повезло и обернутое хранилище является массивом, то вызов MemoryMarshal.TryGetArray завершится успешно (мы рассмотрим класс MemoryMarshal далее в этой главе), и мы получим экземпляр обернутого массива без копирования. В остальных случаях придется скопировать данные во временный массив (и лучше арендовать его у пула, чтобы, по крайней мере, избежать выделения памяти). Заметим, что теперь мы должны вернуть арендованный буфер в пул в методе, из которого вызван FlushAsync.

Будьте готовы к таким решениям при написании низкоуровневого кода. И хотя код в листинге 14.34 может оказаться излишним после доработки Stream API, он все же служит интересным примером взаимодействия различных механизмов, описанных в этой главе.

Листинг 14.34 ❖ Реализация метода FlexibleBufferedWriter.FlushAsync

```
private Task FlushAsync(out byte[] sharedBuffer)
{
   sharedBuffer = null;
   if (writeOffset > 0)
   {
      Task result;
      if (MemoryMarshal.TryGetArray(memoryOwner.Memory, out ArraySegment<byte> array))
      {
         result = stream.WriteAsync(array.Array, array.Offset, writeOffset);
      }
      else
      {
         sharedBuffer = ArrayPool<byte>.Shared.Rent(writeOffset);
         memoryOwner.Memory.Span.Slice(0, writeOffset).CopyTo(sharedBuffer);
         result = stream.WriteAsync(sharedBuffer, 0, writeOffset);
      }
      writeOffset = 0;
      return result;
   }
   return default;
}
```

Обобщенные классы, принимающие буферы, – в общем случае хороший паттерн проектирования, которого должны придерживаться разработчики библио-

тек (по крайней мере, предоставлять эту возможность в числе прочих). В особенности это относится к разного рода сериализаторам и другому коду, интенсивно работающему с памятью, – хорошо, когда он позволяет явно задавать буферы или механизм организации пулов. Тогда вы можете использовать собственные инструменты, а не полагаться на встроенные (или, в худшем случае, вообще отказаться от буферизации и всякий раз выделять память для нового объекта).

> Тип `Memory<T>` можно использовать при работе с P/Invoke, поэтому обернутую память необходимо закреплять. Для этого есть метод `Pin`, который возвращает структуру `MemoryHandle` (IDisposable – объект, представляющий закрепленную память). Строки и массивы закрепляются посредством описателя `GCHandle`. Если экземпляр `Memory<T>` получен от `IMemoryOwner<T>`, ожидается, что объект-владелец реализует абстрактный класс `MemoryManager<T>`. Этот класс реализует интерфейс `IPinnable` с методами `Pin` и `Unpin`. Его метод `Pin` вызывается из метода `Memory<T>.Pin`, а метод `Unpin` – из метода `MemoryHandle.Dispose`. Таким образом, владелец памяти отвечает за закрепление и открепление памяти, которой владеет. Мы не станем подробно распространяться о закреплении `Memory<T>`, поскольку эта тема больше относится к P/Invoke, которая сейчас нас не слишком интересует.

Внутреннее устройство Memory<T>

В отличие от `Span<T>`, реализация `Memory<T>` вполне очевидна и не таит в себе никаких загадок. Объясняется это имеющимися ограничениями на использование управляемых указателей. При проектировании типа `Memory<T>` следует принимать во внимание следующие моменты:

- его время жизни должно быть таким, как у ссылочного типа, хотя изначально он может быть и структурой, а упаковывается только по необходимости;
- объекты, созданные в куче, представляются только ссылкой – в текущей версии внутренние указатели не могут находиться в куче, так что это очевидно. Это упрощает проектирование, потому что есть только два типа, для которых такое поведение имеет смысл: массивы и строки (поскольку они допускают индексирование и вырезание);
- адреса, принадлежащие стеку, представлять необязательно;
- для неуправляемой памяти требуется явное управление ресурсами, и оно может быть реализовано дополнительным классом-владельцем, как объяснялось выше.

Все это подводит нас к простой реализации `Memory<T>`. В листинге 14.35 приведен фрагмент исходного кода CoreFX. Это просто хранимая управляемая ссылка (на массив или на строку), индекс и длина (используется для вырезания). Конструирование также по большей части тривиально.

Листинг 14.35 ❖ Объявление `Memory<T>` в репозитории CoreFX (включая код одного конструктора)

```
public readonly struct Memory<T>
{
   private readonly object _object;
   private readonly int _index;
   private readonly int _length;
   ...
}
```

```
public Memory(T[] array, int start, int length)
{
   ...
   _object = array;
   _index = start;
   _length = length;
}
```

Но в силу своей гибкости тип `Memory<T>` не может использовать обобщенный индексатор. Как уже было сказано, к памяти можно обращаться посредством вырезания и преобразования в `Span<T>`. Само свойство `Span` также реализовано просто (листинг 14.26). Для массива или строки возвращается соответствующий срез. Если же память принадлежит владельцу, то вырезание ему и делегируется (путем вызова метода `GetSpan`).

Листинг 14.36 ❖ Фрагмент реализации свойства `Span` типа `Memory<T>`

```
public Span<T> Span
{
   get
   {
      if (_index < 0)
      {
         return ((MemoryManager<T>)_object).GetSpan().Slice(_index & RemoveFlagsBitMask,
                                                            _length);
      }
      else if (typeof(T) == typeof(char) && _object is string s)
      {
         // вернуть срез строки в виде Span
      }
      else if (_object != null)
      {
         return new Span<T>((T[])_object, _index, _length & RemoveFlagsBitMask);
      }
      ...
   }
}
```

> Просматривая код `Memory<T>`, вы наверняка обратили внимание, что к полям `_index` и `_length` иногда применяются битовые маски, указывающие тип обернутой памяти. Это связано с жесткими требованиями к потреблению памяти. Конечно, можно было бы добавить поле (скажем, типа перечисления – enum), но это увеличило бы размер объекта. Поэтому, например, старший бит `_index` используется, чтобы описать вид `_object`: массив/строка или принадлежащая владельцу память.

Может возникнуть вопрос, как представить неуправляемую память с помощью полей `Memory<T>`, показанных в листинге 14.35. Поскольку для неуправляемой памяти необходима явная очистка, поле `_object` в этом случае является соответствующей реализацией `MemoryManager<T>`, отвечающей за выделение и освобождение обернутой памяти. В листинге 14.37 показана схематичная реализация такого менеджера, идея которой подсказана внутренним классом `NativeMemoryManager` из пространства имен `System.Buffers`.

Листинг 14.37 ❖ Пример менеджера неуправляемой памяти

```
class NativeMemoryManager : MemoryManager<byte>
{
   private readonly int _length;
   private IntPtr _ptr;

   public NativeMemoryManager(int length)
   {
      _length = length;
      _ptr = Marshal.AllocHGlobal(length);
   }

   protected override void Dispose(bool disposing)
   {
      ...
      Marshal.FreeHGlobal(_ptr);
      ...
   }

   public override Memory<byte> Memory => CreateMemory(_length);
   // Создает экземпляр Memory<T>, оборачивающий this

   public override unsafe Span<byte> GetSpan() => new Span<byte>((void*)_ptr, _length);
}
```

Рекомендации по работе с Span<T> и Memory<T>

Мы многое узнали об этих типах, но возникает вопрос: когда их использовать и какой из двух предпочесть? Ниже приведены некоторые рекомендации по этому поводу.

- Используйте `Span<T>` или `Memory<T>` в специализированном высокопроизводительном коде – как правило, включать их в код бизнес-приложений не имеет смысла.
- Отдавайте предпочтение передаче `Span<T>` в качестве аргумента метода, если это возможно, – он быстрее работает (благодаря поддержке со стороны среды выполнения) и может представить больше видов памяти. Но в асинхронном коде альтернативы `Memory<T>` не существует.
- Отдавайте предпочтение варианту, допускающему только чтение, чтобы сделать код безопаснее. Не используйте по умолчанию вариант, допускающий изменение. Кроме того, версия только для чтения более «либеральна»; например, если метод принимает `Span<T>`, то передать ему аргумент типа `ReadOnlySpan<T>` не получится, а если он принимает `ReadOnlySpan<T>`, то можно передать и `Span<T>`, и `ReadOnlySpan<T>`.
- Помните, что интерфейс `IMemoryOwner<T>` (или `MemoryManager<T>`) обладает семантикой владения, а значит, в какой-то момент для реализующего его объекта нужно вызвать метод `Dispose`. В целях безопасности лучше, если в каждый момент времени такой экземпляр будет хранить только один объект. Тип, хранящий `IMemoryOwner<T>` (наследник от IDisposable), сам должен реализовывать IDisposable (чтобы правильно управлять этим ресурсом).

Класс Unsafe

Класс System.Runtime.CompilerServices.Unsafe предоставляет низкоуровневые средства для манипулирования указателями более безопасным способом, чем простой небезопасный код (основанный на указателях и операторе fixed), и для использования функциональности, возможной в CIL, но не напрямую в C#. Однако все эти средства действительно небезопасны! Благодаря своей гибкости класс Unsafe широко используется в современных библиотеках для .NET (многие типы, включая Span<T>, Memory<T> и другие, зависят от него).

Описывать все возможности класса Unsafe в этой книге вряд ли имеет смысл, поскольку это все равно, что описывать арифметику указателей или приведение типов – делать с ними можно все, что угодно. Вместо этого мы дадим краткий обзор методов класса и несколько примеров его использования, чтобы вы могли получить представление о том, что и как можно делать с его помощью.

У класса System.Runtime.CompilerServices.Unsafe есть обширный набор методов (листинг 14.38). Их можно разбить на следующие функциональные группы:

- приведение типов и реинтерпретация – можно переходить от неуправляемого указателя к ссылке, и наоборот. Дополнительно можно преобразовать один ссылочный тип в другой (да, это именно так опасно, как кажется);
- арифметика указателей – экземпляры ссылочных типов можно складывать и вычитать, как обычные указатели (и если вы помните описание управляемых указателей, то сможете представить себе те граничные случаи, где это чертовски опасно);
- информационные – позволяют получать различную информацию, например размер ссылочного типа или расстояние в байтах между двумя экземплярами ссылочного типа;
- доступ к памяти – можно читать и записывать все, что угодно и куда угодно.

Листинг 14.38 ❖ API класса Unsafe – некоторые перегруженные варианты для краткости опущены, методы объединены в группы по функциональному назначению, комментарии добавлены мной

```
public static partial class Unsafe
{
   // Приведение и реинтерпретация
   public unsafe static void* AsPointer<T>(ref T value)
   public unsafe static ref T AsRef<T>(void* source)
   public static ref TTo As<TFrom, TTo>(ref TFrom source)

   // Арифметика указателей
   public static ref T Add<T>(ref T source, int elementOffset)
   public static ref T Subtract<T>(ref T source, int elementOffset)

   // Информационные методы
   public static int SizeOf<T>()
   public static System.IntPtr ByteOffset<T>(ref T origin, ref T target)
   public static bool IsAddressGreaterThan<T>(ref T left, ref T right)
   public static bool IsAddressLessThan<T>(ref T left, ref T right)
   public static bool AreSame<T>(ref T left, ref T right)

   // Методы доступа к памяти
   public unsafe static T Read<T>(void* source)
```

```
   public unsafe static void Write<T>(void* destination, T value)
   public unsafe static void Copy<T>(void* destination, ref T source)

   // Блочный доступ к памяти
   public static void CopyBlock(ref byte destination, ref byte source, uint byteCount)
   public unsafe static void InitBlock(void* startAddress, byte value, uint byteCount)
}
```

Понятно, что `Unsafe` – специализированный класс. Им можно пользоваться только в очень специфических, тщательно контролируемых местах, когда программист точно знает, что делает, и учитывает все граничные случаи. Не нужно считать этот класс способом преодоления странных проблем с безопасностью типов, например чтобы обойти иерархию типов в объектно-ориентированном программировании!

Рассмотрим несколько примеров. Для начала напомним, что мы уже видели важные применения класса `Unsafe` в листингах 14.18, 14.20, 14.23 и 14.24, где приведение типов и арифметика указателей использовались для реализации `Span<T>`.

Но приведение – очень мощное средство. Например, мы можем привести один управляемый тип к другому, никак не связанному с первым (листинг 14.39). Память исходного объекта реинтерпретируется в соответствии с размещением полей конечного объекта. В нашем простом примере мы реинтерпретируем два соседних целых числа как одно длинное целое, и, возможно, это даже имеет смысл. Отметим, что, несмотря на использование таких низкоуровневых операций над указателями, метод `DangerousPlays` не помечен ключевым словом `unsafe`, потому что все происходит внутри класса `Unsafe`.

Листинг 14.39 ❖ Опасный, но работающий код – приведение с помощью метода `Unsafe.As`

```
public class SomeClass
{
   public int Field1;
   public int Field2;
}

public class SomeOtherClass
{
   public long Field;
}

public void DangerousPlays(SomeClass obj)
{
   ref SomeOtherClass target = ref Unsafe.As<SomeClass, SomeOtherClass>(ref obj);
   Console.WriteLine(target.Field);
}
```

> Такое приведение типов используется, например, чтобы обойти правила изменяемости (mutability rules), и позволяет переходить от типа `Memory<T>` к типу `ReadOnlyMemory<T>` и обратно. Разумеется, при этом требуется, чтобы размещение обоих типов в памяти было одинаковым.

Приведение типов активно используется, например, в статическом классе `BitConverter` для преобразования различных типов в массивы байтов и обратно (листинг 14.40).

Листинг 14.40 ❖ Пример использования Unsafe в классе BitConverter

```
public static byte[] GetBytes(double value)
{
   byte[] bytes = new byte[sizeof(double)];
   Unsafe.As<byte, double>(ref bytes[0]) = value;
   return bytes;
}
```

А теперь представьте себе реинтерпретацию примитивного типа как ссылки или наоборот! Очевидно, что это очень опасно и, скорее всего, закончится аварийным завершением самой среды выполнения. В листинге 14.41 дан пример такого беспечного приведения. Метод VeryDangerous вызовет исключение AccessViolationException (если только нам не будет сопутствовать фантастическое везение и значение Long1 окажется допустимой строкой).

Листинг 14.41 ❖ Очень опасный код – приведение методом Unsafe.As

```
public struct UnmanagedStruct
{
   public long Long1;
   public long Long2;
}

public struct ManagedStruct
{
   public string String;
   public long Long2;
}

public void VeryDangerous(ref UnmanagedStruct data)
{
   ref ManagedStruct target = ref Unsafe.As<UnmanagedStruct,
   ManagedStruct>(ref data);
   Console.WriteLine(target.String); // значение Long1 трактуется как ссылка на строку!
}
```

Арифметические операции над указателями – еще одно популярное использование Unsafe. Хорошим примером может служить реализация статического метода Array.Reverse (листинг 14.42). Это не что иное, как реинкарнация стандартного кода на C или C++, в котором манипулирование указателями применяется для разворачивания массива на месте.

Листинг 14.42 ❖ Пример использования Unsafe в статическом методе Array.Reverse

```
public static void Reverse<T>(T[] array, int index, int length)
{
   ...
   ref T first = ref Unsafe.Add(ref Unsafe.As<byte, T>(ref array.GetRawSzArrayData()),
                                                       index);
   ref T last = ref Unsafe.Add(ref Unsafe.Add(ref first, length), -1);
   do
   {
      T temp = first;
      first = last;
```

```
      last = temp;
      first = ref Unsafe.Add(ref first, 1);
      last = ref Unsafe.Add(ref last, -1);
   } while (Unsafe.IsAddressLessThan(ref first, ref last));
}
```

Поскольку применение `Span<T>`, `Memory<T>` и `Unsafe` требует одних и тех же приемов, был разработан вспомогательный класс `MemoryMarshal`, содержащий много статических методов. Перечислим лишь некоторые из них:

- `AsBytes` – преобразует любой `Span<T>` примитивного типа (структуры) в `Span<byte>`;
- `Cast` – выполняет преобразование между двумя `Span<T>` примитивного типа (структур);
- `TryGetArray`, `TryGetMemoryManager`, `TryGetString` – пытается преобразовать `Memory<T>` (или `ReadOnlyMemory<T>`) в указанный тип;
- `GetReference` – возвращает по ссылке объект `Span<T>` или `ReadOnlySpan<T>`.

Класс `MemoryMarshal` даже позволяет легко делать «магические» вещи. Например, можно взять часть некоторой структуры и реинтерпретировать ее как другую структуру, притом без какого-либо копирования (листинг 14.43).

Листинг 14.43 ❖ Пример использования `MemoryMarshal`

```
public struct SmallStruct
{
   public byte B1;
   public byte B2;
   public byte B3;
   public byte B4;
   public byte B5;
   public byte B6;
   public byte B7;
   public byte B8;
}

public unsafe void Reinterpretation(ref UnmanagedStruct data)
{
   var span = new Span<UnmanagedStruct>(Unsafe.AsPointer(ref data), 1);
   ref var part = ref MemoryMarshal
        // привести Span<byte> к Span<SmallStruct>
        .Cast<byte, SmallStruct>(
           // привести Span<UnmanagedStruct> к Span<byte>
           MemoryMarshal.AsBytes(span)
              // вырезать часть и получить первый элемент
              .Slice(0, 8))[0];
   Console.WriteLine(part.B1); // получить первый байт
}
```

Может возникнуть вопрос: а для чего нужна вся эта «магия»? Нужна ли она обычному .NET-разработчику? Честно говоря, не особенно. Я вижу использование класса `Unsafe` только в низкоуровневых библиотеках – для сериализации, бинарного протоколирования, сетевого взаимодействия и т. д. Например, в популярной библиотеке `jemalloc.NET` он используется для реализации строгой типизации неуправляемой памяти (листинг 14.44).

Листинг 14.44 ❖ Пример использования `Unsafe` в jemalloc.NET – метод `FixedBuffer.Read`

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public unsafe ref C Read<C>(int index) where C : struct
{
   return ref Unsafe.AsRef<C>(PtrTo(index));
}
```

jemalloc.NET – замечательная библиотека для .NET, написанная Эллистером Бехарри (Allister Beharry) и размещенная на GitHub (https://github.com/allisterb/jemalloc.NET). По словам автора, это «обертка вокруг неуправляемого распределителя памяти jemalloc, которая предоставляет .NET-приложениям эффективные структуры данных в неуправляемой памяти для массивных вычислений в памяти». jemalloc – действительно популярная и эффективная замена функции `malloc`. Вы можете почитать о ее реализации на сайте http://jemalloc.net/, а также поэкспериментировать с библиотекой jemalloc.NET. Из-за ограниченного размера книги я не без сожаления вынужден опустить описание этой библиотеки.

И раз уж мы заговорили об обертках неуправляемой памяти, должен сказать, что и в компании Майкрософт ведется работа в этом направлении – проект Snowflake. В настоящее время он заморожен, но рано или поздно можно ожидать открытия исходного кода. Прочитать о нем можно по адресу https://www.microsoft.com/en-us/research/publication/project-snowflake-non-blocking-safe-manual-memory-anagement-net/.

Внутреннее устройство Unsafe

На самом деле класс `Unsafe` оборачивает различные возможности языка IL, которые иначе были бы недоступны в C#, поскольку контроль типов в IL менее строгий, чем в компиляторе C#. Реализация большинства методов `Unsafe` на CIL тривиальна (листинг 14.45).

Листинг 14.45 ❖ Пример реализации методов `Unsafe` на общеязыковом промежуточном языке

```
.method public hidebysig static !!TTo& As<TFrom, TTo> (!!TFrom& source) cil managed
{
   IL_0000: ldarg.0
   IL_0001: ret
}

.method public hidebysig static !!T& Add<T> (!!T& source, int32 elementOffset) cil managed
{
   IL_0000: ldarg.0
   IL_0001: ldarg.1
   IL_0002: sizeof !!T
   IL_0008: conv.i
   IL_0009: mul
   IL_000A: add
   IL_000B: ret
}
```

Никакой магии в классе `Unsafe` нет. А полезен он тем, что дает доступ к этим операциям, которые часто можно использовать даже в безопасном коде.

Проектирование, ориентированное на данные

Разрыв между производительностью процессора и временем доступа к памяти постоянно увеличивается. Мы подробно обсуждали этот вопрос в главе 2: как организована иерархическая память и насколько сильно строчная структура кеша и внутренняя реализация памяти влияют на производительность нашего кода. В частности, мы говорили, что предпочтительным является последовательный доступ к памяти с хорошей временной и пространственной локальностью.

Такой низкоуровневый взгляд на память не обязателен в процессе рутинной разработки коммерческих приложений (как веб-, так и настольных). Миллисекунды, отличающие высокую производительность от не очень высокой, попросту незаметны при небольшом объеме обрабатываемых данных, HTTP-запросов или взаимодействия с UI. При разработке таких приложений гораздо важнее удобочитаемость, расширяемость и понятность кода, а также способность быстро писать, поставлять и развивать программное обеспечение. Объектно-ориентированное программирование со всеми его паттернами проектирования и принципами SOLID – воплощение именно такого подхода.

Однако есть небольшая категория приложений, при разработке которых имеет смысл отойти от этих универсальных принципов. Это приложения, которые должны максимально эффективно обрабатывать значительные объемы данных и получать результат как можно быстрее. Здесь играет роль каждая миллисекунда. Назовем лишь несколько примеров:

- финансовое ПО – особенно быстрой реакции требует биржевая торговля в режиме реального времени, а также аналитические решения, принимаемые на основе обработки больших объемов разнородных данных;
- большие данные (Big Data) – обычно с ними ассоциируется медленная пакетная обработка, но миллисекунды, потраченные на обработку одного элемента данных, складываются в часы и дни. И кроме того, существуют приложения, в которых быстрота получения ответа имеет решающее значение, например поисковые системы;
- игры – в мире, где от FPS (Frames per seconds – число кадров в секунду) зависит коммерческий успех игры и качество графики, каждая миллисекунда имеет значение;
- машинное обучение – для выполнения сложных алгоритмов никогда не хватает мощности процессора.

Заметим, что хотя, на первый взгляд, многие из этих приложений могут зависеть от скорости процессора (т. е. выполняют сложные алгоритмы), из-за вышеупомянутого разрыва может оказаться, что узким местом является именно доступ к памяти. И есть еще один, пока не упоминавшийся аспект – параллельная обработка данных, в которой задействовано несколько ядер, установленных в персональном компьютере или сервере.

Это приводит нас к *проектированию ПО, ориентированному на данные*, в фокусе которого находится проектирование представления данных и архитектуры таким образом, чтобы был обеспечен максимально эффективный доступ к памяти. Почти всегда этот подход конфликтует с объектно-ориентированным проектированием, потому что такие концепции, как инкапсуляция и полиморфизм, препятствуют эффективному использованию памяти.

Перечислим цели проектирования, ориентированного на данные:

- проектирование типов и данных, позволяющих всюду, где возможно, использовать последовательный доступ к памяти с учетом ограничений, налагаемых строками кеша (располагать вместе наиболее часто используемые данные), и иерархической природы кеша (хранить как можно больше данных в кешах более высокого уровня);
- проектирование типов и данных, а также работающих с ними алгоритмов так, чтобы их можно было легко распараллелить, не неся затрат на дорогостоящую синхронизацию.

Я бы выделил в проектировании, ориентированном на данные, две дополнительные категории:

- *тактическое проектирование, ориентированное на данные*, – сосредоточено на «локальных» структурах данных, например на оптимальном размещении полей в памяти и на доступе к данным в правильном порядке. Эти соображения достаточно локальны для включения в уже существующие объектно-ориентированные приложения;
- *стратегическое проектирование, ориентированное на данные*, – сосредоточено на верхнеуровневой архитектуре приложения. Обычно для этого требуется отойти от объектно-ориентированных структур в сторону структур, которые более ориентированы на данные.

В следующих двух разделах мы более подробно рассмотрим оба аспекта проектирования.

Тактическое проектирование

Эта книга в основном пронизана духом тактического проектирования, ориентированного на данные, поскольку еще в главе 2 мы узнали, насколько важно правильно использовать кеш, и подытожили это в *правиле 2* «Избегайте произвольного доступа» и в *правиле 3* «Улучшайте пространственную и временную локальность данных».

Такое тактическое проектирование сводится к нескольким паттернам. Остановимся на них, при этом будем ссылаться на другие части книги и приводить дополнительные примеры.

Проектировать типы так, чтобы как можно больше релевантных данных попадало в первую строку кеша

Мы видели это правило в действии, когда рассматривали автоматическое размещение управляемых типов в памяти, – ссылки размещаются в начале объекта, чтобы они были доступны сборщику мусора в уже прочитанной строке кеша, содержащей указатель на таблицу методов. Эту оптимизацию производит CLR, но мы должны знать о ней.

Такое автоматическое размещение не всегда желательно с точки зрения частоты доступа к данным. Рассмотрим класс на рис. 14.46. Программисту, предпочитающему объектно-ориентированный подход, такой дизайн понравился бы[1] – все

[1] Но если принять во внимание предметно-ориентированное проектирование, то он, наверное, был бы еще сложнее – для представления денежных величин и других данных использовались бы отдельные типы.

инкапсулировано в одном объекте, и снаружи видно только поведение (вычисление оценки).

Листинг 14.46 ❖ Пример класса, иллюстрирующего использование строк кеша

```
class Customer
{
   private double Earnings;
   // ... еще какие-то поля ...
   private DateTime DateOfBirth;
   // ... еще какие-то поля ...
   private bool IsSmoking;
   // ... еще какие-то поля ...
   private double Scoring;
   // ... еще какие-то поля ...
   private HealthData Health;
   private AuxiliaryData Auxiliary;

   public void UpdateScoring()
   {
      this.Scoring = this.Earnings * (this.IsSmoking ? 0.8 : 1.0) *
         ProcessAge(this.DateOfBirth);
   }

   private double ProcessAge(DateTime dateOfBirth) => 1.0;
}
```

Такому программисту совсем неинтересно окончательное автоматическое размещение объекта `Customer` в памяти. С другой стороны, предположим, что класс `Customer` используется очень активно, и метод `UpdateScoring` вызывается для миллионов таких объектов в секунду. Поскольку в этом методе используются поля `Scoring`, `Earning`, `IsSmoking` и `DateOfBirth`, их следует разместить в пределах первой строки кеша (той, к которой всегда производится доступ при работе с объектом `Customer`). Понятно, что тип размещения `LayoutKind.Automatic`, подразумеваемый по умолчанию для классов, на это не рассчитан. В этом случае ссылочные поля `HealthData` и `AuxiliaryData`, которые, наверное, используются очень редко, будут помещены в начало объекта, а остальные – в соответствии с требованиями к выравниванию (как было объяснено в предыдущей главе).

Решение нам уже известно: нужно сделать `Customer` неуправляемой структурой с последовательным размещением (листинг 14.47). Для этого следует:

- преобразовать `HealthData` и `AuxiliaryData` в типы значений, чтобы избавиться от ссылок, – это не только делает типы неуправляемыми, но и снимает со сборщика мусора часть накладных расходов на пометку (поскольку экземпляр `Customer` перестанет быть корнем двух дополнительных объектов, которые нужно просматривать);
- заменить `DateTime` другим типом, поскольку из-за его автоматического размещения приходится выбирать автоматическое размещение и для всей структуры (см. главу 13).

Затем можно будет использовать атрибут `LayoutKind.Sequential` и тщательно продумать размещение полей (возможно, при этом появится дополнительное заполнение из-за выравнивания, но мы принесем эту жертву ради быстроты доступа). И таким образом, четыре самых часто используемых поля будут перенесены в начало.

Листинг 14.47 ❖ Размещение структуры с учетом использования строки кеша

```
[StructLayout(LayoutKind.Sequential)]
struct CustomerValue
{
   public double Earnings;
   public double Scoring;
   public long DateOfBirthInTicks;
   public bool IsSmoking;
   // ... еще какие-то поля ...
   public int HealthDataId;
   public int AuxiliaryDataId;
}
```

Но не всегда для достижения хорошей пространственной локальности нужно использовать последовательное размещение. Иногда достаточно позаботиться о локальности примитивных типов (иными словами, гарантировать, что часто используемые поля расположены рядом).

Типы `FrugalObjectList<T>` и `FrugalStructList<T>` – примеры очень интересных коллекций в библиотеке Windows Presentation Library. В качестве внутреннего хранилища в них используется одна из следующих специфических коллекций: `SingleItemList<T>`, `ThreeItemList<T>`, `SixItemList<T>` и `ArrayItemList<T>`. В процессе добавления или удаления элементов тип хранилища может меняться между этими типами коллекций (последний предназначен для хранения семи и более элементов). И что это дает? Очень компактную, простую, основанную на операторе `switch` реализацию таких методов, как `IndexOf`, `SetAt` и `EntryAt`, используемых индексатором, в случаях когда элементов меньше семи (в листинге 14.49 показаны фрагменты кода `ThreeItemList<T>`). Такой подход, с одной стороны, позволяет обойтись без накладных расходов, связанных с обобщенным массивом (в частности, избавиться от проверки выхода за границы массива), а с другой – обеспечивает хорошую локальность данных, поскольку три или шесть полей размещены рядом друг с другом.

Листинг 14.48. Фрагменты класса `ThreeItemList<T>` (одного из типов внутреннего хранилища в типах `FrugalObjectList<T>` и `FrugalStructList<T>`)

```
/// <summary>
/// Простой класс для работы со списком трех элементов. Анализ показал, что при этом
/// достигается лучшая локальность данных и более высокая производительность, чем
/// при использовании массива.
/// </summary>
internal sealed class ThreeItemList<T> : FrugalListBase<T>
{
   public override T EntryAt(int index)
   {
      switch (index)
      {
         case 0:
            return _entry0;

         case 1:
            return _entry1;

         case 2:
```

```
            return _entry2;

         default:
            throw new ArgumentOutOfRangeException("index");
      }
   }
   private T _entry0;
   private T _entry1;
   private T _entry2;
}
```

В комментариях к этим типам написано: «Измерения производительности показывают, что в Avalon[1] часто встречаются списки, содержащие всего несколько элементов, зачастую 0 или один. (...) Поэтому в этих классах модель хранения устроена так, что сначала количество элементов равно 0, а затем понемногу возрастает, чтобы минимизировать потребление памяти в стабилизировавшемся состоянии. (...) Код также структурирован, чтобы добиться более быстрого выполнения процессором. Измерения показывают, что благодаря сниженному количеству непопаданий в процессорный кеш `FrugalList` оказывается быстрее, чем `ArrayList` или `List<T>`, особенно для списков с 6 и менее элементами».

Проектировать данные так, чтобы они помещались в кешах верхних уровней

Накладные расходы кешей разного уровня проиллюстрированы в листинге 2.5 и соответствующем ему рис. 2.11 в главе 2. Вы всегда должны понимать, сколько у вас данных и как их размер соотносится с типичными размерами кешей процессора.

Проектировать данные с расчетом на простоту распараллеливания

Тема параллельной обработки выходит за рамки этой книги. Но благодаря удачно выбранному размещению данных и хорошо спроектированному алгоритму иногда удается обрабатывать часть данных параллельно – несколькими ядрами и (или) командами SIMD. Но помните о феномене ложного разделения (false sharing), который иллюстрируется в листинге 2.6 и подтверждается результатами тестирования производительности в табл. 2.3.

Избегайте непоследовательного, а особенно произвольного доступа к памяти

Это правило объяснялось в главе 2, где были описаны принципы работы DRAM-памяти и сказано, почему следует отдавать предпочтение последовательному доступу. Простой пример доступа к двумерному массиву по строкам и столбцам был приведен в листинге 2.1, а результаты тестирования производительности в табл. 2.1 показывают, что доступ замедляется в несколько раз из-за большого количества непопаданий в кеш.

Доступ к последовательному непрерывному участку `T[]` предпочтительнее, чем доступ к другим коллекциям, особенно если тип `T` – структура (вспомните

[1] Avalon – кодовое название движка WPF.

рис. 4.22, на котором сравнивается локальность данных для массивов). Это правило проектирования еще встретится нам при описании стратегических паттернов проектирования.

Стратегическое проектирование

Стратегическое проектирование выдвигает на первый план проектирование, ориентированное на данные, и знаменует расставание с практиками, типичными для объектно-ориентированного проектирования. Получающийся при этом код может удивить разработчиков, использующих ООП, но, если задуматься, оказывается оправданным. Поэтому, в отличие от тактического, стратегическое проектирование требует от программиста пересмотра своих привычек и убеждений. А теперь рассмотрим некоторые из наиболее популярных приемов.

Переход от массива структур к структуре массивов

В объектно-ориентированном программировании данные инкапсулированы. Объекты и методы описывают тщательно продуманное поведение, где каждый отвечает за что-то одно. Например, можно представить себе, что экземпляры типа `Customer` из листинга 14.46 хранятся в отдельном «контейнере». Его метод `UpdateScorings` перебирает всех клиентов и просит каждого обновить свою оценку (листинг 14.49). Это простой код, понятный каждому разработчику, практикующему ООП.

Листинг 14.49 ❖ Репозиторий объектов `Customer` из листинга 14.46

```
class CustomerRepository
{
   List<Customer> customers = new List<Customer>();

   public void UpdateScorings()
   {
      foreach (var customer in customers)
      {
         customer.UpdateScoring();
      }
   }
}
```

При таком коде возникает много непопаданий в кеш – экземпляры `Customer` могут быть разбросаны по всей управляемой куче, и нет никакой гарантии, что они окажутся рядом (рис. 14.1). Правда, как мы знаем, сборка мусора с уплотнением в конечном счете может привести к хорошей локальности объектов, созданных примерно в одно и то же время. Кроме того, распределитель сдвигом указателя (bump-a-pointer allocator) может изначально размещать объекты по соседству. Но все это лишь предположения, а не гарантии. Например, заполненный контекст выделения памяти может быть заменен новым, который может находиться в любом месте эфемерного сегмента, – тогда даже два объекта `Customer`, созданных сразу один за другим, могут оказаться в совершенно разных местах. В результате следует предполагать, что при работе с массивом ссылочных типов в каждой строке кеша будет лишь небольшая часть интересных данных и куча не относящегося к делу мусора.

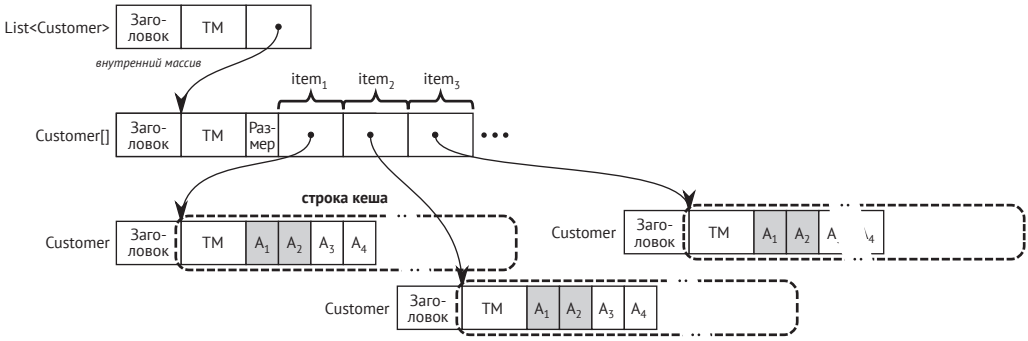


Рис. 14.1 ❖ Из-за плохой локальности данных в массиве ссылочных типов приходится заполнять много строк кеша ненужными данными (нужные данные закрашены серым)

Мы знаем, что массив структур обеспечивает гораздо лучшую локальность данных, поэтому в `CustomerRepository` можно было бы хранить не экземпляры `Customer`, а список упакованных экземпляров структуры `CustomerValue`, определенной в листинге 14.47 (рис. 14.2). При последовательном чтении стоящего за `List` массива строки кеша используются гораздо эффективнее, поскольку устройство предвыборки процессора (CPU's prefetcher) легко распознает такую закономерность и будет выбирать данные заранее. И мусора в каждой строке кеша стало меньше – только другие, пока ненужные поля экземпляров `CustomerValue`.

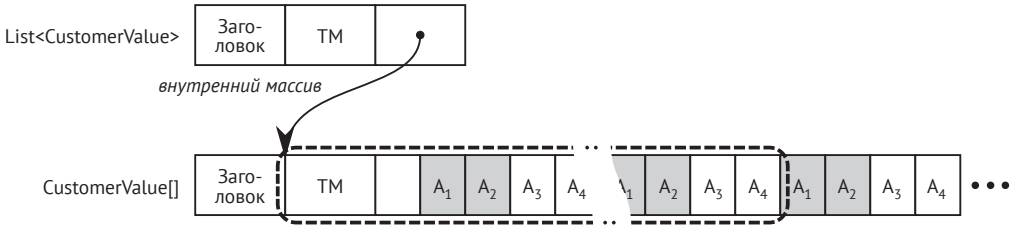


Рис. 14.2 ❖ Локальность данных для массива типов значений намного лучше, теперь в строки кеша читается гораздо меньше ненужных данных (нужные данные закрашены серым)

Однако в случаях, когда требуется максимальная производительность, чтение ненужных данных (полей) все еще обходится слишком дорого. И вот тут настает момент, когда нужно расстаться с хорошо знакомой парадигмой ООП и все поменять. При проектировании, ориентированном на данные, самое важное – не объекты и инкапсулируемое ими поведение, а сами данные. В нашем случае данными являются несколько важных атрибутов Customer (входных и выходных).

Первое, что можно сделать, – разбить данные Customer на два отдельных массива типов значений; один будет содержать «горячие данные», используемые в алгоритме вычисления оценки, а второй – все остальные, не столь важные поля.

Но можно пойти еще дальше. Вместо того чтобы строить код вокруг Customer, мы можем организовать его вокруг самих данных, поместив все элементы данных

в разные массивы (листинг 14.50). Этот подход, один из самых популярных в проектировании, ориентированном на данные, называют переходом от *AoS* (*массива структур*) к *SoA* (*структуре массивов*).

Листинг 14.50 ❖ Пример организации данных в виде структуры массивов

```
class CustomerRepository
{
   int NumberOfCustomers;
   double[] Scoring;
   double[] Earnings;
   DateTime[] DateOfBirth;
   bool[] IsSmoking;
   // ...

   public void UpdateScorings()
   {
      for (int i = 0; i < NumberOfCustomers; ++i)
      {
         Scoring[i] = Earnings[i] * (IsSmoking[i] ? 0.8 : 1.0) *
         ProcessAge(DateOfBirth[i]);
      }
   }
   ...
}
```

При таком подходе вообще нет сущности «Customer». «Customer» – это просто набор данных с определенным индексом в соответствующих массивах. Эти массивы плотно упакованы релевантными данными, и на критическом участке кода обращение к ним производится последовательно. Строки кеша используются оптимально (рис. 14.3). ЦП может распознать чтение последовательных данных, поэтому устройство предвыборки будет задействовано при каждом доступе к массиву.

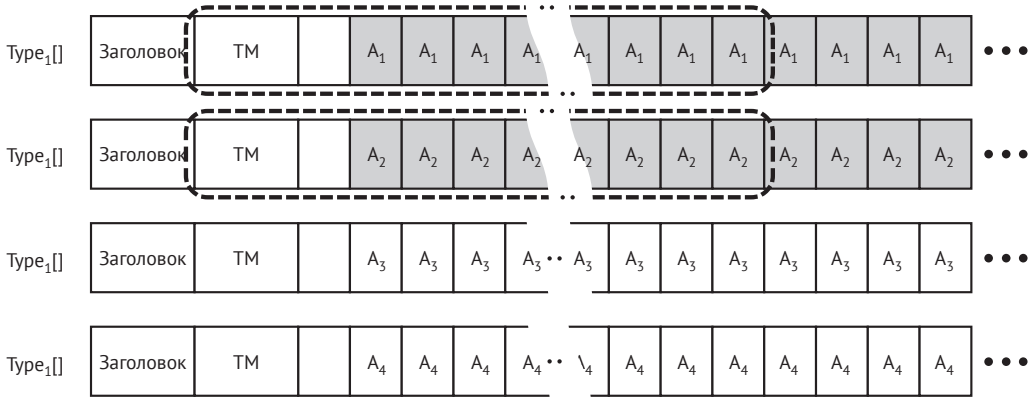


Рис. 14.3 ❖ Оптимальная локальность данных для структуры массивов (нужные данные закрашены серым)

В качестве дополнительного бонуса подход на основе структуры массивов предлагает высокую гибкость. Если в каком-то другом высокопроизводительном алгоритме используются другие поля, то такая организация подойдет и ему.

Аналогично можно линеаризовать иерархические (древовидные) данные. Обычно в каждом узле хранится список его потомков. Понятно, что обход такого дерева может стоить дорого из-за непопадания в кеш, поскольку узлы разбросаны по всей управляемой куче.

Рассмотрим пример дерева в листинге 14.51, в котором реализован простой демонстрационный алгоритм: метод `Process` заменяет значение в каждом узле суммой значений в его предках[1].

Листинг 14.51 ❖ Реализация простого дерева узлов

```
public class Node
{
   public int Value { get; set; }
   public List<Node> Children = new List<Node>();
   public Node(int value) => Value = value;
   public void AddChild(Node child) => Children.Add(child);

   public void Process()
   {
      InternalProcess(null);
   }

   private void InternalProcess(Node parent)
   {
      if (parent != null)
         this.Value = this.Value + parent.Value; // здесь может быть что-то более сложное

      foreach (var child in Children)
      {
         child.InternalProcess(this);
      }
   }
}
```

Но такое дерево можно описать совершенно по-другому, с помощью массива узлов, где в каждом элементе, представляющем узел, хранится ссылка на родителя (а еще лучше – его индекс). Подобный подход наверняка потребует предварительного преобразования начального, более естественного, объектно-ориентированного дерева в массив. Но затем такое дерево можно обрабатывать линейно (листинг 14.52).

Листинг 14.52 ❖ Пример линеаризованного дерева, представленного в виде массива узлов типов значений

```
public class Tree
{
   public struct ValueNode
   {
```

[1] Заметим, что простой алгоритм выбран для краткости, но принципиально от этого ничего не меняется.

```
      public int Value;
      public int Parent;
   }

   private ValueNode[] nodes;

   private static Tree PrecalculateFromRoot(OOP.Node root)
   {
      // Линеаризовать дерево, обойдя его в прямом порядке
      ...
   }

   public void Process()
   {
      for (int i = 1; i < nodes.Length; ++i)
      {
         ref var node = ref nodes[i];
         node.Value = node.Value + nodes[node.Parent].Value;
      }
   }
}
```

Будьте осторожны, проектируя линеаризацию дерева. Пример в листинге 14.52 работает, потому что алгоритм обработки (сложение значений в методе `Process`) зависит только от значений в родителях, поэтому обход дерева в прямом порядке идеален. После такой линеаризации каждый элемент в массиве `nodes` расположен после уже обработанного родителя. Если бы алгоритм зависел от дочерних узлов (например, нужно было заменить значение в узле суммой значений во всех потомках), то следовало бы обходить дерево в обратном порядке, тогда каждый элемент в линеаризованном массиве располагался бы после всех своих дочерних элементов.

Сущность, компонент, система (Entity Component System)

В объектно-ориентированном программировании наследование и инкапсуляция – основные понятия. В сложных приложениях дерево наследования может оказаться весьма запутанным, и многие объекты будут разделять часть возможных видов поведения. Хороший пример – игры, где есть десятки типов, которые представляют по-разному ведущие себя сущности. Например, танки – это вооруженные транспортные средства, а грузовики – тоже транспортные средства, но не вооруженные, а играющие роль контейнера. Обычный солдат обладает способностью к передвижению, и у него есть такие атрибуты, как здоровье, но не всегда вооружение. На рис. 14.4 показан пример дерева наследования.

В более широком контексте разработки ПО такое дерево наследования может стать неудобным, потому что добавить новую сущность, разделяющую с другими только часть поведения, нетривиально – ее нужно куда-то включить, переопределить некоторые методы, чтобы описать новое поведение, и т. д. (например, подумайте о добавлении класса `MagicTree` в дерево на рис. 4.4; эта сущность должна быть «позиционируемой» и живой, но не способной к передвижению).

В контексте, ориентированном на данные, недостатки такого подхода бросаются в глаза – данные разбросаны по всей древовидной иерархии. В ООП совершенно нормально, когда несколько бизнес-объектов взаимодействуют друг с другом. Но если требуется обработать тысячи или миллионы похожих сущно-

стей, скажем, транспортных средств, чтобы обновить их позиции, такая организация становится проблемой.

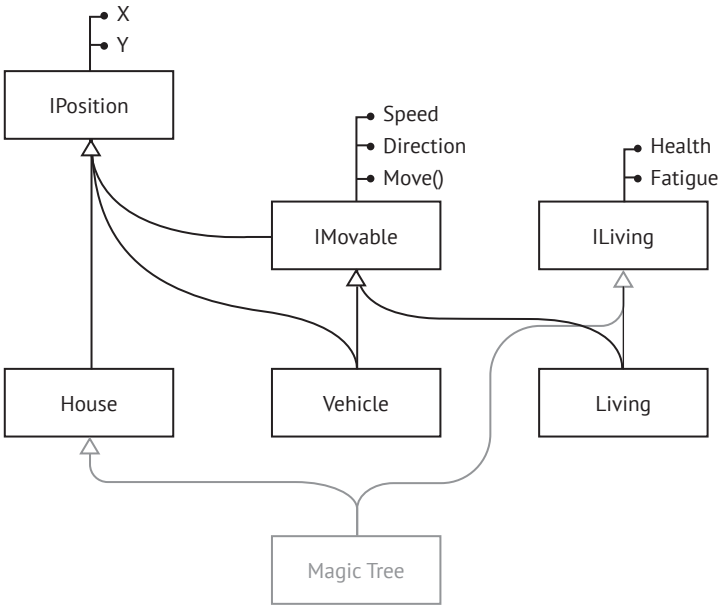


Рис. 14.4 ❖ Пример дерева наследования, представляющего некоторые объекты игры

Мы могли бы воспользоваться архитектурой структуры массивов и вести отдельные списки зданий, транспортных средств, живых существ и т. д. Но это не очень практично, потому что многим алгоритмам все равно требуется доступ к различным наборам свойств, хранящимся в разных списках (что сводит на нет все достигнутые преимущества локальности данных).

Решение проблемы предложено в виде шаблона проектирования «*Система, компонент, сущность*» (Entity Component System – ECS). Проще говоря, следует отдавать предпочтение композиции перед наследованием. Как мы скоро увидим, в основе этого решения лежит, в частности, совместимость хорошей локальности данных с идеей структуры массивов.

В этом шаблоне нет типов, представляющих здание, транспортное средство или живое существо. Сущности компонуются динамически путем добавления и удаления компонентов, представляющих возможности. Затем эти сущности обрабатываются различными системами, представляющими нужную логику. Иными словами, в ECS есть три основных блока (рис. 14.5):

- *сущность* – простой объект с идентификатором, который не содержит ни данных, ни логики. Путем добавления к нему конкретных компонентов мы определяем возможности сущности. Например, если в игре нам понадобится что-то вроде транспортного средства, мы создадим сущность и назначим ей подходящие компоненты (в нашем упрощенном примере – компоненты Position и Movable);

- *компонент* – простой объект, содержащий только данные, но не логику. Эти данные необходимы для представления текущего состояния возможности представленным компонентом (например, положения для компонента Position или скорости для компонента Movable);
- *система* – место, где находится логика конкретных возможностей. Системы оперируют отфильтрованными списками сущностей, обрабатывая их поочередно. Например, система Move оставляет только сущности, у которых есть компоненты Position и Movable (и она знает, как обработать или преобразовать свойства этих компонентов).

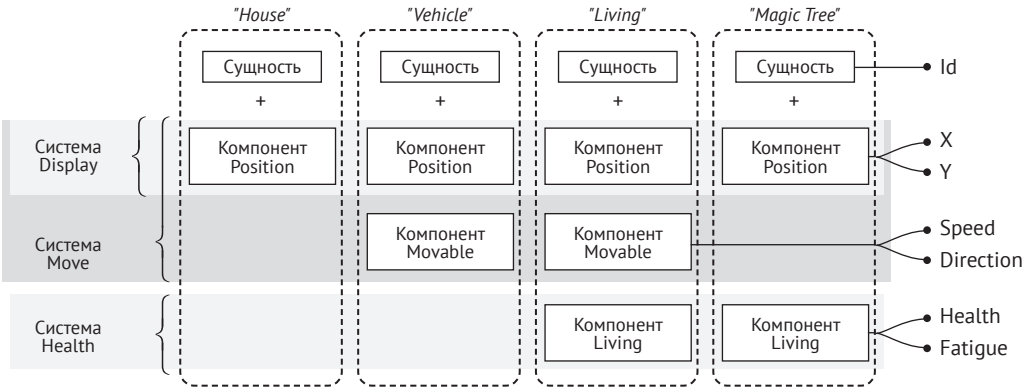


Рис. 14.5 ❖ Общий вид шаблона «Система, компонент, сущность»

В главном цикле игры системы исполняются одна за другой. Надеюсь, вы уже понимаете, в чем прелесть такого подхода. Данные каждого компонента хранятся последовательно и отдельно от других в соответствии с принципом структуры массивов. Например, когда система Display перебирает сущности, она по существу обходит последовательную коллекцию данных компонента Position. Очевидно, что для этого необходима очень эффективная техника фильтрации сущностей (ответа на вопрос, есть ли у сущности данный компонент). Но это детали реализации, которые мы здесь затрагивать не будем. А вместо этого реализуем самый простой шаблон ECS, который только можно себе представить. Надеюсь, это позволит лучше проиллюстрировать идею в целом.

Прежде всего сущность может быть совсем простым типом, содержащим лишь идентификатор (листинг 14.53). Это структура только для чтения – чтобы можно было организовать плотное хранение в массиве сущностей и избежать защитного копирования при передаче в качестве аргумента.

Листинг 14.53 ❖ Определение сущности

```
public readonly struct Entity
{
   public readonly long Id;
   public Entity(long id)
   {
      Id = id;
   }
}
```

Компоненты – также всего лишь простые контейнеры данных. И опять-таки для плотного хранения в массиве они сделаны структурами (листинг 14.54). Эти структуры изменяемые, и мы сможем возвращать ссылки на них в соответствующем хранилище для модификации.

Листинг 14.54 ❖ Определения компонентов

```
public struct PositionComponent
{
   public double X;
   public double Y;
}

public struct MovableComponent
{
   public double Speed;
   public double Direction;
}

public struct LivingComponent
{
   public double Fatigue;
}
```

Для эффективного хранения данных компонентов, используя при этом подход с ориентацией на данные, введем класс `ComponentManager<T>` (листинг 14.55). Его основу составляет массив `registeredComponents` компонентов данного типа. Для регистрации достаточно заполнить следующую свободную позицию в массиве (для краткости я опускаю проблему отмены регистрации и образующейся в результате фрагментации). Проверка того, что сущности (определяемой своим идентификатором `Id`) придан некоторый компонент, основана на использовании дополнительного словаря – это не самое эффективное решение, но мы остановились на нем для краткости (и при этом оставили без внимания вопросы многопоточного доступа). Если компонент есть, то соответствующий элемент массива возвращается по ссылке, безо всякого копирования.

Листинг 14.55 ❖ Класс `ComponentManager<T>`, управляющий компонентами

```
public class ComponentManager<T>
{
   private static T Nothing = default;
   private static int registeredComponentsCount = 0;
   private static T[] registeredComponents = ArrayPool<T>.Shared.Rent(128);
   private static Dictionary<long, int> entityIdtoComponentIndex =
      new Dictionary<long, int>();

   public static void Register(in Entity entity, in T initialValue)
   {
      registeredComponents[registeredComponentsCount] = initialValue;
      entityIdtoComponentIndex.Add(entity.Id, registeredComponentsCount);
      registeredComponentsCount++;
   }

   public static ref T TryGetRegistered(in Entity entity)
   {
```

```
      if (entityIdtoComponentIndex.TryGetValue(entity.Id, out int index))
      {
         //result = true;
         return ref registeredComponents[index];
      }
      //result = false;
      return ref Nothing;
   }
}
```

Далее нам нужно абстрактное представление системы (листинг 14.56) и менеджер, который связывает все воедино (листинг 14.57).

Листинг 14.56 ❖ Определение абстрактного базового класса системы

```
public abstract class SystemBase
{
   public abstract void Update(List<Entity> entities);
}
```

Листинг 14.57 ❖ Менеджер, хранящий список сущностей и систем

```
public class Manager
{
   private List<Entity> entities = new List<Entity>();
   private List<SystemBase> systems = new List<SystemBase>();

   public void RegisterSystem(SystemBase system)
   {
      systems.Add(system);
   }

   public Entity CreateEntity()
   {
      var entity = new Entity(entities.Count);
      entities.Add(entity);
      return entity;
   }

   public void Update()
   {
      foreach (var system in systems)
      {
         system.Update(entities);
      }
   }
}
```

Располагая всеми элементами, мы можем написать пример системы. Для системы `MoveSystem` нужно, чтобы сущностям соответствовали компоненты `Position` и `Movable`, поэтому ее метод `Update` производит соответствующую фильтрацию (листинг 14.58). Здесь отчетливо видна необходимость в очень эффективном способе фильтрации сущностей. Впрочем, при правильном управлении доступ к компонентам с данными с высокой вероятностью выполняется последовательно, что обеспечивает хорошую локальность данных и возможность предвыборки.

Листинг 14.58 ❖ Пример системы `Moving`

```
public class MoveSystem : SystemBase
{
   public override void Update(List<Entity> entities)
   {
      foreach (var entity in entities)
      {
         bool hasPosition = false;
         bool isMovable = false;
         ref var position = ref ComponentManager<PositionComponent>.
            TryGetRegistered(in entity, out hasPosition);
         ref var movable = ref ComponentManager<MovableComponent>.
            TryGetRegistered(in entity, out isMovable);
         if (hasPosition && isMovable)
         {
            position.X += CalculateDX(movable.Speed, movable.Direction);
            position.Y += CalculateDY(movable.Speed, movable.Direction);
         }
      }
   }
}
```

Заметим, что приведенная реализация во многих местах чрезмерно упрощена. Как уже отмечалось, нет никакой синхронизации потоков, и предложенный подход к управлению связи между сущностями и компонентами тривиален. Но описание полной практически применимой реализации выходит далеко за рамки этой книги. В библиотеках вроде Entitas (https://github.com/sschmid/Entitas-CSharp) Саймона Шмита (Simon Schmid) или недавно переписанной Entity Component System в Unity эти вопросы проработаны и реализованы гораздо лучше. Например, система чаще всего не занимается фильтрацией самостоятельно, а получает динамически управляемый, уже отфильтрованный список сущностей (который автоматически обновляется при добавлении или удалении компонентов). Предложенный API также далек от совершенства. Ко всему прочему, зрелая реализация ECS должна поддерживать взаимодействие и связи между системами (с помощью какой-нибудь системы обмена сообщениями), а этот аспект мы полностью проигнорировали.

Шаблон «Система, компонент, сущность» очень популярен в разработке игр, но я полагаю, что его применение оправдано и в высокопроизводительных сценариях, когда проектирование, ориентированное на данные, имеет смысл. У вас на руках куча разных «сущностей» с различными характеристиками, которые нужно обрабатывать огромными порциями? Не пора ли задуматься о ECS?

Еще немного о будущем...

В этом разделе приводится перечень возможностей, которые можно было бы включить в любую другую часть этой (и предыдущей) главы, поскольку они носят очень общий характер. Я решил собрать их в разделе о «будущем», потому что на момент написания книги их планировалось включить в более-менее отдаленные выпуски .NET. Возможно, когда вы читаете данный текст, некоторые или даже большинство из них уже готовы и даже заняли прочное место в экосистеме .NET.

С другой стороны, глядя на восприятие других новых типов (например, уже доступного Span<T>), закрадывается мысль, что пройдет еще несколько лет, прежде чем знание о них широко распространится в среде программистов.

Ссылочные типы, допускающие null

Ссылочные типы, допускающие null (nullable reference types), могут быть включены в C# 8.0, но это не точно. Хотя с управлением памятью они напрямую не связаны – их использование не сказывается ни на производительности, ни на потреблении памяти, – это изменение настолько важно для безопасной работы с памятью вообще, что никакая книга о памяти в .NET не может пройти мимо него.

Что касается null, то просто необходимо процитировать британского ученого Тони Хоара, который придумал нулевую ссылку при проектировании языка ALGOL. В 2009 году он извинялся за это:

> *Я называю это своей ошибкой ценой в миллиард долларов. Речь идет об изобретении нулевой ссылки в 1965 году. В то время я проектировал первую полную систему типов для ссылок в объектно-ориентированном языке (ALGOL W). Я хотел гарантировать, что любое использование ссылок будет безопасно, поскольку проверка автоматически выполнялась компилятором. Но я не смог воспротивиться искушению включить нулевую ссылку, просто потому, что реализовать ее было очень просто. Это привело к неисчислимым ошибкам, уязвимостям и крахам систем, что, вероятно, стало причиной миллиардных убытков и страданий в последующие сорок лет.*

Действительно ли null стоит считать ошибкой ценой в миллиард долларов? Можете ли вы представить себе мир C# и .NET, в котором нет `null` и исключения `NullReferenceException`? Вообще говоря, трудно вообразить язык, в котором нет понятия «ничто». Некоторые значения опциональны, потому что так они определены в своей предметной области (например, отчество). Настоящая проблема состоит в отсутствии ясно высказанного утверждения о том, что в определенном контексте «ничто» имеет смысл и потому разрешено (поскольку `null` всегда разрешен по умолчанию).

В некоторых языках, особенно функциональных, допускающие значение null типы заменены вариантным типом (option type) – полиморфным типом, представляющим опциональное значение (т. е. можно представить как «ничто», так и значение). Например, в F# есть тип `Option`, определенный как размеченное объединение (discriminated union) двух случаев: `Some` (содержит значение) и `None`. Употребление такого вариантного типа явно говорит, что возможно значение «ничто». Программист должен выполнить проверку, прежде чем обращаться к значению такого типа (или, по крайней мере, компилятор должен удостовериться, что такая проверка была).

В идеале для ссылочных типов в C# должна быть возможность задавать такую «факультативную допустимость null» вместо безусловной. Чтобы прояснить намерение относительно допустимости null, предлагается ввести два новых безопасных ссылочных типа:

- *ссылочный тип, допускающий null,* – такому типу можно присвоить значение null, поэтому перед разыменованием необходима проверка на null

(и компилятор C# может сделать такую проверку обязательной). Отметим, что этот тип отличается от нынешнего ссылочного типа, который хотя и допускает null всегда, не защищен от разыменования компилятором. Такие типы представляют факультативные значения, как тип `Option` в F#;

- *ссылочный тип, не допускающий null,* – он никогда не может принимать значение null, поэтому разыменование всегда безопасно.

Конечно, вводить эти типы следует осмотрительно – так чтобы они помогали находить ошибки в существующем коде, но не требовали все переписывать. Чтобы в существующем коде от новых типов была какая-то польза, текущие ссылочные типы должны взять на себя одну из ролей (а не вводить, к примеру, два новых вида ссылочных типов в дополнение к существующему). Было решено, что нынешний ссылочный тип будет трактоваться как тип, не допускающий null. По словам Мэдса Торгерсена, выступающего от имени всей команды разработчиков C#, это объясняется следующими причинами:

- они полагают, что ссылочные типы, действительно требующие значения null, встречаются гораздо реже, чем мы думаем;
- в языке C# уже имеется синтаксическая конструкция ? для типов значений, допускающих значение null, поэтому кажется естественным обобщить ее на ссылочные типы;
- кажется правильным, что именно способность принимать значение null должна быть выражено явно, а не наоборот.

Таким образом, в какой-то из будущих версий C# будут добавлены ссылочные типы, допускающие null (с синтаксисом ?), а нынешние ссылочные типы будут преобразованы в типы, не допускающие null (листинг 14.59). Именно поэтому это языковое средство официально называется «ссылочные типы, допускающие null», хотя следует помнить, что на самом деле оба вида предполагаемого поведения ссылочных типов новые.

Листинг 14.59 ❖ Пример класса, содержащего поля ссылочного типа, не допускающие null (по умолчанию) и допускающие null (путем явного обозначения)

```
public class SomeClass
{
   public int Field;
   public OtherClass? NullableReference;   // может быть равен null
   public OtherClass NonNullableReference; // не может быть равен null
}

public class OtherClass
{
   public int OtherField;
}
```

Ясно, что такое изменение может привести к многочисленным ошибкам при компиляции существующего кода. Но это осознанное решение, поскольку новые типы вводятся прежде всего для того, чтобы помочь в поиске ошибок, связанных с использованием null. Чтобы не парализовать всю работу, было решено, что проблемы, связанные с null, будут считаться предупреждениями, а не ошибками (хотя можно явно указать режим, в котором они считаются ошибками).

После этого нововведения компилятор C# будет делать все возможное для проверки значений null, особенно по отношению к локальным переменным и параметрам (листинг 14.60). При попытке обратиться к объекту, который может принимать значение null, без проверки (как в первой строке листинга 14.60) выдается предупреждение. То же самое происходит, когда компилятор обнаруживает обращение к null (как в последней строке листинга 14.60). Анализируется также порядок выполнения программы (например, условные предложения и циклы), как видно в том же листинге 14.60.

Листинг 14.60 ❖ Поведение компилятора в случае аргумента ссылочного типа, допускающего null

```
public static void UseNullableReference(SomeClass? obj)
{
   Console.WriteLine(obj.Field);  // Предупреждение CS8602: Possible dereference
                                  // of a null reference.
   Console.WriteLine(obj?.Field); // Ok, проверяется
   if (obj == null)
      return;
   Console.WriteLine(obj.Field);  // Ok, проверено выше
   obj = null;
   Console.WriteLine(obj.Field);  // Предупреждение CS8602: Possible dereference
                                  // of a null reference.
}
```

Однако остается проблема: насколько глубоко следует заглядывать при проверке нарушений правил, касающихся допустимости null? В настоящее время вызовы методов игнорируются, поскольку они могут содержать сколь угодно сложную логику. Так что даже если внутри метода `ArgumentsValid` есть проверка на `null` (как в листинге 14.61), предупреждение все равно выдается.

Листинг 14.61 ❖ Поведение компилятора в случае аргумента ссылочного типа, допускающего null

```
public static void UseChainedNullableReference(SomeClass? obj)
{
   if (!ArgumentsValid(obj))
      return;
   Console.WriteLine(obj.Reference.OtherField);
}
```

С другой стороны, обращение к ссылочным полям, не допускающим null, гораздо безопаснее, поэтому компилятор будет выдавать меньше ошибок (листинг 14.62).

Листинг 14.62 ❖ Поведение компилятора в случае аргумента ссылочного типа, не допускающего null

```
public static void UseNonNullableReference(SomeClass obj)
{
   Console.WriteLine(obj.Field);   // Ok
   Console.WriteLine(obj?.Field);  // Ok, проверяется
   if (obj == null)
```

```
      return;
   Console.WriteLine(obj.Field);  // Ok, проверено выше
   obj = null;                    // предупреждение CS8600: Converting null literal or
                                  // possible null value to non-nullable type.
   Console.WriteLine(obj.Field);  // предупреждение CS8602: Possible dereference
                                  // of a null reference.
}
```

Предупреждение CS8600 может показаться удивительным, поскольку оказывается, что мы все-таки можем присвоить значение null ссылочному типу, не допускающему null! Дело в том, что все еще существует много случаев, когда это необходимо (и в большинстве из них выдается соответствующее предупреждение), например явное присваивание null, как в листинге 14.62, или присваивание значения типа, допускающего null, объекту типа, не допускающего null. Но есть еще одно важное исключение, в котором решено не выдавать никакого предупреждения, – создание массива (листинг 14.63). В случае массива объектов типа, не допускающего null, компилятор должен был бы настаивать на инициализации всех его элементов, но тогда многие существующие программы перестали бы работать. Объявления массива, подобные показанному в листинге 14.63, встречаются очень часто, поэтому даже появление предупреждения привело бы к невообразимому количеству сообщений в процессе компиляции.

Листинг 14.63 ❖ Поведение компилятора в случае массива ссылочного типа, не допускающего null

```
SomeClass[] array = new SomeClass[4];
UseNonNullableReference(array[1]);    // Ok, предупреждение не выдается
```

> Отметим, что на момент написания книги ссылочные типы, допускающие null, находились в предвыпускной стадии (планировалось включить их в версию C# 8.0, но это не было подтверждено). В этом разделе описан возможный дизайн и порядок их использования, чтобы вы могли составить представление о том, зачем это нужно и как работает. Но уточняйте текущее состояние дел в официальной документации .NET.

И кстати, что такое null? Вообще говоря, это представление адреса, который никогда не должен встречаться в нормальном коде и потому отличается от любого допустимого указателя (в случае .NET-ссылки). Во всех популярных средах программирования значение этого адреса равно 0, поскольку как минимум первая страница ОС всегда остается свободной (неиспользуемой), так что этот адрес заведомо недопустим. Такое значение удобно и потому, что по умолчанию ссылки и указатели становятся равными null в обнуляемых областях памяти (например, ссылочные поля объекта).

При доступе к недействительной странице (в т. ч. первой) ОС вызывает исключения, которые затем обрабатываются CLR. В результате при попытке доступа к первой странице (обычно первым 64 КБ) исключение ОС преобразуется в хорошо известное исключение `NullReferenceException`. А при попытке неразрешенного доступа к старшим адресам мы получим исключение `AccessViolationException`. Например, если в программе на C# попробовать разыменовать неуправляемый нулевой указатель, мы получим исключение `NullReferenceException` (листинг 14.64).

Листинг 14.64 ❖ Пример небезопасного кода, вызывающего исключение NullReferenceException

```
unsafe { int read = *((int*)IntPtr.Zero); }
```

Если же мы попытаемся обратиться к адресу, большему 64 КБ, то возникнет исключение AccessViolationException (листинг 14.65).

Листинг 14.65 ❖ Пример небезопасного кода, вызывающего исключение AccessViolationException

```
unsafe { int read = *((int*)0x1_0000 + 1); }
```

Чаще всего исключение NullReferenceException встречается в обычном коде на C# при попытке доступа к полю объекта, равного null (листинг 14.66). Но обрабатывается этот случай точно так же, потому что доступ к полю объекта – не что иное, как разыменование адреса объекта, к которому прибавлено небольшое смещение поля (листинг 14.67). В нашем случае, если аргумент-ссылка, переданный в регистре rcx, равен 0, адрес поля будет равен 0x8 (в предположении, что Field – первое поле класса SomeClass). Попытка обратиться к адресу 0x8 все равно приводит к исключению NullReferenceException, потому что этот адрес принадлежит первой странице.

Листинг 14.66 ❖ Пример управляемого кода, вызывающего исключение NullReferenceException (в предположении, что obj равно null)

```
public static void Test(SomeClass obj)
{
   Console.WriteLine(obj.Field);
}
```

Листинг 14.67 ❖ Ассемблерный код метода Test из листинга 14.66

```
C.Test(SomeClass)
L0000: sub rsp, 0x28
L0004: mov ecx, [rcx+0x8]
L0007: call System.Console.WriteLine(Int32)
L000c: nop
L000d: add rsp, 0x28
L0011: ret
```

Сразу же возникает вопрос: а что, если объект больше размера первой страницы и мы пытаемся обратиться к полю в его конце (по нулевой ссылке)? Не будет ли ошибочно вызвано исключение AccessViolationException вместо NullReferenceException? Нет, не будет. От этого защищает JIT-компилятор, генерирующий соответствующий код. Например, при передаче массива в любом случае вставляется код проверки выхода за границы (при котором происходит обращение к полю размера массива), так что мы получим NullReferenceException еще до доступа к запрошенному элементу. А для гигантского объекта с тысячами полей (листинг 14.68) JIT-компилятор добавит проверку на null для всего объекта до обращения к конкретному полю (листинг 14.69). Вторая ассемблерная команда в листинге 14.69 генерируется только при доступе к полям SomeClass с большими смещениями (если rcx равно 0, то мы увидим исключение NullReferenceException).

Листинг 14.68 ❖ Пример управляемого кода, вызывающего исключение `NullReferenceException` (в предположении, что `obj` равно `null`)

```
public class SomeClass
{
   public long Field0;
   public long Field1;
   public long Field2;
   ...
   public long Field8229;
   public long Field8230;
}

public static void Test(SomeClass obj)
{
   Console.WriteLine(obj.Field8000);
}
```

Листинг 14.69 ❖ Ассемблерный код метода `Test` из листинга 14.68

```
C.Test(SomeClass)
L0000: sub rsp, 0x28
L0004: cmp [rcx], ecx
L0006: mov rcx, [rcx+0xfa08]
L000d: call System.Console.WriteLine(Int64)
L0012: nop
L0013: add rsp, 0x28
L0017: ret
```

> Отметим, что 0 и «первая страница» здесь относятся к адресу в виртуальной памяти. Это означает, что физически «нулевая страница» соответствует какой-то произвольной физической странице.

Конвейеры

Потоки (streams) так же стары, как .NET. Они прекрасно справляются со своей работой, но плохо подходят для написания высокопроизводительного кода. Они могут выделять много памяти и часто копируют данные. Также они приводят к накладным расходам на синхронизацию, если используются в многопоточной программе. Для написания эффективного кода с использованием буферов, подобных потокам, необходимо было придумать что-то новое. Именно так и появились *конвейеры* (поначалу называвшиеся *каналами*), в основном для сетевой потоковой передачи в новом веб-сервере Kestrel, где размещаются веб-приложения. Но хотя Kestrel был причиной появления конвейеров, они будут включены в общую библиотеку.

На момент написания книги ожидалось, что в будущих версиях .NET появится совершенно новый API конвейеров, который можно рассматривать как Stream-подобные буферы, нацеленные на задачи, требующие высокопроизводительного и хорошо масштабируемого кода. При их проектировании использовалась парадигма производитель–потребитель, т. е. есть писатель (отправляющий данные) и читатель (получающий эти данные). Процитируем документацию: «Конвейер похож на поток Stream, который отдает (push) вам данные, вместо того чтобы за-

ставлять вас забирать (pull) их. Один участок кода помещает данные в конвейер, а другой участок ждет, когда в конвейере появятся данные». Скорее всего, эта техника (как и другие, рассмотренные в этой главе) будет интересна разработчикам низкоуровневых библиотек для организации сетевого взаимодействия и написания кода сериализации.

Поскольку конвейеры с самого начала проектировались в расчете на высокую производительность и масштабируемость, они обладают следующими характеристиками:

- память берется из пула внутренних буферов – это позволяет избежать выделения из кучи;
- на уровне API повсеместно используются типы `Span<T>` и `Memory<T>` – это позволяет обходиться вообще без копирования данных (данные вырезаются из внутренних буферов);
- эффективно реализованы асинхронность и потокобезопасность.

Несмотря на сложность внутреннего устройства, API конвейеров очень прост. Первым делом мы должны сконфигурировать экземпляр конвейера, предоставив пул, из которого он будет черпать память (листинг 14.70). Существуют и другие конфигурационные параметры, не рассматриваемые в этой книге, в частности относящиеся к планировщикам конвейеров. Моя цель – лишь кратко описать возможности конвейеров, не углубляясь в детали. Все это очень интересно, но нельзя объять необъятное.

Листинг 14.70 ❖ Пример конфигурации конвейера

```
var pool = MemoryPool<byte>.Shared;
var options = new PipeOptions(pool);
var pipe = new Pipe(options);
```

У созданного конвейера есть два важных свойства: `Writer` и `Reader`. Как они используются, показано в листинге 14.71. Имейте в виду, что чтение и запись в подобном примере могут производиться в разных потоках, и это будет безопасно. Как видим, при использовании конвейеров мы должны явно сбрасывать буферы писателя методом `FlushAsyncs` (чтобы данные стали видны читателям). А читатель должен явно обновить позицию чтения методом `AdvanceTo` (чтобы проинформировать конвейер о том, что данные прочитаны и соответствующие буферы можно освободить).

Листинг 14.71 ❖ Основы использования конвейеров

```
static async Task AsynchronousBasicUsage(Pipe pipe)
{
   // Записать данные
   pipe.Writer.Write(new byte[] { 1, 2, 3 }.AsReadOnlySpan());
   await pipe.Writer.FlushAsync();

   // Прочитать данные
   var result = await pipe.Reader.ReadAsync();
   byte[] data = result.Buffer.ToArray();
   pipe.Reader.AdvanceTo(result.Buffer.End);
   data.Print();
}
```

Хотя использование конвейеров, продемонстрированное в листинге 14.71, и полезно в качестве введения, на самом деле это антипаттерн, потому что:

- писатель должен создать в куче байтовый массив, перед тем как отправлять данные;
- читатель должен создать в куче байтовый массив, куда будут скопированы данные.

Очевидно, что это противоречит проектным целям, перечисленным в начале этого раздела. Чтобы полнее использовать возможности конвейеров, мы можем получить буферизованную память непосредственно от самого конвейера.

Для начала улучшим код писателя (листинг 14.72). Как видим, можно получить буфер в виде `Span<byte>` или `Memory<byte>` прямо от `Writer`, вообще не прибегая к выделению памяти (реализация возвращает срез внутреннего буфера требуемого размера). После модификации данных в полученном `Span<byte>` мы должны явно обновить позицию записи методом `Advance`. Тем самым мы сообщаем конвейеру, сколько байтов записано и должно быть сброшено последующим методом `FlushAsync`.

Листинг 14.72 ❖ Использование конвейеров с буферизованной памятью. Из-за применения `Span<byte>` метод не является асинхронным

```
static void SynchronousGetSpanUsage(Pipe pipe)
{
   Span<byte> span = pipe.Writer.GetSpan(minimumLength: 2);
   span[0] = 1;
   span[1] = 2;
   pipe.Writer.Advance(2);
   pipe.Writer.FlushAsync().GetAwaiter().GetResult();

   var readResult = pipe.Reader.ReadAsync().GetAwaiter().GetResult();
   byte[] data = readResult.Buffer.ToArray();
   pipe.Reader.AdvanceTo(readResult.Buffer.End);
   data.Print();
   pipe.Reader.Complete();
}
```

Концептуально мы должны рассматривать данные, возвращаемые методами `GetSpan` и `GetMemory`, как отдельные блоки, которые будут записаны в конвейер. У этих блоков есть конфигурируемый минимальный размер, по умолчанию равный 2048 байтам. Так что даже если мы зададим в качестве `minimumLength` всего несколько байтов, все равно получим 2 КБ памяти (это не проблема, потому что память берется из внутреннего пула, так что никакого выделения из кучи не требуется). Имейте в виду, что полученный блок, скорее всего, уже использовался и может содержать ранее записанные данные. Поэтому важно, чтобы метод `Advance` честно сообщал, сколько байтов действительно было модифицировано. В листинге 14.73 показаны две последовательные операции записи двух полученных блоков, но `Advance` сообщает о большем количестве байтов, чем было модифицировано на деле. В результате часть данных будет иметь неопределенные значения (в нашем примере – 0).

Листинг 14.73 ❖ Использование конвейеров с буферизованной памятью. Благодаря применению Memory<byte> метод может быть асинхронным

```
static async Task AsynchronousGetMemoryUsage(Pipe pipe)
{
   Memory<byte> memory = pipe.Writer.GetMemory(minimumLength: 2);
   memory.Span[0] = 1;
   memory.Span[1] = 2;
   Console.WriteLine(memory.Length); // печатается 2048
   pipe.Writer.Advance(4);
   await pipe.Writer.FlushAsync();

   Memory<byte> memory2 = pipe.Writer.GetMemory(minimumLength: 2);
   memory2.Span[0] = 3;
   memory2.Span[1] = 4;
   Console.WriteLine(memory.Length);  // печатается 2048
   pipe.Writer.Advance(4);

   await pipe.Writer.FlushAsync();
   //pipe.Writer.Complete(); закрыть конвейер на стороне писателя (чтобы читатель
   // больше не ждал данных)

   var readResult = await pipe.Reader.ReadAsync();
   byte[] data = readResult.Buffer.ToArray();
   pipe.Reader.AdvanceTo(readResult.Buffer.End);
   data.Print(); // 1,2,0,0,3,4,0,0
   //pipe.Reader.Complete(); дальнейшее чтение невозможно
}
```

Чтобы полностью отказаться от копирования на стороне читателя, потребуется чуть больше изменений, но все они интуитивно понятны. Вместо того чтобы копировать абсолютно все данные из readResult.Buffer во вновь созданный массив, мы можем получить к ним доступ и проанализировать без копирования. Свойство Reader.Buffer имеет тип ReadOnlySequence<byte>, который предоставляет следующие возможности:

- последовательность (буфер) представляет один или несколько сегментов, полученных от производителя;
- ее свойство IsSingleSegment сообщает, представляет ли последовательность только один сегмент;
- у свойства First тип ReadOnlyMemory<byte>, и оно возвращает первый сегмент;
- этот тип перечислимый (enumerable) и состоит из элементов типа ReadOnlyMemory<byte>, в случае когда сегментов несколько.

Это подсказывает нам общий способ использования буфера чтения (листинг 14.74). Отметим, что в этом коде нет ни одного выделения памяти – читаемые данные представлены срезами типа ReadOnlyMemory<byte> и ReadOnlySpan<byte>.

В листинге 14.73 показана еще одна особенность конвейера – метод читателя AdvanceTo может за одно обращение обновить две разные позиции чтения:

- использованная позиция нужна, чтобы проинформировать, что данные до этой позиции уже прочитаны (использованы) и больше не понадобятся. Эти данные не будут возвращены при последующих вызовах ReadAsync

(и занятая ими память может быть освобождена внутренним механизмом буферизации);

- просмотренная позиция нужна, чтобы проинформировать, что хотя данные прочитаны до этой позиции (мы их уже видели), этого оказалось недостаточно. Например, мы прочитали только часть входящего сообщения и должны дождаться продолжения. Данные между потребленной и просмотренной позициями будут возвращены при последующих вызовах `ReadAsync` вместе с вновь поступившими данными.

Листинг 14.74 ❖ Пример чтения из конвейера без копирования

```
static async Task Process(Pipe pipe)
{
   PipeReader reader = pipe.Reader;
   var readResult = await pipe.Reader.ReadAsync();
   var readBuffer = readResult.Buffer;
   SequencePosition consumed;
   SequencePosition examined;
   try
   {
      ProcessBuffer(in readBuffer, out consumed, out examined);
   }
   finally
   {
      reader.AdvanceTo(consumed, examined);
   }
}

private static void ProcessBuffer(in ReadOnlySequence<byte> sequence,
   out SequencePosition consumed, out SequencePosition examined)
{
   consumed = sequence.Start;
   examined = sequence.End;
   if (sequence.IsSingleSegment)
   {
      // Использовать буфер в виде одного Span
      var span = sequence.First.Span;
      Consume(in span);
   }
   else
   {
      // Использовать буфер в виде коллекции Span'ов
      foreach (var segment in sequence)
      {
         var span = segment.Span;
         Consume(in span);
      }
   }
   // out consumed - до какой позиции мы уже использовали данные (и больше не нуждаемся
   // в них)
   // out examined - до какой позиции мы уже проанализировали данные (данные между
   // consumed и examined будут возвращены снова, когда поступят новые данные)
}
```

```
private static void Consume(in ReadOnlySpan<byte> span) // Защитная копия не создается,
                                                        // потому что ReadOnlySpan –
                                                        // постоянная структура
{
   //...
}
```

Способ чтения из конвейера без копирования, показанный в листинге 14.74, скорее всего, станет распространенным паттерном проектирования. Например, он уже используется в классе `HttpParser`, входящем в `KestrelHttpServer`, который частично был показан в листинге 14.6 (см. листинг 14.75). Такой анализатор должен разбирать входящие данные строка за строкой. Поэтому паттерн, продемонстрированный в методе `ProcessBuffer`, следует модифицировать так, чтобы он читал данные из входного буфера и искал в них символ перевода строки. Если переход на новую строку найден, то устанавливается использованная позиция. Если нет, данные помечаются только как просмотренные и будут снова проанализированы при поступлении новых.

Листинг 14.75 ❖ Полный код метода `ParseRequestLine` из класса `HttpParser`, являющегося частью KestrelHttpServer

```
public unsafe bool ParseRequestLine(TRequestHandler handler,
   in ReadOnlySequence<byte> buffer, out SequencePosition consumed,
   out SequencePosition examined)
{
   consumed = buffer.Start;
   examined = buffer.End;

   // Подготовить первый Span
   var span = buffer.First.Span;
   var lineIndex = span.IndexOf(ByteLF);
   if (lineIndex >= 0)
   {
      consumed = buffer.GetPosition(lineIndex + 1, consumed);
      span = span.Slice(0, lineIndex + 1);
   }
   else if (buffer.IsSingleSegment)
   {
      // Строка запроса не закончилась
      return false;
   }
   else if (TryGetNewLine(buffer, out var found))
   {
      span = buffer.Slice(consumed, found).ToSpan();
      consumed = found;
   }
   else
   {
      // Строка запроса не закончилась
      return false;
   }

   // Закрепить и разобрать Span
   fixed (byte* data = &MemoryMarshal.GetReference(span))
```

```
   {
      ParseRequestLine(handler, data, span.Length);
   }
   examined = consumed;
   return true;
}

private static bool TryGetNewLine(in ReadOnlySequence<byte> buffer,
   out SequencePosition found)
{
   var byteLfPosition = buffer.PositionOf(ByteLF);
   if (byteLfPosition != null)
   {
      // Продвинуться на следующий за \n байт
      found = buffer.GetPosition(1, byteLfPosition.Value);
      return true;
   }
   found = default;
   return false;
}
```

Анализировать входящие сегменты из буфера чтения утомительно. Мы должны хранить состояние анализа и правильно обрабатывать соседние сегменты (поскольку интересующие нас байтовые данные, скорее всего, находятся в нескольких сегментах). Для типичных сценариев анализа сегментов в виде потока байтов существует вспомогательный класс `BufferReader` (листинг 14.76). Внутри он анализирует последовательные сегменты, а наружу представляет один сплошной поток байтов, доступных методу `Read`. При этом он по-прежнему ничего не выделяет в куче, потому что применяет подход с нулевым копированием.

Листинг 14.76 ❖ Пример использования вспомогательного класса `BufferReader`

```
private static void ProcessWithBufferReader(in ReadOnlySequence<byte> sequence,
   out SequencePosition consumed, out SequencePosition examined)
{
   var byteReader = BufferReader.Create(sequence);
   while (!byteReader.End)
   {
      var ch = byteReader.Read();
      // Использовать... и читать дальше.
      // setting:
      consumed = byteReader.Position;
      examined = byteReader.Position; // или меньше, если использовался Peek
      // выйти, если какая-то часть закончена
   }
}
```

Резюме

В этой главе было рассмотрено много разных тем. Это своего рода мешок, в котором лежат не связанные на первые взгляд технические приемы и типы. Но, по-моему, у них есть одна общая черта – все это продвинутые вещи, которые находят

применение в основном в узкоспециализированном коде, предъявляющем высокие требования к производительности. Именно поэтому глава названа «Продвинутые приемы».

Мы много говорили о типах `Span<T>` и `Memory<T>`, которые позволяют писать очень эффективный код без выделения памяти из кучи, а также другие средства, например класс `Unsafe`.

Наконец, мы заглянули в будущее C# и .NET. Конечно, предсказывать будущее трудно. Поэтому я воздержался от заглядывания слишком далеко вперед. Было описано два средства, наиболее важных с точки зрения управления памятью, – ссылочные типы, допускающие значение null, и конвейеры (надо было бы еще упомянуть UTF8-строки, которые тоже планируется реализовать).

В этой главе нет новых правил. Но если бы мне предложили сформулировать одно общее, то оно звучало бы так: не усложняйте. Я хочу сказать, что большая часть технических приемов, описанных в этой главе, относится только к низкоуровневому коду, находящемуся на так называемом инфраструктурном уровне. Желательно делать его максимально универсальным и помещать в библиотеку или в NuGet-пакет. Не загромождайте бизнес-уровень такими чисто техническими типами, как `Span<T>` или `Memory<T>`. Они точно не принадлежат предметной области, а понятность – один из важнейших факторов в ее моделировании. Типы `Span<T>` и `Memory<T>` лучше оставить для обработки без копирования в тех участках кода, где производительность играет решающую роль.

Глава 15

Интерфейсы прикладного программирования (API)

Это последняя глава. Мы рассмотрели много вопросов, относящихся к управлению памятью в .NET, включая полное описание работы сборщика мусора. Были описаны и другие важные темы, в т. ч. управление ресурсами с помощью финализации и уничтожаемых объектов, различные типы описателей, применение структур и разнообразные диагностические сценарии. Все это сопровождалось практическими рекомендациями. В настоящий момент мы уверенно ориентируемся в вопросах управления памятью; учитывая, что объем знаний очень велик, возврат по меньшей мере к некоторым частям книги был бы вполне понятен и даже желателен.

Так что же осталось? Не так уж много. В этой главе я хочу описать некоторые API, относящиеся к GC. Они доступны из кода разного уровня и предлагают различные уровни гибкости. Я полагаю, что это хорошая тема для завершения книги. Уже примерно понимая, как GC работает, мы можем посмотреть, как можно управлять его работой и измерять ее результаты программно. Начнем с хорошо известного нам класса `GC`, в основном для справки, потому что большую часть его методов мы уже использовали в разных местах книги. Затем опишем средства размещения CLR. И напоследок рассмотрим две замечательные библиотеки, предоставляющие развитые средства диагностики: `ClrMD` и `EventTrace`. И в качестве вишенки на торте скажем несколько слов о том, как создать свой собственный GC и заменить им встроенный.

GC API

Как уже было сказано, в предыдущих главах мы неоднократно пользовались статическими методами статического класса `GC`. Здесь я хочу подвести итоги и описать несколько возможностей, которые еще не упоминались или упоминались, но недостаточно подробно. Я не хочу повторяться, поэтому если примеры использования некоторого метода уже приводились, то я просто сошлюсь на них. Все методы разбиты на функциональные группы, каждой из которых посвящен отдельный раздел. Помимо самого класса `GC`, мы рассмотрим еще несколько методов и типов, хорошо укладывающихся в тему «GC API».

Сведения и статистические данные о сборке мусора

В первую группу входят свойства и методы, сообщающие о внутреннем состоянии GC и памяти.

GC.MaxGeneration

Возвращает максимальное количество поколений, реализованных в текущей версии GC. Полезно, когда нужно перебрать все имеющиеся поколения (чтобы не прописывать их количество в коде), например чтобы вызвать для каждого из них описанный ниже метод `GC.CollectionCount`. Или если мы хотим с помощью метода `GC.GetGeneration` проверить, находится ли уже объект в самом старом поколении (такой пример также будет показан ниже). Отметим, что сейчас значение этого свойства равно 2, потому что старшее поколение 2 и LOH рассматриваются как одно поколение (полная сборка мусора затрагивает то и другое).

GC.CollectionCount(Int32)

Сообщает, сколько раз производилась сборка мусора в указанном поколении с начала работы программы. Номер запрашиваемого поколения должен быть не меньше 0 и не больше значения свойства `GC.MaxGeneration`. Заметим, что счетчик включающий, т. е. если сборка мусора была вызвана для поколения 1, то увеличиваются счетчики для обоих поколений – 0 и 1. Поэтому программа в листинге 15.1 выведет результаты, показанные в листинге 15.2 (счетчик каждого поколения включает счетчики всех более старых поколений).

Листинг 15.1 ❖ Пример использования метода `GC.CollectionCount`

```
GC.Collect(0);
Console.WriteLine($"{GC.CollectionCount(0)} {GC.CollectionCount(1)}
{GC.CollectionCount(2)}");
GC.Collect(1);
Console.WriteLine($"{GC.CollectionCount(0)} {GC.CollectionCount(1)} {
GC.CollectionCount(2)}");
GC.Collect(2);
Console.WriteLine($"{GC.CollectionCount(0)} {GC.CollectionCount(1)}
{GC.CollectionCount(2)}");
```

Листинг 15.2 ❖ Результат работы программы в листинге 15.1

```
1 0 0
2 1 0
3 2 1
```

Этот метод можно использовать для диагностики и протоколирования. Но, пожалуй, чаще всего он применяется для реализации «умного» явного запуска сборки мусора только в том случае, когда она не происходит автоматически (листинг 15.3). Таким образом, код, запускающий GC, будет менее агрессивным. Вспомните приведенное в главе 7 разъяснение по поводу явного вызова GC для общего случая. Мы могли бы использовать подобный код также для того, чтобы периодически проверять счетчик каждого поколения и вовремя заметить, что в данном поколении недавно произошла сборка мусора (это позволило бы создать

некий аналог «обратного вызова» (callback), выполняемого после каждой GC, если частота проверки достаточно велика).

Листинг 15.3 ❖ Явный запуск сборки мусора, при условии что она не была произведена автоматически

```
if (lastGen2CollectionCount == GC.CollectionCount(2))
{
   GC.Collect(2);
}
lastGen2CollectionCount = GC.CollectionCount(2);
```

GC.GetGeneration

Сообщает, какому поколению принадлежит данный объект. Для объектов в управляемой куче возвращает значение от 0 до `GC.MaxGeneration`.

Его можно использовать, например, для создания политики кеширования, учитывающей поколение. Предположим, что требуется создать пул закрепляемых объектов, и хорошо было бы повторно использовать только объекты из самого старого поколения, которые, скорее всего, находятся в сегментах, содержащих только поколение 2. Предполагаемые объекты, закрепленные на короткий период времени, выгоднее закреплять в сегментах поколения 2, потому что существует гораздо меньшая вероятность полного GC в течение этого времени.

Благодаря методу `GC.GetGeneration` мы можем создать такой пул, сохранив список уже «старых» объектов (которые предпочтительнее арендовать у пула) и еще один список более молодых объектов (в предположении, что когда-то они состарятся). Набросок такого пула приведен в листинге 15.4. Если поступит запрос на аренду у пула (методом `Rent`), то сначала проверяется, есть ли доступные старые объекты. Если их нет, то методом `RentYoungObject` проверяется список более молодых объектов. Если и там ничего нет, то с помощью фабричного метода создается новый объект. Когда объект возвращается в пул (методом `Return`), вызывается метод `GC.GetGeneration` для проверки его «возраста», и в зависимости от результата он добавляется в тот или иной список для повторного применения. Также мы используем класс `Gen2GcCallback` (описанный в главе 12), чтобы при каждой полной сборке выполнять некоторое действие для управления обоими списками – перемещать объекты, оказавшиеся в самом старом поколении, из списка молодых в список старых объектов.

Листинг 15.4 ❖ Набросок реализации класса `PinnableObjectPool<T>`, в которой предпочтение отдается объектам из самого старого поколения

```
public class PinnableObjectPool<T> where T : class
{
   private readonly Func<T> factory;
   private ConcurrentStack<T> agedObjects = new ConcurrentStack<T>();
   private ConcurrentStack<T> notAgedObjects = new ConcurrentStack<T>();

   public PinnableObjectPool(Func<T> factory)
   {
      this.factory = factory;
      Gen2GcCallback.Register(Gen2GcCallbackFunc, this);
   }
```

```
   public T Rent()
   {
      if (!agedObjects.TryPop(out T result))
         RentYoungObject(out result);
      return result;
   }

   public void Return(T obj)
   {
      if (GC.GetGeneration(obj) < GC.MaxGeneration)
         notAgedObjects.Push(obj);
      else
         agedObjects.Push(obj);
   }

   private void RentYoungObject(out T result)
   {
      if (!notAgedObjects.TryPop(out result))
      {
         result = factory();
      }
   }

   private static bool Gen2GcCallbackFunc(object targetObj)
   {
      ((PinnableObjectPool<T>)(targetObj)).AgeObjects();
      return true;
   }

   private void AgeObjects()
   {
      List<T> notAgedList = new List<T>();
      foreach (var candidateObject in notAgedObjects)
      {
         if (GC.GetGeneration(candidateObject) == GC.MaxGeneration)
         {
            agedObjects.Push(candidateObject);
         }
         else
         {
            notAgedList.Add(candidateObject);
         }
      }
      notAgedObjects.Clear();
      foreach (var notAgedObject in notAgedList)
      {
         notAgedObjects.Push(notAgedObject);
      }
   }
}
```

Конечно, класс `PinnableObjectPool<T>` для краткости упрощен и не включает таких важных вещей, как вытеснение из кеша и синхронизация потоков (особенно в методе `AgeObjects`).

> В главе 12 упоминался внутренний класс `PinnableBufferCache`, находящийся в базовой библиотеке .NET (CoreFX), это полная реализация пула, подобного представленному в листинге 15.4. Он содержит и вытеснение из кеша, и оптимальный многопоточный доступ, и еще одну оптимизацию, относящуюся к управлению обеими коллекциями объектов. Настоятельно рекомендую найти время и внимательно изучить код этого класса. В нем вы встретите многие вещи, обсуждаемые в данной книге.

Отметим, что если методу `GetGeneration` передан недопустимый объект, то результат следует считать неопределенным (листинг 15.5). Например, текущая реализация в .NET Core в таком случае возвращает 2, предполагая, что если объект не принадлежит эфемерному сегменту, то он должен принадлежать сегменту поколения 2 или LOH.

Листинг 15.5 ❖ Передача недопустимого объекта, созданного в стеке, методу `GC.GetGeneration`

```
UnmanagedStruct us = new UnmanagedStruct { Long1 = 1, Long2 = 2 };
int gen = GC.GetGeneration(Unsafe.As<UnmanagedStruct, object>(ref us));
Console.WriteLine(gen);
```

```
Выводится:
2
```

GC.GetTotalMemory

Возвращает общее количество используемых байтов во всех поколениях без учета фрагментации. Иными словами, это суммарный размер всех управляемых объектов в куче. Сюда включается размер уже недостижимых, мертвых объектов, если предварительно мы явно не запустили сборку мусора[1]. Как отмечалось в главе 12, где этот метод был представлен (листинг 12.9), следует помнить, что если его аргументу `forceFullCollection` присвоено значение `true`, то вызов может оказаться очень затратным. В худшем случае для получения стабильного результата может быть 20 раз запущена полная блокирующая сборка мусора!

Очевидно, что метод `GetTotalMemory` можно использовать для целей диагностики и протоколирования. Он очень популярен в различных модульных тестах и экспериментах. Но для отслеживания выделения памяти во время теста есть лучшая альтернатива – метод `GC.GetAllocatedBytesForCurrentThread`, описанный ниже.

Кроме того, будьте осторожны, используя этот метод для обработки запросов с ограничениями на потребление памяти, например для регулирования количества веб-запросов. Поскольку не учитывается фрагментация и накладные расходы на управление сегментами (например, заблаговременная передача нескольких страниц сегмента), этот показатель не точно отражает общее потребление памяти. В подобных случаях лучше использовать результаты измерения общей занятой памяти, полученные от класса `Process` (или, по крайней мере, соотносить с ними результат `GC.GetTotalMemory`). Простой пример программы «Hello world» в листинге 15.6 иллюстрирует это различие (результаты показаны в листинге 15.7). Объекты в управляемой куче занимают около 600 КБ памяти. Но под частную память

---

[1] Строго говоря, поскольку между явной сборкой мусора и вызовом метода `GetTotalMemory` может произойти все, что угодно (если есть другие работающие потоки), то какие-то объекты могут стать недоступными.

всего процесса отведено около 9 МБ (хотя виртуальная память, очевидно, больше; см. классификацию памяти процесса в главе 2).

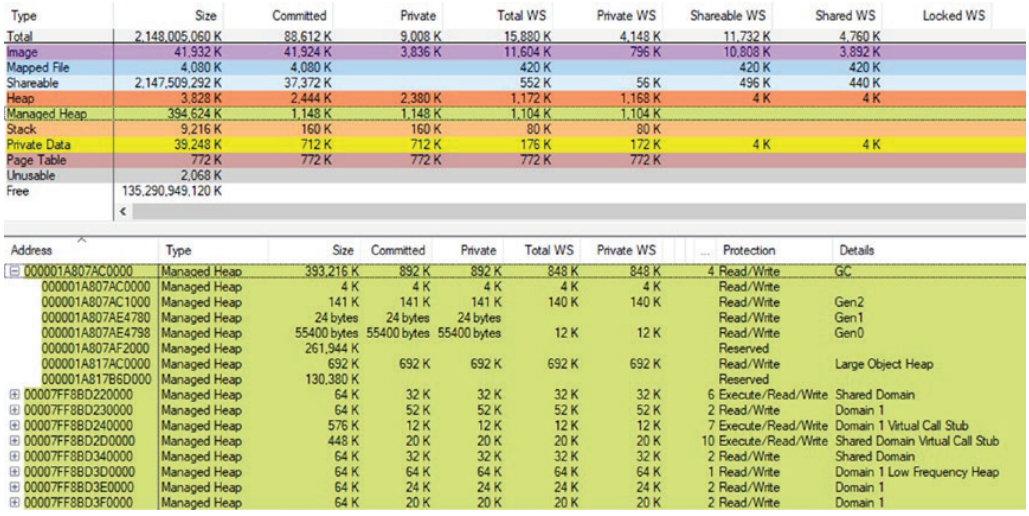
Листинг 15.6 ❖ Метод `GC.GetTotalMemory` и различные измерения памяти, связанной с процессом

```
static void Main(string[] args)
{
   Console.WriteLine("Hello world!");
   var process = Process.GetCurrentProcess();
   Console.WriteLine($"{process.PrivateMemorySize64:N0}");
   Console.WriteLine($"{process.WorkingSet64:N0}");
   Console.WriteLine($"{process.VirtualMemorySize64:N0}");
   Console.WriteLine($"{GC.GetTotalMemory(true):N0}");
   Console.Readline();
}
```

Листинг 15.7 ❖ Результат работы программы в листинге 15.6

```
Hello world!
9,162,752
146,680,064
2,199,553,761,280
620,496
```

Даже память, занятая управляемой кучей, заметно больше суммарного размера находящихся в ней объектов (рис. 15.1). Мы видим, что объем памяти, переданной сегментам GC, равен 1772 КБ, тогда как в листинге 15.7 показано всего около 600 КБ.



Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locked WS
Total	2,148,005,060 K	88,612 K	9,008 K	15,880 K	4,148 K	11,732 K	4,760 K	
Image	41,932 K	41,924 K	3,836 K	11,604 K	796 K	10,808 K	3,892 K	
Mapped File	4,080 K	4,080 K		420 K		420 K	420 K	
Shareable	2,147,509,292 K	37,372 K		552 K	56 K	496 K	440 K	
Heap	3,828 K	2,444 K	2,380 K	1,172 K	1,168 K	4 K	4 K	
Managed Heap	394,624 K	1,148 K	1,148 K	1,104 K	1,104 K			
Stack	9,216 K	160 K	160 K	80 K	80 K			
Private Data	39,248 K	712 K	712 K	176 K	172 K	4 K	4 K	
Page Table	772 K	772 K	772 K	772 K	772 K			
Unusable	2,068 K							
Free	135,290,949,120 K							

Address	Type	Size	Committed	Private	Total WS	Private WS	...	Protection	Details
⊟ 000001A807AC0000	Managed Heap	393,216 K	892 K	892 K	848 K	848 K	4	Read/Write	GC
000001A807AC0000	Managed Heap	4 K	4 K	4 K	4 K	4 K		Read/Write	
000001A807AC1000	Managed Heap	141 K	141 K	141 K	140 K	140 K		Read/Write	Gen2
000001A807AE4780	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen1
000001A807AE4798	Managed Heap	55400 bytes	55400 bytes	55400 bytes	12 K	12 K		Read/Write	Gen0
000001A807AF2000	Managed Heap	261,944 K						Reserved	
000001A817AC0000	Managed Heap	692 K	692 K	692 K	692 K	692 K		Read/Write	Large Object Heap
000001A817B6D000	Managed Heap	130,380 K						Reserved	
⊞ 00007FF8BD220000	Managed Heap	64 K	32 K	32 K	32 K	32 K	6	Execute/Read/Write	Shared Domain
⊞ 00007FF8BD230000	Managed Heap	64 K	52 K	52 K	52 K	52 K	2	Read/Write	Domain 1
⊞ 00007FF8BD240000	Managed Heap	576 K	12 K	12 K	12 K	12 K	7	Execute/Read/Write	Domain 1 Virtual Call Stub
⊞ 00007FF8BD2D0000	Managed Heap	448 K	20 K	20 K	20 K	20 K	10	Execute/Read/Write	Shared Domain Virtual Call Stub
⊞ 00007FF8BD340000	Managed Heap	64 K	32 K	32 K	32 K	32 K	2	Read/Write	Shared Domain
⊞ 00007FF8BD3D0000	Managed Heap	64 K	64 K	64 K	64 K	64 K	1	Read/Write	Domain 1 Low Frequency Heap
⊞ 00007FF8BD3E0000	Managed Heap	64 K	24 K	24 K	24 K	24 K	2	Read/Write	Domain 1
⊞ 00007FF8BD3F0000	Managed Heap	64 K	20 K	20 K	20 K	20 K	2	Read/Write	Domain 1

Рис. 15.1 ❖ Представление программы из листинга 15.6 в VMMAP 6 (остановлена на последней строке)

Различие между двумя числами вызвано в основном фрагментацией, которая не учитывается. Это можно подтвердить, воспользовавшись командой SOS `heap-`

stat в программе WinDbg (листинг 15.8), которая позволяет легко вычислить общий размер свободного пространства.

Листинг 15.8 ❖ Результат работы команды SOS heapstat для программы из листинга 15.6

```
> !heapstat -inclUnrooted
Heap     Gen0     Gen1     Gen2     LOH
Heap0    8216       24   145280   701024
Free space:                                Percentage
Heap0      24        0    94576   131280   SOH: 61% LOH: 18%
Unrooted objects:                          Percentage
Heap0      40        0      184        0   SOH: 0% LOH: 0%
```

К сожалению, чтобы получить самое интересное значение – Рабочий набор, частный (Working Set - Private), – необходимо использовать класс PerformanceCounter и читать счетчики производительности своего процесса. Кроме того, не существует способа программно получить полный размер управляемой кучи с учетом фрагментации, кроме как с применением библиотеки на базе ClrMD или ETW, которая будет описана далее в этой главе. Существует еще внутренний метод GC.GetMemoryInfo, возвращающий такую информацию, который был добавлен в .NET Core 2.1, но на момент написания книги он не был открытым.

GC.GetAllocatedBytesForCurrentThread

Этот метод возвращает общее количество байтов, выделенных к настоящему моменту в текущем потоке. Заметим, что это кумулятивная величина, которая постоянно растет. Учитываются только операции выделения, и не имеет значения, сколько объектов (байтов) было затем убрано в мусор.

Поскольку метод возвращает только значение для текущего потока, с его помощью невозможно узнать о выделении памяти в другом потоке. Поэтому его реализация быстрая и простая (листинг 15.9): складывается количество байтов, выделенных до сих пор в предыдущих контекстах выделения памяти, и уже потребленная часть текущего контекста выделения памяти (контекст выделения памяти подробно описан в главе 5).

Листинг 15.9 ❖ Реализация метода GC.GetAllocatedBytesForCurrentThread в CoreCLR

```
FCIMPL0(INT64, GCInterface::GetAllocatedBytesForCurrentThread)
{
   ...
   INT64 currentAllocated = 0;
   Thread *pThread = GetThread();
   gc_alloc_context* ac = pThread->GetAllocContext();
   currentAllocated = ac->alloc_bytes + ac->alloc_bytes_loh -
                      (ac->alloc_limit - ac->alloc_ptr);
   return currentAllocated;
}
FCIMPLEND
```

Поскольку подсчет выделенной памяти ограничен только текущим потоком, метод GC.GetAllocatedBytesForCurrentThread больше подходит для изолированных модульных тестов или для экспериментов с выделением памяти, где его можно

использовать вместо GC.GetTotalMemory (листинг 15.10). Заметим, что последний возвращает размер памяти, выделенной всему процессу, так что на результат влияют и иные потоки, выделяющие память. С другой стороны, если поток только один, то этот метод дает чистые и воспроизводимые результаты.

Листинг 15.10 ❖ Пример использования метода GC.GetAllocatedBytesForCurrentThread в модульном тесте

```
[Fact]
public void SampleTest()
{
   string input = "Hello world!";
   var startAllocations = GC.GetAllocatedBytesForCurrentThread();
   ReadOnlySpan<char> span = input.AsSpan().Slice(0, 5);
   var endAllocations = GC.GetAllocatedBytesForCurrentThread();
   Assert.Equal(startAllocations, endAllocations);
   Assert.Equal("Hello", span.ToString());
}
```

Учтите, что этот метод добавлен в .NET Core 2.1 и в .NET Framework пока отсутствует. Но .NET Framework предоставляет еще один способ программного измерения памяти с помощью класса AppDomain и двух его свойств[1]:

- MonitoringTotalAllocatedMemorySize – возвращает общее количество байтов, выделенных доменом приложения к данному моменту. В этом смысле свойство похоже на метод GC.GetAllocatedBytesForCurrentThread, но работает на уровне AppDomain, а не потока. Кроме того, оно обновляется при каждой смене контекста выделения памяти (а это происходит чаще, чем сборка мусора). Поэтому точность такая, как размер контекста выделения памяти, т. е. несколько килобайтов;
- MonitoringSurvivedMemorySize – возвращает общее количество байтов, занятых объектами, пережившими последнюю сборку мусора. Точность гарантируется только после полной сборки мусора, хотя обновляется чаще, но точность меньше.

Несовпадение результатов измерения, полученных различными методами, вызывает трудности при написании кода, совместимого с .NET Standard, который мог бы работать и в .NET Core, и в .NET Framework. В библиотеке BenchmarkDotNet эта проблема решается путем использования наилучшего из возможных (самого точного) метода в каждом случае (листинг 15.11).

Листинг 15.11 ❖ Фрагменты класса GcStats из библиотеки BenchmarkDotNet, используемого в классе MemoryDiagnoser

```
public struct GcStats
{
   private static readonly Func<long> GetAllocatedBytesForCurrentThreadDelegate =
      GetAllocatedBytesForCurrentThread();

   private static Func<long> GetAllocatedBytesForCurrentThread()
```

[1] Для использования этих свойств необходимо включить мониторинг ресурсов в домене приложения. Как это сделать, см. в MSDN.

```
   {
      // в одних версиях .NET Core этот метод внутренний, в других – открытый,
      // а в третьих открытый, и его можно использовать ;)
      var method = typeof(GC).GetTypeInfo().GetMethod("GetAllocatedBytesForCurrentThread",
       BindingFlags.Public | BindingFlags.Static) ??
         typeof(GC).GetTypeInfo().GetMethod("GetAllocatedBytesForCurrentThread",
       BindingFlags.NonPublic | BindingFlags.Static);
      return () => (long)method.Invoke(null, null);
   }

   private static long GetAllocatedBytes()
   {
      ...
      // "Это свойство экземпляра типа Int64 возвращает количество байтов, выделенных
      // конкретным AppDomain. Значение точно на момент последней сборки мусора".
      // – CLR via C#
      // Поэтому мы принудительно запускаем GC.Collect, чтобы получить точные
      // результаты.
      GC.Collect();
#if CLASSIC
      return AppDomain.CurrentDomain.MonitoringTotalAllocatedMemorySize;
#elif NETSTANDARD2_0
      ...
      // https://apisof.net/catalog/System.GC.GetAllocatedBytesForCurrentThread()
      // не является частью .NET Standard, поэтому для вызова используется
      // отражение (reflection).
      return GetAllocatedBytesForCurrentThreadDelegate.Invoke();
#elif NETCOREAPP2_1
      // но CoreRT пока не поддерживает отражение, так что только по этой причине
      // мы вынуждены ориентироваться на .NET Core 2.1, чтобы иметь возможность
      // вызвать этот метод без отражения и поддержать MemoryDiagnoser для CoreRT ;)
      return System.GC.GetAllocatedBytesForCurrentThread();
#endif
   }
   ...
}
```

GC.KeepAlive

Метод `GC.KeepAlive` продлевает жизнь стековому корню, поскольку он оставляет переданный аргумент доступным по крайней мере до строки, в которой этот метод вызван (что влияет на сгенерированную информацию для GC). Применение этого метода обсуждается в главе 8 (листинги 8.16 и 8.17). Он встречался и еще в нескольких примерах в разных частях книги.

GCSettings.LargeObjectHeapCompactionMode

Присвоив этому свойству значение `GCLargeObjectHeapCompactionMode.CompactOnce`, мы сможем явно запросить уплотнение LOH во время первой блокирующей полной сборки мусора. Использование этого метода и его влияние на производительность подробно описаны в сценарии 10.1 «Фрагментация кучи больших объектов» в главе 10.

GCSettings.LatencyMode

Это свойство позволяет управлять режимом задержки GC, а значит, и степенью его конкурентности. Оно также включает дополнительные режимы `LowLatency` и `SustainedLowLatency`. Использование различных режимов задержки и рекомендации по поводу выбора подходящего приведены в главе 11.

GCSettings.IsServerGC

Показывает, в каком режиме была запущена CLR: в режиме рабочей станции или в серверном (см. главу 11). Отметим, что это свойство только для чтения, поскольку режим GC нельзя изменить после запуска среды выполнения. На это поле также не влияют другие настройки, в т. ч. режим задержки. Вместе с размером указателя (определяющим разрядность процесса) и количеством процессоров оно предоставляет вполне достаточные диагностические данные, которые можно записать в журнал на этапе запуска приложения (листинг 15.12).

Листинг 15.12 ❖ Пример получения простых диагностических данных

```
Console.WriteLine("{0} on {1}-bit with {2} CPUs",
                  (GCSettings.IsServerGC ? "Server" : "Workstation"),
                  ((IntPtr.Size == 8) ? 64 : 32),
                  Environment.ProcessorCount);
```

Уведомления GC

Частью GC API являются уведомления, позволяющие получать сведения о вероятности полной блокирующей GC. Это было необходимо до выпуска версии NET 4.5, когда в серверном режиме GC был реализован только неконкурентный блокирующий вариант. Поскольку такая сборка мусора занимает длительное время, получение возможности вовремя отреагировать на нее было крайне полезным. Типичный пример – использование этого уведомления для того, чтобы сообщить балансировщику нагрузки, что данный экземпляр сервера будет недоступен на время полной блокирующей GC. В настоящее время уведомления GC потеряли былую важность, поскольку чаще всего веб-приложения работают в фоновом режиме GC, когда время паузы гораздо меньше. Более того, такие уведомления рассылаются только для блокирующей сборки мусора. Поэтому если в конфигурационном файле разрешена конкурентная сборка, то уведомление вообще не генерируется.

API-уведомления состоят из следующих методов:

- `GC.RegisterForFullGCNotification(int maxGenerationThreshold, int largeObjectHeapThreshold)`, регистрирующий уведомление GC, которое должно быть отправлено, если сложились условия для полной блокирующей сборки мусора. Эти условия зависят от достижения пороговых значений расходования памяти для поколения 2 и LOH. Поэтому важно помнить, что уведомления не связаны напрямую с реальной сборкой мусора. Процитируем MSDN: «Отметим, что получение уведомления не означает, что полная сборка мусора обязательно произойдет, а говорит лишь о том, что достигнуты пороговые величины, благоприятствующие полной сборке мусора». Если мы зададим слишком высокие значения, то будем получать много ложноположитель-

ных уведомлений. С другой стороны, если значения слишком малы, можно пропустить реальную сборку мусора;

- `GC.CancelFullGCNotification` отменяет регистрацию уведомления GC;
- `GC.WaitForFullGCApproach` – это блокирующий вызов, который неопределенно долго ждет уведомления GC (существует также перегруженный вариант, принимающий величину тайм-аута);
- `GC.WaitForFullGCComplete` – это блокирующий вызов, который неопределенно долго ждет завершения полной GC (и также существует перегруженный вариант, позволяющий задать тайм-аут).

В листинге 15.13 приведен типичный пример использования уведомлений GC. Выделенный поток ждет уведомления и выполняет некоторое действие, если уведомление поступило.

Листинг 15.13 ❖ Пример использования уведомлений GC

```
GC.RegisterForFullGCNotification(10, 10);
Thread startpolling = new Thread(() =>
{
   while (true)
   {
      GCNotificationStatus s = GC.WaitForFullGCApproach(1000);
      if (s == GCNotificationStatus.Succeeded)
      {
         Console.WriteLine("GC скоро начнется");
      }
      else if (s == GCNotificationStatus.Timeout)
         continue;

      // ...
      // отреагировать на полную GC, например выполнить код, отключающий данный
      // сервер от балансировщика нагрузки
      // ...
      s = GC.WaitForFullGCComplete(10_000);
      if (s == GCNotificationStatus.Succeeded)
      {
         Console.WriteLine("GC закончилась");
      }
      else if (s == GCNotificationStatus.Timeout)
         Console.WriteLine("GC заняла слишком много времени");
      }
   });
   startpolling.Start();
   GC.CancelFullGCNotification();
```

Напомним, что этот API по определению неточный, т. к. мы просим предсказать будущее. Поэтому требуется поэкспериментировать с рабочей нагрузкой, чтобы найти подходящие значения аргументов `GC.RegisterForFullGCNotification`.

> Можно сетовать на необходимость угадывать значения порогов в методе `RegisterForFullGCNotification`, но другой альтернативы просто нет. В реальности ситуация все время меняется, поэтому в отсутствие строгой повторяемости событий трудно ожидать точного предсказания будущего. Тонкая настройка с помощью вышеупомянутых порогов позволяет по крайней мере адаптироваться к типичной рабочей нагрузке.

Контроль потребления неуправляемой памяти

Следующие методы позволяют сообщить GC о том, что некоторые управляемые объекты удерживают (или освобождают) некоторое количество неуправляемой памяти, которую сам GC не видит:

- `GC.AddMemoryPressure(Int64);`
- `GC.RemoveMemoryPressure(Int64).`

Если объем такой памяти превышает некоторый порог, то запускается сборка мусора. В главе 7 (где эти методы были проиллюстрированы в сценарии 7.3 «Анализ явных вызовов GC») объяснялось, что в настоящее время начальное значение этого порога равно 100 000 байт, а затем динамически корректируется. Листинг 12.3 в главе 12 дает еще один типичный пример применения данных методов.

> Заметим также, что при желании можно реализовать собственный механизм такого рода, если реализация по умолчанию вас не устраивает. Хотя этот механизм есть в классе `GC`, он не является внутренним для сборщика мусора (но все-таки реализован в среде выполнения).

Явная сборка мусора

Возможность явного запуска GC подробно описана в главе 7. Дополнительные сведения смотрите в разделе «Явный запуск» главы 7, в т. ч. в сценарии 7.3.

Для полноты перечислим перегруженные варианты метода GC, служащие для запуска явной сборки мусора:

- `Collect();`
- `Collect(int generation);`
- `Collect(int generation, GCCollectionMode mode);`
- `Collect(int generation, GCCollectionMode mode, bool blocking);`
- `Collect(int generation, GCCollectionMode mode, bool blocking, bool compacting).`

Области без GC

Участки программы, в которых среда выполнения пытается запретить GC, могут быть созданы следующими методами:

- `GC.TryStartNoGCRegion(long totalSize);`
- `GC.TryStartNoGCRegion(long totalSize, bool disallowFullBlockingGC);`
- `GC.TryStartNoGCRegion(long totalSize, long);`
- `GC.TryStartNoGCRegion(long totalSize, long lohSize, bool disallowFullBlockingGC);`
- `GC.EndNoGCRegion().`

Обсуждение, объяснение и примеры использования этих методов см. в разделе «Режим без сборки мусора» главы 11.

Управление финализацией

Методы для управления финализацией обсуждались в главе 12. Этот API включает в себя три метода:

- `GC.ReRegisterForFinalize(object obj);`
- `GC.SuppressFinalize(object obj);`
- `GC.WaitForPendingFinalizers().`

Потребление памяти

Обрабатывать исключение `OutOfMemoryException` неудобно, особенно если оно происходит в середине важного процесса. Мы можем заранее позаботиться о предотвращении таких ситуаций, воспользовавшись классом `MemoryFailPoint`, который пытается гарантировать наличие достаточного объема памяти перед началом важной обработки. Напомним, что полной гарантии отсутствия `OutOfMemoryException` этот API не дает, он лишь делает все возможное, чтобы избежать его.

Использовать класс просто (листинг 15.14). Конструктор `MemoryFailPoint` вызывает исключение `InsufficientMemoryException`, если осталось меньше памяти, чем требуется. Из-за внутренних накладных расходов, необходимых при многопоточном использовании, объект `MemoryFailPoint` реализует IDisposable, поэтому нужно не забыть вызвать его метод `Dispose` (или воспользоваться оператором `using`).

Листинг 15.14 ❖ Простой пример использования `MemoryFailPoint`

```
try
{
   using (MemoryFailPoint failPoint = new MemoryFailPoint(sizeInMegabytes:1024))
   {
      // Вычисления
   }
}
catch (InsufficientMemoryException e)
{
   Console.WriteLine(e);
   throw;
}
```

> Отметим, что в настоящее время эта функциональность реализована только в средах выполнения для Windows. В других системах конструктор `MemoryFailPoint` всегда завершается успешно.

В текущей реализации для Windows `MemoryFailPoint` проверяет возможность выделения запрошенного объема управляемой памяти, выполняя следующие шаги:

- проверяется, достаточно ли вообще виртуального адресного пространства, – эта проверка всегда должна завершаться успехом как в случае огромного 64-разрядного адресного пространства, так и в 32-разрядном случае, т. к. трудно представить, что понадобится сразу выделить больше памяти, чем размер виртуального адресного пространства;
- явно вызывается полная блокирующая сборка мусора с уплотнением, чтобы освободить все неиспользуемые сегменты и максимально уплотнить управляемую память;
- проверяется, достаточно ли свободного места в виртуальной памяти;
- проверяется, есть ли необходимость увеличить размер страничного файла ОС, чтобы удовлетворить запрос;
- проверяется, достаточно ли непрерывной свободной виртуальной памяти, чтобы при необходимости создать сегмент GC.

Я очень рекомендую почитать исходный код класса `MemoryFailPoint`, если вам интересно, как осуществляется управление свободной памятью процесса. В его коде используются вызовы Win32 API для получения сведений о доступной памяти (в закрытом методе `CheckForAvailableMemory`) и метод `VirtualQuery`, входящий в состав Virtual API, для нахождения непрерывной свободной области в виртуальном адресном пространстве (в закрытом методе `MemFreeAfterAddress`). Есть также внутренний статический метод `GetMemorySettings(out ulong maxGCSegmentSize, out ulong topOfMemory)`, реализованный в среде выполнения, который возвращает размер сегмента GC и максимальный доступный виртуальный адрес процесса. Зная об этом, мы могли бы даже получить информацию о размере этого сегмента, следующим образом воспользовавшись отражением:

```
var args = new object[2];
var mi = typeof(MemoryFailPoint).GetMethod("GetMemorySettings",
                                           BindingFlags.Static |
                                           BindingFlags.NonPublic);
mi.Invoke(null, args); // В результате args[0] содержит значение maxGCSegmentSize
```

Внутренние вызовы в классе GC

Если вам интересно, сообщу, что статический класс `GC` – на самом деле лишь обертка вокруг методов, реализованных внутри среды выполнения. Большая часть его методов снабжена атрибутом `InternalCall` (листинг 15.15) и отображается на соответствующие методы в файле CoreCLR `.\src\vm\ecalllist.h` (листинг 15.16).

Листинг 15.15 ❖ Фрагменты реализации класса GC из исходного кода CoreFX

```
public static class GC
{
   [MethodImplAttribute(MethodImplOptions.InternalCall)]
   public static extern int GetGeneration(Object obj);
   [MethodImplAttribute(MethodImplOptions.InternalCall)]
   internal static extern bool IsServerGC();
   ...
}
```

Листинг 15.16 ❖ Фрагменты интерфейса класса GC со средой выполнения из исходного кода CoreCLR

```
FCFuncStart(gGCInterfaceFuncs)
   FCFuncElement("IsServerGC", SystemNative::IsServerGC)
   FCFuncElement("GetGeneration", GCInterface::GetGeneration)
   ...
FCFuncEnd()
```

Статические методы `GCInterface` вызывают (по большей части) методы, определенные в файле `gc.cpp` (листинг 15.17).

Листинг 15.17 ❖ Пример реализации метода GC в среде выполнения

```
FCIMPL1(int, GCInterface::GetGeneration, Object* objUNSAFE)
{
   FCALL_CONTRACT;
   if (objUNSAFE == NULL)
      FCThrowArgumentNull(W("obj"));
```

```
    int result = (INT32)GCHeapUtilities::GetGCHeap()->WhichGeneration(objUNSAFE);
    FC_GC_POLL_RET();
    return result;
}
FCIMPLEND
```

Размещение CLR

Среду выполнения CLR в целом можно рассматривать как набор библиотек, умеющих загружать и выполнять CIL-код из .NET-совместимой сборки. Всякий раз, когда мы используем .NET, среда выполнения должна быть размещена в каком-то процессе. В случае .NET Framework благодаря поддержке со стороны ОС Windows код «начальной загрузки» включен в сам EXE-файл. А для .NET Core есть хорошо известное хост-приложение `dotnet`. На случай, если мы собираем CoreCLR самостоятельно, предлагается также упрощенный для тестирования хост `CoreRun`. У всех этих хостов есть одна общая особенность: они загружают подходящую среду выполнения CLR в память процесса, настраивают ее и выполняют загруженный код сборки (находящийся в указанном файле сборки). Такой хост есть, например, в экземпляре SQL Server, в результате чего мы можем выполнять в базе данных управляемый код.

API размещения (хостинга) открыт, так что любой желающий может написать собственный способ размещения CLR. Для этого могут быть разные причины, рассмотрим две самые популярные:

- создать внутреннюю среду выполнения CLR, чтобы можно было вызывать управляемый код из неуправляемого кода, и именно так поступает SQL Server;
- создать среду выполнения со своей конфигурацией, чтобы получить контроль над внутренними механизмами работы CLR, включая GC.

Поскольку при размещении CLR есть много возможностей настройки, мы можем сконфигурировать «собственную среду выполнения», отвечающую нашим потребностям. Ясно, что такая необходимость возникает редко, поэтому я не буду писать подробную инструкцию по размещению CLR, тем более что эта функциональность довольно хорошо документирована. Лучше вместо этого рассмотрим несколько примеров того, как этим можно воспользоваться в контексте управления памятью.

Пользуясь API размещения CLR (CLR Hosting API), мы попадаем в мир C++ и COM, полный точно определенных интерфейсов с четко специфицированной функциональностью. Каждый объект в API размещения CLR представлен конкретным интерфейсом. Главный интерфейс, представляющий саму среду выполнения, называется `ICLRRuntimeHost` (в .NET Framework) или `ICLRRuntimeHost4` (в .NET Core)[1].

API размещения CLR несколько различается в .NET Framework и .NET Core. Поскольку в настоящее время версия для .NET Core не поддерживает многие инте-

[1] Привыкайте к нумерации интерфейсов COM, потому что это стандартный способ обеспечения обратной совместимости. Вместо изменения существующего интерфейса создается новый с номером, увеличенным на единицу.

ресные нам возможности, мы рассмотрим только примеры для .NET Framework. Текущее состояние API для .NET Core описано в MSDN. Пока что версия API размещения CLR для .NET Core поддерживает в основном загрузку среды выполнения и исполнение кода, но не настройку с помощью описанных ниже интерфейсов.

Прежде чем переходить к примерам, проведем беглый обзор интерфейсов размещения CLR, относящихся к управлению памятью (и некоторых более общих интерфейсов, которые используются всегда), чтобы понять, какие у нас есть возможности для управления памятью. Хотя вся эта информация доступна в MSDN, я решил включить сюда краткую выжимку, поскольку, чтобы собрать ее из разных мест (и при этом пропустить устаревшие интерфейсы и т. п.), требуется немало времени. Для нас наибольший интерес представляют следующие интерфейсы:

- `ICLRControl` – интерфейс для получения различных менеджеров, представляющих конкретную функциональность (GC, отладка, управление сборками и т. д.). С точки зрения управления памятью в .NET, интересны два менеджера: `ICLRGCManager2` и `ICLRAppDomainResourceMonitor`;
- `ICLRGCManager2` – интерфейс, представляющий некоторые возможности управления GC. Точнее, он включает следующие методы:
  - `Collect` – явно запустить сборку мусора;
  - `GetStats` – получить текущую статистику сборки мусора, при этом напрямую используются те же данные, которые представлены соответствующими счетчиками производительности (в сборке CoreCLR эти данные недоступны);
  - `SetGCStartupLimitsEx` – установить размер сегмента GC и максимальный размер поколения 0, которые будут использованы на этапе инициализации среды выполнения;
- `ICLRAppDomainResourceMonitor` предоставляет результаты измерения для домена приложения – те же значения, которые возвращают свойства `MonitoringTotalAllocatedMemorySize` и `MonitoringSurvivedMemorySize` объекта `AppDomain`;
- `IHostControl` – интерфейс, позволяющий встраивать различные «менеджеры размещения» в размещенную CLR. С точки зрения управления памятью, интерес представляют два из них: `IHostGCManager` и `IHostMemoryManager`. Если мы хотим встроить собственный менеджер, то должны переопределить метод `GetHostManager`, так чтобы он возвращал нашу реализацию соответствующего интерфейса;
- `IHostGCManager` отправляет уведомления о приостановке GC и содержит следующие методы, которые мы должны реализовать:
  - `SuspensionStarting` срабатывает, когда CLR начинает приостанавливать потоки перед сборкой мусора;
  - `SuspensionEnding` срабатывает, когда CLR возобновляет потоки после завершения сборки мусора в данном поколении;
  - `ThreadIsBlockingForSuspension` срабатывает в каждом работающем потоке перед приостановкой;
- `IHostMemoryManager` предоставляет ряд важных методов, относящихся к управлению памятью. Реализовав этот интерфейс, мы получим полный контроль над тем, как CLR потребляет системную память для своих целей. Например, можно полностью отказаться от Virtual API, встроенного в Windows, в поль-

зу других библиотек (или изменить порядок использования Virtual API). Необходимо реализовать следующие методы:

- `AcquiredVirtualAddressSpace` информирует о том, что CLR получила запрошенное количество памяти от операционной системы. Не вызывается, если мы создаем собственный диспетчер памяти, который не обращается к этому методу;
- `CreateMalloc` получает реализацию интерфейса `IHostmalloc`, отвечающего за запрашивание памяти из кучи внутри CLR. Таким образом, мы можем полностью изменить порядок выделения памяти для внутренних потребностей CLR, например заменить распределитель по умолчанию `malloc` распределителем `jemalloc` (упоминался в главе 14). Отметим, что так можно подменить только распределитель памяти для внутренних данных CLR, но не распределитель GC, используемый для выделения памяти объектам в управляемой куче;
- `GetMemoryLoad` возвращает объем используемой физической памяти;
- `NeedsVirtualAddressSpace` информирует хост о том, что CLR требуется указанное количество памяти;
- `RegisterMemoryNotificationCallback` позволяет зарегистрировать реализацию интерфейса `ICLRMemoryNotificationCallback`, который служит для уведомления CLR о высоком потреблении памяти;
- `ReleasedVirtualAddressSpace` информирует хост о том, что CLR больше не нужно указанное количество памяти;
- `VirtualAlloc` служит для получения виртуальной памяти от системы. Благодаря этому методу мы можем полностью заменить или модифицировать способ использования Virtual API для получения страниц памяти;
- `VirtualFree` служит для возврата виртуальной памяти системе;
- `VirtualProtect` используется, чтобы изменить защиту указанной области виртуальной памяти;
- `VirtualQuery` служит для запроса информации об указанной области виртуальной памяти;

- `IHostMalloc`:
  - `Alloc` – вызывается средой выполнения, чтобы попросить хост о выделении указанного количества памяти из кучи;
  - `DebugAlloc` – делает то же, что и Alloc, но дополнительно отслеживает, где была выделена память;
  - `Free` – вызывается средой выполнения, чтобы освободить память, выделенную методом `Alloc` или `DebugAlloc`.

Схема взаимодействия этих интерфейсов показана на рис. 15.2. Для нас интереснее всего, как в нашем хосте CLR переопределить способ получения CLR страниц памяти и способ выделения памяти в неуправляемой куче.

> Размещение CLR открывает много других возможностей, но мы показали только те, что близки теме книги. Например, интерфейс `ICLROnEventManager` позволяет предпринять некоторое действие при возникновении исключения `StackOverflowException`. Заметим также, что до выпуска версии .NET Framework 2.0 использовался другой набор интерфейсов, в частности среда выполнения была представлена интерфейсом `ICorRuntimeHost`, а для управления GC использовался интерфейс `IGCHost`. Эти интерфейсы здесь не описаны, поскольку они давно устарели и больше не используются.

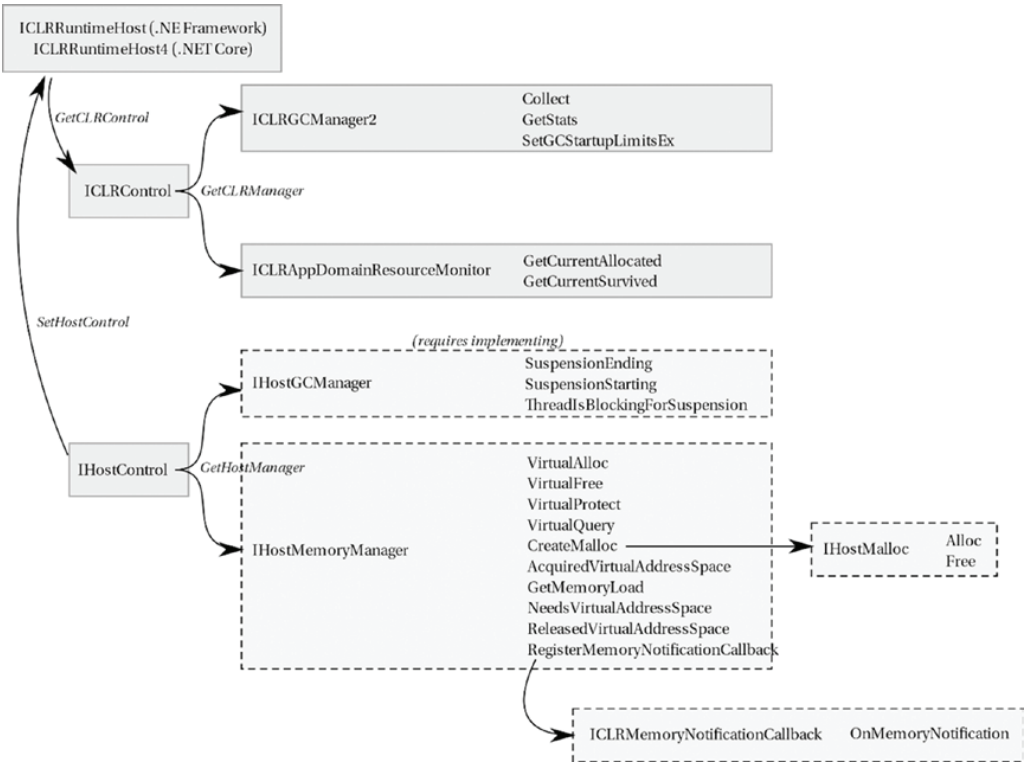


Рис. 15.2 ❖ Схема интерфейсов,
имеющих отношение к выделению памяти в API размещения CLR

В листинге 15.18[1] приведен пример загрузки CLR и получения интерфейсов `ICLRRuntimeHost` и `ICLRControl`. Напомним, что примеры размещения CLR написаны на неуправляемом C++ (а проект сконфигурирован как обычное консольное приложение Windows).

Листинг 15.18 ❖ Инициализация размещения CLR

```
ICLRRuntimeHost* runtimeHost;
ICLRMetaHost *pMetaHost = nullptr;
ICLRRuntimeInfo *pRuntimeInfo = nullptr;
hr = CLRCreateInstance(CLSID_CLRMetaHost, IID_ICLRMetaHost, (LPVOID*)&pMetaHost);
hr = pMetaHost->GetRuntime(L"v4.0.30319", IID_PPV_ARGS(&pRuntimeInfo));
hr = pRuntimeInfo->GetInterface(CLSID_CLRRuntimeHost, IID_ICLRRuntimeHost,
                                (LPVOID*)&runtimeHost);
ICLRControl* clrControl;
hr = runtimeHost->GetCLRControl(&clrControl);
```

Начиная с этого момента мы могли бы просто запустить среду выполнения и выполнить указанный метод из сборки в указанном файле (листинг 15.19). Но нас больше интересует настройка под свои нужды, поэтому пойдем дальше.

---

[1] Для краткости в примерах ниже показаны только релевантные части кода. Полные работающие примеры можно найти в репозитории на GitHib.

Листинг 15.19 ❖ Выполнение кода после размещения CLR

```
DWORD dwReturn;
hr = runtimeHost->Start();
hr = runtimeHost->ExecuteInDefaultAppDomain(targetApp, L"HelloWorld.Program", L"Test",
   L"", &dwReturn);
```

С точки зрения управления памятью со стороны размещенной CLR, мы можем выделить три группы возможностей:

- конфигурирование: помимо стандартных флагов CLR (выбора режима GC – рабочей станции / серверного режима и конкурентности), мы можем воспользоваться методом `ICLRGCManager2::SetGCStartupLimitsEx`, чтобы задать размер сегмента GC по умолчанию и максимальный размер поколения 0 (листинг 15.20);
- диагностические измерения: благодаря методу `ICLRGCManager2::GetStats` и интерфейсу `ICLRAppDomainResourceMonitor` мы можем наблюдать за потреблением памяти размещенным экземпляром CLR (листинг 15.21). Это бывает особенно полезно в средах с высоким уровнем требований (например, в промышленной среде), когда нужно следить за тем, чтобы размещенный управляемый код не нарушал ограничений на потребление памяти;
- адаптация: благодаря интерфейсу `IHostControl` мы можем встроить широкий спектр менеджеров, предоставив собственные реализации (листинг 15.22). Это самая интересная часть данного раздела, поэтому остановимся на ней подробнее.

Листинг 15.20 ❖ Пример вызова `SetGCStartupLimitsEx` при размещении CLR

```
ICLRGCManager2* clrGCManager;
hr = clrControl->GetCLRManager(IID_ICLRGCManager2, (void**)&clrGCManager);
SIZE_T segmentSize = 4 * 1024 * 1024 * 1024;
SIZE_T maxGen0Size = 4 * 1024 * 1024 * 1024;
hr = clrGCManager->SetGCStartupLimitsEx(segmentSize, maxGen0Size);
```

Листинг 15.21 ❖ Пример получения данных о потреблении памяти размещенной CLR

```
_COR_GC_STATS gcStats;
gcStats.Flags = COR_GC_COUNTS | COR_GC_MEMORYUSAGE;
// Основано на счетчиках производительности, поэтому не работает в CoreCLR
hr = clrGCManager->GetStats(&gcStats);
cout << gcStats.CommittedKBytes << endl
     << gcStats.Gen0HeapSizeKBytes << endl
     << gcStats.Gen1HeapSizeKBytes << endl
     << gcStats.Gen2HeapSizeKBytes << endl
     << gcStats.LargeObjectHeapSizeKBytes << endl
     << gcStats.ExplicitGCCount << endl
     << gcStats.GenCollectionsTaken[0] << endl
     << gcStats.GenCollectionsTaken[1] << endl
     << gcStats.GenCollectionsTaken[2] << endl;
```

Листинг 15.22 ❖ Задание собственного контроллера хоста при размещении CLR

```
CustomHostControl customHostControl;
hr = runtimeHost->SetHostControl(&customHostControl);
```

При замене реализации интерфейса IHostControl на свою мы должны реализовать метод GetHostManager, который CLR вызывает для получения необходимых менеджеров (листинг 15.23). Если этот метод возвращает E_NOINTERFACE, будет использован менеджер по умолчанию. Мы хотим подменить реализацию IHostMemoryManager так, чтобы возвращался наш класс CustomHostMemoryManager. Заметим, что любой COM-интерфейс должен реализовать методы интерфейса IUnknown: AddRef, Release и QueryInterface. Здесь они показаны, но в последующих листингах для краткости опущены.

Листинг 15.23 ❖ Пример реализации интерфейса IHostControl

```
class CustomHostControl : public IHostControl
{
   ULONG referenceCounter;

public:
   CustomHostControl()
   {
      referenceCounter = 0;
   }

   // Объявлен в IHostControl
   virtual HRESULT GetHostManager(REFIID riid, void ** ppObject) override
   {
      if (riid == IID_IHostMemoryManager)
      {
         IHostMemoryManager *pMemoryManager = new CustomHostMemoryManager();
         *ppObject = pMemoryManager;
         return S_OK;
      }
      *ppObject = NULL;
      return E_NOINTERFACE;
   }

   virtual HRESULT QueryInterface(const IID &riid, void **ppvObject)
   {
      if (riid == IID_IUnknown)
      {
         *ppvObject = static_cast<IUnknown*>(static_cast<IHostControl*>(this));
         return S_OK;
      }
      if (riid == IID_IHostControl)
      {
         *ppvObject = static_cast<IHostControl*>(this);
         return S_OK;
      }
      *ppvObject = NULL;
      return E_NOINTERFACE;
   }

   virtual ULONG AddRef()
   {
      return referenceCounter++;
   }
```

```
   virtual ULONG Release()
   {
      return referenceCounter--;
   }
};
```

Пользовательский `HostMemoryManager` позволяет заменить весь механизм управления виртуальной памятью и выделения памяти в куче. Напомним, что GC (и его внутренние распределители) рассматривается как черный ящик, то есть страницы памяти для него запрашиваются так же, как для любой другой части программы. На самом деле нет никакой возможности отличить вызов `VirtualAlloc`, запрашивающий страницы для управляемой кучи, от любого другого вызова.

Однако даже при таком уровне гибкости мы можем реализовать интересные вещи. Например, можно переопределить метод `VirtualAlloc` так, что он будет *закреплять* (lock) все полученные страницы в физической памяти, т. е. их нельзя будет выгрузить на диск (с высокой вероятностью). В таком случае другие методы можно сделать обертками вокруг вызовов Virtual API (листинг 15.24). Агрессивное закрепление страниц может повысить производительность .NET-приложения, поскольку его память, скорее всего, будет находиться в физическом ОЗУ.

Листинг 15.24 ❖ Пример пользовательского менеджера памяти хоста, который реализует агрессивное закрепление страниц в физической памяти

```
class CustomHostMemoryManager : public IHostMemoryManager
{
   ULONG referenceCounter;

public:
   CustomHostMemoryManager() : referenceCounter(0) { }

   // Объявлены в IHostMemoryManager
   virtual HRESULT CreateMalloc(DWORD dwMallocType, IHostMalloc ** ppMalloc) override
   {
      *ppMalloc = new CustomHostMalloc();
      return S_OK;
   }

   virtual HRESULT VirtualAlloc(void * pAddress, SIZE_T dwSize, DWORD flAllocationType,
      DWORD flProtect, EMemoryCriticalLevel eCriticalLevel, void ** ppMem) override
   {
      void* result = ::VirtualAlloc(pAddress, dwSize, flAllocationType, flProtect);
      *ppMem = result;
      BOOL locked = false;
      if (flAllocationType & MEM_COMMIT)
      {
         locked = ::VirtualLock(*ppMem, dwSize);
      }
      cout << "VirtualAlloc " << *ppMem << " (" << dwSize << "), flags: "
           << flAllocationType << " " << flProtect << " => "
           << pAddress << " " << locked << endl;
      return S_OK;
   }

   virtual HRESULT VirtualFree(LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType)
```

```
      override
   {
      ::VirtualFree(lpAddress, dwSize, dwFreeType);
      return S_OK;
   }

   virtual HRESULT VirtualQuery(void * lpAddress, void * lpBuffer, SIZE_T dwLength,
                                SIZE_T * pResult) override
   {
      *pResult = ::VirtualQuery(lpAddress, (PMEMORY_BASIC_INFORMATION)lpBuffer,
                                dwLength);
      return S_OK;
   }

   virtual HRESULT VirtualProtect(void * lpAddress, SIZE_T dwSize, DWORD flNewProtect,
                                  DWORD * pflOldProtect) override
   {
      ::VirtualProtect(lpAddress, dwSize, flNewProtect, pflOldProtect);
      return S_OK;
   }

   virtual HRESULT GetMemoryLoad(DWORD * pMemoryLoad, SIZE_T * pAvailableBytes) override
   {
      // Моделируем отсутствие проблем
      *pMemoryLoad = 1;
      *pAvailableBytes = 1024 * 1024 * 1024;
      return S_OK;
   }

   virtual HRESULT RegisterMemoryNotificationCallback(
      ICLRMemoryNotificationCallback * pCallback) override
   {
      return S_OK;
   }

   virtual HRESULT NeedsVirtualAddressSpace(LPVOID startAddress, SIZE_T size) override
   {
      return S_OK;
   }
   virtual HRESULT AcquiredVirtualAddressSpace(LPVOID startAddress, SIZE_T size)
      override
   {
      return S_OK;
   }

   virtual HRESULT ReleasedVirtualAddressSpace(LPVOID startAddress) override
   {
      return S_OK;
   }

   // Объявлены в IUnknown
   // ...
};
```

В представленной пользовательской реализации `IHostMemoryManager` переопределен также метод `CreateMalloc`, который возвращает нашу реализацию `IHostMalloc` (листинг 15.25). Она показана просто для иллюстрации, но можно представить себе целый ряд различных реализаций, в т. ч. с использованием уже упоминавшейся библиотеки jemalloc вместо вызова стандартных функций `malloc` и `free`.

Листинг 15.25 ❖ Пример пользовательской реализации выделения из кучи в размещенной CLR

```
class CustomHostMalloc : public IHostMalloc
{
   ULONG referenceCounter;

public:
   CustomHostMalloc() : referenceCounter(0) { }

   // Объявлены в IHostMalloc
   virtual HRESULT Alloc(SIZE_T cbSize, EMemoryCriticalLevel eCriticalLevel,
                         void ** ppMem) override
   {
      *ppMem = ::malloc(cbSize);
      cout << " Alloc " << *ppMem << " (" << cbSize << ")" << endl;
      return S_OK;
   }

   virtual HRESULT DebugAlloc(SIZE_T cbSize, EMemoryCriticalLevel eCriticalLevel,
                  char * pszFileName, int iLineNo, void ** ppMem) override
   {
      *ppMem = ::malloc(cbSize);
      return S_OK;
   }

   virtual HRESULT Free(void * pMem) override
   {
      ::free(pMem);
      return S_OK;
   }

   // Объявлены в IUnknown
   // ...
};
```

Понятно, что представленный выше «невыгружаемый хост CLR» – всего лишь простой набросок. Полная, гораздо лучше продуманная реализация Саши Гольдштейна и Алона Флисса доступна по адресу https://archive.codeplex.com/?p=nonpagedclrhost. Очень рекомендую почитать ее исходный код. Например, в нем учтены лимиты на закрепление страниц. Очевидно, что чрезмерно агрессивное закрепление может негативно сказаться на производительности системы в целом, поскольку другим приложениям достанется меньше физической памяти. Процитируем MSDN: «Максимальное количество страниц, которые может закрепить один процесс, равно количеству страниц в его минимальном рабочем наборе за вычетом небольших накладных расходов». Поэтому в реализации Саши и Алона используется системный вызов Win32 `SetProcessWorkingSetSize`, чтобы правильно сконфигурировать лимиты рабочего набора.

ClrMD

Библиотека `Microsoft.Diagnostics.Runtime`, известная также под названием ClrMD (или CLR MD), – это набор управляемых API для исследования управляемых процессов и дампов памяти. Она предназначена скорее для создания инструментов диагностики и небольших фрагментов кода (snippets), чем для автомониторинга процесса (хотя такая возможность тоже существует, как мы скоро увидим). Она предлагает примерно такие же возможности, как расширение SOS программы WinDbg, но в виде, гораздо более удобном для использования в коде на C#. Библиотека `Microsoft.Diagnostics.Runtime` доступна в виде NuGet-пакета и может использоваться в приложениях для .NET Framework и .NET Core с целью анализа программ, предназначенных для обеих этих платформ. Ее полный исходный код выложен на GitHub, так что можете посмотреть, как она реализована!

Отметим, что описать все возможности этой библиотеки не представляется возможным из-за ограничений на объем книги. Примеры ниже призваны лишь продемонстрировать, на что она способна и какой потенциал в ней заложен. Данный раздел не является ни пособием по ClrMD, ни полным описанием того, как можно применить эту библиотеку. Дополнительные сведения можно найти в документации по ClrMD и примерах кода.

Для работы с ClrMD необходим корневой объект – экземпляр класса `DataTarget`, который можно получить, подключившись (attach) к работающему процессу или загрузив дамп памяти с помощью следующих статических методов:

- `AttachToProcess` – позволяет подключиться к существующему процессу с заданным PID (идентификатором процесса). Это можно сделать тремя способами:
  - `Invasive` – процесс будет приостановлен, и мы сможем управлять им, как если бы подключили обычный отладчик. Это предпочтительный способ в обычных обстоятельствах;
  - `NonInvasive` – процесс будет приостановлен, но управлять им мы не сможем. Поскольку, вообще говоря, любым процессом может управлять только один отладчик, этот метод полезен, если мы хотим подключиться к процессу, с которым уже работает какой-то другой отладчик;
  - `Passive` – процесс не приостанавливается, и никакой отладчик к нему не подключается. Следует иметь в виду, что ответы на многие запросы, касающиеся динамически изменяющихся данных, например о стеках потоков и ссылках на объекты, могут оказаться несогласованными. Идея этого режима в том, что программа, использующая ClrMD, несет ответственность за все действия, связанные с управлением процессом (например, приостановку наблюдаемого процесса). Это дает разработчику полную гибкость в плане управления целевым процессом;
- `LoadCrashDump` – позволяет загрузить файл с дампом памяти (например, полученным с помощью программы ProcDump).

Заметим, что режим `Passive` теоретически позволяет подключиться даже к собственному процессу, т. е. реализовать автомониторинг. Но если задуматься, то станет ясно, что здесь много проблем, например: как ClrMD должна обрабатывать динамически изменяющееся состояние процесса, инспектировать кучу, когда производится выде-

ление памяти и сборка мусора и т. д. Разработчики ClrMD не стали запрещать автомониторинг, потому что в некоторых пограничных случаях он может быть даже полезен. Но это занятие не для слабонервных, так что если столкнетесь с проблемами, считайте, что такой сценарий не поддерживается.

После того как объект `DataTarget` инициализирован, мы можем приступить к изучению данных, например выяснить, какие среды выполнения в нем используются (или использовались) (листинг 15.26). Сюда входит информация о необходимом DAC (Data Access Component – компоненте доступа к данным), который отвечает за интерпретацию внутренних структур данных CLR.

Листинг 15.26 ❖ Пример простого использования ClrMD – подключение к работающему процессу

```
using (DataTarget target = DataTarget.AttachToProcess(pid, 5000, AttachFlag.Invasive))
{
   foreach (ClrInfo clrInfo in target.ClrVersions)
   {
      Console.WriteLine("Found CLR Version:" + clrInfo.Version.ToString());

      // Это данные, необходимые для запроса dac у сервера символов:
      ModuleInfo dacInfo = clrInfo.DacInfo;
      Console.WriteLine($"Filesize: {dacInfo.FileSize:X}");
      Console.WriteLine($"Timestamp: {dacInfo.TimeStamp:X}");
      Console.WriteLine($"Dac File: {dacInfo.FileName}");
      ClrRuntime runtime = clrInfo.CreateRuntime();
      ...
   }
}
```

Корректно инициализировав экземпляр `ClrRuntime`, мы можем сделать много интересного. Рассмотрим лишь несколько примеров. Заметим, что мы демонстрируем лишь малую толику методов и атрибутов используемых объектов ClrMD. Полный перечень смотрите в документации.

Можно изучить все работающие потоки и распечатать их стеки (листинг 15.27).

Листинг 15.27 ❖ Пример использования ClrMD – распечатка стеков вызовов всех потоков

```
foreach (ClrThread thread in runtime.Threads)
{
   if (!thread.IsAlive)
      continue;
   Console.WriteLine("Thread {0:X}:", thread.OSThreadId);
   foreach (ClrStackFrame frame in thread.StackTrace)
      Console.WriteLine("{0,12:X} {1,12:X} {2}", frame.StackPointer,
                        frame.InstructionPointer, frame.ToString());
      Console.WriteLine();
}
```

Можно перебрать все домены приложений и модули, загруженные средой выполнения, а также все используемые в них управляемые типы (листинг 15.28).

Листинг 15.28 ❖ Пример использования ClrMD – печать всех загруженных доменов приложений, модулей и типов

```
foreach (var domain in runtime.AppDomains)
{
   Console.WriteLine($"AppDomain {domain.Name} ({domain.Address:X})");
   foreach (var module in domain.Modules)
   {
      Console.WriteLine($" Module {module.Name} ({(module.IsFile ? module.FileName
                                                                  : "")})");
      foreach (var type in module.EnumerateTypes())
      {
         Console.WriteLine($"{type.Name} Fields: {type.Fields.Count}");
      }
   }
}
```

> Заметим, что ClrMD показывает состояние процесса, которое видит среда выполнения, а не то, что определено в коде. Предположим, например, что в некотором загруженном модуле определен тип `Foo`, но этот тип ни разу не использовался в процессе. В таком случае метод `EnumerateTypes` может вернуть `Foo`, а может и не вернуть – все зависит от того, решила ли среда выполнения загружать этот тип или нет. А это решение среды зависит от реализации, которая может меняться от версии к версии.

Впрочем, нам, конечно, более интересна информация, относящаяся к памяти. Например, можно исследовать все области памяти, используемые CLR, включая управляемую кучу (см. листинг 15.29 и результаты в листинге 15.30).

Листинг 15.29 ❖ Пример использования ClrMD – перечисление всех областей памяти процесса

```
foreach (var region in runtime.EnumerateMemoryRegions().OrderBy(r => r.Address))
{
   Console.WriteLine($"0x{region.Address:X} (bytes: {region.Size:N0}) - {region.Type} "
     + $"{(region.Type == ClrMemoryRegionType.GCSegment ?
     "(" + region.GCSegmentType.ToString() + ")" : "")}");
}
```

Листинг 15.30 ❖ Результаты работы программы в листинге 15.28

```
0x24198CC1000 (bytes: 4,096) - HandleTableChunk
0x24199541000 (bytes: 200,704) - GCSegment (Ephemeral)
0x24199572000 (bytes: 268,230,656) - ReservedGCSegment
0x241A9541000 (bytes: 69,632) - GCSegment (LargeObject)
0x241A9552000 (bytes: 134,144,000) - ReservedGCSegment
0x7FF9F5250000 (bytes: 12,288) - LowFrequencyLoaderHeap
0x7FF9F5250000 (bytes: 12,288) - LowFrequencyLoaderHeap
0x7FF9F5256000 (bytes: 28,672) - HighFrequencyLoaderHeap
0x7FF9F5256000 (bytes: 28,672) - HighFrequencyLoaderHeap
0x7FF9F525D000 (bytes: 12,288) - StubHeap
0x7FF9F525D000 (bytes: 12,288) - StubHeap
0x7FF9F5260000 (bytes: 12,288) - LowFrequencyLoaderHeap
0x7FF9F5263000 (bytes: 40,960) - HighFrequencyLoaderHeap
0x7FF9F5274000 (bytes: 28,672) - CacheEntryHeap
0x7FF9F527D000 (bytes: 192,512) - DispatchHeap
```

```
0x7FF9F52AC000 (bytes: 344,064) - ResolveHeap
0x7FF9F5300000 (bytes: 24,576) - IndcellHeap
0x7FF9F5300000 (bytes: 24,576) - IndcellHeap
0x7FF9F5306000 (bytes: 24,576) - CacheEntryHeap
0x7FF9F5306000 (bytes: 24,576) - CacheEntryHeap
0x7FF9F530C000 (bytes: 16,384) - LookupHeap
0x7FF9F530C000 (bytes: 16,384) - LookupHeap
0x7FF9F5310000 (bytes: 155,648) - DispatchHeap
0x7FF9F5310000 (bytes: 155,648) - DispatchHeap
0x7FF9F5336000 (bytes: 237,568) - ResolveHeap
0x7FF9F5336000 (bytes: 237,568) - ResolveHeap
0x7FF9F53B0000 (bytes: 65,536) - LowFrequencyLoaderHeap
```

Управляемую кучу можно исследовать и дальше с помощью класса `ClrHeap`, экземпляр которого возвращает свойство `Heap` объекта `ClrRuntime`. Он позволяет обойти все существующие управляемые объекты, а также показать содержимое их полей и ссылок (см. листинг 15.31 и результаты в листинге 15.32).

Листинг 15.31 ❖ Пример использования ClrMD – перечисление некоторых экземпляров управляемых типов

```
ClrHeap heap = runtime.Heap;
foreach (var clrObject in heap.EnumerateObjects())
{
   if (clrObject.Type.Name.EndsWith("SampleClass"))
      ShowObject(heap, clrObject, string.Empty);
}

private static void ShowObject(ClrHeap heap, ClrObject clrObject, string indent)
{
   Console.WriteLine($"{indent}{clrObject.Type.Name} ({clrObject.HexAddress}) -
gen{heap.GetGeneration(clrObject.Address)}");
   foreach (var reference in clrObject.EnumerateObjectReferences())
   {
      ShowObject(heap, reference, " ");
   }
}
```

Листинг 15.32 ❖ Результаты работы программы в листинге 15.31

```
CoreCLR.HelloWorld.SampleClass (24199564fa0) - gen0
   CoreCLR.HelloWorld.AnotherClass (24199564fc0) - gen0
   CoreCLR.HelloWorld.AnotherClass (24199564fd8) - gen0
   CoreCLR.HelloWorld.SomeOtherClass (24199564ff0) - gen0
```

Отдельные сегменты GC также можно исследовать, пользуясь свойством `Segments` объекта `ClrHeap`. Каждый объект `ClrSegment` содержит интересные данные о сегменте, включая его внутреннюю структуру, например то, какие поколения он содержит (см. листинг 15.33 и результаты в листинге 15.34).

Листинг 15.33 ❖ Пример использования ClrMD – перечисление сегментов GC, принадлежащих процессу

```
foreach (var segment in heap.Segments)
{
   Console.WriteLine($"{segment.Start:X16} - {segment.End:X16} ({segment.
```

```
   CommittedEnd:X16}) Heap#: {segment.ProcessorAffinity}");
   if (segment.IsEphemeral)
   {
      Console.WriteLine($" Gen0: {segment.Gen0Start:X16} ({segment.Gen0Length})");
      Console.WriteLine($" Gen1: {segment.Gen1Start:X16} ({segment.Gen1Length})");
      if (segment.Gen2Start >= segment.Start &&
          segment.Gen2Start < segment.CommittedEnd)
      {
         Console.WriteLine($" Gen2: {segment.Gen2Start:X16} ({segment.Gen2Length})");
      }
   }
   else if (segment.IsLarge)
   {
      Console.WriteLine($" LOH: {segment.Start} ({segment.Length})");
   }
   else
   {
      Console.WriteLine($" Gen2: {segment.Gen2Start:X16} ({segment.Gen2Length})");
   }

   foreach (var address in segment.EnumerateObjectAddresses())
   {
      var type = heap.GetObjectType(address);
      if (type == heap.Free)
      {
         Console.WriteLine($"{type.GetSize(address)}");
      }
   }
}
```

Листинг 15.34 ❖ Результаты работы программы в листинге 15.32

```
000002551B871000 - 000002551B896730 (000002551B8A2000) Heap#: 0
   Gen0: 000002551B871030 (153344)
   Gen1: 000002551B871018 (24)
   Gen2: 000002551B871000 (24)
```

> Мы уже знаем об одной детали реализации GC – о сегментах (представляющих кучи), привязанных к процессору, который занимается выделением памяти, пометкой и т. д. Но концептуально поле `ProcessorAffinity` лучше рассматривать как номер кучи, в которой находится сегмент. По сути дела, его следовало бы назвать `HeapNumber`, а не `ProcessorAffinity`.

Загромождать этот раздел все новыми и новыми примерами, наверное, излишне. Полагаю, вы уже убедились в огромном потенциале ClrMD. Упомяну лишь еще несколько интересных возможностей:

- перечисление всех объектов в очереди `fReachable` с помощью метода `runtime.EnumerateFinalizerQueueObjectAddresses()`;
- перечисление всех описателей с помощью метода `runtime.EnumerateHandles()`;
- перечисление всех текущих корней GC с помощью метода `heap.EnumerateRoots()`;
- перечисление всех текущих стековых корней для потока;
- получение адреса кода метода, сгенерированного JIT-компилятором (чтобы можно было с помощью дизассемблера увидеть его машинный код).

Довольно популярен подход, когда ClrMD используется (особенно при анализе дампов памяти) из приложения LINQPad (https://www.linqpad.net). В нем удобно писать скрипты, так что можно использовать ClrMD, обходясь без Visual Studio и создания специальных проектов.

Несмотря на все богатство возможностей, иногда оказывается, что ClrMD пока не предоставляет открыто свойства, которые очень хотелось бы видеть. Один из примеров – исследование контекста выделения памяти текущего потока. Эта информация известна ClrMD, но соответствующие свойства напрямую недоступны. Их можно получить с помощью отражения (но напомним, что нет никакой гарантии, что они не будут изменены в будущих версиях).

```
foreach (ClrThread thread in runtime.Threads)
{
  var mi = runtime.GetType().GetMethod("GetThread", BindingFlags.Instance
                                                  | BindingFlags.NonPublic);
  var threadData = mi.Invoke(runtime, new object[] {thread.Address});
  var pi = threadData.GetType().GetProperty("AllocPtr", BindingFlags.Instance
                                                      | BindingFlags.Public);
  ulong allocPtr = (ulong) pi.GetValue(threadData);
  pi = threadData.GetType().GetProperty("AllocLimit", BindingFlags.Instance
                                                    | BindingFlags.Public);
  ulong allocLimit = (ulong) pi.GetValue(threadData);
}
```

Этот пример показывает, что изучение исходного кода ClrMD иногда приносит реальную пользу!

Если взглянуть на мир моими глазами, вы уже видите воочию, какие замечательные инструменты диагностики можно написать, пользуясь возможностями ClrMD. И действительно, уже есть много крупных и мелких проектов (в основном с открытым исходным кодом), направленных на создание таких инструментов и появившихся по самым разным причинам. Перечислить здесь их все нет возможности, но два самых важных я упомяну: Netext и SOSEX. Это расширения WinDbg, написанные как обертки ClrMD. И да – есть какая-то ирония в том, что одно из лучших расширений WinDbg для диагностики .NET написано на .NET.

Если вас интересует актуальный список инструментов, основанных на ClrMD (или каким-то образом интегрированных с ней), загляните на сайт Tools по адресу http://mattwarren.org/2018/06/15/Tools-for-Exploring-.NET-Internals, который поддерживает Мэтт Уоррен (Matt Warren).

Библиотека TraceEvent

Microsoft.Diagnostics.Tracing.TraceEvent – это библиотека для .NET, содержащая средства сбора и обработки данных ETW. Это существенная часть механизма PerfView, распространяемая в виде отдельного NuGet-пакета (однако исходный код также доступен в репозитории PerfView).

Не стану повторять примеры использования TraceEvent, чтобы не увеличивать размер книги. Полную документацию и примеры можно найти по адресу

https://github.com/Microsoft/perfview/blob/master/documentation/TraceEvent/TraceEventProgrammersGuide.md. Лучше кратко подытожить: библиотека позволяет записать сеанс ETW в файл (обычный ETL-файл, знакомый нам по обзору PerfView), а затем проанализировать его, или просто создать и использовать сеанс ETW в режиме реального времени. Можно включить любой поставщик ETW и получать его события.

Для удобства работы с двумя самыми популярными поставщиками ETW в библиотеку TraceEvent уже встроено два строго типизированных анализатора: `ClrTraceEventParser` и `KernelTraceEventParser` (они представлены свойствами `Clr` и `Kernel` сеанса `Source`). Поскольку первый знает, как разбирать все события CLR, он бывает очень полезен во всех сценариях, имеющих отношение к сборке мусора. Мы просто используем строго типизированные обратные вызовы, которые представляют собой реакцию на интересующие нас события. В листинге 15.35 приведен пример создания сеанса ETW, который в реальном времени реагирует на события начала и завершения GC и печатает статистические данные.

Листинг 15.35 ❖ Пример использования TraceEvent: работа со встроенным анализатором событий поставщика CLR

```
using (var session = new TraceEventSession("SampleETWSession"))
{
   Console.CancelKeyPress += (object sender, ConsoleCancelEventArgs cancelArgs) =>
   {
      session.Dispose();
      cancelArgs.Cancel = true;
   };

   session.EnableProvider(ClrTraceEventParser.ProviderGuid,
      TraceEventLevel.Verbose, (ulong)ClrTraceEventParser.Keywords.Default);
   session.Source.Clr.GCStart += ClrOnGcStart;
   session.Source.Clr.GCStop += ClrOnGcStop;
   session.Source.Clr.GCHeapStats += ClrOnGcHeapStats;
   session.Source.Process();
}

private static void ClrOnGcStart(GCStartTraceData data)
{
   Console.WriteLine($"[{data.ProcessName}] GC gen{data.Depth} because
{data.Reason} started {data.Type}.");
}

private static void ClrOnGcStop(GCEndTraceData data)
{
   Console.WriteLine($"[{data.ProcessName}] GC ended.");
}

private static void ClrOnGcHeapStats(GCHeapStatsTraceData data)
{
   Console.WriteLine($"[{data.ProcessName}] Heapstats -
{data.GenerationSize0:N0}|{data.GenerationSize1:N0}|{data.GenerationSize2:N0}|
{data.GenerationSize3}");
}
```

Использование анализаторов событий CLR и ядра с подходящими обратными вызовами делает получение данных от ETW легким и приятным делом. Конечно, мы можем наблюдать за событиями, относящимися к нашему собственному процессу, фильтруя входящие события по полю `ProcessID`. Это позволяет организовать автомониторинг процесса с очень низкими накладными расходами (в предположении, что мы тщательно отбираем поставщиков и ключевые слова, чтобы не утонуть в потоке входящих событий).

Кроме того, с помощью TraceEvent мы можем воспользоваться тем, что ETW может записывать стеки вызовов в момент события. Для этого следует использовать «высокоуровневый» тип интерпретатора сеанса – `TraceLog`. Если для интересующих нас событий включена регистрация стеков, то мы можем вызвать метод `CallStack()` у полученного объекта с данными и получить коллекцию кадров стека. Работающий пример записи стеков вызовов можно найти в примерах самой библиотеки TraceEvent. Напомним также, что включение сбора стеков вызовов заметно увеличивает накладные расходы сеанса, так что этот режим следует использовать с осторожностью.

Итак, мы описали все возможности мониторинга использования памяти изнутри процесса.

- Мы можем наблюдать за выделением памяти в каждом потоке с помощью метода `GC.GetAllocatedBytesForCurrentThread` (см. листинг 15.10 выше в этой главе). Поверх этой функциональности можно надстроить сбор статистических данных обо всем процессе, собирая данные с каждого потока. Но помните, что это всего лишь информация о выделении памяти, а о том, какая часть выделенной памяти по-прежнему занята, мы ничего не узнаем. Поэтому мы не можем сказать о том, сколько всего памяти потребляет процесс. В случае .NET Framework для той же цели можно использовать свойство `MonitoringTotalAllocatedMemorySize` объекта `AppDomain` (листинг 15.11 выше).
- Мы можем наблюдать за размером памяти, занятой всеми управляемыми объектами (за вычетом фрагментированной памяти) во всех поколениях, для этого предназначен метод `GC.GetTotalMemory` (листинг 15.6). Как уже объяснялось, это очень полезная информация, но без учета фрагментации и суммарной памяти, занятой управляемой кучей, она слабо соотносится со взглядом операционной системы на потребление памяти процессом. Однако это неплохой способ заметить утечку памяти, если в управляемой куче образуется все больше и больше достижимых объектов. Дополнительно к измерению `GC.GetTotalMemory` можно наблюдать за всей памятью процесса с помощью таких свойств объекта `Process`, как `WorkingSet64` или `PrivateMemorySize64`.
- Мы можем наблюдать за счетчиками производительности из группы «Память CLR .NET» («.NET CLR Memory») для собственного процесса. Это дает богатую информацию о процессе (размеры поколений, потребление виртуальной памяти и т. д.), правда, получать эту информацию можно не более одного раза в секунду, но в большинстве случаев этого достаточно. Основной недостаток заключается в том, что счетчики производительности поддерживаются только для .NET Framework в Windows.
- Мы можем наблюдать за относящимися к GC событиями ETW с помощью библиотеки TraceEvent. Она дает гораздо более точную и глубокую картину

происходящего в процессе, поскольку, как мы неоднократно видели в этой книге, ETW предоставляет огромный объем информации. Величина накладных расходов ETW пропорциональна количеству собираемых событий. Наблюдение за не особенно частыми событиями начала и окончания GC и GCHeapStats – разумный подход к получению высокоуровневой информации о памяти.

- Мы можем подключить библиотеку ClrMD к своему процессу в пассивном режиме, что дает нам разнообразную информацию об управляемой куче (о распределении памяти по сегментам, об объектах и ссылках между ними, о корнях, об очередях финализации и т. д.). Это полезный подход к диагностике в отладочной сборке, но я советую хорошенько подумать, прежде чем включать его в выпускную сборку (Release build). Напомню, что автоподключение в пассивном режиме не поддерживается разработчиками библиотеки ClrMD, так что это рискованно и чревато странными проблемами.

Пользовательский сборщик мусора

Начиная с .NET Core 2.1 связь между сборщиком мусора и самим движком выполнения значительно ослабла. До этой версии код сборщика мусора был довольно сильно связан с остальным кодом CoreCLR. Но в .NET Core 2.1 была введена концепция *локального GC*, означающая, что теперь среда выполнения может использовать GC, распространяемый в виде отдельного dll-файла, т. е. сборщик мусора стал подключаемым. Чтобы подключить собственный GC, достаточно установить одну переменную среды (листинг 15.36).

Листинг 15.36 ❖ Задание переменной среды для замены реализации GC

```
set COMPlus_GCName=f:\GithubProjects\CoreCLR.ZeroGC\x64\Release\ZeroGC.dll
```

На этапе инициализации .NET Core видит эту переменную среды и пытается загрузить код GC из указанной библиотеки вместо загрузки встроенного GC. Реализация пользовательского GC может полностью отличаться от реализации по умолчанию. Такие понятия, как поколение, сегмент, распределитель и финализация, в пользовательском GC могут отсутствовать.

Простейшая реализация локального GC не очень сложна. Чтобы код скомпилировался, нужно всего несколько файлов, взятых непосредственно из кода CoreCLR: `debugmacros.h`, `gcenv.base.h` и `gcinterface.h`. Заметим, что для краткости ниже приведены только наиболее показательные части кода. Полный работающий пример имеется в сопроводительном репозитории.

Пользовательская библиотека GC обязана предоставить реализацию только для двух обязательных функций, вызываемых на этапе инициализации CoreCLR: `GC_Initialize` и `GC_VersionInfo` (листинг 15.37). Первая должна указывать на пользовательские реализации двух важнейших интерфейсов: `IGCHeap` и `IGCHandleManager`, а вторая используется для обеспечения обратной совместимости, чтобы можно было указать, какая версия среды выполнения (точнее, ее интерфейса с GC) нужна нашему сборщику мусора.

Листинг 15.37 ❖ Две обязательные экспортируемые функции в библиотеке локальной GC

```
extern "C" DLLEXPORT HRESULT
GC_Initialize(
   /* In */ IGCToCLR* clrToGC,
   /* Out */ IGCHeap** gcHeap,
   /* Out */ IGCHandleManager** gcHandleManager,
   /* Out */ GcDacVars* gcDacVars
)
{
   IGCHeap* heap = new ZeroGCHeap(clrToGC);
   IGCHandleManager* handleManager = new ZeroGCHandleManager();
   *gcHeap = heap;
   *gcHandleManager = handleManager;
   return S_OK;
}

extern "C" DLLEXPORT void
GC_VersionInfo(
   /* Out */ VersionInfo* result
)
{
   result->MajorVersion = GC_INTERFACE_MAJOR_VERSION;
   result->MinorVersion = GC_INTERFACE_MINOR_VERSION;
   result->BuildVersion = 0;
   result->Name = "Zero GC";
}
```

Дополнительно мы должны сохранить переданный адрес интерфейса `IGCToCLR`, который используется для взаимодействия с CLR из кода нашего GC. Он содержит много методов, перечислим только самые интересные:

- `SuspendEE` и `RestartEE` просят среду выполнения приостановить и возобновить управляемые потоки по указанной причине (их можно использовать для реализации неконкурентных частей пользовательского GC);
- `GcScanRoots` обходит стеки всех управляемых потоков и вызывает заданную функцию `promote_func` для всех корней GC, встретившихся в стеке (этот метод необходим на этапе пометки);
- `GcStartWork` и `GcDone` информирует среду выполнения о начале и завершении сборки мусора.

Пользовательская реализация интерфейса `IGCHeap` содержит основную функциональность сборки мусора (листинг 15.38). Нам предстоит реализовать аж 71 метод! Впрочем, не все они должны иметь настоящую реализацию, поскольку объявлены в расчете на дизайн встроенного GC. Поэтому мы предоставим фиктивные реализации таких методов, как `SetGcLatencyMode` или `SetLOHCompactionMode`, т. к. в нашем GC нет понятий режима задержки и LOH.

Листинг 15.38 ❖ Фрагмент пользовательской реализации `IGCHeap`

```
class ZeroGCHeap : public IGCHeap
{
private:
```

```
   IGCToCLR* gcToCLR;
public:
   ZeroGCHeap(IGCToCLR* gcToCLR)
   {
      this->gcToCLR = gcToCLR;
   }

   // Объявлены в IGCHeap
   ...
}
```

На верхнем уровне IGCHeap находятся методы для выделения памяти ((IGCHeap::Alloc) и сборки мусора (IGCHeap::GarbageCollect). Простейший *нулевой GC* (который умеет только выделять память для объектов, но никогда не возвращает ее) можно было бы реализовать, как показано в листинге 15.39. Отметим, что наш пользовательский GC не различает «малых» и «больших» объектов (а следовательно, SOH и LOH). Мы можем выделять память как угодно вне зависимости от размера объекта, например всегда использовать Heap API и стандартную функцию calloc.

Листинг 15.39 ❖ Примеры пользовательской реализации двух верхнеуровневых методов IGCHeap

```
class ObjHeader
{
private:
#ifdef _WIN64
   DWORD m_alignpad;
#endif // _WIN64
   DWORD m_SyncBlockValue;
};

Object * ZeroGCHeap::Alloc(gc_alloc_context * acontext, size_t size, uint32_t flags)
{
   int sizeWithHeader = size + sizeof(ObjHeader);
   ObjHeader* address = (ObjHeader*)calloc(sizeWithHeader, sizeof(char*));
   return (Object*)(address + 1);
}

HRESULT ZeroGCHeap::GarbageCollect(int generation, bool low_memory_p, int mode)
{
   return NOERROR;
}
```

Забавно смотреть на однострочный метод GarbageCollect – тот, который подразумеваем по умолчанию. .NET GC занимает несколько тысяч строк, описанных на сотнях страниц этой книги. Его возможности ограничены только вашим воображением. Давайте, реализуйте собственный GC!

Взявшись за написание пользовательского GC, мы должны заменить всю стандартную функциональность сборки мусора. Не так-то просто «чуть-чуть» модифицировать поведение по умолчанию. Но если взять весь код встроенного GC и опубликовать его виде автономной библиотеки, то эта задача стала бы гораздо проще.

Поскольку барьеры записи (write barriers) представляют собой специальным образом обрабатываемые функции, написанные на языке ассемблера и вставляемые JIT-компилятором, в настоящее время не существует API для их подмены. Как мы помним из главы 5, барьеры записи отвечают за обновление таблиц карт, поэтому они должны существовать, даже если не используются в нашей реализации. Взгляните на метод `ZeroGCHeap::Initialize` в сопроводительном примере – там показано, как путем манипулирования младшими и старшими адресами эфемерного сегмента можно сконфигурировать метод `IGCToCLR::StompWriteBarrier` так, чтобы он не использовался. И даже если в пользовательском GC нет различий между режимом рабочей станции и серверным режимом, из-за барьеров записи такое разделение все равно существует: лишь в режиме рабочей станции барьер записи проверяет границы эфемерного сегмента (см. листинг 5.8 в главе 5), поэтому мы можем использовать его, чтобы опустить обновление таблицы карт. Однако в серверном режиме наш пользовательский GC вызовет аварийную остановку среды выполнения, поскольку используется функция `JIT_WriteBarrier_SVR64`, которая требует действительного адреса таблицы карт.

Заметим, что фиктивные реализации интерфейсов `IGCHandleManager` и `IGCHandleStore` для краткости опущены. Вы можете ознакомиться с их кодом, заглянув в реализацию Zero GC, прилагаемую к этой книге.

Резюме

В этой главе описаны различные способы программного управления и мониторинга использования памяти в .NET. Вооруженные знаниями, полученными в предыдущих главах, мы вполне можем написать код, в котором эти возможности используются. Как вы имели случай заметить, знание внутреннего устройства CLR и GC часто оказывается полезным, а то и необходимым, для правильного конфигурирования библиотек и интерпретации полученных от них данных.

Сначала мы привели полный перечень статических методов и свойств класса `GC`, подводя итог тому, что уже знали о его возможностях, и дополнив наши знания (например, информацией об уведомлениях GC). Класс `GC` использовался в книге очень часто, так что вы, вероятно, понимаете, насколько он может быть полезен в различных ситуациях. Из всего, что описано в этой главе, класс `GC` (и несколько вспомогательных классов) чаще всего применяется в повседневной работе программиста.

Затем мы рассказали о размещении CLR и наиболее интересных интерфейсах, имеющих отношение к управлению памятью, чтобы показать, что можно сделать с их помощью. Я не думаю, что вы так уж часто будете размещать CLR в собственных проектах, но хотел показать, как это делается, чтобы расширить ваш арсенал. Быть может, вам приходится вызывать управляемый код из неуправляемых приложений (как в SQL Server, который позволяет писать скрипты на .NET-совместимых языках), тогда умение настраивать способ использования памяти размещенной CLR (а также некоторые средства мониторинга) может пригодиться.

Мы познакомились с двумя замечательными библиотеками, ClrMD и EventTrace, которые служат для диагностики и мониторинга .NET-процессов (включая и ваш собственный процесс в случае автомониторинга). Вместе или по отдельности, они позволяют получить очень подробную информацию о среде выполнения .NET и поведении вашего приложения. Хотя они активно используются при

реализации различных инструментов диагностики, можно рассмотреть возможность их применения для автомониторинга, поскольку накладные расходы сравнительно невелики (особенно соблазнительно использовать их до развертывания в промышленной среде).

Для любопытствующих в последнем разделе этой главы рассказано о новой возможности, пока реализованной только в .NET Core 2.1, – полной подмене реализации GC. Полагаю, есть ирония в том, чтобы завершить книгу, посвященную исключительно описанию встроенного по умолчанию сборщика мусора, словами о том, что теперь его можно убрать и заменить чем-то совершенно другим. Настоятельно рекомендую вам поэкспериментировать со сборщиком Zero GC, включенным в сопроводительные материалы к книге как пример такого пользовательского GC. Учитывая знания, полученные после прочтения этой книги, в т. ч. теоретического введения в первых главах, у вас есть все необходимое для написания собственной, не столь тривиальной, как Zero GC, реализации сборщика мусора!

Предметный указатель

Символы

.NET
  варианты, 203
  детали внутреннего устройства, 205
  заблуждения, 208
  управляемый код, 206
.NET, управление памятью
  бимодальное распределение, 130
  вмешательство, 125
  выборка, 126
  граф объектов, 127
  дамп памяти, 133
  дерево вызовов, 126
  динамическая отладка, 134
  задержка и пропускная способность, 132
  измерения, 124
  инструменты, 123
    Linux, 200
    Windows, 199
  медиана, перцентиль, гистограмма, 129
  многомодальное распределение, 130
  мониторинг, 133
  накладные расходы, 125
  нормальное распределение, 130
  операционная система, 124
  статистика, 129
  трассировка, 126
  Энскомба квартет, 129

A

AccessViolationException, 26
Address Windowing Extensions (AWE), 107
Allocator.Allocate(amount), метод, 329
AMD CodeAnalyst Performance Analyzer, 183
AppDynamics, 185

B

BenchmarkDotNet, 174
byref-подобные типы, 678

C

Clojure, 71
COBOL (Common Business Language), 40
Common Trace Format (CTF), 189
CoreCLR в Linux
  perfcollect, 187
  Trace Compass, 189
  дамп памяти, 198
  механизмы, 186
CPU-Z, 81

D

DebugDiag, инструмент, 171
Dynatrace, 185

E

Eclipse Trace Compass, 189
  CoreCLR.GC.collections, 191
  CoreCLR.GC.generations.ranges, 196
  CoreCLR.threads.state, 193
  окончательные результаты, 197
  открытие файла, 189

H

Heap API, 107

I

IMemoryOwner<T>, 719
Intel VTune Amplifier, 183

J

JetBrains DotMemory, 180

M

Memory<T>, 716
MESI, протокол, 96

O

OutOfMemoryException, исключение, 353

P

perfcollect, скрипт, 187
PerfView
  анализ данных, 165
  главные действия, 163
  задание пути к символам, 163
  запуск, 163
  описание, 162
  сбор данных, 163
  снимки памяти, 168
ProcDump, инструмент, 171

R

RedGate ANTS Memory Profiler, 182

S

Scitech .NET Memory Profiler, 178
SGen, сборщик мусора, 66
Span<T>, 702
SuperBenchmarker, 154

U

unmanaged, ограничение, 694
Unsafe, класс, 726

V

Virtual API, 107
Visual Studio, 176
VMMap, 135

W

WinDbg, 171
  msos, утилита, 174
  расширения, 172
  установка, 172
Windows Driver Kit (WDK), 172
Windows Performance Analyzer
  SuperBenchmarker, 154
  метки стеков, 161
  области интереса, 159
  описание, 154
  открытие файла и настройка, 154
  пламенные диаграммы, 160
  пользовательские графики, 161
  профили, 161
Windows Performance Recorder, 152

Y

yield return, 386

А

Аварийный дамп, 134
Автоматическая вычислительная машина (ACE), 36
Автоматический вычислитель, управляемый последовательностями, 29
Адресная арифметика, 44
Аккумулятор, 30
Амдала закон, 133
Анализ локальности, 242
Анатомия сегментов и кучи, 321
Арифметика указателей, 650
Арифметико-логическое устройство (АЛУ), 32
Архитектура компьютера, 76
Архитектура с неравномерным доступом к памяти (NUMA), 118
Асинхронный закрепленный описатель, 470
Ассемблерный код, 33

Б

Безопасная точка, 440
Бимодальное распределение, 130
Буфер фиксированного размера, 680
Быстрый Span, 714

В

Виртуальная машина Java (JVM), 42
Виртуальная память, 100
Виртуальное адресное пространство, 107
Висячий указатель, 50
Внутренние ячейки памяти, 78
Внутренний указатель, 449
Возвращаемое ссылочное значение (ref return), 652
Волокна, 55
Временная локальность, 85
Выборка, 126
Выбранное поколение, 402
Выделение памяти
  введение, 329
  в куче больших объектов, 347
  в куче малых объектов, 343
  в стеке, 356
  избегание, 358
  из списка свободных блоков, 337
  кортежи, 362
  сдвигом указателя, 330
  скрытое, 381
    в библиотеках, 389
  создание большого числа объектов, 373
  создание массивов, 366
  создание потоков, 371
  указатель, 330
  явное, 360

Г

Гипотеза о поколениях, 285
Граф объектов, 127
  поверхностный размер, 128
  подграф зависимостей, 128
  полный размер, 128
  собственный подграф, 128
  собственный размер, 129
Группы процессоров, 119

Д

Дамп памяти, 133, 198

Деконструкция кортежей, 364
Дерево вызовов, 126
Динамическая отладка, 134
Динамические оперативные запоминающие устройства (DRAM), 82
Динамическое выделение памяти, 45
Домен приложений (AppDomain), 214
Достижимость объекта, 57

Ж

Жизненные циклы объекта и ресурса, 575

З

Зависимые описатели, 632
Задержка доступа к памяти, 93
Задержка и пропускная способность, 132
Заимствование пометки (mark stealing), 553
Замыкания, 383
Запоминающее устройство с произвольным доступом, 31
Запомненные наборы (Remembered sets), 292

И

Индексирование перемещаемого фиксированного буфера, 682
Интернирование строк, 252
Интерфейсы прикладного программирования (API)
  ClrMD, 782
  GC API, 759
  TraceEvent, 787
Информация для GC (GC info), 441

К

Карта строковых литералов, 254
Кеширование, 80
Когерентность кешей, 96
Коммерческие инструменты
  AMD CodeAnalyst Performance Analyzer, 183
  Dynatrace и AppDynamics, 185
  Intel VTune Amplifier, 183
  JetBrains DotMemory, 180
  RedGate ANTS Memory Profiler, 182
  Scitech .NET Memory Profiler, 178
  Visual Studio, 176
Компилятор, 33
Компиляция, 33
Компьютер с хранимой программой, 32
Конфигурационная рукоять (Configuration Knob), 538
Корни, 57
Кратчайший путь к корню, 128
Критические финализаторы, 585
Куча, 45
  балансировка, 351
  динамическое выделение памяти, 45
  и стек, 47
  несбалансированная, 352
  освобождение, 45
  фрагментация, 45

Л

Линии адреса, 82
Литтла закон, 132
Локальная память потока, 334, 638
Локальность типов данных, 266
Локальные переменные, 450
Локальный буфер выделения памяти потока, 334

М

Массив параметров, 387
Медленный Span, 712
Межпоколенческие ссылки, 295
Механизм одновременной многопоточности, 95
Модификатор, 54
Модифицированная гарвардская архитектура, 93
Модули памяти, 79

Н

Наблюдение за записью в память, 118
Недетерминированная финализация, 576
Неоднородная память, 32
Неотслеживаемые корни, 465
Непопадание в кеш, 85
Непреобразуемые (blittable) типы, 698
Неуправляемые типы, 358
Нулевой сборщик мусора, 57

О

Области памяти процесса
  VMMap, 216
  измерение, 220
Оборудование
  архитектура компьютера, 76
  микросхема памяти DDR4, 79
  память, 81
  современная архитектура, 80
  центральный процессор (ЦП)
    выравнивание данных, 90
    иерархический кеш, 92

иерархический кеш в многоядерных процессорах, 95
кеш, 84
локальность данных, 85
некешируемый доступ, 90
попадание и непопадание в кеш, 85
реализация кеша, 86
упреждающая выборка, 91
Общеязыковая инфраструктура (CLI), 203
Общеязыковая среда выполнения (CLR), 205
Общий промежуточный язык (CIL), 205
Операционная система (ОС)
Linux
организация памяти, 116
управление памятью, 114
Windows
организация памяти, 112
управление памятью, 107
большие страницы, 104
виртуальная память, 100
диспетчер памяти, 100
структура памяти, 105
фрагментация виртуальной памяти, 105
Отслеживающий сборщик мусора, 63
Очередь закрепленных заполненных блоков, 495
Ошибка сегментации, 49

П

Передача по ссылке, 264
Повторное использование сегментов, 324
Подсчет ссылок, 58
висячий указатель, 62
достоинства, 62
жизнеспособность объекта, 58
модификатор, 59
недостатки, 62
обработка исключений, 61
псевдокод, 58
умные (интеллектуальные) указатели, 60
циклические ссылки, 60
Полная сборка мусора, 286
Получение ресурса есть инициализация (RAII), 61
Пользовательское пространство, 105
Помеченная карта, 299
Попадание в кеш, 85
Послеаварийный анализ, 134
Постоянные ссылочные переменные (readonly ref), 654
Постоянные структуры, 676
Потоковые данные, 642
Привязка к потоку, 334
Приоритетная GC (Foreground GC), 544
Проектирование ПО, ориентированное на данные, 731
Промежуточный язык (IL), 42
Пространственная локальность, 86
Пространство ядра, 105
Профилирование GC, 412
Прямой доступ к памяти (ПДП), 81

Р

Разделение
по времени жизни, 284
по размеру, 279
физическое, 306
Размеченные объединения (discriminated unions), 690
Ранняя сборка корней, 456
Распаковка, 263
Распределитель, 55
Регистровая машина, 34

С

Сборка, 213
Сборка мусора
выбор варианта GC, 562
выбор поколения, 444
данные для настройки производительности, 414
инициаторы, 428
конкурентный режим, 536
конфигурирование режимов, 536
на основе поколений, 284
неконкурентный режим, 535
описание режимов, 540
очистка, 513
пример процесса, 402
приостановка движка выполнения, 440
приостановка и накладные расходы GC, 538
режим рабочей станции, 533
режимы, 400
режимы задержки, 556
серверный режим, 534
уплотнение, 69, 515
шаги процесса, 408
этап планирования
куча больших объектов, 509
куча малых объектов, 486
принятие решения об уплотнении, 511
этап пометки
анализ утечек памяти, 476
внутренние корни GC, 469
корни – локальные переменные, 450
корни – описатели GC, 470

корни финализации, 468
обход и пометка объектов, 449
Сборщик мусора Бема–Демерса–Вейзера, 66
Сборщик мусора (GC), 56
пользовательский, 790
управляемая куча, 277
Свободное хранилище, 45
Связки карт, 303
Сегменты памяти, 106
Семантика владения, 719
Симметричная многопроцессорность, 118
Синтаксис деструктора, 577
«Система, компонент, сущность» (ECS), шаблон проектирования, 741
Система типов
время жизни, 231
идентичность, 232
неизменяемые типы, 232
разделение, 231
таблица методов, 230
хранение, 232
Системная шина, 76
Слабые события, 620
Слабый описатель (weak handle), 614
Список наблюдения за записью, 549
Ссылочная локальная переменная (ref local), 651
Ссылочные структуры, 677
Ссылочные типы, 241
классы, 242
создание нового объекта, 341
Статические данные, 269
внутреннее устройство, 270
статические поля, 269
Статические оперативные запоминающие устройства (SRAM), 82
Статическое выделение памяти, 33
Стек пометки, 449
Стратегии разделения памяти, 278
Строки, 246
Строки кеша, 86
Структуры
определение, 236
упаковка, 237
хранение, 237
Счетчики производительности
архитектура, 136
атрибуты, 136
достоинства и недостатки, 136
категория Память CLR .NET, 137
пул приложений, 141
системный монитор, 138
Счетчик команд, 32

Т

Таблицы карт (Card tables), 298
Типы значений
определение, 233
структуры, 235
хранение, 234
Типы, передаваемые по ссылке (byref), 651
Трассировка, 126
Трассировка событий для Windows (ETW), 134, 142

У

Указатели заполнения, 589
Указатель, 43
выделения памяти, 330
команд, 32
Умные (интеллектуальные) указатели, 60
Уничтожаемые объекты, 605
Упаковка, 259
Управление памятью
автоматическое, 52
ручное, 47
ALGOL, 48
висячий указатель, 50
в программе на C, 48
в программе на C++, 51
проблемы, 49
функция free, 50
Управление производительностью приложения (APM), 185
Управляемые указатели, 650

Ф

Финализатор, 577
Финализация, 577
Финализируемые объекты, 578
Фрагментация, порог виртуальной памяти, 105

Ц

Цели (уровни) оптимизации задержки, 562

Ш

Шина памяти, 76

Э

Энскомба квартет, 129
Эфемерный сегмент, 307

Я

Явная детерминированная финализация, 603

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Конрад Кокоса

Управление памятью в .NET для профессионалов

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура PT Serif. Печать офсетная.
Усл. печ. л. 65. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**