

ПУТЬ RUBY

ТРЕТЬЕ ИЗДАНИЕ

Хэл Фултон

совместно с Андрэ Арко



Содержание

Предисловие 22

Благодарности 26

Об авторах 29

Введение 30

Глава 1. Обзор Ruby 39

1.1. Введение в объектно-ориентированное программирование 40
1.1.1. Что такое объект 40
1.1.2. Наследование 41
1.1.3. Полиморфизм 43
1.1.4. Еще немного терминов 44
1.2. Базовый синтаксис и семантика Ruby 45
1.2.1. Ключевые слова и идентификаторы 46
1.2.2. Комментарии и встроенная документация 47
1.2.3. Константы, переменные и типы 47
1.2.4. Операторы и приоритеты 49
1.2.5. Пример программы 50
1.2.6. Циклы и ветвление 52
1.2.7. Исключения 57
1.3. ООП в Ruby 59
1.3.1. Объекты 59
1.3.2. Встроенные классы 60
1.3.3. Модули и классы-примеси 61
1.3.4. Создание классов 62
1.3.5. Методы и атрибуты 66
1.4. Динамические аспекты Ruby 68
1.4.1. Кодирование во время выполнения 68
1.4.2. Отражение 70
1.4.3. Отсутствующие методы 71
1.4.4. Сборка мусора 71
1.5. Потренируйте свою интуицию: что следует запомнить 72

1.5.1. Синтаксис....72
1.5.2. Отличия от других языков....74
1.5.3. Предложение case в Ruby....77
1.5.4. Рубизмы и идиомы....79
1.5.5. Ориентация на выражения и прочие вопросы....84
1.6. Жаргон Ruby....86
1.7. Заключение....89

Глава 2. Строки....90

2.1. Представление обычных строк....90
2.2. Альтернативная нотация для представления строк....91
2.3. Встроенные документы....92
2.4. Получение длины строки....93
2.5. Построчная обработка....93
2.6. Побайтовая обработка....94
2.7. Специализированное сравнение строк....94
2.8. Разбиение строки на лексемы....96
2.9. Форматирование строк....97
2.10. Строки в качестве объектов ввода-вывода....98
2.11. Управление регистром....98
2.12. Вычленение и замена подстрок....99
2.13. Подстановка в строках....101
2.14. Поиск в строке....102
2.15. Преобразование символов в коды ASCII и обратно....103
2.16. Явные и неявные преобразования....103
2.17. Дописывание в конец строки....105
2.18. Удаление хвостовых символов новой строки и прочих....105
2.19. Убирание лишних пропусков....106
2.20. Повтор строк....106
2.21. Включение выражений в строку....107
2.22. Отложенная интерполяция....107
2.23. Разбор данных, разделенных запятыми....108
2.24. Преобразование строки в число (десятичное или иное)....108
2.25. Кодирование и декодирование строк в кодировке rot13....110
2.26. Шифрование строк....110
2.27. Сжатие строк....111
2.28. Подсчет числа символов в строке....112
2.29. Обращение строки....112
2.30. Удаление дубликатов....112
2.31. Удаление заданных символов....113
2.32. Печать специальных символов....113

2.33. Генерирование последовательности строк 113
2.34. Вычисление 32-разрядного CRC 114
2.35. Вычисление SHA-256-свертки строки 114
2.36. Вычисление расстояния Левенштейна между двумя строками 115
2.37. base64-кодирование и декодирование строк 117
2.38. Замена символов табуляции пробелами и сворачивание пробелов в табуляторы 117
2.39. Перенос строк по словам 118
2.40. Заключение 119

Глава 3. Регулярные выражения 120

3.1. Синтаксис регулярных выражений 120
3.2. Компиляция регулярных выражений 122
3.3. Экранирование специальных символов 123
3.4. Якоря 123
3.5. Кванторы 124
3.6. Позитивное и негативное заглядывание вперед 126
3.7. Позитивное и негативное оглядывание назад 127
3.8. Обратные ссылки 128
3.9. Именованные соответствия 130
3.10. Классы символов 132
3.11. Обобщенные регулярные выражения 133
3.12. Сопоставление точки символу конца строки 134
3.13. Внутренние модификаторы 134
3.14. Внутренние подвыражения 135
3.14.1. Рекурсия в регулярных выражениях 135
3.15. Примеры регулярных выражений 136
3.15.1. Сопоставление с IP-адресом 136
3.15.2. Сопоставление с парой «ключ-значение» 137
3.15.3. Сопоставление с числами, записанными римскими цифрами 138
3.15.4. Сопоставление с числовыми константами 138
3.15.5. Сопоставление с датой и временем 139
3.15.6. Обнаружение удвоенных слов в тексте 140
3.15.7. Поиск слов, записанных одними заглавными буквами 140
3.15.8. Сопоставление с номером версии 140
3.15.9. Еще несколько образцов 140
3.16. Заключение 141

Глава 4. Интернационализация в Ruby 142
4.1. Исторические сведения и терминология 143
4.2. Работа с кодировками 147
4.2.1. Нормализация 148
4.2.2. Преобразование из одной кодировки в другую 151
4.2.3. Транслитерация 152
4.2.4. Упорядочение строк 152
4.3. Перевод 155
4.3.1. Значения по умолчанию 157
4.3.2. Пространства имен 158
4.3.3. Интерполяция 158
4.3.4. Формы множественного числа 159
4.4. Локализованное форматирование 160
4.4.1. Дата и время 161
4.4.2. Числа 161
4.4.3. Денежные величины 162
4.5. Заключение 162

Глава 5. Численные методы 163
5.1. Представление чисел в языке Ruby 163
5.2. Основные операции над числами 164
5.3. Округление чисел с плавающей точкой 165
5.4. Сравнение чисел с плавающей точкой 167
5.5. Форматирование чисел для вывода 168
5.6. Вставка разделителей при форматировании чисел 169
5.7. Работа с очень большими числами 169
5.8. Использование класса BigDecimal 169
5.9. Работа с рациональными числами 171
5.10. Перемножение матриц 172
5.11. Комплексные числа 176
5.12. Библиотека mathn 176
5.13. Разложение на простые множители, вычисление НОД и НОК 177
5.14. Простые числа 178
5.15. Явные и неявные преобразования чисел 179
5.16. Приведение числовых значений 180
5.17. Поразрядные операции над числами 181
5.18. Преобразование системы счисления 182
5.19. Извлечение кубических корней, корней четвертой степени и т.д. 183

5.20. Определение порядка байтов 183
5.21. Численное вычисление определенного интеграла 184
5.22. Тригонометрия в градусах, радианах и градах 185
5.23. Вычисление логарифмов по произвольному основанию ...186
5.24. Вычисление среднего, медианы и моды набора данных ...187
5.25. Дисперсия и стандартное отклонение 188
5.26. Вычисление коэффициента корреляции 189
5.27. Генерирование случайных чисел 190
5.28. Кэширование функций с помощью метода memoize 191
5.29. Заключение 192

Глава 6. Символы и диапазоны 193

6.1. Символы....193
6.1.1. Символы как перечисления 195
6.1.2. Символы как метазначения....195
6.1.3. Символы, переменные и методы 196
6.1.4. Преобразование строки в символ и обратно....196
6.2. Диапазоны....197
6.2.1. Открытые и замкнутые диапазоны 198
6.2.2. Нахождение границ диапазона 198
6.2.3. Обход диапазона 198
6.2.4. Проверка принадлежности диапазону 199
6.2.5. Преобразование в массив 200
6.2.6. Обратные диапазоны....200
6.2.7. Оператор переключения....200
6.2.8. Нестандартные диапазоны 203
6.3. Заключение 205

Глава 7. Дата и время 207

7.1. Определение текущего момента времени 208
7.2. Работа с конкретными датами (после точки отсчета) 208
7.3. Определение дня недели....209
7.4. Определение даты Пасхи....209
7.5. Вычисление n-ого дня недели в месяце 210
7.6. Преобразование из секунд в более крупные единицы....211
7.7. Вычисление промежутка времени, прошедшего от точки отсчета 212
7.8. Високосные секунды 212
7.9. Определение порядкового номера дня в году....213
7.10. Контроль даты и времени 213
7.11. Определение недели в году 214

7.12. Проверка года на високосность 214
7.13. Определение часового пояса....215
7.14. Манипулирование временем без даты....215
7.15. Сравнение моментов времени....216
7.16. Прибавление интервала к моменту времени....216
7.17. Вычисление разности между двумя моментами времени....217
7.18. Работа с конкретными датами (до точки отсчета)....217
7.19. Взаимные преобразования объектов Date, Time и DateTime 217
7.20. Извлечение даты и времени из строки 218
7.21. Форматирование и печать даты и времени 219
7.22. Преобразование часовых поясов....219
7.23. Определение числа дней в месяце 220
7.24. Разбиение месяца на недели 220
7.25. Заключение 221

Глава 8. Массивы, хэши и другие перечисляемые структуры 223

8.1. Массивы....223
8.1.1. Создание и инициализация массива....224
8.1.2. Доступ к элементам массива и присваивание им значений....224
8.1.3. Определение размера массива 226
8.1.4. Сравнение массивов 226
8.1.5. Сортировка массива....228
8.1.6. Выборка из массива по заданному критерию 230
8.1.7. Специализированные функции индексирования....231
8.1.8. Реализация разреженной матрицы....233
8.1.9. Массивы как математические множества 233
8.1.10. Рандомизация массива 237
8.1.11. Многомерные массивы....237
8.1.12. Нахождение элементов, принадлежащих одному массиву и не принадлежащих другому....238
8.1.13. Преобразование или отображение массивов....238
8.1.14. Удаление элементов, равных nil, из массива 239
8.1.15. Удаление заданных элементов из массива 239
8.1.16. Конкатенирование массивов и добавление в конец массива....240
8.1.17. Использование массива в качестве стека или очереди....241

8.1.18. Обход массива 241
8.1.19. Преобразование массива в строку с разделителями 242
8.1.20. Обращение массива 242
8.1.21. Удаление дубликатов из массива 242
8.1.22. Чередование массивов 243
8.1.23. Вычисление частоты различных значений в массиве 243
8.1.24. Инвертирование массива для получения хэша 243
8.1.25. Синхронная сортировка нескольких массивов 244
8.1.26. Задание значения по умолчанию для новых элементов массива 245
8.2. Хэши 245
8.2.1. Создание нового хэша 246
8.2.2. Задание значения по умолчанию для хэша 247
8.2.3. Доступ к парам ключ-значение и добавление новых пар 247
8.2.4. Удаление пар ключ-значение 248
8.2.5. Обход хэша 249
8.2.6. Инвертирование хэша 249
8.2.7. Поиск ключей и значений в хэше 250
8.2.8. Копирование хэша в массив 250
8.2.9. Выборка пар ключ-значение по заданному критерию 250
8.2.10. Сортировка хэша 251
8.2.11. Объединение двух хэшей 251
8.2.12. Создание хэша из массива 252
8.2.13. Вычисление разности и пересечения хэшей 252
8.2.14. Хэш как разреженная матрица 252
8.2.15. Реализация хэша с повторяющимися ключами 253
8.2.16. Другие операции с хэшем 256
8.3. Перечисляемые структуры в общем 256
8.3.1. Метод inject 257
8.3.2. Кванторы 258
8.3.3. Метод partition 258
8.3.4. Обход с группировкой 259
8.3.5. Преобразование в массив или множество 260
8.3.6. Перечислители 260
8.4. Дополнительные сведения о перечисляемых объектах 262
8.4.1. Поиск и выборка 262
8.4.2. Подсчет и сравнение 263

8.4.3. Итерирование 263
8.4.4. Извлечение и преобразование 264
8.4.5. Ленивые перечислители 265
8.5. Заключение 265

Глава 9. Более сложные структуры данных 266

9.1. Множества 266
9.1.1. Простые операции над множествами 267
9.1.2. Более сложные операции над множествами 268
9.2. Стеки и очереди 269
9.2.1. Более строгая реализация стека 271
9.2.2. Обнаружение несбалансированных скобок....271
9.2.3. Стек и рекурсия 272
9.2.4. Более строгая реализация очереди 274
9.3. Деревья....275
9.3.1. Реализация двоичного дерева....275
9.3.2. Сортировка с помощью двоичного дерева....277
9.3.3. Использование двоичного дерева как справочной таблицы 279
9.3.4. Преобразование дерева в строку или массив 279
9.4. Графы....280
9.4.1. Реализация графа в виде матрицы смежности 281
9.4.2. Является ли граф связным?....283
9.4.3. Есть ли в графе эйлеров цикл?....284
9.4.4. Есть ли в графе эйлеров путь? 285
9.4.5. Инструменты для работы с графами в Ruby 286
9.5. Заключение 286

Глава 10. Ввод-вывод и хранение данных 287

10.1. Файлы и каталоги 288
10.1.1. Открытие и закрытие файлов....288
10.1.2. Обновление файла 289
10.1.3. Дописывание в конец файла 290
10.1.4. Прямой доступ к файлу 290
10.1.5. Работа с двоичными файлами....291
10.1.6. Блокировка файлов 292
10.1.7. Простой ввод-вывод....293
10.1.8. Буферизованный и небуферизованный ввод-вывод....294
10.1.9. Манипулирование правами владения и разрешениями на доступ к файлу....295
10.1.10. Получение и установка временных меток....296

10.1.11. Проверка существования и получение размера файла 297
10.1.12. Опрос специальных свойств файла....298
10.1.13. Каналы....300
10.1.14. Специальные операции ввода-вывода....301
10.1.15. Неблокирующий ввод-вывод 302
10.1.16. Применение метода readpartial 302
10.1.17. Манипулирование путевыми именами 303
10.1.18. Класс Pathname 304
10.1.19. Манипулирование файлами на уровне команд....305
10.1.20. Ввод символов с клавиатуры 306
10.1.21. Чтение всего файла в память 306
10.1.22. Построчное чтение из файла....307
10.1.23. Побайтное и посимвольное чтение из файла 307
10.1.24. Работа со строкой как с файлом....308
10.1.25. Копирование потока 308
10.1.26. Работа с кодировками....308
10.1.27. Чтение данных, встроенных в текст программы 309
10.1.28. Чтение исходного текста программы 309
10.1.29. Работа с временными файлами 309
10.1.30. Получение и изменение текущего каталога....310
10.1.31. Изменение текущего корня....310
10.1.32. Обход каталога 311
10.1.33. Получение содержимого каталога....311
10.1.34. Создание цепочки каталогов....311
10.1.35. Рекурсивное удаление каталога....311
10.1.36. Поиск файлов и каталогов....311
10.2. Доступ к данным более высокого уровня 312
10.2.1. Простой маршалинг....313
10.2.2. «Глубокое копирование» с помощью метода Marshal....313
10.2.3. Более сложный маршалинг....314
10.2.4. Маршалинг в формате YAML 315
10.2.5. Сохранение данных с помощью библиотеки JSON 316
10.2.6. Работа с данными в формате CSV....317
10.2.7. SQLite3 как SQL-хранилище данных 319
10.3. Подключение к внешним базам данных....320
10.3.1. Подключение к базе данных MySQL 320
10.3.2. Подключение к базе данных PostreSQL 322
10.3.3. Объектно-реляционные отображения (ORM) 323

10.3.4. Подключение к хранилищу данных Redis 324
10.4. Заключение 326

Глава 11. ООП и динамические механизмы в Ruby 327

11.1. Рутинные объектно-ориентированные задачи 328
11.1.1. Применение нескольких конструкторов 328
11.1.2. Создание атрибутов экземпляра 329
11.1.3. Более сложные конструкторы 331
11.1.4. Создание атрибутов и методов уровня класса 333
11.1.5. Наследование суперклассу 336
11.1.6. Опрос класса объекта 338
11.1.7. Проверка объектов на равенство 340
11.1.8. Управление доступом к методам 341
11.1.9. Копирование объектов 343
11.1.10. Метод initialize_copy 344
11.1.11. Метод allocate 345
11.1.12. Модули 346
11.1.13. Трансформация или преобразование объектов 349
11.1.14. Классы, содержащие только данные (Struct) 351
11.1.15. Замораживание объектов 352
11.1.16. Использование метода tap в цепочке методов 353
11.2. Более сложные механизмы 354
11.2.1. Посылка объекту явного сообщения 354
11.2.2. Специализация отдельного объекта 356
11.2.3. Вложенные классы и модули 359
11.2.4. Создание параметрических классов 360
11.2.5. Хранение кода в виде объектов Proc 362
11.2.6. Хранение кода в виде объектов Method 364
11.2.7. Использование символов в качестве блоков 364
11.2.8. Как работает включение модулей? 365
11.2.9. Опознание параметров, заданных по умолчанию 367
11.2.10. Делегирование или перенаправление 367
11.2.11. Автоматическое определение методов чтения и установки на уровне класса 370
11.2.12. Поддержка различных стилей программирования ... 371
11.3. Динамические механизмы 373
11.3.1. Динамическая интерпретация кода 373
11.3.2. Метод const_get 375
11.3.3. Получение класса по имени 375
11.3.4. Метод define_method 375
11.3.5. Получение списка определенных сущностей 378

11.3.6. Удаление определений 380
11.3.7. Ссылки на несуществующие константы 382
11.3.8. Вызовы несуществующих методов 383
11.3.9. Повышение безопасности с помощью taint 384
11.3.10. Определение чистильщиков для объектов 386
11.4. Интроспекция программы 387
11.4.1. Обход пространства объектов 387
11.4.2. Просмотр стека вызовов....388
11.4.3. Отслеживание изменений в определении класса или объекта....388
11.4.4. Мониторинг выполнения программы 391
11.5. Заключение 393

Глава 12. Графические интерфейсы для Ruby 394

12.1. Shoes 4 395
12.1.1. Начало работы с Shoes 395
12.1.2. Интерактивная кнопка 396
12.1.3. Текст и поле ввода 396
12.1.4. Компоновка 397
12.1.5. Картинки и фигуры 399
12.1.6. События....400
12.1.7. Прочие замечания 400
12.2. Ruby/Tk....401
12.2.1. Обзор 401
12.2.2. Простое оконное приложение....402
12.2.3. Кнопки 403
12.2.4. Текстовые поля 407
12.2.5. Прочие виджеты 411
12.2.6. Дополнительные замечания 413
12.3. Ruby/GTK3 414
12.3.1. Обзор 414
12.3.2. Простое оконное приложение....415
12.3.3. Кнопки 416
12.3.4. Текстовые поля 417
12.3.5. Прочие виджеты 420
12.3.6. Дополнительные замечания 424
12.4. QtRuby 425
12.4.1. Обзор 425
12.4.2. Простое оконное приложение....426
12.4.3. Кнопки 426
12.4.4. Текстовые поля 428

12.4.5. Прочие виджеты 430
12.4.6. Дополнительные замечания 434
12.5. Swing 435
12.6. Другие библиотеки для создания графических интерфейсов 437
12.6.1. UNIX и X11 437
12.6.2. FXRuby (FOX) 437
12.6.3. RubyMotion для iOS и Mac OS X 437
12.6.4. Windows Win32 API 438
12.7. Заключение 438

Глава 13. Потоки и параллелизм 439

13.1. Создание потоков и манипулирование ими 440
13.1.1. Создание потоков 441
13.1.2. Доступ к поточно-локальным переменным 442
13.1.3. Опрос и изменение состояния потока 443
13.1.4. Назначение рандеву (и получение возвращенного значения) 446
13.1.5. Обработка исключений 447
13.1.6. Группы потоков 449
13.2. Синхронизация потоков 450
13.2.1. Простая синхронизация 451
13.2.2. Синхронизация доступа с помощью мьютекса 452
13.2.3. Встроенные классы очередей 454
13.2.4. Условные переменные 456
13.2.5. Другие способы синхронизации 457
13.2.6. Таймаут при выполнении операций 460
13.2.7. Ожидание события 462
13.2.8. Параллельный поиск в коллекции 463
13.2.9. Параллельное рекурсивное удаление 464
13.3. Волокна и кооперативная многозадачность 465
13.4. Заключение 467

Глава 14. Скрипты и системное администрирование 468

14.1. Запуск внешних программ 468
14.1.1. Методы system и exec 469
14.1.2. Запоминание вывода программы 469
14.1.3. Манипулирование процессами 470
14.1.4. Стандартный ввод и вывод 473
14.2. Флаги и аргументы в командной строке 474
14.2.1. Константа ARGV 474

14.2.2. Константа ARGF 474
14.2.3. Разбор флагов в командной строке 475
14.3. Библиотека Shell 476
14.3.1. Использование библиотеки Shell для перенаправления ввода-вывода 477
14.3.2. Дополнительные замечания по поводу библиотеки Shell 478
14.4. Переменные окружения 479
14.4.1. Чтение и установка переменных окружения 479
14.4.2. Хранение переменных окружения в виде массива или хэша 480
14.5. Работа с файлами, каталогами и деревьями 481
14.5.1. Несколько слов о текстовых фильтрах 481
14.5.2. Копирование дерева каталогов (с символическими ссылками) 481
14.5.3. Удаление файлов по времени модификации и другим критериям 483
14.5.4. Вычисление свободного места на диске 484
14.6. Различные скрипты 484
14.6.1. Распространение программ на Ruby 484
14.6.2. Подача входных данных Ruby по конвейеру 485
14.6.3. Работает ли Ruby в интерактивном режиме? 486
14.6.4. Определение текущей платформы или операционной системы 486
14.6.5. Модуль Etc 487
14.7. Заключение 487

Глава 15. Ruby и форматы данных 489

15.1. Разбор JSON 490
15.1.1. Обход JSON-данных 490
15.1.2. Типы данных, не представимые в JSON 491
15.1.3. Другие библиотеки для работы с JSON 492
15.2. Разбор XML (и HTML) 492
15.2.1. Разбор документа 493
15.2.2. Потоковый разбор 495
15.3. RSS и Atom 497
15.3.1. Разбор новостной ленты 497
15.3.2. Создание новостных лент 499
15.4. Обработка изображений с помощью RMagick 499
15.4.1. Типичные графические задачи 500
15.4.2. Специальные эффекты и преобразования 503

15.4.3. API рисования 505
15.5. Создание документов в формате PDF с помощью библиотеки Prawn 508
15.5.1. Основные концепции и приемы 509
15.5.2. Пример документа 509
15.6. Заключение 513

Глава 16. Тестирование и отладка 514
16.1. Тестирование с помощью RSpec 515
16.2. Тестирование с помощью Minitest 517
16.3. Тестирование с помощью Cucumber 522
16.4. Работа с отладчиком byebug 524
16.5. Отладка с помощью pry 527
16.6. Измерение производительности 528
16.7. Объекты форматированной печати 532
16.8. О том, что осталось за кадром 534
16.9. Заключение 534

Глава 17. Создание пакетов и распространение программ 535
17.1. Библиотеки и система Rubygems 535
17.1.1. Работа с RubyGems 536
17.1.2. Создание gem-пакетов 536
17.2. Управление зависимостями с помощью Bundler 537
17.2.1. Семантическое версионирование 539
17.2.2. Загрузка зависимостей из Git 539
17.2.3. Создание gem-пакетов с помощью Bundler 540
17.2.4. Частные gem-пакеты 540
17.3. Программа RDoc 541
17.3.1. Простая разметка 543
17.3.2. Создание улучшенной документации с помощью Yard 545
17.4. Заключение 546

Глава 18. Сетевое программирование 547
18.1. Сетевые серверы 549
18.1.1. Простой сервер: время дня 549
18.1.2. Реализация многопоточного сервера 550
18.1.3. Пример: сервер для игры в шахматы по сети 551
18.2. Сетевые клиенты 558
18.2.1. Получение истинно случайных чисел из веб 558

18.2.2. Запрос к официальному серверу времени 561
18.2.3. Взаимодействие с POP-сервером 562
18.2.4. Отправка почты по протоколу SMTP 563
18.2.5. Взаимодействие с IMAP-сервером 566
18.2.6. Кодирование и декодирование вложений 568
18.2.7. Пример: шлюз между почтой и конференциями 570
18.2.8. Получение веб-страницы с известным URL 575
18.2.9. Библиотека Open-URI 575
18.3. Заключение 576

Глава 19. Ruby и веб-приложения 577

19.1. HTTP-серверы 577
19.1.1. Простой HTTP-сервер 577
19.1.2. Rack и веб-серверы 579
19.2. Каркасы приложений 581
19.2.1. Маршрутизация в Sinatra 582
19.2.2. Маршрутизация в Rails 583
19.2.3. Параметры в Sinatra 585
19.2.4. Параметры в Rails 586
19.3. Хранение данных 586
19.3.1. Базы данных 587
19.3.2. Хранилища данных 589
19.4. Генерация HTML 589
19.4.1. ERB 590
19.4.2. Haml 592
19.4.3. Другие шаблонные системы 592
19.5. Конвейер активов 593
19.5.1. CSS и Sass 593
19.5.2. JavaScript и CoffeeScript 595
19.6. Предоставление веб-служб по протоколу HTTP 597
19.6.1. Использование JSON в API 597
19.6.2. REST API и его вариации 598
19.7. Генерация статических сайтов 599
19.7.1. Middleman 599
19.7.2. Другие генераторы статических сайтов 600
19.8. Заключение 601

Глава 20. Распределенный Ruby 602

20.1. Обзор: библиотека drb 602
20.2. Пример: эмуляция биржевой ленты 605
20.3. Rinda: пространство кортежей в Ruby 608

20.4. Обнаружение служб в распределенном Ruby 611
20.5. Заключение 613

Глава 21. Инструменты разработки для Ruby 614
21.1. Программа Rake 614
21.2. Оболочка irb 618
21.3. Основы pry 621
21.4. Утилита ri 623
21.5. Поддержка со стороны редакторов 623
21.5.1. Vim 624
21.5.2. Emacs 624
21.6. Менеджеры версий Ruby 625
21.6.1. Работа с rvm 625
21.6.2. Работа с rbenv 625
21.6.3. Работа с chruby 626
21.7. Заключение 627

Глава 22. Сообщество пользователей Ruby 628
22.1. Ресурсы в веб 628
22.2. Новостные группы и списки рассылки 628
22.3. Извещения об ошибках и предложения новых функций 629
22.4. Каналы IRC 629
22.5. Конференции по Ruby 630
22.6. Локальные группы пользователей Ruby 631
22.7. Заключение 631

Предметный указатель 632

Предисловие

Предисловие к третьему изданию

Вчера я читал статью на сайте Wired.com о моде в среде компьютерных фриков. Так там пишут, что если человек носит футболку с надписью Rubyconf 2012, значит, он хочет сообщить «Я работаю в Oracle».

Ну надо же! Далеко же мы ушли за последние 10 лет!

Было время, когда Ruby определенно выбивался из господствующих тенденций. Но теперь мы, похоже, в струе. Однако для этого пришлось пройти долгий и необычный путь.

По нынешним стандартам, чтобы войти в обиход, Ruby потребовалось много времени. Я читал эту книгу в 2005 году, и уже тогда первому изданию исполнилось четыре года. Тогда только-только намечался второй всплеск интереса к Ruby благодаря DHH и началу повальной увлеченности Rails. Казалось, что во всем англоговорящем мире не наберется и двух сотен человек, использующих Ruby. А первому изданию книги уже было четыре года. Вот насколько она опередила свое время.

В новом издании книги сохранен стиль, завоевавший ей признание у опытных программистов. В первых четырех длинных главах описываются основы объектно-ориентированного подхода и самого языка Ruby. Их обязательно должен прочитать всякий, кто не знаком с языком. Но изложение стремительное, без задержки на деталях – предполагается, что читатель уже знает, как создается программное обеспечение.

Последующие главы устроены иначе. Краткая предыстория, а затем мощный залп сведений о языке Ruby. Для иллюстрации обсуждаемой темы приводятся многочисленные фрагменты кода. Примеры можно вставлять в собственные программы практически без изменений – особенно когда вы дойдете до глав, в которых рассматриваются практические приложения.

Будет уместно сказать пару слов о себе. Я очень благодарен Хэлу за эту книгу и за то, как именно он ее написал. В 2005 году по договору с издательством Addison Wesley я начал работать над книгой об использовании Ruby on Rails на предприятии. То была моя первая попытка попробовать себя в роли автора и, сочинив две главы, я застрял. В то время Ruby и Rails можно было встретить на предприятиях лишь эпизодически, и я должен был постоянно напоминать себе, что пишу не беллетристику.

Обсудив варианты с редактором, мы решили, что будет правильнее отказаться от первоначальной идеи и подойти к книге по-другому. Книга «Путь Rails» стала попыткой осветить только зарождающийся каркас Ruby on Rails в духе, присущем этой книге. Я взял на вооружение краткий повествовательный стиль с изобилием приме-

ров кода. Вместо длинных листингов я чередовал код и комментарии, стремясь не переусердствовать, а просто проиллюстрировать смысл отдельных частей каркаса.

Как и в «Пути Ruby», я ставил себе целью добиться ширины охвата, а не глубины рассмотрения. Я хотел, чтобы «Путь Rails» занял постоянное место на столах серьезных разработчиков приложений под Rails. Я хотел, чтобы моя книга, как и «Путь Ruby», стала справочным пособием по умолчанию. В отличие от других книг по Rails, я полностью опустил вводный материал и проигнорировал интересы начинающих.

И это был успех! Будет справедливо сказать, что без книги Хэла не было бы и моей книги, а моя карьера сложилась бы не так удачно.

Но довольно ретроспективных поздравлений. Вернемся к дню сегодняшнему и только что вышедшему изданию «Пути Ruby», которое вы сейчас читаете. На этот раз к Хэлу присоединился безмерно талантливый Андрэ Арко[1]. Отличная получилась команда! Изрядно потрудившись, они привели текст в соответствие с последней версией всеми нами любимого языка Ruby.

Лично я хотел бы отметить следующие отличия от предыдущих изданий.

- Целая глава, посвященная углубленному рассмотрению новой подсистемы регулярных выражений Onigmo. Мне очень нравятся их красивые и краткие объяснения таких концепций, как позитивное и негативное заглядывание и оглядывание.
- В главе об интернационализации обсуждаются трудные вопросы, касающиеся кодировки объектов типа String и нормализации в Unicode. Блогеры уже много лет обсуждают различные аспекты этой темы, но наконец-то все собрано в одном месте.
- В главе о приложениях Ruby в веб авторам удалось соединить в одном кратком курсе, занимающем меньше 30 страниц, начальные сведения о Rack, Sinatra и Rails.

Я предсказываю этому изданию «Пути Ruby» не меньший успех, чем предыдущим. С удовольствием включаю его в нашу серию книг «Professional Ruby».

Оби Фернандес
15 сентября 2014

Предисловие ко второму изданию

В древнем Китае люди, в особенности философы, полагали, что под внешней оболочкой мира и любого существа скрыто нечто. Его нельзя ни объяснить, ни описать словами. Это нечто китайцы называли Тао, а японцы – До. На русский язык это слово можно перевести как Путь. Слово «до» входит в такие названия, как дзюдо, кендо, карате-до и айкидо. Это не просто боевые искусства, а целая философия и взгляд на жизнь.

---

[1] Желаете увидеть пример изобретательности Андрэ? Почитайте статью по адресу http://andre.arko.net/2014/06/27/rails-in-05-seconds/, где он описывает, как уменьшить время загрузки реального приложения Rails до 500 мс, а то и меньше.

Так и в языке программирования Ruby есть своя философия и способ мышления. Этот язык заставляет думать по-новому. Он помогает программистам получать удовольствие от своей работы. И не потому, что Ruby был создан в Японии, а потому что программирование стало важной частью существования (по крайней мере, для некоторых людей), а Ruby призван улучшить жизнь.

Как всегда, описать, что такое Tao, трудно. Я чувствую, но не могу подыскать нужных слов. Это трудно сделать даже на японском, моем родном языке. Но парень по имени Хэл Фултон попытался, и его первая попытка (первое издание этой книги) оказалась довольно удачной. А результат второго подхода к задаче описать Tao Rubi еще лучше, чему немало способствовала помощь многих людей из сообщества пользователей Ruby. По мере того как Ruby набирает популярность (отчасти благодаря продукту Ruby on Rails), все важнее становится овладение секретами мастерства производительного программирования на этом языке. Надеюсь, что эта книга поможет вам в решении этой задачи.

Удачной работы.

Юкихиро «Мац» Мацумото
Август 2006, Япония

Предисловие к первому изданию

Вскоре после того, как я впервые познакомился с компьютерами в начале 1980-х годов, меня заинтересовали языки программирования. И с тех пор я помешался на этой теме. Думаю, что причина такого интереса состоит в том, что языки программирования – это способ выражения мыслей. Они по сути своей предназначены для человека.

Но вопреки этому факту языки программирования почему-то всегда оказывались в большей степени машинно-ориентированными. Многие из них спроектированы с учетом удобства для компьютеров.

По мере того как компьютеры становятся мощнее и дешевле, ситуация постепенно меняется. Возьмем к примеру структурное программирование. Машине все равно, насколько хорошо структурирована программа, она просто исполняет ее команда за командой. Идеи структурного программирования обращены к людям, а не к машинам. То же относится и к объектно-ориентированному программированию.

Пришло время проектировать языки, удобные для людей.

В 1993 году я разговаривал со своим коллегой о сценарных языках, их выразительности и перспективах. Я считал, что программирование пойдет именно по этому пути и будет ориентироваться на человека.

Но я не был удовлетворен такими существующими языками, как Perl и Python. Я хотел видеть язык, более мощный, чем Perl, и более объектно-ориентированный, чем Python. Найти идеальный язык мне не удалось, поэтому я решил изобрести свой собственный.

Ruby – не самый простой язык, но и человеческая душа не проста. Ей равно нравятся простота и сложность. Она не приемлет ни слишком простых, ни слишком сложных вещей. Она ищет равновесия.

Поэтому при проектировании ориентированного на человека языка – Ruby – я следовал принципу наименьшего удивления. Я считал, что хорошо то, что не кажется мне странным. Поэтому я ощущаю себя естественно и даже испытываю радость, когда программирую на Ruby. А с момента выхода в свет первой версии в 1995 году многие программисты во всем мире разделили со мной эту радость.

Как всегда, я хочу выразить величайшую благодарность всем членам сообщества, сложившегося вокруг Ruby. Они – причина успеха Ruby.

Я благодарен также автору этой книги, Хэлу Фултону за то, что он показал другим Путь Ruby.

В этой книге объясняется философия, стоящая за языком Ruby. Это квинтэссенция моих мыслей и ощущений членов сообщества. Интересно, как Хэлу удалось прочитать мои мысли и раскрыть секрет Пути Ruby. Я никогда не встречался с ним лично, надеюсь, что скоро это все-таки произойдет.

Я хотел бы, чтобы эта книга и сам язык Ruby помогли вам получить удовольствие и радость от программирования.

Юкихиро «Мац» Мацумото
Сентябрь 2001, Япония

Благодарности

Благодарности за третье издание

Теперь нас уже не удивляет, что процесс переработки книги в ходе подготовки третьего издания оказался тяжким трудом. Со времени версии 1.8 Ruby претерпел существенные изменения, а быть программистом на Ruby теперь куда престижнее, чем раньше.

Проверка, дополнение и переработка книги заняли гораздо больше времени, чем предполагалось. Сменились аж три версии Ruby – 1.9, 2.0 и 2.1 – и столько же раз редактировалась и переписывалась эта книга.

Ее выходу поспособствовали многие люди. В издательстве Addison-Wesley помощь и поддержку оказывали Дебра Уильямс Коули (Debra Williams Cauley), Соньлин Киу (Songlin Qiu), Энди Бистер (Andy Beaster) и Барт Рид (Bart Reed). А вклад Русса Олсена (Russ Olsen) и Андрэ Арко *поистине бесценен*.

Техническими редакторами книги стали Русс Олсен и Стив Клэбник (Steve Klabnik), благодаря их замечаниям и предложениям текст стал точнее и понятнее. Русс предоставил также библиотеки и скрипты на Ruby для компиляции самого текста книги. Как обычно, за все ошибки несу ответственность я, а не они.

Советы, примеры коды, а также просто полезные пояснения предлагали также Дэйв Томас (Dave Thomas), Дэвид Алан Блэк (David Alan Black), Эрик Ходел (Eric Hodel), Чэд Файлер (Chad Fowler), Брэд Эдигер (Brad Ediger), Свен Фукс (Sven Fuchs), Джесси Стоример (Jesse Storimer), Люк Франкль (Luke Francl) и другие.

Отдельное спасибо Полу Харрисону (Paul Harrison) и другим моим коллегам по компании Simpli.fi за поддержку и ободрение.

Хочу также почтить память Гая Декукса (Guy Decoux) и недавно ушедшего от нас Джима Вейриха (Jim Weirich). Джим в особенности сделал очень много как для этой книги, так и для сообщества в целом.

И как всегда, в заключение приношу благодарность самому Мацу за создание Ruby, и тебе, читатель этой книги. Надеюсь, что она сможет чему-то научить, о чем-то рассказать, а то и позабавить тебя.

Благодарности за второе издание

Здравый смысл подсказывает, что второе издание требует в два раза меньше работы, чем первое. Но здравый смысл ошибается.

Хотя значительная часть текста этой книги перекочевала прямиком из первого издания, даже эту часть пришлось сильно править. Для каждого предложения задавался вопрос: «Сохранилось ли в 2006 году то, что было верно в 2001?» И это только начало.

Короче говоря, я потратил много сотен часов на подготовку второго издания, примерно столько же, сколько ушло на первое. И все же я «всего лишь автор».

Книга появляется на свет в результате усилий многих людей. Если говорить об издательстве, то я благодарен Дебре Уильямс Коули (Debra Williams Cauley), Соньлин Киу (Songlin Qiu) и Менди Фрэнк (Mandie Frank) за тяжелый труд и бесконечное терпение. Спасибо Дженил Бриз (Geneil Breeze) за неутомимое вычитывание и выпалывание сорняков из моего английского. Есть много других сотрудников, которых я не могу назвать, поскольку их работа проходила за кулисами, и я никогда с ними не встречался.

За техническое редактирование отвечали главным образом Шашанк Дейт (Shashank Date) и Фрэнсис Хван (Francis Hwang). Они прекрасно справились, я это ценю по достоинству. Если остались какие-то ошибки, то это всецело моя вина.

Большое спасибо людям, которые предлагали объяснения, писали код примеров и отвечали мне на многочисленные вопросы. Это сам Мац (Юкихиро Мацумото), Дейв Томас (Dave Thomas), Кристиан Нойкирхен (Christian Neukirchen), Чед Фаулер (Chad Fowler), Дэниэл Бергер (Daniel Berger), Армин Рерль (Armin Roehrl), Стефан Шмидль (Stefan Schmiedl), Джим Вайрих (Jim Weirich), Райан Дэвис (Ryan Davis), Дженни У. (Jenny W.), Джим Фриз (Jim Freeze), Лайл Джонсон (Lyle Johnson), Мартин ДеМелло (Martin DeMello), Март Лоуренс (Mart Lawrence), Рон Джеффрис (Ron Jeffries), Тим Хантер (Tim Hunter), Чет Хендриксон (Chet Hendrickson), Натаниэль Талбот (Nathaniel Talbott) и Бил Клеб (Bil Kleb).

Особая благодарность активным помощникам. Эндрю Джонсон (Andrew Johnson) здорово обогатил мои знания о регулярных выражениях. Пол Бэтли (Paul Battley) предоставил обширный материал для главы об интернационализации. Масао Муто (Masao Mutoh) помог в написании той же главы, а также снабдил меня материалами по GTK. Остин Зиглер (Austin Ziegler) научил меня секретам подготовки PDF-файлов. Калеб Теннис (Kaleb Tennis) дал материал по Qt. Эрик Ходел (Eric Hodel) помог в описании продуктов Rinda и Ring, а Джеймс Бритт (James Britt) внес большой вклад в главу о разработке приложений для веб.

И еще раз выражаю искреннюю благодарность и восхищение Мацу, не только за помощь, но и — прежде всего — за создание Ruby. *Domo arigato gozaimasu!*

Еще раз хочу поблагодарить своих родителей. Они постоянно подбадривали меня и с нетерпением ждали выхода книги. Я еще сделаю из них программистов.

И опять — спасибо всем членам сообщества пользователей Ruby за неутомимую энергию и дух коллективизма. Особенно я благодарен читателям этой книги (обоих изданий). Я надеюсь, что вы найдете ее информативной, полезной и, быть может, даже увлекательной.

Благодарности за первое издание

Создание книги — плод усилий целого коллектива. Я не понимал этого в полной мере, пока не взялся за это дело сам. Рекомендую всем пройти это испытание, хотя оно и не из легких. Нет сомнений, что без помощи многих и многих людей эта книга не увидела бы свет.

Прежде всего, выражаю благодарность и восхищение Мацу (Юкихиро Мацумото), который создал язык Ruby. Domo arigato gozaimasu!

Спасибо Конраду Шнейкеру (Conrad Schneiker), который подал мне идею написать эту книгу и помог выработать ее общий план. Он же оказал мне услугу, познакомив с языком Ruby в 1999 году.

Материалом для книги меня снабжали несколько человек. Первым хочу назвать Гая Хэрста (Guy Hurst), который написал значительную часть начальных глав, а также два приложения. Его помощь поистине бесценна.

Спасибо также другим помощникам, называя которых, я не придерживаюсь какого-то определенного порядка. Кэвин Смит (Kevin Smith) многое сделал для раздела главы 6, посвященного GTK, избавив меня от изучения сложной темы в условиях жесткого графика. Патрик Логан (Patrick Logan) пролил свет на графическую систему FOX GUI, описанную в той же главе. Чед Фаулер (Chad Fowler) в главе 9 углубился в тайны XML, а также помог при написании раздела о CGI.

Благодарю всех, кто правил корректуру, писал рецензии и помогал иными способами: Дона Мучоу (Don Muchow), Майка Стока (Mike Stok), Михо Огисима (Miho Ogishima) и уже упомянутых выше. Спасибо Дэвиду Эпштейну (David Eppstein), профессору математики, который ответил на вопросы по теории графов.

Одна из замечательных особенностей Ruby – поддержка со стороны сообщества пользователей. В списке рассылки и в конференции многие отвечали на мои вопросы, подавали идеи и всячески помогали. Опять же не придерживаясь определенного порядка, хочу упомянуть Дейва Томаса (Dave Thomas), Энди Ханта (Andy Hunt), Хи-Соб Парка (Hee-Sob Park), Майка Уилсона (Mike Wilson), Ави Брайанта (Avi Bryant), Ясуси Шоджи (Yasushi Shoji «Yashi»), Шуго Маэда (Shugo Maeda), Джима Вайриха (Jim Weirich) и Масаки Сукета (Masaki Suketa). Как это ни печально, но, скорее всего, кого-то я пропустил.

Очевидно, что книга никогда не вышла бы без помощи издателя. Многие работали над ней незримо для меня, но особенно я хотел бы поблагодарить Уильяма Брауна (William Brown), который тесно сотрудничал со мной и постоянно поощрял, и Скотта Мейера (Scott Meyer), который скрупулезно занимался объединением всех материалов. Других назвать не могу, потому что никогда о них не слышал. Но они знают себя сами.

Хочу поблагодарить своих родителей, которые следили за проектом издалека, подбадривали меня и даже дали себе труд ради меня освоить азы программирования.

Один мой друг как-то сказал: «Если ты написал книгу, которую никто не читает, значит, ты не написал ее вовсе». Поэтому напоследок я хочу поблагодарить читателей. Эта книга для вас. Надеюсь, что она окажется полезной.

Об авторах

Хэл Фултон начал использовать Ruby в 1999. В 2001 году он приступил к работе над книгой «The Ruby Way», второй книгой об этом языке на английском языке. Фултон был участником самой первой конференции по Ruby, состоявшейся в 2001 году, и выступал с докладами на многих других конференциях, проходивших на трех континентах, в том числе на первой европейской конференции по Ruby в 2003 году. Он обладатель двух ученых степеней по информатике, полученных в Университете штата Миссисипи, и в течение четырех лет преподавал информатику. Больше 25 лет он работает с различными версиями операционных систем UNIX и Linux. Сейчас он работает в компании Simpli.fi, которая находится в городе Форт Уорт, штат Техас, где пишет преимущественно на Ruby.

Андрэ Арко впервые познакомился с Ruby в 2004 году, когда еще учился в университете, а первое издание этой книги сыграло немалую роль в его решении строить карьеру программиста на Ruby. Он руководит проектом Bundler, менеджером зависимостей Ruby, и является автором или соавтором десятка других проектов с открытым исходным кодом. Андрэ работает консультантом в компании Cloud City Development, где отвечает за обучение сотрудников Ruby и Rails, а также занимается разработкой веб-приложений.

Андрэ с удовольствием делится добытыми тяжким трудом знаниями и опытом с другими разработчиками, он выступал на дюжине конференций по Ruby, проходивших на четырех континентах. Он регулярно работает волонтером на информационных мероприятиях RailsBridge и RailsGirls, посвященных программированию, и всячески стремится увеличить разнообразие и расширить качественный состав сообщества Ruby и способствовать признанию этой технологии как отрасли знаний. Живет он в Сан-Франциско, штат Калифорния.

Введение

Путь, который можно описать словами, –
это не истинный Путь.

– Лао Цзы, «Tao Te Ching»

Эта книга называется «Путь Ruby». Название нуждается в некотором пояснении.

Я ставил себе целью выразить в этой книге философию языка Ruby, насколько это в моих силах. Ту же цель преследовали мои добровольные помощники. Успех должно разделить между всеми, а ошибки остаются моей и только моей виной.

Конечно, я не могу абсолютно точно сказать, в чем же состоит истинный дух Ruby. Эту задачу я оставляю Мацу, но подозреваю, что даже ему трудно будет выразить ее словами.

Короче говоря, «Путь Ruby» – это всего лишь книга, а Путь Ruby – удел создателя языка и сообщества в целом. Втиснуть его в рамки книги довольно трудно.

И все же я попытаюсь в этом введении поймать неуловимый дух Ruby. Мудрый ученик не воспримет эту попытку как окончательный вердикт.

О третьем издании

Все меняется, и Ruby – не исключение. В это издание внесено немало изменений и добавлено много нового материала. В каком-то смысле все главы книги «новые». Я подверг пересмотру и переработке каждую, внес сотни мелких и десятки крупных изменений. Я исключил вещи, которые устарели или утратили значимость, изменил материал, так чтобы он лучше соответствовал самому языку Ruby, добавил примеры и комментарии.

Будучи второй книгой о Ruby на английском языке (после «Programming Ruby» Дэйва Томаса и Энди Ханта), «Путь Ruby» был построен так, чтобы служить дополнением к первой, не пересекаясь с ней; это верно и сегодня.

Между версией Ruby 1.8, рассматривавшейся во втором издании, и текущей версией 2.1 существует много различий. Но важно понимать, что изменения вносились очень бережно, на протяжении нескольких лет. Ruby по-прежнему остается Ruby. Своей красотой Ruby в немалой степени обязан политике неторопливых и продуманных изменений, мудро направляемой Мацем и другими разработчиками.

Сегодня нет недостатка в книгах по Ruby, а статей публикуется больше, чем мы в состоянии охватить взглядом. Пособия и документация представлены в веб в изобилии.

Появились новые инструменты и библиотеки. Наиболее распространенные — инструменты, создаваемые одними разработчиками для других: веб-каркасы, средства ведения блогов, средства разметки и интерфейсы к экзотическим хранилищам данных. Но, конечно, есть и много других: графические интерфейсы, средства численных расчетов, веб-службы, обработка изображений, управления версиями исходного кода и т. д.

Поддержка Ruby включена практически во все редакторы и весьма развита. Имеются полезные и зрелые интегрированные среды (IDE), которые отчасти включают средства построения графического интерфейса пользователя (ГИП).

Безусловно, значительно выросло и изменилось сообщество. Сегодня Ruby уже никто не назовет нишевым языком; он используется в государственных учреждениях, например в НАСА и в Национальном управлении океанических и атмосферных исследований (NOAA), в корпорациях, например IBM и Motorola, и на таких хорошо известных сайтах, как Wikipedia, GitHub и Twitter. Он применяется для создания графических приложений, работы с базами данных, численных расчетов и многого другого. Короче говоря, Ruby вошел в струю – я говорю в самом что ни на есть положительном смысле.

Я занимался переработкой этой книги с любовью. Надеюсь, что плод моих трудов окажется вам полезен.

Как организована эта книга

Вряд ли вы станете изучать Ruby по этой книге. В ней не так уж много вводного и учебного материала. Если вы еще ничего не знаете о Ruby, то лучше начать с какой-нибудь другой книги.

Но при этом программисты – народ упорный, и я допускаю, что научиться Ruby только по этой книге возможно. В главе 1 «Обзор Ruby» приводится краткое введение в язык и очень скромное руководство.

Также в главе 1 есть довольно полный перечень «скользких мест» (который трудно поддерживать в актуальном состоянии). Для разных читателей этот перечень полезен в разной мере, поскольку что для одного интуитивно очевидно, для другого выглядит странно.

В основном, эта книга призвана отвечать на вопросы типа «Как сделать?». И потому вы, вероятно, многое будете пропускать. Я почту за честь, если кто-то прочтет книгу от корки до корки, но не надеюсь на это. Скорее я ожидаю, что вы будете искать в оглавлении темы, которые вас интересуют в конкретный момент.

Впрочем, с момента выхода первого издания я беседовал с разными людьми, и оказалось, что многие прочли книгу целиком. Более того, несколько человек писали мне, что выучили по ней Ruby. Что ж, все возможно.

Некоторые рассматриваемые в книге вопросы могут показаться элементарными. Но ведь у разных людей и опыт разный; то, что очевидно одному, будет откровением для другого. Я старался сделать изложение как можно более полным. С другой стороны, было стремление уложиться в разумный объем (ясно, что эти цели противоречивы).

Можно назвать эту книгу «справочником наоборот». Вы ищете то, что нужно, не по имени класса или метода, а по функции или назначению. Например, в классе `String` есть несколько методов для манипулирования регистром букв: `capitalize`, `upcase`, `casecmp`, `downcase` и `swapcase`. В настоящем справочнике они встречались бы в алфавитном порядке, а в этой книге собраны в одном месте.

Конечно, в борьбе за полноту я иногда сворачивал на путь, которому следуют справочные руководства. Во многих случаях я старался компенсировать это, предлагая не совсем обычные примеры или разнообразя их по сравнению со справочниками.

Я старался не перегружать код комментариями. Если не считать первой главы, то думаю, что достиг этой цели. Писатель может стать не в меру болтливым, но программист-то хочет видеть код (а если не хочет, то должен хотеть).

Иногда примеры выглядят искусственными, за что я приношу свои извинения. Проиллюстрировать какой-то прием или принцип в отрыве от реальной задачи бывает сложно. Но чем сложнее задача, чем выше ее уровень, тем большие усилия я прилагал к подысканию реального примера. Так, если речь идет о конкатенации строк, то, наверное, вы увидите безыскусный фрагмент кода с упоминанием пресловутых "`foo`" и "`bar`", но когда рассматривается тема разбора XML-документа, будет приведен куда более содержательный и реалистичный пример.

Есть в этой книге два-три каприза, в которых хочу заранее сознаться. Во-первых, я всеми силами старался избегать «уродливых» пришедших из языка Perl глобальных переменных типа `$_` и ей подобных. Они есть в Ruby и прекрасно работают, даже применяются в повседневной работе всеми или большинством программистов. Но почти всегда от их использования можно уйти, что я и позволил себе чуть ли не во всех примерах.

Другой каприз состоит в том, что я избегаю пользоваться обособленными выражениями, если у них нет побочных эффектов. В Ruby выражения – одна из основ языка, и это прекрасно, я старался извлечь их этой особенности максимум пользы. Но во фрагментах кода я предпочитаю не употреблять выражения, которые просто возвращают никак не используемое значение. Например, для иллюстрации конкатенации строк достаточно было бы написать `"abc" + "def"`, но я в этом случае напишу что вроде `str = "abc" + "def"`. Кому-то это покажется многословием, но выглядит естественным для программиста на языке C, привыкшего к тому, что бывают функции типа void и не-void (а также программисту на Pascal'е, мыслящему в терминах процедур и функций).

Третий каприз заключается в моем нежелании употреблять символ решетки для обозначения методов экземпляра. Многие поклонники Ruby считают, что я проявляю излишнюю болтливость, когда пишу «метод экземпляра `crypt` класса `String`», а не просто `String#crypt`, но я полагаю, что так никто не запутается. (На самом деле, я постепенно смиряюсь с использованием такой нотации, так как ясно, что она уже не исчезнет.)

Я старался давать ссылки на внешние ресурсы там, где это уместно. Ограничения по времени и объему не позволили мне включить в книгу все, что я хотел бы, но надеюсь, что это хотя бы отчасти компенсируется указаниями на то, где найти

недостающую информацию. Из всех источников самым главным, наверное, следует считать архив приложений Ruby (Ruby Application Archive) в сети; вы не раз встретите ссылки на него.

В начале книги принято приводить соглашения об использовании шрифтов, применяемых для выделения кода, и о том, как отличить пример от обычного текста. Но я не стану оскорблять вас недоверием к вашим умственным способностям, все вы и раньше читали техническую литературу.

Хочу подчеркнуть, что примерно 10% текста книги было написано другими людьми. И это не считая технического редактирования и корректуры. Вы просто обязаны прочитать благодарности, приведенные в этой (и любой другой) книге. Большинство читателей пропускают их. Прошу, прочтите прямо сейчас. Это будет так же полезно, как питание овощами.

Об исходных текстах, приведенных в книге

Все сколько-нибудь значительные фрагменты кода собраны в архив, который можно загрузить из сети. Этот архив есть на сайте informit.com и на сайте самой книги therubyway.io.

Он предлагается в виде tgz-файла и в виде zip-файла. Совсем короткие фрагменты, которые нельзя исполнить «вне контекста», в архив обычно не включаются.

Что такое «Путь Ruby»?

Попробуем сразиться со злом, не выразимым словами,
и кто знает, возможно, мы его все же одолеем.

— Дуглас Адамс
«Детективное агентство Дирка Джентли»

Что мы имеем в виду, говоря о Пути Ruby? Я полагаю, что тут есть два взаимосвязанных аспекта: философия проектирования Ruby и философия использования этого языка. Естественно, что дизайн и применение связаны друг с другом, будь то программное или аппаратное обеспечение. А иначе зачем бы существовала наука эргономика? Если я снабжаю устройство ручкой, то, наверное, предполагаю, что кто-то за эту ручку возьмется.

В языке Ruby имеется невыразимое словами качество, которое делает его тем, что он есть. Мы наблюдаем это качество в дизайне синтаксиса и семантики языка, присутствует оно и в написанных на нем программах. Но стоит попытаться сформулировать, в чем состоит эта отличительная особенность, как она размывается.

Очевидно, Ruby – не просто инструмент для написания программ, но и сам по себе является программой. Почему работа программ, написанных на Ruby, должна следовать законам, отличным от тех, которым подчинена работа интерпретатора? В конце концов, Ruby – исключительно динамичный и расширяемый язык. Могут найтись причины, по которым эти два уровня где-то расходятся, быть может, ста-

раясь приспособиться к несовершенству реального мира. Но в общем случае мыслительные процессы могут и должны быть сходными. Интерпретатор Ruby можно было бы написать на самом Ruby, в полном соответствии с принципом Хофштадтера, хотя в настоящее время это еще не сделано.

Мы не часто задумываемся над этимологией слова «путь», но оно употребляется в двух разных смыслах. Во-первых, это метод или техника, а во-вторых – дорога. Ясно, что оба значения взаимосвязаны, и, говоря «путь Ruby», я имею в виду как то, так и другое.

Следовательно, мы говорим о мыслительном процессе, но одновременно и о дороге, по которой движемся. Даже величайшие гуру программирования не могут сказать о себе, что достигли совершенства, они лишь идут по дороге к нему. Таких дорог может быть несколько, но я здесь говорю только об одной.

Привычная мудрость гласит, что форма определяется функцией. Это, конечно, правильно. Но Фрэнк Ллойд Райт[1] (имея в виду свою собственную область интересов) как-то сказал: «Форма определяется функцией, которая было понята неправильно. Форма и функция должны быть едины, сливаться в духовном единении».

Что Райт имел в виду? Я бы сказал, что на этот вопрос вы найдете ответ не в книгах, а в собственном опыте.

Однако я думаю, что Райт выразил эту мысль где-то еще, разбив ее на части, которые проще переварить. Он был великим поборником простоты, который однажды заметил: «Самые полезные инструменты архитектора – это ластик рядом с чертежной доской и гвоздодер на строительной площадке».

Итак, одним из достоинств Ruby является простота. Надо ли цитировать других мыслителей, высказывавшихся на эту тему? Согласно Антуану де Сент-Экзюпери: «Совершенство достигнуто не тогда, когда нечего добавить, а тогда, когда нечего убрать».

Но Ruby – сложный язык. Почему же я называю его простым?

Если бы мы лучше понимали мироздание, то, наверное, открыли бы «закон сохранения сложности» – факт, который вмешивается в нашу жизнь подобно энтропии, которую мы не можем избежать, а способны лишь рассеивать.

И в этом ключ. Нельзя избежать сложности, но можно укрыться от нее. Мы можем убрать ее из виду. Это тот же старый добрый принцип черного ящика, внутри которого решается сложная задача, хотя на поверхности лишь голая простота.

Если вам еще не наскучили цитаты, то будет уместно привести слова Альберта Эйнштейна: «Все должно быть просто настолько, насколько возможно, но не проще».

Таким образом, на взгляд программиста, Ruby – это воплощенная простота (хотя у человека, отвечающего за сопровождение интерпретатора, взгляд может быть иной). Но вместе с тем имеется пространство для компромиссов. В реальном мире всем нам приходится немного сгибаться. К примеру, все сущности в програм-

---

[1] Фрэнк Ллойд Райт (1867–1959) – знаменитый архитектор и дизайнер. Одна из самых известных работ – музей Музей Гуггенхайма в Нью-Йорке. – *Прим. перев.*

ме на Ruby должны были бы быть истинными объектам, однако некоторые, в том числе целые числа, хранятся как непосредственные значения. Это компромисс, знакомый всем студентам отделений информатики уже много десятилетий: элегантность дизайна приносится в жертву практичности реализации. По существу, мы обменяли одну простоту на другую.

То, что Ларри Уолл говорил о языке Perl, остается справедливым: «Когда вы хотите что-то выразить на маленьком языке, оно становится большим. А когда вы пытаетесь выразить то же самое на большом языке, оно становится маленьким.» Это верно и в отношении английского языка. Если биолог Эрнст Хэккель смог всего тремя словами выразить глубокую мысль «онтогенез повторяет филогенез», то лишь потому, что эти слова с весьма специфическим смыслом были в его распоряжении. Мы соглашаемся на внутреннюю сложность языка, потому что она позволяет избежать сложности в отдельных высказываниях.

Переформулирую этот принцип по-другому: «не пишите 200 строк кода, когда достаточно 10».

Я считаю само собой разумеющимся, что краткость в общем случае хороша. Короткий фрагмент кода занимает меньше места в мозгу программиста, его проще воспринять как единое целое. Благоприятным побочным эффектом следует считать и то, что в короткой программе будет меньше ошибок.

Конечно, не нужно забывать предупреждение Эйнштейна о простоте. Если расположить краткость слишком высоко в списке приоритетов, то получится совершенно загадочный код. Согласно теории информации, сжатые данные статистически похожи на белый шум. Если вы видели код на C или APL или регулярное выражение – особенно плохо написанные, то понимаете, что я имею в виду. «Просто, но не слишком просто» – это ключевая фраза. Стремитесь к краткости, но не жертвуйте понятностью.

Трюизмом следует считать мысль о том, что краткость в сочетании с понятностью – это хорошо. Но тому есть причина, причем настолько фундаментальная, что мы часто о ней забываем. А состоит она в том, что компьютер создан для человека, а не человек для компьютера.

В старые добрые дни все было почти наоборот. Компьютеры стоили миллионы долларов и пожирали многие киловатты электричества. Люди же вели себя так, будто компьютер – божество, а программисты – его скромные жрецы. Час машинного времени стоил дороже часа личного времени.

Когда компьютеры стали меньше и дешевле, приобрели популярность языки высокого уровня. Они неэффективны с точки зрения машины, зато эффективны с позиции человека. Ruby – это просто одно из последних достижений на этом пути. Некоторые даже называют его языком сверхвысокого уровня (VHLL – Very High-Level Language). Хотя этот термин еще не получил четкого определения, я думаю, что он оправдан.

Компьютер призван быть слугой, а не хозяином, а, как сказал Мац, толковый слуга должен выполнять сложное задание при минимуме указаний. Так было на протяжении всей истории информатики. Мы начали с машинного языка, перешли к языку ассемблера, а потом добрались и до языков высокого уровня.

Мы сейчас говорим о смещении парадигмы: от машинно-центрической к человеко-центрической. На мой взгляд, Ruby дает великолепный пример человеко-центрического программирования.

Я хочу взглянуть на вопрос под несколько иным углом. В 1980 вышла чудесная книжка Джеффри Джеймса «Tao программирования» (*The Tao of Programming*, Geoffrey James). Каждая строчка из нее достойна цитирования, но я ограничусь лишь одной выдержкой: «Программа должна следовать 'закону наименьшего удивления'. Что это за закон? Все просто: программа должна отвечать пользователю так, чтобы вызывать у него как можно меньше удивления.» (Конечно, если речь идет об интерпретаторе языка, то пользователем является программист.)

Не знаю, Джеймс ли придумал этот термин, но я впервые узнал его из этой книги. Этот закон хорошо известен и часто цитируется в сообществе пользователей Ruby. Правда, обычно его называют «Принципом Наименьшего Удивления» или POLS (Principle of Least Surprise). (Лично я упрямо придерживаюсь акронима LOLA – Law of Least Astonishment.)

Но, как ни называй, правило остается справедливым и служит основополагающим принципом продолжающейся работы над языком Ruby. О нем полезно помнить и тем, кто разрабатывает библиотеки и пользовательские интерфейсы.

Конечно, одна проблема остается: разные люди удивляются разным вещам; не существует всеобщего согласия о том, как «должен» вести себя объект или метод. Но мы можем стремиться быть последовательными и находить веские обоснования принимаемым проектным решениям, а каждый человек должен тренировать собственную интуицию.

Кстати, Мац как-то заметил, что «принцип наименьшего удивления» должен относиться и к нему как к дизайнеру. Чем больше ваше мышление походит на его, тем меньше удивления будет вызывать Ruby. И смею уверить, подражание Мацу большинству из нас только пойдет на пользу.

Какой бы ни была логическая конструкция системы, тренировать свою интуицию необходимо. Каждый язык программирования – это отдельный мир, со своими допущениями, в этом они ничем не отличаются от естественных языков. Когда я начал изучать немецкий, то обнаружил, что все существительные пишутся с заглавной буквы. За исключением слова *deutsch*. Я пожаловался профессору, подчеркивая, что ведь это же *название* самого языка. Он улыбнулся и ответил: «Не надо с этим бороться».

Он говорил, что надо дать немцу оставаться немцем. Продолжая эту мысль, хочу дать совет всем, кто приходит в Ruby из других языков. Пусть Ruby остается Ruby. Не ожидайте, что это будет Perl, это не так. Не ждите от него поведения, характерного для языков LISP или Smalltalk. С другой стороны, у Ruby есть элементы, присущие любому из этих трех языков. Для начала действуйте в соответствии с априорными представлениями, но когда они оказываются неверны, не боритесь с языком. (Если только Мац не согласится с тем, что необходимо изменение.)

Каждый программист сегодня знает о принципе ортогональности (хотя лучше было бы назвать его принципом *ортогональной полноты*). Вообразим пару осей, по одной из которых откладываются языковые сущности, а по другой – множество

атрибутов и возможностей. Когда мы говорим об «ортогональности», то обычно имеем в виду, что пространство, определяемое этими осями настолько «полно», насколько это логически возможно.

Одна из составных частей Пути Ruby – стремление к ортогональности. Массив в некоторых отношениях подобен хэшу, а потому и операции над ними должны быть похожи. До тех пор, пока мы не вступаем в область, где эти сущности начинают отличаться друг от друга.

Мац говорит что «естественность» важнее ортогональности. Но чтобы понять, что естественно, а что нет, надо долго думать и писать программы.

Ruby стремится быть дружелюбным к программисту. Например, у многих методов есть синонимы; оба метода `size` и `length` возвращают число элементов в массиве. Два разных написания слова – `indexes` и `indices` – относятся к имени одного и того же метода. Некоторые называют это досадным недоразумением, но я склонен считать такую избыточность хорошим дизайном.

Ruby стремится к последовательности и единообразию. В этом нет ничего мистического, во всех жизненных ситуациях мы жаждем регулярности и размеренности. Сложнее научиться понимать, когда этот принцип следует нарушить.

Например, в Ruby принято добавлять вопросительный знак (?) в конец имени метода, ведущего себя как предикат. Это хорошо и удобно, программа становится яснее, а в пространстве имен легче ориентироваться. Но менее последовательным является аналогичное употребление восклицательного знака для обозначения потенциально «деструктивных» или «опасных» методов (в том смысле, что они модифицируют внутреннее состояние вызывающего объекта). Непоследовательность состоит в том, что не все деструктивные методы помечаются таким образом. Нужно ли восстановить справедливость?

Нет, на самом деле не нужно. Некоторые методы по сути своей изменяют состояние (например, методы `replace` и `concat` класса `Array`). Одни являются «методами установки», которые допускают присваивание атрибуту класса; ясно, что не следует добавлять восклицательный знак к имени атрибута или к знаку равенства. Другие в каком-то смысле изменяют состояние объекта, например, `read`, но это происходит так часто, что нет смысла особо отмечать этот факт. Если бы имя каждого деструктивного метода заканчивалось символом `!`, то программа превратилась бы в рекламную брошюру фирмы, занимающейся многоуровневым маркетингом.

Вы замечаете действие разнонаправленных сил, тенденцию нарушать все правила? Тогда позвольте мне сформулировать второй закон Фултона: «У каждого правила есть исключения, кроме второго закона Фултона». (Доля шутки тут есть, но небольшая.)

В Ruby мы видим не «педантичную непротиворечивость», а строгое следование набору простых правил. Быть может, отчасти Путь Ruby состоит в том, что его подход не является закостенелым и неподвижным. Мац как-то сказал, что при проектировании языка нужно «следовать велениям своего сердца».

И еще один аспект философии Ruby: «Не бойтесь изменений во время выполнения, не бойтесь быть динамичными.» Мир динамичен, так почему язык про-

граммирования должен быть статичным? Ruby – один из самых динамичных среди существующих языков.

Я бы с некоторой долей неуверенности выделил еще один аспект: «Не будьте рабом производительности». Если производительность оказывается недопустимо низкой, проблему придется решать, но не следует с самого начала выводить ее на первый план. Предпочитайте элегантность эффективности в тех случаях, когда эффективность не слишком критична. Впрочем, когда вы пишете библиотеку, которая будет использоваться непредвиденными способами, о производительности следует задуматься с самого начала.

Когда я смотрю на язык Ruby, то вижу равновесие между разными проектными целями, вижу сложное взаимодействие, напоминающее о задаче n тел в физике. Я могу представить себе, что он моделировался как мобил Александра Кальдера. Быть может, больше всего завораживает само взаимодействие, гармония, лежащая в основе философии Ruby, а не отдельные составные части. Программисты знают, что их ремесло – не просто сплав науки и технологии, но еще и искусство. Мне неловко говорить, что в компьютерных дисциплинах есть какой-то духовный аспект, но строго между нами – он, безусловно, присутствует. (Если вы не читали книгу Роберта Пирсига «Искусство ухода за мотоциклом» (Robert Pirsig *Zen and the Art of Motorcycle Maintenance*), советую прочитать.)

Источником Ruby стала человеческая потребность создавать полезные и красивые вещи. Программы, написанные на Ruby, должны проистекать из того же боговдохновенного источника. Это, на мой взгляд, и является квинтэссенцией Пути Ruby.

Глава 1. Обзор Ruby

Язык формирует способ нашего мышления
и диктует, о чем мы можем размышлять.

Бенджамин Ди Уорф

Стоит напомнить, что в новом языке программирования иногда видят панацею, особенно его адепты. Но ни один язык не сможет заменить все остальные. Не существует инструмента, безусловно пригодного для решения любой мыслимой задачи. Есть много предметных областей и много ограничений, налагаемых решаемыми в них задачами.

А самое главное — есть разные пути обдумывания задач, и это следствие разного опыта и личных качеств самих программистов. Поэтому в обозримой перспективе будут появляться все новые и новые языки. А пока есть много языков, будет много людей, которые их критикуют и защищают. Короче говоря, «языковым войнам» конца не предвидится, но мы в этой книге не станем принимать в них участия.

И тем не менее, в постоянном поиске новой, более удачной системы записи программ, нас иногда озаряют идеи, переживающие контекст, в котором зародились. Как Pascal многое позаимствовал у Algol'a, как Java вырос из C, так и каждый язык что-то берет у своих предшественников.

Язык – это одновременно набор инструментов и площадка для игр. У него есть практическая сторона, но он же служит и полигоном для испытания новых идей, которые могут быть приняты или отвергнуты сообществом программистов.

Одна из наиболее далеко идущих идей – концепция объектно-ориентированного программирования (ООП). Многие скажут, что значимость ООП имеет скорее эволюционный, нежели революционный характер, но никто не возразит против того, что оно оказало огромное влияние на индустрию. Двадцать пять лет назад объектная ориентированность представляла в основном академический интерес, сегодня это универсально принятая парадигма.

Вездесущность ООП породила много «рекламной чепухи» в индустрии. В классической работе, написанной в конце 1980-х годов, Роджер Кинг отметил: «Если вы хотите продать кошку специалисту по компьютерам, скажите, что она объектно-ориентированная». Существуют различия во мнениях о том, что на самом деле представляет собой ООП, и даже среди тех, кто разделяет общую точку зрения, имеются разногласия относительно терминологии.

Мы не ставим себе целью поучаствовать в спорах. Мы согласны, что ООП – полезный инструмент и ценная методология решения задач; мы не утверждаем, что она способна излечить рак.

Что касается истинной природы ООП, то у нас есть любимые определения и термины, но мы пользуемся ими лишь для эффективного общения, пререкаться по поводу смысла слов мы не станем.

Все это мы упомянули лишь потому, что знакомство с основами ООП необходимо для чтения этой книги и понимания примеров и подходов. Что бы ни говорить о Ruby, он безусловно является объектно-ориентированным языком.

1.1. Введение в объектно-ориентированное программирование

Прежде чем начать разговор о самом языке Ruby, неплохо было бы потолковать об объектно-ориентированном программировании вообще. На этих первых страницах мы кратко рассмотрим общие идеи, лишь слегка касаясь Ruby.

1.1.1. Что такое объект

В объектно-ориентированном программировании *объект* – фундаментальное понятие. Объект – это сущность, служащая контейнером для данных и управляющая доступом к этим данным. С объектом связан набор *атрибутов*, которые в сущности представляют собой просто переменные, принадлежащие объекту. (В этой книге мы будем без стеснения употреблять привычный термин «переменная» в применении к атрибутам.) Кроме того, с объектом ассоциирован набор функций, предоставляющих интерфейс к функциональным возможностям объекта. Эти функции называются *методами*.

Важно отметить, что любой объектно-ориентированный язык предоставляет механизм *инкапсуляции*. В общепринятом смысле это означает, во-первых, что атрибуты и методы объекта ассоциированы именно с этим объектом, а, во-вторых, что область видимости атрибутов и методов по умолчанию ограничена самим объектом (применение принципа *сокрытия информации*).

Объект считается экземпляром *класса объекта* (обычно он называется просто *классом*). Класс можно представлять себе как чертеж или образец, а объект – как вещь, изготовленную по этому чертежу. Также класс часто называют *абстрактным типом*, то есть типом, более сложным, нежели целое или строка символов.

Создание объекта (экземпляра класса) называют *инстанцированием*. В некоторых языках имеются явные *конструкторы* и *деструкторы* – функции, выполняющие действия, необходимые соответственно для инициализации и уничтожения объекта. Отметим попутно, что в Ruby есть нечто, что можно назвать конструктором, но никакого аналога деструктора не существует (благодаря наличию механизма сборки мусора).

Иногда возникает ситуация, когда некоторые данные имеют широкую область видимости, не ограниченную одним объектом, и помещать копию такого атрибу-

та в каждый экземпляр класса неправильно. Рассмотрим, к примеру, класс MyDogs и три объекта этого класса: fido, rover и spot. У каждой собаки могут быть такие атрибуты, как возраст и дата вакцинации. Предположим, однако, что нужно сохранить еще и имя владельца всех собак. Можно, конечно, поместить его в каждый объект, но это пустая трата памяти, к тому же искажающая смысл дизайна. Ясно, что атрибут «имя владельца» принадлежит не отдельному объекту, а классу в целом. Такие атрибуты (синтаксис их определения в разных языках различен) называются *атрибутами класса* (или *переменными класса*).

Есть немало ситуаций, в которых может понадобиться переменная класса. Допустим, например, что нужно знать, сколько всего было создано объектов некоторого класса. Можно было бы завести переменную класса, инициализировать ее нулем и увеличивать на единицу при создании каждого объекта. Эта переменная ассоциирована именно с классом, а не с каким-то конкретным объектом. С точки зрения области видимости, она не отличается от любого другого атрибута, но существует лишь одна ее копия для всего множества объектов данного класса.

Чтобы отличить атрибуты класса от обыкновенных атрибутов, последние часто называют *атрибутами объекта* (или *атрибутами экземпляра*). Условимся, что в этой книге под словом «атрибут» понимается атрибут экземпляра, если явно не оговорено, что это атрибут класса.

Точно также методы объекта служат для управления доступом к его атрибутам и предоставляют четко определенный интерфейс для этой цели. Но иногда желательно или даже необходимо определить метод, ассоциированный с самим классом. Не удивительно, что *метод класса* управляет доступом к переменным класса, а также выполняет действия, распространяющиеся на весь класс, а не на какой-то конкретный объект. Как и в случае атрибутов, мы будем считать, что метод принадлежит объекту, если явно не оговорено противное.

Стоит отметить, что в некотором смысле все методы являются методами класса. Не нужно думать, что, создав 100 объектов, мы породили 100 копий кода методов! Однако правила ограничения области видимости гласят, что метод каждого объекта оперирует данными только того объекта, от имени которого вызван. Тем самым у нас создается иллюзия, будто методы объекта ассоциированы с самими объектами.

1.1.2. Наследование

Мы подходим к одной из самых сильных сторон ООП – *наследованию*. Наследование - это механизм, позволяющий расширять ранее определенную сущность путем добавления новых возможностей. Короче говоря, наследование – это способ повторного использования кода. (Простой и эффективный механизм повторного использования долго был Святым Граалем в информатике. Много десятилетий назад его поиски привели к изобретению параметризованных процедур и библиотек. ООП -- это лишь одна из последних попыток реализации искомого.)

Обычно наследование рассматривается на уровне класса. Если нам необходим какой-то класс, а в наличии имеется более общий, то можно определить свой класс

так, чтобы он наследовал поведение уже существующего класса. Предположим, например, что есть класс Polygon, описывающий выпуклые многоугольники. Тогда класс прямоугольника Rectangle можно унаследовать от Polygon. При этом Rectangle будет иметь все атрибуты и методы класса Polygon. Так, может уже быть написан метод, вычисляющий периметр путем суммирования длин всех сторон. Если все было реализовано правильно, то этот метод автоматически будет работать и для нового класса, переписывать код не придется.

Если класс B наследует классу A, мы говорим, что B является *подклассом* A, а A – суперкласс B. По-другому говорят, что A – *базовый* или *родительский* класс, а B – *производный* или *дочерний* класс.

Как мы видели, производный класс может трактовать методы своего базового класса как свои собственные. С другой стороны, он может *переопределить* метод базового класса, предоставив иную его реализацию. Кроме того, в большинстве языков есть возможность вызвать из переопределенного метода метод базового класса с тем же именем. Иными словами, метод foo класса B знает, как вызвать метод foo класса A. (Любой язык, не предоставляющий такого механизма, можно заподозрить в отсутствии истинной объектной ориентированности.) То же верно и в отношении атрибутов.

Отношение между классом и его суперклассом интересно и важно, обычно его называют отношением «*является*». Действительно, квадрат Square «является» прямоугольником Rectangle, а прямоугольник Rectangle «является» многоугольником Polygon и т.д. Поэтому, рассматривая иерархию наследования (а такие иерархии в том или ином виде присутствуют в любом объектно-ориентированном языке), мы видим, что в любой ее точке специализированные сущности «являются» подклассами более общих. Отметим, что это отношение транзитивно, – если обратиться к предыдущему примеру, то квадрат «является» многоугольником. Однако отношение «является» не коммутативно – каждый прямоугольник есть многоугольник, но не каждый многоугольник – прямоугольник.

Это подводит нас к теме *множественного наследования*. Можно представить себе класс, который наследует нескольким классам. Например, классы Dog и Cat могут наследовать классу Mammal (млекопитающее), а Sparrow (воробей) и Raven (ворон) – классу WingedCreature (крылатое). Но как быть с классом Bat (летучая мышь)? Он с равным успехом может наследовать и Mammal, и WingedCreature. Это хорошо согласуется с нашим жизненным опытом, ведь многие вещи можно отнести не к одной категории, а сразу к нескольким, не вложенным друг в друга.

Множественное наследование, вероятно, наиболее противоречивая часть ООП. Некоторые указывают на потенциальные неоднозначности, которые необходимо разрешать. Например, если в обоих классах Mammal и WingedCreature есть атрибут size (или метод eat), то какой из них имеется в виду, когда мы обращаемся к нему из объекта класса Bat? С этой трудностью тесно связана *проблема ромбовидного наследования*; она называются так из-за формы диаграммы наследования, возникающей, когда оба суперкласса наследуют одному классу. Представьте себе, что классы Mammal и WingedCreature наследуют общему предку Organism, тогда иерар-

хия наследования от `Organism` к `Bat` будет иметь форму ромба. Но как быть с атрибутами, которые оба промежуточных класса наследуют от своего родителя? Получает ли `Bat` две копии? Или они должны быть объединены в один атрибут, поскольку все равно заимствованы у общего предка?

Это скорее проблемы проектировщика языка, а не программиста. В разных объектно-ориентированных языках они решаются по-разному. Иногда вводятся правила, согласно которым какое-то одно определение атрибута «выигрывает». Или предоставляется возможность различать одноименные атрибуты. Иногда даже язык позволяет вводить псевдонимы или переименовывать идентификаторы. Многими это рассматривается как аргумент против множественного наследования – о механизмах разрешения подобных конфликтов имен нет единого мнения, поэтому все они языково-зависимы. В языке C++ предлагается минимальный набор средств для разрешения неоднозначностей, механизмы языка Eiffel, наверное, получше, а в Perl проблема решается совсем по-другому.

Есть и альтернатива – полностью запретить множественное наследование. Такой подход принят в языках Java и Ruby. На первый взгляд, это даже не назовешь компромиссным решением, но, вскоре мы убедимся, что все не так плохо, как кажется. Мы познакомимся с приемлемой альтернативой традиционному множественному наследованию, но сначала обсудим *полиморфизм* – еще одно понятие из арсенала ООП.

1.1.3. Полиморфизм

Термин «полиморфизм», наверное, вызывает самые жаркие семантические споры. Каждый знает, что это такое, но все понимают его по-разному. (Не так давно вопрос «Что такое полиморфизм?» стал популярным во время собеседования при поступлении на работу. Если его зададут вам, рекомендую процитировать какого-нибудь признанного авторитета, например, Бертрана Мейера или Бьярна Страуструпа; если собеседник не согласится, то пусть он спорит с классиком, а не с вами.)

Буквально слово «полиморфизм» означает «способность принимать разные формы или обличья». В самом широком смысле так называют ситуацию, когда различные объекты по-разному отвечают на одно и то же сообщение (или вызов метода).

Дамиан Конвей (Damian Conway) в книге «Object-Oriented Perl» проводит смысловое различие между двумя видами полиморфизм. Первый, *наследственный полиморфизм*, – то что, имеют в виду большинство программистов, говорящих о полиморфизме.

Если некоторый класс наследует своему суперклассу, то по определению все методы суперкласса присутствуют также и в подклассе. Таким образом, цепочка наследования представляет собой линейную иерархию классов, отвечающих на одни и те же методы. Нужно, конечно, помнить, что в любом подклассе метод может быть переопределен, именно это и составляет сильную сторону наследования. При вызове метода объекта обычно отвечает либо метод, унаследованный от

суперкласса, либо более специализированный вариант этого метода, созданный в интересах именно данного подкласса.

В языках со статической типизацией, например в C++, наследственный полиморфизм гарантирует совместимость типов вниз по цепочке наследования (но не в обратном направлении). Например, если `B` наследует `A`, то указатель на объект класса `A` может указывать также и на объект класса `B`, обратное же неверно. Совместимость типов – существенная черта ООП в подобных языках, можно даже сказать, что полиморфизм ей и исчерпывается. Но конечно же, полиморфизм можно реализовать и в отсутствии статической типизации (как в Ruby).

Второй вид полиморфизма, упомянутый Конвеем, – это *интерфейсный полиморфизм*. Для него не требуется наличия отношения наследования между классами, нужно лишь, чтобы в интерфейсах объектов были методы с одним и тем же именем. Такие объекты можно трактовать как принадлежащие одному виду, и потому мы имеем некую разновидность полиморфизма (хотя в большинстве работ он так не называется).

Читатели, знакомые с языком Java, понимают, что в нем реализованы оба вида полиморфизма. Класс в Java может расширять другой класс, наследуя ему с помощью ключевого слова `extends`, а может с помощью ключевого слова `implements` реализовывать интерфейс, за счет чего приобретает заранее известный набор методов (которые необходимо переопределить). Такой синтаксис позволяет интерпретатору Java во время компиляции определить, можно ли вызывать данный метод для конкретного объекта.

Ruby поддерживает интерфейсный полиморфизм, но по-другому. Он позволяет определять *модули*, методы которых можно *«подмешивать»* к существующим классам. Но обычно модули так не используются. Модуль состоит из методов и констант, которые можно использовать так, будто они являются частью класса или объекта. Когда модуль подмешивается с помощью предложения `include`, мы получаем ограниченную форму множественного наследования. (По словам проектировщика языка Юкихиро Мацумото, это можно рассматривать как одиночное наследование с разделением реализации.) Таким образом удается сохранить преимущества множественного наследования, не страдая от его недостатков.

1.1.4. Еще немного терминов

В языках, подобных C++, существует понятие *абстрактного класса*. Такому классу разрешается наследовать, но создать его экземпляр невозможно. В более динамичном языке Ruby такого понятия нет, но если программист пожелает, то может смоделировать его, потребовав, чтобы все методы были переопределены в производных классах. Полезно это или нет, оставляем на усмотрение читателя.

Создатель языка C++ Бьярн Страуструп определяет также понятие *конкретного типа*. Это класс, существующий только для удобства. Он спроектирован не для наследования; более того, ожидается, что ему никто никогда наследовать не будет. Другими словами, преимущества ООП в этом случае сводятся только к инкапсу-

ляции. Ruby не поддерживает такой конструкции синтаксически (как и C++), но по природе своей прекрасно приспособлен для создания подобных классов.

Считается, что некоторые языки поддерживают более «чистую» модель ООП, чем другие. (Мы также применяем термин *«радикально объектно-ориентированный»*.) Это означает, что любая сущность в языке является объектом, даже примитивные типы представлены полноценными классами, а переменные и константы рассматриваются как экземпляры. В таких языках, как Java, C++ и Eiffel, дело обстоит иначе. В них примитивные типы (особенно константы) не являются настоящими объектами, хотя иногда могут рассматриваться как таковые с помощью «классов-оберток». Вероятно, есть языки, которые *более* радикально объектно ориентированы, чем Ruby, но их немного.

Большинство объектно-ориентированных языков статичны; методы и атрибуты, принадлежащие классу, глобальные переменные и иерархия наследования определяются во время компиляции. Быть может, самый сложный концептуальный переход заключается в том, что в Ruby все это происходит *динамически*. И определения, и даже порядок наследования можно задавать во время исполнения. Честно говоря, каждое объявление или определение *исполняется* во время работы программы. Помимо прочих достоинств, это позволяет избавиться от условной компиляции, и во многих случаях получается более эффективный код.

На этом мы завершаем беглую экскурсию в мир ООП. Мы старались последовательно применять введенные здесь термины на протяжении всей книги. Перейдем теперь к краткому обзору самого языка Ruby.

1.2. Базовый синтаксис и семантика Ruby

Выше мы отметили, что Ruby — *настоящий динамический* объектно-ориентированный язык.

Прежде чем переходить к обзору синтаксиса и семантики, упомянем некоторые другие его особенности.

Ruby — *гибкий* (agile) язык. Он пластичен и поощряет частую переработку (рефакторинг), которая выполняется без особого труда.

Ruby — *интерпретируемый* язык. Разумеется, в будущем ради повышения производительности могут появиться и компиляторы Ruby, но мы считаем, что у интерпретатора много достоинств. Он не только позволяет быстро создавать прототипы, но и сокращает весь цикл разработки.

Ruby *ориентирован на выражения*. Зачем писать предложение, когда выражения достаточно? Это означает, в частности, что программа становится более компактной, поскольку общие части выносятся в отдельное выражение и повторения удается избежать.

Ruby — *язык сверхвысокого уровня* (VHLL). Один из принципов, положенных в основу его проектирования, заключается в том, что компьютер должен работать для человека, а не наоборот. Под «плотностью» Ruby понимают тот факт, что сложные, запутанные операции можно записать гораздо проще, чем в языках более низкого уровня.

Начнем мы с рассмотрения общего духа языка и некоторых применяемых в нем терминов. Затем мы вкратце остановимся на природе программы на Ruby, а потом уже перейдем к примерам.

Прежде всего, отметим, что программа на Ruby состоит из отдельных строк, как в C, но не как в древних языках наподобие FORTRAN. В одной строке может быть сколько угодно лексем, лишь бы они правильно отделялись пропусками. В одной строке может быть несколько предложений, разделенных точками с запятой; только в этом случае точка с запятой и необходима. Логическая строка может быть разбита на несколько физических при условии, что все, кроме последней, заканчиваются обратной косой чертой или лексическому анализатору дан знак, что предложение еще не закончено. Таким знаком может, например, быть запятая в конце строки.

Главной программы как таковой (функции main) не существует; исполнение происходит сверху вниз. В более сложных программах в начале текста могут располагаться многочисленные определения, за которыми следует (концептуально) главная программа. Но даже в этом случае программа исполняется сверху вниз, так как в Ruby все определения исполняются.

1.2.1. Ключевые слова и идентификаторы

Ключевые (или зарезервированные) слова в Ruby обычно не применяются ни для каких иных целей. Вот их полный перечень:

```
BEGIN     END       alias     and       begin
break     case      class     def       defined?
do        else      elsif     end       ensure
false     for       if        in        module
next      nil       not       or        redo
rescue    retry     return    self      super
then      true      undef     unless    until
when      while     yield
```

Имена переменных и других идентификаторов обычно начинаются с буквы или специального модификатора. Основные правила таковы:

- имена локальных переменных (и таких псевдопеременных, как `self` и `nil`) начинаются со строчной буквы или подчерка;
- имена глобальных переменных начинаются со знака доллара `$`;
- имена переменных экземпляра (принадлежащих объекту) начинаются со знака `@`;
- имена переменных класса (принадлежащих классу) начинаются с двух знаков `@`;
- имена констант начинаются с заглавной буквы;
- в именах идентификаторов подчерк (_) можно использовать наравне со строчными буквами;
- специальные переменные, имена которых начинаются со знака доллара (например, `$1` и `$/`), устанавливаются самим интерпретатором Ruby.

Примеры:

- **локальные переменные** alpha, _ident, some_var;
- **псевдопеременные** self, nil, __FILE__;
- **константы** K6chip, Length, LENGTH;
- **переменные экземпляра** @foobar, @thx1138, @NOT_CONST;
- **переменные класса** @@phydeaux, @@my_var, @@NOT_CONST;
- **глобальные переменные** $beta, $B12vitamin, $NOT_CONST.

1.2.2. Комментарии и встроенная документация

Комментарии в Ruby начинаются со знака решетки (#), находящегося вне строки или символьной константы, и продолжаются до конца строки:

```
x = y + 5   # Это комментарий.
# Это тоже комментарий.
print "# А это не комментарий."
```

Комментарии, расположенные непосредственно перед определениями, обычно документируют то, что определяется ниже. Часто такую встроенную документацию можно извлечь из текста программы каким-нибудь внешним инструментом. Типичный документирующий комментарий может занимать несколько соседних строк:

```
# Назначение этого класса -
# излечить рак
# и установить мир во всем мире.
class ImpressiveClass
```

Весь текст, расположенный между строками, которые начинаются лексемами =begin и =end (включительно), игнорируется интерпретатором (этим лексемам не должны предшествовать пробелы).

```
=begin
Все, что находится между
этими строками, также
является комментарием.
=end
```

1.2.3. Константы, переменные и типы

В Ruby переменные не имеют типа, однако объекты, на которые переменные ссылаются, тип имеют. Простейшие типы — это символ, число и строка.

Числовые константы интуитивно наиболее понятны, равно как и строки. В общем случае строка, заключенная в двойные кавычки, допускает интерполяцию выражений, а заключенная в одиночные кавычки интерпретируется почти буквально — в ней распознается только экранированная обратная косая черта.

Ниже показана «интерполяция» переменных и выражений в строку, заключенную в двойные кавычки:

```
a = 3
b = 79
puts "#{a} умноженное на #{b}   #{a*b}"   #  3 умноженное на 79   237
```

Более подробная информация о литералах (числах, строках, регулярных выражениях и т.п.) приведена в следующих главах.

Стоит упомянуть особую разновидность строк, которая полезна прежде всего в небольших скриптах, применяемых для объединения более крупных программ. Строка, выводимая программой, посылается операционной системе в качестве подлежащей исполнению команды, а затем результат выполненной команды подставляется обратно в строку. В простейшей форме для этого применяются строки, заключенные в обратные кавычки. В более сложном варианте используется синтаксическая конструкция %x:

```
`whoami`
`ls -l`
%x[grep -i meta *.html | wc -l]
```

Регулярные выражения в Ruby похожи на символьные строки, но используются по-другому. Обычно в качестве ограничителя выступает символ косой черты.

Синтаксис регулярных выражений в Ruby и Perl имеет много общего. Подробнее о регулярных выражениях см. главу 3.

Массивы в Ruby – очень мощная конструкция; они могут содержать данные любого типа. Более того, в одном массиве можно хранить данные разных типов. В главе 8 мы увидим, что все массивы – это экземпляры класса Array, а потому к ним применимы разнообразные методы. Массив-константа заключается в квадратные скобки. Примеры:

```
[1, 2, 3]
[1, 2, "застегни мне молнию на сапоге"]
[1, 2, [3,4], 5]
["alpha", "beta", "gamma", "delta"]
```

Во втором примере показан массив, содержащий целые числа и строки. В третьем примере мы видим вложенный массив, а в четвертом – массив строк. Как и в большинстве других языков, нумерация элементов массива начинается с нуля. Так, в последнем из показанных выше примеров элемент "gamma" имеет индекс 2. Все массивы динамические, задавать размер при создании не нужно.

Поскольку массивы строк встречаются очень часто (а набирать их неудобно), для них предусмотрен специальный синтаксис:

```
%w[alpha beta gamma delta]
%w(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)
%w/am is are was were be being been/
```

Такую сокращенную запись часто называют «синтаксической глазурью», потому что это более удобная альтернатива другой синтаксической конструкции. В данном случае не нужны ни кавычки, ни запятые; элементы разделяются пробелами. Если встречаются элементы, содержащие внутренние пробелы, такой синтаксис, конечно, неприменим.

Для доступа к конкретному элементу массива по индексу применяются квадратные скобки. Можно как получить элемент, так и присвоить ему новое значение:

```
val = myarray[0]
print stats[j]
x[i] = x[i+1]
```

Еще одна важная конструкция в Ruby – *хэш*. Его также называют *ассоциативным массивом* или *словарем*. Хэш – это множество пар данных; обыкновенно он применяется в качестве справочной таблицы или как обобщенный массив, в котором индекс не обязан быть целым числом. Все хэши являются экземплярами класса `Hash`.

Хэш-константа как правило заключается в фигурные скобки, а ключи отделяются от значений символом =>. Ключ можно считать индексом для доступа к ассоциированному с ним значению. На типы ключей и значений не налагается никаких ограничений. Примеры:

```
{1 => 1, 2 => 4, 3 => 9, 4 => 16, 5 => 25, 6 => 36}
{"cat" => "cats", "ox" => "oxen", "bacterium" => "bacteria"}
{"odds" => [1,3,5,7], "evens" => [2,4,6,8]}
{"foo"=>123, [4,5,6]=>"my array", "867-5309"=>"Jenny"}
```

Для хэшей имеется еще одна синтаксическая конструкция, она позволяет создавать ключи, являющиеся экземплярами класса `Symbol` (что это такое, будет объяснено ниже):

```
{hydrogen: 1, helium: 2, carbon: 12}
```

К содержимому хэша-переменной доступ осуществляется так же, как для массивов, – с помощью квадратных скобок:

```
print phone_numbers["Jenny"]
plurals["octopus"] = "octopi"
atomic_numbers[:helium] #-> 2
```

Однако следует подчеркнуть, что у массивов и хэшей много методов, именно они и делают эти контейнеры полезными. В разделе «ООП в Ruby» ниже в этой главе мы раскроем эту тему глубже.

1.2.4. Операторы и приоритеты

Познакомившись с основными типами данных, перейдем к операторам в языке Ruby. В приведенной ниже таблице они представлены в порядке убывания приоритета:

`::`	Разрешение области видимости
`[]`	Взятие индекса
`**`	Возведение в степень
`+ - ! ~`	Унарный плюс/минус, НЕ ...
`* / %`	Умножение, деление ...
`+ -`	Сложение/вычитание
`<< >>`	Логические сдвиги ...
`&`	Поразрядное И
`| ^`	Поразрядное ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ
`> >= < <=`	Сравнение

== === <=> != =~ !~	Равенство, неравенство ...
&&	Логическое И
||	Логическое ИЛИ
.. ...	Операторы диапазона
= (also +=, -=, ...)	Присваивание
?:	Тернарный выбор
not	Логическое отрицание
and or	Логическое И, ИЛИ

Некоторые из приведенных выше символов служат сразу нескольким целям. Например, оператор << обозначает поразрядный сдвиг влево, но также применяется для добавления в конец (массива, строки и т.д.) и как маркер встроенного документа. Аналогично, + означает сложение чисел и конкатенацию строк. Ниже мы увидим, что многие операторы – это просто сокращенная запись вызова методов.

Итак, мы определили большую часть типов данных и многие из возможных над ними операций. Прежде чем двигаться дальше, приведем пример программы.

1.2.5. Пример программы

В любом руководстве первой всегда приводят программу, печатающую строку Hello, world!. Но мы рассмотрим что-нибудь более содержательное. Вот небольшая интерактивная консольная программа, позволяющая переводить температуру из шкалы Фаренгейта в шкалу Цельсия и наоборот.

```
print "Введите температуру и шкалу (C or F): "
str = gets
exit if str.nil? or str.empty?
str.chomp!
temp, scale = str.split(" ")

abort "#{temp} недопустимое число." if temp !~ /-?\d+/

temp = temp.to_f
case scale
  when "C", "c"
    f = 1.8*temp + 32
  when "F", "f"
    c = (5.0/9.0)*(temp-32)
else
  abort "Необходимо задать C или F."
end

if f.nil?
  print "#{c} градусов C\n"
else
  print "#{f} градусов F\n"
end
```

Ниже приведены примеры прогона этой программы. Показано, как она переводит градусы Фаренгейта в градусы Цельсия и наоборот, а также как обрабатывает неправильно заданную шкалу или число:

```
Введите температуру и шкалу (C or F): 98.6 F
37.0 градусов C

Введите температуру и шкалу (C or F): 100 C
212.0 градусов F

Введите температуру и шкалу (C or F): 92 G
Необходимо задать C или F.

Введите температуру и шкалу (C or F): junk F
junk недопустимое число.
```

Теперь рассмотрим, как эта программа работает. Все начинается с предложения `print`, которое есть не что иное, как вызов метода `print` из модуля `Kernel`. Этот метод выполняет печать на стандартный вывод. Это самый простой способ оставить курсор в конце строки.

Далее мы вызываем метод `gets` (прочитать строку из стандартного ввода) и присваиваем полученное значение переменной `str`. Для удаления хвостового символа новой строки вызывается метод `chomp!`.

Обратите внимание, что `print` и `gets`, которые выглядят как «свободные» функции, на самом деле являются методами класса `Object` (который, вероятно, наследует `Kernel`). Точно также, `chomp!` – метод, вызываемый от имени объекта `str`. При вызовах методов в Ruby обычно можно опускать скобки: `print "foo"` и `print("foo")` – одно и то же.

В переменной `str` хранится символьная строка, но могли бы храниться данные любого другого типа. В Ruby данные имеют тип, а переменные – нет. Переменная начинает существовать, как только интерпретатор распознает присваивание ей; никаких предварительных объявлений не существует.

Метод `exit` завершает программу. В той же строке мы видим управляющую конструкцию, которая называется «модификатор if». Он аналогичен предложению `if`, существующему в большинстве языков, только располагается после действия. Для модификатора if нельзя задать ветвь else, и он не требует закрытия. Что касается условия, то мы проверяем две вещи: имеет ли переменная `str` значение (то есть не равна `nil`) и не является ли она пустой строкой? Если встретится конец файла, то будет истинно первое условие, если же пользователь нажмет клавишу Enter, не введя никаких данных, – второе.

Оператор `||` делает то же, что и `or`, но лучше использовать его, потому что он имеет более высокий приоритет и потому дает более предсказуемые результаты. Это предложение можно было бы записать и по-другому:

```
exit if not str or not str[0]
```

Эти проверки работают потому, что переменная может иметь значение nil, а nil в Ruby в логическом контексте вычисляется как «ложно». На самом деле, как «ложно» вычисляются nil и false, а все остальное – как «истинно». Это означает, кстати, что пустая строка "" и число 0 – не «ложно».

В следующем предложении над строкой выполняется операция chomp! (для удаления хвостового символа новой строки). Восклицательный знак в конце предупреждает, что операция изменяет значение самой строки, а не возвращает новую. Восклицательный знак применяется во многих подобных ситуациях как напоминание программисту о том, что у метода есть побочное действие или что он более «опасен», чем аналогичный метод без восклицательного знака. Так, метод chomp возвращает такой же результат, но не модифицирует значение строки, для которой вызван.

В следующем предложении мы видим пример множественного присваивания. Метод split разбивает строку на куски по пробелам и возвращает массив. Двум переменным в левой части оператора присваиваются значения первых двух элементов массива в правой части.

В следующем предложении if с помощью простого регулярного выражения выясняется, введено ли допустимое число. Если строка не соответствует образцу, который состоит из необязательного знака минус и одной или более цифр, то число считается недопустимым, и программа завершается. Отметим, что предложение if оканчивается ключевым словом end. Хотя в данном случае это не нужно, мы могли бы включить перед end ветвь else. Ключевое слово then необязательно, в этой книге мы стараемся не употреблять его.

Метод to_f преобразует строку в число с плавающей точкой. Это число записывается в ту же переменную temp, в которой раньше хранилась строка.

Предложение case выбирает одну из трех ветвей: пользователь указал C, F или какое-то другое значение в качестве шкалы. В первых двух случаях выполняется вычисление, в третьем мы печатаем сообщение об ошибке и выходим.

Кстати, предложение case в Ruby позволяет гораздо больше, чем показано в примере. Нет никаких ограничений на типы данных, а все выражения могут быть произвольно сложными, в том числе диапазонами или регулярными выражениями.

В самом вычислении нет ничего интересного. Но обратите внимание, что переменные c и f впервые встречаются внутри ветвей case. В Ruby нет никаких объявлений, переменная начинает существовать только в результате присваивания. А это означает, что после выхода из case лишь одна из переменных c и f будет иметь действительное значение.

Мы воспользовались этим фактом, чтобы понять, какая ветвь исполнялась, и в зависимости от этого вывести то или другое сообщение. Сравнение f с nil позволяет узнать, есть ли у переменной осмысленное значение. Мы использовали этот прием только для демонстрации возможности, ясно, что при желании можно было бы поместить печать прямо внутрь предложения case.

Внимательный читатель заметит, что мы пользовались только «локальными» переменными. Это может показаться странным, так как, на первый взгляд, их об-

ластью видимости является вся программа. На самом деле они локальны относительно верхнего уровня программы. Глобальными они кажутся лишь потому, что в этой простой программе нет контекстов более низкого уровня. Но если бы мы объявили какие-нибудь классы или методы, то в них переменные верхнего уровня были бы не видны.

1.2.6. Циклы и ветвление

Потратим немного времени на изучение управляющих конструкций. Мы уже видели простое предложение `if` и модификатор `if`. Существуют также парные структуры, в которых используется ключевое слово `unless` (в них также может присутствовать необязательная ветвь `else`), а равно применяемые в выражениях формы `if` и `unless`. Так, следующие два предложения эквивалентны:

```
if x < 5 then
        statement1

    unless x >= 5 then
      statement1
```

И эти два тоже:

```
if x < 5 then
  statement1
else
  statement2
end

unless x < 5 then
  statement2
else
   statement1
end
```

И эти:

```
statement1 if y == 3

statement1 unless y != 3
```

И вот эти:

```
x = if a>0 then b else c end
x = unless a<=0 then c else b end
```

Обратите внимание, что слово `then` можно опускать во всех случаях, кроме последнего (предназначенного для использования в выражениях). Также заметьте, что в модификаторах ветвь `else` не допускается.

Предложение `case` в Ruby позволяет больше, чем в других языках. В его ветвях можно проверять различные условия, а не только сравнивать на равенство. Так, например, разрешено сопоставление с регулярным выражением. Проверки в предложении `case` эквивалентны *оператору ветвящегося равенства* (`===`), поведение которого зависит от объекта. Рассмотрим пример:

```
case "Это строка символов."
  when "одно значение"
    puts "Ветвь 1"
  when "другое значение"
    puts "Ветвь 2"
  when /симв/
    puts "Ветвь 3"
  else
    puts "Ветвь 4"
end
```

Этот код напечатает Ветвь 3. Почему? Сначала проверяемое выражение сравнивается на равенство с двумя строками: «одно значение» и «другое значение». Эта проверка завершается неудачно, поэтому мы переходим к третьей ветви. В третьей ветви находится образец, с которым сопоставляется выражение. Поскольку оно соответствует образцу, то выполняется предложение print. В ветви else обрабатывается случай, когда ни одна из предшествующих проверок не прошла.

Если проверяемое выражение – целое число, то его можно сравнивать с целочисленным диапазоном (например, 3..8). В таком случае проверяется, что число попадает в диапазон. В любом случае выполняется код в первой подошедшей ветви.

В Ruby имеется богатый набор циклических конструкций. while и until – циклы с предварительной проверкой условия, и оба работают привычным образом: в первом случае задается условие продолжения цикла, а во втором – условие завершения. Есть также их формы с модификатором, как для предложений if и unless. Кроме того, в модуле Kernel есть метод loop (по умолчанию бесконечный цикл), а в некоторых классах реализованы итераторы.

В примерах ниже предполагается, что где-то определен такой массив list:

```
list = %w[alpha bravo charlie delta echo]
```

В цикле этот массив обходится и печатается каждый его элемент.

```
i=0                          # Цикл 1 (while)
while i < list.size do
  print "#{list[i]} "
  i += 1
end


i=0                          # Цикл 2 (until)
until i == list.size do
  print "#{list[i]} "
  i += 1
end

i = 0                        # Цикл 3 (while с проверкой в конце)
begin
   print "#{list[i]} "
   i += 1
```

```
end while i < list.size

i = 0                        # Цикл 4 (until с проверкой в конце)
begin
     print "#{list[i]} "
     i += 1
end until i == list.size

for x in list do  # Цикл 5 (for)
  print "#{x} "
end

list.each do |x|             # Цикл 6 (итератор 'each')
  print "#{x} "
end


i=0                          # Цикл 7 (метод 'loop')
n=list.size-1
loop do
  print "#{list[i]} "
  i += 1
  break if i > n
end

i=0                          # Цикл 8 (метод 'loop')
n=list.size-1
loop do
  print "#{list[i]} "
  i += 1
  break unless i <= n
end


n=list.size                  # Цикл 9 (итератор 'times')
n.times do |i|
  print "#{list[i]} "
end


n=list.size-1                # Цикл 10 (итератор 'upto')
0.upto(n) do |i|
  print "#{list[i]} "
end


n=list.size-1                # Цикл 11(for)
for
```

```
  i in 0..n do
  print "#{list[i]} "
end

list.each_index do |x|     # Цикл 12 (итератор 'each_index')
  print "#{list[x]} "
end
```

Рассмотрим эти примеры более подробно. Циклы 1 и 2 – «стандартные» формы циклов `while` и `until`; ведут они себя практически одинаково, только условия противоположны. Циклы 3 и 4 – варианты предыдущих с проверкой условия в конце, а не в начале итерации. Отметим, что использование слов `begin` и `end` в этом контексте просто грязный трюк; на самом деле это был бы блок `begin/end` (применяемый для обработки исключений), за которым следует модификатор `while` или `until`. Иначе говоря, эта конструкция приведена только для иллюстрации. Не пишите так.

На мой взгляд, цикл 6 – самый «правильный» способ кодирования циклов. Обратите внимание, насколько циклы 5 и 6 проще всех остальных; в них нет ни явной инициализации, ни явной проверки или инкремента. Это возможно потому, что массив «знает» свой размер, а стандартный итератор `each` (цикл 6) обрабатывает такие детали автоматически. На самом деле в цикле 3 просто производится неявное обращение к этому итератору, поскольку цикл `for` работает с любым объектом, для которого определен итератор `each`. Цикл `for` – всего лишь другой способ обращения к `each`.

В циклах 7 и 8 используется конструкция `loop`. Выше мы уже отмечали, что хотя `loop` выглядит как ключевое слово, которое начинает управляющую конструкцию, на самом деле это метод модуля `Kernel`.

В циклах 9 и 10 используется тот факт, что у массива есть числовой индекс. Итератор `times` исполняется заданное число раз, а итератор `upto` увеличивает свой параметр до заданного значения. И тот, и другой для данной ситуации приспособлены плохо.

Цикл 11 – это вариант цикла `for`, предназначенный специально для работы со значениями индекса, заданными с помощью диапазона. В цикле 12 мы пробегаем весь диапазон индексов массива с помощью итератора `each_index`.

В предыдущих примерах мы уделили недостаточно внимания вариантам циклов `while` и `loop` с модификаторами. Они довольно часто используются благодаря своей краткости. Вот еще два примера, в которых делается одно и то же:

```
perform_task() until finished

perform_task() while not finished
```

Также приведенные примеры недостаточно ясно показывают, что циклы не всегда выполняются от начала до конца. Число итераций и момент завершения цикла не всегда предсказуемы. Нужны дополнительные средства управления циклами.

Первое из них – ключевое слово `break`, встречающееся в циклах 7 и 8. Оно позволяет досрочно выйти из цикла; в случае вложенных циклов происходит выход из самого внутреннего. Для программистов на C это интуитивно очевидно.

Ключевое слово redo – обобщение retry на циклы общего вида. Оно работает в циклах while и until, как retry в итераторах.

Ключевое слово next осуществляет переход на конец самого внутреннего цикла и возобновляет исполнение с этой точки. Работает для любого цикла и итератора.

Как мы только что видели, итератор – важное понятие в Ruby. Но следует отметить, что язык позволяет определять и пользовательские итераторы, не ограничиваясь встроенными.

Стандартный итератор для любого объекта называется each. Это существенно отчасти из-за того, что позволяет использовать цикл for. Но итераторам можно давать и другие имена и применять для разных целей.

С помощью yield можно также передать параметры, которые будут подставлены в список параметров блока (между вертикальными черточками). В следующем искусственном примере итератор всего лишь генерирует целые числа от 1 до 10, а вызов итератора порождает кубические степени этих чисел:

```
def my_sequence
   (1..10).each do |i|
        yield i
   end
end

my_sequence {|x| puts x**3 }
```

Отметим, что вместо фигурных скобок, в которые заключен блок, можно написать ключевые слова do и end. Различия между этими формами есть, но довольно тонкие.

1.2.7. Исключения

Как и многие другие современные языки, Ruby поддерживает исключения.

Исключения – это механизм обработки ошибок, имеющий существенные преимущества по сравнения с прежними подходами. Нам удается избежать возврата кодов ошибок и запутанной логики их анализа, а код, который обнаруживает ошибку, можно отделить от кода, который ее обрабатывает (чаще всего они так или иначе разделены).

Предложение raise возбуждает исключение. Отметим, что raise – не зарезервированное слово, а метод модуля Kernel. (У него есть синоним fail.)

```
raise    # Пример 1
raise "Произошла ошибка"                          # Пример 2
raise ArgumentError                               # Пример 3
raise ArgumentError, "Неверные данные"            # Пример 4
raise ArgumentError.new("Неверные данные ")       # Пример 5
raise ArgumentError, " Неверные данные ", caller[0]   # Пример 6
```

В первом примере повторно возбуждается последнее встретившееся исключение. В примере 2 создается исключение RuntimeError (подразумеваемый тип), которому передается сообщение "Произошла ошибка".

В примере 3 возбуждается исключение типа ArgumentError, а в примере 4 такое же исключение, но с сообщением "Неверные данные". Пример 5 – просто другая запись примера 4. Наконец, в примере 6 еще добавляется трассировочная информация вида "filename:line" или "filename:line:in 'method'" (хранящаяся в массиве caller).

А как обрабатываются исключения в Ruby? Для этой цели служит блок begin-end. В простейшей форме внутри него нет ничего, кроме кода:

```
begin
  # Ничего полезного.
  # ...
end
```

Просто перехватывать ошибки не очень осмысленно. Но у блока может быть один или несколько обработчиков rescue. Если произойдет ошибка в любой точке программы между begin и rescue, то управление сразу будет передано в подходящий обработчик rescue.

```
begin
  x = Math.sqrt(y/z)
  # ...
rescue ArgumentError
  puts "Ошибка при извлечении квадратного корня."
rescue ZeroDivisionError
  puts "Попытка деления на нуль."
end
```

Того же эффекта можно достичь следующим образом:

```
begin
  x = Math.sqrt(y/z)
  # ...
rescue => err
  puts err
end
```

Здесь в переменной err хранится объект-исключение; при выводе ее на печать объект будет преобразован в осмысленную символьную строку. Отметим, что коль скоро тип ошибки не указан, то этот обработчик rescue будет перехватывать все исключения, производные от класса StandardError. В конструкции rescue => variable можно перед символом => дополнительно указать тип ошибки.

Если типы ошибок указаны, то может случиться так, что тип реально возникшего исключения не совпадает ни с одним из них. На этот случай после всех обработчиков rescue разрешается поместить ветвь else.

```
begin
  # Код, в котором может возникнуть ошибка...
rescue Type1
  # ...
rescue Type2
  # ...
else
```

```
  # Прочие исключения...
end
```

Часто мы хотим каким-то образом восстановиться после ошибки. В этом поможет ключевое слово retry (внутри тела обработчика rescue). Оно позволяет повторно войти в блок begin и попытаться еще раз выполнить операцию:

```
begin
  # Код, в котором может возникнуть ошибка...
rescue
  # Пытаемся восстановиться...
  retry # Попробуем еще раз
end
```

Наконец, иногда необходим код, который «подчищает» что-то после выполнения блока begin-end. В этом случае можно добавить часть ensure:

```
begin
  # Код, в котором может возникнуть ошибка...
rescue
  # Обработка исключений
ensure
  # Этот код выполняется в любом случае
end
```

Код, помещенный внутрь части ensure, выполняется при любом способе выхода из блока begin-end – произошло исключение или нет.

Исключения можно перехватывать еще двумя способами. Во-первых, существует форма rescue в виде модификатора:

```
x = a/b rescue puts("Деление на нуль!")
```

Кроме того, тело определения метода представляет собой неявный блок begin-end; слово begin опущено, а все тело метода подготовлено к обработке исключения и завершается словом end:

```
def some_method
  # Код...
rescue
  # Восстановление после ошибки...
end
```

На этом мы завершаем как обсуждение обработки исключений, так и рассмотрение основ синтаксиса и семантики в целом.

У языка Ruby есть многочисленные аспекты, которых мы не коснулись. Оставшаяся часть главы посвящена более развитым возможностям языка, в том числе рассмотрению ряда практических приемов, которые помогут программисту среднего уровня научиться «думать в духе Ruby».

1.3. ООП в Ruby

В языке Ruby есть все элементы, которые принято ассоциировать с объектно-ориентированными языками: объекты с инкапсуляцией и сокрытием данных, методы с полиморфизмом и переопределением, классы с иерархией и наследованием. Но

он идет дальше, добавляя ограниченные возможности создания метаклассов, синглетные методы, модули и классы-примеси.

Похожие идеи, только под иными именами, встречаются и в других объектно-ориентированных языках, но одно и то же название может скрывать тонкие различия. В этом разделе мы уточним, что в Ruby понимается под каждым из элементов ООП.

1.3.1. Объекты

В Ruby все числа, строки, массивы, регулярные выражения и многие другие сущности фактически являются объектами. Работа программы состоит в вызове методов разных объектов:

```
3.succ   # 4
"abc".upcase  # "ABC"
[2,1,5,3,4].sort # [1,2,3,4,5]
someObject.someMethod            # какой-то результат
```

В Ruby каждый объект есть экземпляр какого-то класса. Класс содержит реализацию методов:

```
"abc".class   # String
"abc".class.class# Class
```

Помимо инкапсуляции собственных атрибутов и операций, объект в Ruby имеет уникальный идентификатор:

```
"abc".object_id  # 53744407
```

Этот идентификатор объекта обычно не представляет интереса для программиста.

1.3.2. Встроенные классы

Свыше 30 классов уже встроено в Ruby. Как и во многих других ОО-языках, в Ruby не допускается множественное наследование, но это еще не означает, что язык стал менее выразительным. Современные языки часто построены согласно модели одиночного наследования. Ruby поддерживает модули и классы-примеси, которые мы обсудим в следующей главе. Также реализованы идентификаторы объектов, что позволяет строить устойчивые, распределенные и перемещаемые объекты.

Чтобы создать объект существующего класса, обычно используется метод `new`:

```
myFile = File.new("textfile.txt","w")
myString = String.new("Это строковый объект")
```

Однако не всегда его обязательно вызывать явно. При создании *литеральных объектов* можно и не упоминать метод `new`:

```
your_string = "Это тоже строковый объект"
number = 5     # и здесь метод new не нужен
```

Ссылки на объекты хранятся в переменных. Выше уже отмечалось, что сами переменные не имеют типа и не являются объектами, они лишь ссылаются на объекты.

```
x = "abc"
```

Из этого правила есть исключение: небольшие неизменяемые объекты некоторых встроенных классов, например Fixnum, непосредственно копируются в переменные, которые на них ссылаются. (Размер этих объектов не превышает размера указателя, поэтому хранить их таким образом более эффективно.) В таком случае во время присваивания делается копия объекта, а куча не используется.

При присваивании переменных ссылки на объекты обобществляются.

```
y = "abc"
x = y
x              # "abc"
```

После выполнения присваивания x = y и x, и y ссылаются на один и тот же объект:

```
x.object_id    # 53732208
y.object_id    # 53732208
```

Если объект изменяемый, то модификация, примененная к одной переменной, отражается и на другой:

```
x.gsub!(/a/,"x")
y              # "xbc"
```

Однако новое присваивание любой из этих переменных не влияет на другую:

```
# Продолжение предыдущего примера
x = "abc"


y        # по-прежнему равно "xbc"
```

Изменяемый объект можно сделать неизменяемым, вызвав метод freeze:

```
x.freeze
x.gsub!(/b/,"y") # Ошибка!
```

Символ – вещь не вполне обычная, напоминающая атом в Lisp. Он ведет себя как неизменяемая строка, при этом все вхождения символа ссылаются на одно и то же значение. Символ можно преобразовать в строку с помощью метода to_s.

```
suits = [:hearts, :clubs, :diamonds, :spades]
lead = suits[1].to_s # "clubs"
```

Подобно массивам строк, можно создавать и массивы символов – с помощью синтаксической конструкции %i:

```
suits = %i[hearts clubs diamonds spades] # массив символов
```

1.3.3. Модули и классы-примеси

Многие встроенные методы наследуются от классов-предков. Особо стоит отметить методы модуля Kernel, подмешиваемые к суперклассу Object. Поскольку класс Object повсеместно доступен, то и добавленные в него из Kernel методы также доступны в любой точке программы. Эти методы играют важную роль в Ruby.

Термины *модуль* и *примесь* – почти синонимы. Модуль представляет собой набор методов и констант, внешних по отношению к программе на Ruby. Его можно

использовать просто для управления пространством имен, но основное применение модулей связано с «подмешиванием» его возможностей в класс (с помощью директивы include). В таком случае он используется как класс-примесь.

Этот термин очевидно заимствован из языка Python. Стоит отметить, что в некоторых вариантах LISP такой механизм существует уже больше двадцати лет.

Не путайте описанное выше употребление термина *модуль* с другим значением, которое часто придается ему в информатике. Модуль в Ruby – это не внешний исходный текст и не двоичный файл (хотя может храниться и в том, и в другом виде). Это объектно-ориентированная абстракция, в чем-то похожая на класс.

Примером использования модуля для управления пространством имен служит модуль Math. Так, чтобы получить определение числа π, необязательно включать модуль Math с помощью предложения include, достаточно просто написать Math::PI.

Примесь дает способ получить преимущества множественного наследования, не отягощенные характерными для него проблемами. Можно считать, что это ограниченная форма множественного наследования, но создатель языка Мац называет его *одиночным наследованием с разделением реализации*.

Отметим, что предложение include включает имена из указанного пространства имен (модуля) в текущее. Метод extend добавляет объекту функции из модуля. В случае применения include методы модуля становятся доступны как методы экземпляра, а в случае extend – как методы класса.

Необходимо оговориться, что операции load и require не имеют ничего общего с модулями, они относятся к исходным и двоичным файлам Ruby (загружаемым динамически или статически). Операция load читает файл и исполняет его в текущем контексте, так что, начиная с этой точки, становятся видимы все находящиеся в прочитанном файле определения. Операция require аналогична load, но не загружает файл, если он уже был загружен ранее.

Программисты, только начинающие осваивать Ruby, особенно имеющие опыт работы с языком C, могут поначалу путать операции require и include, которые никак не связаны между собой. Вы еще поймаете себя на том, что сначала вызываете require, а потом include для того, чтобы воспользоваться каким-то внешним модулем.

1.3.4. Создание классов

В Ruby есть множество встроенных классов, и вы сами можете определять новые. Для определения нового класса применяется такая конструкция:

```
class ClassName
   # ...
end
```

Само имя класса – это глобальная константа, поэтому оно должно начинаться с заглавной буквы. Определение класса может содержать константы, переменные класса, методы класса, переменные экземпляра и методы экземпляра. Данные уровня класса доступны всем объектам этого класса, тогда как данные уровня экземпляра доступны только одному объекту.

Попутное замечание. Строго говоря, классы в Ruby не имеют имен. «Имя» класса – это всего лишь константа, ссылающаяся на объект типа Class (поскольку в Ruby Class – это класс). Ясно, что на один и тот же класс могут ссылаться несколько констант и их можно присваивать переменным точно так же, как мы поступаем с любыми другими объектами (поскольку в Ruby Class – это объект). Если вы немного запутались, не расстраивайтесь. Удобства ради новичок может считать, что в Ruby имя класса – то же самое, что в C++.

Вот как определяется простой класс:

```
class Friend
  @@myname = "Эндрю" # переменная класса

  def initialize(name, sex, phone)
    @name, @sex, @phone = name, sex, phone
    # Это переменные экземпляра
  end

  def hello   # метод экземпляра
    puts "Привет, я #{@name}."
  end

  def Friend.our_common_friend   # метод класса
    puts "Все мы друзья #{@@myname}."
  end

end

f1 = Friend.new("Сюзанна","F","555-0123")
f2 = Friend.new("Том","M","555-4567")

f1.hello    # Привет, я Сюзанна.
f2.hello    # Привет, я Том.
Friend.our_common_friend # Все мы друзья Эндрю.
```

Поскольку данные уровня класса доступны во всем классе, их можно инициализировать в момент определения класса. Если определен метод с именем initialize, то гарантируется, что он будет вызван сразу после выделения памяти для объекта. Этот метод похож на традиционный конструктор, но не выполняет выделения памяти. Память выделяется методом new, а освобождается неявно сборщиком мусора.

Теперь взгляните на следующий фрагмент, обращая особое внимание на методы getmyvar, setmyvar и myvar=:

```
class MyClass

  NAME = "Class Name"          # константа класса
  @@count = 0                  # инициализировать переменную класса

  def initialize               # вызывается после выделения памяти для объекта
    @@count += 1
```

```
    @myvar = 10
  end

  def MyClass.getcount          # метод класса
    @@count                     # переменная класса
  end

  def getcount                  # экземпляр возвращает переменную класса!
    @@count                     # переменная класса
  end

  def getmyvar                  # метод экземпляра
    @myvar                      # переменная экземпляра
  end

  def setmyvar(val)             # метод экземпляра устанавливает @myvar
    @myvar = val
  end

  def myvar=(val)               # другой способ установить @myvar
    @myvar = val
  end
end

foo = MyClass.new   # @myvar равно 10
foo.setmyvar 20     # @myvar равно 20
foo.myvar = 30      # @myvar равно 30
```

Здесь мы видим, что `getmyvar` возвращает значение переменной `@myvar`, а `setmyvar` устанавливает его. (Многие программисты говорят о методах чтения и установки). Все это работает, но не является характерным способом действий в Ruby. Метод `myvar=` похож на перегруженный оператор присваивания (хотя, строго говоря, таковым не является); это более удачная альтернатива `setmyvar`, но есть способ еще лучше.

Класс `Module` содержит методы `attr`, `attr_accessor`, `attr_reader` и `attr_writer`. Ими можно пользоваться (передавая символы в качестве параметров) для автоматизации управления доступом к данным экземпляра. Например, все три метода `getmyvar`, `setmyvar` и `myvar=` можно заменить одной строкой в определении класса:

```
attr_accessor :myvar
```

При этом создается метод `myvar`, который возвращает значение `@myvar`, и метод `myvar=`, который позволяет изменить значение той же переменной. Методы `attr_reader` и `attr_writer` создают соответственно версии методов доступа к атрибуту для чтения и для изменения.

Внутри методов экземпляра, определенных в классе, можно при необходимости пользоваться переменной `self`. Это просто ссылка на объект, от имени которого вызван метод экземпляра.

Для управления видимостью методов класса можно пользоваться модификаторами private, protected и public. (Переменные экземпляра всегда закрыты, обращаться к ним извне класса можно только с помощью методов доступа.) Каждый модификатор принимает в качестве параметра символ, например, :foo, а если он опущен, то действие модификатора распространяется на все последующие определения в классе. Пример:

```
class MyClass

  def method1
  # ...
  end
  def method2
  # ...
  end

  def method3
  # ...
  end

  private :method1
  public

  :method2
  protected :method3

  private

  def my_method
  # ...
  end

  def another_method
  # ...
  end

end
```

В этом классе метод method1 закрытый, method2 открытый, а method3 защищенный. Поскольку далее вызывается метод private без параметров, то методы my_method и another_method будут закрытыми.

Уровень доступа public не нуждается в объяснениях, он не налагает никаких ограничений ни на доступ к методу, ни на его видимость. Уровень private означает, что метод доступен только внутри класса или его подклассов и может вызываться только в «функциональной форме» от имени self, причем вызывающий объект может указываться явно или подразумеваться неявно. Уровень protected означает, что метод может вызываться только внутри класса, но в отличие от закрытого метода, не обязательно от имени self.

По умолчанию все определенные в классе методы открыты. Исключение составляет только метод initialize. Методы, определенные на верхнем уровне программы, тоже по умолчанию открыты. Если они объявлены закрытыми, то могут вызываться только в функциональной форме (как, например, методы, определенные в классе Object).

Классы в Ruby сами являются объектами – экземплярами метакласса Class. Классы в Ruby всегда конкретны, абстрактных классов не существует. Однако теоретически можно реализовать и абстрактные классы, если вам это для чего-то понадобится.

Класс Object является корнем иерархии. Он предоставляет все методы, определенные во встроенном модуле Kernel.

Чтобы создать класс, наследующий другому классу, нужно поступить следующим образом:

```
class MyClass < OtherClass
  # ...
end
```

Помимо использования встроенных методов, вполне естественно определить и собственные либо переопределить унаследованные. Если определяемый метод имеет то же имя, что и существующий, то старый метод замещается. Если новый метод должен обратиться к замещенному им «родительскому» методу (так бывает часто), то можно воспользоваться ключевым словом super.

Перегрузка операторов, строго говоря, не является неотъемлемой особенностью ООП, но этот механизм знаком программистам на C++ и некоторых других языках. Поскольку большинство операторов в Ruby так или иначе являются методами, то не должен вызывать удивления тот факт, что их можно переопределять или определять в пользовательских классах. Переопределять семантику оператора в существующем классе редко имеет смысл, зато в новых классах определение операторов – обычное дело.

Можно создавать синонимы методов. Для этого внутри определения класса предоставляется такой синтаксис:

```
alias_method :newname, :oldname
```

Число параметров будет таким же, как для старого имени, и вызываться метод-синоним будет точно так же. При использовании синонима создается копия метода, поэтому если впоследствии исходный метод будет изменен, на синониме это никак не отразится.

В Ruby существует также ключевое слово alias, которое ведет себя похоже, но в отличие от метода alias_method, оно позволяет создавать синонимы не только для методов, но и для глобальных переменных, а его аргументы не разделяются запятыми.

1.3.5. Методы и атрибуты

Как мы уже видели, методы обычно используются в сочетании с простыми экземплярами классов и переменными, причем вызывающий объект отделяется от име-

ни метода точкой (receiver.method). Если имя метода является знаком препинания, то точка опускается. У методов могут быть аргументы:

```
Time.mktime(2000, "Aug", 24, 16, 0)
```

Поскольку каждое выражение возвращает значение, то вызовы методов могут сцепляться:

```
3.succ.to_s
/(x.z).*?(x.z).*?/.match("x1z_1a3_x2z_1b3_").to_a[1..3]
3+2.succ
```

Отметим, что могут возникать проблемы, если выражение, являющееся результатом сцепления, имеет тип, который не поддерживает конкретный метод. Точнее, при определенных условиях некоторые методы возвращают nil, а вызов любого метода от имени такого объекта приведет к ошибке. (Конечно, nil – полноценный объект, но он не обладает теми же методами, что, например, массив.)

Некоторым методам можно передавать блоки. Это верно для всех итераторов, как встроенных, так и определенных пользователем. Блок обычно заключается в операторные скобки do-end или в фигурные скобки. Но он не рассматривается так же, как предшествующие ему параметры, если таковые существуют. Вот пример вызова метода File.open:

```
my_array.each do |x|
  some_action
end

File.open(filename) { |f| some_action }
```

Методы могут принимать переменное число аргументов:

```
receiver.method(arg1, *more_args)
```

В данном случае вызванный метод трактует more_args как массив и обращается с ним, как с любым другим массивом. На самом деле, звездочка в списке формальных параметров (перед последним или единственным параметром) может «свернуть» последовательность фактических параметров в массив:

```
def mymethod(a, b, *c)
  print a, b
  c.each do |x| print x end
end

mymethod(1,2,3,4,5,6,7)

# a=1, b=2, c=[3,4,5,6,7]
```

В Ruby поддерживаются также *именованные параметры*, которые в Python называются *ключевыми аргументами*; эта идея восходит еще к языку Ada, разработанному в 60–70-х годах прошлого века. Именованные параметры позволяют, с одной стороны, задавать значения по умолчанию, а, с другой, перечислять аргументы в произвольном порядке, потому что они снабжены явными метками:

```
def mymethod(name: "default", options: {})
    options.merge!(name: name)
    some_action_with(options)
end
```

Если подразумеваемое по умолчанию значение именованного параметра в определении метода опущено, считается, что такой именованный параметр обязателен:

```
def other_method(name:, age:)
    puts "Возраст человека #{name} составляет #{age} лет."
    # Попытка вызвать этот метод, не указав значения
    # name и age, будет считаться ошибкой.
end
```

В Ruby есть возможность определять методы на уровне объекта (а не класса). Такие методы называются синглетными; они принадлежат одному-единственному объекту и не оказывают влияния ни на класс, ни на его суперклассы. Такая возможность может быть полезна, например, при разработке графических интерфейсов пользователя – чтобы определить действие кнопки, вы задаете синглетный метод для данной и только данной кнопки.

Вот пример определения синглетного метода для строкового объекта:

```
str = "Hello, world!"
str2 = "Goodbye!"

def str.spell
  self.split(/./).join("-")
end

str.spell      # "H-e-l-l-o-,- -w-o-r-l-d-!"
str2.spell     # ошибка!
```

Имейте в виду, что метод определяется для объекта, а не для переменной.

Теоретически с помощью синглетных методов можно было бы создать систему объектов на базе прототипов. Это менее распространенная форма ООП без классов[1]. Основной структурный механизм в ней состоит в конструировании нового объекта путем использования существующего в качестве образца; новый объект ведет себя как старый за исключением тех особенностей, который были переопределены. Тем самым можно строить системы на основе прототипов, а не наследования. Хотя у нас нет опыта в этой области, мы полагаем, что создание такой системы позволило бы полнее раскрыть возможности Ruby.

1.4. Динамические аспекты Ruby

Ruby – динамический язык в том смысле, что объекты и классы можно изменять во время выполнения. Ruby позволяет конструировать и интерпретировать фрагменты кода в ходе выполнения статически написанной программы. В нем есть хи-

[1] Типичный пример – язык JavaScript. – *Прим. перев.*

троумный API отражения, с помощью которого программа может получать информацию о себе самой. Это позволяет сравнительно легко создавать отладчики, профилировщики и другие подобные инструменты, а также применять нетривиальные способы кодирования.

Наверное, это самая трудная тема для программиста, приступающего к изучению Ruby. В этом разделе мы вкратце рассмотрим некоторые следствия, вытекающие из динамической природы языка.

1.4.1. Кодирование во время выполнения

Мы уже упоминали директивы `load` и `require`. Важно понимать, что это не встроенные предложения и не управляющие конструкции, на самом деле это методы. Поэтому их можно вызывать, передавая переменные или выражения как параметры, в том числе условно. Сравните с директивой `#include` в языках C и C++, которая обрабатывается во время компиляции.

Код можно строить и интерпретировать по частям. В качестве несколько искусственного примера рассмотрим приведенный ниже метод `calculate` и вызывающий его код:

```
def calculate(op1, operator, op2)
  string = op1.to_s + operator + op2.to_s
    # предполагается, что operator - строка; построим длинную
    # строку, состоящую из оператора и операндов
  eval(string)    # вычисляем и возвращаем значение
end

@alpha = 25
@beta = 12

puts calculate(2, "+", 2)           # Печатается 4
puts calculate(5, "*", "@alpha")    # Печатается 125
puts calculate("@beta", "**", 3)    # Печатается 1728
```

Вот та же идея, доведенная чуть ли не до абсурда: программа запрашивает у пользователя имя метода и одну строку кода. Затем этот метод определяется и вызывается:

```
puts "Имя метода: "
meth_name = gets
puts "Строка кода: "
code = gets

string = %[def #{meth_name}\n #{code}\n end]    # Строим строку
eval(string)                                    # Определяем метод
eval(meth_name)                                 # Вызываем метод
```

Зачастую необходимо написать программу, которая могла бы работать на разных платформах или при разных условиях, но при этом сохранить общий набор исходных текстов. Для этого в языке C применяются директивы `#ifdef`, но в Ruby все определения исполняются. Не существует такого понятия, как «этап

компиляции», все конструкции динамические, а не статические. Поэтому для принятия решения такого рода мы можем просто вычислить условие во время выполнения:

```
if platform == Windows
  action1
elsif platform == Linux
  action2
else
  default_action
end
```

Конечно, за такое кодирование приходится расплачиваться некоторым снижением производительности, поскольку иногда условие приходится вычислять много раз. Но рассмотрим следующий пример, который делает практически то же самое, однако весь платформенно-зависимый код помещен в один метод, имя которого от платформы не зависит:

```
if platform == Windows
  def my_action
    action1
  end
elsif platform == Linux
  def my_action
    action2
  end
else
  def my_action
    default_action
  end
end
```

Таким способом мы достигаем желаемого результата, но условие вычисляется только один раз. Когда программа вызовет метод `my_action`, он уже будет правильно определен.

1.4.2. Отражение

В языках Smalltalk, LISP и Java реализована (с разной степенью полноты) идея *рефлексивного* программирования – активная среда может опрашивать структуру объектов и расширять либо модифицировать их во время выполнения.

В языке Ruby имеется развитая поддержка отражения, но все же он не заходит так далеко, как Smalltalk, где даже управляющие конструкции являются объектами. В Ruby управляющие конструкции и блоки объектами не являются. (Объект `Proc` можно использовать для того, чтобы представить блок в виде объекта, но управляющие конструкции объектами не бывают никогда.)

Для определения того, используется ли идентификатор с данным именем, служит ключевое слово `defined?` (обратите внимание на вопросительный знак в конце слова):

```
if defined? some_var
  puts "some_var = #{some_var}"
else
  puts "Переменная some_var неизвестна."
end
```

Аналогично, метод respond_to? выясняет, может ли объект отвечать на вызов указанного метода (то есть определен ли данный метод для данного объекта). Метод respond_to? определен в классе Object.

В Ruby запрос информации о типе во время выполнения поддерживается очень полно. Тип (или класс) объекта можно определить, воспользовавшись методом type (из класса Object). Метод is_a? сообщает, принадлежит ли объект некоторому классу (включая и его суперклассы); синонимом служит имя kind_of?. Например:

```
puts "abc".class ""              # Печатается String
puts 345.class                   # Печатается Fixnum
rover = Dog.new

print rover.class                # Печатается Dog

if rover.is_a? Dog
  puts "Конечно, является."
end

if rover.kind_of? Dog
  puts "Да, все еще собака."
end

if rover.is_a? Animal
  puts "Да, он к тому же и животное."
end
```

Можно получить полный список всех методов, которые можно вызвать для данного объекта. Для этого предназначен метод methods из класса Object. Имеются также его варианты private_instance_methods, public_instance_methods и т.д.

Аналогично можно узнать, какие переменные класса или экземпляра ассоциированы с данным объектом. По самой природе ООП, в перечни методов и переменных включаются те, что определены как в классе самого объекта, так и во всех его суперклассах. В классе Module имеется метод constants, позволяющий получить список всех констант, определенных в модуле.

В классе Module есть метод ancestors, возвращающий список модулей, включенных в данный модуль. В этот список входит и сам данный модуль, то есть список, возвращаемый вызовом Mod.ancestors, содержит по крайней мере элемент Mod. В этот список входят не только родительские классы (отобранные в силу наследования), но и «родительские» модули (отобранные в силу включения).

В классе BasicObject есть метод superclass, который возвращает суперкласс объекта или nil. Не имеет суперкласса лишь класс BasicObject, и, значит, только для него может быть возвращен nil.

Модуль ObjectSpace применяется для получения доступа к любому «живому» объекту. Метод _idtoref преобразует идентификатор объекта в ссылку на него, можно считать, что это операция, обратная методу object_id. В модуле ObjectSpace есть также итератор each_object, который перебирает все существующие в данный момент объекты, включая и те, о которых никаким иным образом узнать невозможно. (Напомним, что некоторые неизменяемые объекты небольшого размера, например принадлежащие классам Fixnum, NilClass, TrueClass и FalseClass, не хранятся в стеке из соображений оптимизации.)

1.4.3. Отсутствующие методы

При вызове метода (my_object.my_method), Ruby ищет поименованный метод в следующем порядке:

1. Синглетные методы, определенные для объекта my_object.
2. Методы, определенные в классе объекта my_object.
3. Методы, определенные в предках класса объекта my_object.

Если найти метод my_method не удается, Ruby ищет метод с именем method_missing. Если он определен, то ему передается имя отсутствующего метода (в виде символа) и все переданные ему параметры. Этот механизм можно применять для динамической обработки неизвестных сообщений, посланных во время выполнения.

1.4.4. Сборка мусора

Управлять памятью на низком уровне трудно и чревато ошибками, особенно в таком динамичном окружении, какое создает Ruby. Наличие механизма сборки мусора – весомое преимущество. В таких языках, как C++, за выделение и освобождение памяти отвечает программист. В более поздних языках, например Java, память освобождается сборщиком мусора (когда объект покидает область видимости).

Явное управление памятью может приводить к двум видам ошибок. Если освобождается память, занятая объектом, на который еще есть ссылки, то при последующем доступе к нему объект может оказаться в противоречивом состоянии. Так называемые *висячие указатели* трудно отлаживать, поскольку вызванные ими ошибки часто проявляются далеко от места возникновения. Утечка памяти имеет место, когда не освобождается объект, на который больше никто не ссылается. В этом случае программа потребляет все больше и больше памяти и в конечном итоге аварийно завершается; такие ошибки искать тоже трудно. В языке Ruby для отслеживания неиспользуемых объектов и освобождения занятой ими памяти применяется механизм сборки мусора. Для тех, кто разбирается, скажем, что в Ruby используется алгоритм пометки и удаления, а не подсчета ссылок (у последнего возникают трудности при обработке рекурсивных структур).

Сборка мусора влечет за собой некоторое снижение производительности. Модуль GC предоставляет ограниченные средства управления, позволяющие программисту настроить его работу в соответствии с нуждами конкретной программы. Можно также определить чистильщик (finalizer) объекта, но это уже продвинутая тема (см. раздел 11.3.15).

1.5. Потренируйте свою интуицию: что следует запомнить

Надо честно сказать: «все становится интуитивно очевидным после того, как поймешь». Эта истина и составляет суть данного раздела, поскольку в Ruby немало особенностей, отличающих его от всего, к чему привык программист на одном из традиционных языков.

Кто-то из читателей решит, что не нужно зря тратить время на повторение известного. Если вы из их числа, можете пропустить разделы, которые кажутся вам очевидными. Программисты имеют неодинаковый опыт; искушенные пользователи C и Smalltalk воспримут Ruby совсем по-разному. Впрочем, мы надеемся, что внимательное прочтение последующих разделов поможет многим читателям разобраться в том, что же такое Путь Ruby.

1.5.1. Синтаксис

Синтаксический анализатор Ruby сложен и склонен прощать многие огрехи. Он пытается понять, что хотел сказать программист, а не навязывать ему жесткие правила. Но к такому поведению надо еще привыкнуть. Вот перечень того, что следует знать о синтаксисе Ruby.

- Скобки при вызове методов как правило можно опускать. Все следующие вызовы допустимы:

```
foobar
foobar()
foobar(a,b,c)
foobar a, b, c
```

- Попробуем передать методу хэш:

```
my_method {a: 1, b: 2, 5 => 25}
```

  Это приведет к синтаксической ошибке, поскольку левая фигурная скобка воспринимается как начало блока. В данном случае скобки необходимы:

```
my_method({a: 1, b: 2, 5 => 25})
```

- Предположим теперь, что хэш — *единственный* (или последний) параметр метода. Ruby снисходительно разрешает опускать фигурные скобки:

```
my_method(a: 1, b: 2, 5 => 25)
```

  Кто-то увидит здесь вызов метода с именованными параметрами. На самом деле, это не так, хотя никто не запрещает применять подобную конструкцию и в таком смысле.

- Есть и другие случаи, когда пропуски имеют некоторое значение. Например, на первый взгляд все четыре выражения ниже означают одно и то же:

```
x = y + z
x = y+z
x = y+ z
x = y +z
```

Фактически же лишь первые три эквивалентны. А в четвертом случае анализатор считает, что вызван метод y с параметром +z! И выдаст сообщение об ошибке, так как метода с именем y не существует. Мораль – пользуйтесь пробелами разумно.

- Аналогично, x = y*z – это умножение y на z, тогда как x = y *z – вызов метода y, которому в качестве параметра передается расширение массива z.
- В именах идентификаторов подчерк считается строчной буквой. Следовательно, имя идентификатора может начинаться с подчерка, но такой идентификатор не будет считаться константой, даже если следующая буква заглавная.
- В линейной последовательности вложенных предложений if применяется ключевое слово elsif, а не else if или elif, как в некоторых других языках.
- Ключевые слова в Ruby нельзя назвать по-настоящему зарезервированными. Если метод вызывается от имени некоторого объекта (и в других случаях, когда не возникает неоднозначности), имя метода может совпадать с ключевым словом. Но поступайте так с осторожностью, не забывая, что программу будут читать люди.
- Ключевое слово then (в предложениях if и case) необязательно. Если вам кажется, что с ним программа понятнее, включайте. То же относится к слову do в циклах while и until.
- Вопросительный и восклицательный знаки не являются частью идентификатора, который модифицируют, их следует рассматривать как суффиксы. Таким образом, хотя идентификаторы chop и chop! считаются различными, использовать восклицательный знак в любом другом месте имени не разрешается. Аналогично, в Ruby есть конструкция defined?, но defined – ключевое слово.
- Внутри строки символ решетки – признак начала выражения. Значит, в некоторых случаях его следует экранировать обратной косой чертой, но лишь тогда, когда сразу за ним идет символ {, $ или @.
- Поскольку вопросительный знак можно добавлять в конец идентификатора, то следует аккуратно расставлять пробелы в тернарном операторе. Пусть, например, имеется переменная my_flag, которая может принимать значения true или false. Тогда первое из следующих предложений правильно, а второе содержит синтаксическую ошибку:

```
x = my_flag ? 23 : 45    # правильно
x = my_flag? 23 : 45     # синтаксическая ошибка
```

- Концевой маркер для встроенной документации не следует считать лексемой. Он помечает строку целиком, поэтому все находящиеся в той же строке символы не являются частью текста программы, а принадлежат встроенному документу.
- В Ruby нет произвольных блоков, то есть нельзя начать блок в любом месте, как в C. Блоки разрешены только там, где они нужны, например, могут

присоединяться к итератору. Исключение составляет блок begin-end, который можно употреблять практически везде.

- Не забывайте, что ключевые слова BEGIN и END не имеют ничего общего с begin и end.
- При статической конкатенации строк приоритет конкатенации ниже, чем у вызова метода. Например:

```
str = "Первая " 'second'.center(20)      # Примеры 1 and 2
str = "Вторая " + 'second'.center(20)    # дают одно и то же.
str = "Первая вторая".center(20)         # Примеры 3 and 4
str = ("Первая " + 'вторая').center(20) # дают одно и то же.
```

- В Ruby есть несколько псевдопеременных, которые выглядят как локальные переменные, но применяются для специальных целей. Это self, nil, true, false, __FILE__ и __LINE__.

1.5.2. Отличия от других языков

Наверное, каждый, кто знает Ruby (сегодня), в прошлом изучал или пользовался другими языками. Это, с одной стороны, облегчает изучение Ruby, так как многие средства похожи на аналогичные средства в других языках. С другой стороны, у программиста может возникнуть ложное чувство уверенности при взгляде на знакомые конструкции Ruby. Он может прийти к неверным выводам, основанным на прошлом опыте; можно назвать это явление «багажом эксперта».

Многие переходят на Ruby из Smalltalk, Perl, C/C++ и других языков. Их предположения и ожидания сильно различаются, но так или иначе присутствуют. Поэтому рассмотрим некоторые вещи, на которых многие спотыкаются.

- Не существует булева типа. TrueClass и FalseClass – это два разных класса, а единственными их экземплярами являются объекты true и false.
- Многие операторы в Ruby напоминают операторы в языке C. Два заметных исключения – операторы инкремента и декремента (++ и --). Их в Ruby нет – ни в «пост», ни в «пред» форме.
- Известно, что в разных языках оператор деления по модулю работает поразному для отрицательных чисел. Не вдаваясь в споры о том, что правильно, проиллюстрируем поведение в Ruby:

```
puts (5 % 3)    # Печатается 2
puts (-5 % 3)   # Печатается 1
puts (5 % -3)   # Печатается -1
puts (-5 % -3)  # Печатается -2
```

- Некоторые привыкли думать, что «ложь» можно представлять нулем, пустой строкой, нулевым символом и т.п. Но в Ruby все это равно «истине». На самом деле, истиной будет все, кроме объектов false и nil.
- В Ruby переменные не принадлежат никакому классу, класс есть только у значений.
- Переменные в Ruby не объявляются. Но считается хорошим тоном присваивать переменной начальное значение nil. Разумеется, при этом с перемен-

ной не ассоциируется никакой тип и даже не происходит истинной инициализации, но анализатор знает, что данное имя принадлежит переменной, а не методу.

- `ARGV[0]` – первый аргумент в командной строке, они нумеруются, начиная с нуля. Это не имя файла или скрипта, предшествующего параметрам, как `argv[0]` в языке C.
- Большинство операторов в Ruby на самом деле являются методами; их запись в виде «знаков препинания» – не более чем удобство. Первое исключение из этого правила – набор операторов составного присваивания (`+=`, `-=` и т.д.). Второе исключение – операторы `= .. ... ! not && and || or != !~`.
- Как и в большинстве современных языков (хотя и не во всех), булевы операции закорачиваются, то есть вычисление булева выражения заканчивается, как только его значение истинности становится известным. В последовательности операций `or` вычисление заканчивается, когда получено первое значение `true`, а в последовательности операций `and` – когда получено первое значение `false`.
- Префикс `@@` применяется для переменных класса (то есть ассоциированных с классом в целом, а не с отдельным экземпляром).
- `loop` – не ключевое слово. Это метод модуля `Kernel`, а не управляющая конструкция.
- Кому-то синтаксис `unless-else` может показаться интуитивно неочевидным. Поскольку `unless` – противоположность `if`, то ветвь `else` выполняется, когда условие истинно.
- Простой тип `Fixnum` передается как непосредственное значение и, стало быть, не может быть изменен внутри метода. То же относится к значениям `true`, `false`, и `nil`.
- Не путайте операторы `&&` и `||` с операторами `&` и `|`. Те и другие используются в языке C, первые два предназначены для логических операций, последние два – для поразрядных.
- Операторы `and` и `or` имеют более низкий приоритет, чем `&&` и `||`. Взгляните на следующий фрагмент:

```
a = true
b = false
c = true
d = true
a1 = a && b or c && d   # операции && выполняются первыми
a2 = a && (b or c) && d # операция or выполняется первой
puts a1                 # печатается false
puts a2                 # печатается true
```

- Не забывайте, что «оператор» присваивания имеет *более высокий* приоритет, чем операторы `and` и `or` (это относится и к составным операторам присваивания `+=`, `-=` и прочим). Например, код `x = y or z` выглядит как обычное предложение присваивания, но на самом деле это обособленное выраже-

ние (эквивалент (x=y) or z). Вероятно, программист имел в виду следующее: x = (y or z).

```
y = false
z = true

x = y or z    # = выполняется РАНЬШЕ or!
puts x        # печатается false

(x = y) or z  # строка 5: то же, что и выше
puts x        # печатается false

x = (y or z)  # or вычисляется сначала
puts x        # печатается true
```

- Не путайте атрибуты объектов с локальными переменными. Если вы привыкли к C++ или Java, то можете забыть об этом. Переменная @my_var в контексте класса – это переменная экземпляра (или атрибут), но my_var в том же контексте – локальная переменная.
- Во многих языках, и в Ruby в том числе, есть цикл for. Рано или поздно возникает вопрос, можно ли модифицировать индексную переменную. В некоторых языках эту управляющую переменную запрещено изменять вовсе (печатается предупреждение либо сообщение об ошибке на этапе компиляции или выполнения), в других это допустимо, хотя и приводит к изменению поведения цикла. В Ruby принят третий подход. Переменная, управляющая циклом for, считается обычной переменной, которую можно изменять в любой момент, но это изменение не оказывает влияния на поведение цикла! Цикл for присваивает этой переменной последовательные значения, что бы ни происходило с ней внутри тела цикла. Например, следующий цикл будет выполнен ровно 10 раз и напечатает значения от 1 до 10:

```
for var in 1..10
  puts "var = #{var}"
  if var > 5
    var = var + 2
  end
end
```

- Имена переменных не всегда легко «на глаз» отличить от имен методов. Как решает этот вопрос анализатор? Правило такое: если анализатор видит, что идентификатору присваивается значение до его использования, то он считается переменной, в противном случае это имя метода. (Отметим, что операция присваивания может и не выполняться, достаточно того, что интерпретатор ее видел.)

1.5.3. Предложение case в Ruby

Во всех современных языках есть та или иная форма многопутевого ветвления. В C/C++ и Java это предложение switch, а в Pascal – предложение case. Служат они одной и той же цели и функционируют примерно одинаково.

Предложение case в Ruby похоже, но при ближайшем рассмотрении оказывается настолько уникальным, что варианты ветвления, принятые в C и в Pascal, кажутся близкими родственниками. Точного аналога предложению case в Ruby нет ни в каком другом знакомом мне языке, поэтому оно заслуживает особого внимания.

Выше мы уже рассматривали синтаксис этого предложения, а теперь сосредоточимся на его семантике.

- Для начала рассмотрим тривиальный пример. Выражение expression сравнивается со значением value, и, если они совпадают, выполняется некоторое действие. Ничего удивительного.

```
case expression
  when value
    некоторое действие
end
```

  В Ruby для этой цели есть специальный оператор === (называется *оператором отношения*). Иногда его называют еще (не совсем правильно) *оператором ветвящегося равенства*. Неправильность в том, что он не всегда относится именно к сравнению на равенство.
- Предыдущее предложение можно записать и так:

```
if value === expression
  некоторое действие
end
```

- Но не путайте оператор отношения с оператором сравнения на равенство (==). Они принципиально различны, хотя во многих случаях ведут себя одинаково. Оператор отношения определен по-разному в разных классах, а для данного класса его поведение может зависеть от типа переданного операнда.
- Не думайте, что проверяемое выражение – это объект, которому сравниваемое значение передается в качестве параметра. На самом деле как раз наоборот (мы это только что видели).
- Это подводит нас к наблюдению, что x === y обычно вовсе не то же самое, что y === x! Иногда результат совпадает, но в общем случае оператор отношения не коммутативен. (Именно поэтому нам не нравится термин *оператор ветвящегося равенства* – ведь сравнение на равенство всегда коммутативно.) Другими словами, если перевернуть исходный пример, то окажется, что следующий код ведет себя иначе:

```
case value
  when expression
    некоторое действие
end
```

- В качестве примера рассмотрим строку str и образец (регулярное выражение) pat, с которым эта строка успешно сопоставляется. Выражение str =~ pat истинно, как в языке Perl. Поскольку Ruby определяет противоположную семантику оператора =~ в классе Regexp, то можно также сказать, что вы-

ражение pat =~ str истинно. Следуя той же логике, мы обнаруживаем, что pat === str также истинно (вследствие того, как определен оператор === в классе Regexp). Однако выражение str === pat истинным *не* является. А значит, фрагмент

```
case "Hello"
  when /Hell/
    puts "Есть соответствие."
else
  puts "Нет соответствия."
end
```

делает не то же самое, что фрагмент

```
case /Hell/
  when "Hello"
    puts "Есть соответствие."
else
  puts "Нет соответствия."
end
```

Если это вас смущает, просто постарайтесь запомнить. А если не смущает, тем лучше.

- Программисты, привыкшие к C, могут быть озадачены отсутствием предложений break в ветвях case. Такое использование break в Ruby необязательно (и недопустимо). Связано это с тем, что «проваливание» редко бывает желательно при многопутевом ветвлении. В конце каждой ветви when имеется неявный переход на конец предложения case. В этом отношении Ruby напоминает Pascal.
- Значения в каждой ветви case могут быть произвольны. На типы никаких ограничений не налагается. Они не обязаны быть константами, допускаются и переменные, и сложные выражения. Кроме того, в ветви может проверяться попадание в диапазон.
- В ветвях case могут находиться пустые действия (пустые предложения). Значения в разных ветвях не обязательно должны быть уникальными, допускаются перекрытия, например:

```
case x
  when 0
  when 1..5
    puts "Вторая ветвь"
  when 5..10
    puts "Третья ветвь"
else
  puts "Четвертая ветвь"
end
```

Если x принимает значение 0, ничего не делается. Для значения 5 печатается строка «Вторая ветвь» несмотря на то, что 5 удовлетворяет также условию в третьей ветви.

- Перекрытие ветвей допускается потому, что они вычисляются в строгом порядке и выполняется закорачивание. Иными словами, если вычисление выражения в какой-то ветви оказалось успешным, то следующие ветви не вычисляются. Поэтому не стоит помещать в ветви case выражения, в которых вызываются методы с побочными эффектами. (Впрочем, такие вызовы в любом случае сомнительны). Имейте также в виду, что такое поведение может замаскировать ошибки, которые произошли бы во время выполнения, если бы выражение вычислялось. Например:

```
case x
  when 1..10
    puts "Первая ветвь"
  when foobar()               # Возможен побочный эффект?
    puts "Вторая ветвь"
  when 5/0                    # Деление на нуль!
    puts "Третья ветвь"
else
  puts "Четвертая ветвь"
end
```

  Если x находится в диапазоне от 1 до 10, то метод foobar() не вызывается, а выражение 5 / 0 (которое, естественно, привело бы к ошибке) не вычисляется.

1.5.4. Рубизмы и идиомы

Материал в этом разделе во многом пересекается с изложенным выше. Но не задумывайтесь особо, почему мы решили разбить его именно таким образом. Просто многие вещи трудно точно классифицировать и организовать единственно правильным образом. Мы ставили себе задачу представить информацию в удобном для усвоения виде.

Ruby проектировался как непротиворечивый и ортогональный язык. Но вместе с тем это сложный язык, в котором есть свои идиомы и странности. Некоторые из них мы обсудим ниже.

- С помощью ключевого слова alias можно давать глобальным переменным и методам альтернативные имена (синонимы).
- Пронумерованные глобальные переменные $1, $2, $3 и т.д. не могут иметь синонимов.
- Мы не рекомендуем использовать «специальные переменные» $=, $_, $/ и им подобные. Иногда они позволяют написать более компактный код, но при этом он не становится более понятным. Поэтому в этой книге мы прибегаем к ним очень редко, что и вам рекомендуем. При необходимости можно использовать длинные и более понятные синонимы этих переменных, например $LAST_READ_LINE или $PROCESS_ID; для этого нужно добавить в код предложение require 'English'.
- Не путайте операторы диапазона .. и ... . Первый *включает* верхнюю границу, второй *исключает*. Так, диапазон 5..10 включает число 10, а диапазон 5...10 – нет.

- С диапазонами связана одна мелкая деталь, которая может вызвать путаницу. Если дан диапазон `m..n`, то метод `end` вернет конечную его точку `n`, равно как и его синоним `last`. Но те же методы возвращают значение `n` и для диапазона `m...n`, хотя `n` и не включается в него. Чтобы различить эти две ситуации, предоставляется метод `end_excluded?`.
- Не путайте диапазоны с массивами. Следующие два присваивания абсолютно различны:

  ```
  x = 1..5
  x = [1, 2, 3, 4, 5]
  ```

  Однако есть удобный метод `to_a` для преобразования диапазона в массив. (Во многих других типах тоже есть такой метод.)
- Часто бывает необходимо присвоить переменной значение лишь в том случае, когда у нее еще нет никакого значения. Поскольку «неприсвоенная» переменная имеет значение `nil`, можно решить эту задачу так: `x = x || 5` или сокращенно `x ||= 5`. Имейте в виду, что значение `false`, а равно и `nil`, будет при этом перезаписано.
- В большинстве языков для обмена значений двух переменных нужна дополнительная временная переменная. В Ruby наличие механизма множественного присваивания делает ее излишней: выражение `x, y = y, x` обменивает значения `x` и `y`.
- Четко отличайте класс от экземпляра. Например, у переменной класса `@@foobar` областью видимости является весь класс, а переменная экземпляра `@foobar` заново создается в каждом объекте класса.
- Аналогично, метод класса ассоциирован с тем классом, в котором определен; он не принадлежит никакому конкретному объекту и не может вызываться от имени объекта. При вызове метода класса указывается имя класса, а при вызове метода экземпляра – имя объекта.
- В публикациях, посвященных Ruby, часто для обозначения метода экземпляра применяют *решеточную нотацию*. Например, мы пишем `File.chmod`, чтобы обозначить метод `chmod` класса `File`, и `File#chmod` для обозначения метода экземпляра с таким же именем. Эта нотация не является частью синтаксиса Ruby. Мы старались не пользоваться ей в этой книге.
- В Ruby константы не являются истинно неизменными. Их нельзя изменять в теле методов экземпляра, но из других мест это вполне возможно.
- Ключевое слово `yield` пришло из языка CLU и некоторым программистам может быть непонятно. Оно используется внутри итератора, чтобы передать управление блоку, с которым итератор был вызван. В данном случае yield не означает, что нужно получить результат или вернуть значение. Скорее, речь идет о том, чтобы уступить процессор для работы.
- Составные операторы присваивания `+=`, `-=` и прочие – это не методы (собственно, это даже не операторы). Это всего лишь «синтаксическая глазурь» или сокращенная форма записи более длинной формы. Поэтому `x += y` значит в точности то же самое, что `x = x + y`. Если оператор `+` перегружен, то оператор `+=` «автомагически» учитывает новую семантику.

- Из-за того, как определены составные операторы присваивания, их нельзя использовать для инициализации переменных. Если первое обращение к переменной `x` выглядит как `x += 1`, возникнет ошибка. Это интуитивно очевидно для программистов, если только они не привыкли к языку, в котором переменные автоматически инициализируются нулем или пустым значением.
- Такое поведение можно в некотором смысле обойти. Можно определить операторы для объекта `nil`, так что в случае, когда начальное значение переменной равно `nil`, мы получим желаемый результат. Так, метод `nil.+`, приведенный ниже, позволит инициализировать объект типа `String` или `Fixnum`, для чего достаточно вернуть аргумент `other`. Таким образом, `nil + other` будет равно `other`.

```
def nil.+(other)
  other
end
```

  Мы привели этот код для иллюстрации возможностей Ruby, но стоит ли поступать так на практике, оставляем на усмотрение читателя.
- Уместно будет напомнить, что `Class` – это объект, а `Object` – это класс. Мы попытаемся прояснить этот вопрос в следующей главе, а пока просто повторяйте это как мантру.
- Некоторые операторы нельзя перегружать, потому что они встроены в сам язык, а не реализованы в виде методов. К таковым относятся `= .. ... and or not && || ! != !~`. Кроме того, нельзя перегружать составные операторы присваивания (`+=`, `-=` и т.д.). Это не методы и, пожалуй, даже не вполне операторы.
- Имейте в виду, что хотя оператор присваивания перегружать нельзя, тем не менее возможно написать метод экземпляра с именем `foo=` (тогда станет допустимым предложение `x.foo = 5`). Можете рассматривать знак равенства как суффикс.
- Напомним, что «голый» оператор разрешения области видимости подразумевает наличие `Object` перед собой, то есть `::Foo` – то же самое, что `Object::Foo`.
- Напомним, что `fail` – синоним `raise`.
- Напомним, что определения в Ruby исполняются. Вследствие динамической природы языка можно, например, определить два метода совершенно по-разному в зависимости от значения признака, проверяемого во время выполнения.
- Напомним, что конструкция `for` (`for x in a`) на самом деле вызывает итератор `each`. Любой класс, в котором такой итератор определен, можно обходить в цикле `for`.
- Не забывайте, что метод, определенный на верхнем уровне, добавляется в модуль `Kernel` и, следовательно, становится членом класса `Object`.
- Методы *установки* (например, `foo=`) должны вызываться от имени объекта, иначе анализатор решит, что речь идет о присваивании переменной с таким именем.

- Ключевое слово retry используется только для обработки исключений (в прежних версиях Ruby его можно было использовать также в итераторах).
- Метод объекта initialize всегда является закрытым.
- Когда блок заканчивается левой фигурной скобкой (или словом end) и возвращает значение, это значение можно использовать для вызова последующих методов, например:

```
squares = [1,2,3,4,5].collect do |x| x**2 end.reverse
# squares теперь равно [25,16,9,4,1]
```

- В конце программы на Ruby часто можно встретить идиому if $0 == __FILE__. Таким образом проверяется, исполняется ли файл как автономный кусок кода (true) или как дополнительный, например, библиотека (false). Типичное применение – поместить некую «главную программу» (обычно с тестовым кодом) в конец библиотеки.
- Обычное наследование (порождение подкласса) обозначается символом <:

```
class Dog < Animal
  # ...
end
```

Однако для создания синглетного класса (анонимного класса, который расширяет единственный экземпляр) применяется символ <<:

```
class << platypus
  # ...
end
```

- При передаче блока итератору есть тонкое различие между фигурными скобками ({}) и операторными скобками do-end. Связано оно с приоритетом:

```
mymethod param1, foobar do ... end
# Здесь do-end связано с mymethod

mymethod param1, foobar { ... }
# А здесь {} связано с именем foobar, предполагается, что это метод
```

- Традиционно в Ruby однострочные блоки заключают в фигурные скобки, а многострочные – в скобки do-end, например:

```
my_array.each { |x| puts x }

my_array.each do |x|
  print x
  if x % 2 == 0
    puts " четно."
  else
    puts " нечетно."
  end
end
```

Это необязательно и в некоторых случаях даже нежелательно.

- Замыкание (closure) запоминает контекст, в котором было создано. Один из способов создать замыкание – воспользоваться объектом Proc. Например:

```
def power(exponent)
  proc {|base| base**exponent}
end

square = power(2)
cube = power(3)

a = square.call(11)    # Результат равен 121
b = square.call(5)     # Результат равен is 25
c = cube.call(6)       # Результат равен is 216
d = cube.call(8)       # Результат равен is 512
```

Обратите внимание, что замыкание «знает» значение показателя степени, переданное ему в момент создания.

- Не забывайте однако, что в замыкании используется переменная, определенная во внешней области видимости (что вполне допустимо). Это свойство может оказаться полезным, однако приведем пример неправильного использования:

```
$exponent = 0

def power
  proc {|base| base**$exponent}
end

$exponent = 2
square = power

$exponent = 3
cube = power

a = square.call(11)    # Неверно! Результат равен 1331

b = square.call(5)     # Неверно! Результат равен 125

# Оба результата неверны, поскольку используется ТЕКУЩЕЕ
# значение $exponent. Так было бы даже в том случае, когда
# используется локальная переменная, покинувшая область
# видимости (например, с помощью define_method).

c = cube.call(6)   # Результат равен 216
d = cube.call(8)   # Результат равен 512
```

- Напоследок рассмотрим несколько искусственный пример. Внутри блока итератора times создается новый контекст, так что x – локальная переменная. Переменная closure уже определена на верхнем уровне, поэтому для блока она не будет локальной.

```
closure = nil          # Определим замыкание, чтобы его имя было известно
1.times do             # Создаем новый контекст
  x = 5                # x локальная в этом блоке
  closure = Proc.new { puts "В замыкании, x = #{x}" }
end

x = 1

# Определяем x на верхнем уровне

closure.call           # Печатается: В замыкании, x = 5
```

Обратите внимание, что переменная `x`, которой присвоено значение 1, – это новая переменная, определенная на верхнем уровне. Она не совпадает с одноименной переменной, определенной внутри блока. Замыкание печатает 5, так как запоминает контекст своего создания, в котором была определена переменная `x` со значением 5.

- Переменные с именами, начинающимися с одного символа `@`, определенные внутри класса, – это, вообще говоря, переменные экземпляра. Однако, если они определены вне любого метода, то становятся переменными экземпляра класса. (Это несколько противоречит общепринятой терминологии ООП, в которой «экземпляр класса» – то же самое, что просто «экземпляр» или «объект».) Пример:

```
class Myclass

  @x = 1       # Переменная экземпляра класса
  @y = 2       # Еще одна

  def mymethod
    @x = 3     # Переменная экземпляра
    # Заметим, что в этой точке @y недоступна.
  end

end
```

Переменная экземпляра класса `@y` в предыдущем примере – в действительности атрибут объекта класса `Myclass`, являющегося экземпляром класса `Class`. (Напомним, что `Class` – это объект, а `Object` – это класс.) На переменные экземпляра класса нельзя ссылаться из методов экземпляра и, вообще говоря, они не очень полезны.

- `attr`, `attr_reader`, `attr_writer` и `attr_accessor` – сокращенная запись для определения методов чтения и установки атрибутов. В качестве аргументов они принимают символы (экземпляры класса `Symbol`).
- Присваивание переменной, имя которой содержит оператор разрешения области видимости недопустимо. Например, `Math::PI = 3.2` – ошибка.

1.5.5. Ориентация на выражения и прочие вопросы

В Ruby выражения важны почти так же, как предложения. Для программиста на C это звучит знакомо, а для программиста на Pascal – откровенная нелепость. Но Ruby ориентирован на выражения даже в большей степени, чем C.

Заодно в этом разделе мы остановимся на паре мелких вопросов, касающихся регулярных выражений, считайте это небольшим бонусом.

- В Ruby любое присваивание возвращает то же значение, которое стоит в правой части. Поэтому иногда мы можем немного сократить код, как показано ниже, но будьте осторожны, имея дело с объектами! Не забывайте, что это почти всегда ссылки.

```
x = y = z = 0       # Все переменные сейчас равны 0.

a = b = c = []      # Опасно! a, b и c ссылаются на ОДИН И
                    # ТОТ ЖЕ пустой массив.
x = 5
y = x += 2          # Сейчас x и y равны 7
```

  Напомним однако, что значения типа `Fixnum` и им подобные хранятся непосредственно, а не как ссылки на объекты.

- Многие управляющие конструкции возвращают значения, в частности, `if`, `unless` и `case`. Следующий код корректен, он показывает, что при принятии решения ветви могут быть выражениями, а не полноценными предложениями.

```
a = 5
x = if a < 8 then 6 else 7 end    # x равно 6

y = if a < 8        # y тоже равно 6;
  6                 # предложение if может располагаться
else                # на одной строке
  7                 # или на нескольких.
end

# unless тоже работает; z присваивается значение 4
z = unless x == y then 3 else 4 end

t = case a          # t получает
  when 0..3         # значение
    "low"           # "medium"
  when 4..6
    "medium"
else
  "high"
end
```

  Здесь мы сделали такие отступы, будто `case` является присваиванием. Мы воспринимаем такую запись спокойно, хотя вам она может не понравиться.

- Отметим, что циклы `while` и `until` напротив не возвращают никаких полезных значений, обычно их значением является `nil`:

```
i = 0
x = while (i < 5)    # x равно nil
  puts i+=1
end
```

- Тернарный оператор можно использовать как в предложениях, так и в выражениях. В силу синтаксических причин (или ограничений анализатора) скобки здесь обязательны:

```
x = 6
y = x == 5 ? 0 : 1                          # y равно 1
x == 5 ? puts("Привет") : puts("Пока")      # Печатается Пока
```

- Предложение `return` в конце метода можно опускать. Метод всегда возвращает значение последнего вычисленного выражения, в каком бы месте это вычисление ни происходило.
- Когда итератор вызывается с блоком, последнее выражение, вычисленное в блоке, возвращается в качестве значения блока. Если при этом в теле итератора есть предложение `x = yield`, то `x` будет присвоено это значение.
- Напомним, что после регулярного выражения можно написать модификатор многострочности `/m`, и в этом случае точка (.) будет сопоставляться с символом новой строки.
- Опасайтесь соответствий нулевой длины в регулярных выражениях. Если все элементы регулярного выражения необязательны, то такому образцу будет соответствовать «ничто», причем соответствие всегда будет найдено в начале строки. Это типичная ошибка, особенно часто ее допускают начинающие.

1.6. Жаргон Ruby

Заново начинать учить английский для освоения Ruby необязательно. Но нужно знать кое-какие жаргонные выражения, обычные в сообществе. Некоторые из них имеют другой смысл, чем принято в компьютерном мире. Им и посвящен настоящий раздел.

В Ruby термин *атрибут* носит неофициальный характер. Можно считать, что атрибут — это переменная экземпляра, которая раскрывается внешнему миру с помощью одного из методов семейства `attr`. Но тут нет полной определенности, могут существовать методы `foo` и `foo=`, не соответствующие переменной `@foo`, как можно было бы ожидать. И, конечно, не все переменные экземпляра считаются атрибутами. Как обычно, нужно придерживаться здравого смысла.

Атрибуты в Ruby можно подразделить на методы *чтения* (reader) и *установки* (writer). *Метод доступа* или *акцессор* (accessor) является одновременно методом чтения и установки. Это согласуется с названием метода `attr_accessor`, но противоречит принятой в других сообществах семантике, согласно которой акцессор дает доступ только для чтения.

Оператор `===` имеется только в Ruby (насколько мне известно). Обыкновенно он называется *оператором ветвящегося равенства* (case equality operator), поскольку неявно используется в предложениях `case`. Но это название, как я уже говорил, не вполне точно, потому что речь идет не только о «равенстве». В этой книге я часто употребляю термин *оператор отношения* (relationship operator). Изобрел

его не я, но проследить происхождение мне не удалось, к тому же он употребляется не часто. Жаргонное название — *«оператор тройного равенства»* (threequal operator) или просто «три равно».

Оператор <=>, наверное, лучше всего называть *оператором сравнения*. На жаргоне его называют *космическим оператором* (spaceship operator), поскольку он напоминает вид сбоку на летающую тарелку, как ее изображали в старых видеоиграх.

Термин *поэтический режим* (poetry mode) подчеркивает, что можно опускать ненужные знаки препинания и лексемы (насмешливый намек на отношение поэтов к пунктуации на протяжении последних шестидесяти лет). Поэтический режим также часто означает «опускание скобок при вызове метода».

```
some_method(1, 2, 3)      # избыточные скобки
some_method 1, 2, 3       # "поэтический режим"
```

Но мне этот принцип представляется более общим. Например, когда хэш передается в качестве последнего или единственного параметра, можно опускать фигурные скобки. В конце строки можно не ставить точку с запятой (и таки никто этого не делает). В большинстве случае можно опускать ключевое слово `then` в предложениях `if` и `case`.

Некоторые программисты заходят еще дальше, опуская скобки даже в определении методов, но большинство так не поступает:

```
def my_method(a, b, c)  # Можно и так:  def my_method a, b, c
  # ...
end
```

Стоит отметить, что в некоторых случаях сложность грамматики Ruby приводит к сбоям анализатора. Во вложенных вызовах методов скобки для ясности лучше оставлять.

```
def alpha(x)
  x*2
end

def beta(y)
  y*3
end

gamma = 5
delta = alpha beta gamma  # 30 -- то же, что alpha(beta(gamma))
```

Термин ***duck typing*** (динамическая, или ***утиная*** типизация), насколько я знаю, принадлежит Дэйву Томасу (Dave Thomas). Он восходит к старой поговорке: «ес-

дать объект методу, зная, что при неправильном использовании будет возбуждено исключение. Так оно рано или поздно и случается, но возбуждаемые при этом исключения малопонятны, а отлаживать их трудно.

Унарную звездочку, которая служит для расширения массива, можно было бы назвать оператором *расширения массива*, но не думаю, что кто-нибудь слышал такое выражение. В неформальной беседе часто употребляют слова *звездочка* (star) и *разглаживание* (splat) и, естественно, прилагательные, *разглаженный* (splatted) и *сморщенный* (unsplatted). Дэвид Алан Блэк, автор книги «The Well-Grounded Rubyist», придумал остроумное название *унарный оператор линеаризации* (unary unarray operator)[1].

Термин *синглет* (singleton) многие считают перегруженным. Это вполне обычное английское слово, означающее вещь, существующую в единственном экземпляре. Пока мы используем его в качестве модификатора, никакой путаницы не возникает.

Но *Singleton* (Одиночка) – это еще и хорошо известный паттерн проектирования, относящийся к классу, для которого может существовать лишь один объект. В Ruby для реализации этого паттерна имеется библиотека `singleton`.

Синглетный класс (singleton class, некоторые говорят также *eigenclass*) в Ruby – это подобная классу сущность, методы которой хранятся на уровне объекта, а не класса. Пожалуй, это не «настоящий класс», потому что его нельзя инстанцировать. Ниже приведен пример открытия синглетного класса для строкового объекта с последующим добавлением метода:

```
str = "hello"
class << str                    # Альтернатива:
  def hyphenated                # def str.hyphenated
    self.split("").join("-")
  end
end

str.hyphenated                  # "h-e-l-l-o"
```

Вернемся к предыдущему примеру. Поскольку метод `hyphenate` не существует ни в каком-либо другом объекте, ни в классе, то это *синглетный метод* данного объекта. Это не вызывает неоднозначности. Иногда сам объект называется синглетным, поскольку он единственный в своем роде – больше ни у кого такого метода нет.

Однако вспомним, что в Ruby сам класс является объектом. Поэтому мы можем добавить метод в синглетный класс *класса*, и этот метод будет уникален для объекта, который – по чистой случайности – оказался классом. Пример:

```
class MyClass
  class << self                 # Альтернатива: def self.hello
```

---

[1] Прошу читателя извинить меня за это нагромождение выдуманных слов, но жаргон вряд ли следует переводить академически правильным языком. Хочется надеяться, что это все же лучше, чем кальки английских слов. – *Прим. перев.*

```
    def hello                    # или: def MyClass.hello
      puts "Привет от #{self}!"
    end
  end
end
```

Поэтому необязательно создавать объект класса `MyClass` для вызова этого метода.

```
MyClass.hello                    # Привет от MyClass!
```

Впрочем, вы, наверное, заметили, что это не что иное, как метод класса в Ruby. Иными словами, метод класса – это синглетный метод объекта-класса. Можно также сказать, что синглетный метод, определенный для объекта, который *случайно оказался* классом.

Осталась еще парочка терминов. *Переменная класса* – это, разумеется, то, имя чего начинается с двух символов `@`. Возможно, название неудачно из-за нетривиального поведения относительно наследования. *Переменная экземпляра класса* – нечто совсем иное. Это обычная переменная экземпляра, только объект, которому она принадлежит, является классом. Дополнительную информацию по этой теме см. в главе 11.

Выход на авансцену Ruby on Rails сопровождался появления словечка «партизанское латание» (monkeypatching); имеется в виду открытие класса (особенно системного) с целью добавить в него новые методы (либо модифицировать имеющиеся) или другие элементы. Я не пользуюсь этим термином ни в этой книге, ни где-либо еще, потому что он привнесен в сообщество извне и имеет пренебрежительный оттенок. Модификация классов в Ruby – не ошибка, а осознанно предоставленный механизм. Им, как и любым другим языковым средством, можно пользоваться правильно и безопасно, а можно и ненадлежащим образом.

1.7. Заключение

На этом завершается наш обзор объектно-ориентированного программирования и краткая экскурсия по языку Ruby. В последующих главах изложенный материал будет раскрыт более полно.

Хотя в мои намерения не входило «учить Ruby» новичков, но не исключено, что даже начинающие программисты на Ruby почерпнут что-то полезное из этой главы. (Некоторые читатели говорили мне, что выучили Ruby по первому или второму изданию этой книги.) Как бы то ни было, последующие главы будут полезны «рубистам» начального и среднего уровня. Надеюсь, что даже опытные программисты на Ruby найдут для себя что-то новенькое.

Глава 2. Строки

Когда-то элементарными кирпичиками мироздания считались атомы, потом протоны, потом кварки. Теперь таковыми считаются струны[1].

Дэвид Гросс, профессор теоретической физики, Принстонский университет

В начале 1980-х годов один профессор информатики начал первую лекцию по структурам данных с вопроса. Он не представился студентам, не сказал, как называется курс, не рассказал о его программе и не порекомендовал никаких учебников. Вместо этого он встал перед аудиторией и спросил: «Какой тип данных самый важный?»

Было высказано несколько предположений. Когда профессор услышал слово «указатели», он выразил удовлетворение, но все-таки он не согласился со студентом, а высказал свое мнение: «Самым важным является тип символ».

У него были на то основания. Компьютерам предназначено быть нашими слугами, а не хозяевами, а человеку понятны только символьные данные. (Есть, конечно, люди, которые без труда читают и двоичные данные, но о них мы сейчас говорить не будем.) Символы, а, стало быть, и строки позволяют человеку общаться с компьютером. Любую информацию, в том числе и текст на естественном языке, можно закодировать в виде строк.

Строка — это просто последовательность символов. Подобно большинству сущностей в Ruby, строки являются полноценными объектами. В программах приходится выполнять разнообразные операции со строками: конкатенировать, выделять лексемы, анализировать, выполнять поиск и замену и т.д. Язык Ruby позволяет все это делать без труда.

На протяжении большей части истории Ruby считалось, что байт и символ — синонимы. Однако это не так для специальных символов, смайликов и нелатинских шрифтов. Подробнее вопрос о том, почему байты и символы — не всегда одно и то же, обсуждается в главе 4 «Интернационализация в Ruby».

2.1. Представление обычных строк

Строка в Ruby — это просто последовательность 8-битовых байтов. Она не завершается нулевым символом, как в C, и, следовательно, может содержать такие сим-

[1] В английском языке словом «string» обозначается как «строка», так и «струна». — *Прим. перев.*

волы. В строке могут быть символы с кодами больше 0xFF, но такие строки имеют смысл, лишь в кодировках, отличных от ASCII. Предполагается, что все строки представлены в кодировке UTF-8. До выхода версии Ruby 2.0 предполагалась кодировка ASCII (дополнительную информацию о кодировках см. в главе 4).

Простейшая строка в Ruby заключается в одиночные кавычки. Такие строки воспринимаются буквально; в качестве управляющих символов в них распознаются только экранированная одиночная кавычка (\') и экранированный символ обратной косой черты (\\):

```
s1 = 'Это строка'           # Это строка
s2 = 'Г-жа О\'Лири'         # Г-жа О'Лири
s3 = 'Смотри в C:\\TEMP'    # Смотри в C:\TEMP
```

Строки, заключенные в двойные кавычки, обладают большей гибкостью. В них допустимо много других управляющих последовательностей, в частности, для представления символов забоя, табуляции, возврата каретки и перевода строки. Можно также включать произвольные символы, представленные восьмеричными цифрами:

```
s1 = "Это знак табуляции: (\t)"
s2 = "Несколько символов забоя: xyz\b\b\b"
s3 = "Это тоже знак табуляции: \011"
s4 = "А это символы подачи звукового сигнала: \a \007"
s5 = "Это символ снеговика в unicode: \u2603"
```

Символы, не входящие в ASCII, при отображении строки «экранированы обратной косой чертой». Внутри строки, заключенной в двойные кавычки, могут встречаться выражения (см. раздел 2.21).

2.2. Альтернативная нотация для представления строк

Иногда встречаются строки, в которых много метасимволов, например, одиночных и двойных кавычек и т.д. В этом случае можно воспользоваться конструкциями %q и %Q. Вслед за ними должна идти строка, обрамленная с обеих сторон символами-ограничителями; лично я предпочитаю квадратные скобки ([ ]).

При этом %q ведет себя как одиночные кавычки, а %Q — как двойные.

```
S1 = %q[Как сказал Магритт, "Ceci n'est pas une pipe."]
s2 = %q[Это не табуляция: (\t)]     # то же, что: 'Это не табуляция: \t'
s3 = %Q[А это табуляция: (\t)]      # то же, что: "А это табуляция: \t"
```

В обоих вариантах можно применять и другие ограничители, помимо квадратных скобок: круглые, фигурные, угловые скобки:

```
s1 = %q(Билл сказал: "Боб сказал: 'This is a string.'")
s2 = %q{Другая строка.}
s3 = %q<В этой строке есть специальные символы '"[](){}.>
```

Допустимы также непарные ограничители. В этом качестве может выступать любой символ, кроме букв, цифр и пропусков (пробелов и им подобных), который имеет визуальное представление и не относится к числу перечисленных выше парных ограничителей.

```
s1 = %q:"Я думаю, что это сделала корова г-жи О'Лири," сказал он.:
s2 = %q*\r - это control-M, а \n - это control-J.*
```

2.3. Встроенные документы

Для представления длинной строки, занимающей несколько строк в тексте, можно, конечно, воспользоваться обычными строками в кавычках:

```
str = "Три девицы под окном
   Пряли поздно вечерком..."
```

Но тогда отступ окажется частью строки.

Можно вместо этого воспользоваться встроенным документом, изначально предназначенным для многострочных фрагментов. (Идея и сам термин заимствованы из более старых языков.) Синтаксически он начинается с двух знаков <<, за которыми следует концевой маркер, нуль или более строк текста и в завершение тот же самый концевой маркер в отдельной строке:

```
str = <<EOF
Три девицы под окном
Пряли поздно вечерком...
EOF
```

Но следите внимательно, чтобы после завершающего концевого маркера не было пробелов. В текущей версии Ruby маркер в такой ситуации не распознается.

Встроенные документы могут быть вложенными. В примере ниже показано, как передать методу три представленных таким образом строки:

```
some_method(<<str1, <<str2, <<str3)
первый кусок
текста...
str1
второй кусок...
str2
третий кусок
текста.
str3
```

По умолчанию встроенный документ ведет себя как строка в двойных кавычках, то есть внутри него интерпретируются управляющие последовательности и интерполируются выражения. Но если концевой маркер заключен в одиночные кавычки, то и весь документ ведет себя как строка в одиночных кавычках:

```
str = <<'EOF'
Это не знак табуляции: \t
а это не символ новой строки: \n
EOF
```

Если концевому маркеру встроенного документа предшествует дефис, то маркер может начинаться с красной строки. При этом удаляются только пробелы из той строки, на которой расположен сам маркер, но не из предшествующих ей строк документа.

```
str = <<-EOF
   Каждая из этих строк
   начинается с пары
   пробелов.
   EOF
```

Чтобы убрать символы в начале каждой строки, нам понадобится другой метод. В gem-пакете ActiveSupport (входит в состав Rails) определен метод strip_heredoc, который работает примерно так:

```
class String
   def strip_heredoc
      # Найти отступ в начале первой строчки
      margin = self[/\A\s*/]
      # Удалить столько пробелов в начале каждой строчки, чему
      # равен размер отступа
      gsub(/\s{#{margin.size}}/,"")
   end
end
```

Здесь определяется, сколько пробелов находится в начале первой строчки и ровно столько же удаляется из каждой последующей. Используется этот метод так:

```
str = <<end.strip_heredoc
   Этот встроенный документ имеет "левое поле",
   отбитое пробелами в первой строке.

   Мы можем включать цитаты,
   делать выступы и т.д.
end
```

В качестве концевого маркера естественно употребить слово end. (Впрочем, это дело вкуса. Выглядит такой маркер как зарезервированное слово end, но на самом деле это просто произвольно выбранная строка.) Во многих текстовых редакторах концевой маркер помогает подсветить синтаксис. Поэтому использование таких маркеров, как <<SQL или <<RUBY, значительно упрощает чтение блоков кода внутри встроенных документов.

2.4. Получение длины строки

Для получения длины строки служит метод length. У него есть синоним size.

```
str1 = "Карл"
x = str1.length  # 4
str2 = "Дойль"
x = str2.size  # 5
```

2.5. Построчная обработка

Строка в Ruby может содержать символы новой строки. Например, можно прочитать в память файл и сохранить его в виде одной строки. Применяемый по умолчанию итератор each в этом случае перебирает отдельные строки:

```
str = "Когда-то\nдавным-давно...\nКонец\n"
num = 0
str.each do |line|
   num += 1
   print "Строка #{num}: #{line}"
end
```

Выполнение этого кода дает следующий результат:

```
Строка 1: Когда-то
Строка 2: давным-давно...
Строка 3: Конец
```

Одни итераторы (например, each_line) можно сцеплять с другими (например, with_index). Технику соединения выхода одной функции с входом другой в одной строке иногда называют *композицией функций* (или *сцеплением методов*). Вместо того чтобы следить за номером строки самому, можно получить такой же результат, воспользовавшись *композицией* итераторов with_index и each_line:

```
str = "Когда-то\nдавным-давно...\nКонец\n"
str.each_line.with_index do |line, num|
   print "Строка #{num + 1}: #{line}"
end
```

2.6. Побайтовая обработка

Раньше в Ruby каждый байт считался символом, но теперь это уже не так. Байты строки можно получить в виде массива с помощью метода bytes. Для последовательной обработки байтов пользуйтесь итератором each_byte:

```
str = "ABC"
str.each_byte {|byte| print byte, " " }
puts
# Результат: 65 66 67
```

Символ — это, по существу, то же самое, что односимвольная строка. В многобайтовых кодировках односимвольная строка может содержать более одного байта:

```
str = "ABC"
str.each_char {|char| print char, " " }
puts
# Результат: A B C
```

В любой версии Ruby строку можно преобразовать в массив односимвольных строк с помощью метода scan, которому передается простое регулярное выражение, соответствующее одному символу:

```
str = "ABC"
chars = str.scan(/./)
chars.each {|char| print char, " " }
# Результат: A B C
```

2.7. Специализированное сравнение строк

В язык Ruby уже встроен механизм сравнения строк: строки сравниваются в привычном лексикографическом порядке (то есть на основе упорядочения, присущего данному набору символов). Но при желании можно задать собственные правила сравнения любой сложности.

Предположим, например, что мы хотим игнорировать английские артикли a, an и the, если они встречаются в начале строки, а также не обращать внимания на большинство знаков препинания. Для этого следует переопределить встроенный метод <=> (он вызывается из методов <, <=, > и >=). В листинге 2.1 показано, как это сделать.

Листинг 2.1. Специализированное сравнение строк

```
class String

   alias old_compare <=>

   def <=>(other)
      a = self.dup
      b = other.dup
      # Удалить знаки препинания
      a.gsub!(/[\,\.\?\!\:\;]/, "")
      b.gsub!(/[\,\.\?\!\:\;]/, "")
      # Удалить артикли из начала строки
      a.gsub!(/^(a |an |the )/i, "")
      b.gsub!(/^(a |an |the )/i, "")
      # Удалить начальные и хвостовые пробелы
      a.strip!
      b.strip!
      # Вызвать старый метод <=>
      a.old_compare(b)
   end
end

title1 = "Calling All Cars"
title2 = "The Call of the Wild"

# При стандартном сравнении было бы напечатано "yes"

if title1 < title2
   puts "yes"
else
   puts "no" # А теперь печатается "no"
end
```

Обратите внимание, что мы «сохранили» старый метод <=> с помощью ключевого слова alias и в конце вызвали его. Если бы мы вместо этого воспользовались

методом <, то был бы вызван новый метод <=>, что привело бы к бесконечной рекурсии и в конечном итоге к аварийному завершению программы.

Отметим также, что оператор == не вызывает метод <=> (принадлежащий классу-примеси Comparable). Это означает, что для специализированной проверки строк на равенство пришлось бы отдельно переопределить метод ==. Но в рассмотренном случае в этом нет необходимости.

Допустим, что мы хотим сравнивать строки без учета регистра. Для этого есть встроенный метод casecmp, надо лишь добиться, чтобы он вызывался при сравнении. Вот как это можно сделать:

```
class String
   def <=>(other)
      casecmp(other)
   end
end
```

Есть и более простой способ:

```
String
   alis <=> casecmp(other)
end
```

Но это не все. Надо еще переопределить оператор ==, чтобы он вел себя точно так же:

```
class String
   def ==(other)
      casecmp(other) == 0
   end
end
```

Теперь все строки будут сравниваться без учета регистра. И при всех операциях сортировки, которые определены в терминах метода <=>, регистр тоже не будет учитываться.

2.8. Разбиение строки на лексемы

Метод split разбивает строку на части и возвращает массив лексем. Ему передаются два параметра: разделитель и максимальное число полей (целое).

По умолчанию разделителем является пробел, а точнее – значение специальной переменной $; или ее англоязычного эквивалента $FIELD_SEPARATOR. Если же первым параметром задана некоторая строка, то она и будет использоваться в качестве разделителя лексем.

```
s1 = "Была темная грозовая ночь."
words = s1.split                    # ["Была", "темная", "грозовая", "ночь"]

s2 = "яблоки, груши, персики"
list = s2.split(", ")               # ["яблоки", "груши", "персики"]

s3 = "львы и тигры и медведи"
zoo = s3.split(/ and /)             # ["львы", "тигры", "медведи"]
```

Второй параметр ограничивает число возвращаемых полей, при этом действуют следующие правила:

- Если параметр опущен, то пустые поля в конце отбрасываются.
- Если параметр – положительное число, то будет возвращено не более указанного числа полей (если необходимо, весь «хвост» строки помещается в последнее поле). Пустые поля в конце сохраняются.
- Если параметр – отрицательное число, то количество возвращаемых полей не ограничено, а пустые поля в конце сохраняются.

Ниже приведены примеры:

```
str = "alpha,beta,gamma,,"
list1 = str.split(",")      # ["alpha","beta","gamma"]
list2 = str.split(",",2)    # ["alpha", "beta,gamma,,"]
list3 = str.split(",",4)    # ["alpha", "beta", "gamma", ","]
list4 = str.split(",",8)    # ["alpha", "beta", "gamma", "", ""]
list5 = str.split(",",-1)   # ["alpha", "beta", "gamma", "", ""]
```

Для сопоставления строки с регулярным выражением или с другой строкой служит метод scan:

```
str = "I am a leaf on the wind..."

# Строка интерпретируется буквально, а не как регулярное выражение
arr = str.scan("a")  # ["a","a","a"]

# При сопоставлении с регулярным выражением возвращаются все соответствия
arr = str.scan(/\w+/)
# ["I", "am", "a", "leaf", "on", "the", "wind"]

# Блоку будут по одному передаваться найденные соответствия
str.scan(/\w+/) {|x| puts x }
```

Класс StringScanner из стандартной библиотеки отличается тем, что сохраняет состояние сканирования, а не выполняет все за один раз:

```
require 'strscan'
str = "Смотри как я парю!"
ss = StringScanner.new(str)
loop do
   word = ss.scan(/\w+/)    # Получать по одному слову
   break if word.nil?
   puts word
   sep = ss.scan(/\W+/)     # Получить следующий фрагмент, не являющийся словом
   break if sep.nil?
end
```

2.9. Форматирование строк

В Ruby, как и в языке C, для этой цели предназначен метод sprintf. Он принимает строку и список выражений, а возвращает строку. Набор спецификаторов в фор-

матной строке мало чем отличается от принятого в функции sprintf (или printf) из библиотеки C.

```
name = "Боб"
age = 28
str = sprintf("Привет, %s... Похоже, тебе %d лет.", name, age)
```

Спрашивается, зачем нужен этот метод, если можно просто интерполировать значения в строку помощью конструкции #{expr}? А затем, что sprintf позволяет выполнить дополнительное форматирование, например, задать максимальную ширину поля или максимальное число цифр после запятой, добавить или подавить начальные нули, выровнять по левой или правой границе и т.д.

```
str = sprintf("%-20s %3d", name, age)
```

В классе String есть еще метод %, который делает почти то же самое. Он принимает одно значение или массив значений любых типов:

```
str = "%-20s %3d" % [name, age] # То же, что и выше
```

Имеются также методы ljust, rjust и center; они принимают длину результирующей строки и дополняют ее до указанной длины пробелами, если это необходимо.

```
str = "Моби Дик"
s1 = str.ljust(12)     # "Моби Дик"
s2 = str.center(12)    # "  Моби Дик  "
s3 = str.rjust(12)     # "    Моби Дик"
```

Можно задать и второй параметр, который интерпретируется как строка заполнения (при необходимости она будет усечена):

```
str = "Капитан Ахав"
s1 = str.ljust(20,"+")      # "Капитан Ахав++++++++"
s2 = str.center(20,"-")     # "—Капитан Ахав—"
s3 = str.rjust(20,"123")    # "12312312Капитан Ахав"
```

2.10. Строки в качестве объектов ввода-вывода

Помимо методов sprintf и scanf, есть еще один способ имитировать ввод-вывод в строку: класс StringIO.

Из-за сходства с объектом IO мы рассмотрим его в главе, посвященной вводу-выводу (см. раздел 10.1.24).

2.11. Управление регистром

В классе String есть множество методов для управления регистром. В этом разделе мы приведем их краткий обзор.

Метод downcase преобразует всю строку в нижний регистр, а метод upcase в верхний:

```
s1 = "Бостонское чаепитие"
s2 = s1.downcase                # "бостонское чаепитие"
s3 = s2.upcase                  # "БОСТОНСКОЕ ЧАЕПИТИЕ"
```

Метод capitalize преобразует в верхний регистр первый символ строки, а все остальные – в нижний:

```
s4 = s1.capitalize     # "Бостонское чаепитие"
s5 = s2.capitalize     # "Бостонское чаепитие"
s6 = s3.capitalize     # "Бостонское чаепитие"
```

Метод swapcase изменяет регистр каждой буквы на противоположный:

```
s7 = "ЭТО БЫВШИЙ попугай."
s8 = s7.swapcase                # "это бывший ПОПУГАЙ."
```

Начиная с версии 1.8, в язык Ruby включен метод casecmp, который работает аналогично стандартному методу <=>, но игнорирует регистр:

```
n1 = "abc".casecmp("xyz")   # -1
n2 = "abc".casecmp("XYZ")   # -1
n3 = "ABC".casecmp("xyz")   # -1
n4 = "ABC".casecmp("abc")   # 0
n5 = "xyz".casecmp("abc")   # 1
```

У каждого из перечисленных методов имеется аналог, осуществляющий модификацию «на месте» (upcase!, downcase!, capitalize!, swapcase!).

Не существует встроенных методов, позволяющих узнать регистр буквы, но это легко сделать с помощью регулярных выражений:

```
if string =~ /[a-z]/
   puts "строка содержит символы из нижнего регистра"
end

if string =~ /[A-Z]/
   puts "строка содержит символы из верхнего регистра"
end

if string =~ /[A-Z]/ and string =~ /a-z/
   puts "строка содержит символы из разных регистров"
end

if string[0..0] =~ /[A-Z]/
   puts "строка начинается с заглавной буквы"
end
```

Такие регулярные выражения сопоставляются только с символами ASCII. Для сопоставления с символами Unicode верхнего и нижнего регистра пользуйтесь именованными классами символов:

```
if string =~ /\p{Upper}/
   puts "строка содержит символы Unicode верхнего регистра, например Ü"
end
```

Дополнительные сведения о регулярных выражениях см. в главе 3.

2.12. Вычленение и замена подстрок

В Ruby к подстрокам можно обращаться разными способами. Обычно применяются квадратные скобки, как для массивов, но внутри скобок может находиться

пара объектов класса Fixnum, диапазон, регулярное выражение или строка. Ниже мы рассмотрим все варианты.

Если задана пара объектов класса Fixnum, то они трактуются как смещение от начала строки и длина, а возвращается соответствующая подстрока.

```
str = "Шалтай-Болтай"
sub1 = str[7,4]          # "Болт"
sub2 = str[7,99]             # "Болтай" (выход за границу строки допускается)
sub3 = str[10,-4]            # nil (отрицательная длина)
```

Важно помнить, что это именно смещение и длина (число символов), а не начальное и конечное смещение.

Если индекс отрицателен, то отсчет ведется от конца строки. В этом случае индекс начинается с единицы, а не с нуля. Но при нахождении подстроки указанной длины все равно берутся символы правее, а не левее начального:

```
str1 = "Алиса"
sub1 = str1[-3,3]            # "иса"
str2 = "В Зазеркалье"
sub3 = str2[-8,6]            # "зеркал"
```

Можно задавать диапазон. Он интерпретируется как диапазон позиций внутри строки. Диапазон может включать отрицательные числа, но в любом случае нижняя граница не должна быть больше верхней. Если диапазон «инвертированный» или нижняя граница оказывается вне строки, возвращается nil:

```
str = "Уинстон Черчилль"
sub1 = str[8..13]            # "Черчил"
sub2 = str[-4..-1]           # "илль"
sub3 = str[-1..-4]           # nil
sub4 = str[25..30]           # nil
```

Если задано регулярное выражение, то возвращается строка, соответствующая образцу. Если соответствия нет, возвращается nil:

```
str = "Alistair Cooke"
sub1 = str[/l..t/]           # "list"
sub2 = str[/s.*r/]           # "stair"
sub3 = str[/foo/]            # nil
```

Если задана строка, то она и возвращается, если встречается в качестве подстроки в исходной строке, в противном случае возвращается nil:

```
str = "theater"
sub1 = str["heat"]           # "heat"
sub2 = str["eat"]            # "eat"
sub3 = str["ate"]            # "ate"
sub4 = str["beat"]           # nil
sub5 = str["cheat"]          # nil
```

Наконец, в тривиальном случае, когда в качестве индекса задано одно число Fixnum, возвращается один символ (или nil, если индекс выходит за границы строки):

```
str = "Aaron Burr"
ch1 = str[0]                # 65
ch1 = str[1]                # 97
ch3 = str[99]               # nil
```

Важно понимать, что все описанные выше способы могут использоваться не только для доступа к подстроке, но и для ее замены:

```
str1 = "Шалтай-Болтай"
str1[7,3] = "Хва"           # "Шалтай-Хватай"

str2 = "Алиса"
str2[-3,3] = "ександра"     # "Александра"

str3 = "В Зазеркалье"
str3[-9,9] = "стеколье"     # "В Застеколье"

str4 = "Уинстон Черчилль"
str4[8..11] = "X"           # "Уинстон Хилль"

str5 = "Alistair Cooke"
str5[/e$/] ="ie Monster"    # "Alistair Cookie Monster"

str6 = "theater"
str6["er"] = "re"           # "theatre"

str7 = "Aaron Burr"
str7[0] = 66                # "Baron Burr"
```

Присваивание выражения, равного nil, не оказывает никакого действия.

2.13. Подстановка в строках

Мы уже видели, как выполняются простые подстановки. Методы sub и gsub предоставляют более развитые средства, основанные на сопоставлении с образцом. Имеются также варианты sub! и gsub!, позволяющие выполнить подстановку «на месте».

Метод sub заменяет первое вхождение строки, соответствующей образцу, другой строкой или результатом вычисления блока:

```
s1 = "spam, spam, and eggs"
s2 = s1.sub(/spam/,"bacon")                 # "bacon, spam, and eggs"

s3 = s2.sub(/(\w+), (\w+),/,'\2, \1,')      # "spam, bacon, and eggs"

s4 = "Don't forget the spam."
s5 = s4.sub(/spam/) { |m| m.reverse }       # "Don't forget the maps."

s4.sub!(/spam/) { |m| m.reverse }
# s4 теперь равно "Don't forget the maps."
```

Как видите, в подставляемой строке могут встречаться специальные символы \1, \2 и т.д. Но такие специальные переменные, как $& (или ее англоязычная версия $MATCH), не допускаются.

Если употребляется форма с блоком, то допустимы и специальные переменные. Если вам нужно лишь получить сопоставленную с образцом строку, то она будет передана в блок как параметр. Если эта строка вообще не нужна, то параметр, конечно, можно опустить.

Метод gsub (глобальная подстановка) отличается от sub лишь тем, что заменяются все вхождения, а не только первое:

```
s5 = "alfalfa abracadabra"
s6 = s5.gsub(/a[bl]/,"xx")       # "xxfxxfa xxracadxxra"
s5.gsub!(/[lfdbr]/) { |m| m.upcase + "-" }
# s5 теперь равно "aL-F-aL-F-a aB-R-acaD-aB-R-a"
```

Метод Regexp.last_match эквивалентен действию специальной переменной $& (она же $MATCH).

2.14. Поиск в строке

Помимо различных способов доступа к подстрокам, есть и другие методы поиска в строке. Метод index возвращает начальную позицию заданной подстроки, символа или регулярного выражения. Если подстрока не найдена, возвращается nil:

```
str = "Albert Einstein"
pos1 = str.index(?E)             # 7
pos2 = str.index("bert")         # 2
pos3 = str.index(/in/)           # 8
pos4 = str.index(?W)             # nil
pos5 = str.index("bart")         # nil
pos6 = str.index(/wein/)         # nil
```

Метод rindex начинает поиск с конца строки. Но номера позиций отсчитываются тем не менее от начала:

```
str = "Albert Einstein"
pos1 = str.rindex(?E)            # 7
pos2 = str.rindex("bert")        # 2
pos3 = str.rindex(/in/)          # 13 (найдено самое правое соответствие)
pos4 = str.rindex(?W)            # nil
pos5 = str.rindex("bart")        # nil
pos6 = str.rindex(/wein/)        # nil
```

Метод include? сообщает, встречается ли в данной строке указанная подстрока или один символ:

```
str1 = "mathematics"
flag1 = str1.include? ?e         # true
flag2 = str1.include? "math"     # true
str2 = "Daylight Saving Time"
flag3 = str2.include? ?s         # false
flag4 = str2.include? "Savings"  # false
```

Метод scan многократно просматривает строку в поисках указанного образца. Будучи вызван внутри блока, он возвращает массив. Если образец содержит несколько (заключенных в скобки) групп, то массив окажется вложенным:

```
str1 = "abracadabra"
sub1 = str1.scan(/a./)
# sub1 теперь равно ["ab","ac","ad","ab"]
str2 = "Acapulco, Mexico"
sub2 = str2.scan(/(.)(c.)/)
# sub2 теперь равно [ ["A","ca"], ["l","co"], ["i","co"] ]
```

Если при вызове задан блок, то метод поочередно передает этому блоку найденные значения:

```
str3 = "Kobayashi"
str3.scan(/[^aeiou]+[aeiou]/) do |x|
   print "Слог: #{x}\n"
end
```

Этот код выводит такой результат:

```
Слог: Ko
Слог: ba
Слог: ya
Слог: shi
```

2.15. Преобразование символов в коды ASCII и обратно

Одиночные символы в Ruby возвращаются в виде односимвольных строк, например:

```
str = "Martin"
print str[0]                    # "M"
```

В классе Integer имеется метод chr, который преобразует целое число в символ. По умолчанию целые интерпретируются в кодировке ASCII, но можно задать и другие кодировки для значений, больших 127. В классе String имеется метод ord для выполнения противоположного действия:

```
str = 77.chr                    # "M"
s2 = 233.chr("UTF-8")           # "é"
num = "M".ord                   # 77
```

2.16. Явные и неявные преобразования

На первый взгляд, методы to_s и to_str могут вызвать недоумение. Ведь оба преобразуют объект в строковое представление, так?

Но есть и различия. Во-первых, *любой* объект в принципе можно как-то преобразовать в строку, поэтому почти все системные классы обладают методом to_s. Однако метод to_str в системных классах не реализуется никогда.

Как правило, метод to_str применяется для объектов, очень похожих на строки, способных «замаскироваться» под строку. В общем, можете считать, что метод to_s – это *явное преобразование*, а метод to_str – *неявное преобразование*.

Я уже сказал, что ни в одном системном классе не *определен* метод to_str (по крайней мере, мне о таких классах неизвестно). Но иногда они *вызывают* to_str (если такой метод существует в соответствующем классе).

Первое, что приходит на ум, – *подкласс* класса String; но на самом деле объект любого класса, производного от String, уже является строкой, так что определять метод to_str излишне.

На практике методы to_s и to_str обычно возвращают одно и то же значение, но это *необязательно*. Неявное преобразование должно давать «истинное строковое значение» объекта, а явное можно расценивать как «принудительное» преобразование.

Метод puts обращается к методу to_s объекта, чтобы получить его строковое представление. Можно считать, что это неявный вызов явного преобразования. То же самое справедливо в отношении интерполяции строк. Вот пример:

```
class Helium
   def to_s
      "He"
   end
   def to_str
      "гелий"
   end
end

e = Helium.new
print "Элемент "
puts e    # Элемент He
puts "Element is " + e      # Элемент гелий
puts "Element is #{e}"      # Элемент He
```

Как видите, разумное определение этих методов в собственном классе может несколько повысить гибкость применения. Но что сказать об идентификации объектов, переданных методам вашего класса?

Предположим, например, что вы написали метод, который ожидает в качестве параметра объект String. Вопреки философии динамической типизации, так делают часто, и это вполне оправдано. Например, предполагается, что первый параметр метода File.new – строка.

Решить эту проблему просто. Если вы ожидаете на входе строку, проверьте, имеет ли объект метод to_str, и при необходимости вызывайте его.

```
def set_title(title)
   if title.respond_to? :to_str
      title = title.to_str
   end
   # ...
end
```

Ну а если объект *не* отвечает на вызов метода to_str? Есть несколько вариантов действий. Можно принудительно вызвать метод to_s; можно проверить, принадлежит ли объект классу String или его подклассу; можно, наконец, продолжать работать, понимая, что при попытке выполнить операцию, которую объект не поддерживает, мы получим исключение ArgumentError.

Короткий путь к цели выглядит так:

```
title = title.to_str if title.respond_to?(:to_str)
```

и в этом случае значение title заменяется, только если в его классе имеется метод to_str.

При интерполяции в строку, заключенную в двойные кавычки, неявно вызывается метод to_s, обычно это самый простой способ преобразовать в строки сразу несколько объектов:

```
e = Helium.new
str = "Pi #{3.14} и элемент #{e}
# str теперь равно "3.14 и элемент He"
```

С помощью неявного преобразования можно было бы сделать строки и числа практически эквивалентными:

```
class Fixnum
   def to_str
      self.to_s
   end
end

str = "Число равно " + 345        # Число равно 345
```

Но я не рекомендую так поступать: «много хорошо тоже нехорошо». В Ruby, как и в большинстве языков, строки и числа – разные сущности. Мне кажется, что ясности ради преобразования как правило должны быть явными.

И еще: в методе to_str нет ничего *волшебного*. Предполагается, что он возвращает строку, но если вы пишете такой метод сами, то ответственность за то, что он действительно так и поступает, ложится на вас.

2.17. Дописывание в конец строки

Для конкатенации строк применяется оператор <<. Он «каскадный», то есть позволяет выполнять подряд несколько операций над одним и тем же операндом-приемником.

```
str = "A"
str << [1,2,3].to_s << " " << (3.14).to_s
# str теперь равно "A123 3.14"
```

2.18. Удаление хвостовых символов новой строки и прочих

Часто бывает необходимо удалить лишние символы в конце строки. Типичный пример – удаление символа новой строки после чтения строки из внешнего источника.

Метод chop удаляет последний символ строки (обычно это символ новой строки). Если перед символом новой строки находится символ перевода каретки (\r), то он тоже удаляется. Причина такого поведения заключается в том, что разные операционные системы неодинаково трактуют понятие «новой строки». В UNIX-подобных системах новая строка представляется символом \n. А в Windows для этой цели используется пара символов \r\n.

```
str = gets.chop                 # Прочитать, удалить символ новой строки
s2 = "Some string\n"            # "Some string" (нет символа новой строки)
s3 = s2.chop!                   # s2 теперь тоже равно "Some string"
s4 = "Other string\r\n"
s4.chop!                        # "Other string" (нет символа новой строки)
```

Обратите внимание, что при вызове варианта chop! операнд-источник модифицируется.

Важно еще отметить, что последний символ удаляется, даже если это не символ новой строки:

```
str = "abcxyz"
s1 = str.chop                   # "abcxy"
```

Поскольку символ новой строки присутствует не всегда, иногда удобнее применять метод chomp:

```
str = "abcxyz"
str2 = "123\n"
str3 = "123\r"
str4 = "123\r\n"
s1 = str.chomp                  # "abcxyz"
s2 = str2.chomp                 # "123"
# Если установлен стандартный разделитель записей, то удаляется не только
# \n, но также \r и \r\n
s3 = str3.chomp                 # "123"
s4 = str4.chomp                 # "123"
```

Как и следовало ожидать, имеется также метод chomp! для замены «на месте».

Если методу chomp передана строка-параметр, то удаляются перечисленные в ней символы, а не подразумеваемый по умолчанию разделитель записей. Кстати, если разделитель записей встречается в середине строки, то он не удаляется:

```
str1 = "abcxyz"
str2 = "abcxyz"
s1 = str1.chomp("yz")           # "abcx"
s2 = str2.chomp("x")            # "abcxyz"
```

2.19. Убирание лишних пропусков

Метод strip удаляет пропуски в начале и в конце строки, а вариант strip! делает то же самое «на месте».

```
str1 = "\t \nabc \t\n"
str2 = str1.strip# "abc"
```

```
str3 = str1.strip!                    # "abc"
# str1 теперь тоже равно "abc"
```

Под пропусками, как обычно, понимаются пробелы, символы табуляции и перехода на новую строку.

Чтобы удалить пропуски только в начале или только в конце строки, применяйте методы lstrip и rstrip:

```
str = " abc "
s2 = str.lstrip                       # "abc "
s3 = str.rstrip                       # " abc"
```

Имеются также варианты lstrip! и rstrip! для удаления «на месте».

2.20. Повтор строк

В Ruby оператор (или метод) умножения перегружен так, что в применении к строкам выполняет операцию повторения. Если строку умножить на n, то получится строка, состоящая из n конкатенированных копий исходной:

```
etc = "Etc. "*3                       # "Etc. Etc. Etc. "
ruler = "+" + ("."*4+"5"+"."*4+"+")*3
# "+....5....+....5....+....5....+"
```

2.21. Включение выражений в строку

Это легко позволяет сделать синтаксическая конструкция #{}. Нет нужды думать о преобразовании, добавлении и конкатенации; нужно лишь интерполировать переменную или другое выражение в любое место строки:

```
puts "#{temp_f} по Фаренгейту равно #{temp_c} по Цельсию"
puts "Значение определителя равно #{b*b - 4*a*c}."
puts "#{word} это #{word.reverse} наоборот."
```

Внутри фигурных скобок могут находиться даже полные предложения. При этом возвращается результат вычисления последнего выражения.

```
str = "Ответ равен #{ def factorial(n)
      n==0 ? 1 : n*factorial(n-1)
      end
answer = factorial(3) * 7}, естественно."
# Ответ равен 42, естественно.
```

При интерполяции глобальных переменных, а также переменных класса и экземпляра фигурные скобки можно опускать:

```
puts "$gvar = #$gvar и ivar = #@ivar."
```

Интерполяция не производится внутри строк, заключенных в одиночные кавычки (поскольку их значение не интерпретируется), но применима к заключенным в двойные кавычки встроенным документам и к регулярным выражениям.

2.22. Отложенная интерполяция

Иногда желательно отложить интерполяцию значений в строку. Идеального способа решить эту задачу не существует.

Наивный подход — сохранить строку в одиночных кавычках, а затем вычислить ее:

```
str = '#{name} - мое имя, а #{nation} - моя страна.'
name, nation = "Стивен Дедалус", "Ирландия"
s1 = eval('"' + str + '"')
# Стивен Дедалус - мое имя, а Ирландия - моя страна.
```

Однако использование `eval` — почти всегда худшее из возможных решений. При этом возможно великое множество проблем — от чрезвычайно медленного выполнения до неожиданных уязвимостей к эксплойтам, поэтому лучше держаться от этого подальше.

Гораздо менее опасный способ — воспользоваться блоком:

```
str = proc do |name, nation|
  "#{name} - мое имя, а #{nation} - моя страна."
end
s2 = str.call("Гулливер Фойл", "Терра")
# Гулливер Фойл - мое имя, а Терра - моя страна.
```

2.23. Разбор данных, разделенных запятыми

Данные, разделенные запятыми, часто встречаются при программировании. Это в некотором роде «общий знаменатель» всех форматов обмена данными. Например, так передаются данные между несовместимыми СУБД или приложениями, которые не поддерживают никакого другого общего формата.

Будем предполагать, что данные представляют собой строки и числа, а все строки заключены в кавычки. Еще предположим, что все символы должным образом экранированы (например, запятые и кавычки внутри строки).

Задача оказывается простой, поскольку такой формат данных подозрительно напоминает встроенные в Ruby массивы данных разных типов. Достаточно заключить все выражение в квадратные скобки, чтобы получить массив.

```
string = gets.chop!
# Предположим, что прочитана такая строка:
# "Doe, John", 35, 225, "5'10\"", "555-0123"
data = eval("[" + string + "]")          # Преобразовать в массив
data.each {|x| puts "Значение = #{x}"}
Этот код выводит такой результат:
Значение = Doe, John
Значение = 35
Значение = 225
Значение = 5' 10"
Значение = 555-0123
```

Более общее решение дает стандартная библиотека CSV.

2.24. Преобразование строки в число (десятичное или иное)

Есть два основных способа преобразовать строку в число: методы `Integer` и `Float` модуля `Kernel` и методы `to_i` и `to_f` класса `String`. (Имена, начинающиеся с заглавной буквы, например `Integer`, обычно резервируются для специальных функций преобразования.)

Простой случай тривиален, следующие два предложения эквивалентны:

```
x = "123".to_i                  # 123
y = Integer("123")              # 123
```

Но если в строке хранится не число, то поведение этих методов различается:

```
x = "junk".to_i                 # молча возвращает 0
y = Integer("junk")             # ошибка
```

Метод `to_i` прекращает преобразование, как только встречает первый символ, не являющийся цифрой, а метод `Integer` в этом случае возбуждает исключение:

```
x = "123junk".to_i              # 123
y = Integer("123junk")          # error
```

Оба метода допускают наличие пропусков в начале и в конце строки:

```
x = " 123 ".to_i                # 123
y = Integer(" 123 ")            # 123
```

Преобразование строки в число с плавающей точкой работает аналогично:

```
x = "3.1416".to_f               # 3.1416
y = Float("2.718")              # 2.718
```

Оба метода понимают научную нотацию:

```
x = Float("6.02e23")            # 6.02e23
y = "2.9979246e5".to_f          # 299792.46
```

Методы `to_i` и `Integer` также по-разному относятся к системе счисления. По умолчанию, естественно, подразумевается система по основанию 10, но другие тоже допускаются (это справедливо и для чисел с плавающей точкой).

Говоря о преобразовании из одной системы счисления в другую, мы всегда имеем в виду строки. Ведь целое число неизменно хранится в двоичном виде.

Следовательно, преобразование системы счисления – это всегда преобразование одной строки в другую. Здесь мы рассмотрим преобразование *из* строки (обратное преобразование рассматривается в разделах 5.18 и 5.5).

Числу в тексте программы может предшествовать префикс, обозначающий основание системы счисления. Префикс `0b` обозначает двоичное число, `0` – восьмеричное, а `0x` – шестнадцатеричное.

Метод `Integer` такие префиксы понимает, а метод `to_i` – нет:

```
x = Integer("0b111")            # двоичное - возвращает 7
y = Integer("0111")             # восьмеричное - возвращает 73
z = Integer("0x111")            # шестнадцатеричное - возвращает 291

x = "0b111".to_i                # 0
```

```
y = "0111".to_i                 # 0
z = "0x111".to_i                # 0
```

Однако у метода to_i есть необязательный второй параметр для указания основания. Обычно применяют только четыре основания: 2, 8, 10 (по умолчанию) и 16. Впрочем, префиксы не распознаются даже при задании основания.

```
x = "111".to_i(2)               # 7
y = "111".to_i(8)               # восьмеричное - возвращает 73
z = "111".to_i(16)              # шестнадцатеричное - возвращает 291

x = "0b111".to_i # 0
y = "0111".to_i  # 0
z = "0x111".to_i # 0
```

Из-за «стандартного» поведения этих методов, цифры, недопустимые при данном основании, обрабатываются по-разному:

```
x = "12389".to_i(8)             # 123 (8 игнорируется)
y = Integer("012389")           # ошибка (8 недопустима)
```

Хотя полезность этого и сомнительна, метод to_i понимает основания вплоть до 36, когда в представлении числа допустимы все буквы латинского алфавита. (Возможно, это напомнило вам о base64-кодировании; дополнительную информацию по этому поводу см. в разделе 2.37.)

```
x = "123".to_i(5)               # 66
y = "ruby".to_i(36)             # 1299022
```

Для преобразования символьной строки в число можно также воспользоваться методом scanf из стандартной библиотеки, которая добавляет его в модуль Kernel, а также классы IO и String:

```
str = "234 234 234"
x, y, z = str.scanf("%d %o %x")     # 234, 156, 564
```

Метод scanf реализует всю имеющую смысл функциональность стандартных функций scanf, sscanf и fscanf из библиотеки языка C. Но строки, представляющие двоичные числа, он не обрабатывает.

2.25. Кодирование и декодирование строк в кодировке rot13

Rot13 – наверное, самый слабый из известных человечеству шифров. Исторически он просто препятствовал «случайному» прочтению текста. Он часто встречается в конференциях Usenet; например, так можно закодировать потенциально обидную шутку или весь сценарий «Звездных войн. Эпизод 13» накануне премьеры.

Принцип кодирования состоит в смещении символов относительно начала алфавита (латинского) на 13: A превращается в N, B – в O и т.д. Строчные буквы смещаются на ту же величину; цифры, знаки препинания и прочие игнорируются. Поскольку 13 – это ровно половина от 26 (число букв в латинском алфавите), то функция является обратной самой себе, то есть ее повторное применение восстанавливает исходный текст.

Ниже приведена реализация этого метода, добавленного в класс String, никаких особых комментариев она не требует:

```
class String

   def rot13
      self.tr("A-Ma-mN-Zn-z","N-Zn-zA-Ma-m")
   end

end

joke = "Y2K bug"
joke13 = joke.rot13 # "L2X oht"

episode2 = "Fcbvyre: Naanxva qbrfa'g trg xvyyrq."
puts episode2.rot13
```

2.26. Шифрование строк

Иногда нежелательно, чтобы строки можно было легко распознать. Например, пароли не следует хранить в открытом виде, какими бы ограничительными ни были права доступа к файлу.

В стандартном методе crypt применяется стандартная функция с тем же именем для шифрования строки по алгоритму DES. Она принимает в качестве параметра «затравку» (ее назначение то же, что у затравки генератора случайных чисел). На платформах, отличных от UNIX, параметр может быть иным.

Ниже показано тривиальное приложение, которое запрашивает пароль, знакомый любителям Толкиена:

```
coded = "hfCghHIE5LAM."

puts "Говори, друг, и жми Enter!"

print "Пароль: "
password = gets.chop

if password.crypt("hf") == coded
   puts "Добро пожаловать!"
else
   puts "Кто ты, орк?"
end
```

Стоит отметить, что никогда не следует применять шифрование для хранения паролей. Лучше хранить *свертку пароля*, вычисленную с помощью алгоритма хэширования, специально предназначенного для этой цели, например bcrypt. Кроме того, никогда не полагайтесь на шифрование при обмене данных с серверным веб-приложением. Для обеспечения безопасности в этом случае пользуйтесь протоколом HTTPS, в котором применяется протокол SSL (Secure Sockets Layer) для

шифрования всего трафика. Разумеется, никто не запрещает пользоваться шифрованием на сервере, но по другой причине – чтобы защитить данные в хранилище, а не во время передачи по сети.

2.27. Сжатие строк

Для сжатия строк и файлов применяется библиотека Zlib.

Зачем может понадобиться сжимать строки? Возможно, чтобы ускорить ввод-вывод из базы данных, оптимизировать использование сети или усложнить распознавание строк.

В классах Deflate и Inflate имеются методы класса deflate и inflate соответственно. У метода deflate (он выполняет сжатие) есть дополнительный параметр, задающий режим сжатия. Он определяет компромисс между качеством сжатия и скоростью. Если значение равно BEST_COMPRESSION, то строка сжимается максимально, но это занимает сравнительно много времени. Значение BEST_SPEED задает максимальную скорость, но при этом строка сжимается хуже. Подразумеваемое по умолчанию значение DEFAULT_COMPRESSION выбирает компромиссный режим.

```
require 'zlib'
include Zlib

long_string = ("abcde"*71 + "defghi"*79 + "ghijkl"*113)*371
# long_string состоит из 559097 символов

s1 = Deflate.deflate(long_string,BEST_SPEED)       # 4188 символов
s2 = Deflate.deflate(long_string)                  # 3568 символов
s3 = Deflate.deflate(long_string,BEST_COMPRESSION) # 2120 символов
```

Неформальные эксперименты показывают, что скорость отличается примерно в два раза, а плотность сжатия – в обратной пропорции на ту же величину. И скорость, и плотность сильно зависят от состава строки. Разумеется, на скорость влияет и имеющееся оборудование.

Имейте в виду, что существует пороговое значение длины строки. Если строка короче, то сжимать ее практически бесполезно (если только вы не хотите сделать ее нечитаемой). В этом случае неизбежные накладные расходы могут даже привести к тому, что сжатая строка окажется *длиннее* исходной.

2.28. Подсчет числа символов в строке

Метод count подсчитывает число вхождений в строку символов из заданного набора:

```
s1 = "abracadabra"
a = s1.count("c")              # 1
b = s1.count("bdr")            # 5
```

Строковый параметр ведет себя как простое регулярное выражение. Если он начинается с символа ^, то берется дополнение к списку:

```
c = s1.count("^a")              # 6
d = s1.count("^bdr")            # 6
```

Дефис обозначает диапазон символов:

```
e = s1.count("a-d")             # 9
f = s1.count("^a-d")            # 2
```

2.29. Обращение строки

Для обращения строки служит метод reverse (или его вариант для обращения «на месте» reverse!):

```
s1 = "Star Trek"
s2 = s1.reverse                 # "kerT ratS"
s1.reverse!                     # s1 теперь равно "kerT ratS"
```

Пусть требуется обратить порядок слов (а не символов). Тогда можно сначала воспользоваться методом String#split, который вернет массив слов. В классе Array тоже есть метод reverse, поэтому можно обратить массив, а затем с помощью метода join объединить слова в новую строку:

```
phrase = "Now here's a sentence"
phrase.split(" ").reverse.join(" ")
# "sentence a here's Now"
```

2.30. Удаление дубликатов

Цепочки повторяющихся символов можно сжать до одного методом squeeze:

```
s1 = "bookkeeper"
s2 = s1.squeeze         # "bokeper"
s3 = "Hello..."
s4 = s3.squeeze         # "Helo."
```

Если указан параметр, то будут удаляться только дубликаты заданных в нем символов:

```
s5 = s3.squeeze(".")    # "Hello."
```

Этот параметр подчиняется тем же правилам, что и параметр метода count (см. раздел 2.28 выше), то есть допускаются дефис и символ ^

Имеется также метод squeeze!.

2.31. Удаление заданных символов

Метод delete удаляет из строки те символы, которые включены в список, переданный в качестве параметра:

```
s1 = "To be, or not to be"
s2 = s1.delete("b")         # "To e, or not to e"
s3 = "Veni, vidi, vici!"
s4 = s3.delete(",!")        # "Veni vidi vici"
```

Этот параметр подчиняется тем же правилам, что и параметр метода count (см. раздел 2.28 выше), то есть допускаются дефис и символ ^.

Имеется также метод delete!.

2.32. Печать специальных символов

Метод dump позволяет получить графическое представление символов, которые обычно не печатаются вовсе или вызывают побочные эффекты:

```
s1 = "Внимание" << 7 << 7 << 7      # Добавлено три символа ASCII BEL
puts s1.dump                        # Печатается: Внимание\007\007\007
s2 = "abc\t\tdef\tghi\n\n"
puts s2.dump                        # Печатается: abc\t\tdef\tghi\n\n
s3 = "Двойная кавычка: \""
puts s3.dump                        # Печатается: Двойная кавычка: \"
```

2.33. Генерирование последовательности строк

Изредка бывает необходимо получить «следующую» строку. Так, следующей для строки «aaa" будет строка "aab" (затем "aac", "aae" и так далее).

В Ruby для этой цели есть метод succ:

```
droid = "R2D2"
improved = droid.succ               # "R2D3"
pill = "Vitamin B"
pill2 = pill.succ                   # "Vitamin C"
```

Не рекомендуется применять этот метод, если точно не известно, что начальное значение предсказуемо и разумно. Если начать с какой-нибудь экзотической строки, то рано или поздно вы получите странный результат.

Существует также метод upto, который в цикле вызывает succ, пока не будет достигнуто конечное значение:

```
"Files, A".upto "Files, X" do |letter|
   puts "Opening: #{letter}"
end

# Выводится 24 строки
```

Еще раз подчеркнем, что эта возможность используется редко, да и то на ваш страх и риск. Кстати, метода, возвращающего «предшествующую» строку, не существует.

2.34. Вычисление 32-разрядного CRC

Контрольный код циклической избыточности (Cyclic Redundancy Checksum CRC) – хорошо известный способ получить «сигнатуру» файла или произвольного массива байтов. CRC обладает тем свойством, что вероятность получения одинакового кода для разных входных данных равна 1 / 2**N, где N – число битов результата (чаще всего 32).

Вычислить его позволяет библиотека zlib, написанная Уэно Кацухиро (Ueno Katsuhiro). Метод crc32 вычисляет CRC для строки, переданной в качестве параметра:

```
require 'zlib'
include Zlib
crc = crc32("Hello")   # 4157704578
crc = crc32(" world!",crc)  # 461707669
crc = crc32("Hello world!") # 461707669 (то же, что и выше)
```

В качестве необязательного второго параметра можно передать ранее вычисленный CRC. Результат получится такой, как если бы конкатенировать обе строки и вычислить CRC для объединения. Это полезно, например, когда нужно вычислить CRC файла, настолько большого, что прочитать его можно только по частям.

2.35. Вычисление SHA-256-свертки строки

Класс Digest::SHA256 вырабатывает 256-разрядный *хэш* или *дайджест сообщения* произвольной длины. Это односторонняя функция хэширования, т. е. восстановить исходное сообщение по дайджесту невозможно. В библиотеке Digest существуют также классы MD5, SHA384 и SHA512, реализующие соответствующие алгоритмы.

Чаще всего применяется метод класса hexdigest, но существуют также методы digest и base64digest. Все они принимают строку сообщения и возвращают его дайджест в виде строки:

```
require 'digest'
Digest::SHA256.hexdigest("foo")[0..20]    # "2c26b46b68f"
Digest::SHA256.base64digest("foo")[0..20] # "LCa0a2j/xo/"
Digest::SHA256.digest("foo")[0..5]        # ",&\xB4kh\xFF"
```

Хотя метод digest возвращает 64-байтовую строку, содержащую 512-разрядный дайджест, на практике наиболее полезен метод hexdigest. Он возвращает дайджест в виде строки из 64 шестнадцатеричных символов, представляющей 64 байта.

Метод экземпляр update позволяет строить хэш инкрементно, например, когда данные поступают из внешнего потока:

```
secret = Digest::SHA256.new
source.each { |chunk| secret.update(chunk) }
```

Повторные вызовы эквивалентные одному вызову с конкатенированными аргументами:

```
# Эти два предложения...
cryptic.update("Data...")
cryptic.update(" and more data.")

# ...эквивалентные такому:
cryptic.update("Data... and more data.")

cryptic.hexdigest[0..20] # "50605ba0a90"
```

2.36. Вычисление расстояния Левенштейна между двумя строками

Расстояние между строками важно знать в индуктивном обучение (искусственный интеллект), криптографии, исследовании структуры белков и других областях.

Расстоянием Левенштейна называется минимальное число элементарных модификаций, которым нужно подвергнуть одну строку, чтобы преобразовать ее в другую. Элементарными модификациями называются следующие операции: *del* (удаление одного символа), *ins* (замена символа) и *sub* (замена символа). Замену можно также считать комбинацией удаления и вставки (*indel*).

Существуют разные подходы к решению этой задачи, но не будем вдаваться в технические детали. Достаточно знать, что реализация на Ruby (см. листинг 2.2) позволяет задавать дополнительные параметры, определяющие стоимость всех трех операций модификации. По умолчанию за базовую принимается стоимость одной операции indel (стоимость вставки = стоимость удаления).

Листинг 2.2. Расстояние Левенштейна

```
class String

   def levenshtein(other, ins=2, del=2, sub=1)
      # ins, del, sub - взвешенные стоимости
      return nil if self.nil?
      return nil if other.nil?
      dm = []    # матрица расстояний

      # Инициализировать первую строку
      dm[0] = (0..self.length).collect { |i| i * ins }
      fill = [0] * (self.length - 1)

      # Инициализировать первую колонку
      for i in 1..other.length
         dm[i] = [i * del, fill.flatten]
      end

      # заполнить матрицу
      for i in 1..other.length
         for j in 1..self.length
            # главное сравнение
            dm[i][j] = [
               dm[i-1][j-1] +
                  (self[j-1] == other[i-1] ? 0 : sub),
                     dm[i][j-1] + ins,
               dm[i-1][j] + del
            ].min
         end
```

```
      end

      # Последнее значение в матрице и есть
      # расстояние Левенштейна между строками
      dm[other.length][self.length]
   end

end


s1 = "ACUGAUGUGA"
s2 = "AUGGAA"
d1 = s1.levenshtein(s2)  # 9
s3 = "pennsylvania"
s4 = "pencilvaneya"
d2 = s3.levenshtein(s4)  # 7
s5 = "abcd"
s6 = "abcd"
d3 = s5.levenshtein(s6)  # 0
```

Определив расстояние Левенштейна, мы можем написать метод similar? , вычисляющий меру схожести строк. Например:

```
class String

   def similar?(other, thresh=2)
      if self.levenshtein(other) < thresh
        true
      else
        false
      end
   end

end


if "polarity".similar?("hilarity")
   puts "Электричество - забавная штука!"
end
```

Разумеется, можно было бы передать методу similar? три взвешенных стоимости, которые он в свою очередь передал бы методу levenshtein. Но для простоты мы не стали этого делать.

2.37. base64-кодирование и декодирование строк

Алгоритм base64 часто применяется для преобразования двоичных данных в текстовую форму, не содержащую специальных символов. Например, изображения и шрифты, хранящиеся непосредственно в CSS-файлах, представлены в кодировке base64.

Простейший способ осуществить base64-кодирование и декодирование — воспользоваться встроенным модулей Base64. В классе Base64 имеется метод encode64, который возвращает строку в кодировке base64 (с добавленным в конец символом новой строки). Имеется также метод decode64, который декодирует такую строку:

```
require "base64"
str = "\xAB\xBA\x02abdce"
encoded  = Base64.encode64(str)        # "q7oCYWJkY2U=\n"
original = Base64.decode64(encoded)    # "\xAB\xBA\x02abdce"
```

2.38. Замена символов табуляции пробелами и сворачивание пробелов в табуляторы

Бывает, что имеется строка с символами табуляции, а мы хотели бы преобразовать их в пробелы (или наоборот). Ниже показано два метода, реализующих эти операции:

```
class String

   def detab(ts=8)
      str = self.dup
      while (leftmost = str.index("\t")) != nil
         space = " "*(ts-(leftmost%ts))
         str[leftmost]=space
      end
      str
   end

   def entab(ts=8)
      str = self.detab
      areas = str.length/ts
      newstr = ""
      for a in 0..areas
         temp = str[a*ts..a*ts+ts-1]
         if temp.size==ts
            if temp =~ / +/
               match=Regexp.last_match[0]
               endmatch = Regexp.new(match+"$")
               if match.length>1
                  temp.sub!(endmatch,"\t")
               end
            end
         end
         newstr += temp
      end
      newstr
   end

end
```

```
foo = "Это  всеголишь  тест. "

puts foo
puts foo.entab(4)
puts foo.entab(4).dump
```

Отметим, что этот код не распознает символы забоя.

2.39. Перенос строк по словам

Иногда бывает необходимо напечатать длинные строки текста, задав ширину поля. Приведенный ниже код решает эту задачу, разбивая текст по границам слов и учитывая символы табуляции (но символы забоя не учитываются, а табуляция не сохраняется):

```
str = <<-EOF
   When in the Course of human events it becomes necessary
   for one people to dissolve the political bands which have
   connected them with another, and to assume among the powers
   of the earth the separate and equal station to which the Laws
   of Nature and of Nature's God entitle them, a decent respect
   for the opinions of mankind requires that they should declare
   the causes which impel them to the separation.
EOF

max = 20

line = 0
out = [""]

input = str.gsub(/\n/," ")
words = input.split(" ")

while input != ""
   word = words.shift
   break if not word
   if out[line].length + word.length > max
      out[line].squeeze!(" ")
      line += 1
      out[line] = ""
   end
   out[line] << word + " "
end

out.each {|line| puts line} # Печатает 24 очень коротких строки
```

Gem-пакет ActiveSupport содержит очень похожую функциональность в методе word_wrap, есть в нем и много других методов для работы с текстом. Поищите в сети.

2.40. Заключение

Мы рассмотрели основы представления строк (заключенных в одиночные или двойные кавычки). Мы видели, как интерполировать выражения в строку в двойных кавычках, и узнали, что в таких строках допустимы некоторые специальные символы, представленные управляющими последовательностями. Мы также познакомились с конструкциями %q и %Q, которые позволяют по своему вкусу выбирать ограничители. Наконец, мы рассмотрели синтаксис встроенных документов, унаследованных из старых продуктов, в том числе командных интерпретаторов в UNIX.

В этой главе были продемонстрированы все наиболее важные операции, которые программисты обычно выполняют над строками: конкатенация, поиск, извлечение подстрок, разбиение на лексемы и т.д. Мы видели, как можно кодировать строки (например, по алгоритму base64) и сжимать их.

Пришло время перейти к тесно связанной со строками теме – регулярным выражениям. Регулярные выражения – это мощное средство сопоставления строк с образцами. Мы рассмотрим их в следующей главе.

Глава 3. Регулярные выражения

Я провела бы его по лабиринту,
где тропы орнаментом украшены...

– Эми Лоуэлл

Мощь регулярных выражений как инструмента программирования часто недооценивается. Первые теоретические исследования на эту тему датируются сороковыми годами прошлого века, в вычислительные системы они проникли в 1960-х годах, а затем были включены в различные инструментальные средства операционной системы UNIX. В 1990-х годах популярность языка Perl привела к тому, что регулярные выражения вошли в обиход, перестав быть уделом бородатых гуру.

Красота регулярных выражений заключается в том, что почти весь наш опыт можно выразить в терминах образцов. А если имеется образец, то можно провести сопоставление с ним, можно найти то, что ему соответствует, и заменить найденное чем-то другим по своему выбору.

Используемый в Ruby движок регулярных выражений называется Onigmo. Это переработка библиотеки Oniguruma, которая была включена в Ruby 1.9. Авторы, не являющиеся японцами, обычно пишут Oniguruma неправильно, поэтому переименование очень кстати – слово Onigmo гораздо проще.

3.1. Синтаксис регулярных выражений

Обычно регулярное выражение ограничено с двух сторон символами косой черты. Применяется также форма `%r`. В таблице 3.1 приведены примеры простых регулярных выражений:

Таблица 3.1. Простые регулярные выражения

Регулярное выражение	Пояснение
`/Ruby/`	Соответствует одному слову *Ruby*
`/[Rr]uby/`	Соответствует *Ruby* или *ruby*
`/^abc/`	Соответствует *abc* в начале строки
`%r{xyz$}`	Соответствует *xyz* в конце строки
`%r|[0-9]*|`	Соответствует любой последовательности из нуля или более цифр

Сразу после регулярного выражения можно поместить однобуквенный модификатор. В таблице 3.2 приведены наиболее часто употребляемые модификаторы.

Таблица 3.2. Модификаторы регулярных выражений

Модификатор	Назначение
I	Игнорировать регистр
O	Выполнять подстановку выражения только один раз
M	Многострочный режим (точка сопоставляется с символом новой строки)
X	Обобщенное регулярное выражение (допускаются пробелы и комментарии)

Дополнительные примеры будут рассмотрены в главе 4.

Чтобы завершить введение в регулярные выражение, в таблице 3.3 мы приводим наиболее употребительные символы и обозначения.

Таблица 3.3. Общеупотребительные обозначения в регулярных выражениях

Обозначение	Пояснение
^	Начало строки текста (line) или строки символов (string)
$	Конец строки текста или строки символов
.	Любой символ, кроме символа новой строки (если не установлен многострочный режим)
\w	Символ-часть слова (цифра, буква или подчерк)
\W	Символ, не являющийся частью слова
\s	Пропуск (пробел, знак табуляции, символ новой строки и т.д.
\S	Символ, не являющийся пропуском
\d	Цифра (то же, что [0-9])
\D	Не-цифра
\h	Шестнадцатеричная цифра (то же, что [0-9a-f])
\H	Нешестнадцатеричная цифра
\A	Начало строки символов (string)
\Z	Конец строки символов или позиция перед конечным символом новой строки
\z	Конец строки символов (string)
\b	Граница слова (только вне [ ])
\B	Не граница слова
\b	Забой (только внутри [ ])
[]	Произвольный набор символов
*	0 или более повторений предыдущего подвыражения
*?	0 или более повторений предыдущего подвыражения (нежадный алгоритм)
+	1 или более повторений предыдущего подвыражения
+?	1 или более повторений предыдущего подвыражения (нежадный алгоритм)
{m,n}	От m до n вхождений предыдущего подвыражения
{m,n}?	От m до n вхождений предыдущего подвыражения (нежадный алгоритм)

Таблица 3.3 (окончнаие)

Обозначение	Пояснение
?	0 или 1 повторений предыдущего подвыражения
|	Альтернативы
(?=)	Позитивное заглядывание вперед
(?!)	Негативное заглядывание вперед
()	Группировка подвыражений
(?>)	Вложенное подвыражение
(?:)	Несохраняющая группировка подвыражений
(?imx-imx)	Включить/выключить режимы, начиная с этого места
(?imx-imx:expr)	Включить/выключить режимы для этого выражения
(?#)	Комментарий

Внутри классов символов в квадратных скобках можно использовать оператор && для комбинирования классов:

```
reg1 = /[a-z&&[^aeiou]]/ # любая буква, кроме гласных a, e, i, o, u
reg2 = /[a-z&&[^m-p]]/   # весь алфавит, кроме букв от m до p
```

Но такая нотация не всегда понятна, поэтому я не рекомендую злоупотреблять ею.

Умение работать с регулярными выражениями – большой плюс для современного программиста. Полное рассмотрение этой темы выходит далеко за рамки настоящей книги, но, если вам интересно, можете обратиться к книге Jeffrey Friedl «Mastering Regular Expressions»[1].

3.2. Компиляция регулярных выражений

Для компиляции регулярных выражений предназначен метод `Regexp.compile` (синоним `Regexp.new`). Первый параметр обязателен, он может быть строкой или регулярным выражением. (Отметим, что если этот параметр является регулярным выражением с дополнительными флагами, то флаги не будут перенесены в новое откомпилированное выражение.)

```
pat1 = Regexp.compile("^foo.*")  # /^foo.*/
pat2 = Regexp.compile(/bar$/i)   # /bar/ (i не переносится)
```

Если второй параметр задан, то обычно это поразрядное объединение (ИЛИ) каких-то из следующих констант: `Regexp::EXTENDED`, `Regexp::IGNORECASE`, `Regexp::MULTILINE`. При этом любое отличное от `nil` значение приведет к тому, что регулярное выражение не будет различать регистры; мы рекомендуем опускать второй параметр.

```
options = Regexp::MULTILINE || Regexp::IGNORECASE
```

[1] Фридл Дж. Регулярные выражения. СПб.: Питер, 2003. – *Прим. перев.*

```
pat3 = Regexp.compile("^foo", options)
pat4 = Regexp.compile(/bar/, Regexp::IGNORECASE)
```

Третий параметр, если он задан, включает поддержку многобайтных символов. Он может принимать одно из четырех значений:

```
"N" или "n" означает отсутствие поддержки
"E" или "e" означает EUC
"S" или "s" означает Shift-JIS
"U" или "u" означает UTF-8
```

Литеральное регулярное выражение можно задавать, и не вызывая метод new или compile. Достаточно заключить его в ограничители (символы косой черты).

```
pat1 = /^foo.*/
pat2 = /bar$/i
```

Более подробную информацию см. в главе 4.

3.3. Экранирование специальных символов

Метод класса Regexp.escape экранирует все специальные символы, встречающиеся в регулярном выражении. К их числу относятся звездочка, вопросительный знак и квадратные скобки.

```
str1 = "[*?]"
str2 = Regexp.escape(str1)  # "\[\*\?\]"
```

Синонимом является метод Regexp.quote.

3.4. Якоря

Якорь – это специальное выражение, соответствующее позиции в строке, а не конкретному символу или последовательности символов. Позже мы увидим, что это простой частный случай утверждения нулевой длины, то есть соответствия, которое не продвигает просмотр исходной строки ни на одну позицию.

Наиболее употребительные якоря уже были представлены в начале главы. Простейшими из них являются ^ и $, которые соответствуют началу и концу строки символов.

```
string  = "abcXdefXghi"
 def/  =~ string     # 4
 abc/  =~ string     # 0
 ghi/  =~ string     # 8
^def/  =~ string     # nil
def$/  =~ string     # nil
^abc/  =~ string     # 0
ghi$/  =~ string     # 8
```

Впрочем, я немного уклонился от истины. Эти якоря на самом деле соответствуют началу и концу не *строки символов* (string), а *строки текста* (line). Вот что произойдет, если те же самые образцы применить к строке, внутри которой есть символы новой строки:

```
string  = "abc\ndef\nghi"
/def/  =~ string     # 4
/abc/  =~ string     # 0
/ghi/  =~ string     # 8
/^def/ =~ string     # 4
/def$/ =~ string     # 4
/^abc/ =~ string     # 0
/ghi$/ =~ string     # 8
```

Однако имеются якоря \A и \Z, которые соответствуют именно началу и концу самой строки символов.

```
string  = "abc\ndef\nghi"
/\Adef/ =~ string    # nil
/def\Z/ =~ string    # nil
/\Aabc/ =~ string    # 0
/ghi\Z/ =~ string    # 8
```

Якорь \z отличается от \Z тем, что последний устанавливает соответствие *перед* конечным символом новой строки, а первый должен соответствовать явно.

```
string  = "abc\ndef\nghi"
str2 << "\n"
/ghi\Z/ =~ string    # 8
/\Aabc/ =~ str2      # 8
/ghi\z/ =~ string    # 8
/ghi\z/ =~ str2      # nil
```

Можно также устанавливать соответствие на *границе слова* с помощью якоря \b или с позицией, которая не находится на границе слова (\B). Примеры использования метода gsub показывают, как эти якоря работают:

```
str = "this is a test"
str.gsub(/\b/,"|")      # "|this| |is| |a| |test|"
str.gsub(/\B/,"-")      # "t-h-i-s i-s a t-e-s-t"
```

Не существует способа отличить начало слова от конца.

3.5. Кванторы

Немалая часть аппарата регулярных выражений связана с обработкой необязательных элементов и повторений. Элемент, за которым следует вопросительный знак, необязателен; он может присутствовать или отсутствовать, а общее соответствие зависит от прочих частей регулярного выражения. (Этот квантор имеет смысл применять только к подвыражению ненулевой длины, но не к якорям.)

```
pattern = /ax?b/
pat2    = /a[xy]?b/
pattern =~ "ab"     # 0
pattern =~ "acb"    # nil
pattern =~ "axb"    # 0
pat2    =~ "ayb"    # 0
pat2    =~ "acb"    # nil
```

Элементы часто повторяются неопределенное число раз (для формулировки этого условия служит квантор +). Например, следующий образец соответствует любому положительному целому числу:

```
pattern = /[0-9]+/
pattern =~ "1" # 0
pattern =~ "2345678"   # 0
```

Еще один типичный случай – образец, повторяющийся *нуль или более* раз. Конечно, это условие можно выразить с помощью кванторов + и ?. Вот, например, как сказать, что после строки Huzzah должно быть нуль или более восклицательных знаков:

```
pattern = /Huzzah(!+)?/   # Скобки здесь обязательны
pattern =~ "Huzzah"       # 0
pattern =~ "Huzzah!!!!"   # 0
```

Но есть и способ лучше. Требуемое поведение описывается квантором *.

```
pattern = /Huzzah!*/      # * применяется только к символу !
pattern =~ "Huzzah"       # 0
pattern =~ "Huzzah!!!!"   # 0
```

Как распознать американский номер социального страхования? С помощью такого образца:

```
ssn = "987-65-4320"
pattern = /\d\d\d-\d\d-\d\d\d\d/
pattern =~ ssn       # 0
```

Но это не вполне понятно. Лучше явно сказать, сколько цифр должно быть в каждой группе. Это можно сделать, указав число повторений в фигурных скобках:

```
pattern = /\d{3}-\d{2}-\d{4}/
```

Необязательно, что такой образец будет короче, но он более понятен читателю программы.

Можно также использовать диапазоны, границы которых разделены запятой. Предположим, что номер телефона в Элбонии состоит из двух частей: в первой может быть от трех до пяти цифр, а во второй – от трех до семи. Вот как выглядит соответствующий образец:

```
elbonian_phone = /\d{3,5}-\d{3,7}/
```

Нижняя и верхняя границы диапазона необязательны (но хотя бы одна должна быть задана):

```
/x{5}/        # Соответствует 5 x'ам
/x{5,7}/      # Соответствует 5-7 x'ам
/x{,8}/       # Соответствует не более 8 x'ам
/x{3,}/       # Соответствует по меньшей мере 3 x'ам
```

Ясно, что кванторы ?, + и * можно переписать и так:

```
/x?/          # то же, что /x{0,1}/
/x*/          # то же, что /x{0,}
/x+/          # то же, что /x{1,}
```

Фразеология, применяемая при описании регулярных выражений, изобилует яркими терминами: *жадный* (greedy), *неохотный* (reluctant), *ленивый* (lazy) и *собственнический* (possessive). Самым важным является различие между жадными и нежадными выражениями.

Рассмотрим следующий фрагмент кода. На первый взгляд, это регулярное выражение должно сопоставляться со строкой "Where the", но на самом деле ему соответствует более длинная подстрока "Where the sea meets the":

```
str = "Where the sea meets the moon-blanch'd land,"
match = /.*the/.match(str)
p match[0]  # Вывести полученное соответствие:
            # "Where the sea meets the"
```

Причина состоит в том, что оператор * выполняет жадное сопоставление, то есть продвигается так далеко по строке, как только можно, в поисках самого длинного соответствия. Чтобы излечить его от жадности, нужно добавить вопросительный знак:

```
str = "Where the sea meets the moon-blanch'd land,"
match = /.*?the/.match(str)
p match[0]  # Вывести полученное соответствие:
            # "Where the"
```

Итак, оператор * жадный, если за ним не стоит ?. То же самое относится к квантором + и {m,n} и даже к самому квантору ?.

Я не сумел найти разумных примеров применения конструкций {m,n}? и ??. Если вам о них известно, пожалуйста, поделитесь.

3.6. Позитивное и негативное заглядывание вперед

Понятно, что регулярное выражение сопоставляется со строкой линейно (осуществляя при необходимости возвраты). Поэтому существует понятие «текущей позиции» в строке, это аналог указателя файла или курсора.

Термин *заглядывание* означает попытку сопоставить часть строки, находящуюся дальше текущего положения. Это утверждение нулевой длины, поскольку даже если соответствие будет найдено, никакого продвижения по строке не произойдет (то есть текущая позиция не изменится).

В следующем примере строка "New World" будет сопоставлена, если за ней следует одна из строк "Symphony" или "Dictionary". Однако третье слово не будет частью соответствия.

```
s1 = "New World Dictionary"
s2 = "New World Symphony"
s3 = "New World Order"

reg = /New World(?= Dictionary| Symphony)/
m1 = reg.match(s1)
m.to_a[0]              # "New World"
m2 = reg.match(s2)
```

```
m.to_a[0]              # "New World"
m3 = reg.match(s3)     # nil
```

Вот пример негативного заглядывания:

```
reg2 = /New World(?! Symphony)/
m1 = reg.match(s1)
m.to_a[0]              # "New World"
m2 = reg.match(s2)
m.to_a[0]              # nil
m3 = reg.match(s3)     # "New World"
```

В данном случае строка "New World" подходит, только если за ней *не* следует строка "Symphony".

3.7. Позитивное и негативное оглядывание назад

Для тех, кому недостаточно заглядывания вперед, есть еще и *оглядывание назад* – проверка того, что текущей позиции предшествует указанный образец.

Как и многое из связанного с регулярными выражениями, понять эту идею и придумать ей применения нелегко. Спасибо Эндрю Джонсону за следующий пример.

Допустим, мы анализируем генетическую последовательность (молекула ДНК состоит из четырех азотистых оснований, обозначаемых A, C, G, T.) Предположим, что мы ищем все непересекающиеся цепочки нуклеотидов (длины 4), которые следуют за T. Мы не можем просто искать образец, состоящий из T и четырех последующих символов, потому что T, возможно, является последним символом предыдущего соответствия.

```
gene = 'GATTACAAACTGCCTGACATACGAA'
segs = gene.scan(/T(\w{4})/)
# найденные цепочки: [["TACA"], ["GCCT"], ["ACGA"]]
```

В этом примере мы пропустили цепочку GACA, которая следует за GCCT. Показанное ниже применение позитивного оглядывания позволяет найти все цепочки:

```
gene = 'GATTACAAACTGCCTGACATACGAA'
segs = gene.scan(/(?<=T)(\w{4})/)
# найденные цепочки: [["TACA"], ["GCCT"], ["GACA"], ["ACGA"]]
```

Следующий пример заимствован у К. Косако и немного модифицирован. Пусть имеется фрагмент текста в формате XML (или HTML) и мы хотим перевести в верхний регистр весь текст, не находящийся между тегами (то есть cdata). Вот как это можно сделать с помощью оглядывания:

```
text = <<-EOF
<body> <h1>This is a heading</h1>
<p> This is a paragraph with some
<i>italics</i> and some <b>boldface</b>
in it...</p>
</body>
```

```
EOF

pattern = /(?:^|          # В начале иди...
              (?<=>)      # после '>'
            )
            ([^<]*)       # Затем все символы, отличные от '<' (запоминаем)
           /x

puts text.gsub(pattern) {|s| s.upcase }

# Выводится:
# <body> <h1>THIS IS A HEADING</h1>
# <p>THIS IS A PARAGRAPH WITH SOME
# <i>ITALICS</i> AND SOME <b>BOLDFACE</b>
# IN IT...</p>
# </body>
```

3.8. Обратные ссылки

Каждая заключенная в круглые скобки часть регулярного выражения является отдельным соответствием. Они нумеруются, и есть несколько способов сослаться на такие части по номерам. Сначала рассмотрим традиционный «некрасивый» способ.

Сослаться на группы можно с помощью глобальных переменных $1, $2 и т.д:

```
str = "a123b45c678"
if /(a\d+)(b\d+)(c\d+)/ =~ str
  puts "Частичные соответствия: '#$1', '#$2', '#$3'"
  # Печатается: Частичные соответствия: 'a123', 'b45', 'c768'
end
```

Эти переменные ***нельзя*** использовать в подставляемой строке в методах sub и gsub:

```
str = "a123b45c678"
str.sub(/(a\d+)(b\d+)(c\d+)/, "1st=#$1, 2nd=#$2, 3rd=#$3")
# "1st=, 2nd=, 3rd="
```

Почему такая конструкция не работает? Потому что аргументы sub вычисляются перед вызовом sub. Вот эквивалентный код:

```
str = "a123b45c678"
s2 = "1st=#$1, 2nd=#$2, 3rd=#$3"
reg = /(a\d+)(b\d+)(c\d+)/
str.sub(reg,s2)
# "1st=, 2nd=, 3rd="
```

Отсюда совершенно понятно, что значения $1, $2, $3 ***никак не связаны*** с сопоставлением, которое делается внутри вызова sub.

В такой ситуации на помощь приходят специальные коды \1, \2 и т.д.:

```
str = "a123b45c678"
str.sub(/(a\d+)(b\d+)(c\d+)/, '1st=\1, 2nd=\2, 3rd=\3')
# "1st=a123, 2nd=b45, 3rd=c768"
```

Обратите внимание на одиночные (твердые) кавычки в предыдущем примере. Если бы мы воспользовались двойными (мягкими) кавычками, не приняв никаких мер предосторожности, то элементы, которым предшествует обратная косая черта, были бы интерпретированы как восьмеричные числа:

```
str = "a123b45c678"
str.sub(/(a\d+)(b\d+)(c\d+)/, "1st=\1, 2nd=\2, 3rd=\3")
# "1st=\001, 2nd=\002, 3rd=\003"
```

Обойти эту неприятность можно за счет двойного экранирования:

```
str = "a123b45c678"
str.sub(/(a\d+)(b\d+)(c\d+)/, "1st=\\1, 2nd=\\2, 3rd=\\3")
# "1st=a123, 2nd=b45, 3rd=c678"
```

Допустима и блочная форма подстановки, в которой можно использовать глобальные переменные:

```
str = "a123b45c678"
str.sub(/(a\d+)(b\d+)(c\d+)/)  { "1st=#$1, 2nd=#$2, 3rd=#$3" }
# "1st=a123, 2nd=b45, 3rd=c678"
```

При таком применении блока числа с обратной косой чертой ***нельзя*** использовать ни в двойных, ни в одиночных кавычках. Немного поразмыслив, вы поймете, что это разумно.

Упомяну попутно о том, что существуют незапоминаемые группы (noncapturing groups). Иногда при составлении регулярного выражения нужно сгруппировать символы, но чему будет соответствовать в конечном итоге такая группа, несущественно. На этот случай и предусмотрены незапоминаемые группы, описываемые синтаксической конструкцией `(?:...)`:

```
str = "a123b45c678"
str.sub(/(a\d+)(?:b\d+)(c\d+)/, "1st=\\1, 2nd=\\2, 3rd=\\3")
# "1st=a123, 2nd=c678, 3rd="
```

В предыдущем примере вторая группа не запоминается, поэтому та группа, которая должна была бы быть третьей, становится *второй*.

Лично мне не нравится ни одна из двух нотаций (`\1` и `$1`). Иногда они удобны, но никогда не бывают необходимы. Все можно сделать «красивее», в объектно-ориентированной манере.

Метод класса `Regexp.last_match` возвращает объект класса `MatchData` (как и метод экземпляра `match`). У этого объекта есть методы экземпляра, с помощью которых программист может получить обратные ссылки.

Обращаться к объекту `MatchData` можно с помощью квадратных скобок, как если бы это был массив соответствий. Специальный элемент с индексом 0 содержит текст всей сопоставляемой строки, а элемент с индексом n ссылается на *n*-ую запомненную группу:

```
pat = /(.+[aiu])(.+[aiu])(.+[aiu])(.+[aiu])/i
# В этом образце есть четыре одинаковых группы
refs = pat.match("Fujiyama")
# Теперь refs равно: ["Fujiyama","Fu","ji","ya","ma"]
x = refs[1]
y = refs[2..3]
refs.to_a.each {|x| print "#{x}\n"}
```

Отметим, что объект refs – не настоящий массив. Поэтому, если мы хотим обращаться с ним как с таковым, применяя итератор each, то нужно сначала преобразовать его в массив с помощью метода to_a (как показано в примере).

Есть и другие способы нахождения сопоставленной подстроки внутри исходной строки. Методы begin и end возвращают смещения начала и конца соответствия. (Важно понимать, что смещение конца – это индекс символа, следующего за найденным соответствием.)

```
str = "alpha beta gamma delta epsilon"
#      0....5....0....5....0....5....
#      (для удобства подсчета)

pat = /(b[^ ]+ )(g[^ ]+ )(d[^ ]+ )/
# Три слова, каждое из которых представляет собой отдельное соответствие
refs = pat.match(str)

# "beta "
p1 = refs.begin(1)          # 6
p2 = refs.end(1)            # 11
# "gamma "
p3 = refs.begin(2)          # 11
p4 = refs.end(2)            # 17
# "delta "
p5 = refs.begin(3)          # 17
p6 = refs.end(3)            # 23
# "beta gamma delta"
p7 = refs.begin(0)          # 6
p8 = refs.end(0)            # 23
```

Аналогично, метод offset возвращает массив из двух чисел: смещение начала и смещение конца соответствия. Продолжая предыдущий пример:

```
range0 = refs.offset(0)     # [6,23]
range1 = refs.offset(1)     # [6,11]
range2 = refs.offset(2)     # [11,17]
range3 = refs.offset(3)     # [17,23]
```

Части строки, которые находятся перед и после сопоставленной подстроки, можно получить методами pre_match и post_match соответственно. В том же примере:

```
before = refs.pre_match     # "alpha "
after  = refs.post_match    # "epsilon"
```

3.9. Именованные соответствия

Существует особый вид подвыражений, которые называются *именованными* выражениями. Идея в том, чтобы сопоставить образцу имя (а не просто номер).

Синтаксис простой: (?<name>expr), где name – некоторое имя, начинающееся с буквы (как идентификатор в Ruby). Отметим несомненное сходство с неименованным атомарным подвыражением.

Что можно делать с именованным выражением? Ну, например, использовать его в качестве обратной ссылки.

Ниже приведен пример простого регулярного выражения, которому соответствует удвоенное слово (см. раздел 3.15.6 «Поиск удвоенных слов в тексте»):

```
re1 = /\s+(\w+)\s+\1\s+/
str = "Now is the the time for all..."
re1.match(str).to_a    # ["the the","the"]
```

Обратите внимание: мы запоминаем слово, а затем ссылаемся на него с помощью \1. Именованные ссылки можно использовать точно так же. Мы присваиваем подвыражению имя при первом использовании, а затем формируем обратную ссылку на него: маркер \k, за которым следует присвоенное имя (обязательно в угловых скобках):

```
re2 = /\s+(?<anyword>\w+)\s+\k<anyword>\s+/
```

Второй вариант длиннее, но, как мне кажется, понятнее. (Хочу предостеречь – в одном и том же регулярном выражении *нельзя* одновременно использовать нумерованные и именованные обратные ссылки.) Каким механизмом пользоваться, решайте сами.

В Ruby давно существовала возможность использовать обратные ссылки в строках, передаваемых методам sub и gsub, но раньше она распространялась только на нумерованные обратные ссылки, а в самых последних версиях обобщена и на именованные:

```
str = "I breathe when I sleep"

# Нумерованные соответствия ...
r1 = /I (\w+) when I (\w+)/
s1 = str.sub(r1,'I \2 when I \1')

# Именованные соответствия ...
r1 = /I (?<verb1>\w+) when I (?<verb2>\w+)/
s2 = str.sub(r2,'I \k<verb2> when I \k<verb1>')

puts s1    # I sleep when I breathe
puts s2    # I sleep when I breathe
```

Еще одно применение именованных выражений – *повторное использование* выражения. В таком случае имени должно предшествовать \g (а не \k).

Например, определим образец spaces, так чтобы его можно было использовать повторно. Предыдущее регулярное re2 тогда принимает вид:

```
re3 = /(?<spaces>\s+)(?<anyword>\w+)\g<spaces>\k<anyword>\g<spaces>/
```

Определенный таким образом образец можно употреблять многократно с помощью маркера \g. Эта возможность обретает куда больший смысл в случае рекурсивных регулярных выражений; этой теме посвящен следующий раздел.

Когда именованных подвыражений нет, можно использовать нотацию вида \g<1>. В этом случае повторно используется запомненное подвыражение с указанным номером (а не именем).

И последнее замечание по поводу именованных соответствий. Имя можно использовать (в виде символа или строки) в качестве индекса MatchData. Ниже приведен соответствующий пример:

```
str = "My hovercraft is full of eels"
reg = /My (?<noun>\w+) is (?<predicate>.*)/
m = reg.match(str)
puts m[:noun]          # hovercraft
puts m["predicate"]    # full of eels
puts m[1]              # то же, что m[:noun] или m["noun"]
```

3.10. Классы символов

Классы символов – это просто форма перечисления (задание альтернатив), в котором каждая группа состоит ровно из одного символа. В простейшем случае список возможных символов заключается в квадратные скобки:

```
/[aeiou]/      # Соответствует любой из букв a, e, i, o, u; эквивалентно
               # /(a|e|i|o|u)/, только группа не запоминается
```

Внутри класса символов управляющие последовательности типа \n по-прежнему распознаются, но такие метасимволы, как . и ?, не имеют специального смысла:

```
/[.\n?]/       # Сопоставляется с точкой, символом новой строки,
               # вопросительным знаком
```

Символ каре ^ внутри класса символов имеет специальный смысл, если находится в начале; в этом случае он формирует дополнение к списку символов:

```
[^aeiou]       # Любой символ, Кроме EXCEPT a, e, i, o, u
```

Дефис внутри класса символов обозначает диапазон (в лексикографическом порядке):

```
/[a-mA-M]/     # Любой символ из первой половины алфавита
/[^a-mA-M]/    # Любой ДРУГОЙ символ, а также цифры и символы, отличные
               # от букв и цифр
```

Дефис в начале или в конце класса символов, а также каре в середине теряют специальный смысл и интерпретируются буквально. То же относится к левой квадратной скобке, но правая квадратная скобка, очевидно, должна экранироваться:

```
/[-^[\]]/      # Сопоставляется с дефисом, каре и правой квадратной скобкой
```

Регулярные выражения в Ruby могут содержать ссылки на именованные классы символов вида [[:name:]]. Так, [[:digit:]] означает то же самое, что образец

[0-9]. Во многих случаях такая запись оказывается короче или, по крайней мере, понятнее.

Есть еще такие именованные классы: [[:print:]] (символы, имеющие графическое начертание) и [[:alpha:]] (буквы):

```
s1 = "abc\007def"
 [[:print:]]*/.match(s1)
m1 = Regexp::last_match[0]                # "abc"

s2 = "1234def"
 [[:digit:]]*/.match(s2)
m2 = Regexp::last_match[0]                # "1234"

 [[:digit:]]+[[:alpha:]]/.match(s2)
m3 = Regexp::last_match[0]                # "1234d"
```

Каре перед именем класса символов формирует его дополнение:

```
[[:^alpha:]]/   # Все символы, кроме букв
```

Для многих классов имеется также сокращенная нотация. Наиболее распространены сокращения \d (любая цифра), \w (любой символ, входящий в состав «слова») и \s (пропуски – пробел, знак табуляции или новой строки):

```
str1 = "Wolf 359"
 \w+/.match(str1)      # соответствует "Wolf" (то же, что /[a-zA-Z_0-9]+/)
 \w+ \d+/.match(str1)  # соответствует "Wolf 359"
 \w+ \w+/.match(str1)  # соответствует "Wolf 359"
 \s+/.match(str1)      # соответствует " "
```

«Дополнительные» формы обычно записываются в виде заглавной буквы:

```
\W/     # Любой символ, не входящий в состав слова
\D/     # Все, кроме цифр
\S/     # Все, кроме пропусков
```

3.11. Обобщенные регулярные выражения

Регулярные выражения, особенно длинные, часто выглядят загадочно. Модификатор x позволяет записывать регулярное выражение на нескольких строках. При этом пробелы и символы новой строки игнорируются, так что можно делать для наглядности отступы. Заодно разрешается оставлять комментарии, хотя это возможно даже в простых регулярных выражениях.

Чтобы привести несколько искусственный пример умеренно сложного регулярного выражения, предположим, что имеется такой список адресов:

```
addresses =
  [ "409 W Jackson Ave",           "No. 27 Grande Place",
    "16000 Pennsylvania Avenue",   "2367 St. George St.",
    "22 Rue Morgue",               "33 Rue St. Denis",
    "44 Rue Zeeday",               "55 Santa Monica Blvd.",
    "123 Main St., Apt. 234",      "123 Main St., #234",
    "345 Euneva Avenue, Suite 23", "678 Euneva Ave, Suite A"]
```

Здесь каждый адрес состоит из трех частей: номер дома, название улицы и необязательный номер квартиры. Я предполагаю, что перед числом может быть необязательная строка `No.`, а точку в ней можно опускать. Еще предположим, что название улицы может включать символы, обычно входящие в состав слова, а также апостроф, дефис и точку. Наконец, если адрес содержит необязательный номер квартиры, то ему должна предшествовать запятая и одна из строк `Apt.`, `Suite` или `#` (знак номера).

Вот какое регулярное выражение я составил для разбора адреса. Обратите внимание, насколько подробно оно прокомментировано (может быть, даже излишне подробно):

```
regex = / ^                    # Начало строки
          ((No\.?)\s+)?        # Необязательно: No[.]
          \d+ \s+              # Цифры и пробелы
          ((\w|[.'-])+         # Название улицы... может
           \s*                 #   состоять из нескольких слов.
          )+
          (,\s*                # Необязательно: запятая и т.д.
           (Apt\.?|Suite|\#)   # Apt[.], Suite, #
           \s+                 # Пробелы
           (\d+|[A-Z])         # Цифры или одна буква
          )?
          $                    # Конец строки
        /x
```

Идея понятна. Когда сложность регулярного выражения достигает некоего порога (какого именно, дело вкуса), делайте его обобщенным, чтобы можно было добавить форматирование и комментарии.

Возможно, вы заметили, что я пользовался обычными комментариями Ruby (`# ...`), а не специальными комментариями, применяемыми в регулярных выражениях (`(?#...)`). Почему? Просто потому, что это разрешено. Специальный комментарий необходим только тогда, когда его следует закончить раньше конца строки (например, если в той же строке за комментарием продолжается регулярное выражение).

3.12. Сопоставление точки символу конца строки

Обычно точка соответствует любому символу, кроме конца строки. Если задан модификатор многострочности `m`, то точка будет сопоставляться и с этим символом. Другой способ – задать флаг `Regexp::MULTILINE` при создании регулярного выражения:

```
str = "Rubies are red\nAnd violets are blue.\n"
pat1 = /red./
pat2 = /red./m

str =~ pat1       # nil
str =~ pat2       # 11
```

Этот режим не оказывает влияния на то, где устанавливается соответствие якорям (^, $, \A, \Z). Изменяется только способ сопоставления с точкой.

3.13. Внутренние модификаторы

Обычно модификаторы (например, i или m) задаются после регулярного выражения. Но что, если мы хотим применить модификатор только к части выражения?

Существует специальная нотация для включения и выключения модификаторов. Заключенный в круглые скобки вопросительный знак, за которым следует один или несколько модификаторов, «включает» их до конца регулярного выражения. А если некоторым модификаторам предшествует минус, то соответствующие режимы «выключаются»:

```
/abc(?i)def/       # Соответствует abcdef, abcDEF, abcDef, ...
                   #   но не ABCdef
/ab(?i)cd(?-i)ef/  # Соответствует abcdef, abCDef, abcDef, ...
                   #   но не ABcdef или abcdEF
/(?imx).*/         # То же, что /.*/imx
/abc(?i-m).*/m     # Для последней части регулярного выражения включить
                   #   распознавание регистра, выключить многострочный
                   #   режим
```

При желании можно поставить перед подвыражением двоеточие, и тогда заданные модификаторы будут действовать только для этого подвыражения:

```
/ab(?i:cd)ef/        # То же, что /ab(?i)cd(?-i)ef/
```

По техническим причинам, использовать таким образом модификатор o нельзя. Модификатор x можно, но я не знаю, кому бы это могло понадобиться.

3.14. Внутренние подвыражения

Для задания подвыражений применяется нотация ?>:

```
re = /(?>abc)(?>def)/    # То же, что /abcdef/
re.match("abcdef").to_a  # ["abcdef"]
```

Отметим, что наличие подвыражения еще не означает группировки. С помощью дополнительных скобок их, конечно, можно превратить в запоминаемые группы.

Еще обратим внимание на то, что эта конструкция *собственническая*, то есть жадная и при этом *не* допускает возврата в подвыражение.

```
str = "abccccdef"
re1 = /(abc*)cdef/
re2 = /(?>abc*)cdef/
re1 =~ str  # 0
re2 =~ str  # nil
re1.match(str).to_a # ["abccccdef", "abccc"]
re2.match(str).to_a # []
```

В предыдущем примере подвыражение abc* выражения re2 поглощает все вхождения буквы c и (в соответствии с собственническим инстинктом) не отдает их назад, препятствуя возврату.

Помимо (?>), существует еще один способ выразить идею собственничества: с помощью постфиксного квантора +. Это употребление отличается от семантики «один или больше», и оба могут сочетаться. На самом деле, это «добавочный» квантор – как ? (допускающий комбинации ??, +? и *?).

По существу, применение + к повторяющемуся образцу эквивалентно оформлению этого образца в виде независимого подвыражения. Вот пример:

```
r1 = /x*+/    # То же, что /(?>x*)/
r2 = /x++/    # То же, что /(?>x+)/
r3 = /x?+/    # То же, что /(?>x?)/
```

По техническим причинам, нотация {n,m}+ не считается в Ruby собственнической.

3.14.1. Рекурсия в регулярных выражениях

Возможность повторно обращаться к подвыражению позволяет создавать рекурсивные регулярные выражения. Например, следующее находит любое вложенное выражение с правильно расставленными скобками (еще раз спасибо Эндрю Джексону):

```
str = "a * ((b-c)/(d-e) - f) * g"

reg = /(?                    # начало именованного выражения
        \(                   # открывающая круглая скобка
        (?:                  # незапоминаемая группа
          (?>                # сопоставление с собственническим выражением:
            \\[()]           #  экранированная скобка
           |                 # ЛИБО
            [^()]            #  вообще не скобка
          )                  # конец собственнического выражения
          |                  # ЛИБО
          \g                 # вложенная группа в скобках (рекурсивный вызов)
        )*                   # незапоминаемая группа повторяется нуль или
                             #  более раз
        \)                   # закрывающая круглая скобка
      )                      # конец именованного выражения
    /x
m = reg.match(str).to_a      # ["((b-c)/(d-e) - f)", "((b-c)/(d-e) - f)"]
```

Отметим, что *левосторонняя рекурсия* запрещена. Следующий пример допустим:

```
str = "bbbaccc"
re1 = /(?<foo>a|b\g<foo>c)/
re1.match(str).to_a        # ["bbbaccc","bbbaccc"]
```

А такой – нет:

```
re2 = /(?<foo>a|\g<foo>c)/ # Синтаксическая ошибка!
```

Ошибка объясняется наличием рекурсивного обращения в начале каждой альтернативы. Немного подумав, вы поймете, что это приведет к бесконечному возврату.

3.15. Примеры регулярных выражений

В этом разделе мы приведем краткий перечень регулярных выражений, которые могут оказаться полезны на практике или просто послужат учебными примерами.

3.15.1. Сопоставление с IP-адресом

Пусть мы хотим понять, содержит ли строка допустимый IPv4-адрес. Стандартно он записывается в *точечно-десятичной нотации*, то есть в виде четырех десятичных чисел, разделенных точками, причем каждое число должно находиться в диапазоне от 0 до 255.

Приведенный ниже образец решает эту задачу (за немногими исключениями типа «127.1»). Для удобства восприятия мы разобьем его на части. Отметим, что символ \d дважды экранирован, чтобы косая черта не передавалась из строки в регулярное выражение (чуть ниже мы решим и эту проблему).

```
num = "(\\d|[01]?\\d\\d|2[0-4]\\d|25[0-5])"
pat = "^(#{num}\.){3}#{num}$"
ip_pat = Regexp.new(pat)

ip1 = "9.53.97.102"

if ip1 =~ ip_pat                # Печатается "да"
  puts "да"
else
  puts "нет"
end
```

Надо признать, что в определении переменной num слишком много символов обратной косой черты. Определим ее в виде регулярного выражения, а не строки:

```
num = /(\d|[01]?\d\d|2[0-4]\d|25[0-5])/
```

Когда одно регулярное выражение интерполируется в другое, вызывается метод to_s, который сохраняет всю информацию из исходного регулярного выражения.

```
num.to_s    # "(?-mix:(\\d|[01]?\\d\\d|2[0-4]\\d|25[0-5]))"
```

Иногда для встраивания удобно использовать регулярное выражение, а не строку. Хорошее эвристическое правило: интерполируйте регулярные выражения, если нет веских причин интерполировать строки.

IPv6-адреса пока не очень широко распространены, но для полноты рассмотрим и их тоже. Они записываются в виде восьми 16-ричных чисел, разделенных двоеточиями, с подавлением начальных нулей.

```
num = /[0-9A-Fa-f]{0,4}/
pat = /^(#{num}:){7}#{num}$/
```

```
ipv6_pat = Regexp.new(pat)

v6ip = "abcd::1324:ea54::dead::beef"

if v6ip =~ ipv6_pat     # Печатается "да"
  puts "да"
else
  puts "нет"
end
```

3.15.2. Сопоставление с парой «ключ-значение»

Иногда приходится работать со строками вида «ключ=значение» (например, при разборе конфигурационного файла приложения).

Следующий код извлекает ключ и значение. Предполагается, что ключ состоит из одного слова, значение продолжается до конца строки, а знак равенства может быть окружен пробелами:

```
pat = /(\w+)\s*=\s*(.*?)$/
str = "color = blue"

matches = pat.match(str)

puts matches[1]         # "color"
puts matches[2]         # "blue"
```

3.15.3. Сопоставление с числами, записанными римскими цифрами

Следующее довольно сложное регулярное выражение сопоставляется с любым правильно записанным римскими цифрами числом (до 3999 включительно). Как и раньше, для удобства восприятия образец разбит на части:

```
rom1 = /m{0,3}/i
rom2 = /(d?c{0,3}|c[dm])/i
rom3 = /(l?x{0,3}|x[lc])/i
rom4 = /(v?i{0,3}|i[vx])/i
roman = /^#{rom1}#{rom2}#{rom3}#{rom4}$/

year1985 = "MCMLXXXV"

if year1985 =~ roman      # Печатается "да"
  puts "да"
else
  puts "нет"
end
```

Возможно, у вас появилось искушение поставить в конец всего выражения модификатор i, чтобы сопоставлялись и строчные буквы:

```
# Это не работает!

rom1 = /m{0,3}/
```

```
rom2 = /(d?c{0,3}|c[dm])/
rom3 = /(l?x{0,3}|x[lc])/
rom4 = /(v?i{0,3}|i[vx])/
roman = /^#{rom1}#{rom2}#{rom3}#{rom4}$/i
```

Почему такое выражение не годится? Взгляните на этот пример и поймете:

```
rom1.to_s   # "(?-mix:m{0,3})"
```

Обратите внимание, что метод to_s запоминает флаги для каждого выражения, тем самым флаг всего выражения перекрывается.

3.15.4. Сопоставление с числовыми константами

Сопоставление с простым целым десятичным числом – самое простое. Число состоит из необязательного знака и последовательности цифр (правда, Ruby позволяет использовать подчерк в качестве разделителя цифр). Отметим, что первая цифра не должна быть нулем, иначе число будет интерпретироваться как восьмеричное.

```
int_pat = /^[+-]?[1-9][\d_]*$/
```

Целые константы в других системах счисления обрабатываются аналогично. Образцы для шестнадцатеричных и двоичных чисел сделаны не чувствительными к регистру, так как они содержат букву:

```
hex_pat = /^[+-]?0x[\da-f_]+$/i
oct_pat = /^[+-]?0[0-7_]+$/
bin_pat = /^[+-]?0b[01_]+$/i
```

Сопоставить число с плавающей точкой в обычной нотации несколько сложнее. Последовательности цифр по обе стороны десятичной точки необязательны, но хотя бы одна цифра должна быть:

```
float_pat = /^(\d[\d_]*)*\.[\d_]*$/
```

Образец для чисел, записанных в научной нотации, основан на предыдущем:

```
sci_pat = /^(\d[\d_]*)?\.[\d_]*(e[+-]?)?(_*\d[\d_]*)$/i
```

Эти образцы могут оказаться полезны, если вы хотите убедиться, что строка содержит число, перед тем как пытаться преобразовать ее.

3.15.5. Сопоставление с датой и временем

Пусть надо выделить дату и время, записанные в формате mm/dd/yy hh:mm:ss. Вот первая попытка: datetime = /(\d\d)\/(\d\d)\/(\d\d) (\d\d):(\d\d):(\d\d)/.

Но такой образец распознает некоторые некорректные даты и отвергает правильные. Следующий вариант более избирателен. Обратите внимание, как мы строим его путем интерполяции мелких регулярных выражений в более крупное:

```
mo = /(0?[1-9]|1[0-2])/          # от 01 до 09 или от1 до 9 или 10-12
dd = /([0-2]?[1-9]|[1-3][01])/   # 1-9 или 01-09 или 11-19 и т.д.
yy = /(\d\d)/                    # 00-99
hh = /([01]?[1-9]|[12][0-4])/    # 1-9 или 00-09 или...
mm = /([0-5]\d)/                 # 00-59, обе цифры должны присутствовать
```

```
ss = /([0-6]\d)?/                    # разрешены еще и доли секунды ;-)

date = /(#{mo}\/#{dd}\/#{yy})/
time = /(#{hh}:#{mi}:#{ss})/

datetime = /(#{date} #{time})/
```

Вот как можно вызвать это регулярное выражение из метода String#scan, чтобы получить массив соответствий:

```
str="Recorded on 11/18/07 20:31:00"
str.scan(datetime)
# [["11/18/07 20:31:00", "11/18/07", "11", "18", "00",
#   "20:31:00", "20", "31", ":00"]]
```

Разумеется, все это можно было сделать с помощью одного большого регулярного выражения:

```
datetime = %r{(
  (0?[1-9]|1[0-2])/          # mo: от 01 до 09 или от 1 до 9 или 10-12
  ([0-2]?[1-9]|[1-3][01])/ # dd: 1-9 или 01-09 или 11-19 и т.д.
  (\d\d) [ ]                 # yy: 00-99
  ([01]?[1-9]|[12][0-4]):  # hh: 1-9 или 00-09 или...
  ([0-5]\d):                 # mm: 00-59, обе цифры должны присутствовать
  (([0-6]\d))?               # ss: допускает високосные секунды ;-)
)}x
```

Обратите внимание на конструкцию %r{}, позволяющую не экранировать символы обратной косой черты.

3.15.6. Обнаружение удвоенных слов в тексте

В этом разделе мы реализуем детектор удвоенных слов. Повторение одного и того же слова два раза подряд – типичная опечатка. Следующий код распознает такие ситуации:

```
double_re = /\b(['A-Z]+) +\1\b/i

str="There's there's the the pattern."
str.scan(double_re)  #  [["There's"],["the"]]
```

Обратите внимание на модификатор i в конце выражения – он позволяет проводить сопоставление без учета регистра. Каждой группе соответствует массив, поэтому в результате получается массив массивов.

3.15.7. Поиск слов, записанных одними заглавными буквами

Мы упростили пример, предположив, что в тексте нет чисел, подчерков и т.д.

```
allcaps = /\b[A-Z]+\b/

string = "This is ALL CAPS"
string[allcaps]  #  "ALL"
Suppose you want to extract every word in all-caps:
string.scan(allcaps)  #  ["ALL", "CAPS"]
```

При желании можно было бы обобщить эту идею на идентификаторы Ruby и аналогичные вещи.

3.15.8. Сопоставление с номером версии

Принято присваивать библиотекам и приложениям номера версий, состоящие из трех чисел, разделенных точками. Следующее регулярное выражение сопоставляется с подобной строкой, выделяя попутно имя пакета и отдельные части номера версии:

```
package = "mylib-1.8.12"
matches = package.match(/(.*)-(\d+)\.(\d+)\.(\d+)/)
name, major, minor, tiny = matches[1..-1]
```

3.15.9. Еще несколько образцов

Завершим наш список несколькими выражениями из категории «разное». Как обычно, почти все эти задачи можно решить несколькими способами.

Пусть нужно распознать двузначный почтовый код американского штата. Проще всего, конечно, взять выражение /[A-Z]{2}/. Но оно сопоставляется с такими строками, как XX или ZZ, которые допустимы, но бессмысленны. Следующий образец распознает все 51 стандартных аббревиатур (50 штатов и DC – округ Колумбия):

```
state =  /^A[LKZR] | C[AOT] | D[EC] | FL | GA | HI | I[DLNA] |
          K[SY] | LA | M[EDAINSOT] | N[EVHJMYCD] | O[HKR] |
          PA | RI | S[CD] | T[NX] | UT | V[TA] | W[AVIY]$/x
```

Для ясности я воспользовался обобщенным регулярным выражением (модификатор x). Пробелы и символы новой строки в нем игнорируются.

Продолжая эту тему, приведем регулярное выражение для распознавания почтового индекса США (он может состоять из пяти или девяти цифр):

```
zip = /^\d{5}(-\d{4})?$/
```

Якоря (в этом и других выражениях) призваны лишь гарантировать, что ни до, ни после сопоставленной строки никаких лишних символов нет. Отметим, что это выражение не отбрасывает несуществующие индексы, поэтому оно не так полезно, как предыдущее.

Следующее регулярное выражение распознает номер телефона в формате NANP (североамериканский план нумерации). Есть три способа записи такого номера:

```
phone = /^((\(\d{3}\) |\d{3}-)\d{3}-\d{4}|\d{3}\.\d{3}\.\d{4})$/
"(512) 555-1234" =~ phone   # true
"512.555.1234"   =~ phone   # true
"512-555-1234"   =~ phone   # true
"(512)-555-1234" =~ phone   # false
"512-555.1234"   =~ phone   # false
```

Распознать денежную сумму в долларах также тривиально:

```
dollar = /^\$\d+(\.\d\d)?$/
```

Ясно, что слева от десятичной точки должна быть хотя бы одна цифра, а после знака доллара не должно быть пробелов. Отметим еще, что если вы хотите только выделить, а не проконтролировать суммы в долларах, то якоря следовало бы удалить, а центы сделать необязательными.

3.16. Заключение

На этом мы заканчиваем обсуждение регулярных выражений в Ruby. Познакомившись со строками и регулярными выражениями, мы можем перейти к теме интернационализации, где пригодится рассмотренный выше материал.

Глава 4. Интернационализация в Ruby

Посему дано ему имя: Вавилон,
ибо там смешал Господь язык всей земли,
и оттуда рассеял их Господь по всей земле.

– Бытие, 11:9

Мы уже говорили, что тип символа, наверное, самый важный из всех. Но что такое символьные данные? Какие символы? Из какого алфавита? Какой язык? Какие культурные особенности?

В прошлом в вычислительной технике и информатике применялся исключительно английский язык. Эта традиция восходит, вероятно, еще к Чарльзу Беббиджу. Это не так уж и плохо, ведь надо же с чего-то начинать, а алфавит из 26 букв без диакритических знаков – неплохое начало.

Но теперь компьютеры распространились повсеместно. Наверное, в каждой стране есть хотя бы несколько компьютеров и тот или иной вид доступа в Сеть. Естественно, любой человек предпочитает читать веб-страницы, электронную почту и прочие данные на своем родном языке, а не только на английском.

Человеческие языки поразительно разнообразны. Некоторые являются почти фонетическими, к другим это определение применимо с большой натяжкой. В одних есть настоящий алфавит, другие – это набор из нескольких тысяч символов, ведущих происхождение от пиктограмм. В некоторых языках более одного алфавита. На каких-то языках пишут сверху вниз, на других – по горизонтали, но не слева направо, как большинство из нас, а справа налево. Некоторые алфавиты очень просты, в других буквы украшены приводящими в трепет точечками, черточками, кружочками, линиями и штрихами. Есть языки, в которых при определенных условиях соседние буквы комбинируются; иногда это обязательно, а иногда и нет. В каких-то языках имеется различие между строчными и заглавными буквами, но в большинстве такого различия нет.

За 30 лет мы прошли длинный путь. Мы худо-бедно научились приводить в порядок хаотическое нагромождение символов и языков. Если вам часто приходится иметь дело с программами, спроектированными для работы в различной языковой среде, то вы знаете, что такое *интернационализация*. Это способность программы поддерживать более одного естественного языка.

С интернационализацией тесно связаны *многоязычность* и *локализация*. Почему-то принято сокращать эти слова, удаляя средние буквы и подставляя вместо них число, равное количеству удаленных букв:

```
def shorten(str)
  (str[0..0] + str[1..-2].length.to_s + str[-1..-1]).upcase
end

shorten("internationalization")        # I18N
shorten("multilingualization")         # M17N
shorten("localization")                # L10N
```

Локализация подразумевает полную поддержку местных соглашений, в том числе: обозначения валюты, способы форматирования даты и времени, использование запятой в качестве разделителя десятичных разрядов и многое другое. Общепринятого мнения по вопросу о том, что означает многоязычность, не существует. Многие, даже большинство, понимают под этим комбинацию I18N и L10N; другие считают, что это синоним I18N. Я стараюсь не употреблять этот термин в настоящей книге.

По сути дела, интернационализация приложения – одноразовый процесс: исходный текст необходимо модифицировать, так чтобы он мог поддерживать разные кодировки и данные на разных языках. Полностью интернационализированная программа способна принимать, обрабатывать и выводить строки на разных языках.

Локализованная программа этим не ограничивается, она позволяет во время выполнения выбирать язык, на котором будут выводиться приглашения, сообщения об ошибках и прочее. Подразумевается также, что в сообщения будут подставлены подходящие данные, а формы множественного числа образуются правильно. Сюда же относится умение правильно сортировать строки, а также форматирование чисел, даты, времени и денежных величин в соответствии с местными обычаями.

Разумеется, процедура локализации включает перевод сообщений об ошибках на все поддерживаемые языки.

В этой главе мы рассмотрим инструменты и методы, упрощающие решение описанных задач в Ruby. Начнем с терминологии, поскольку в этой области широко используется жаргон.

4.1. Исторические сведения и терминология

Когда-то существовало множество кодировок. В начале 1960-х годов была разработана кодировка ASCII, получившая широкое распространение в 1970-х и первой половине 1980-х. Аббревиатура ASCII означает American Standard Code for Information Interchange (Американский стандартный код обмена информацией). Это был большой шаг вперед, но ключевое слово здесь «американский». Эта кодировка проектировалась даже без учета европейских языков, не говоря уже об азиатских.

Но в ней были и огрехи. Набор символов ASCII состоит из 128 символов (он 7-разрядный). Но стандартный байт состоит из 8 битов, поэтому возникла естественная идея расширить набор ASCII, воспользовавшись кодами от 128 до 255 для других целей. Беда в том, что эта идея была реализована многократно и по-

разному компанией IBM и другими. Не было общепринятого соглашения о том, какому символу соответствует, например, код 221.

Недостатки такого подхода очевидны. Даже если отправитель и получатель договорятся об используемом наборе символов, все равно они не смогут общаться на нескольких языках, так для всех сразу не хватает символов. Если вы хотите писать по-немецки, но вставить в текст несколько цитат на греческом или иврите, то, скорее всего, ничего не получится. И эта схема не позволила даже приблизиться к решению проблем, связанных с азиатскими языками, например, китайскому, японскому и корейскому.

Было два основных способа решить эту задачу. Первый – использовать гораздо более обширный набор символов, например, представляя каждый символ 16 битами (так называемые *широкие символы*). Второй – обратиться к многобайтовым кодировкам переменной длины.

В случае многобайтовой схемы одни символы представляются единственным байтом, другие – двумя, а третьи – тремя или даже большим числом. Очевидно, при этом возникает множество вопросов. В частности, любая строка должна однозначно декодироваться. Первый байт многобайтового символа мог бы принадлежать специальному классу, так что мы сумели бы понять, что следует ожидать дополнительный байт, но как быть со вторым и последующими? Разрешено ли им перекрываться с набором однобайтовых символов? Могут ли определенные символы выступать в роли второго и третьего байта или это следует запретить? Сможем ли мы перейти в середину строки и при этом не запутаться? Сможем ли просматривать строку в обратном направлении? Для разных кодировок были приняты различные проектные решения.

В конечном итоге родилась идея кодировки Unicode. Считайте, что это «всемирный набор символов». Увы, на практике все не так просто.

Возможно, вы слышали, что Unicode был (или остается) ограничен 65536 символами (столько различных комбинаций можно представить 16 битами). Распространенное заблуждение. При проектировании Unicode такие ограничения не закладывались. С самого начала было ясно, что во многих случаях это будет многобайтовая схема. Количество представимых с помощью Unicode символов практически безгранично, и это хорошо, так как 65000 никогда не хватит для всех языков мира.

Говоря об интернационализации, нужно прежде всего понимать, что интерпретация строки не является внутренне присущей самой строке. Это заблуждение проистекает из уже неактуального представления, будто существует лишь один способ хранения строки.

Подчеркну, это исключительно важное положение. Внутренне строка – это просто последовательность байтов. Представьте себе, что в памяти машины хранится один байт в кодировке ASCII. Если это буква, которую мы называем «заглавная латинская А», то реально хранится число 65.

Почему мы считаем, что 65 – это А? Потому что так мы договорились использовать (интерпретировать) это значение. Если мы складываем его с другим чис-

лом, то оно используется (интерпретируется) как число. А если мы отправляем его на терминал по последовательной линии связи, значит, интерпретируем как ASCII-символ.

Если можно по-разному интерпретировать одиночный байт, то почему же нельзя так сделать для последовательности байтов? На самом деле, чтобы получилась осмысленная строка, предполагаемая схема интерпретации (или кодировка) должна быть известна заранее. Кодировка – это просто соответствие между двоичными числами и символами. И снова не все так просто.

Поскольку Ruby появился в Японии, он прекрасно справляется с двумя различными японскими кодировками (и ASCII). Не буду тратить время на рассказ о поддержке японского языка; если вы японец, то в вашем распоряжении сколько угодно книг по Ruby на этом языке. А для всех остальных наиболее распространенной кодировкой является Unicode. О ней мы и будем говорить в этой главе.

Но перед тем как перейти к деталям, познакомимся с некоторыми терминами. Называть вещи полезными именами – одна из основ мудрости.

- *Байт* – это просто восемь битов (хотя когда-то даже это было неверно). По традиции многие считают, что байт соответствует одному символу. Ясно, что в контексте I18N это не так.
- *Кодовая позиция* – это один элемент воображаемой таблицы, с помощью которой представляется набор символов. Хотя это и не совсем верно, можете считать, что кодовые позиции взаимно однозначно отображаются на символы. Точнее будет сказать, что иногда для уникального задания символа требуется несколько кодовых позиций.
- *Глиф* (типографический символ) – это визуальное представление кодовой позиции. Хотя интуитивно это и не совсем очевидно, символ и его визуальное представление – разные вещи. (Я могу открыть текстовый редактор и набрать заглавную А десятком разных шрифтов, но все это будет один и тот же символ А.)
- Понятие *графемы* близко к глифу, но о графемах мы говорим в контексте языка, а не программного обеспечения. Графема может быть комбинацией (простой или не очень) двух и более глифов. Так пользователь воспринимает символ в контексте своего родного языка. Разница настолько тонкая, что большинство программистов могут о ней никогда не задумываться.

Что же тогда такое *символ*? Даже в мире Unicode на этот счет нет четкого понимания, поскольку языки ведут себя по-разному, а программисты мыслят иначе, чем прочие люди. Будем говорить, что символ – это абстракция написания знака, который визуально может быть представлен одним или несколькими способами.

Перейдем к конкретике. Сначала я хочу познакомить вас с нотацией. Традиционно кодовые позиции Unicode записываются как `U+`, а затем четыре или более 16-ричных цифр в верхнем регистре. То, что мы называем латинской буквой "A", можно представить в виде `U+0041`.

Теперь возьмем букву «é» (строчная е с акутом). Ее можно представить в Unicode двумя способами. Во-первых, это одна кодовая позиция `U+00E9` (СТРОЧНАЯ

ЛАТИНСКАЯ Е С АКУТОМ). С другой стороны, это сочетание двух кодовых позиций: строчная e и диакритический знак акут – U+0065 и U+0301. Иными словами, СТРОЧНАЯ ЛАТИНСКАЯ Е, за которой следует АКУТ.

Обе формы одинаково правильны. Более короткая называется *монолитной* (precomposed) формой. Однако имейте в виду, что не для каждого языка имеются монолитные варианты, поэтому не всегда можно свести подобный символ к одной кодовой позиции.

Я назвал Unicode кодировкой, но это не вполне верно. Unicode отображает символы на кодовые позиции; существуют разные способы отобразить кодовые позиции на двоичное представление. По существу, Unicode – это семейство кодировок.

Возьмем, к примеру, строку "Matz". Она состоит их четырех кодовых позиций Unicode:

```
"Matz"    # U+004d U+0061 U+0074 U+007a
```

Естественнее всего сохранить их в виде простой последовательности байтов.

```
00 4d 00 61 00 74 00 7a
```

Такая кодировка называется UCS-2 (два байта) или UTF-16 (16 битов). Отметим, что эта кодировка имеет две разновидности: тупоконечную (big-endian) и остроконечную (little-endian) – в зависимости от того, старший или младший байт хранится первым. Заметим, однако, что каждый второй байт в этой последовательности нулевой. Это не просто совпадение, английский язык редко выходит за пределы кодовой позиции U+00FF. Так разбрасываться памятью расточительно.

И это наблюдение подводит нас к идее кодировки UTF-8. В ней «традиционные» символы представлены одним байтом, а остальные – несколькими. Вот как записывается та же строка в кодировке UTF-8:

```
4d 61 74 7a
```

Мы всего лишь избавились от нулей. Однако более важен тот факт, что мы получили обычную кодировку ASCII. Так и задумано: «простой ASCII» можно считать собственным подмножеством UTF-8. На самом деле, 8-битная кодировка ISO-8859-1 (которая сильно пересекается с ASCII) также является собственным подмножеством (не путайте кодировку ISO-8859-1, которую еще называют Latin-1, с кодировкой Windows-1252.) Короче говоря, это означает, что UTF-8 «обратно совместима» с кодировками ASCII и Latin-1.

Отсюда, в частности, следует, что при интерпретации текста в кодировке UTF-8 как ASCII-текста он выглядит «как обычно» (особенно если это преимущественно англоязычный текст). Иногда вы видите, что браузер или другое приложение отображает английский текст правильно, но местами появляются «крокозябры». Это, скорее всего, означает, что программа сделала неверные предположения об используемой кодировке.

Итак, можно сказать, что UTF-8 экономит память. Конечно, я снова становлюсь на англоцентрическую точку зрения (по крайней мере, ASCII-центрическую). Если текст в основном состоит из ASCII-символов, то да, память экономится, но для других языков, например греческого или русского, размер строк даже увеличится.

Еще одно очевидное достоинство UTF-8 – «обратная совместимость» с кодировкой ASCII, которая, по-видимому, все еще остается самой распространенной однобайтовой кодировкой в мире. Наконец, у UTF-8 есть некоторые особенности, делающие ее удобной для программистов.

Во-первых, байты, входящие в состав многобайтовых символов, тщательно подобраны. Нулевой байт (ASCII 0) никогда не встречается в качестве n-го байта в последовательности (где n > 1), то же самое справедливо для таких распространенных символов как косая черта (обычно используется для разделения компонентов пути к файлу). На самом деле, никакой байт из диапазона 0x00–0x7F не может быть частью никакого другого символа.

Второй байт многобайтового символа однозначно определяет, сколько байтов за ним следует. Этот второй байт всегда выбирается из диапазона от `0xC0` до `0xFD`, а следующие за ним – из диапазона от `0x80` до `0xBF`. Таким образом, схема кодирования свободна от состояния и позволяет восстанавливать пропущенные или искаженные байты.

UTF-8 – одна из самых распространенных и гибких кодировок в мире. Она применяется с начала 1990-х годов и является кодировкой по умолчанию для XML-документов. Если явно не оговорено противное, Ruby предполагает, что весь код, а также входной и выходной текст представлены в кодировке UTF-8. Поэтому в этой главе мы будем иметь дело главным образом с использованием этой кодировки в Ruby.

В части интернационализации есть две концепции, которые я считаю фундаментальными, можно даже сказать – почти аксиоматическими. Во-первых, у строки нет никакой внутренней интерпретации. Ее надлежит интерпретировать согласно тому или иному внешнему стандарту. Во-вторых, между байтами и символами нет взаимно однозначного соответствия – символ может быть представлен одним или несколькими байтами. Существуют и другие правила, но это самые важные.

4.2. Работа с кодировками

Даже если вы, будучи программистом, пользуетесь только английским языком, пользователи вашей программы вполне могут захотеть вводить текст на другом языке. Имена людей, названия мест и многие другие слова часто содержат символы, не встречающиеся в английском.

Поддержка таких языков может оказаться весьма непростой задачей. В разных языках приняты различные правила сортировки. Например, в словацком языке «ch» следует за «h», а в шведском «Å» идет после «Z». В русском языке окончания слов во множественном числе различаются в зависимости от количества: для 2, 3 и 4 предметов окончание одно, а для 5, 6, 7, 8, 9 и 0 – другое. Современные программы, особенно рассчитанные на веб, должны быть готовы принимать многоязычный текст, даже если они не были локализованы для других языков.

Полностью интернационализированные программы должны принимать входные данные и выводить результаты на разных языках и, в частности, правильно записывать слова во множественном числе. Они должны форматировать числа,

даты и денежные величины, как принято в текущем языке, а также правильно сортировать строки.

Давайте познакомимся с различными инструментами, которыми программист на Ruby может воспользоваться для выполнения интернационализации и локализации. Прежде всего, нужно понимать, как в Ruby хранятся строки символов.

Я уже говорил, что по умолчанию в Ruby применяется кодировка UTF-8. Если интерпретатору не сказано противное, он предполагает, что исходный код, а также входной и выходной текст представлены в этой кодировке.

Букве «е» без акута соответствует кодовая позиция U+00E9. В Ruby символы Unicode можно также записывать в виде u плюс шестнадцатеричный номер кодовой позиции. Иначе говоря, следующие строки равны:

```
"épée" == "\u00E9p\u00E9e"       # true
# Французское слово "épée" означает разновидность меча
```

В Ruby строки всегда хранятся в виде последовательности кодовых позиций, но некоторые кодовые позиции кодируются несколькими байтами. Интерпретатор дает нам доступ к этим цепочкам байтов. Рассмотрим несколько примеров:

```
sword = "épée"
sword.length         # 4
sword.bytes          # [195, 169, 112, 195, 169, 101]
sword.bytes.length   # 6
```

Эта строка содержит четыре символа, но для их кодирования необходимо 6 байтов. Символ é в UTF-8 представляется двумя байтами 195 и 169, именно в таком порядке.

```
cp = sword.codepoints     # [233, 112, 233, 101]
cp.map {|c| c.to_s(16) }  # ["e9", "70", "e9", "65"]
sword.scan(/./)           # ["é", "p", "é", "e"]
```

Функция String#unpack возвращает номера кодовых позиций Unicode. Преобразовав номер в шестнадцатеричную форму, мы увидим, что кодовая позиция символа «é» равна U+00E9.

Из предыдущего примера также видно, что регулярные выражения полностью поддерживают Unicode. При сопоставлении производится сравнение символов, а не байтов.

4.2.1. Нормализация

Концептуально многие символы составлены из двух других символов. Например, буква е и знак акута в сочетании дают букву é. Помимо символа СТРОЧНАЯ ЛАТИНСКАЯ БУКВА Е С АКУТОМ, в Unicode имеются отдельные кодовые позиции СТРОЧНАЯ ЛАТИНСКАЯ БУКВА Е (U+0065) и МОДИФИЦИРУЮЩИЙ АКУТ (U+0301)

Комбинирование кодовых позиций называется *композицией*, а их разделение – *декомпозицией*. Чтобы понять, представляют ли две последовательности кодовых позиций один и тот же символ, программа, поддерживающая Unicode, должна сначала выполнить композицию (или декомпозицию) кодовых позиций, применяя

один и тот же набор правил. Этот процесс называется *нормализацией*. Нормализованную последовательность кодовых позиций можно корректно сравнивать с другими последовательностями, нормализованными точно таким же способом.

До сих пор мы пользовались уже составленными – монолитными – символами, в которых базовый символ и диакритический знак объединены в одну кодовую позицию. Но, вообще говоря, Unicode поддерживает раздельную кодировку символов и сопровождающих их диакритических знаков.

Зачем это может понадобиться? Это дает дополнительную гибкость и позволяет применять диакритические знаки к любому символу, а не ограничивать себя комбинациями, которые предусмотрел проектировщик кодировки. На самом деле, в шрифты включены глифы для наиболее распространенных комбинаций символов и диакритических знаков, но отображение символа и его кодирование – вещи разные. Таблица 4.1 проясняет это различие.

Таблица 4.1. Составные и монолитные формы

Монолитная форма «é»				
Название символа	**Глиф**	**Кодовая позиция**	**Байты UTF-8**	**Примечания**
СТРОЧНАЯ ЛАТИНСКАЯ Е С АКУТОМ	é	U+00E9	0xC3 0xA9	Один символ, одна кодовая позиция, один байт
Составная форма «é»				
Название символа	**Глиф**	**Кодовая позиция**	**Байты UTF-8**	**Примечания**
СТРОЧНАЯ ЛАТИНСКАЯ Е	e	U+0065	0x65	Один символ, две кодовых позиции (два «программистских символа»), три байта UTF-8
МОДИФИЦИРУЮЩИЙ АКУТ	◌́	U+0301	0xCC 0x81	

При проектировании Unicode приходилось учитывать такие вещи, как эффективность и совместимость с существующими национальными кодировками. Иногда это приводит к избыточности; например, в Unicode имеются как кодовые позиции для составных форм, так и для многих уже применяющихся монолитных форм. Иначе говоря, существует кодовая позиция СТРОЧНАЯ ЛАТИНСКАЯ БУКВА Е С АКУТОМ, а также позиции, соответствующие, например, лигатуре двойная-f.

Рассмотрим, к примеру, немецкое слово «öffnen» (открывать). Даже если забыть о регистре, его можно закодировать четырьмя способами:

1) o + МОДИФИЦИРУЮЩАЯ ТРЕМА (U+0308) + f + f + n + e + n;
2) СТРОЧНАЯ ЛАТИНСКАЯ БУКВА O С ТРЕМОЙ (U+00F6) + f + f + n + e + n;
3) o + МОДИФИЦИРУЮЩАЯ ТРЕМА + ЛИГАТУРА ДВОЙНОЕ-F (U+FB00) + n + e + n;
4) СТРОЧНАЯ ЛАТИНСКАЯ БУКВА O С ТРЕМОЙ + ЛИГАТУРА ДВОЙНОЕ-F + n + e + n.

Трема – это двоеточие над буквой. В немецком языке она называется *умляут*.

Нормализацией называется процедура приведения разных представлений символа к стандартной форме. Можно быть уверенным, что после нормализации данный символ закодирован вполне определенным образом. Каким именно, зависит

от того, чего мы хотим достичь. В приложении 15 к стандарту Unicode перечислены четыре формы нормализации:

1) D (каноническая декомпозиция);

2) C (каноническая декомпозиция с последующей канонической композицией);

3) KD (совместимая декомпозиция);

4) KC (совместимая декомпозиция с последующей канонической композицией).

Иногда можно встретить аббревиатуры NKFC (Normalization Form KC) и т.д.

Точные правила, сформулированные в стандарте, довольно сложны, в них проведено различие между «канонической эквивалентностью» и «совместимой эквивалентностью». (Корейский и японский языки требуют особого рассмотрения, но мы не станем тратить на это время.) В таблице 4.2 показано, как форма нормализации влияет на приведенные выше строки.

Таблица 4.2. Нормализованные формы в Unicode

Исходная	NFD	NFC	NFKD	NFKC
o+"+f+f+n+e+n	o+"+f+f+n+e+n	"+f+f+n+e+n	o+"+f+f+n+e+n	"+f+f+n+e+n
"+f+f+n+e+n	o+"+f+f+n+e+n	"+f+f+n+e+n	o+"+f+f+n+e+n	"+f+f+n+e+n
o+"+ff+n+e+n	o+"+ff+n+e+n	"+ff+n+e+n	o+"+f+f+n+e+n	"+f+f+n+e+n
"+ff+n+e+n	o+"+ff+n+e+n	"+ff+n+e+n	o+"+f+f+n+e+n	"+f+f+n+e+n

Какую форму лучше выбрать, зависит от задачи. Форма C «полностью монолитная», форма D – «полностью составная», обе они обратимы. Формы KC и KD предназначены для обеспечения «совместимости», для них требуется преобразование, например, разложение лигатуры «ffi» на три составляющих ее буквы (поэтому эти форму необратимы). Таким образом, формы совместимости лучше подходят для хранения, особенно если текст предполагается сравнивать с другими текстами, как в случае поиска и контроля правильности.

В gem-пакете activesupport имеется класс ActiveSupport::Multibyte::Chars, оберты вающий класс String. Он дает возможность манипулировать кодовыми позициями **Unicode** такими способами, которых класс String не допускает. И, в частности, изменять регистр – как ни удивительно, эта возможность в классе String отсутствует.

```
require 'active_support'

chars = ActiveSupport::Multibyte::Chars.new("épée")
"épée".upcase                # "éPéE"
chars.upcase.to_s            # "ÉPÉE"
"épée".capitalize            # "épée"
chars.capitalize.to_s        # "Épée"
```

Эта возможность, предоставляемая activesupport, «позабыта» в Ruby не случайно. Дело в том, что во многих языках понятие регистра вообще отсутствует. В та-

ких языках операция изменения регистра лишена смысла, а, значит, она бессмысленна и в общем случае. Выполнять такого рода преобразования следует лишь тогда, когда вы уверены, что поведение разумно для всех языков, которые планируется поддерживать.

В том же gem-пакете имеются полезные методы для отсечения, композиции и декомпозиции кодовых позиций и нормализации в любую из четырех форм.

```
sword_kc = chars.normalize(:kc)
sword_kd = chars.normalize(:kd)
sword_kc.bytes      # [195, 169, 112, 195, 169, 101]
sword_kd.bytes      # [101, 204, 129, 112, 101, 204, 129, 101]
sword_kc.scan(/./)  # ["é", "p", "é", "e"]
sword_kd.scan(/./)  # ["e", "´", "p", "e", "´", "e"]
```

4.2.2. Преобразование из одной кодировки в другую

У любого объекта String в Ruby имеется кодировка, которая определяет, как интерпретировать байты строки. Это позволяет вводить и использовать в одной программе данные в разных кодировках, но одновременно может приводить к неожиданным проблемам. Попытка объединить строки с несовместимыми кодировками заканчивается исключением Encoding::CompatibilityError. По счастью, существует метод encode, который преобразует строки из одной кодировки в другую. С его помощью мы можем привести строки к единой кодировке, перед тем как объединять их:

```
sword = "épée"
sword_mac = sword.encode("macRoman")
sword.bytes           # [195, 169, 112, 195, 169, 101]
sword_mac.bytes       # [142, 112, 142, 101]

str = sword + sword_mac
# несовместимые кодировки UTF-8 и macRoman

str = sword + sword_mac.encode("UTF-8")
# "épée"
```

Аналогично получение входных данных в неожиданной кодировке может впоследствии, при обработке данных, стать причиной исключения:

```
File.write("sword.txt", sword.encode("macRoman"))
invalid_sword = File.read("sword.txt")   # "\x8Ep\x8Ee"
invalid_sword.encoding                   # #<Encoding:UTF-8>
invalid_sword.valid_encoding?            # false

strings = invalid_sword.split("p")
# ошибка из-за неверной последовательности байтов в UTF-8
```

Исправить это можно, например, вызвав метод force_encoding, который корректирует кодировку уже прочитанных байтов:

```
forced_sword = invalid_sword.force_encoding("macRoman")
```

```
forced_sword.encoding           # #<Encoding:macRoman>
forced_sword.valid_encoding?    # true
forced_sword.split("p")         # ["\x8E", "\x8Ee"]
```

Другой способ обработки недопустимой последовательности байтов – сообщить Ruby, в какой кодировке представлен файл, чтобы он правильно читался с самого начала:

```
read_sword = File.read("sword.txt", :encoding => "macRoman")
read_sword.encoding       # #<Encoding:macRoman>
read_sword.split("p")     # ["\x8E", "\x8Ee"]

open_sword = File.open("sword.txt", "r:macRoman:UTF-8") do |f|
  f.read
end
open_sword.encoding       # #<Encoding:UTF-8>
open_sword.split("p")     # ["é", "ée"]
```

Во втором примере мы воспользовались методом IO.open, чтобы Ruby читал файл в одной кодировке, но перед возвратом преобразовывал данные в другую.

Однако иногда кодировка текста попросту неизвестна или данные настолько испорчены, что определить правильную кодировку невозможно. Но и в таком случае можно избежать исключения кодировки, нужно лишь заменять недопустимые байты допустимыми:

```
bad_sword = "\x8Ep\x8Ee"
bad_sword.encode!("UTF-8", :invalid => :replace,
                           :undef => :replace)
bad_sword                 # "p"
bad_sword.valid_encoding? # true
```

Параметр :replace означает, что байты, которые не удается декодировать, следует заменять – принятым в Unicode символом замены. При желании нечитаемые байты можно заменить любой другой строкой, в том числе пустой.

4.2.3. Транслитерация

Совершенно другой способ преобразования из одной кодировки в другую называется *транслитерацией*. Смысл ее в том, чтобы символы из одного алфавита заменить символами из другого алфавита. В контексте интернационализации транслитерация почти всегда сводится к замене ближайшим эквивалентом из набора символов ASCII.

Например, «épée» можно заменить на «epee», «dziękuję» – на «dziekuje», а «εὐδαιμονία» – на «eudaimonia». При использовании различных систем письма возможно даже несколько способов транслитерации одной и той же строки. Хотя транслитерация бывает полезна для оформления вывесок, указателей или записи имен на чужом алфавите, зачастую она приводит к потере информации. Поэтому транслитерация не обеспечивает истинной перекодировки и по возможности ее следует избегать.

4.2.4. Упорядочение строк

В информатике под *упорядочением* (collation) понимается организация текста в определенном порядке. Обычно, но не всегда, строки упорядочиваются по алфавиту или сходным образом. Упорядочение также связано с нормализацией, потому что в большинстве языков буквы с акутами и без них при сортировке объединяются в группы.

Предположим, например, что мы хотим отсортировать следующий массив строк. Обратите внимание на присутствие как составных, так и монолитных символов. Что произойдет при использовании метода `Array#sort`?

```
eacute = [0x00E9].pack('U')
acute  = [0x0301].pack('U')
array  = ["epicurian", "#{eacute}p#{eacute}e", "e#{acute}lan"]
array.sort # ["epicurean", "élan", "épée"]
```

Это не то, что мы хотели. Но попытаемся понять, почему так получилось. Рассмотрим первые два символа каждой строки и байты, которыми они кодируются:

```
array.map {|word| {word[0,2] => word[0,2].bytes} }
# [{"ep"=>[101, 112]},
#  {"é"=>[101, 204, 129]},
#  {"ép"=>[195, 169, 112]}]
```

В первом слове два символа кодируются двумя байтами. Во втором слове две кодовые позиции, составляющие один символ, кодируются тремя байтами. В третьем слове есть один двухбайтовый и один однобайтовый символ. Взглянув на значения байтов, нетрудно понять, что слова отсортированы по составляющим их байтам.

У буквы «e» значение меньше, чем у первого байта буквы «é», поэтому в результате сортировки она оказывается раньше. В кодировке UTF-8 наименьшие значения у символов ASCII, поэтому любой не-ASCII символ окажется позже них. Средняя буква «é» всегда оказывается после «e», но перед «f» из-за большого значения первого байта акута.

Имейте в виду, что подобная проблема возникает даже в случае кодировки ASCII. Например, коды букв верхнего регистра меньше кодов соответственных букв нижнего регистра, поэтому после сортировки они оказываются в разных местах. Казалось бы, слова «pyre» и «PyrE» должна после сортировки оказаться рядом, но на самом деле слово «Pyramus» встает между ними. По существу, это означает, что лексикографическая сортировка *не является* сортировкой по алфавиту, она не учитывает многие сложные правила, принятые при составлении словарей и телефонных книг.

В каждом языке действует свой набор правил, поэтому начнем с простого. Отсортируем в порядке английского алфавита, игнорируя акуты. Для этого можно просто подвергнуть все строки декомпозиции, а затем удалить диакритические знаки. В Unicode модифицирующим диакритическим знакам отведен диапазон от `U+0300` до `U+036F`.

Предположим, что наш список обрабатывается согласно правилам английского языка, причем диакритические знаки игнорируются. Первым делом нужно

определить методику трансформации. Мы преобразуем все символы в составную форму, затем исключим диакритические знаки, оставив только базовые символы, и отсортируем то, что получится:

```
Chars = ActiveSupport::Multibyte::Chars # for convenience

def english_sort(str)
      kd = Chars.new(str).downcase.normalize(:kd)
      cp = kd.codepoints.select {|c| c < 0x0300 || c > 0x036F }
      cp.pack("U*")
end
array.map{|s| english_sort(s) }
= ["epicurean", "epee", "elan"]

array.sort_by{|s| english_sort(s) }
= ["élan", "épée", "epicurean"]
```

Уже лучше. Мы учли наличие заглавных букв (воспользовавшись методом downcase, который поддерживает Unicode), но не учли эквивалентность символов. Возьмем для примера немецкий язык.

На самом деле в немецком языке есть несколько способов упорядочения, мы остановимся на стандарте DIN-2 (как в телефонном справочнике). Согласно этому стандарту, символ "ß" эквивалентен "ss", а умляут эквивалентен букве "e", то есть "ö" — то же самое, что "oe" и т.д.

Наш метод трансформации должен учитывать эти детали. Снова начнем с приведения к составной форме. Например, модифицирующая трема (умляут) представляется кодовой позицией U+0308. За основу мы возьмем метод преобразования регистра, имеющийся в Ruby, но несколько дополним его. Вот как выглядит теперь код трансформации:

```
def sort_german(str)
      mb = ActiveSupport::Multibyte::Chars.new(str)
      kd = mb.downcase.normalize(:kd)
      kd.gsub('ß', 'ss').gsub("\u0308", 'e').to_s
end

["Straße", "öffnen"].map {|x| sort_german(x) }
= ["strasse", "oeffnen"]
```

В реальности алгоритмы упорядочения сложнее, чем показано в примерах выше, в них имеется несколько уровней обработки. Обычно на первом уровне сравниваются только базовые символы (без учета диакритических знаков и регистра), на втором учитываются диакритические знаки, а на третьем – регистр и так далее. Каждый следующий уровень необходим лишь в случае, когда на предыдущем уровне сортировка не была доведена до конца.

Но даже при использовании нескольких уровней сортировка все равно остается чрезвычайно сложным делом, зачастую требуется знание особенностей языка. В некоторых языках последовательности, состоящие из нескольких символов, со-

ртируются как единая семантическая единица (например, в хорватском "lj" расположено между "l" и "m").

В датском языке «aa» (или «å») следует за «z», но только для незаимствованных слов. А для иностранных слов «aa» сортируется, как обычные «a». Это означает, что город Aachen предшествует Atlanta, но Aalborg располагается после Zurich.

В общем, невозможно придумать по-настоящему универсальный алгоритм сортировки, который давал бы правильные результаты для всех языков, поскольку во многих языках требования к сортировке прямо противоположны. Помните об этом, когда будете сортировать списки строк на других языках.

4.3. Перевод

Ложбан не зависит от национальных особенностей.
Его словарь был создан алгоритмически
на основе шести наиболее распространенных в мире
разговорных языков: китайского, хинди,
английского, русского, испанского и арабского.

— Nick Nicholas, John Cowan «What is Lojban?»

Рассмотрев, как программа обрабатывает ввод и вывод символов любого алфавита, мы теперь обратимся к обработке входных и выходных данных на любом *языке*. Перевод программы на другие языки — это большая работа, зато аудитория программы существенно расширяется.

Для полной локализации программы необходимо перевести весь текст, в том числе инструкции, сообщения об ошибках и выходные данные. Кроме того, нужно правильно отформатировать все числовые данные, включая денежные суммы, числа, даты и время. Мы начнем с простых аспектов перевода, а затем перейдем к более сложным примерам, в частности, формам множественного числа и форматированию числовых данных.

Gem-пакет `i18n` позволяет брать переводы из внешних файлов, составленных и отредактированных переводчиками. Файлы представлены в формате YAML (Yet Another Markup Language), ориентированном на чтение человеком. Каждое сообщение в файле идентифицируется ключом. Метод `I18n.translate` (у него имеется также краткий синоним `I18n.t`) принимает ключ и возвращает соответствующее этому ключу сообщение.

Рассмотрим пример. Первым делом мы затребуем gem-пакет `i18n` (при необходимости его можно установить командой `gem install i18n`), а затем сохраняем несколько фраз, нуждающихся в переводе:

```
require 'i18n'
I18n.backend.store_translations(:en,
      greeting: { hello: "Hello there!" })
```

```
I18n.backend.store_translations(:ja,
      greeting: { hello: " " })
```

Двухбуквенные сокращенные названия языков определены в международном стандарте ISO 639. Точно такие коды используются во многих языках программирования, в том числе C, Java, PHP, Perl и Python.

Второй аргумент метода содержит *ключ*, по которому можно будет найти *сообщение*. Для каждого кода языка сообщения могут быть организованы в виде произвольной иерархии ключей.

Метод `translate` (или для краткости `t`) принимает ключ и ищет сообщение с таким ключом в разделе файла, соответствующем текущей локали. По умолчанию подразумевается локаль `:en`, но ее в любой момент можно изменить.

```
I18n.locale                 # :en
I18n.t "greeting.hello"     # "Hello there!"

I18n.locale = :ja           # :ja
I18n.t "greeting.hello"     # " "
```

Каждая точка в ключе, переданном методу `I18n.t`, означает один уровень иерархии. Каждый сегмент ключа после точки соответствует ключу в хэше переводов.

В больших программах переводы обычно хранятся в специальных файлах, написанных на Ruby или YAML и находящихся в каталоге `locale/`. Ключи и сообщения из несколько таких файлов объединяются, а файлы обычно организуются по принципу одного языка или общего типа сообщений.

Напишем небольшую программу, которая задает вопросы пользователю и печатает результаты анкетирования. Нас будет интересовать имя, город проживания и количество детей, а для перевода выходных сообщений программы мы воспользуется gem-пакетом `i18n`.

Сначала затребуем пакет `i18n`, скажем ему, где искать файлы переводов, и установим локаль. В UNIX-системах локаль традиционно задается с помощью переменной окружения `LANG` в формате `en_US.UTF-8`, на это и будем ориентироваться.

```
# survey.rb
require 'i18n'
I18n.load_path = Dir["locale/*"]
I18n.enforce_available_locales = true
I18n.locale = ENV["LANG"].split("_").first || :en

puts I18n.t("ask.name")
name = gets.chomp
puts I18n.t("ask.location")
place = gets.chomp
puts I18n.t("ask.children")
childnum = gets.chomp.to_i
puts I18n.t("ask.thanks")

puts name, place, childnum
```

Если сейчас выполнить эту программу, то дело закончится исключением I18n::InvalidLocale. Объясняется это тем, что мы не предоставили переводы на английский язык. Создадим файлы, содержащие переводы на английский и русский.

```
# locale/en.yml
en:
   ask:
      name: "What is your name?"
      location: "Where do you live?"
      children: "How many children do you have?"
      thanks: "Thank you!"

# locale/ru.yml
ru:
   ask:
      name: "Как вас зовут?"
      location: "Где вы живете?"
      children: "Сколько у вас детей?"
      thanks: "Спасибо"
```

Теперь программа может задавать вопросы на английском или русском в зависимости от значения переменной окружения LANG:

```
$ ruby survey.rb
What is your name?
[…]
$ LANG=ru ruby survey.rb
Как вас зовут?
[?]
```

4.3.1. Значения по умолчанию

Перевод программы на несколько языков может значительно расширить круг ее пользователей, но на этом пути нас подстерегают и некоторые опасности. Метод translate ожидает, что перевод существует для каждого ключа. Отсутствие перевода для какого-то ключа не приводит к исключению, но делает программу бесполезной. Если файл для перевода на японский окажется пустым, то при анкетировании на японском языке мы увидим сообщение об отсутствующем переводе:

```
$ echo "ja:\n ask:" > locale/ja.yml
$ LANG=ja ruby survey.rb
translation missing: ja.ask.name
```

Естественно, такое поведение не дает нормально работать с программой. Бо-

Но в режиме эксплуатации возбуждать исключение в случае отсутствия перевода вряд ли разумно. В таком случае лучше перейти на запасной вариант и предложить существующий перевод на какой-то другой язык. Это проще всего сделать, включив расширение Fallback и установив локаль, подразумеваемую по умолчанию:

```
require "i18n/backend/fallback"
I18n::Backend::Simple.send(:include, I18n::Backend::Fallback)
I18n.default_locale = :en
```

После установки локали по умолчанию :en вместо отсутствующих переводов будет подставляться перевод на английский:

```
$ LANG=ja ruby survey.rb
What is your name?
[…]
```

Отметим, что если одновременно установить локаль по умолчанию и обработчик исключения, то программа будет искать ключ сначала в текущей локали, затем в локали по умолчанию и, только если он отсутствует в обоих местах, возбудит исключение.

4.3.2. Пространства имен

Вы, наверное, обратили внимание, что все ключи в анкете начинаются строкой ask. Аргумент namespace метода translate позволяет создать вспомогательный метод, который требует только название вопроса.

```
def ask(key)
      I18n.translate(key, namespace: "ask")
end

puts ask("name")
name = gets.chomp
```

Рекомендуется использовать именованные вспомогательные методы или определить общий метод перевода, который подставляет правильное пространство имен при работе в разных контекстах. В этом случае будет проще управлять сложными наборами переводов.

4.3.3. Интерполяция

В отсутствие перевода вставить переменную в строку очень просто. Чтобы сформировать приветствия, имея переменную name, нужно просто написать "Hello, #{name}!".

Когда строки переведены, так поступить не получится. В зависимости от языка переменная может находиться до, после или даже между другими словами. Решение о том, в каком месте производить интерполяцию, должен принимать переводчик, а не разработчик.

Для решения этой проблемы в строках перевода применяется специальный формат интерполяции, включающий знак процента и фигурные скобки. Разра-

ботчик должен просто указать в сообщении имя переменной, а переводчик вставит эту переменную в нужное место.

Добавим в наши переводы сообщение с интерполяцией, а в программу строку, которая печатает результаты анкетирования в более наглядном виде:

```
# survey.rb
puts I18n.t("result.lives_in", name: name,
      place: place, childnum: childnum)

# locale/en.yml
en:
   result:
      lives_in: >
        %{name} lives in %{place},
        and has %{childnum} child(ren).

# locale/ru.yml
ru:
   result:
      lives_in: >
        %{name} проживает в  %{place}
        и имеет %{childnum} детей.
```

Теперь результаты печатаются на английском и русском в виде более-менее связного предложения.

```
John Smith lives in San Francisco, and has 4 child(ren).
John Smith проживает в San Francisco и имеет 4 детей
```

Но в одном месте наш «улучшенный» вывод выглядит плохо. Нам приходится писать «child(ren)», потому что может быть как один ребенок, так и несколько, в том числе нуль. В русском дело обстоит еще хуже: фраза «имеет 1 детей» составлена грамматически неверно.

4.3.4. Формы множественного числа

Употребление числительных в разных языках сложнее, чем просто перевод. Перевод одной и той же фразы может выглядеть совершенно по-разному в зависимости от того, сколько имеется объектов.

В английском языке с формами множественного числа дело обстоит относительно хорошо. У имени существительного есть форма единственного и множественного числа (например, «child» и «children»), которые иногда совпадают (как в словах «sheep» и «deer»). В японском все еще проще — форма имени существительного вообще не зависит от количества.

В других языках ситуация может быть гораздо сложнее. В русском есть три формы: единственное число, множественное число для случаев, когда количество оканчивается на 2, 3 или 4, множественное число для остальных случаев. Русский в этом отношении не одинок: в польском и других языках принцип изменения по числу похожий.

Чтобы можно было учесть формы множественного числа в переводе, необходимо включать в перевод отдельные ключи для каждой формы. В английском это означает «zero», «one» и «other». Вот как выглядит файл для английской локали с учетом форм множественного числа:

```
# locale/en.yml
en:
  result:
    lives_in: "%{name} lives in %{place}, and has "
    children:
      zero: "no children."
      one: "a single child."
      other: "%{count} children."
```

При использовании таких переводов нужно вызвать метод translate, передав ему параметр count. Тогда будет автоматически подставлена правильная форма множественного числа:

```
# survey.rb
puts I18n.t("result.lives_in ", name: name, place: place) +
     " " + I18n.t("result.children", count: childnum)
```

Чтобы поддержать формы множественного числа в японском языке, мы должны затребовать пакет Pluralization и воспользоваться им:

```
# survey.rb
require "i18n/backend/pluralization"
I18n::Backend::Simple.send(:include,
     I18n::Backend::Pluralization)
```

Затем мы добавляем написанное на Ruby правило в новый файл локали, содержащий правила образования множественного числа:

```
# locale/plurals.rb
{ ja: { 18n: { plural: {
  keys: [:other],
  rule: -> (n) { n.zero? ? :zero : :other }
}}}}
```

После этого можно добавить ключи только для «zero» и «other», потому что в японском один и несколько объектов грамматически неразличимы:

```
# locale/ja.yml
ja:
  result:
    lives_in: "%{name}は%{place}に住んでいて、そして"
    children:
      zero: "子供がいません。"
      other: "%{count}人子供がいます。"
```

Правила образования множественного числа для других языков, более или менее сложные, реализуются аналогично. Вместо того чтобы снова и снова открывать такие правила для каждого языка по отдельности, разработчики из разных стран предприняли совместные усилия для создания весьма полных списков правил образования множественного числа и других правил форматирования.

Консорциум Unicode публикует большой набор правил в стандартном репозитории локальных данных (Common Locale Data Repository – CLDR) по адресу cldr.unicode.org. Компания Twitter выпустила версию этих данных, доступную непосредственно из Ruby, в виде gem-пакета `twitter_cldr`.

4.4. Локализованное форматирование

При обсуждении форм множественного числа мы уже отмечали, что существуют и другие правила форматирования, помимо изменения существительных во множественном числе. В каждом языке имеются свои стандарты представления даты, времени, чисел и денежных величин.

Пакет `i18n` сам по себе включает поддержку форматирования даты и времени, но не предлагает никаких переводов. Для форматирования даты, времени, чисел и денежных величин в текущей локали проще всего взять gem-пакет `twitter_cldr`. Мы будем использовать его в примерах ниже, но имейте в виду, что форматы CLDR в нем жестко зашиты, изменить их путем редактирования какого-нибудь `yml`-файла невозможно.

4.4.1. Дата и время

В стандарте CLDR определены четыре формата даты и времени для любого языка: полный, длинный, средний и короткий. Для некоторых языков существуют и другие форматы, поэтому справляйтесь с документацией по соответствующему gem-пакету. Представление даты и времени в нестандартном формате (с помощью метода `strftime`) рассматривается в разделе 7.21 «Форматирование и печать временных значений».

Перед тем как воспользоваться любым из методов `localize`, необходимо установить gem-пакет `twitter_cldr` командой `install twitter_cldr`, а затем затребовать его в Ruby-скрипте:

```
require 'twitter_cldr'
```

Для совместного форматирования даты и времени в составе одной локализованной строки служит метод `DateTime#localize`:

```
date = DateTime.parse("2014-12-15")
date.localize(:en).to_short_s     # "12/15/14, 12:00 AM"
date.localize(:fr).to_short_s     # "15/12/2014 00:00"
```

Для форматирования одного лишь времени имеется аналогичный метод `Time#localize`:

```
time = Time.parse("9:00 PM GMT")
time.localize(:en).to_short_s     # "9:00 PM"
time.localize(:fr).to_short_s     # "21:00"
```

Печать дат выделяется из общего правила – локализованный объект `DateTime` необходимо сначала преобразовать в дату, а только потом форматировать. Метода `Date#localize` не существует.

```
date = DateTime.parse("2014-12-15")
date.localize(:en).to_date.to_short_s   # "12/15/14"
date.localize(:fr).to_date.to_short_s   # "15/12/2014"
```

Нетрудно догадаться, что существуют также методы to_medium_s, to_long_s и to_full_s для вывода строк, в которых названия месяцев и дней недели написаны словами:

```
date.localize(:en).to_medium_s
# "Dec 15, 2014, 12:00:00 AM"

date.localize(:en).to_long_s
# "December 15, 2014 'at' 12:00:00 AM UTC"

date.localize(:en).to_full_s
# "Monday, December 15, 2014 'at' 12:00:00 AM UTC +00:00"
```

4.4.2. Числа

Форматирование чисел столь же просто – любое число наделяется методом localize:

```
num = 1_337
num.localize(:en).to_s    # "1,337"
num.localize(:fr).to_s    # "1 337"
```

Для форматирования десятичных чисел нужно всего лишь локализовать тип Float или вызвать метод to_decimal для локализованного числа:

```
1337.00.localize(:en).to_s(precision: 2)         # "1,337.00"
num.localize(:fr).to_decimal.to_s(precision: 2)  # "1 337,00"
```

Наконец, локализованные числа можно форматировать в виде процентных величин с помощью метода to_percent, который принимает также параметр precision:

```
num.localize(:en).to_percent.to_s                # "1,337%"
num.localize(:fr).to_percent.to_s(precision: 2)  # "1 337,00 %"
```

4.4.3. Денежные величины

По умолчанию денежные величины форматируются, как в США, но это легко изменить, задав трехбуквенный код валюты, например:

```
num.localize(:en).to_currency.to_s
# "$1,337.00"

num.localize(:fr).to_currency.to_s(currency:"EUR")
# "1 337,00 €"
```

4.5. Заключение

В этой главе мы рассмотрели проблемы, с которыми сталкиваются программисты в процессе интернационализации и локализации приложений. То и другое многие

пользователи и разработчики считают чрезвычайно важным – если все сделано правильно, то аудитория приложения значительно расширяется.

Говоря об интернационализации, мы рассмотрели кодировки символов и узнали, каким образом стандарт Unicode обеспечивает возможность закодировать практически любой из существующих символов. Затем мы видели, как локализовать приложение, в полной мере поддержав перевод на другие языки, включая и формы множественного числа. Наконец, мы показали, как в соответствии с выбранной локалью форматируются числа, дата, время и денежные величины.

Мы видели также, как gem-пакет `i18n` вкупе с другими пакетами используется для перевода строк, образования формы множественного числа и правильного форматирования чисел.

Теперь мы немного отдохнем от строк и форматирования. В следующей главе мы поговорим о представлении в Ruby чисел и выполнении вычислений.

Глава 5. Численные методы

Дважды [члены Парламента] задавали мне вопрос:
«А скажите, мистер Бэббидж, если вы заложите в эту машину
неверные числа, то получите правильный результат?»
Не могу даже представить себе, насколько извращенно
должен мыслить человек, задающий такие вопросы.

-- Чарльз Бэббидж

Числа -- самый первичный тип данных, естественный для любого компьютера. Придется сильно постараться, чтобы найти такую область знания, в которой нет места числам. Будь вы бухгалтером или конструктором воздухоплавательных аппаратов, без чисел вам не обойтись. В этой главе мы обсудим различные способы обработки, преобразования и анализа числовых данных.

Как и всякий современный язык, Ruby прекрасно умеет работать с любыми числами - как целыми, так и с плавающей точкой. В нем есть полный набор ожидаемых математических операторов и функций, а равно и кое-какие приятные сюрпризы: классы `Bignum`, `BigDecimal` и `Rational`.

Помимо средств для манипуляции числами, имеющихся в системной и стандартной библиотеках, рассмотрены также более специфические темы (тригонометрия, математический анализ и статистика). Примеры приведены не только для сведения, но и как образцы кода на языке Ruby, иллюстрирующие принципы, которые изложены в других частях книги.

5.1. Представление чисел в языке Ruby

Если вы знакомы с любым другим языком программирования, то представление чисел в Ruby не вызовет у вас никакого удивления. Объект класса `Fixnum` может представлять число со знаком или без знака:

```
# число без знака (положительное)
# то же, что и выше
# отрицательное число
```

Если число длинное, то между любыми цифрами можно вставлять знак подчеркивания. Это сделано исключительно для удобства, на значении константы никак не сказывается. Обычно подчерки вставляются в те же места, где бухгалтеры вставляют пробелы.

```
1048576     # число в обычной записи
1_048_576   # то же самое значение
```

Целые числа можно представлять и в других системах счисления (по основанию 2, 8 и 16). Для этого в начале ставятся префиксы 0b, 0 и 0x соответственно.

```
0b10010110      # двоичное
0b1211          # ошибка!
01234           # восьмеричное (основание 8)
01823           # ошибка!
0xdeadbeef      # шестнадцатеричное (основание 16)
0xDEADBEEF      # то же самое
0xdeadpork      # ошибка!
```

В числах с плавающей точкой десятичная точка должна присутствовать, а показатель степени, возможно со знаком, необязателен:

```
3.14            # число пи с двумя знаками
-0.628          # -2*pi, поделенное на 10, с тремя знаками
6.02e23         # число Авогадро
6.626068e-34    # постоянная Планка
```

В классе Float есть константы, определяющие минимальные и максимальные значения чисел с плавающей точкой. Они машинно-зависимы. Вот некоторые наиболее важные:

```
Float::MIN      # 2.2250738585072e-308 (на конкретной машине)
Float::MAX      # 1.79769313486232e+308
Float::EPSILON  # 2.22044604925031e-16
```

5.2. Основные операции над числами

Обычные операции сложения, вычитания, умножения и деления в Ruby, как и во всех распространенных языках программирования, обозначаются операторами +, -, * и /. Операторы в большинстве своем реализованы в виде методов (и потому могут быть переопределены).

Возведение в степень обозначается оператором **, как в языках BASIC и FORTRAN. Эта операция подчиняется обычным математическим правилам.

```
a = 64**2       # 4096
b = 64**0.5     # 8.0
c = 64**0       # 1
d = 64**-1      # 0.015625
```

При делении одного целого числа на другое дробная часть результата отбрасывается. Это не ошибка, так и задумано. Если вы хотите получить результат с плавающей точкой, позаботьтесь о том, чтобы хотя бы один из операндов был числом с плавающей точкой.

```
3 / 3           # 3
5 / 3           # 1
3 / 4           # 0
3.0 / 4         # 0.75
```

```
3 / 4.0        # 0.75
3.0 / 4.0      # 0.75
```

Если вы работаете с переменными и сомневаетесь относительно их типа, воспользуйтесь приведением типа к Float или методом to_f:

```
z = x.to_f / y
z = Float(x) / y
```

См. также раздел 5.17 «Поразрядные операции над числами».

5.3. Округление чисел с плавающей точкой

Кирк: Какие, вы говорите,
у нас шансы выбраться отсюда?
Спок: Трудно сказать точно, капитан.
Приблизительно 7824.7 к одному.

— Стар Трек, «Миссия милосердия»

Метод round округляет число с плавающей точкой до целого:

```
pi = 3.14159
new_pi = pi.round   # 3
temp = -47.6
temp2 = temp.round  # -48
```

Иногда бывает нужно округлить не до целого, а до заданного числа знаков после запятой. В таком случае можно воспользоваться функциями sprintf (которая умеет округлять) и eval:

```
pi = 3.1415926535
pi6 = eval(sprintf("%8.6f",pi))  # 3.141593
pi5 = eval(sprintf("%8.5f",pi))  # 3.14159
pi4 = eval(sprintf("%8.4f",pi))  # 3.1416
```

Это не слишком красиво. Поэтому инкапсулируем оба вызова функций в метод, который добавим в класс Float:

```
class Float

  def roundf(places)
    temp = self.to_s.length
    sprintf("%#{temp}.#{places}f",self).to_f
  end

end
```

Иногда требуется округлять до целого по-другому. Традиционное округление n+0.5 с избытком со временем приводит к небольшим ошибкам; ведь n+0.5 все-таки

ближе к n+1, чем к n. Есть другое соглашение: округлять до ближайшего четного числа, если дробная часть равна 0.5. Для реализации такого правила можно было бы расширить класс Float, добавив в него метод round_down:

```
class Float

  def round_down
    whole = self.floor
    fraction = self - whole
    if fraction == 0.5
      if (whole % 2) == 0
        whole
      else
        whole+1
      end
    else
      self.round
    end
  end

end

a = (33.4).round_down  # 33
b = (33.5).round_down  # 34
c = (33.6).round_down  # 34
d = (34.4).round_down  # 34
e = (34.5).round_down  # 34
f = (34.6).round_down  # 35
```

Видно, что round_down отличается от round только в том случае, когда дробная часть в точности равна 0.5. Отметим, кстати, что число 0.5 можно точно представить в двоичном виде. Не так очевидно, что этот метод правильно работает и для отрицательных чисел (попробуйте). Отметим еще, что скобки в данном случае необязательны и включены только для удобства восприятия.

Попутно заметим, что добавление метода в системный класс – вещь, к которой следует относиться вдумчиво и ответственно. Иногда это бывает уместно, но тут важно не переборщить. Примеры выше приведены только для иллюстрации, а не как предложение писать такой код при каждом удобном случае.

Ну а если мы хотим округлять до заданного числа знаков после запятой, но при этом использовать метод «округления до четного»? Тогда можно было бы передать параметр в метод round_down:

```
class Float

   def roundf_down(places = 0)
      shift = 10 ** places
      num = (self * shift).round_down / shift.to_f
      num.round(places)
```

```
    end
end

a = 6.125
b = 6.135
x = a.roundf_down(2)    # 6.12
y = b.roundf_down(2)    # 6.14
z = b.roundf_down       # 6
```

У метода roundf_down есть некоторые ограничения. Большое число с плавающей точкой может привести к переполнению при умножении на большую степень 10. На этот случай следовало бы предусмотреть проверку ошибок.

5.4. Сравнение чисел с плавающей точкой

Печально, но факт: в компьютере числа с плавающей точкой представляются не точно. В идеальном мире следующий код напечатал бы «да», но на всех машинах, где мы его запускали, печатается «нет»:

```
x = 1000001.0/0.003
y = 0.003*x
if y == 1000001.0
  puts "да"
else
  puts "нет"
end
```

Объясняется это тем, что для хранения числа с плавающей точкой выделено конечное число битов, а с помощью любого, сколь угодно большого, но конечного числа битов, нельзя представить периодическую десятичную дробь с бесконечным числом знаков после запятой.

Из-за этой неустранимой неточности при сравнении чисел с плавающей точкой мы можем оказаться в ситуации (продемонстрированной выше), когда с практической точки зрения два числа равны, но аппаратура упрямо считает их различными.

Ниже показан простой способ выполнения сравнения с «поправкой», когда числа считаются равными, если отличаются не более чем на величину, задаваемую программистом:

```
class Float

  EPSILON = 1e-6   # 0.000001

  def ==(x)
    (self-x).abs < EPSILON
  end

end


x = 1000001.0/0.003
y = 0.003*x
```

```
if y == 1.0    # Пользуемся новым оператором ==
  puts "да"    # Теперь печатается "да"
else
  puts "нет"
end
```

В зависимости от ситуации может понадобиться задавать разные погрешности. Для этого определим в классе Float новый метод nearly_equal?. (При таком выборе имени мы избежим конфликта со стандартными методами equal? и eql?; последний, кстати, вообще не следует переопределять).

```
class Float

  EPSILON = 1e-6

  def nearly_equal?(x, tolerance=EPSILON)
    (self-x).abs < tolerance
  end

end

flag1 = (3.1416).nearly_equal? Math::PI          # false
flag2 = (3.1416).nearly_equal?(Math::PI, 0.001)  # true
```

Можно также ввести совершенно новый оператор для приближенного сравнения, назвав его, например =~.

Имейте в виду, что это нельзя назвать настоящим решением. При последовательных вычислениях погрешность накапливается. Если вам совершенно необходимы числа с плавающей точкой, приготовьтесь к неточности. (См. также разделы 5.8 «Использование класса BigDecimal» и 5.9 «Работа с рациональными числами».)

5.5. Форматирование чисел для вывода

Для вывода числа в заданном формате применяется метод printf из модуля Kernel. Он практически не отличается от одноименной функции в стандартной библиотеке C. Дополнительную информацию см. в документации по методу printf.

```
x = 345.6789
i = 123
printf("x = %6.2f\n", x)    # x = 345.68
printf("x = %9.2e\n", x)    # x = 3.457e+02
printf("i = %5d\n", i)      # i =   123
printf("i = %05d\n", i)     # i = 00123
printf("i = %-5d\n", i)     # i = 123
```

Чтобы сохранить результат в строке, а не печатать его немедленно, воспользуйтесь методом sprintf. При следующем обращении возвращается строка:

```
str = sprintf("%5.1f",x)    # "345.7"
```

Наконец, в классе String есть метод %, решающий ту же задачу. Слева от знака % должна стоять форматная строка, а справа – единственный аргумент (или массив значений), результатом является строка.

```
# Порядок вызова: 'формат % значение'
str = "%5.1f" % x            # "345.7"
str = "%6.2f, %05d" % [x,i]  # "345.68, 00123"
```

5.6. Вставка разделителей при форматировании чисел

Возможно, есть и более удачные способы достичь цели, но приведенный ниже код работает. Мы инвертируем строку, чтобы было удобнее выполнять глобальную замену, а в конце инвертируем ее еще раз:

```
def commas(x)
  str = x.to_s.reverse
  str.gsub!(/([0-9]{3})/,"\\1,")
  str.gsub(/,$/,"").reverse
end

puts commas(123)         # "123"
puts commas(1234)        # "1,234"
puts commas(12345)       # "12,435"
puts commas(123456)      # "123,456"
puts commas(1234567)     # "1,234,567"
```

5.7. Работа с очень большими числами

Управлять массами все равно,
что управлять немногими: дело в частях и в числе.

— Сунь-Цзы[1]

При необходимости Ruby позволяет работать с произвольно большими целыми числами. Переход от Fixnum к Bignum производится автоматически, прозрачно для программиста. В следующем разделе результат оказывается настолько большим, что преобразуется из объекта Fixnum в Bignum:

```
num1 = 1000000           # Один миллион (10**6)
num2 = num1*num1         # Один триллион (10**12)
puts num1                # 1000000
puts num1.class          # Fixnum
puts num2                # 1000000000000
puts num2.class          # Bignum
```

[1] Сунь-Цзы «Искусство войны».

Размер Fixnum зависит от машинной архитектуры. Вычисления с объектами Bignum ограничены только объемом памяти и быстродействием процессора. Конечно, они потребляют больше памяти и выполняются несколько медленнее, тем не менее операции над очень большими целыми (сотни знаков) реальны.

5.8. Использование класса BigDecimal

Стандартная библиотека bigdecimal позволяет работать с дробями, имеющими много значащих цифр. Число хранится как массив цифр, а не преобразуется в двоичное представление. Тем самым достижима произвольная точность, естественно, ценой замедления работы.

Чтобы оценить преимущества, рассмотрим следующий простой фрагмент кода, в котором используются числа с плавающей точкой:

```
if (3.2 - 2.0) == 1.2
  puts "равны"
else
  puts "не равны"     # печатается "не равны"!
end
```

В подобной ситуации на помощь приходит класс BigDecimal. Однако в случае бесконечных периодических дробей проблема остается. Другой подход обсуждается в разделе 5.9 «Работа с рациональными числами».

Объект BigDecimal инициализируется строкой. (Объекта типа Float было бы недостаточно, поскольку погрешность вкралась бы еще до начала конструирования BigDecimal.) Метод BigDecimal эквивалентен BigDecimal.new; это еще один особый случай, когда имя метода начинается с заглавной буквы. Поддерживаются обычные математические операции, например, + и *. Отметим, что метод to_s может принимать в качестве параметра форматную строку. Дополнительную информацию см. в документации по классу BigDecimal.

```
require 'bigdecimal'

x = BigDecimal("3.2")
y = BigDecimal("2.0")
z = BigDecimal("1.2")

if (x - y) == z
  puts "равны"   # печатается "равны"!
else
  puts "не равны"
end

a = x*y*z
a.to_s           # "0.768E1" (по умолчанию: научная нотация)
a.to_s("F")      # "7.68"    (обычная запись)
```

Если необходимо, можно задать количество значащих цифр. Метод precs возвращает эту информацию в виде массива, содержащего два числа: количество использованных байтов и максимальное число значащих цифр.

```
x = BigDecimal("1.234",10)
y = BigDecimal("1.234",15)
x.precs                    # [8, 16]
y.precs                    # [8, 20]
```

В каждый момент количество использованных байтов может оказаться меньше максимального. Максимум может также оказаться больше запрошенного вами (поскольку BigDecimal пытается оптимизировать использование внутренней памяти).

У обычных операций (сложение, вычитание, умножение и деление) есть варианты, принимающие в качестве дополнительного параметра число значащих цифр. Если результат содержит больше значащих цифр, чем указано, производится округление до заданного числа знаков.

```
a = BigDecimal("1.23456")
b = BigDecimal("2.45678")

# В комментариях "BigDecimal:objectid" опущено
c  = a+b          # <'0.369134E1',12(20)>
c2 = a.add(b,4)   # <'0.3691E1',8(20)>

d  = a-b          # <'-0.122222E1',12(20)>
d2 = a.sub(b,4)   # <'-0.1222E1',8(20)>

e  = a*b          # <'0.3033042316 8E1',16(36)>
e2 = a.mult(b,4)  # <'0.3033E1',8(36)>

f  = a/b          # <'0.5025114173 8372992290 7221E0',24(32)>
f2 = a.div(b,4)   # <'0.5025E0',4(16)>
```

В классе BigDecimal определено и много других функций, например: floor, abs и т. д. Как и следовало ожидать, имеются операторы % и **, а также операторы сравнения, к примеру, <). Оператор == не умеет округлять свои операнды, эта обязанность возлагается на программиста.

В модуле BigMath определены константы E и PI с произвольной точностью. (На самом деле, это методы, а не константы.) Там же определены функции sin, cos, exp и прочие; все они принимают число значащих цифр в качестве параметра.

Следующие подбиблиотеки являются дополнениями к BigDecimal.

- bigdecimal/math – модуль BigMath;
- bigdecimal/jacobian – методы для вычисления матрицы Якоби;
- bigdecimal/ludcmp – модуль LUSolve, разложение матрицы в произведение верхнетреугольной и нижнетреугольной;
- bigdecimal/newton – методы nlsolve и norm.

В настоящей главе эти подбиблиотеки не описываются. Для получения дополнительной информации обратитесь к сайту ruby-doc.org или любому подробному справочному руководству.

5.9. Работа с рациональными числами

Класс Rational позволяет (во многих случаях) производить операции с дробями с «бесконечной» точностью, но лишь если это настоящие рациональные числа (то

есть частное от деления двух целых чисел). К иррациональным числам, например π, e или квадратный корень из 2, он неприменим.

Для создания рационального числа мы вызываем специальный метод Rational (еще один из немногих методов, чье имя начинается с заглавной буквы, обычно такие методы служат для преобразования данных или инициализации).

```
r = Rational(1,2)   # 1/2 или 0.5
s = Rational(1,3)   # 1/3 или 0.3333...
t = Rational(1,7)   # 1/7 или 0.14...
u = Rational(6,2)   # "то же самое, что" 3.0
z = Rational(1,0)   # ошибка!
```

Результатом операции над двумя рациональными числами, как правило, снова является рациональное число.

```
r+t          #  Rational(9, 14)
r-t          #  Rational(5, 14)
r*s          #  Rational(1, 6)
r/s          #  Rational(3, 2)
```

Вернемся к примеру, на котором мы демонстрировали неточность операций над числами с плавающей точкой (см. раздел 5.4). Ниже мы выполняем те же действия над рациональными, а не вещественными числами и получаем «математически ожидаемый» результат:

```
x = Rational(1000001,1)/Rational(3,1000)
y = Rational(3,1000)*x
if y == 1000001.0
  puts "да"          # Теперь получаем "да"!
else
  puts "нет"
end
```

Конечно, не любая операция дает рациональное же число в качестве результата:

```
x = Rational(9,16)     #  Rational(9, 16)
Math.sqrt(x)           #  0.75
x**0.5                 #  0.75
x**Rational(1,2)       #  0.75
```

Однако библиотека mathn в какой-то мере изменяет это поведение (см. раздел 5.12).

5.10. Перемножение матриц

Стандартная библиотека matrix предназначена для выполнения операций над числовыми матрицами. В ней определено два класса: Matrix и Vector.

Следует также знать о прекрасной библиотеке NArray, которую написал Масахиро Танака (Masahiro Tanaka) (ее можно найти в виде gem-пакета на сайте rubygems.org). Хотя эта библиотека не относится к числу стандартных, она широко известна и очень полезна. Если ваша программа предъявляет повышенные требования

к быстродействию, нуждается в особом представлении данных или должна выполнять быстрое преобразование Фурье, то обязательно ознакомьтесь с этим пакетом. Впрочем, для типичных применений стандартной библиотеки matrix должно хватить, поэтому именно ее мы и рассмотрим.

Чтобы создать матрицу мы, конечно же, обращаемся к методу класса. Сделать это можно несколькими способами. Самый простой – вызвать метод Matrix.[] и перечислить строки в виде массивов. Ниже мы записали вызов на нескольких строчках, но, разумеется, это необязательно:

```
m = Matrix[[1,2,3],
           [4,5,6],
           [7,8,9]]
```

Вместо этого можно вызвать метод rows, передав ему массив массивов (в этом случае «дополнительные» скобки необходимы). Необязательный параметр copy, по умолчанию равный true, указывает, надо ли скопировать переданные массивы или просто сохранить на них ссылки. Оставляйте значение true, если нужно защитить исходные массивы от изменения, и задавайте false, если это несущественно и вы хотите сэкономить немного памяти.

```
row1 = [2,3]
row2 = [4,5]
m1 = Matrix.rows([row1,row2])          # copy=true
m2 = Matrix.rows([row1,row2],false)    # не копировать
row1[1] = 99                           # Теперь изменим row1
p m1                                   # Matrix[[2, 3], [4, 5]]
p m2                                   # Matrix[[2, 99], [4, 5]]
```

Можно задать матрицу и путем перечисления столбцов, если воспользоваться методом columns. Ему параметр copy не передается, потому что столбцы в любом случае расщепляются, так как во внутреннем представлении матрица хранится по строкам:

```
m1 = Matrix.rows([[1,2],[3,4]])
m2 = Matrix.columns([[1,3],[2,4]])    # m1 == m2
```

Предполагается, что все матрицы прямоугольные. Если вы создадите матрицу, в которой отдельные строки или столбцы длиннее или короче остальных, то сразу же получите ошибку с сообщением о том, что входная матрица не прямоугольная.

Некоторые специальные матрицы, особенно квадратные, конструируются проще. Так, тождественную матрицу конструирует метод identity (или его синонимы I и unit):

```
im1 = Matrix.identity(3)  # Matrix[[1,0,0],[0,1,0],[0,0,1]]
im2 = Matrix.I(3)         # то же самое
im3 = Matrix.unit(3)      # то же самое
```

Более общий метод scalar строит диагональную матрицу, в которой все элементы на диагонали одинаковы, но не обязательно равны 1:

```
sm = Matrix.scalar(3,8)   # Matrix[[8,0,0],[0,8,0],[0,0,8]]
```

Еще более общим является метод diagonal, который формирует диагональную матрицу с произвольными элементами (ясно, что параметр, задающий размерность, в этом случае не нужен).

```
dm = Matrix.diagonal(2,3,7)  # Matrix[[2,0,0],[0,3,0],[0,0,7]]
```

Метод zero создает нулевую матрицу заданной размерности (все элементы равны 0):

```
zm = Matrix.zero(3)  # Matrix[[0,0,0],[0,0,0],[0,0,0]]
```

Понятно, что методы identity, scalar, diagonal и zero создают квадратные матрицы.

Чтобы создать матрицу размерности 1×N или N×1, воспользуйтесь методом row_vector или column_vector соответственно.

```
a = Matrix.row_vector(2,4,6,8)      # Matrix[[2,4,6,8]]
b = Matrix.column_vector(6,7,8,9)   # Matrix[[6],[7],[8],[9]]
```

К отдельным элементам матрицы можно обращаться, указывая индексы в квадратных скобках (оба индекса заключаются в одну пару скобок). Отметим, что не существует метода [ ]=. По той же причине, по которой его нет в классе Fixnum: матрицы – неизменяемые объекты (такое решение было принято автором библиотеки).

```
m = Matrix[[1,2,3],[4,5,6]]
puts m[1,2]    # 6
```

Индексация начинается с 0, как и для массивов в Ruby. Возможно, это противоречит вашему опыту работы с матрицами, но индексация с 1 в качестве альтернативы не предусмотрена. Можно реализовать эту возможность самостоятельно:

```
# Наивный подход... не поступайте так!

class Matrix
  alias bracket []

  def [](i,j)
    bracket(i-1,j-1)
  end
end

m = Matrix[[1,2,3],[4,5,6],[7,8,9]]
p m[2,2]      # 5
```

На первый взгляд, этот код должен работать. Большинство операций над матрицами даже будет давать правильный результат при такой индексации. Так в чем же проблема? В том, что мы не знаем деталей внутренней реализации класса Matrix. Если в нем для доступа к элементам матрицы всегда используется собственный метод [], то все будет хорошо. Но если где-нибудь имеются прямые обращения к внутреннему массиву или применяются иные оптимизированные решения, то возникнет ошибка. Поэтому, решившись на такой трюк, вы должны тщательно протестировать новое поведение.

На практике придется изменить также методы row и vector. В них индексы тоже начинаются с 0, но метод [] не вызывается. Я не проверял, что еще придется модифицировать.

Иногда необходимо узнать размерность или форму матрицы. Для этого есть разные методы, например, row_size и column_size, которые возвращают число строк и столбцов соответственно.

Для вырезания части матрицы имеется несколько методов. Метод row_vectors возвращает массив объектов класса Vector, представляющих строки. (См. обсуждение класса Vector ниже.) Метод column_vectors работает аналогично, но для столбцов. Наконец, метод minor возвращает матрицу меньшего размера; его параметрами являются либо четыре числа (нижняя и верхняя границы номеров строк и столбцов), либо два диапазона.

```
m = Matrix[[1,2,3,4],[5,6,7,8],[6,7,8,9]]

rows = m.row_vectors        # три объекта Vector
cols = m.column_vectors     # четыре объекта Vector
m2 = m.minor(1,2,1,2)       # Matrix[[6,7,],[7,8]]
m3 = m.minor(0..1,1..3)     # Matrix[[[2,3,4],[6,7,8]]
```

К матрицам применимы обычные операции: сложение, вычитание, умножение и деление. Для выполнения некоторых из них должны соблюдаться ограничения на размеры матриц-операндов, в противном случае будет возбуждено исключение (например, при попытке перемножить матрицы размерностей 3×3 и 4×4).

Поддерживаются стандартные преобразования: inverse (обращение), transpose (транспонирование) и determinant (вычисление определителя). Для целочисленных матриц определитель лучше вычислять с помощью библиотеки mathn (раздел 5.12).

Класс Vector – это, по существу, частный случай одномерной матрицы. Его объект можно создать с помощью методов [ ] или elements; в первом случае параметром является развернутый массив, а во втором – обычный массив и необязательный параметр copy (по умолчанию равный true).

```
arr = [2,3,4,5]
v1 = Vector[*arr]                  # Vector[2,3,4,5]
v2 = Vector.elements(arr)          # Vector[2,3,4,5]
v3 = Vector.elements(arr,false)    # Vector[2,3,4,5]
arr[2] = 7                         # Теперь v3 - Vector[2,3,7,5]
```

Метод covector преобразует вектор длины N в матрицу размерности N 1 (выполняя попутно транспонирование).

```
v = Vector[2,3,4]
m = v.covector     # Matrix[[2,3,4]]
```

Поддерживается сложение и вычитание векторов одинаковой длины. Вектор можно умножать на матрицу и на скаляр. Все эти операции подчиняются обычным математическим правилам.

```
v1 = Vector[2,3,4]
v2 = Vector[4,5,6]
v3 = v1 + v2          # Vector[6,8,10]
v4 = v1*v2.covector   # Matrix[[8,10,12],[12,15,18],[16,20,24]]
v5 = v1*5             # Vector[10,15,20]
```

Имеется метод inner_product (скалярное произведение):

```
v1 = Vector[2,3,4]
v2 = Vector[4,5,6]
x  = v1.inner_product(v2)   # 47
```

Дополнительную информацию о классах Matrix и Vector можно найти в документации по классу в сети или воспользовавшись командой ri.

5.11. Комплексные числа

Стандартная библиотека complex предназначена для работы с комплексными числами в Ruby. Большая ее часть не требует пояснений.

Для создания комплексного числа применяется следующая несколько необычная нотация:

```
z = Complex(3,5)      # 3+5i
```

Необычно в ней то, что имя метода совпадает с именем класса. В данном случае наличие скобок указывает на то, что это вызов метода, а не ссылка на константу. Вообще говоря, имена методов не похожи на константы, и я не рекомендую начинать имена методов с заглавной буквы, разве что в подобных специальных случаях. (Отметим, что имеются также методы Integer и Float; вообще, имена, начинающиеся с заглавной буквы, зарезервированы для методов, которые выполняют преобразование данных и аналогичные действия.)

Метод im преобразует вещественное число в мнимое (по существу, умножая его на i). Поэтому представлять комплексные числа можно и с помощью более привычной нотации:

```
a = 3.im       # 3i
b = 5 - 2.im   # 5-2i
```

Если вас больше интересуют полярные координаты, то можно обратиться к методу polar:

```
z = Complex.polar(5,Math::PI/2.0)   # радиус, угол
```

В классе Complex имеется также константа I, которая представляет число i – квадратный корень из минус единицы:

```
z1 = Complex(3,5)
z2 = 3 + 5*Complex::I    # z2 == z1
```

После загрузки библиотеки complex некоторые стандартные математические функции изменяют свое поведение. Тригонометрические функции – sin, sinh, tan и tanh – (а также некоторые другие, например, exp и log) начинают принимать еще и комплексные аргументы. Некоторые функции, например, sqrt даже возвращают комплексные числа в качестве результата.

```
x = Math.sqrt(Complex(3,5))   # приближенно Complex(2.1013, 1.1897)
  = Math.sqrt(-1)             # Complex(0,1)
```

Дополнительную информацию ищите в документации по классу `Complex`.

5.12. Библиотека mathn

В программах, выполняющих большой объем математических вычислений, очень пригодится замечательная библиотека `mathn`, которую написал Кейдзу Исидзука (Keiju Ishitsuka). В ней есть целый ряд удобных методов и классов, кроме того, она унифицирует все классы Ruby для работы с числами так, что они начинают хорошо работать совместно.

Простейший способ воспользоваться этой библиотекой – включить ее с помощью директивы `require` и забыть. Поскольку она сама включает библиотеки `complex`, `rational` и `matrix` (в таком порядке), то вы можете этого не делать. В общем случае библиотека `mathn` пытается вернуть «разумные» результаты вычислений. Например, при извлечении квадратного корня из `Rational` будет возвращен новый объект `Rational`, если это возможно, в противном случае `Float`. В таблице 5.1 приведены некоторые последствия загрузки этой библиотеки.

Таблица 5.1. Результаты вычислений в случае наличия и отсутствия библиотеки mathn

Выражение	Без mathn	С mathn
`1/2`	`0`	`Rational(1,2)`
`Math.sqrt(64/25)`	`1.4142...`	`Rational(8,5)`
`Rational(1,10).inspect`	`Rational(1,10)`	`1/10`
`Math.sqrt(Rational(9,16))`	`0.75`	`Rational(3,4)`
`Matrix.identity(3)/3`	`Matrix[[0,0,0], [0,0,0], [0,0,0]]`	`Matrix[[1/3,0,0], [0,1/3,0], [0,0,1/3]]`

Библиотека `mathn` добавляет методы `**` и `power2` в класс `Rational`. Она изменяет поведение метода `Math.sqrt` и добавляет метод `Math.rsqrt`, умеющий работать с рациональными числами.

См. также разделы 5.13 и 5.14.

5.13. Разложение на простые множители, вычисление НОД и НОК

В библиотеке `mathn` определены также некоторые новые методы в классе `Integer`. Так, метод `gcd2` служит для нахождения наибольшего общего делителя (НОД) объекта, от имени которого он вызван, и другого числа.

```
n = 36.gcd2(120)     # 12
x = 237.gcd2(79)     # 79
```

Метод `prime_division` выполняет разложение на простые множители. Результат возвращается в виде массива массивов, в котором каждый вложенный массив содержит простое число и показатель степени, с которым оно входит в произведение:

```
factors = 126.prime_division  # [[2,1], [3,2], [7,1]]
                              # т.е. 2**1 * 3**2 * 7**1
```

Имеется также метод класса Integer.from_prime_division, который восстанавливает исходное число из его сомножителей. Это именно метод класса, потому что выступает в роли «конструктора» целого числа.

```
factors = [[2,1],[3,1],[7,1]]
num = Integer.from_prime_division(factors)    # 42
```

Ниже показано, как разложение на простые множители можно использовать для отыскания наименьшего общего кратного (НОК) двух чисел:

```
require 'mathn'

class Integer
  def lcm(other)
    pf1 = self.prime_division.flatten
    pf2 = other.prime_division.flatten
    h1 = Hash[*pf1]
    h2 = Hash[*pf2]
    hash = h2.merge(h1) {|key,old,new| [old,new].max }
    Integer.from_prime_division(hash.to_a)
  end
end

p 15.lcm(150)      # 150
p 2.lcm(3)         # 6
p 4.lcm(12)        # 12
p 200.lcm(30)      # 600
```

5.14. Простые числа

В библиотеке mathn есть класс для порождения простых чисел. Итератор each возвращает последовательные простые числа в бесконечном цикле. Метод succ порождает следующее простое число.

Вот, например, два способа получить первые 100 простых чисел:

```
require 'mathn'

list = []
gen = Prime.new
gen.each do |prime|
  list << prime
  break if list.size == 100
end

# или:

list = []
gen = Prime.new
100.times { list << gen.succ }
```

В следующем фрагменте проверяется, является ли данное число простым. Отметим, что если число велико, а машина медленная, то на выполнение может уйти заметное время:

```
require 'mathn'

class Integer
  def prime?
    max = Math.sqrt(self).ceil
    max -= 1 if max % 2 == 0
    pgen = Prime.new
    pgen.each do |factor|
      return false if self % factor == 0
      return true if factor > max
    end
  end
end

31.prime?           # true
237.prime?          # false
1500450271.prime?   # true
```

5.15. Явные и неявные преобразования чисел

Программисты, только начинающие изучать Ruby, часто удивляются, зачем нужно два метода `to_i` и `to_int` (и аналогичных им `to_f` и `to_flt`). В общем случае метод с коротким именем применяется для явных преобразований, а метод с длинным именем – для неявных.

Что это означает? Во-первых, в большинстве классов определены явные конверторы, но нет неявных.

Во-вторых, в своих собственных классах вы, скорее всего, будете определять неявные конверторы, но не станете вызывать их вручную (если только не заняты написанием «клиентского» кода или библиотеки, которая пытается не конфликтовать с внешним миром).

Следующий пример, конечно, надуманный. В нем определен класс `MyClass`, который возвращает константы из методов `to_i` и `to_int`. Такое поведение лишено смысла, зато иллюстрирует идею:

```
class MyClass

  def to_i
    3
  end

  def to_int
    5
  end

end
```

Желая явно преобразовать объект класса MyClass в целое число, мы вызовем метод to_i:

```
m = MyClass.new
x = m.to_i        # 3
```

Но при передаче объекта MyClass какой-нибудь функции, ожидающей целое число, будет неявно вызван метод to_int. Предположим, к примеру, что мы хотим создать массив с известным начальным числом элементов. Метод Array.new может принять целое, но что если вместо этого ему будет передан объект MyClass?

```
m = MyClass.new
a = Array.new(m)    # [nil,nil,nil,nil,nil]
```

Как видите, метод new оказался достаточно «умным», чтобы вызвать to_int и затем создать массив из пяти элементов.

Дополнительную информацию о поведении в другом контексте (строковом) см. в разделе 2.16 «Явные и неявные преобразования». См. также раздел 5.16.

5.16. Приведение числовых значений

Приведение можно считать еще одним видом неявного преобразования. Если некоторому методу (например, +) передается аргумент, которого он не понимает, он пытается привести объект, от имени которого вызван, и аргумент к совместимым типам, а затем сложить их. Принцип использования метода coerce в вашем собственном классе понятен из следующего примера:

```
class MyNumberSystem

def +(other)
   if other.kind_of?(MyNumberSystem)
     result = some_calculation_between_self_and_other
     MyNumberSystem.new(result)
   else
     n1, n2 = other.coerce(self)
     n1 + n2
   end
 end

end
```

Метод coerce возвращает массив из двух элементов, содержащий аргумент и вызывающий объект, приведенные к совместимым типам.

В данном примере мы полагаемся на то, что приведение выполнит тип аргумента. Но если бы хотим быть законопослушными гражданами, то должны реализовать приведение в своем классе, сделав его пригодным для работы с другими типами чисел. Для этого нужно знать, с какими типами мы в состоянии работать непосредственно и при необходимости выполнять приведение к одному из этих типов. Если мы не можем сделать это самостоятельно, то должны обратиться за помощью к родительскому классу.

```
def coerce(other)
  if other.kind_of?(Float)
    return other, self.to_f
  elsif other.kind_of?(Integer)
    return other, self.to_i
  else
    super
  end
end
```

Разумеется, это будет работать только, если наш объект реализует методы `to_i` и `to_f`.

Метод `coerce` можно применить для реализации автоматического преобразования строк в числа, как в языке Perl:

```
class String

  def coerce(n)
    if self['.']
      [n, Float(self)]
    else
      [n, Integer(self)]
    end
  end

end

x = 1 + "23"        # 24
y = 23 * "1.23"     # 28.29
```

Это может привести к различным неожиданным результатам, поэтому мы предостерегаем от такой практики. Но рекомендуем автоматически приводить другие числовые типы при создании любого класса для работы с числовыми данными.

5.17. Поразрядные операции над числами

Иногда требуется работать с двоичным представлением объекта `Fixnum`. На прикладном уровне такая необходимость возникает не часто, но все-таки возникает.

Ruby обладает всеми средствами для таких операций. Для удобства числовые константы можно записывать в двоичном, восьмеричном или шестнадцатеричном виде. Поразрядным операциям И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ и НЕ соответствуют операторы `&`, `|`, `^` и `~`.

```
x = 0377              # Восьмеричное       (десятичное 255)
y = 0b00100110        # Двоичное           (десятичное 38)
z = 0xBEEF            # Шестнадцатеричное (десятичное 48879)

a = x | z             # 48895 (поразрядное ИЛИ)
b = x & z             #   239 (поразрядное И)
c = x ^ z             # 48656 (поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ)
d = ~ y               #   -39 (отрицание или дополнение до 1)
```

Метод экземпляра size позволяет узнать размер слова для той машины, на которой исполняется программа.

```
bytes = 1.size        # Для конкретной машины возвращает 4
```

Имеются операторы сдвига влево и вправо (<< и >> соответственно). Это логические операторы сдвига, они не затрагивают знаковый бит (хотя оператор >> распространяет его).

```
x = 8
y = -8

a = x >> 2          # 2
b = y >> 2          # -2
c = x << 2          # 32
d = y << 2          # -32
```

Конечно, если сдвиг настолько велик, что дает нулевое значение, то знаковый бит теряется, поскольку –0 и 0 – одно и то же.

Квадратные скобки позволяют трактовать числа как битовые массивы. Бит с номером 0 всегда является младшим, вне зависимости от порядка битов в конкретной машинной архитектуре.

```
x = 5             # То же, что 0b0101
a = x[0]          # 1
b = x[1]          # 0
c = x[2]          # 1
d = x[3]          # 0
# И так далее     # 0
```

Присваивать новые значения отдельным битам с помощью такой нотации невозможно (поскольку Fixnum хранится как непосредственное значение, а не как ссылка на объект). Но можно имитировать это действие путем сдвига 1 влево на нужное число позиций с последующим выполнением операции ИЛИ или И.

```
# Выполнить присваивание x[3] = 1 нельзя,
# но можно поступить так:
x |= (1<<3)
# Выполнить присваивание x[4] = 0 нельзя,
# но можно поступить так:
x &= ~(1<<4)
```

5.18. Преобразование системы счисления

Ясно, что любое целое число можно представить в любой системе счисления, поскольку хранятся они в двоичном виде. Мы знаем, что Ruby умеет работать с целыми константами, записанными в любой из четырех наиболее популярных систем. Следовательно, разговор о преобразовании системы счисления может вестись только применительно к числам, записанным в виде строк.

Вопрос о преобразовании строки в целое рассмотрен в разделе 2.24.

Для преобразования числа в строку проще всего воспользоваться методом to_s, которому можно еще передать основание системы счисления. По умолчанию оно равно 10, но в принципе может быть любым вплоть до 36 (когда задействованы все буквы латинского алфавита).

```
237.to_s(2)          # "11101101"
237.to_s(5)          # "1422"
237.to_s(8)          # "355"
237.to_s             # "237"
237.to_s(16)         # "ed"
237.to_s(30)         # "7r"
```

Другой способ – обратиться к методу % класса String:

```
hex = "%x" % 1234    # "4d2"
oct = "%o" % 1234    # "2322"
bin = "%b" % 1234    # "10011010010"
```

Метод sprintf тоже годится:

```
str = sprintf(str,"Nietzsche is %x\n",57005)
# str теперь равно: "Nietzsche is dead\n"
```

Если нужно сразу же вывести преобразованное в строку значение, то подойдет и метод printf.

5.19. Извлечение кубических корней, корней четвертой степени и т.д.

В Ruby встроена функция извлечения квадратного корня (Math.sqrt), поскольку именно она применяется чаще всего. А если надо извлечь корень более высокой степени? Если вы еще не забыли математику, то эта задача не вызовет затруднений.

Можно, например, воспользоваться логарифмами. Напомним, что e в степени x – обратная функция к натуральному логарифму x и что умножение чисел эквивалентно сложению их логарифмов.

```
x = 531441
cuberoot = Math.exp(Math.log(x)/3.0)      # 81.0
fourthroot = Math.exp(Math.log(x)/4.0)    # 27.0
```

Но можно просто использовать дробные показатели степени (оператор возведения в степень принимает в качестве аргумента произвольное целое число или число с плавающей точкой).

```
include Math

y = 4096
cuberoot = y**(1.0/3.0)       # 16.0
fourthroot = y**(1.0/4.0)     # 8.0
fourthroot = sqrt(sqrt(y))    # 8.0 (то же самое)
twelfthroot = y**(1.0/12.0)   # 2.0
```

Отметим, что во всех примерах мы пользовались при делении числами с плавающей точкой (чтобы избежать отбрасывания дробной части).

5.20. Определение порядка байтов

Интересно, что производители компьютеров никак не могут договориться, в каком порядке лучше хранить двоичные байты. Следует ли размещать старший бит по большему или по меньшему адресу? При передаче сообщения по проводам нужно сначала посылать старший или младший бит?

Хотите верьте, хотите нет, но решение не произвольно. Существуют убедительные аргументы в пользу обеих точек зрения (обсуждать их здесь мы не будем).

Вот уже больше тридцати лет как для описания противоположных позиций применяются термины *остроконечный* (little-endian) и *тупоконечный* (big-endian). Кажется, впервые их употребил Дэнни Коэн (Danny Cohen); см. его классическую статью "On Holy Wars and a Plea for Peace" (IEEE Computer, October 1981). Взяты они из романа Джонатана Свифта «Путешествия Гулливера».

Обычно нам безразличен порядок байтов в конкретной машинной архитектуре. Но как быть, если все-таки его нужно знать?

Можно воспользоваться показанным ниже методом. Он возвращает одну из строк `LITTLE`, `BIG` или `OTHER`. Решение основано на том факте, что директива `l` выполняет упаковку в машинном формате, а директива `N` распаковывает в сетевом порядке байтов (по определению, тупоконечном).

```
def endianness
  num=0x12345678
  little = "78563412"
  big    = "12345678"
  native = [num].pack('l')
  netunpack = native.unpack('N')[0]
  str = "%8x" % netunpack
  case str
    when little
      "LITTLE"
    when big
      "BIG"
    else
      "OTHER"
  end
end

puts endianness   # В данном случае печатается "LITTLE"
```

Этот прием может оказаться удобным, если, например, вы работаете с двоичными данными (скажем, отсканированным изображением), импортированными из другой системы.

5.21. Численное вычисление определенного интеграла

Я очень хорошо владею дифференциальным
и интегральным исчислением...

У. С. Джильберт «Пираты Пензанса», акт 1

Для приближенного вычисления определенного интеграла имеется проверенная временем техника. Любой студент, изучавший математический анализ, вспомнит, что она называется суммой Римана.

Приведенный ниже метод `integrate` принимает начальное и конечное значения зависимой переменной, а также приращение. Четвертый параметр (который на самом деле параметром не является) – это блок. В блоке должно вычисляться значение функции от переданной в него зависимой переменной. (Здесь слово «переменная» употребляется в математическом, а не программистском смысле.) Необязательно отдельно определять функцию, которая вызывается в блоке, но для ясности мы это сделаем.

```
def integrate(x0, x1, dx=(x1-x0)/1000.0)
  x = x0
  sum = 0
  loop do
    y = yield(x)
    sum += dx * y
    x += dx
    break if x > x1
  end
  sum
end

def f(x)
  x**2
end

z = integrate(0.0,5.0) { x| f(x)

puts z, "\n"           # 41.72918749999876
```

Здесь мы опираемся на тот факт, что блок возвращает значение, которое может быть получено с помощью `yield`. Кроме того, сделаны некоторые допущения. Во-первых, мы предполагаем, что `x0` меньше `x1` (в противном случае получится бесконечный цикл). Читатель легко сам устранит подобные огрехи. Во-вторых, мы считаем, что функцию можно вычислить в любой точке заданной области. Если это не так, мы получим хаотическое поведение. (Впрочем, подобные функции все равно,

как правило, не интегрируемы, по крайней мере, на указанном интервале. В качестве примера возьмите функцию f(x)=x/(x-3) в точке x=3.)

Призвав на помощь полузабытые знания об интегральном исчислении, мы могли бы вычислить, что в данном случае результат равен примерно 41.666 (5 в кубе, поделенное на 3). Почему же ответ не так точен, как хотелось бы? Из-за выбранного размера приращения; чем меньше величина dx, тем точнее результат (ценой увеличения времени вычисления).

Напоследок отметим, что подобная методика более полезна для действительно сложных функций, а не таких простых, как f(x) = x**2.

5.22. Тригонометрия в градусах, радианах и градах

При измерении дуг математической, а заодно и «естественной» единицей измерения является радиан. По определению, угол в один радиан соответствует длине дуги, равной радиусу окружности. Немного поразмыслив, легко понять, что угол 2π радиан соответствует всей окружности.

Дуговой градус, которым мы пользуемся в повседневной жизни, – пережиток древневавилонской системы счисления по основанию 60, в ней окружность делится на 360°. Менее известна псевдометрическая единица измерения *град*, определенная так, что прямой угол составляет 100° (а вся окружность – 400°).

При вычислении тригонометрических функций в языках программирования по умолчанию чаще всего используются радианы, и Ruby в этом отношении не исключение. Но мы покажем, как производить вычисления и в градусах, и в градах для тех читателей, которые по образованию не инженеры, а по происхождению не древние вавилоняне.

Поскольку число любых угловых единиц в окружности – константа, можно легко переходить от одних единиц к другим. Мы определим соответствующие константы и будем пользоваться ими в коде. Для удобства поместим их в модуль Math.

```
module Math

  RAD2DEG  = 360.0/(2.0*PI)  # Радианы в градусы
  RAD2GRAD = 400.0/(2.0*PI)  # Радианы в грады

end
```

Теперь можно определить и новые тригонометрические функции. Поскольку мы всегда преобразуем в радианы, то будем делить на определенные выше коэффициенты. Можно было бы поместить определения функций в тот же модуль Math, но мы этого делать не стали.

```
def sin_d(theta)
  Math.sin (theta/Math::RAD2DEG)
end

def sin_g(theta)
  Math.sin (theta/Math::RAD2GRAD)
end
```

Функции cos и tan можно было бы определить аналогично.

С функцией atan2 дело обстоит несколько сложнее. Она принимает два аргумента (длины противолежащей и прилежащей сторон прямоугольного треугольника). Поэтому мы преобразуем результат, а не аргумент:

```
def atan2_d(y,x)
  Math.atan2(y,x)/Math::RAD2DEG
end

def atan2_g(y,x)
  Math.atan2(y,x)/Math::RAD2GRAD
end
```

5.23. Вычисление логарифмов по произвольному основанию

Чаще всего мы пользуемся натуральными логарифмами (по основанию e, часто натуральный логарифм обозначается как ln), иногда также десятичными (по основанию 10). Эти функции реализованы в методах Math.log и Math.log10 соответственно.

В информатике, а в особенности в таких ее областях, как кодирование и теория информации, обычно применяются логарифмы по основанию 2. Например, так вычисляется минимальное число битов, необходимых для представления числа. Определим функцию с именем log2:

```
def log2(x)
  Math.log(x)/Math.log(2)
end
```

Ясно, что обратной к ней является функция 2**x (как обращением log x служит Math::E**x или Math.exp(x)).

Эта идея обобщается на любое основание. В том маловероятном случае, если вам понадобится логарифм по основанию 7, можно поступить так:

```
def log7(x)
  Math.log(x)/Math.log(7)
end
```

На практике знаменатель нужно вычислить один раз и сохранить в виде константы.

5.24. Вычисление среднего, медианы и моды набора данных

Пусть дан массив x, вычислим среднее значение по всем элементам массива. На самом деле, есть три общеупотребительных разновидности среднего значения. *Среднее арифметическое* – это то, что мы называем средним в обыденной жизни. *Среднее гармоническое* – это число элементов, поделенное на сумму обратных к ним.

И, наконец, *среднее геометрическое* – это корень *n*-ой степени из произведения *n* значений. Вот эти определения, воплощенные в коде:

```
def mean(x)
  sum=0
  x.each {|v| sum += v}
  sum/x.size
end

def hmean(x)
  sum=0
  x.each {|v| sum += (1.0/v)}
  x.size/sum
end

def gmean(x)
  prod=1.0
  x.each {|v| prod *= v}
  prod**(1.0/x.size)
end

data = [1.1, 2.3, 3.3, 1.2, 4.5, 2.1, 6.6]

am = mean(data)   # 3.014285714285714
hm = hmean(data)  # 2.1019979464765117
gm = gmean(data)  # 2.5084114744285384
```

Медианой набора данных называется значение, которое оказывается приблизительно в середине отсортированного набора (ниже приведен код для вычисления медианы). Примерно половина элементов набора меньше медианы, а другая половина – больше. Ясно, что такая статистика показательна не для всякого набора.

```
def median(x)
  sorted = x.sort
  mid = x.size/2
  sorted[mid]
end

data = [7,7,7,4,4,5,4,5,7,2,2,3,3,7,3,4]
puts median(data)# 4
```

Мода набора данных – это наиболее часто встречающееся в нем значение. Если такое значение единственно, набор называется *унимодальным*, в противном случае – *мультимодальным*. Мультимодальные наборы более сложны, здесь мы их рассматривать не будем. Интересующийся читатель может обобщить и улучшить приведенный ниже код:

```
def mode(x)
   f = {}      # таблица частот
```

```
    fmax = 0    # максимальная частота
    m = nil     # мода
    x.each do |v|
      f[v] ||= 0
      f[v] += 1
      fmax,m = f[v], v if f[v] > fmax
    end
    return m
end

data = [7,7,7,4,4,5,4,5,7,2,2,3,3,7,3,4]
puts mode(data)  # 7
```

5.25. Дисперсия и стандартное отклонение

Дисперсия – это мера «разброса» значений из набора. (Здесь мы не различаем смещенные и несмещенные оценки.) Стандартное отклонение, которое обычно обозначается буквой σ равно квадратному корню из дисперсии.

```
data = [2, 3, 2, 2, 3, 4, 5, 5, 4, 3, 4, 1, 2]

def variance(x)
  m = mean(x)
  sum = 0.0
  x.each {|v| sum += (v-m)**2 }
  sum/x.size
end

def sigma(x)
  Math.sqrt(variance(x))
end

puts variance(data)    # 1.4615384615384615
puts sigma(data)       # 1.2089410496539776
```

Отметим, что функция `variance` вызывает определенную выше функцию `mean`.

5.26. Вычисление коэффициента корреляции

Коэффициент корреляции – это одна из самых простых и полезных статистических мер. Он измеряет «линейность» набора, состоящего из пар (x, y), и изменяется от –1.0 (полная отрицательная корреляция) до +1.0 (полная положительная корреляция).

Для вычисления воспользуемся функциями `mean` и `sigma` (стандартное отклонение), которые были определены в разделах 5.25 и 5.26. О смысле этого показателя можно прочитать в любом учебнике по математической статистике.

В следующем коде предполагается, что есть два массива чисел одинакового размера:

```
def correlate(x,y)
  sum = 0.0
  x.each_index do |i|
    sum += x[i]*y[i]
  end
  xymean = sum/x.size.to_f
  xmean  = mean(x)
  ymean  = mean(y)
  sx = sigma(x)
  sy = sigma(y)
  (xymean-(xmean*ymean))/(sx*sy)
end

a = [3, 6, 9, 12, 15, 18, 21]
b = [1.1, 2.1, 3.4, 4.8, 5.6]
c = [1.9, 1.0, 3.9, 3.1, 6.9]

c1 = correlate(a,a)          # 1.0
c2 = correlate(a,a.reverse)  # -1.0
c3 = correlate(b,c)          # 0.8221970227633335
```

Приведенная ниже версия отличается лишь тем, что работает с одним массивом, каждый элемент которого – массив, содержащий пару (x, y):

```
def correlate2(v)
  sum = 0.0
  v.each do |a|
    sum += a[0]*a[1]
  end
  xymean = sum/v.size.to_f
  x = v.collect {|a| a[0]}
  y = v.collect {|a| a[1]}
  xmean  = mean(x)
  ymean  = mean(y)
  sx = sigma(x)
  sy = sigma(y)
  (xymean-(xmean*ymean))/(sx*sy)
end

d = [[1,6.1], [2.1,3.1], [3.9,5.0], [4.8,6.2]]

c4 = correlate2(d)    # 0.22778224915306602
```

И, наконец, в последнем варианте предполагается, что пары (x, y) хранятся в хэше. Код основан на предыдущем примере:

```
def correlate_h(h)
  correlate2(h.to_a)
end

e = { 1 => 6.1, 2.1 => 3.1, 3.9 => 5.0, 4.8 => 6.2}

c5 = correlate_h(e) # 0.22778224915306602
```

5.27. Генерирование случайных чисел

Если вас устраивают псевдослучайные числа, вам повезло. Именно они предоставляются в большинстве языков, включая и Ruby.

Метод rand из модуля Kernel возвращает псевдослучайное число x с плавающей точкой такое, что x>=0.0 и x<1.0. Например (вы можете получить совсем другое число):

```
= rand # 0.6279091137
```

Если при вызове задается целочисленный параметр max, то возвращается целое число из диапазона 0...max (верхняя граница не включена). Например:

```
= rand(10)   # 7
```

Чтобы задать начальное значение для генератора случайных чисел, применяется метод srand из модуля Kernel, который принимает один числовой параметр. Если не передавать никакого значения, то метод srand самостоятельно выберет начальное значение, учитывая (среди прочего) текущее время. Если же параметр передан, то именно он и становится начальным значением. Это бывает полезно при тестировании, когда для воспроизводимости результатов многократно вызываемая программа должна получать одну и ту же последовательность псевдослучайных чисел.

```
srand(5)
, j, k = rand(100), rand(100), rand(100)
= 26, 45, 56

srand(5)
, m, n = rand(100), rand(100), rand(100)
= 26, 45, 56
```

5.28. Кэширование функций с помощью метода memoize

Пусть имеется вычислительно сложная математическая функция, которую нужно многократно вызывать по ходу работы программы. Если быстродействие критично и при этом можно пожертвовать небольшим количеством памяти, то имеет смысл сохранить результаты вычисления функции в таблице и обращаться к ней во время выполнения. (Тут неявно предполагается, что функция будет часто вызываться с одними и теми же параметрами, то есть получается, что мы «выбрасываем» результат дорогостоящего вычисления и снова повторяем его позже.) Такая техника иногда называется *запоминанием* (memoizing), отсюда и название библиотеки memoize.

Эта библиотека не входит в стандартный дистрибутив, поэтому придется установить ее вручную.

В следующем примере демонстрируется сложная функция zeta. (Она применяется при решении одной задачи из области популяционной генетики, но вдаваться в объяснения мы не станем.)

```
require 'memoize'
include Memoize

def zeta(x,y,z)
  lim = 0.0001
  gen = 0
  loop do
    gen += 1
    p,q = x + y/2.0, z + y/2.0
    x1, y1, z1 =  p*p*1.0, 2*p*q*1.0, q*q*0.9
    sum = x1 + y1 + z1
    x1 /= sum
    y1 /= sum
    z1 /= sum
    delta = [[x1,x],[y1,y],[z1,z]]
    break if delta.all? {|a,b| (a-b).abs < lim }
    x,y,z = x1,y1,z1
  end

  gen
end

g1 = zeta(0.8,0.1,0.1)
```

Определив функцию, мы можем написать другой метод, который сохраняет результаты вычисления для каждого набора аргументов. Оба метода можно было бы объединить в один, но здесь они показаны по отдельности, чтобы продемонстрировать технику запоминания:

```
def memoized_zeta(x, y, z)
      @results ||= {}
      @results[[x, y, z]] ||= zeta(x, y, z)
end

g2 = memoized_zeta(0.8, 0.1, 0.1)
g3 = memoized_zeta(0.8, 0.1, 0.1)
```

В этом примере значение g3 вычисляться не будет. Оператор ||= проверяет, существует ли уже данное значение (то есть не равно ни nil, ни false). Если переменной присвоено значение, «похожее на true», то оно и возвращается, а правая часть даже не вычисляется

Конечно, ускорение всецело зависит от быстроты вычисления запоминаемого результата, но в принципе разница может оказаться очень существенна. Однако такое ускорение достигается ценой потребления дополнительной памяти под хранение всех предыдущих результатов.

Отметим еще, что библиотека memoize предназначена не только для математических функций. Ее можно использовать для запоминания результатов работы любого вычислительно сложного метода.

5.29. Заключение

В этой главе мы рассмотрели различные представления чисел, в том числе целых (в разных системах счисления) и с плавающей точкой. Мы видели, какие трудности возникают при работе с числами с плавающей точкой, и как можно частично обойти эти трудности, применяя рациональные числа. Мы познакомились с явными и неявными преобразованиями, а также с приведениями типов.

Мы изучили разнообразные способы манипулирования числами, векторами и матрицами. Был приведен обзор стандартных библиотек, полезных для численного анализа, в частности, библиотеки mathn.

Пойдем дальше. В следующей главе мы обсудим два очень характерных для Ruby типа данных: символы и диапазоны.

Глава 6. Символы и диапазоны

Я слышу и забываю. Я вижу и помню.
Я делаю – и понимаю.

– Конфуций

Символы и диапазоны – объекты, весьма характерные для языка Ruby. Они рассматриваются в одной главе не потому, что тесно связаны между собой, а потому, что сказать о них можно не так уж много.

Концепцию символа в Ruby понять не просто. Они напоминают «атомы» в языке Lisp. Вместо того чтобы давать длинное и сложное определение, я расскажу о том, что можно делать с символами и как они применяются. В конце концов, на вопрос «что такое число» можно дать очень глубокомысленный ответ, но нам нужно всего лишь знать, как манипулировать числами.

Диапазоны проще. Это всего лишь представление множества, заданного конечными точками. Аналогичные конструкции существуют в языках Pascal, PHP и даже SQL.

Познакомимся с символами и диапазонами поближе, чтобы понять, как они практически используются в программах на Ruby.

6.1. Символы

Символ в Ruby – это экземпляр класса `Symbol`. Синтаксически он обычно обозначается двоеточием (:), за которым следует идентификатор.

Символ похож на строку, он тоже соответствует последовательности символов. Но отличается от строки тем, что у каждого символа есть только один экземпляр (как и в случае объектов `Fixnum`). Следовательно, имеет место проблема потребления памяти или производительности, о которой нужно помнить. Например, в коде ниже строка `"foo"` представлена в памяти тремя различными объектами, а символ `:foo` – одним объектом (на который есть несколько ссылок):

```
array = ["foo", "foo", "foo", :foo, :foo, :foo]
```

Некоторых смущает двоеточие перед именем символа. Не волнуйтесь, это всего лишь синтаксическое соглашение. У строк, массивов и хэшей есть начальный и конечный ограничители, а у символов – только начальный. Считайте, что это *унарный*, а не бинарный ограничитель. На первый взгляд синтаксис кажется странным, но ничего таинственного в нем нет. На внутреннем уровне каждый сим-

вол представляется в Ruby числом, а не литерами, составляющими символ. Получить это число можно с помощью метода to_i, но в этом редко возникает нужда.

По словам Джима Вайриха, символ – это «объект, у которого есть имя». Остин Зиглер предпочитает говорить об «объекте, который сам является именем». Как бы то ни было, существует взаимно-однозначное соответствие между символами и именами. К чему можно применить имена? Например, к переменным, методам и произвольным константам.

Типичное применение символов – для представления имени переменной или метода. Например, чтобы добавить в класс атрибут, допускающий чтение и изменение, можно поступить следующим образом:

```
class SomeClass
  attr_accessor :whatever
end
```

То же самое можно выразить иначе:

```
class SomeClass
  def whatever
    @whatever
  end
  def whatever=(val)
    @whatever = val
  end
end
```

Другими словами, символ :whatever говорит методу attr_accessor, что методам чтения и установки (а равно и самой переменной экземпляра) следует присвоить имена, определяемые указанным символом.

Но почему не воспользоваться просто строкой? Вообще-то можно. Многие, даже большинство системных методов, ожидающих символ в качестве параметра, соглашаются и на строку.

```
attr_reader :alpha
attr_reader "beta"   # Так тоже можно
```

На самом деле, символ «похож» на строку в том смысле, что ему соответствует последовательность символов. Поэтому некоторые говорят, что «символ – это просто неизменяемая строка». Но класс Symbol не наследует классу String, а типичные операции над строками необязательно применимы к символам.

Также неправильно думать, что символы напрямую соответствуют идентификаторам. Из-за этого непонимания некоторые говорят о «таблице символов» (как если бы речь шла об ассемблированном объектном коде). В действительности это представление бессмысленно; хотя символы и хранятся в какой-то внутренней таблице (а как же иначе?), Ruby не дает к ней доступа, поэтому программистам все равно, существует она или нет.

Более того, символы даже не всегда выглядят как идентификаторы. Обычно это так, что бы под этим ни понимать, но символ может содержать и знаки препинания, если заключен в кавычки. Все показанные ниже символы допустимы:

```
sym1 = :"This is a symbol"
sym2 = :"This is, too!"
sym3 = :")(*&^%$"            # и даже такой
```

Можно даже использовать символы для определения переменных и методов экземпляра, но тогда для ссылки на них пришлось бы применять такие методы, как send и instance_variable_get. Вообще говоря, такая практика не рекомендуется.

6.1.1. Символы как перечисления

В языке Pascal и в поздних версиях C есть понятие перечисляемого типа. В Ruby ничего подобного быть не может, ведь никакого контроля типов не производится. Но символы часто используются как мнемонические имена; стороны света можно было бы представить как :north, :south, :east и :west. Быть может, немного понятнее хранить их в виде констант:

```
Directions = [:north, :south, :east, :west]
```

Если бы это были строки, а не символы, то определение их в виде констант могло бы сэкономить память, но каждый символ все равно существует в объектном пространстве в единственном экземпляре. (Символы, подобно объектам Fixnum, хранятся как непосредственные значения.)

6.1.2. Символы как метазначения

Мы нередко пользуемся исключениями, чтобы уйти от кодов возврата. Но никто не мешает возвращать коды ошибки, если вам так хочется. К тому же в Ruby метод может возвращать более одного значения.

В таком механизме часто возникает необходимость. Когда-то символ NUL кода ASCII вообще не считался символом. В языке C есть понятие нулевого указателя (NULL), в Pascal есть указатель nil, в SQL NULL означает отсутствие какого бы то ни было значения. В Ruby, конечно, тоже есть свой nil.

Проблема в том, что такие метазначения часто путают с действительными значениями. В наши дни все считают NUL настоящим символом кода ASCII. И в Ruby нельзя сказать, что nil не является объектом; его можно хранить, над ним можно выполнять какие-то операции. Поэтому не вполне понятно, как интерпретировать ситуацию, когда hash[key] возвращает nil: то ли указанный ключ вообще не найден, то ли с ним ассоциировано значение nil.

Идея в том, что иногда символы могут выступать в роли подходящих метазначений. Представьте метод, который получает строку из сети (возможно, по протоколу HTTP или иным способом). При желании можно было бы вернуть нестроковое значение, как индикатор исключительной ситуации.

```
str = get_string
case str
  when String
    # Нормальная обработка
  when :eof
    # Конец файла, закрытие сокета и т.п.
  when :error
    # Ошибка сети или ввода-вывода
```

```
  when :timeout
    # Ответ не получен вовремя
end
```

Можно ли сказать, что это «лучше», чем механизм исключений? Необязательно. Но такую методику стоит иметь в виду, особенно когда приходится обрабатывать «граничные случаи», которые не считаются ошибками.

6.1.3. Символы, переменные и методы

Наверное, чаще всего символы применяются для определения атрибутов класса:

```
class MyClass
  attr_reader :alpha, :beta
  attr_writer :gamma, :delta
  attr_accessor :epsilon
  # ...
end
```

Имейте в виду, что в этом фрагменте на самом деле исполняется некий код. Например, `attr_accessor` использует имя символа для определения имени переменной экземпляра, а также методов для ее чтения и изменения. Это *не* означает, что всегда имеется точное соответствие между символом и именем переменной экземпляра. Например, обращаясь к методу `instance_variable_set`, мы должны задать точное имя переменной, включая и знак @:

```
instance_variable_set(:@foo, "str")   # Правильно
instance_variable_set(:foo, "str")    # Ошибка
```

Короче говоря, символ, передаваемый методам из семейства `attr`, – всего лишь аргумент, а сами эти методы создают требуемые переменные и методы экземпляра, основываясь на значении символа. (В конец имени метода изменения добавляется знак равенства, а в начало имени переменной экземпляра – знак @.) Бывают также случаи, когда символ должен точно соответствовать идентификатору, на который ссылается.

В большинстве, если не во всех случаях, методы, ожидающие на входе символ, принимают также строку. Обратное не всегда верно.

6.1.4. Преобразование строки в символ и обратно

Строки и символы можно преобразовывать друг в друга с помощью методов `to_str` и `to_sym`:

```
a = "foobar"
b = :foobar
a == b.to_str    # true
b == a.to_sym    # true
```

Для метапрограммирования иногда бывает полезен такой метод:

```
class Symbol
  def +(other)
    (self.to_s + other.to_s).to_sym
  end
end
```

Он позволяет конкатенировать символы (или дописывать строку в конец символа). Вообще говоря, не рекомендуется изменять поведение классов из базовой библиотеки в коде, которым может воспользоваться кто-то другой, но в экспериментах ради понимания сути вещей эта техника бывает очень полезна.

Ниже приведен пример использования; мы принимаем на входе символ и пытаемся определить, представляет ли он какой-нибудь метод доступа (то есть существует ли метод чтения или установки атрибута с таким именем):

```
class Object
  def accessor?(sym)
    return (self.respond_to?(sym) and self.respond_to?(sym+"="))
  end
end
```

Упомяну также о более изощренном способе применения символов. Иногда при выполнении операции `map` нужно указать сложный блок. Однако во многих случаях мы просто вызываем некоторый метод для каждого элемента массива или набора:

```
list = words.map {|x| x.capitalize }
```

Не кажется ли вам, что для такой простой задачи слишком много знаков препинания? Давайте вместо этого определим метод `to_proc` в классе `Symbol`? Он будет приводить любой символ к типу объекта `proc`. По существу, `proc` – это метод, с которым можно обращаться, как с объектом: присваивать переменной или вызывать. В классе `Symbol` возвращенный объект `proc` будет просто вызывать метод с таким же именем, как у символа; иными словами, он пошлет сам символ в виде сообщения объекту. Интересующий нас метод можно определить следующим образом:

```
def to_proc
  proc {|obj, *args| obj.send(self, *args) }
end
```

Имея такой метод, мы можем следующим образом переписать первоначальный код:

```
list = words.map(&:capitalize)
```

Стоит потратить немного времени и разобраться, как это работает. Метод `map` обычно принимает только блок (никаких других параметров). Наличие амперсанда позволяет передать объект `proc` вместо явно указанного блока. Поскольку мы применяем амперсанд к объекту, не являющемуся `proc`, то интерпретатор пытается вызвать метод `to_proc` этого объекта. Получающийся в результате объект `proc` подставляется вместо явного блока, чтобы метод `map` вызывал его для каждого элемента массива. А зачем передавать `self` в виде сообщения элементу массива? Затем, что объект `proc` является замыканием и, следовательно, помнит контекст, в котором был создан. А в момент создания `self` был ссылкой на символ, для которого вызывался метод `to_proc`.

Далее мы рассмотрим диапазоны, которые несмотря на простоту оказываются на удивление полезны.

6.2. Диапазоны

Понятие диапазона интуитивно понятно, но и у него имеются некоторые неочевидные особенности и способы применения. Одним из самых простых является числовой диапазон:

```
digits = 0..9
scale1 = 0..10
scale2 = 0...10
```

Оператор .. *включает* конечную точку, а оператор ... *не включает*. (Если это для вас не очевидно, просто запомните.) Таким образом, диапазоны digits и scale2 из предыдущего примера одинаковы.

Но диапазоны могут состоять не только из целых чисел, да и вообще не из чисел. Началом и концом диапазона в Ruby может быть любой объект. Однако, как мы вскоре увидим, не все диапазоны осмыслены или полезны.

Основные операции над диапазоном – обход, преобразование в массив, а также выяснение, попадает ли некоторый объект в данный диапазон. Рассмотрим разнообразные варианты этих и других операций.

6.2.1. Открытые и замкнутые диапазоны

Диапазон называет замкнутым, если включает конечную точку, и открытым – в противном случае:

```
r1 = 3..6     # замкнутый
r2 = 3...6    # открытый
a1 = r1.to_a  # [3,4,5,6]
a2 = r2.to_a  # [3,4,5]
```

Нельзя сконструировать диапазон, который не включал бы начальную точку. Можно считать это ограничением языка.

6.2.2. Нахождение границ диапазона

Методы first и last возвращают соответственно левую и правую границу диапазона. У них есть синонимы begin и end (это еще и ключевые слова, но интерпретируются как вызов метода, если явно указан вызывающий объект).

```
r1 = 3..6
r2 = 3...6
r1a, r1b = r1.first, r1.last     # 3, 6
r1c, r1d = r1.begin, r1.end      # 3, 6
r2a, r2b = r1.begin, r1.end      # 3, 6
```

Метод exclude_end? сообщает, включена ли в диапазон конечная точка:

```
r1.exclude_end?   # false
r2.exclude_end?   # true
```

6.2.3. Обход диапазона

Обычно диапазон можно обойти. Для этого класс, которому принадлежат границы диапазона, должен предоставлять осмысленный метод succ (следующий).

```
(3..6).each {|x| puts x }  # печатаются четыре строки
                           # (скобки обязательны)
```

Пока все хорошо. Но будьте очень осторожны при работе со строковыми диапазонами! В классе String имеется метод succ, но он не слишком полезен. Пользоваться этой возможностью следует только при строго контролируемых условиях, поскольку метод succ определен не вполне корректно. (В определении используется «интуитивно очевидный», а не лексикографический порядок, поэтому существуют строки, для которых понятие «следующей» неожиданно или вовсе не имеет смысла.)

```
r1 = "7".."9"
r2 = "7".."10"
r1.each {|x| puts x }   # печатаются три строки
r2.each {|x| puts x }   # ничего не печатается!
```

Предыдущие примеры похожи, но ведут себя по-разному. Отчасти причина в том, что границы второго диапазона – строки разной длины. Мы ожидаем, что в диапазон входят строки "7", "8", "9" и "10", но что происходит на самом деле?

При обходе диапазона r2 мы начинаем со значения "7" и входим в цикл, который завершается, когда текущее значение окажется больше правой границы. Но ведь "7" и "10" – не числа, а строки, и сравниваются они как строки, то есть лексикографически. Поэтому левая граница оказывается больше правой, и цикл не выполняется ни разу.

А что можно сказать по поводу диапазонов чисел с плавающей точкой? Такой диапазон можно сконструировать и, конечно, можно проверить, попадает ли в него конкретное число. Это полезно. Но обойти такой диапазон нельзя, так как метод succ отсутствует.

```
fr = 2.0..2.2
fr.each {|x| puts x }   # ошибка!
```

Почему для чисел с плавающей точкой нет метода succ? Теоретически можно было бы увеличивать число на некоторое приращение. Но величина такого приращения сильно зависела бы от конкретной машины, при этом даже для обхода «небольшого» диапазона понадобилось бы гигантское число итераций, а полезность такой операции весьма сомнительна.

6.2.4. Проверка принадлежности диапазону

Зачем нужен диапазон, если нельзя проверить, принадлежит ли ему конкретный объект? Эта задача легко решается с помощью метода include?:

```
r1 = 23456..34567
x = 14142
y = 31416
r1.include?(x)      # false
r1.include?(y)      # true
```

У этого метода есть также синоним member?.

Но как он работает? Как интерпретатор определяет, принадлежит ли объект диапазону? Просто путем сравнения с границами (поэтому проверка принадлежности диапазону возможна лишь, если определен осмысленный оператор <=>).

Следовательно, запись (a..b).include?(x) эквивалентна x >= a and x <= b.

Еще раз предупреждаем: будьте осторожны со строковыми диапазонами:

```
s1 = "2".."5"
str = "28"
s1.include?(str)    # true (сбивает с толку!)
```

6.2.5. Преобразование в массив

Когда диапазон преобразуется в массив, интерпретатор последовательно вызывает метод succ, пока не будет достигнута правая граница, и помещает каждый элемент диапазона в возвращаемый массив:

```
r = 3..12
arr = r.to_a     # [3,4,5,6,7,8,9,10,11,12]
```

Ясно, что для диапазонов чисел типа Float такой подход не работает. Со строковыми диапазонами иногда будет работать, но лучше этого не делать, поскольку результат не всегда очевиден или осмыслен.

6.2.6. Обратные диапазоны

Имеет ли смысл говорить об *обратном диапазоне*? И да, и нет. Следующий диапазон допустим:

```
r = 6..3
x = r.begin              # 6
y = r.end                # 3
flag = r.end_excluded?   # false
```

Как видите, мы можем определить обе границы и узнать, что правая граница включена. Но этим перечень возможных операций практически исчерпывается.

```
arr = r.to_a       # []
r.each {|x| p x}   # ни одной итерации
y = 5
r.include?(y)      # false (для любого значения y)
```

Означает ли это, что обратные диапазоны всегда бесполезны? Вовсе нет. В некоторых случаях разумно инкапсулировать границы в один объект.

На самом деле, массивы и строки часто принимают обратные диапазоны в качестве индексов, поскольку индексация для них начинается с 0, если отсчитывать от левой границы, и с -1 – если от правой. Поэтому допустимы такие выражения:

```
string = "flowery"
str1   = string[0..-2]   # "flower"
str2   = string[1..-2]   # "lower"
str3   = string[-5..-3]  # "owe" (по существу, прямой диапазон)
```

6.2.7. Оператор переключения

Диапазон в составе условия обрабатывается особым образом. В этом случае .. называется *оператором переключения* (flip-flop operator), поскольку это, по существу, переключатель, который сохраняет свое состояние.

Такой прием, позаимствованный из языка Perl, бывает полезен. Но понять, как он работает, довольно трудно.

Представьте себе исходный текст программы на Ruby, в который встроена документация, ограниченная маркерами =begin и =end. Как бы вы подошли к задаче отыскания и вывода этих и только этих фрагментов? (Состояние переключается между «внутри раздела» и «вне раздела», отсюда и понятие переключения.) Решение, хотя интуитивно и не очевидное, дает следующий код:

```
file.each_line do |line|
      puts line if (line=~/=begin/)..(line=~/=end/)
end
```

Волшебство объясняется принципом работы оператора переключения.

Во-первых, надо осознать, что «диапазон» сохраняет свое состояние, хотя оно и скрыто. Когда становится истинным условие, заданное в качестве левой границы, сам диапазон принимает значение true. Он сохраняет это состояние до тех пор, пока не станет истинным условие на правой границе, и в этот момент состояние переключается в false.

Такое поведение полезно во многих случаях, в частности, для конфигурационных файлов, разбитых на разделы, выбора диапазонов элементов из списков и т.д.

Но лично мне такой синтаксис не нравится. Недовольны им и многие другие. Его исключение публично обсуждается по адресу bugs.ruby-lang.org/issues/5400, и сам Мац сказал, что рано или поздно он будет убран.

Что же не так с оператором переключения? Вот мое мнение на этот счет.

В контексте предыдущего примера рассмотрим строку, начинающуюся с маркера =begin. Напомним, что оператор =~ не возвращает true или false, как можно было бы ожидать; он возвращает начальную позицию найденного соответствия (Fixnum) или nil, если соответствие не найдено. Следовательно, при вычислении выражений для строк, попадающих и не попадающих в диапазон, мы получаем 0 и nil соответственно.

Однако при попытке сконструировать диапазон от 0 до nil возникает ошибка, так как такой диапазон не имеет смысла:

```
range = 0..nil    # ошибка!
```

Далее, напомню, что в Ruby только false и nil дают значение «ложь», все остальные объекты в логическом контексте вычисляются как «истина». А, значит, следуя общей идеологии диапазон не должен вычисляться как «ложь».

```
puts "hello" if x..y
# Печатается "hello" для любого допустимого диапазона x..y
```

Но предположим, что мы сохранили эти значения в переменных, а потом из них сконструировали диапазон. Все перестанет работать, так как проверка всегда дает true.

```
file.each_line do |line|
      start = line=~/=begin/
      stop = line=~/=end/
      puts line if start..stop
end
```

А что если сам диапазон поместить в переменную? Тоже не получится, проверка снова дает true.

```
file.each_line do |line|
  range = (line=~/=begin/)..(line=~/=end/)
  puts line if range
end
```

Чтобы понять, в чем тут дело, нужно осознать, что весь диапазон (включая обе границы) вычисляется на каждой итерации цикла, но с учетом внутреннего состояния. Поэтому оператор переключения – вообще не настоящий диапазон. Тот факт, что он выглядит похожим на диапазон, хотя по сути таковым не является, многие считают «злом».

И, наконец, задумаемся о границах в операторе переключения. Они вычисляются каждый раз, но результат вычисления нельзя сохранить в переменной и затем просто подставить ее. В некотором смысле, граничные точки оказываются похожи на объекты proc. Это не значения, а исполняемый код. Тот факт, что нечто, выглядящее как обычное выражение, на самом деле представляет собой proc, тоже не вызывает восторга.

И несмотря на все вышесказанное, функциональность-то полезная. Можно ли написать класс, который инкапсулирует ее, но при этом не будет таким «магическим»? Можно, и даже не очень трудно. В листинге 6.1 приведен простой класс Transition, имитирующий поведение оператора переключения.

Листинг 6.1. Класс `Transition`

```
class Transition
  A, B = :A, :B
  T, F = true, false

         # state,p1,p2  => newstate, result
  Table = {[A,F,F]=>[A,F], [B,F,F]=>[B,T],
           [A,T,F]=>[B,T], [B,T,F]=>[B,T],
           [A,F,T]=>[A,F], [B,F,T]=>[A,T],
           [A,T,T]=>[A,T], [B,T,T]=>[A,T]}

  def initialize(proc1, proc2)
    @state = A
    @proc1, @proc2 = proc1, proc2
  end

  def check?
    p1 = @proc1.call ? T : F
    p2 = @proc2.call ? T : F
```

```
    @state, result = *Table[[@state,p1,p2]]
    return result
  end
end
```

В классе `Transition` для управления переходами применяется простой конечный автомат. Он инициализируется парой объектов `proc` (теми же, что для оператора переключения). Мы утратили небольшое удобство: все переменные (например, `line`), которые используются внутри этих объектов, должны уже находиться в области видимости. Зато теперь у нас есть решение, свободное от «магии», и все выражения ведут себя так, как в любом другом контексте Ruby.

Вот слегка измененный вариант того же подхода. Здесь метод `initialize` принимает `proc` и два произвольных выражения:

```
def initialize(var,flag1,flag2)
      @state = A
      @flag1, @flag2 = flag1, flag2
end

def check?(item)
      p1 = (@flag1 === item) ? T : F
      p2 = (@flag2 === item) ? T : F
      @state, result = *Table[[@state, p1, p2]]
      return result
end
```

Оператор ветвящегося равенства проверяет соотношение между границами и переменной.

Вот как используется новая версия:

```
trans = Transition.new(/=begin/, /=end/)
file.each_line do |line|
      puts line if trans.check?(line)
end
```

Я рекомендую именно такой подход, поскольку в нем все делается открыто, не прибегая к волшебству.

6.2.8. Нестандартные диапазоны

Рассмотрим пример диапазона, состоящего из произвольных объектов. В листинге 6.2 приведен класс для работы с римскими числами.

Листинг 6.2. Класс для работы с римскими числами

```
class Roman
  include Comparable

  I,IV,V,IX,X,XL,L,XC,C,CD,D,CM,M =
    1, 4, 5, 9, 10, 40, 50, 90, 100, 400, 500, 900, 1000

  Values = %w[M CM D CD C XC L XL X IX V IV I]
```

```
def Roman.encode(value)
  return "" if self == 0
  str = ""
  Values.each do |letters|
    rnum = const_get(letters)
    if value >= rnum
      return(letters + str=encode(value-rnum))
    end
  end
  str
end

def Roman.decode(rvalue)
  sum = 0
  letters = rvalue.split('')
  letters.each_with_index do |letter,i|
    this = const_get(letter)
    that = const_get(letters[i+1]) rescue 0
    op = that > this ? :- : :+
    sum = sum.send(op,this)
  end
  sum
end

def initialize(value)
  case value
    when String
      @roman = value
      @decimal = Roman.decode(@roman)
    when Symbol
      @roman = value.to_s
      @decimal = Roman.decode(@roman)
    when Numeric
      @decimal = value
      @roman = Roman.encode(@decimal)
  end
end

def to_i
  @decimal
end

def to_s
  @roman
end

def succ
  Roman.new(@decimal-1)
```

```
  end

  def <=>(other)
    self.to_i <=> other.to_i
  end
end

def Roman(val)
  Roman.new(val)
end
```

Сначала несколько слов о самом классе. Его конструктору можно передать строку, символ (представляющий число, записанное римскими цифрами) или Fixnum (число, записанное обычными арабскими цифрами). Внутри выполняется преобразование и сохраняются обе формы. Имеется вспомогательный метод Roman, это просто сокращенная запись вызова Roman.new. Методы класса encode и decode занимаются преобразованием из арабской формы в римскую и наоборот.

Для простоты я опустил контроль данных. Кроме того, предполагается, что римские цифры представлены заглавными буквами.

Метод to_i, конечно же, возвращает десятичное значение, а to_s — число, записанное римскими цифрами. Метод succ возвращает следующее римское число, например, Roman(:IV).succ вернет Roman(:V).

Оператор сравнения сравнивает десятичные эквиваленты. Мы включили с помощью директивы include модуль Comparable, чтобы получить доступ к операторам «меньше» и «больше» (реализация которых опирается на наличие метода сравнения <=>).

Обратите внимание на использование символов в следующем фрагменте:

```
op = that > this ? :- : :+
sum = sum.send(op,this)
```

Здесь мы решаем, какую будем выполнять операцию (она обозначается символом): сложение или вычитание. Это не более чем краткий способ выразить следующую идею:

```
if that > this
  sum -= this
else
  sum += this
end
```

Второй вариант длиннее, зато более понятен.

Поскольку в этом классе есть метод succ и полный набор операторов сравнения, его можно использовать для конструирования диапазонов. Пример:

```
require 'roman'

y1 = Roman(:MCMLXVI)
y2 = Roman(:MMIX)
range = y1..y2                # 1966..2009
```

```
range.each {|x| puts x}        # выводится 44 строки

epoch = Roman(:MCMLXX)
range.include?(epoch)          # true

doomsday = Roman(2038)
range.include?(doomsday)       # false

Roman(:V) == Roman(:IV).succ   # true
Roman(:MCM) < Roman(:MM)       # true
```

6.3. Заключение

В этой главе мы познакомились с тем, что такое символы в Ruby и как они применяются. Мы продемонстрировали как стандартные, так и определенные пользователем способы употребления символов.

Также мы подробно остановились на диапазонах. Мы видели, как преобразовать диапазон в массив, как применить его в качестве индекса для массива или строки, как обойти диапазон и т.д. Мы рассмотрели оператор переключения (и альтернативу старому синтаксису). Наконец, мы создали класс, который корректно работает в сочетании с операторами диапазона.

На этом обсуждение символов и диапазонов заканчивается. Но, поскольку они используются в Ruby очень часто (и, несомненно, полезны), то мы еще не раз встретимся в ними в примерах кода.

Глава 7. Дата и время

Знает ли кто-нибудь,
что такое время на самом деле?

— Чикаго, Чикаго IV

Один из самых сложных и противоречивых аспектов человеческой жизни - измерение времени. Чтобы приблизиться к истинному пониманию предмета, необходимо хорошо знать физику, астрономию, историю, юриспруденцию, бизнес и религию. Астрономам известно (в отличие от большинства из нас!), что солнечное и звездное время - не совсем одно и то же. Ведомо им и то, почему иногда к году добавляется «високосная секунда». Историки знают, что в октябре 1582 года, когда Италия переходила с григорианского календарь на юлианский, из календаря было изъято несколько дней. Немногим известна разница между астрономической и церковной Пасхой (почти всегда они совпадают). Многие не в курсе, что код, который не делится на 400 (например, 1900), високосным не является.

Вычисления, в которых участвуют дата и время, выполняются компьютерами уже давно, но в большинстве языков программирования это довольно утомительное занятие. Это относится и к Ruby в силу самой природы данных. Но в Ruby было последовательно предпринято несколько шагов, чтобы упростить эти операции.

Для удобства читателя мы определим некоторые термины, которые, возможно, не всем известны. Они пришли как из естественного языка, так и из других языков программирования.

Среднее время по Гринвичу (Greenwich Mean Time – GMT) - устаревший термин, который теперь официально не употребляется. Новый глобальный стандарт называется *всеобщее скоординированное время* (Coordinated Universal Time или UTC от французской аббревиатуры). GMT и UTC - по существу, одно и то же. По прошествии ряда лет разница между ними составит несколько секунд. В большинстве промышленных программ (в Ruby в том числе) эти системы измерения времени не различаются.

На *летнее время* переходят раз в полгода, сдвигая официальное время на один час. Поэтому обозначения часовых поясов в США обычно заканчиваются на ST (Standard Time – стандартное время) или DT (Daylight Time – летнее время). Это происходит в большинстве (если не во всех) штатов США и во многих других странах.

Точка отсчета (epoch) – термин, пришедший из мира UNIX. В этой системе время обычно хранится как число секунд, прошедших с определенного момента (называемого точкой отсчета), а именно с полуночи 1 января 1970 года по Гринвичу. (Отметим, что во временных поясах США точка отсчета оказывается 31 декабря предыдущего года). Тем же словом обозначается не только начальный момент, но и время, прошедшее с этого момента.

Для выполнения большинства операций используется класс `Time`. Классы `Date` и `DateTime` обеспечивают дополнительную гибкость.

7.1. Определение текущего момента времени

Самый главный вопрос при манипуляциях с датами и временем: какой сегодня день и сколько сейчас времени? В Ruby при создании объекта класса `Time` без параметров, устанавливаются текущие дата и время.

```
t0 = Time.new
```

Синонимом служит `Time.now`:

```
t0 = Time.now
```

Отметим, что разрешающая способность системного таймера на разных машинах различна. Иногда это микросекунды, в этом случае два объекта `Time`, созданных подряд, могут фиксировать разное время.

7.2. Работа с конкретными датами (после точки отсчета)

Большинству программ нужно работать только с датами, относящимися к будущему или недавнему прошлому. Для таких целей класса `Time` достаточно. Наиболее интересны методы `mktime`, `local`, `gm` и `utc`.

Метод `mktime` создает новый объект `Time` на основе переданных параметров. Параметры задаются по убыванию длительности промежутка: год, месяц, день, часы, минуты, секунды, микросекунды. Все параметры, кроме года, необязательны; по умолчанию предполагается минимально возможное значение. В некоторых машинных архитектурах микросекунды игнорируются. Час должен быть числом от 0 до 23.

```
t1 = Time.mktime(2014)                  # 1 января 2014 в 0:00:00
t2 = Time.mktime(2014,3)
t3 = Time.mktime(2014,3,15)
t4 = Time.mktime(2014,3,15,21)
t5 = Time.mktime(2014,3,15,21,30)
t6 = Time.mktime(2014,3,15,21,30,15)  # 15 марта 2014 21:30:15
```

Отметим, что в методе `mktime` используется местное поясное время. Поэтому у него есть синоним `Time.local`.

```
t7 = Time.local(2014,3,15,21,30,15)   # 15 марта 2014 21:30:15
```

Метод Time.gm, по существу делает то же самое, но в нем предполагается время GMT (или UTC). Поскольку автор книги проживает в Тихоокеанском часовом поясе США, то разница составляет 8 часов:

```
t8 = Time.gm(2014,3,15,21,30,15)      # 15 марта 2014 21:30:15 pm
# Это 14:30:15 по Тихоокеанскому  времени!
```

У этого метода есть синоним Time.utc:

```
t9 = Time.utc(2001,3,15,21,30,15)     # 15 марта 2014 21:30:15 pm
# Снова 13:30:15 по Тихоокеанскому времени.
```

Отметим одну важную вещь. Все эти методы могут принимать и альтернативный набор параметров. Метод экземпляра to_a (который преобразует время в массив отдельных компонентов) возвращает набор значений в следующем порядке: секунды, минуты, часы, день, месяц, год, день недели (0..6), порядковый номер дня в году (1..366), летнее время (true или false), часовой пояс (строка).

Поэтому такие вызовы тоже допустимы:

```
t0 = Time.local(0,15,3,20,11,1979,2,324,false,"GMT-7:00")
t1 = Time.gm(*Time.now.to_a)
```

Однако, глядя на первый пример, не думайте, что сможете изменить вычисляемые параметры, например день недели (в данном случае 2 означает вторник). Такое действие противоречило бы принципам организации календаря, поэтому на созданном объекте Time оно никак не отражается. 20 ноября 1979 был вторник, и никакой код не сможет этого изменить.

И, наконец, отметим, что есть много способов задать время некорректно, например, указав тринадцатый месяц или 35-й день месяца. При любой подобной попытке возникнет исключение ArgumentError.

7.3. Определение дня недели

Есть несколько способов определить день недели. Во-первых, метод экземпляра to_a возвращает массив, содержащий всю информацию о моменте времени. Можно обратиться к его седьмому элементу; это число от 0 до 6, причем 0 соответствует воскресенью, а 6 – субботе.

```
time = Time.now
day = time.to_a[6]              # 2 (вторник)
```

Еще лучше воспользоваться методом экземпляра wday:

```
day = time.wday                 # 2 (вторник)
```

Но и тот, и другой способ не очень удобны. Иногда нужно получить день недели в виде числа, но чаще нас интересует его название в виде строки. Для этого можно обратиться к методу strftime. Его название знакомо программистам на C. Он распознает около двадцати спецификаторов, позволяя по-разному форматировать дату и время (см. раздел 7.21).

```
day = time.strftime("%A")       # "Tuesday"
```

Можно получить и сокращенное название

```
long  = time.strftime("%a")     # "Tue"
```

7.4. Определение даты Пасхи

Дату этого праздника всегда было вычислить сложно, так как она привязана к лунному календарю. Солнечный год не делится нацело на лунные месяцы, поэтому все, что основано на таком исчислении времени, будет из года в год меняться.

Представленный ниже алгоритм хорошо известен с давних времен. Мы видели его реализацию на языках BASIC, Pascal и C. А теперь перевели и на Ruby:

```
def easter(year)
  c = year/100
  n = year - 19*(year/19)
  k = (c-17)/25
  i = c - c/4 - (c-k)/3 - 19*n + 15
  i = i - 30*(i/30)
  i = i - (i/28)*(1 -(i/28)*(29/(n+1))*((21-n)/11))
  j = year + year/4 + i - 2 - c - c/4
  j = j - 7*(j/7)
  l = i - j
  month = 3 + (l-40)/44
  day = l + 28 - 31*(month/4)
  [month, day]
end

date = easter 2014              # Найти месяц и день для 2014 года
t = Time.local 2014, *date      # Передать параметры Time.local
puts t                          # 2014-04-20 00:00:00 -0700
```

Кто-то, прочитав этот раздел о Пасхе, непременно спросит: «Церковная или астрономическая?» Честно говоря, не знаю. Если вам удастся выяснить, сообщите всем нам.

Я бы с удовольствием объяснил вам этот алгоритм, только вот сам его не понимаю. Что-то надо принимать на веру, а в случае даты Пасхи это особенно уместно.

7.5. Вычисление n-ого дня недели в месяце

Иногда, зная год и месяц, хочется вычислить дату, скажем, третьего понедельника или второго вторника в этом месяце. Эту задачу решает код в листинге 7.1.

Чтобы найти *n*-ое вхождение данного дня недели, мы передаем n в качестве первого параметра. Второй параметр – номер дня недели (0 – воскресенье, 1 – понедельник и т.д.). Третий и четвертый параметры – месяц и год соответственно.

Листинг 7.1. Вычисление n-го дня недели

```
def nth_wday(n, wday, month, year)
  if (!n.between? 1,5) or
     (!wday.between? 0,6) or
     (!month.between? 1,12)
    raise ArgumentError
  end
  t = Time.local year, month, 1
```

```
  first = t.wday
  if first == wday
    fwd = 1
  elsif first < wday
    fwd = wday - first + 1
  elsif first > wday
    fwd = (wday+7) - first + 1
  end
  target = fwd + (n-1)*7
  begin
    t2 = Time.local year, month, target
  rescue ArgumentError
    return nil
  end
  if t2.mday == target
    t2
  else
    nil
  end
end
```

Странный код в конце текста метода призван скорректировать давнюю традицию, принятую в функциях работы с датами. Если вы думаете, что попытка создать объект для представления 31 ноября приведет к ошибке, то разочарую вас. Почти все системы молчаливо преобразуют эту дату в 1 декабря. Если вы давным-давно программируете в UNIX, то, наверное, полагаете, что так и должно быть. Другие сочтут это ошибкой.

Не станем спорить о том, что должна делать системная библиотека и должен ли Ruby изменить это поведение. Но мы не хотим, чтобы наша процедура продолжала эту традицию. Если вы ищете, к примеру, пятую пятницу в ноябре 2014 года, то она вернет `nil` (а не 5 декабря 2014 года).

7.6. Преобразование из секунд в более крупные единицы

Иногда нужно преобразовать заданное число секунд в дни, часы, минуты и секунды. Это можно сделать следующим образом:

```
def sec2dhms(secs)
  time = secs.round          # Отбрасываем микросекунды
  sec = time % 60            # Извлекаем секунды
  time /= 60                 # Отбрасываем секунды
  mins = time % 60           # Извлекаем минуты
  time /= 60                 # Отбрасываем минуты
  hrs = time % 24            # Извлекаем часы
  time /= 24                 # Отбрасываем часы
  days = time                # Дни (последний остаток)
  [days, hrs, mins, sec]     # Возвращаем массив [d,h,m,s]
end
```

```
t = sec2dhms(1000000)          # Миллион секунд равно...

puts "#{t[0]} days,"           # 11 дней,
puts "#{t[1]} hours,"          # 13 часов,
puts "#{t[2]} minutes,"        # 46 минут
puts " and #{t[3]} seconds." # и 40 секунд.
```

Можно было пойти и дальше. Но неделю вряд ли назовешь полезной единицей, месяц не слишком точно определен, а год не всегда содержит одно и то же число дней.

Ниже приведена также обратная функция:

```
def dhms2sec(days,hrs=0,min=0,sec=0)
  days*86400 + hrs*3600 + min*60 + sec
end
```

7.7. Вычисление промежутка времени, прошедшего от точки отсчета

По разным причинам может понадобиться перейти от внутреннего (традиционного) представления времени к стандартному. В системе время хранится как число секунд, прошедших с точки отсчета.

Метод класса `Time.at` создает новый объект `Time`, зная, сколько секунд прошло с точки отсчета:

```
epoch = Time.at(0)          # Найти точку отсчета (1 января 1970 GMT)
newmil = Time.at(978307200) # Счастливого миллениума! (1 января 2001)
```

Обратная функция — это метод экземпляра `to_i`, который преобразует дату в целое число.

```
now = Time.now              # 2014-07-23 17:24:26 -0700
sec = now.to_i              # 1406161466
```

Если нужны микросекунды, и система поддерживает такую точность, то можно воспользоваться методом `to_f` для преобразования в число с плавающей точкой.

7.8. Високосные секунды

Вот снова день исчез, как ветра легкий стон,
Из нашей жизни, друг, навеки выпал он.
Но я, покуда жив, тревожиться не стану
О дне, что отошел, и дне, что не рожден.

Омар Хайям «Рубаи» (перевод О. Румер)

Хотите иметь дело с «високосными» секундами? Мой совет: не делайте этого.

Високосные секунды — это не миф. Одна была добавлена в 2005 году, когда в последней минуте 30 июня была 61 секунда, а не 60. Библиотечные функции уже

много лет учитывают возможность появления минут, состоящих из 61 секунды, но наш опыт показывает, что большинство операционных систем високосные секунды игнорируют. Говоря «большинство», мы имеем в виду все, с которыми когда-либо сталкивались.

```
t0 = Time.gm(2012, 6, 30, 23, 59, 59)
puts t0 + 1 # 2012-07-01 00:00:00 UTC
```

Быть может (хотя и маловероятно), Ruby скорректирует эту ситуацию. Но во время работы над этой книгой таких планов не было.

7.9. Определение порядкового номера дня в году

Порядковый номер дня в году иногда еще называют юлианской датой, хотя это не имеет прямого отношения к юлианскому календарю, давно вышедшему из употребления. Многие считают, что такое название неправильно, поэтому мы им больше пользоваться не будем.

Но как бы ни называть, иногда хочется узнать порядковый номер дня, то есть получить число от 1 до 366. В Ruby это просто, достаточно вызвать метод yday:

```
t = Time.now
day = t.yday      # 315
```

7.10. Контроль даты и времени

В разделе 7.5 мы видели, что стандартные функции не проверяют корректность данных, а «переносят» ее вперед, если необходимо. Например, 31 ноября становится 1 декабря.

Иногда такое поведение даже желательно. А если нет, то спешу обрадовать: стандартная библиотека Date не считает такие даты правильными. Мы можем воспользоваться этим фактом для контроля переданной даты.

```
class Time

  def Time.validate(year, month=1, day=1,
                    hour=0, min=0, sec=0, usec=0)
    require "date"

    begin
      d = Date.new(year,month,day)
    rescue
      return nil
    end
    Time.local(year,month,day,hour,min,sec,usec)
  end

end

t1 = Time.validate(2014,11,30)  # Создается корректный объект
t2 = Time.validate(2014,11,31)  # Возвращается nil
```

Здесь мы не стали мудрствовать лукаво, а просто возвращаем nil, если переданные параметры не соответствуют правильной дате (полагаясь на вердикт, вынесенный классом Date). Мы оформили этот метод как метод класса Time по аналогии с другими методами создания объектов.

Отметим, что класс Date может работать и с датами, предшествующими точке отсчета, то есть дата 31 мая 1961 с точки зрения этого класса вполне допустима. Но при попытке передать такие значения классу Time возникнет исключение ArgumentError. Мы не пытаемся его перехватить, так как полагаем, что это лучше делать на том же уровне пользовательского кода, где обрабатывались бы исключения, скажем, от метода Time.local.

Раз уж зашла речь о Time.local, то отметим, что мы воспользовались именно этим методом. Захоти мы работать со временем по Гринвичу, нужно было бы вызывать метод gmt. Лучше реализовать оба варианта.

7.11. Определение недели в году

Что такое «порядковый номер недели», не вполне ясно. Разные компании, коалиции, правительственные учреждения и органы стандартизации по-разному определяют это понятие. Путаница, понятно, связана с тем, что год может начинаться с любого дня недели. Возможно, мы хотим учитывать неполные недели, а, возможно, и нет. Кроме того, неделя в зависимости от страны может начинаться в воскресенье или в понедельник.

В этом разделе мы предложим только три варианта. Первые два основаны на методе strftime класса Time. Спецификатор %U отсчитывает недели, начинающиеся с воскресенья, а спецификатор %W – начинающиеся с понедельника.

Третью возможность предоставляет класс Date. В нем имеется метод доступа cweek, который возвращает порядковый номер недели, следуя определению из стандарта ISO 8601 (согласно которому первой считается неделя, содержащая первый вторник года).

Если все это вам не подходит, можете придумать собственный алгоритм. Все три решения включены в один фрагмент кода:

```
require "date"

# Посмотрим, в какую неделю попадает 1 мая
# в 2005 и 2015 годах.

t1 = Time.local(2005,5,1)
d1 = Date.new(2005,5,1)

week1a = t1.strftime("%U").to_i   # 18
week1b = t1.strftime("%W").to_i   # 17
week1c = d1.cweek                 # 17

t2 = Time.local(2015,5,1)
d2 = Date.new(2015,5,1)

week2a = t2.strftime("%U").to_i   # 17
```

```
week2b = t2.strftime("%W").to_i   # 17
week2c = d2.cweek                 # 18
```

7.12. Проверка года на високосность

В классе Date есть два метода класса julian_leap? и gregorian_leap?; но только последний применим к относительно недавнему времени. Есть также метод leap? , но это просто синоним gregorian_leap?.

```
require "date"
flag1 = Date.julian_leap? 1700     # true
flag2 = Date.gregorian_leap? 1700  # false
flag3 = Date.leap? 1700            # false
```

Любой ребенок знает первое правило проверки на вискосность: год должен делиться на 4. Меньшее число людей знают второе правило: год не должен делиться на 100. И уж совсем немногие знают про исключение из второго правила: если год делится на 400, то он високосный. Другими словами, последний год тысячелетия является високосным, только если он делится на 400; так 1900 год не был високосным, а 2000 был. (Поправка необходима, потому что в году не ровно 365.25 дней, а приблизительно 365.2422.)

В классе Time нет аналогичного метода, но при желании его легко можно добавить.

```
class Time

  def Time.leap? year
    if year % 400 == 0
      true
    elsif year % 100 == 0
      false
    elsif year % 4 == 0
      true
    else
      false
  end

end
```

Я привел этот код только для того, чтобы прояснить алгоритм; конечно, гораздо проще вызвать метод Date.leap?. В моей реализации это метод класса по аналогии с классом Date, но можно было бы сделать его и методом экземпляра.

7.13. Определение часового пояса

Метод zone класса Time возвращает название часового пояса в виде строки:

```
z1 = Time.gm(2000,11,10,22,5,0).zone     # "UTC"
z2 = Time.local(2000,11,10,22,5,0).zone  # "FST"
```

К сожалению, время хранится относительно текущего часового пояса, а не того, для которого был создан объект. При желании можно скорректировать его самостоятельно.

7.14. Манипулирование временем без даты

Иногда нужно работать с временем дня в виде строки. На помощь снова приходит метод strftime. Можно напечатать время, разбитое на часы, минуты и секунды:

```
t = Time.now
puts t.strftime("%H:%M:%S")    # Печатается 22:07:45
```

А можно только часы и минуты (прибавив 30 секунд, мы даже можем округлить до ближайшей минуты):

```
puts t.strftime("%H:%M")       # Печатается 22:07
puts (t+30).strftime("%H:%M")  # Печатается 22:08
```

Наконец, вместо стандартного представления с 24 часами, можно переключиться на 12-часовой формат, добавив признак перехода через полдень (AM/PM):

```
puts t.strftime("%I:%M %p")    # Печатается 10:07 PM
```

Есть и другие возможности, пробуйте.

7.15. Сравнение моментов времени

К классу Time подмешан модуль Comparable, поэтому моменты времени можно сравнивать непосредственно:

```
t0 = Time.local(2014,11,10,22,15)      # 10 Nov 2014 22:15
t1 = Time.local(2014,11,9,23,45)       #  9 Nov 2014 23:45
t2 = Time.local(2014,11,12,8,10)       # 12 Nov 2014  8:10
t3 = Time.local(2014,11,11,10,25)      # 11 Nov 2014 10:25

if t0 < t1 then puts "t0 < t1" end
if t1 != t2 then puts "t1 != t2" end
if t1 <= t2 then puts "t1 <= t2" end
if t3.between?(t1,t2)
  puts "t3 находится между t1 и t2"
end

# Все четыре предложения if возвращают true
```

7.16. Прибавление интервала к моменту времени

Можно получить новый момент времени, прибавив к существующему интервал. Интервал представляется целым числом, которое интерпретируется как число секунд.

```
t0 = Time.now
t1 = t0 + 60          # Ровно одна минута с момента t0
```

```
t2 = t0 + 3600        # Ровно один час с момента t0
t3 = t0 + 86400       # Ровно один день с момента t0
```

Может пригодиться функция dhms2sec (определена в разделе 7.6 выше). Напомним, что по умолчанию параметры, соответствующие часам, минутам и секундам, равны 0.

```
t4 = t0 + dhms2sec(5,10)        # 5 дней, 10 часов в будущем
t5 = t0 + dhms2sec(22,18,15)    # 22 дня, 18 часов, 15 минут в будущем
t6 = t0 - dhms2sec(7)           # Ровно неделю назад
```

Не забывайте, что для получения момента времени в прошлом нужно вычитать, как при вычислении t6 в примере выше.

7.17. Вычисление разности между двумя моментами времени

Можно вычислить интервал между двумя моментами времени. В результате вычитания одного объекта Time из другого получаем число секунд:

```
today = Time.local(2014,11,10)
yesterday = Time.local(2014,11,9)
diff = today - yesterday            # 86400 секунд
```

И снова оказывается полезной функция sec2dhms (определена в разделе 7.6 выше).

```
past = Time.local(2012,9,13,4,15)
now = Time.local(2014,11,10,22,42)
diff = now - past                   # 68153220.0
unit = sec2dhms(diff)
puts "#{unit[0]} дней,"             # 788 дней,
puts "#{unit[1]} часов,"            # 19 часов,
puts "#{unit[2]} минут"             # 27 минут
puts "и #{unit[3]} секунд."         # и 0 секунд.
```

7.18. Работа с конкретными датами (до точки отсчета)

В стандартной библиотеке Date есть одноименный класс для работы с датами, предшествующими полуночи 1 января 1970 года.

Несмотря на некоторое перекрытие с классом Time, между ними есть существенные различия. Самое главное состоит в том, что класс Date вообще игнорирует время, то есть работает с точностью до одного дня. Кроме того, класс Date строже контролирует ошибки, чем класс Time; попытка обратиться к 31 июня (или к 29 февраля невисокосного года) приведет к исключению. Код даже «знает» о различных датах перехода на григорианский календарь в Италии и Англии (в 1582 и 1752 году соответственно) и может обнаружить «несуществующие» даты, появившиеся в результате такого перехода. Эта стандартная библиотека – паноптикум интересного и местами загадочного кода. К сожалению, у нас нет места для более подробного разговора о ней.

7.19. Взаимные преобразования объектов Date, Time и DateTime

В Ruby есть три основных класса для работы с датами и временем: `Time`, `Date` и `DateTime`. Опишем их особенности:

- Класс `Time` – преимущественно обертывает соответствующие функции из стандартной библиотеки языка C. Они как правило опираются на точку отсчета в UNIX и потому не способны представлять моменты времени раньше 1970 года.
- Класс `Date` создан для преодоления недостатков класса `Time`. Он без труда справляется с датами в более отдаленном прошлом, например, позволяет представить день рождения Леонардо да Винчи (15 апреля 1452), и знает о реформе календаря. Но у него есть свои слабые места: он работает только с датами, игнорируя время.
- Класс `DateTime` наследует `Date` и пытается компенсировать отсутствующие в нем возможности. Он может представлять даты не хуже `Date` и время не хуже `Time`. Часто его способ представления даты и времени оказывается «наилучшим».

Однако не думайте, что объект `DateTime` – это просто объект `Date`, к которому механически присоединен объект `Time`. На самом деле, в классе `DateTime` отсутствуют такие методы как `usec`, `dst?` и некоторые другие.

Классы `Date` и `DateTime` входят в стандартную библиотеку; прежде чем использовать, их необходимо загрузить командой `require "date"`. После того как библиотека `Date` загружена, становятся доступны вполне предсказуемые преобразования, осуществляемые методами `to_time`, `to_date` и `to_datetime`.

7.20. Извлечение даты и времени из строки

Дата и время могут быть представлены в виде строки самыми разными способами: в полной или сокращенной форме, с разной пунктуацией, различным порядком компонентов и т.д. Из-за такого разнообразия очень сложно написать код, интерпретирующий символьную строку как дату. Рассмотрим несколько примеров:

```
s1 = "9/13/98 2:15am"
s2 = "1961-05-31"
s3 = "11 July 1924"
s4 = "April 17, 1929"
s5 = "20 July 1969 16:17 EDT"  # время высадки человека на Луну
s6 = "Mon Nov 13 2000"
s7 = "August 24, 79"           # День гибели Помпеи
s8 = "8/24/79"
```

Библиотеки `Time` и `Date` предоставляют минимальные средства разбора посредством метода класса `parse`. К сожалению, в этом методе предполагается, что двузначный год относится к прошлому столетию, поэтому дата гибели Помпеи сме-

щена на 1900 лет. Также неправильно интерпретируются даты, записанные, как принято в Америке, например, s1 и s8 в примерах выше.

Для решения проблемы со столетием задайте второй аргумент метода Date.parse. Если он равен false, то относительно века уже не делается никаких предположений:

```
Date.parse("August 24, 79", false) # 0079-08-24
```

Имейте в виду, что по умолчанию метод пытается «угадать» год, иногда это соответствует вашим намерениям, а иногда нет. Двузначные числа до 69 включительно интерпретируются как год 20XX, а, начиная с 70, – как 19XX.

Для разбора «американских» дат и многих еще более эксцентричных форматов пользуйтесь gem-пакетом Chronic (устанавливается командой gem install chronic). Он разделяет с Time предубеждения относительно двузначных номеров года, но в остальном разбирает даты весьма квалифицированно.

```
Chronic.parse "next tuesday"
Chronic.parse "3 weeks ago monday at 5pm"
Chronic.parse "time to go home"   # Что ж, не всякое время допустимо
```

Если Chronic не может разобрать строку как момент времени, он просто возвращает nil, не возбуждая исключение.

При разборе даты и времени внимательно относитесь к часовым поясам. Например, аббревиатура CST в разных странах имеет по меньшей мере пять разных интерпретаций. Подробнее о поясном времени см. раздел 7.22.

7.21. Форматирование и печать даты и времени

Для получения канонического представления даты и времени служит метод to_s. Тот же самый результат мы получили бы, просто передав временное значение методу puts.

Традиционный для UNIX формат, содержащий день недели, можно получить от метода asctime («ASCII time»); у него есть также синоним ctime – для тех, кому это имя уже знакомо.

С помощью метода strftime класса Time можно отформатировать дату и время почти произвольным образом. В этой главе мы уже встречали спецификаторы %a, %A, %U, %W, %H, %M, %S, %I и %p, а ниже приведены оставшиеся:

```
%b Сокращенное название месяца ("Jan")
%B Полное название месяца ("January")
%c Предпочтительное представление локальной даты и времени
%d День месяца (1..31)
%j Порядковый номер дня в году (1..366); так называемая "юлианская дата"
%m Номер месяца (1..12)
%w Номер дня недели (0..6)
%x Предпочтительное представление даты без времени
%y Двузначный год (без указания века)
%Y Четырехзначный год
%Z Название часового пояса
%% Знак "%"
```

Дополнительную информацию см. в справочном руководстве по языку Ruby.

7.22. Преобразование часовых поясов

Обычно приходится работать только с двумя часовыми поясами: GMT (или UTC) и тем, в котором вы находитесь.

Метод `utc` преобразует время в UTC (модифицируя сам вызывающий объект). У него есть синоним `gmtime`.

Может быть, вы думаете, что можно просто преобразовать момент времени в массив, подменить часовой пояс и выполнить обратное преобразование? Проблема в том, что все методы класса, к примеру `local` и `gm` (а также их синонимы `mktime` и `utc`), готовы создавать объект `Time` только в предположении, что указано либо местное время, либо время UTC.

Есть обходной путь для преобразования часового пояса. Но предполагается, что вы заранее знаете разницу во времени. Взгляните на следующий фрагмент:

```
mississippi = Time.local(2014,11,13,9,35) # 9:35 am CST
california  = mississippi - 2*3600        # Минус два часа

time1 = mississippi.strftime("%X CST")    # 09:35:00 CST
time2 = california.strftime("%X PST")     # 07:35:00 PST
```

Спецификатор `%X` в методе `strftime` просто выводит время в формате `hh:mm:ss`.

Но такое преобразование не всегда годится, потому что некоторые аббревиатуры в разных странах означают разные часовые пояса.

Если вам приходится часто преобразовывать время из одного часового пояса в другой или просто необходимо поддерживать время в любом часовом поясе, то можно воспользоваться расширением класса `Time`, предоставляемым gem-пакетом `ActiveSupport`.

```
require 'active_support/time'
Time.zone = -8
Time.zone.name     # "Тихоокеанское время (США и Канада)"
Time.zone.now      # Wed, 25 Jun 2014 12:20:35 PDT -07:00
Time.zone.now.in_time_zone("Hawaii") # 09:20:36 HST -10:00
```

Дополнительные сведения о том, как использовать `ActiveSupport` для поддержки любого имеющегося в мире часового пояса, см. в документации по пакету, особенно по классу `TimeWithZone`.

7.23. Определение числа дней в месяце

Хотя встроенной функции для этой цели нет, ее предоставляет метод `days_in_month`, добавленный в класс `Time` gem-пакетом `ActiveSupport`. Но и без `ActiveSupport` его было бы нетрудно написать самостоятельно:

```
require 'date'
def month_days(month,year=Date.today.year)
  mdays = [nil,31,28,31,30,31,30,31,31,30,31,30,31]
```

```
  mdays[2] = 29 if Date.leap?(year)
  mdays[month]
end

days = month_days(5)              # 31 (May)
days = month_days(2,2000)         # 29 (February 2000)
days = month_days(2,2100)         # 28 (February 2000)
```

7.24. Разбиение месяца на недели

Представьте, что нужно разбить месяц на недели, например, чтобы напечатать календарь. Эту задачу решает приведенный ниже код. Возвращаемый массив состоит из подмассивов, по семь элементов в каждом. При этом первому элементу каждого внутреннего массива соответствует воскресенье. Начальные элементы для первой недели и конечные для второй могут быть равны nil.

```
def calendar(month,year)
  days = month_days(month,year)
  t = Time.mktime(year,month,1)
  first = t.wday
  list = *1..days
  weeks = [[]]
  week1 = 7 - first
  week1.times { weeks[0] << list.shift }
  nweeks = list.size/7 + 1
  nweeks.times do |i|
    weeks[i+1] ||= []
    7.times do
      break if list.empty?
      weeks[i+1] << list.shift
    end
  end
  pad_first = 7-weeks[0].size
  pad_first.times { weeks[0].unshift(nil) }
  pad_last = 7-weeks[0].size
  pad_last.times { weeks[-1].unshift(nil) }
  weeks
end

arr = calendar(12,2008)  # [[nil, 1, 2, 3, 4, 5, 6],
                         #  [7, 8, 9, 10, 11, 12, 13],
                         #  [14, 15, 16, 17, 18, 19, 20],
                         #  [21, 22, 23, 24, 25, 26, 27],
                         #  [28, 29, 30, 31, nil, nil, nil]]
```

Чтобы было понятнее, распечатаем этот массив массивов:

```
def print_calendar(month,year)
  weeks = calendar(month,year)
  weeks.each |wk|
```

```
    wk.each do |d|
      item = d.nil? ? " "*4 : " %2d " % d
      print item
    end
    puts
  end
  puts
end

# Выводится:
#       1   2   3   4   5   6
#   7   8   9  10  11  12  13
#  14  15  16  17  18  19  20
#  21  22  23  24  25  26  27
#  28  29  30  31
```

7.25. Заключение

В этой главе мы рассмотрели класс `Time`, который является оберткой для функций из стандартной библиотеки языка C. Мы видели его возможности и ограничения.

Мы также узнали, зачем существуют классы `Date` и `DateTime` и какую функциональность они предоставляют. Мы научились выполнять преобразования между этими классами и добавили несколько собственных полезных методов.

На этом обсуждение даты и времени завершается. Переходим к массивам, хэшам и другим перечисляемым структурам в Ruby.

Глава 8. Массивы, хэши и другие перечисляемые структуры

Все детали должны соединяться без усилий. Помните, что механизм, который вы пытаетесь собрать, вами же был и разобран. Если не удается соединить детали, на то должна быть причина. Ни в коем случае не пользуйтесь молотком.

— Руководство по техническому обслуживанию компании IBM (1925)

Простых переменных для практического программирования недостаточно. В любом современном языке поддерживаются более сложные виды структурированных данных и предоставляются механизмы для создания новых абстрактных типов данных.

Исторически самой первой и широко распространившейся составной структурой данных был массив. Давным-давно, еще в языке FORTRAN, массивы назывались *индексированными переменными*; сегодня они несколько видоизменились, но основная идея во всех языках одна и та же.

Относительно недавно очень популярной структурой стали хэши. Как и массив, хэш представляет собой индексированный набор данных. Но в отличие от массива, в качестве индекса может выступать любой объект. (В Ruby, как и в большинстве других языков, элементы массива индексируются числами.)

Наконец, мы рассмотрим сам модуль `Enumerable` и разберемся, как он работает. И массивы, и хэши подмешивают этот модуль. То же самое может сделать и любой другой класс, которому необходима аналогичная функциональность.

Но не будем забегать вперед. Начнем с массивов.

8.1. Массивы

В Ruby массивы индексируются целыми числами, индексация начинается с нуля, как в языке C. На этом, впрочем, сходство и заканчивается.

Массивы в Ruby динамические. Можно (хотя это и не обязательно) задать размер массива при создании. Но после создания он может расти без вмешательства со стороны программиста.

Массивы в Ruby *неоднородны*, то есть в них могут храниться данные разных типов. На самом деле, в массиве хранятся только ссылки на объекты, а не сами объекты. Исключение составляют только непосредственные значения, например, объекта класса `Fixnum`.

Вместе с массивом хранится и его длина, поэтому нам не нужно тратить время на ее вычисление или сохранение во внешней переменной, обновляемой синхронно с массивом. К тому же, итераторы определены таким образом, что на практике нам вообще редко приходится задумываться о длине массива.

Наконец, класс `Array` в Ruby предоставляет немало полезных функций для работы с массивами: доступ, поиск, конкатенирование и т.п. В этом разделе мы изучим встроенную функциональность и расширим ее.

8.1.1. Создание и инициализация массива

Для создания массива применяется специальный метод класса `[ ]`; перечисленные внутри скобок данные помещаются во вновь созданный массив. Ниже показаны три способа вызвать этот метод (массивы `a`, `b` и `c` будут инициализированы одинаково).

```
a = Array.[](1,2,3,4)
b = Array[1,2,3,4]
c = [1,2,3,4]
```

Имеется также метод класса `new`, который принимает 0, 1 или 2 параметра. Первый параметр задает начальный размер массива (число элементов в нем). Второй определяет начальное значение каждого элемента:

```
d = Array.new                  # Создать пустой массив
e = Array.new(3)               # [nil, nil, nil]
f = Array.new(3, "blah")       # ["blah", "blah", "blah"]
```

Обратите особое внимание на последний пример. Типичная «ошибка начинающего» – думать, что все объекты в этом массиве различны. На самом деле, это три ссылки на один и тот же объект. Поэтому, если вы измените этот объект (а не замените его другим), то изменятся все элементы массива. Чтобы не попасть в эту ловушку, воспользуйтесь блоком. Блок будет вычисляться по одному разу для каждого элемента, поэтому все элементы окажутся различными объектами:

```
f[0].upcase!                   # теперь f содержит: ["BLAH", "BLAH", "BLAH"]
g = Array.new(3) { "blah" }    # ["blah", "blah", "blah"]
g[0].upcase!                   # теперь g содержит: ["BLAH", "blah", "blah"]
```

Наконец, метод `Array()` принимает один аргумент и при необходимости обертывает его массивом. Это часто бывает полезно, если переменная может быть как массивом, так и одиночным элементом, потому что одиночное значение преобразуется в одноэлементный массив:

```
h = Array(1)        # [1] не-массив обертывается массивом
i = Array([1])      # [1] массив остается массивом
j = Array(nil)      # [] nil игнорируется
j = Array([nil])    # массив [nil], содержащий элемент nil, не изменяется
```

8.1.2. Доступ к элементам массива и присваивание им значений

Получить ссылку на элемент и присвоить ему значение можно с помощью методов класса [ ] и [ ]= соответственно. Каждый из них принимает один целочисленный параметр, либо пару целых чисел (начало и конец), либо диапазон. Отрицательные индексы отсчитываются от конца массива, начиная с –1.

Специальный метод экземпляра at реализует простейший случай получения ссылки на элемент. Поскольку он может принимать только один целочисленный параметр, то работает чуть быстрее.

```
a = [1, 2, 3, 4, 5, 6]
b = a[0]                # 1
c = a.at(0)             # 1
d = a[-2]               # 5
e = a.at(-2)            # 5
f = a[9]                # nil
g = a.at(9)             # nil
h = a[3,3]              # [4, 5, 6]
i = a[2..4]             # [3, 4, 5]
j = a[2...4]            # [3, 4]

a[1] = 8                # [1, 8, 3, 4, 5, 6]
a[1,3] = [10, 20, 30]   # [1, 10, 20, 30, 5, 6]
a[0..3] = [2, 4, 6, 8]  # [2, 4, 6, 8, 5, 6]
a[-1] = 12              # [2, 4, 6, 8, 5, 12]
```

В следующем примере ссылка на элемент, расположенный за концом массива, приводит к росту массива. Отметим, что подмассив можно заменить другим массивом, содержащим больше элементов, чем было. В этом случае массив также автоматически вырастет.

```
k = [2, 4, 6, 8, 10]
k[1..2] = [3, 3, 3]     # [2, 3, 3, 3, 8, 10]
k[7] = 99               # [2, 3, 3, 3, 8, 10, nil, 99]
```

Наконец, если одному элементу присвоить в качестве значения массив, то на место этого элемент будет вставлен вложенный массив (в отличие от присваивания диапазону):

```
m = [1, 3, 5, 7, 9]
m[2] = [20, 30]          # [1, 3, [20, 30], 7, 9]

# С другой стороны...
m = [1, 3, 5, 7, 9]
m[2..2] = [20, 30]       # [1, 3, 20, 30, 7, 9]
```

Метод slice – синоним метода [ ]:

```
x = [0, 2, 4, 6, 8, 10, 12]
a = x.slice(2)           # 4
b = x.slice(2,4)         # [4, 6, 8, 10]
c = x.slice(2..4)        # [4, 6, 8]
```

Специальные методы first и last возвращают первый и последний элемент массива соответственно. Если массив пуст, они возвращают nil:

```
x = %w[alpha beta gamma delta epsilon]
a = x.first                # "alpha"
b = x.last                 # "epsilon"
```

Мы уже видели ранее, что иногда ссылка на элементы может возвращать целый подмассив. Но существуют и другие способы обратиться к нескольким элементам.

Метод values_at принимает список индексов и возвращает массив, содержащий только указанные элементы. Его можно использовать в тех случаях, когда диапазон не годится (так как нужные элементы находятся не в соседних позициях).

```
x = [10, 20, 30, 40, 50, 60]
y = x.values_at(0, 1, 4)          # [10, 20, 50]
z = x.values_at(0..2,5)           # [10, 20, 30, 60]
```

8.1.3. Определение размера массива

Метод length и его синоним size возвращают число элементов в массиве. (Как всегда, эта величина на единицу больше индекса последнего элемента.)

```
x = ["a", "b", "c", "d"]
a = x.length               # 4
b = x.size                 # 4
```

8.1.4. Сравнение массивов

При сравнении массивов возможны неожиданности, будьте осторожны.

Для сравнения массивов служит метод экземпляра <=>. Он работает так же, как в других контекстах, то есть возвращает -1 (меньше), 0 (равно) или 1 (больше). Методы == и != опираются на реализацию метода <=>.

Массивы сравниваются поэлементно; первая же пара несовпадающих элементов определяет результат всего сравнения. (Предпочтение отдается левее расположенным элементам, как при сравнении двух длинных целых чисел «на глазок», когда мы сравниваем по одной цифре за раз.)

```
a = [1, 2, 3, 9, 9]
b = [1, 2, 4, 1, 1]
c = a <=> b            # -1 (то есть a < b)
```

Если все элементы равны, то массивы считаются равными. Если один массив длиннее другого и все элементы вплоть до длины более короткого массива равны, то более длинный массив считается большим.

```
d = [1, 2, 3]
e = [1, 2, 3, 4]
f = [1, 2, 3]
if d < e                    # false
  puts "d меньше e"
end
if d == f
  puts "d равно f"          # Печатается "d равно f"
end
```

Поскольку класс Array не подмешивает модуль Comparable, то обычные операторы сравнения <, >, <= и >= для массивов не определены. Но при желании их легко определить самостоятельно:

```
class Array

  def <(other)
    (self <=> other) == -1
  end

  def <=(other)
    (self < other) or (self == other)
  end

  def >(other)
    (self <=> other) == 1
  end

  def >=(other)
    (self > other) or (self == other)
  end

end
```

Впрочем, было бы проще включить модуль Comparable:

```
class Array
  include Comparable
end
```

Определив эти операторы, можно пользоваться ими как обычно:

```
if a < b
  print "a < b"        # Печатается "a < b"
else
  print "a >= b"
end
if d < e
  puts "d < e"        # Печатается "d < e"
end
```

Может статься, что при сравнении массивов мы столкнемся с необходимостью сравнивать два элемента, для которых оператор <=> не определен или не имеет смысла. Следующий код приводит к возбуждению исключения ArgumentError, так как сравнение 3 <=> "x" лишено смысла:

```
g = [1, 2, 3]
h = [1, 2, "x"]
if g < h              # Ошибка!
  puts "g < h"        # Ничего не выводится
end
```

Если и это вас не смущает, то добавим, что сравнение на равенство и неравенство в этом случае работает. Объясняется это тем, что объекты разных типов считаются неравными, хотя мы и не можем сказать, какой из них больше.

```
if g != h              # Здесь ошибка не возникает.
  puts "g != h"        # Печатается "g != h"
end
```

Наконец, не исключено, что два массива, содержащих несравнимые типы данных, все равно можно сравнить с помощью операторов < и >. В примере ниже мы получаем определенный результат еще до того, как натолкнемся на несравнимые элементы:

```
i = [1, 2, 3]
j = [1, 2, 3, "x"]
if i < j               # Здесь ошибка не возникает.
  puts "i < j"         # Печатается "i < j"
end
```

8.1.5. Сортировка массива

Самый простой способ отсортировать массив – воспользоваться встроенным методом sort:

```
words = %w(the quick brown fox)
list = words.sort  # ["brown", "fox", "quick", "the"]
# Или отсортировать на месте:
words.sort!        # ["brown", "fox", "quick", "the"]
```

Здесь предполагается, что все элементы массивы сравнимы между собой. При сортировке неоднородного массива, например [1, 2, "three", 4], обычно возникает ошибка.

В подобных случаях можно воспользоваться также блочной формой того же метода. Ниже предполагается, что у каждого элемента есть хотя бы метод to_s (преобразующий его в строку):

```
a = [1, 2, "three", "four", 5, 6]
b = a.sort {|x,y| x.to_s <=> y.to_s}
# b равно [1, 2, 5, 6, "four", "three"]
```

Конечно, такое упорядочение (в данном случае основанное на кодировке ASCII) может оказаться бессмысленным. При работе с неоднородным массивом нужно прежде всего задать себе вопрос, зачем вообще его сортировать? И почему приходится хранить в массиве объекты разных типов?

Описанная методика работает, потому что блок возвращает целое число (–1, 0 или 1) при каждом вызове. Если возвращена -1, то есть x меньше y, то два элемента меняются местами. Чтобы отсортировать массив в порядке убывания, достаточно просто изменить порядок сравнения:

```
x = [1, 4, 3, 5, 2]
y = x.sort {|a,b| b <=> a}     # [5, 4, 3, 2, 1]
```

Блоки можно применять и для более сложных сортировок. Предположим, что нужно отсортировать список названий книг и фильмов следующим способом: регистр игнорируется, полностью игнорируются пробелы, а также некоторые знаки препинания. Ниже приведен простой пример (и преподаватели английского языка, и программисты будут удивлены таким способом упорядочения по алфавиту).

```
titles = ["Starship Troopers",
          "A Star is Born",
          "Star Wars",
          "Star 69",
          "The Starr Report"]
sorted = titles.sort do |x,y|
  # Удалить артикли
  a = x.sub(/^(a |an |the )/i, "")
  b = y.sub(/^(a |an |the )/i, "")
  # Удалить пробелы и знаки препинания
  a.delete!(" .,-?!")
  b.delete!(" .,-?!")
  # Преобразовать в верхний регистр
  a.upcase!
  b.upcase!
  # Сравнить a и b
  a <=> b
end

# Теперь sorted равно:
# [ "Star 69", "A Star is Born", "The Starr Report"
#   "Starship Troopers", "Star Wars"]
```

Данный пример не слишком полезен и, конечно, его можно было бы записать более компактно. Но идея в том, что для сравнения двух операндов в определенном порядке над ними можно выполнять произвольно сложный набор операций. (Отметим, однако, что мы не изменили исходные операнды, так как работали с их копиями.) Эта общая техника полезна во многих ситуациях, например, для сортировки по нескольким ключам или по ключам, вычисляемым во время выполнения.

В модуле `Enumerable` имеется также метод `sort_by` (который, конечно, подмешивается к классу `Array`). Важно понимать, что он делает.

В методе `sort_by` применяется то, что программисты на Perl называют преобразованием Шварца (в честь Рэндала Шварца (Randal Schwartz)). Вместо того чтобы сортировать сами элементы массива, мы применяем к ним некоторую функцию и сортируем возвращаемые ей результаты.

В качестве искусственного примера рассмотрим список файлов, который необходимо отсортировать по размеру. Прямолинейный способ выглядит так:

```
files = files.sort {|x,y| File.size(x) <=> File.size(y) }
```

Однако тут есть две проблемы. Во-первых, слишком многословно. Надо бы сделать покомпактнее.

Во-вторых, при такой сортировке приходится многократно обращаться к диску, а это довольно дорогая операция (по сравнению с операциями в оперативной памяти). Хуже того, одна и та же операция может выполняться несколько раз.

Метод sort_by решает обе проблемы. Вот «правильный» способ:

```
files = files.sort_by {|x| File.size(x) }
```

Здесь каждый ключ вычисляется ровно один раз, а затем сохраняется в виде пары ключ-данные. Для небольших массивов производительность при таком подходе может даже *снизиться*, зато код получается более понятным.

Не существует метода sort_by!. Но при желании вы можете написать его самостоятельно.

А как обстоит дело с сортировкой по нескольким ключам? Предположим, что имеется массив объектов, который нужно отсортировать по трем атрибутам: имени, возрасту и росту. Из того, что массивы можно сравнивать, следует, что такое решение будет работать:

```
list = list.sort_by {|x| [x.name, x.age, x.height] }
```

Конечно, элементы массива могут быть и не такими простыми. Допустимы произвольно сложные выражения.

8.1.6. Выборка из массива по заданному критерию

Иногда нужно найти в массиве один или несколько элементов так, как будто мы опрашиваем таблицу в базе данных. Для этого есть несколько способов; рассмотренные ниже реализованы в модуле-примеси Enumerable.

Метод detect находит не больше одного элемента. Он принимает блок (которому элементы передаются последовательно) и возвращает первый элемент, для которого значение блока оказывается равным true.

```
x = [5, 8, 12, 9, 4, 30]
# Найти первый элемент, кратный 6
x.detect {|e| e % 6 == 0 }          # 12
# Найти первый элемент, кратный 7
x.detect {|e| e % 7 == 0 }          # nil
```

Разумеется, хранящиеся в массиве объекты могут быть произвольно сложными, равно как и условие, проверяемое в блоке.

Метод find – синоним detect; метод find_all возвращает несколько элементов, а не один-единственный, а select – синоним find_all.

```
# Продолжение предыдущего примера...
x.find {|e| e % 2 == 0}             # 8
x.find_all {|e| e % 2 == 0}         # [8, 12, 4, 30]
x.select {|e| e % 2 == 0}           # [8, 12, 4, 30]
```

Метод grep вызывает оператор сравнения (то есть оператор ветвящегося равенства) для сопоставления каждого элемента с заданным образцом. В простейшей форме он возвращает массив, состоящий из элементов, соответствующих образцу. Так как используется оператор ===, то образец не обязан быть регулярным

выражением. (Имя grep пришло из UNIX и связано с командой старого редактора g/re/p.)

```
a = %w[January February March April May]
a.grep(/ary/)      # ["January, "February"]
b = [1, 20, 5, 7, 13, 33, 15, 28]
b.grep(12..24)     # [20, 13, 15]
```

Существует также блочная форма, которая позволяет преобразовать каждый результат перед записью в массив. Получающийся в результате массив содержит значения, возвращенные блоком, а не те, что были в блок первоначально переданы:

```
# Продолжение предыдущего примера...
# Будем сохранять длины строк
a.grep(/ary/) {|m| m.length}     # [7, 8]
# Будем сохранять квадраты исходных элементов
b.grep(12..24) {|n| n*n}         # [400, 169, 225]
```

Метод reject — полная противоположность select. Он исключает из массива элементы, для которых блок возвращает значение true. Имеется также вариант reject! для модификации массива «на месте»:

```
c = [5, 8, 12, 9, 4, 30]
d = c.reject {|e| e % 2 == 0}    # [5, 9]
c.reject! {|e| e % 3 == 0}
# c равно [5, 8, 4]
```

Методы min и max ищут минимальное и максимальное значение в массиве. У каждого метода есть две формы. В первой используется сравнение «по умолчанию», что бы это ни означало в конкретной ситуации (на базе оператора <=>). Во второй форме применяется блок для выполнения нестандартного сравнения.

```
a = %w[Elrond Galadriel Aragorn Saruman Legolas]
b = a.min                                    # "Aragorn"
c = a.max                                    # "Saruman"
d = a.min {|x,y| x.reverse <=> y.reverse} # "Elrond"
e = a.max {|x,y| x.reverse <=> y.reverse} # "Legolas"
```

Чтобы найти *индекс* минимального или максимального элемента (в предположении, что такой элемент один), применяется метод index:

```
# Продолжение предыдущего примера...
i = a.index a.min     # 2
j = a.index a.max     # 3
```

Такую же технику можно использовать и в других похожих ситуациях. Однако, если элемент не единственный, то будет найден только первый.

8.1.7. Специализированные функции индексирования

Для отображения индексов на элементы массива интерпретатор языка пользуется *функцией индексирования*. Поскольку методы доступа к элементам массива можно переопределять, то мы можем реализовать любой способ индексирования.

Например, ниже реализован массив, в которым индексы начинаются с 1, а не с нуля:

```
class Array2 < Array

  def [](index)
    if index>0
      super(index-1)
    else
      raise IndexError
    end
  end

  def []=(index,obj)
    if index>0
      super(index-1,obj)
    else
      raise IndexError
    end
  end

end

x = Array2.new

x[1]=5
x[2]=3
x[0]=1  # Ошибка
x[-1]=1 # Ошибка
```

Отметим, что отрицательные индексы (от конца массива) здесь запрещены. И имейте в виду, что в реальной задаче придется внести и другие изменения, например, переопределить метод slice и прочие. Но общую идею вы поняли.

Аналогичный подход можно применить для реализации многомерных массивов (мы еще вернемся к ним в разделе 8.1.11).

Можно также реализовать нечто вроде треугольной матрицы, как показано ниже. Это частный случай двумерного массива, в котором элемент в позиции x, y совпадает с элементом в позиции y, x (поэтому хранить можно только один). Иногда это бывает полезно, например, для хранения неориентированного графа (как мы покажем ближе к концу этой главы).

```
class TriMatrix

  def initialize
    @store = []
  end

  def [](x,y)
```

```
      index = (x*x+x)/2 + y
      @store[index]
    else
      raise IndexError
    end
  end

  def []=(x,y,v)
    if x > y
      index = (x*x+x)/2 + y
      @store[index] = v
    else
      raise IndexError
    end
  end

end


t = TriMatrix.new
t[3,2] = 1
puts t[3,2]  # 1

puts t[2,3]  # IndexError
```

В этом примере мы реализовали матрицу так, что номер строки должен быть больше или равен номеру столбца. Но можно было бы просто отобразить симметричные пары индексов на один тот же элемент. Проектное решение зависит от предполагаемого способа использования матрицы.

Можно было унаследовать классу `Array`, но нам кажется, что наше решение понять легче. Формула индексирования довольно сложна, но 10 минут с карандашом и бумагой хватит, чтобы убедить любого в ее правильности. Чтобы сделать этот класс по-настоящему полезным, надо бы немного усовершенствовать его, оставляем вам это в качестве упражнения.

Кроме того, треугольную матрицу можно реализовать в виде массива, содержащего массивы, размер которых увеличивается по мере увеличения номера строки. Примерно так мы и поступили в разделе 8.1.11 «Многомерные массивы». Нетривиальная часть – гарантировать, что строка случайно не окажется больше, чем положено.

8.1.8. Реализация разреженной матрицы

Иногда бывает нужен массив, в котором определена лишь небольшая часть элементов, а остальные не определены вовсе или (даже чаще) равны 0. Подобная *разреженная матрица* потребляет так много памяти зря, что были найдены способы более изощренной ее реализации.

Конечно, в большинстве случаев обычного массива Ruby вполне достаточно, так как в современных компьютерах недостатка памяти не ощущается. Элемент,

которому не присвоено значение, будет равен nil, так что на его хранение расходуется всего несколько байтов.

С другой стороны, присваивание значения элементу массива, лежащему за текущей правой границей, приводит к созданию всех промежуточных элементов, причем они получают значение nil. Например, если определены элементы от 0 до 9 и затем производится присваивание элементу 1000, то создаются также элементы с индексами от 10 до 999, равные nil. Если это неприемлемо, надо поискать альтернативу.

В предлагаемом нами варианте массивы вообще не используются. Для реализации разреженной матрицы лучше подойдет хэш. См. раздел 8.2.14 «Хэш как разреженная матрица».

8.1.9. Массивы как математические множества

В большинстве языков множества напрямую не реализованы (Pascal составляет исключение). Но массивы в Ruby обладают некоторыми свойствами, которые позволяют использовать их как множества. В этом разделе мы рассмотрим эти свойства и добавим свои собственные.

Впрочем, для более серьезных задач рекомендуем пользоваться не массивами, а классом Set из стандартной библиотеки. Этот класс рассмотрен в главе 9.

Массив нельзя назвать идеальным средством для представления множества, поскольку он может содержать дубликаты. Если вы хотите трактовать массив как множество, то дубликаты можно удалить (с помощью метода uniq или uniq!).

Над множествами производятся две основные операции: объединение и пересечение. Для этого применяются операторы | (или) и & (и) соответственно. Поскольку множество, по определению, не содержит дубликатов, то повторяющиеся элементы удаляются. (Вопреки ожиданиям тех, кому доводилось работать с объединением и пересечением массивов в других языках.)

```
a = [1, 2, 3, 4, 5]
b = [3, 4, 5, 6, 7]
c = a | b            # [1, 2, 3, 4, 5, 6, 7]
d = a & b            # [3, 4, 5]

# Дубликаты удаляются...
e = [1, 2, 2, 3, 4]
f = [2, 2, 3, 4, 5]
g = e & f            # [2, 3, 4]
```

Для объединения множеств можно использовать и оператор конкатенации (+), но он не удаляет дубликаты.

Метод - соответствует операции "разность множеств", результатом является множество, в которое входят те элементы первого множества, которые не являются элементами второго (см. раздел 8.1.12).

```
a = [1, 2, 3, 4, 5]
b = [4, 5, 6, 7]
c = a - b            # [1, 2, 3]
# Отметим, что наличие элементов 6 and 7 не отражается на результате.
```

Для «аккумулирования» множеств можно применять оператор |=; как и следовало ожидать, a |= b — то же самое, что a = a | b. Аналогично, оператор &= последовательно «сужает» множество.

Для массивов не определена операция ИСКЛЮЧАЮЩЕЕ ИЛИ, но мы можем без труда реализовать ее. В терминах теории множеств она соответствует выборке тех элементов, которые входят в объединение двух множеств, но не входят в их пересечение.

```
class Array

  def ^(other)
    (self | other) - (self & other)
  end

end

x = [1, 2, 3, 4, 5]
y = [3, 4, 5, 6, 7]
z = x ^ y            # [1, 2, 6, 7]
```

Чтобы проверить, входит ли некий элемент в множество, пользуйтесь методом include? или member? (синоним, подмешанный из модуля Comparable):

```
x = [1, 2, 3]
if x.include? 2
  puts "yes"     # Печатается "yes"
else
  puts "no"
end
```

Конечно, это некоторое отступление от канонов математики, где для обозначения принадлежности множеству применяется символ, похожий на греческую букву эпсилон. Отступление в том смысле, что множество находится слева, а не справа от оператора, то есть мы спрашиваем не «принадлежит ли данный элемент множеству», а «содержит ли множество данный элемент».

Многим это безразлично. Но привыкшие к языку Pascal или Python (или впитавшие математический формализм с молоком матери) хотели бы, чтобы было по-другому. Такую возможность мы реализуем в следующем фрагменте:

```
class Object

  def in(other)
    other.include? self
  end

end

x = [1, 2, 3]
if 2.in x
  puts "yes"     # Печатается "yes"
```

```
else
  puts "no"
end
```

Лично я предложил ввести в Ruby оператор `in`. Он должен походить на одноименный оператор в языках Pascal, Python и даже SQL. У этой идеи есть свои достоинства (к тому же `in` – уже зарезервированное слово), но единодушного одобрения она не получила. Может быть, оператор `in` появится в Ruby, а, может, и нет (в языке Elixr, который многим обязан Ruby, он реализован).

Теперь обратимся к подмножествам и надмножествам. Как определить, является ли данное множество подмножеством или надмножеством другого? Встроенных методов для этого нет, но мы можем поступить следующим образом:

```
class Array

  def subset?(other)
    self.each  do |x|
      if !(other.include? x)
        return false
      end
    end
    true
  end
  def superset?(other)
    other.subset?(self)
  end

end

a = [1, 2, 3, 4]
b = [2, 3]
c = [2, 3, 4, 5]

flag1 = c.subset? a     # false
flag2 = b.subset? a     # true
flag3 = c.superset? b   # true
```

Отметим, что мы выбрали «естественный» порядок, то есть задаем вопрос `x.subset? y` – «является ли `x` подмножеством `y`?», а не наоборот.

Для распознавания пустого множества достаточно проверить, пуст ли массив. Это делает метод `empty?`.

Операция дополнения опирается на идею *универсального множества*. Но «универсальное множество» в каждой конкретной ситуации определяется по-разному, поэтому лучшим решением будет самое простое: сначала определим, что такое универсальное множество, а потом вычислим разность:

```
universe = [1, 2, 3, 4, 5, 6]
a = [2, 3]
b = universe - a   # дополнение a = [1, 4, 5, 6]
```

Если считаете необходимым, можете определить и унарный оператор (например, - или ~) для выполнения этой операции.

Элементы множества можно перебирать, обходя массив. Единственная разница заключается в том, что элементы будут появляться в определенном порядке, а это может оказаться нежелательным. О том, как перебирать массив в случайном порядке, см. раздел 8.1.18.

Наконец, иногда возникает необходимость вычислить степень множества. Это не что иное, как множество всех подмножеств данного множества (включая его само и пустое множество). В Ruby имеется метод массива, который возвращает все комбинации его элементов длиной *n*. Степень множества состоит из объединения таких комбинаций для всех *n* от 0 до длины массива:

```
class Array

   def powerset
      (0..length).flat_map{|n| combination(n).to_a }
   end

end

x = [1, 2, 3]
y = x.powerset
# y равно:
#   [[], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]]
```

Метод flat_map пользуется методом map для получения массивов комбинаций для каждого значения длины *n*, а затем объединяет результаты в единый массив, который и возвращается.

8.1.10. Рандомизация массива

Иногда нужно переставить элементы массива в случайном порядке. Первое, что приходит на ум, – тасование карточной колоды, но есть и другие применения – например, случайная сортировка списка вопросов.

Для решения этой задачи можно воспользоваться методом shuffle:

```
x = [1, 2, 3, 4, 5]
y = x.shuffle           # [3, 2, 4, 1 ,5]
x.shuffle!              # x равно [3, 5, 4, 1, 2]
```

Выбрать случайный элемент массива (не запрещая дубликатов) можно с помощью метода sample:

```
x = [1, 2, 3, 4, 5]
x.sample                # 4
x.sample(2)             # [5, 1]
```

Наконец, не стоит забывать, что метод rand позволяет сгенерировать предсказуемую последовательность (например, для тестирования), если задать известное начальное значение с помощью метода srand (см. раздел 5.28 «Генерирование случайных чисел»).

8.1.11. Многомерные массивы

Для совсем простых случаев дополнительные измерения можно имтировать с помощью вложенных массивов. В следующем примере показан способ работы с многомерными массивами за счет перегрузки методов [ ] и [ ]= для отображения элементов на вложенный массив. Представленный класс Array3 обеспечивает рудиментарные операции с трехмерными массивами, но он далеко не полон:

```
class Array3

  def initialize
     @store = [[[]]]
  end

  def [](a,b,c)
    if @store[a]==nil ||
       @store[a][b]==nil ||
       @store[a][b][c]==nil
      return nil
    else
      return @store[a][b][c]
    end
  end

  def []=(a,b,c,x)
    @store[a] = [[]] if @store[a]==nil
    @store[a][b] = [] if @store[a][b]==nil
    @store[a][b][c] = x
  end

end

x = Array3.new
x[0,0,0] = 5
x[0,0,1] = 6
x[1,2,3] = 99

puts x[1,2,3]
```

Единственное, чего мы реально добились, -- так это удобного использования запятой в обозначении [x,y,z] вместо употребляемой в языке C нотации [x][y][z]. Если C-подобная нотация вас устраивает, то можете просто воспользоваться вложенными массивами Ruby. Еще одно мелкое достоинство -- это предотвращение ситуации, когда объектом, от имени которого вызывается оператор [ ], оказывается nil.

Если требуются более развитые многомерные массивы, то в стандартной библиотеке имеется класс Matrix, который уже упоминался в разделе 5.10. Однако для серьезных численных расчетов производительности стандартной библиотеки, скорее всего, будет недостаточно. Поэтому пользуйтесь вместо нее gem-пакетом

nmatrix – оптимизированной библиотекой многомерных массивов, созданной в рамках проекта SciRuby. Дополнительные сведения см. на сайте SciRuby.com.

8.1.12. Нахождение элементов, принадлежащих одному массиву и не принадлежащих другому

В Ruby эта задача решается проще, чем во многих других языках. Нужно всего лишь вычислить «разность множеств»:

```
text = %w[the magic words are squeamish ossifrage]
dictionary = %w[an are magic the them these words]
# Найти неправильно написанные слова
unknown = text - dictionary   # ["squeamish", "ossifrage"]
```

8.1.13. Преобразование или отображение массивов

Метод collect из модуля Enumerable часто позволяет сэкономить время и силы. Тем, кто привык к языку Smalltalk, он покажется более интуитивно очевидным, чем программистам на C.

Этот метод просто воздействует неким произвольным образом на каждый элемент массива, порождая в результате новый массив. Иными словами, он «отображает» один массив на другой (отсюда и синоним map).

```
x = %w[alpha bravo charlie delta echo foxtrot]
# Получить начальные буквы
a = x.collect {|w| w[0..0]}        # %w[a b c d e f]
# Получить длины строк
b = x.collect {|w| w.length}       # [5, 5, 7, 5, 4, 7]
# map - просто синоним
c = x.map {|w| w.length}           # [5, 5, 7, 5, 4, 7]
```

Имеется также вариант collect! (или map!) для модификации на месте.

```
x.collect! {|w| w.upcase}
# x равно %w[ALPHA BRAVO CHARLIE DELTA ECHO FOXTROT]
x.map! {|w| w.reverse}
# x равно %w[AHPLA OVARB EILRAHC ATLED OHCE TORTXOF]
```

8.1.14. Удаление элементов, равных nil, из массива

Метод compact (и его вариант compact! для модификации на месте) удаляет из массива элементы, равные nil, оставляя все остальные без изменения:

```
a = [1, 2, nil, 3, nil, 4, 5]
b = a.compact                  # [1, 2, 3, 4, 5]
a.compact!                     # a равно [1, 2, 3, 4, 5]
```

8.1.15. Удаление заданных элементов из массива

В Ruby легко удалить элементы из массива, для этого даже существует много способов. Чтобы удалить элемент с известным индексом, достаточно вызвать метод delete_at:

```
a = [10, 12, 14, 16, 18]
a.delete_at(3)              # Возвращает 16
# a равно [10, 12, 14, 18]
a.delete_at(9)              # Возвращает nil (вне диапазона)
```

Все элементы с заданным значением поможет удалить метод delete. Он возвращает значения удаленных элементов или nil, если искомый элемент не найден:

```
b = %w(spam spam bacon spam eggs ham spam)
b.delete("spam")            # Возвращает "spam"
# b равно ["bacon", "eggs", "ham"]
b.delete("caviar")          # Возвращает nil
```

Метод delete принимает также блок. Это не вполне согласуется с интуицией; если объект не найден, происходит вычисление блока (при этом могут выполняться разнообразные операции) и возвращается вычисленное значение.

```
c = ["alpha", "beta", "gamma", "delta"]
c.delete("delta") { "Nonexistent" }
# Возвращается "delta" (блок не вычисляется)
c.delete("omega") { "Nonexistent" }
# Возвращается "Nonexistent"
```

Метод delete_if передает каждый элемент массива в блок и удаляет те элементы, для которых вычисление блока дает true. Примерно так же ведет себя метод reject! с тем отличием, что последний может возвращать nil, когда массив не изменяется.

```
email = ["job offers", "greetings", "spam", "news items"]
# Удалить слова из четырех букв
email.delete_if {|x| x.length==4 }
# email равно ["job offers", "greetings", "news items"]
```

Метод slice! получает доступ к тем же элементам, что и slice, но, помимо возврата их значений, еще и удаляет из массива:

```
x = [0, 2, 4, 6, 8, 10, 12, 14, 16]
a = x.slice!(2)                         # 4
# x равно [0, 2, 6, 8, 10, 12, 14, 16]
b = x.slice!(2,3)                       # [6, 8, 10]
# x равно  [0, 2, 12, 14, 16]
c = x.slice!(2..3)                      # [12, 14]
# x равно  [0, 2, 16]
```

Для удаления элементов из массива можно также пользоваться методами shift и pop. (Дополнительную информацию об их исходном предназначении см. в разделе 9.2).

```
x = [1, 2, 3, 4, 5]
x.pop                   # Удалить последний элемент
# x равно [1, 2, 3, 4]
x.shift                 # Удалить первый элемент
# x равно [2, 3, 4]
```

Метод reject принимает блок и формирует новый массив без тех элементов, для которых блок возвращает true:

```
arr = [1,2,3,4,5,6,7,8]
odd = arr.reject {|x| x % 2 == 0 }       # [1,3,5,7]
```

Наконец, метод clear удаляет из массива все элементы. Это эквивалентно присваиванию переменной пустого массива, но чуть-чуть эффективнее:

```
x = [1, 2, 3]
x.clear
# x равно []
```

8.1.16. Конкатенирование массивов и добавление в конец массива

Часто нужно добавить в конец существующего массива отдельный элемент или целый массив. В Ruby это можно сделать разными способами.

Оператор << добавляет объект в конец массива, в качестве значения он возвращает сам массив, поэтому можно объединять несколько таких операций в цепочку.

```
x = [1, 5, 9]
x << 13        # x равно [1, 5, 9, 13]
x << 17 << 21  # x равно [1, 5, 9, 13, 17, 21]
```

Аналогичную операцию выполняют методы unshift и push, которые добавляют элемент в начало и в конец массива соответственно. См. также раздел 8.1.17 ниже.

```
x = [1, 5, 9]
x.push *[2, 6, 10]   # x равно [1, 5, 9, 2, 6, 10]
x.unshift 3          # x равно [3, 1, 5, 9, 2, 6, 10]
```

Массивы можно конкатенировать методом concat или с помощью операторов + и +=:

```
x = [1,2]
y = [3,4]
z = [5,6]
b = y + z      # [3,4,5,6]
b += x         # [3,4,5,6,1,2]
z.concat y     # z равно [5,6,3,4]
```

Имейте в виду, что оператор += всегда создает новый объект. Также не забывайте, что оператор << добавляет в конец новый элемент, который сам может быть массивом.

```
a = [1,2]
b = [3,4]
a += b           # [1,2,3,4]

a = [1,2]
b = [3,4]
a <<  b          # [1,2,[3,4]]

a = [1,2]
b = [3,4]
a = a.concat(b)  # [1,2,3,4]
```

8.1.17. Использование массива в качестве стека или очереди

Базовые операции со стеком называются push и pop, они добавляют и удаляют элементы в конец массива. Базовые операции с очередью – это shift (удаляет элемент из начала массива) и unshift (добавляет элемент в начало массива). Для добавления в конец массива можно также пользоваться оператором << (по существу, синоним push).

Постарайтесь не запутаться. Методы shift и unshift модифицируют массив в начале, а push, pop и << – в конце.

Продолжение этой темы см. в разделе 9.2 «Стеки и очереди».

8.1.18. Обход массива

Как и следовало ожидать, в классе Array есть стандартный итератор each. Но имеются и другие полезные итераторы.

Метод reverse_each обходит массив в обратном порядке. Результат такой же, как если бы вызвать сначала метод reverse, а потом each, но работает быстрее.

```
words = %w{Son I am able she said}
str = ""
words.reverse_each { |w| str += "#{w} "}
# str равно "said she able am I Son "
```

Если нужно только перебрать все индексы, можно воспользоваться итератором each_index. Конструкция x.each_index эквивалентна (0..(x.size-1)).each (то есть обходу всего диапазона индексов).

Итератор each_with_index (подмешанный из модуля Comparable) передает в блок как сам элемент, так и его индекс.

```
x = ["alpha", "beta", "gamma"]
x.each_with_index do |x,i|
  puts "Элемент #{i} равен #{x}"
end
# Выводится три строки
```

Предположим, что нужно обойти массив в случайном порядке. Ниже представлен итератор random_each (который просто вызывает метод shuffle).

```
class Array

# Предполагается, что метод randomize определен

  def random_each
        self.shuffle.each { |x| yield x}
  end

end

dwarves = %w{Sleepy Dopey Happy Sneezy Grumpy Bashful Doc}
list = ""
dwarves.random_each {|x| list += "#{x} "}
```

```
# list равен:
# "Bashful Dopey Sleepy Happy Grumpy Doc Sneezy "
# (На вашей машине порядок может быть другим.)
```

8.1.19. Преобразование массива в строку с разделителями

Часто бывает необходимо вставить разделители между элементами массива, но не перед первым и не после последнего. Для этого предназначен метод join и оператор *.

```
been_there = ["Veni", "vidi", "vici."]
journal = been_there.join(", ")        # "Veni, vidi, vici."

letters = ["Phi","Mu","Alpha"]
musicians = letters.join(" ")          # "Phi Mu Alpha"

people = ["Bob","Carol","Ted","Alice"]
movie = people * " and "
# movie равно "Bob and Carol and Ted and Alice"
```

Если необходимо последний элемент обрабатывать особым образом, например, вставить перед ним слово «and», то это можно сделать вручную:

```
list = %w[A B C D E F]
with_commas = list[0..-2]*", " + ", and " + list[-1]
# with_commas равно "A, B, C, D, E, and F"
```

8.1.20. Обращение массива

Чтобы переставить элементы массива в обратном порядке, воспользуйтесь методом reverse или reverse!:

```
inputs = ["red", "green", "blue"]
outputs = inputs.reverse          # ["green","blue","red"]
priorities = %w(eat sleep code)
priorities.reverse!               # ["code","sleep","eat"]
```

8.1.21. Удаление дубликатов из массива

Чтобы удалить из массива повторяющиеся экземпляры, воспользуйтесь методом uniq (или его вариантом для модификации на месте uniq!):

```
breakfast = %w[spam spam eggs ham eggs spam]
lunch = breakfast.uniq   # ["spam","eggs","ham"]
breakfast.uniq!          # массив breakfast изменился
```

8.1.22. Чередование массивов

Предположим, что есть два массива и надо построить из них третий, который содержит массивы из двух элементов, взятых из соответственных позиций исходных массивов. Это делает метод zip из модуля Enumerable:

```
a = [1, 2, 3, 4]
b = ["a", "b", "c", "d"]
```

```
c = a.zip(b)
# c равно [[1,"a"], [2,"b"], [3,"c"], [4,"d"]]
# Чтобы устранить вложенность, воспользуйтесь методом flatten
d = c.flatten
# d равно [1, "a", 2, "b", 3, "c", 4, "d"]
```

Если задан блок, то создаваемые подмассивы будут один за другим передаваться ему:

```
a.zip(b) {|x1, x2| puts x2 + "-" + x1.to_s }
# Печатается: a-1
#             b-2
#             c-3
#             d-4
# и возвращается nil
```

8.1.23. Вычисление частоты различных значений в массиве

Для массивов нет метода `count`, как для строк (чтобы подсчитать число вхождений каждого элемента). Поэтому создадим свой собственный:

```
class Array

  def count
        each_with_object(Hash.new(0)){|x,h| h[x] += 1 }
  end

end

meal = %w[spam spam eggs ham eggs spam]
items = meal.count
# items равно {"ham" => 1, "spam" => 3, "eggs" => 2}
spams = items["spam"]   # 3
```

Обратите внимание, что метод возвращает хэш.

8.1.24. Инвертирование массива для получения хэша

Массив нужен для того, чтобы ассоциировать целое число (индекс) с данными. А если нужно инвертировать это отношение, то есть ассоциировать данные с индексом? Иными словами, породить хэш? Это можно сделать так:

```
class Array

  def invert
        each_with_object({}).with_index{|(x,h),i| h[x] = i }
  end

end

a = ["red","yellow","orange"]
h = a.invert     # {"orange"=>2, "yellow"=>1, "red"=>0}
```

8.1.25. Синхронная сортировка нескольких массивов

Предположим, что необходимо отсортировать массив, которому соответствуют «параллельные» массивы, то есть в соответственных позициях находятся логически связанные данные. Не хотелось бы, чтобы в результате сортировки это соответствие нарушилось.

В представленном ниже решении мы сортируем массив и сохраняем получившийся набор индексов. Затем список индексов (который сам является массивом) можно применить к любому другому массиву, чтобы расставить его элементы в том же порядке.

```
class Array

  def sort_index
    d=[]
    self.each_with_index{|x,i| d[i]=[x,i]}
    if block_given?
      d.sort {|x,y| yield x[0],y[0]}.collect{|x| x[1]}
    else
      d.sort.collect{|x| x[1]}
    end
  end

  def sort_with(ord=[])
    return nil if self.length!=ord.length
    self.values_at(*ord)
  end

end

a = [21, 33, 11, 34, 36, 24, 14]
b = a.sort_index
a2 = a.sort_with(b)
c = a.sort_index {|x,y| x%2 <=> y%2 }
a3 = a.sort_with(c)

p a      # [21, 33, 11, 34, 36, 24, 14]
p b      # [2, 6, 0, 5, 1, 3, 4]
p a2       # [11, 14, 21, 24, 33, 34, 36]
p c      # [6, 5, 4, 3, 2, 1, 0]
p a3       # [14, 24, 36, 34, 11, 33, 21]
```

8.1.26. Задание значения по умолчанию для новых элементов массива

Когда массив растет, и в нем создаются новые элементы, по умолчанию им присваивается значение nil:

```
a = Array.new
a[0]="x"
```

```
a[3]="y"
# a равно ["x", nil, nil, "y"]
```

А если мы хотим, чтобы новые элементы получали другое значение? В качестве конкретного применения общего принципа предлагаем класс ZArray, описывающий массив, в котором вновь созданные элементы будут равны 0:

```
class ZeroArray < Array

  def [](x)
    if x > size
            (size+1..x).each{|i| self[i] = 0 }
    end
    v = super(x)
  end

  def []=(x,v)
    max = size
    super(x,v)
    if size - max > 1
            (max..size-2).each{|i| self[i] = 0 }
    end
  end

end


num = ZeroArray.new
num[1] = 1
num[2] = 4
num[5] = 25
# num равно [0, 1, 4, 0, 0, 25]
```

8.2. Хэши

Хэши еще называют ассоциативными массивами, словарями и т.д. Особенно хорошо эта структура данных знакома программистам на языках Python, Perl и Java.

Массив можно представлять себе структурой, которая создает ассоциацию между индексом x и элементом данных y. Хэш тоже создает подобную ассоциацию, но с двумя отличиями. Во-первых, в случае массива x – целое число, а для хэша это не обязательно. Во-вторых, массив – упорядоченная структура, тогда как элементы в хэше обычно располагаются в непредсказуемом порядке. (Правда, начиная с версии 1.9, в Ruby отслеживается упорядоченность хэша.)

Ключ хэша может иметь произвольный тип. Как следствие, хэш является не последовательной структурой данных. Мы знаем, что в массиве четвертый элемент следует за третьим. А в хэше тип ключа может быть таким, что понятия следующего и предыдущего значения не определены. Однако, начиная с версии Ruby 1.9,

хэш «помнит» порядок вставки элементов, что превращает его в своего рода упорядоченную структуру данных.

Можно считать, что хэш – это массив со специальным индексом, или некий аналог «таблицы синонимов» в базе данных, только оба поля хранятся в памяти. Как бы вы ни представляли себе хэш, это полезный и мощный инструмент программирования.

8.2.1. Создание нового хэша

Как и в случае класса Array, для создания хэша служит специальный метод класса [ ]. Данные, перечисленные в квадратных скобках, образуют ассоциированные пары. Ниже показаны шесть способов вызвать этот метод (все хэши с a1 до c2 содержат одни и те же данные).

```
a1 = Hash.[]("flat",3,"curved",2)
a2 = Hash.[]("flat"=>3,"curved"=>2)
b1 = Hash["flat",3,"curved",2]
b2 = Hash["flat"=>3,"curved"=>2]
c1 = {"flat",3,"curved",2}
c2 = {"flat"=>3,"curved"=>2}
# Для a1, b1 и c1: число элементов должно быть четным.
```

Существует также альтернативный синтаксис для задания литерального хэша в частном случае, когда ключами являются символы. Он был добавлен в версии Ruby 1.9. Начальное двоеточие в нем опускается, а стрелка заменяется двоеточием:

```
h1 = {:alpha => 123, :beta => 456}
h2 = {alpha: 123, beta: 456}
h1 == h2 # true
```

Имеется также метод класса new, который может принимать параметр, задающий значение *по умолчанию*. Отметим, что это значение не является частью хэша, оно просто возвращается вместо nil.

```
d = Hash.new          # Создать пустой хэш
e = Hash.new(99)      # Создать пустой хэш
f = Hash.new("a"=>3)  # Создать пустой хэш
e["angled"]           # 99
e.inspect             # {}
f["b"]                # {"a"=>3} (значением по умолчанию
                      #   является тоже хэш)
f.inspect             # {}
```

Наконец, упомянем о методе to_h из класса Array. Он преобразует произвольный массив двухэлементных массивов в хэш, состоящий из ключей и значений:

```
g = [["a", 1]].to_h # {"a" => 1}
```

8.2.2. Задание значения по умолчанию для хэша

Значением по умолчанию для хэша является объект, возвращаемый вместо nil в случае, когда указанный ключ не найден. Это полезно, если вы планируете вызы-

вать для возвращенного значения методы, которые для nil не определены. Задать значение по умолчанию можно в момент создания хэша или позже с помощью метода default=.

Все отсутствующие ключи указывают на один и тот же объект по умолчанию, поэтому изменение этого объекта имеет побочный эффект.

```
a = Hash.new("missing")  # объект по умолчанию - строка "missing"
a["hello"]               # "missing"
a.default="nothing"
a["hello"]               # "nothing"
a["good"] << "bye"       # "nothingbye"
a.default                # "nothingbye"
```

В отличие от метода default, блок позволяет задать свое значение вместо каждого отсутствующего ключа. Типичная идиома – хэш, для которого значением по умолчанию является массив, так что элементы можно добавлять, не проверяя явно, что значение отсутствует, и не создавая пустой массив:

```
a = Hash.new{|h, key| h[key] = []} # значением по умолчанию является new []
a["hello"]                         # []
a["good"] << "bye"                 # {"good" => ["bye"]}
```

Имеется также специальный метод экземпляра fetch, который возбуждает исключение IndexError, если в объекте типа Hash нет указанного ключа. Он принимает также второй параметр, играющий роль значения по умолчанию. Кроме того, методу fetch можно передать необязательный блок, который выработает значение по умолчанию, если ключ не будет найден. Принцип здесь такой же, как при создании значений по умолчанию блоком.

```
a = {"flat",3,"curved",2,"angled",5}
a.fetch("pointed")                  # IndexError
a.fetch("curved","na")              # 2
a.fetch("x","na")                   # "na"
a.fetch("flat") {|x| x.upcase}      # 3
a.fetch("pointed") {|x| x.upcase}   # "POINTED"
```

8.2.3. Доступ к парам ключ-значение и добавление новых пар

В классе Hash есть методы класса [ ] and [ ]=. Используются они почти так же, как одноименные методы в классе Array, но принимают лишь один параметр. В качестве параметра может выступать любой объект, а не только строка (хотя строки используются чаще всего).

```
a = {}
a["flat"] = 3           # {"flat"=>3}
a.[]=("curved",2)       # {"flat"=>3,"curved"=>2}
a.store("angled",5)     # {"flat"=>3,"curved"=>2,"angled"=>5}
```

Метод store – просто синоним [ ]=, оба могут принимать два аргумента, как показано в примере выше.

Метод [] аналогичен методу fetch, но не возбуждает исключение IndexError, когда ключ отсутствует, а возвращает nil.

```
a.fetch("flat")    # 3
a.[]("flat")       # 3
a["flat"]          # 3
a["bent"]          # nil
```

Предположим, что мы не уверены, существует ли объект Hash, но хотели бы избежать очистки имеющегося хэша. Очевидное решение — проверить, определен ли интересующий нас объект:

```
a = {} unless defined? a
a["flat"] = 3
```

Но есть и другой, более идиоматический, способ:

```
a ||= {}
a["flat"] = 3
# Или даже так:
(a ||= {})["flat"] = 3
```

Тот же вопрос можно поставить для отдельных ключей, когда новое значение следует присваивать лишь, если такого ключа еще нет:

```
a=Hash.new(99)
a[2]                # 99
a                   # {}
a[2] ||= 5          # 99
a                   # {}
b=Hash.new
b                   # {}
b[2]                # nil
b[2] ||= 5          # 5
b                   # {2=>5}
```

Отметим, что nil может выступать и в качестве ключа, и в качестве значения:

```
b={}
b[2]                # nil
b[3]=nil
b                   # {3=>nil}
b[2].nil?           # true
b[3].nil?           # true
b[nil]=5
b                   # {3=>nil,nil=>5}
b[nil]              # 5
b[b[3]]             # 5
```

8.2.4. Удаление пар ключ-значение

Удалить пары ключ-значение из хэша можно с помощью методов clear, delete, delete_if, reject, reject! и shift.

Метод clear удаляет из хэша все пары. Эффект такой же, как от присваивания переменной нового пустого хэша, но работает чуть быстрее.

Метод shift удаляет незаданную пару ключ-значение и возвращает ее в виде массива из двух элементов или nil, если никаких ключей не осталось.

```
a = {1=>2, 3=>4}
b = a.shift          # [1,2]
# a равно {3=>4}
```

Метод delete удаляет конкретную пару ключ-значение. Он принимает в качестве параметра ключ и возвращает ассоциированное с ним значение, если такой ключ существовал (и был удален). В противном случае возвращается значение по умолчанию. Метод также принимает блок, который вырабатывает уникальное значение по умолчанию вместо того, чтобы возвращать ссылку на общий объект.

```
a = {1=>1, 2=>4, 3=>9, 4=>16}
a.delete(3)                          # 9
# a равно {1=>1, 2=>4, 4=>16}
a.delete(5)                          # в этом случае nil
a.delete(6) { "не найдено" }         # "не найдено"
```

Пользуйтесь методами delete_if, reject или reject! в сочетании с обязательным блоком, чтобы удалить все ключи, для которых блок возвращает значение true. Метод reject работает с копией хэша, а метод reject! возвращает nil, если не было произведено никаких изменений.

8.2.5. Обход хэша

В классе Hash имеется стандартный итератор each, а кроме него итераторы each_key, each_pair и each_value (each_pair – синоним each).

```
{"a"=>3,"b"=>2}.each do |key, val|
  print val, " из ", key, "; "# 3 из a; 2 из b;
end
```

Остальные два итератора передают в блок только ключ или только значение:

```
{"a"=>3,"b"=>2}.each_key do |key|
  print "ключ = #{key};" # Печатается: ключ = a; key = b;
end

{"a"=>3,"b"=>2}.each_value do |value|
  print "значение = #{value};"# Печатается: значение = 3; val = 2;
end
```

8.2.6. Инвертирование хэша

Инвертирование хэша осуществляется в Ruby тривиально с помощью метода invert:

```
a = {"fred"=>"555-1122","jane"=>"555-7779"}
b = a.invert
b["555-7779"]    # "jane"
```

Поскольку ключи в хэше уникальны, такая операция может привести к потере данных. Значения-дубликаты будут преобразованы в уникальный ключ, которому соответствует какое-то одно из множества прежних значений. Предсказать, какое именно, невозможно.

8.2.7. Поиск ключей и значений в хэше

Определить, было ли присвоено значение некоторому ключу, позволяет метод has_key? или любой из его синонимов include?, key?, member?:

```
a = {"a"=>1,"b"=>2}
a.has_key? "c"    # false
a.include? "a"    # true
a.key? 2          # false
a.member? "b"     # true
```

Можно также воспользоваться методом empty?, чтобы узнать, остался ли в хэше хотя бы один ключ. А метод length и его синоним size позволяют узнать, сколько ключей имеется в хэше:

```
a.empty?          # false
a.length          # 2
```

Можно проверить также, существует ли указанное значение. Для этого предназначены методы has_value? или value?:

```
a.has_value? 2    # true
a.value? 99       # false
```

8.2.8. Копирование хэша в массив

Чтобы преобразовать весь хэш в массив, пользуйтесь методом to_a. Получившийся массив содержит двухэлементные массивы, содержащие пары ключ-значение:

```
h = {"a" => 1, "b" => 2}
h.to_a # [["a", 1], ["b", 2]]
```

Можно также получить массив, содержащий только ключи или только значения:

```
h.keys          # ["a","b"]
h.values        # [1,2]
```

Наконец, можно поместить в массив только значения, соответствующие заданному списку ключей. Этот метод работает для хэшей примерно так же, как одноименный метод для массивов. (Кроме того, как и в случае массивов, метод values_at заменяет устаревшие методы indices и indexes.)

```
h = {1=>"one",2=>"two",3=>"three",4=>"four","cinco"=>"five"}
h.values_at(3,"cinco",4)      # ["three","five","four"]
h.values_at(1,3)              # ["one","three"]
```

8.2.9. Выборка пар ключ-значение по заданному критерию

К классу Hash подмешан модуль Enumerable, поэтому можно обращаться к методам detect (find), select (find_all), grep, min, max и reject (как и для массивов).

Метод detect (синоним find) находит одну пару ключ-значение. Он принимает блок (которому передается по одной паре за раз) и возвращает первую пару, для которой вычисление блока дает true.

```
names = {"fred"=>"jones","jane"=>"tucker",
         "joe"=>"tucker","mary"=>"SMITH"}
```

```
# Найти tucker
names.detect {|k,v| v=="tucker" }      # ["joe","tucker"]
# Найти имена, записанные заглавными буквами
names.find {|k,v| v==v.upcase }        # ["mary", "SMITH"]
```

Разумеется, объекты в хэше могут быть сколь угодно сложными, как и условие, проверяемое в блоке, но сравнение объектов разных типов может оказаться проблематичным.

Метод select (синоним find_all) возвращает все пары, удовлетворяющие условию, а не только первую:

```
names.select {|k,v| v=="tucker" }
# [["joe", "tucker"], ["jane", "tucker"]]
names.find_all {|k,v| k.count("r")>0}
# [["mary", "SMITH"], ["fred", "jones"]]
```

8.2.10. Сортировка хэша

Хэши по природе своей не упорядочены ни по ключам, ни по значениям. Чтобы отсортировать хэш, Ruby преобразует его в массив, а затем сортирует этот массив. Понятно, что и результатом является массив.

```
names = {"Jack"=>"Ruby","Monty"=>"Python",
         "Blaise"=>"Pascal", "Minnie"=>"Perl"}
list = names.sort
# list равно:
# [["Blaise","Pascal"], ["Jack","Ruby"],
#  ["Minnie","Perl"], ["Monty","Python"]]
```

Ниже показано, как такой массив преобразовать обратно в хэш:

```
list_hash = list.to_h
```

См. также раздел 8.2.12 «Создание хэша из массива».

8.2.11. Объединение двух хэшей

Иногда бывает нужно объединить хэши. Метод merge получает два хэша и формирует из них третий, перезаписывая обнаружившиеся дубликаты:

```
dict = {"base"=>"foundation", "pedestal"=>"base"}
added = {"base"=>"non-acid", "salt"=>"NaCl"}
new_dict = dict.merge(added)
# {"base"=>"non-acid", "pedestal"=>"base", "salt"=>"NaCl"}
```

У метода merge есть синоним update.

Если задан блок, то он может содержать алгоритм устранения коллизий. В примере ниже, если два ключа совпадают, то в объединенном хэше остается меньшее значение (по алфавиту, по числовому значению или в каком-то ином смысле):

```
dict = {"base"=>"foundation", "pedestal"=>"base"}
added = {"base"=>"non-acid", "salt"=>"NaCl"}
new_dict = dict.merge(added) {|key,old,new| old < new ? old : new }
# {"salt"=>"NaCl", "pedestal"=>"base", "base"=>"foundation"}
```

Таким образом, при использовании блока результат может получиться не такой, как в случае, когда блок не задан. Имеются также методы merge! и update!, которые изменяют вызывающий объект «на месте».

8.2.12. Создание хэша из массива

Простейший способ сделать это прибегнуть к методу to_h массива, содержащего двухэлементные массивы. Можно также воспользоваться методом [] класса Hash, которому передаются либо двухэлементные массивы, либо один массив, содержащий четное число элементов.

```
pairs = [[2, 3], [4, 5], [6, 7]]
array = [2, 3, 4, 5, 6, 7]
h1 = pairs.to_h      # {2 => 3, 4 => 5, 6 => 7}
h2 = Hash[pairs]     # {2 => 3, 4 => 5, 6 => 7}
h3 = Hash[*array]    # {2 => 3, 4 => 5, 6 => 7}
```

8.2.13. Вычисление разности и пересечения хэшей

Ключи хэша можно скопировать в отдельный массив, а к получившимся из разных хэшей массивам применить методы & и - класса Array. В результате мы получим пересечение и разность множеств ключей. Соответствующие им значения можно получить с помощью метода each, примененного к третьему хэшу, который представляет объединение обоих хэшей (чтобы все ключи можно было гарантированно найти в одном месте).

```
a = {"a" => 1,"b" => 2,"z" => 3}
b = {"x" => 99,"y" => 88,"z" => 77}
intersection = a.keys & b.keys
difference = a.keys - b.keys
c = a.dup.update(b)
inter = {}
intersection.each {|k| inter[k]=c[k] }
# inter равно {"z"=>77}
diff={}
difference.each { k| diff[k]=c[k] }
# diff равно {"a"=>1, "b"=>2}
```

8.2.14. Хэш как разреженная матрица

Часто в массиве или матрице заполнена лишь небольшая часть элементов. Можно хранить их как обычно, но такое расходование памяти неэкономно. Хэш позволяет хранить только реально существующие значения.

В следующем примере предполагаются, что несуществующие значения по умолчанию равны нулю:

```
values = Hash.new(0)
values[1001] = 5
values[2010] = 7
values[9237] = 9
x = values[9237]      # 9
y = values[5005]      # 0
```

Ясно, что обычный массив в таком случае содержал бы более 9000 неиспользуемых элементов, что не всегда приемлемо.

А если нужно реализовать разреженную матрицу размерности два или более? В этом случае можно было бы использовать массивы в качестве ключей:

```
cube = Hash.new(0)
cube[[2000,2000,2000]] = 2
z = cube[[36,24,36]]       # 0
```

Здесь обычная матрица содержала бы миллиарды элементов.

8.2.15. Реализация хэша с повторяющимися ключами

Приверженцы математической строгости скажут, что хэш с повторяющимися ключами – вообще не хэш. Не станем спорить. Называйте, как хотите, но на практике бывают случаи, когда нужна структура данных, обладающая гибкостью и удобством хэша и в то же время содержащая ключи-дубликаты.

В листинге 8.1 предложено частичное решение. Оно неполно по двум причинам. Во-первых, мы не стали реализовывать всю желательную функциональность, а лишь некоторое достаточно представительное подмножество. Во-вторых, внутреннее устройство Ruby таково, что литеральный хэш всегда является экземпляром класса `Hash`, но, даже если бы мы наследовали классу `Hash`, литерал все равно не мог бы содержать повторяющихся ключей.

Листинг 8.1. Хэш с повторяющимися ключами

```
class HashDup

  def initialize(*all)
      all.flatten!
      raise IndexError if all.size % 2 != 0
      @store = {}
      all.each do |key, val|
      self.store(key, val)
        end
  end

   def store(k, v)
      if @store.has_key?(k)
        @store[k] += [v]
      else
        @store[k] = [v]
      end
   end

   def [](key)
      @store[key]
   end

   def []=(key,value)
```

```
    self.store(key,value)
end

def to_s
    @store.to_s
end

def to_a
    @store.to_a
end

def inspect
    @store.inspect
end

def keys
    result=[]
    @store.each do |k,v|
       result += ([k]*v.size)
    end
    result
end

def values
    @store.values.flatten
end

def each
    @store.each {|k,v| v.each {|y| yield k, y }}
end

alias each_pair each

def each_key
    self.keys.each {|k| yield k }
end

def each_value
    self.values.each {|v| yield v }
end

def has_key? k
    self.keys.include? k
end

def has_value? v
    self.values.include? v
end
```

```
    def length
      self.values.size
    end

    alias size length

    def delete(k)
      val = @store[k]
      @store.delete k
      val
    end

    def delete(k,v)
      @store[k] -= [v] if @store[k]

  # Остальные методы опущены...

end


# Не будет работать... для повторяющихся ключей
# первое вхождение игнорируется.
h = {1 => 1, 2 => 4, 3 => 9, 4 => 16, 2 => 0}

# А так будет...
h = HashDup.new(1,1, 2,4, 3,9, 4,16, 2,0)

k = h.keys          # [4, 1, 2, 2, 3]
v = h.values        # [16, 1, 4, 0, 9]

n = h.size          # 5

h.each {|k,v| puts "#{k} => #{v}"}
# Печатается:
# 4 => 16
# 1 => 1
# 2 => 4
# 2 => 0
# 3 => 9
```

Но если не пользоваться литеральными хэшами, то задача решаема. В листинге 8.1 реализован класс, содержащий атрибут `@store`, который является обычным хэшем; каждое значение этого хэша представляет собой массив. Доступ к этому хэшу организован так, что при необходимости добавить ключ, который уже существует, мы на самом деле добавляем новое значение в массив, ассоциированный с этим ключом.

Что должен возвращать метод `size`? Очевидно, «истинное» число пар ключ-значение, *включая* и дубликаты. Аналогично метод `keys` возвращает массив, кото-

рый может содержать дубликаты. Итераторы ведут себя естественно; как и в случае обычного хэша, порядок обхода не предсказуем.

Помимо стандартного метода delete, мы реализовали метод delete_pair. Первый удаляет *все* значения, ассоциированные с данным ключом, второй — только конкретную пару ключ-значение. (Отметим, что было бы затруднительно реализовать единственный метод вида delete(k,v=nil), так как nil — допустимое значение в любом хэше.)

Для краткости мы не стали реализовывать весь класс целиком и, честно говоря, для некоторых методов, например invert, пришлось бы принимать нетривиальные решения по поводу желательного поведения. Интересующийся читатель может восполнить пробелы.

8.2.16. Другие операции с хэшем

Хэши обычно индексируются на основе *значений* ключей. При желании можно изменить поведение, так чтобы сравнение производилось по идентификатору объекта. Метод compare_by_identity переводит хэш в этот специальный режим, а метод compare_by_identity? сообщает, в каком режиме находится хэш:

```
h1 = { "alpha" => 1, :echo => 2, 35 => 3 }
h2 = h1.dup.compare_by_identity

h1.compare_by_identity?                  # false
h2.compare_by_identity?                  # true

a1 = h1.values_at("alpha", :echo, 35)    # [1, 2, 3]
a2 = h2.values_at("alpha", :echo, 35)    # [nil, 2, 3]
```

Причина продемонстрированного выше поведения, конечно же, в том, что символы и целые числа в Ruby являются непосредственными значениями (как и true, false и nil). Однако же для строк в кавычках создаются разные объекты с разными идентификаторами.

8.3. Перечисляемые структуры в общем

Что делает набор *перечисляемым*? Вообще-то сам тот факт, что это коллекция. Модуль Enumerable требует, чтобы был определен стандартный итератор each. Последовательность обхода не имеет значения, так как даже неупорядоченные наборы, например хэш, могут обладать итераторами.

Кроме того, если предполагается пользоваться методами min, max и sort, то для коллекции должен быть определен метод сравнения (<=>). Это все достаточно очевидно.

Итак, перечисляемая структура — это коллекция, в которой можно производить поиск, которую можно обойти и, быть может, отсортировать. В любую определенную пользователем коллекцию, не являющуюся подклассом существующего системного класса, имеет смысл подмешивать модуль Enumerable.

Имейте в виду – все сказанное о какой-то одной перечисляемой структуре относится ко всем. В качестве примеров таких структур можно назвать массив, хэш, дерево и т. д.

Конечно, у каждой структуры есть свои нюансы. Массив – это упорядоченная коллекция отдельных элементов, а хэш – неупорядоченная коллекция пар ключ-значение. Понятно, что в каких-то отношениях они будут вести себя по-разному.

Многие методы, с которыми мы познакомились при изучении массивов и хэшей (например, `map` и `find`), на самом деле определены в модуле `Enumerable`. Часто было трудно решить, как подать материал. Любая путаница или неточность – моя вина.

Массив – наиболее часто употребляемая коллекция, к которой подмешивается этот модуль. Поэтому по умолчанию я буду пользоваться в примерах именно массивами.

8.3.1. Метод inject

Метод `inject` пришел в Ruby из языка Smalltalk (впервые он появился в версии Ruby 1.8). Его поведение интересно, хотя с первого раза понять его нелегко.

Он отражает тот факт, что мы часто хотим обойти список и по ходу «аккумулировать» некоторый результат. Конечно, самый естественный пример – суммирование чисел в списке. Но и для других операций обычно есть некий «аккумулятор» (которому присваивается начальное значение) и применяемая функция (в Ruby она представлена блоком).

В качестве тривиального примера рассмотрим массив чисел, которые нужно просуммировать:

```
nums = [3,5,7,9,11,13]
sum = nums.inject(0) {|x,n| x+n }
```

Обратите внимание, что начальное значение аккумулятора равно 0 («нейтральный элемент» для операции сложения). Затем блок получает текущее значение аккумулятора и значение текущего элемента списка. Действие блока заключается в прибавлении нового значения к текущей сумме.

Ясно, что этот код эквивалентен следующему:

```
sum = 0
nums.each {|n| sum += n }
```

В данном случае уровень абстракции лишь немногим выше. Если идея метода `inject` не укладывается у вас в голове, не пользуйтесь им. Но если удалось преодолеть первоначальное непонимание, то вы сможете найти ему новые элегантные применения.

Начальное значение аккумулятора задавать необязательно. Если оно опущено, то в качестве такового используется значение первого элемента, который при последующих итерациях пропускается.

```
sum = nums.inject {|x,n| x+n }

# То же самое, что:

sum = nums[0]
nums[1..-1].each {|n| sum += n }
```

Похожий пример – вычисление произведения чисел. В данном случае аккумулятору следует присвоить начальное значение 1 (нейтральный элемент для операции умножения).

```
prod = nums.inject(1) {|x,n| x*n }

# или

prod = nums.inject {|x,n| x*n }
```

В следующем немного более сложном примере мы находим самое длинное слово в списке:

```
words = %w[ alpha beta gamma delta epsilon eta theta ]
longest_word = words.inject do |best,w|
  w.length > best.length ? w : best
end
# возвращается значение "epsilon"
```

8.3.2. Кванторы

Кванторы any? и all? упрощают проверку свойств коллекции. Оба квантора принимают в качестве параметра блок (который должен возвращать значение true или false).

```
nums = [1,3,5,8,9]

# Есть ли среди чисел четные?
flag1 = nums.any? {|x| x % 2 == 0 }    # true

# Все ли числа четные?
flag2 = nums.all? {|x| x % 2 == 0 }    # false
```

Если блок не задан, то просто проверяется значение истинности каждого элемента. Иными словами, неявно добавляется блок {|x| x }.

```
flag1 = list.all?    # list не содержит ни одного false или nil
flag1 = list.any?    # list содержит хотя бы одно истинное значение
                     #   не nil и не false)
```

8.3.3. Метод partition

Как говорится: «В мире есть два сорта людей: те, что делят людей по сортам, и те, что не делят». Метод partition относится не к людям (хотя мы можем представить их в Ruby как объекты), но тоже делит коллекцию на две части.

Если при вызове partition задан блок, то этот блок вычисляется для каждого элемента набора. В результате создаются два массива, в первый попадают элементы, для которых блок вернул значение true, во второй – все остальные. Метод возвращает массив, двумя элементами которого являются эти массивы.

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9]

odd_even = nums.partition {|x| x % 2 == 1 }
# [[1,3,5,7,9],[2,3,4,6,8]]
```

```
under5 = nums.partition {|x| x < 5 }
# [[1,2,3,4],[5,6,7,8,9]]

squares = nums.partition { x| Math.sqrt(x).to_i**2 == x }
# [[1,4,9],[2,3,5,6,7,8]]
```

Чтобы разбить коллекцию больше чем на две группы, можно воспользоваться методом group_by, который возвращает хэш, содержащий по одному ключу для каждого результата, порождаемого блоком.

```
nums = [1,2,3,4,5,6,7,8,9]
mod3 = nums.group_by { x| x % 3 }
# { 0 => [3,6,9], 1 => [1,4,7], 2 => [2,5,8] }

words = %w[ area arboreal brick estrous clear
            donor ether filial patina ]
vowels = words.group_by {|x| x.count("aeiou") } # How many vowels?
# {1 => ["brick"], 2 => ["clear", "donor", "ether"],
#  3 => ["area", "estrous", "filial", "patina"], 4 => ["arboreal"]}

initials = words.group_by {|x| x[0..0] } # By first letter
# {"a" => ["area", "arboreal"], "b" => ["brick"],
#  "c" => ["clear"], "d" => ["donor"], "f" => ["filial"],
#  "p" => ["patina"], "e" =>["estrous", "ether"]}
```

8.3.4. Обход с группировкой

До сих пор мы обходили список по одному элементу за раз. Но иногда желательно на каждой итерации анализировать по два, три или более элементов.

Итератор each_slice принимает в качестве параметра число n, равное числу просматриваемых на каждой итерации элементов. Если не осталось достаточного количества элементов, размер последнего фрагмента будет меньше.

```
arr = [1,2,3,4,5,6,7,8,9,10]
arr.each_slice(3) do |triple|
  puts triple.join(",")
end

# Выводится:
# 1,2,3
# 4,5,6
# 7,8,9
# 10
```

Имеется также итератор each_cons, который позволяет обходить набор методом «скользящего окна» заданного размера. (Если название кажется вам странным, знайте, что это наследие языка Lisp. Считайте, что cons – сокращение от consecutive.) В таком случае фрагменты всегда будут иметь одинаковый размер.

```
arr = [1,2,3,4,5,6,7,8,9,10]
arr.each_cons(3) do |triple|
  puts triple.join(",")
```

```
end
# Выводится:
# 1,2,3
# 2,3,4
# 3,4,5
# 4,5,6
# 5,6,7
# 6,7,8
# 7,8,9
# 8,9,10
```

8.3.5. Преобразование в массив или множество

Каждая перечисляемая структура теоретически может быть тривиально преобразована в массив (методом to_a). Например, такое преобразование для хэша дает вложенный массив пар:

```
hash = {1=>2, 3=>4, 5=>6}
arr  =  hash.to_a           #  [[5, 6], [1, 2], [3, 4]]
```

Синонимом to_a является метод entries.

Если была затребована библиотека set, становится доступен также метод to_set. Дополнительную информацию о множествах см. в разделе 9.1.

```
require 'set'
hash = {1=>2, 3=>4, 5=>6}
set = hash.to_set           # #<Set: {[1, 2], [3, 4], [5, 6]}>
```

8.3.6. Перечислители

По существу, перечислитель – это объект, который можно использовать для внешнего или внутреннего итерирования. В случае внутреннего итерирования мы просто перебираем все объекты коллекции и выполняем для каждого некоторый блок. Внешнее итерирование означает, что программа может получить следующий элемент последовательности «по запросу».

Чтобы лучше понять различия между внутренним и внешним итерированием, можно взять метод getline – в некотором смысле он предоставляет внешний итератор для объекта IO. Вы его вызываете, а он возвращает данные. Сравните с внутренним итератором each_line, который просто передает строки одну за другой блоку кода.

Иногда внутренние итераторы не подходят для стоящей задачи. Решение с их помощью, конечно, можно найти, но не всегда оно удобно. Бывает, что внешний итератор оказывается удобнее.

Итераторный метод each не требует задания блока. Если блок не задан, то метод возвращает объект-перечислитель:

```
items = [1,2,3,4]
enum = items.each
enum.each {|x| puts x }     # Печатает числа по одному в строке
```

Очевидно, что в этом примере преобразование в перечислитель – пустая трата сил. Но вот вам пример внешнего итерирования. Допустим, имеется массив, содер-

жащий числа и имена. Каждое число равно количеству слов (отдельных элементов массива) в следующем за ним имени. Мы хотим извлечь и напечатать имена. Внутренний итератор здесь был бы неудобен, но ниже показано простое решение:

```
people = [2, "George ", "Washington",
          3, "Edgar ", "Allan ", "Poe",
          2, "John ", "Glenn",
          4, "Little ", "Red ", "Riding ", "Hood",
          1, "Sting"]
enum = people.each
loop do
      count = enum.next          # Получить следующий элемент из массива
      count.times { print enum.next }
      puts
end
```

Здесь мы извлекаем элемент из массива методом `next`. Для простоты я включил в имена пробелы и опустил проверку ошибок.

Наверное, вы недоумеваете, когда завершается этот кажущийся бесконечным цикл. «Волшебство» состоит в том, что при попытке получить несуществующее значение не только возвращается `nil`, но происходит и еще кое-что. Перечислитель возбуждает исключение `StopIteration`, которое неявно перехватывается. Если это происходит вне цикла, то перехватывать его нужно явно. В противном случае программа завершится с сообщением об этом исключении.

Методы `each_slice` и `each_cons` также возвращают объекты-перечислители:

```
deck = (1..52).sort_by { rand }                # перетасовать
dealer = deck.each_slice(5)
hands = []
4.times do |i|
      hands << dealer.next }                   # человек сдает карты не так
end

sequence = %w[In Xanadu did Kubla Khan
              a stately pleasure dome decree]
search_term = %w[stately pleasure dome]
enum = sequence.each_cons(search_term.length)
index = 0

loop do
      triplet = enum.next
      break if triplet == search_term
      index += 1
end

if index < sequence.length - 1
      puts "Искомый терм найден в позиции #{index}"
else
      puts "Искомый терм не найден"
end
```

Существует метод rewind, который «сбрасывает» внутреннее состояние, возвращаясь к началу перечисляемой последовательности:

```
list = [10, 20, 30, 40, 50]
enum = list.each
puts enum.next    # 10
puts enum.next    # 20
puts enum.next    # 30
enum.rewind
puts enum.next    # 10
```

Метод with_index является простым (внутренним) итератором. Его можно использовать вместе с другим перечислителем, и возвращает он итератор:

```
list = [10, 20, 30, 40, 50]
list.each.with_index {|x,i| puts "list[#{i}] = #{x}"} }
# или...
enum = list.each.with_index
loop { x, i = enum2.next; puts "list[#{i}] = #{x}" }   # Результат тот же
```

8.4. Дополнительные сведения о перечисляемых объектах

В модуле Enumerable есть много других методов, о некоторых из них я расскажу ниже. Для удобства я разбил их, несколько произвольно, на четыре категории: поиск и выборка, подсчет и сравнение, итерирование, а также извлечение и преобразование.

8.4.1. Поиск и выборка

Помимо уже рассмотренных методов select и reject, существует метод find_index – «обобщенный» вариант индекса массива (на самом деле, для массивов оба метода возвращают один и тот же результат). Этот метод ищет первый объект, равный переданному параметру, и возвращает его индекс (индексация начинается с нуля). Например:

```
array = [10, 20, 30, 40, 50, 30, 20]
location = array.find_index(30)          # результат равен 2
```

Не для каждого перечисляемого объекта можно вернуть осмысленный результат (например, для множества, которое не упорядочено, это невозможно). В таком случае возвращается nil.

Методы first и last возвращают *n* первых или последних элементов коллекции (по умолчанию один):

```
array = [14, 17, 23, 25, 29, 35]
head = array.first           # 14
tail = array.last            # 35
front = array.first(2)       # [14, 17]
back = array.last(3)         # [25, 29, 35]
```

Существуют также два квантора: one? и none?. Метод one? понять легко: чтобы он вернул true, вычисление блока кода должно дать результат true ровно один раз:

```
array = [1, 3, 7, 10, 15, 17, 21]
array.one? {|x| x % 2 == 0 }   # true (одно четное число)
array.one? {|x| x > 16 }       # false
[].one? {|x| true }            # для пустого массива всегда возвращается false
```

Но вот none? не столь интуитивно очевиден. Он возвращает true, если при вычислении блока *ни разу* не получилось true.

```
array = [1, 3, 7, 10, 15, 17, 21]
array.none? {|x| x > 50 }      # true (значение блока равно false для всех элементов)
array.none? {|x| x == 1 }      # false
[].none? {|x| true }           # true (для пустого массива блок не выполняется)
```

Если блок опущен, то для каждого элемента коллекции проверяется, похож ли он на true или на false. Метод возвращает true, только если все проверки дают false.

8.4.2. Подсчет и сравнение

С методом count все просто. Он может принимать параметр, блок или ничего. В последнем случае просто возвращается размер коллекции. Если параметром является объект, то подсчитывается количество вхождений этого объекта (сравнение производится с помощью оператора ==). Если задан блок, то подсчитывается количество элементов коллекции, для которых блок возвращает true.

```
days = %w[Sunday Monday Tuesday Wednesday Thursday Friday Saturday]
days.count                      # всего 7
days.count("Saturday")          # 1 (суббота только одна!)
days.count {|x| x.length == 6 } # есть три 3 слова из шести букв
```

С методами min и max мы уже встречались. Для их работы необходимо, чтобы существовал оператор <=> (или был включен модуль Comparable). Существует также метод minmax, который возвращает сразу и минимальный, и максимальный элемент массива:

```
days = %w[Sunday Monday Tuesday Wednesday Thursday Friday Saturday]
days.minmax # ["Friday", "Wednesday"]
```

Если для сравнения элементов нужно использовать более сложное правило, то можно прибегнуть к методам min_by, max_by и minmax_by, которые принимают произвольный блок кода (по аналогии с методом sort_by, рассмотренным выше в этой главе):

```
days = %w[Sunday Monday Tuesday Wednesday Thursday Friday Saturday]
days.min_by {|x| x.length }     # "Sunday" (хотя есть и другие)
days.max_by {|x| x.length }     # "Wednesday"
days.minmax_by {|x| x.reverse } # ["Friday", "Thursday"]
```

В последнем примере результат содержит первый и последний элементы массива, буквы которых переставлены «задом наперед».

8.4.3. Итерирование

Мы уже видели почти все виды итерирования. Еще не встречался только метод cycle, который умеет обходить коллекцию более одного раза или даже «беско-

нечно». Параметр задает число циклов, по умолчанию подразумевается бесконечное число.

```
months = %w[Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec]
months.cycle(2) {|m| puts m }   # всего будет 24 итерации 24
months.cycle {|m| puts m }      # бесконечный цикл
```

Метод each_with_object работает так же, как inject, но в виде дополнительной любезности возвращает тот же объект, который был ему передан. Так удается избежать некрасивого решения, когда приходится явно возвращать аккумулятор в конце блока (пример предложил Дэвид Алан Блэк).

```
h = Hash.new(0)
result = [1,2,3,4].inject(h) {|acc,n| acc[n] += 1; acc }   # Некрасиво

h = Hash.new(0)
result = [1,2,3,4].each_with_object(h) do |n, acc|
      acc[n] += 1 }                                          # Так лучше
end
```

8.4.4. Извлечение и преобразование

Все элементы любой перечисляемой коллекции можно извлечь в массив с помощью метода to_a или entries, это мы уже видели. Существуют также способы извлекать в массив только части коллекции.

В качестве примера рассмотрим метод take и родственный ему метод take_while. Они возвращают список первых элементов коллекции. В первом примере ниже рассматривается хэш, элементом такой коллекции является пара ключ-значение, возвращаемая в виде подмассива. В следующих примерах те же операции применяются к массивам:

```
hash = {1 => 2, 3 => 6, 4 => 8, 5 => 10, 7 => 14}
arr1 = hash.take(2)                     # [[1,2], [3,6]]
arr2 = hash.take_while {|k,v| v <= 8 }  # [[1,2], [3,6], [4,8]]
arr3 = arr1.take(1)                     # [[1,2]]
arr4 = arr2.take_while {|x| x[0] < 4 }  # [[1,2], [3,6]]
```

Метод drop дополняет take. Он пропускает первые элементы коллекции и возвращает остальные. Существует также метод drop_while.

```
hash = {1 => 2, 3 => 6, 4 => 8, 5 => 10, 7 => 14}
arr1 = hash.drop(2)                     # [[4,8], [5,10], [7 => 14]]
arr2 = hash.take_while {|k,v| v <= 8 }  # [[5,10], [7 => 14]]
```

Метод reduce также навеян теми же идеями, что inject. Он применяет бинарную операцию (заданную символом) к каждой паре элементов коллекции, но может принять также и блок. Если задано начальное значение аккумулятора, оно и используется, в противном случае начальным значением будет первый элемент коллекции. Ниже показаны основные варианты применения:

```
range = 3..6
# символ
range.reduce(:*)                        # 3*4*5*6 = 360
```

```
# с начальным значением, символ
range.reduce(2, :*)                          # 2*3*4*5*6 = 720
# с начальным значением, блок
range.reduce(10) {|acc, item| acc += item }  # 10+3+4+5+6 = 28
# блок
range.reduce {|acc, item| acc += item }      # 3+4+5+6 = 28
```

Отметим, что задать одновременно символ бинарного оператора и блок невозможно.

Перечисляемые объекты можно также преобразовывать в формат JSON (если затребована библиотека `json`) или в множество (если затребована библиотека `set`):

```
require 'set'
require 'json'

array = [3,4,5,6]
p array.to_set     # #<Set: {3, 4, 5, 6}>
p array.to_json    # "[3,4,5,6]"
```

8.4.5. Ленивые перечислители

Любой объект `Enumerable`, рассмотренный в этой главе (а равно и другие, например, `Range`), допускает вызов методов итерирования без блока. В таком случае возвращается объект `Enumerator`. Позже этот объект можно использовать для итерирования самостоятельно или в сочетании с другими перечислителями, например `with_index`, для реализации дополнительной функциональности (мы видели это в разделе 8.1.18).

Метод `lazy` объекта `Enumerable` возвращает специальный объект `Enumerator`, который вычисляет следующий элемент по мере необходимости. Это позволяет обходить группы, слишком большие для хранения в памяти, например, все нечетные числа от 1 до бесконечности:

```
enum = (1..Float::INFINITY).each      # Enumerator
lazy = enum.lazy                      # LazyEnumerator по всем целым числам
odds = lazy.select(&:odd?)            # LazyEnumerator по нечетным числам
odds.first(5)                         # [1, 3, 5, 7, 9]
odds.next                             # 1
odds.next                             # 3
```

Ленивые перечислители открывают новые возможности для экономии памяти и времени при обходе больших коллекций, поэтому я рекомендую внимательно ознакомиться с документацией по классу `LazyEnumerator`.

8.5. Заключение

Мы подробно рассмотрели массивы, хэши и перечисляемые структуры в общем. Мы видели, что между массивами и хэшами есть определенное сходство, объясняемое тем, что в оба класса подмешан модуль `Enumerable`. Но есть и различия. Мы показали, как преобразовать массив в хэш и наоборот, и познакомились с несколькими интересными способами расширить стандартное поведение.

Мы изучили различные методы обхода структур, например `each_slice` и `each_cons`, а также узнали, как работают перечислители.

В главе 9 мы продолжим изучение высокоуровневых структур данных. Не все они входят в ядро Ruby или в стандартные библиотеки. Речь пойдет о множествах, стеках, очередях, деревьях и графах.

Глава 9. Более сложные структуры данных

Графическое представление данных
абстрагирует банки памяти любого компьютера.
Невообразимая сложность. Лучи света,
протянувшиеся в не-пространстве разума,
скопления и созвездия данных.
Как гаснущие огни большого города.

– Вильям Гибсон

Есть, конечно, более сложные и интересные структуры данных, чем массивы и хэши. Некоторые из тех, с которыми мы познакомимся в этой главе, имеют прямую или косвенную поддержку в Ruby, другие приходится программировать самостоятельно. К счастью, Ruby упрощает создание нестандартных структур данных.

Математические множества можно, как мы видели, моделировать с помощью массивов. Но в последних версиях Ruby есть также класс `Set`, который хорошо поддерживает эту структуру.

Стеки и очереди – две весьма распространенные в информатике структуры данных. В первом издании этой книги им было уделено чрезмерно много внимания. Для тех, кого интересуют общие вопросы, я оставил кое-какой материал, для остальных есть немало великолепных книг по структурам данных и алгоритмам.

Деревья полезны для сортировки, поиска и просто представления иерархических данных. Мы рассмотрим двоичные деревья и сделаем несколько замечаний о деревьях более высокой степени.

Граф – это обобщение понятия дерева. Граф представляет собой множество вершин, соединенных ребрами, причем с каждым ребром может быть связан вес или направление. Они полезны для решения многих задач, в том числе при анализе сетей и организации знаний.

Но самыми простыми структурами являются множества. С них мы и начнем.

9.1. Множества

Мы уже видели, что некоторые методы класса `Array` позволяют использовать массивы для представления математических множеств. Однако для написания более строгого и компактного кода в Ruby есть также класс `Set`, который скрывает от программиста большую часть деталей реализации.

Чтобы получить в свое распоряжение класс Set, достаточно написать:

```
require 'set'
```

При этом также добавляется метод to_set в модуль Enumerable, так что любой перечисляемый объект становится возможно преобразовать в множество.

Создать новое множество нетрудно. Метод [] работает почти так же, как для хэшей. Метод new принимает в качестве необязательных параметров перечисляемый объект и блок. Если блок задан, то он выступает в роли «препроцессора» для списка (подобно операции map).

```
s1 = Set[3,4,5]                  # в математике обозначается {3,4,5}
arr = [3,4,5]
s2 = Set.new(arr)                # то же самое
s3 = Set.new(arr) {|x| x.to_s }  # множество строк, а не чисел
```

9.1.1. Простые операции над множествами

Для объединения множеств служит метод union (синонимы | и +):

```
x = Set[1,2,3]
y = Set[3,4,5]

a = x.union(y)           # Set[1,2,3,4,5]
b = x | y                # то же самое
c = x + y                # то же самое
```

Пересечение множеств вычисляется методом intersection (синоним &):

```
x = Set[1,2,3]
y = Set[3,4,5]

a = x.intersection(y)    # Set[3]
b = x & y                # то же самое
```

В случае бинарных операторов в правой части необязательно должно быть множество. Подойдет любой перечисляемый объект, порождающий множество в качестве результата.

Унарный минус обозначает разность множеств, мы обсуждали эту операцию в разделе 8.1.9.

```
diff = Set[1,2,3] - Set[3,4,5]    # Set[1,2]
```

Принадлежность элемента множеству проверяют методы member? или include?, как для массивов. Напомним, что порядок операндов противоположен принятому в математике.

```
Set[1,2,3].include?(2)    # true
Set[1,2,3].include?(4)    # false
Set[1,2,3].member?(4)     # false
```

Чтобы проверить, является ли множество пустым, мы вызываем метод empty?, как и в случае массивов. Метод clear очищает множество, то есть удаляет из него все элементы.

```
s = Set[1,2,3,4,5,6]
s.empty?              # false
```

```
s.clear
s.empty?              # true
```

Можно проверить, является ли одно множество подмножеством, собственным подмножеством или надмножеством другого.

```
x = Set[3,4,5]
y = Set[3,4]

x.subset?(y)          # false
y.subset?(x)          # true
y.proper_subset?(x)   # true
x.subset?(x)          # true
x.proper_subset?(x)   # false
x.superset?(y)        # true
```

Метод add (синоним <<) добавляет в множество один элемент и обычно возвращает его в качестве значения. Метод add? возвращает nil, если такой элемент уже присутствовал в множестве. Метод merge полезен, если надо добавить сразу несколько элементов. Все они, конечно, могут изменить состояние вызывающего объекта. Метод replace работает так же, как в случае строки или массива.

Наконец, два множества можно сравнить на равенство интуитивно очевидным способом:

```
Set[3,4,5] == Set[5,4,3]    # true
```

9.1.2. Более сложные операции над множествами

Разумеется, можно обойти множество, но (как и для хэшей) не ожидайте какого-то определенного порядка появления элементов, потому что множества по сути своей неупорядочены, и Ruby не гарантирует никакой последовательности. (Временами можно получить повторяющиеся, ожидаемые результаты, но полагаться на это неразумно.)

```
s = Set[1,2,3,4,5]
s.each {|x| puts x; break }    # Выводится: 5
```

Метод classify подобен методу partition, но с разбиением на несколько частей; он послужил источником идеи для реализации нашей версии метода classify в разделе 8.3.3.

```
files = Set.new(Dir["*"])
hash = files.classify do |f|
  if File.size(f) <= 10_000
    :small
  elsif File.size(f) <= 10_000_000
    :medium
  else
    :large
  end
end

big_files = hash[:large]    # big_files - это Set
```

Метод divide аналогичен, но вызывает блок, чтобы выяснить «степень общности» элементов, и возвращает множество, состоящее из множеств.

Если арность (число аргументов) блока равна 1, то метод выполняет вызовы вида block.call(a) == block.call(b), чтобы определить, принадлежат ли a и b одному подмножеству. Если арность равна 2, то для той же цели выполняются вызовы вида block.call(a,b).

Например, следующий блок (с арностью 1) разбивает множество на два подмножества, одно из которых содержит четные числа, а другое – нечетные:

```
require 'set'
numbers = Set[1,2,3,4,5,6,7,8,9,0]
set = numbers.divide{|i| i % 2}
p set #  #<Set: {#<Set: {5, 1, 7, 3, 9}>,  #<Set: {0, 6, 2, 8, 4}>}>
```

Вот еще один, несколько искусственный пример. Простыми числами-близнецами называются простые числа, отличающиеся на 2 (например, 11 и 13); все остальные называются одиночными (например, 23). Следующий код разбивает множество на группы, помещая числа-близнецы в одно и то же подмножество. В данном случае применяется блок с арностью 2:

```
primes = Set[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
set = primes.divide{|i,j| (i-j).abs == 2}
# set is: #<Set: {#<Set: {23}>, #<Set: {11, 13}>,
#          #<Set: {17, 19}>,     #<Set: {5, 7, 3}>,
#          #<Set: {2}>, #<Set: {29, 31}>}>
# Более компактно: {{23},{11,13},{17,19},{5,7,3},{2},{29,31}}
```

Как я уже говорил, важно понимать, что класс Set не всегда требует, чтобы параметр или операнд также был множеством (вспомните обсуждение динамической типизации в главе 1, если вам это кажется странным). На самом деле, большая часть методов этого класса принимает в качестве параметра *любой перечисляемый объект*. Считайте, что так и задумано.

Есть и другие методы, которые применяются в частности к множествам (в том числе все методы из модуля Enumerable). Я не стану рассматривать здесь такие методы, как flatten. Дополнительную информацию можно найти на сайте http://ruby-doc.org/ или в любом другом справочном руководстве.

9.2. Стеки и очереди

Стеки и очереди – первые из встретившихся нам структур, которые, строго говоря, не встроены в Ruby. Иными словами, в Ruby нет классов Stack и Queue, в отличие от Array и Hash (впрочем, класс Queue есть в библиотеке thread.rb, которую мы еще будем рассматривать).

И все же в некотором смысле они встроены в Ruby. Ведь класс Array реализует все, что нужно для того, чтобы рассматривать его как стек или очередь. Вскоре мы в этом убедимся.

Хотя использовать Array в качестве стека или очереди возможно, он не является *потокобезопасным*. Это означает, что несколько потоков, одновременно модифи-

цирующих структуру данных, могут повредить данные. Если требуется разделять объекты между потоками, пользуйтесь классом `Queue`, неизменяемыми объектами данных или еще какой-нибудь потокобезопасной структурой данных.

Стек организован по принципу «последним пришел, первым обслужен» (LIFO – last-in first-out). Традиционный пример – стопка подносов на подпружиненной подставке в кафетерии, подносы кладутся сверху и сверху же снимаются.

Над стеком можно выполнять ограниченный набор операций. Как минимум, это операции *заталкивания* (push) и *выталкивания* (pop), то есть помещения в стек и извлечения из него. Обычно также предоставляется способ проверить, пуст ли стек, и исследовать верхний элемент, не извлекая его из стека. Но никогда реализация не позволяет получить доступ к элементу в середине стека.

Как же можно реализовать стек на базе массива, если к элементам массива можно обращаться в произвольном порядке, а стек таким свойством не обладает. Ответ прост. Стек – более абстрактная структура, чем массив. Он является стеком лишь до тех пор, пока мы обращаемся с ним, как с таковым. В тот момент, когда вы пытаетесь обратиться к элементу недопустимым образом, стек перестает быть стеком.

Но несложно определить класс `Stack` так, что к элементам можно будет обращаться только законно. И мы покажем, как это сделать.

Стоит отметить, что во многих алгоритмах стек применяется как основа элегантного рекурсивного решения. Причина станет понятна, если чуточку подумать. При вызове функции или метода параметры заталкиваются в системный стек и выталкиваются из него при возврате. Таким образом, рекурсивный алгоритм просто подменяет явно определенный пользователем стек системным. Что лучше? Зависит от того, какое значение вы придаете понятности программы, ее эффективности и другим аспектам.

Очередь организована по принципу «первым пришел, первым обслужен» (FIFO – first-in first out). Аналогом может служить очередь за билетами в кинотеатр. Вновь подходящие становятся в конец очереди, а те, кто пришел раньше, обслуживаются первыми. В программировании очереди используются реже, чем стеки.

Очереди полезны в системах реального времени, когда события нужно обрабатывать в порядке возникновения. Находят они применение и в ситуации «производитель-потребитель» (особенно в многопоточных программах и многозадачных средах). Неплохой пример – очередь к принтеру; задания на печать помещаются в один конец и ожидают, пока не будут извлечены с другого конца.

Две основные операции над очередью называются *«поместить»* (enqueue) и *«извлечь»* (dequeue). Им соответствуют методы `unpush` и `shift` в классе `Array`.

Отметим, что метод `unshift` может использоваться в сочетании с `shift` при реализации массива, но никак не очереди, поскольку `unshift` добавляет элемент в тот же конец массива, из которого `shift` его удаляет. С помощью различных комбинаций этих методов можно реализовать и стек, и очередь, но рассматривать все возможные сочетания мы не будем.

На этом мы закончим введение в стеки и очереди. Самое время рассмотреть некоторые примеры.

9.2.1. Более строгая реализация стека

Мы обещали показать, как можно сделать стек защищенным от некорректного доступа. Выполняем обещание. Вот пример простого класса, который хранит внутри себя массив и управляет доступом к этому массиву. (Есть и другие способы, например, путем делегирования, но описанная реализация проста и прекрасно работает.)

```
class Stack

  def initialize
    @store = []
  end

  def push(x)
    @store.push x
  end

  def pop
    @store.pop
  end

  def peek
    @store.last
  end

  def empty?
    @store.empty?
  end

end
```

Мы добавили одну операцию, которая для массивов не определена; метод peek возвращает элемент, находящийся на вершине стека, не выталкивая его.

В некоторых примерах ниже предполагается именно такое определение класса.

9.2.2. Обнаружение несбалансированных скобок

В силу самой природы употребления различного вида скобок в выражениях проверить корректность написания можно с помощью стека. При открытии каждого следующего уровня вложенности скобок стек растет. Как только встречается закрывающая скобка, соответствующий элемент выталкивается из стека. Если при обнаружении закрывающей скобки в стеке ничего не оказалось или, наоборот, выражение уже закончилось, а в стеке что-то осталось, значит, выражение записано неверно.

```
def paren_match(str)
  stack = Stack.new
  lsym = "{[(<"
  rsym = "}])>"
  str.each_char do |sym|
    if lsym.include? sym
      stack.push(sym)
    elsif rsym.include? sym
      top = stack.peek
      if lsym.index(top) != rsym.index(sym)
        return false
      else
        stack.pop
      end
      # Игнорируем символы, отличные от скобок...
    end
  end
  # Убедимся, что стек пуст...
  return stack.empty?
end

str1 = "(((a+b))*((c-d)-(e*f))"
str2 = "[[(a-(b-c))], [[x,y]]]"

paren_match str1          # false
paren_match str2          # true
```

Наличие вложенности естественно наводит на мысль о применении стека. Несколько сложнее распознать несбалансированные теги в HTML или XML-документе. Лексемы состоят из нескольких символов, но логическая структура задачи остается той же самой. Вот еще типичные примеры задач, требующих стека: преобразование выражений из инфиксной формы в постфиксную (и наоборот), вычисление постфиксного выражения (как делается в виртуальной машине Java и многих других интерпретаторах) и вообще любая задача, имеющая рекурсивное решение. В следующем разделе мы немного поговорим о связи между стеком и рекурсией.

9.2.3. Стек и рекурсия

В качестве примера изоморфизма, существующего между стеком и рекурсией, рассмотрим классическую задачу о «Ханойской башне».

По легенде где-то далеко на востоке существует старинный храм. Обитающие в нем монахи заняты решением единственной задачи: перемещением дисков с одного шеста на другой с соблюдением определенных правил. Первоначально на первом шесте было 64 диска. Когда все диски будут перемещены, настанет конец света.

Попутно разоблачим миф. Похоже, что на самом деле эту задачу впервые сформулировал французский математик Эдуард Люка в 1883 году, и никаких истоков в восточной культуре она не имеет. Сам Люка называл ее «Ханойской башней».

Так что если вас пугает конец света, можете успокоиться. Да и в любом случае для перемещения 64 дисков потребуется $2^{64}-1$ ходов. Небольшой расчет на калькуляторе покажет, что монахи будут заняты своим делом несколько миллионов лет.

Однако вернемся к правилам игры. (Сформулируем их, хотя эту загадку знал уже самый первый студент самого первого факультета информатики.) Имеется шест, на который надето несколько дисков, назовем его исходным шестом. Мы хотим переместить все диски на целевой шест, используя еще один вспомогательный шест как место промежуточного хранения. Проблема в том, что за один ход можно перемещать только один диск, при этом нельзя класть больший диск на меньший.

В следующем примере приведено решение этой задачи с использованием стека. Мы ограничились тремя дисками, потому что для перемещения 64 компьютеру потребовались бы века.

```
def towers(list)
  while !list.empty?
    n, src, dst, aux = list.pop
    if n == 1
      puts "Перемещаем диск с #{src} на #{dst}"
    else
      list.push [n-1, aux, dst, src]
      list.push [1, src, dst, aux]
      list.push [n-1, src, aux, dst]
    end
  end
end


list = []
list.push([3, "a", "c", "b"])

towers(list)
```

Вот что напечатает эта программа:

```
Перемещаем диск с a на c
Перемещаем диск с a на b
Перемещаем диск с c на b
Перемещаем диск с a на c
Перемещаем диск с b на a
Перемещаем диск с b на c
Перемещаем диск с a на c
```

Конечно, классическое решение этой задачи рекурсивно. Но, как мы отмечали, тесная связь между обоими алгоритмами не должна вызывать удивления, так как для рекурсии применяется невидимый системный стек.

```
def towers(n, src, dst, aux)
  if n==1
    puts "Перемещаем диск с #{src} на #{dst}"
```

```
  else
    towers(n-1, src, aux, dst)
    towers(1, src, dst, aux)
    towers(n-1, aux, dst, src)
  end
end

towers(3, "a", "c", "b")
```

Печатается точно такой же результат. Возможно, вам будет интересно знать, что мы закомментировали предложения, осуществляющие вывод, и сравнили время работы. Никому не говорите, но рекурсивное решение оказалось в два раза быстрее.

9.2.4. Более строгая реализация очереди

Мы определим очередь примерно так же, как стек. Если вы хотите защититься от некорректного доступа к структуре данных, рекомендуем поступать аналогично.

```
class Queue

  def initialize
    @store = []
  end

  def enqueue(x)
    @store << x
  end

  def dequeue
    @store.shift
  end

  def peek
    @store.first
  end

  def length
    @store.length
  end

  def empty?
    @store.empty?
  end

end
```

Отметим, что класс `Queue` имеется в библиотеке `thread` для поддержки многопоточных программ. Имеется даже вариант `SizedQueue` для организации очереди ограниченного размера.

В упомянутых классах методы имеют короткие имена: `enq` и `deq`. У них есть также синонимы `push` и `pop`, что лично мне кажется неоправданным. Это структура данных FIFO, а не LIFO, то есть именно очередь, а не стек.

Если вы хотите реализовать потокобезопасный класс `Stack`, рекомендую взять `Queue` в качестве отправной точки. Потребуется внести не так много изменений.

9.3. Деревья

Я не увижу никогда, наверное,
Поэму столь прекрасную как дерево.

— «Деревья», Джойс Килмер
(перевод Я. Фельдмана)

В информатике идея дерева считается интуитивно очевидной (правда, изображаются они обычно с корнем наверху, а листьями снизу). Объясняется это тем, что в повседневной жизни мы постоянно сталкиваемся с иерархическими данными: генеалогическое древо, организационная схема компании, структура каталогов на диске.

Терминология, описывающая деревья, богата, но понять ее легко. Элементы дерева называются *узлами*, верхний или самый первый узел называется *корневым* или *корнем*. У узла могут быть *потомки*, расположенные ниже него, а непосредственные потомки называются *детьми* или *дочерними узлами*. Узел, не имеющий потомков, называется *листовым* или просто *листом*. *Поддерево* состоит из некоторого узла и всех его потомков. Посещение всех узлов дерева (например, с целью распечатки) называется *обходом* дерева.

Нас будут интересовать в основном двоичные деревья, хотя в принципе узел может иметь произвольное число детей. Мы покажем, как создавать дерево, добавлять в него узлы и выполнять обход. Рассмотрим также некоторые реальные задачи, при решении которых используются деревья.

Отметим, что во многих языках, например в C или Pascal, деревья реализуются с помощью адресных указателей. Но в Ruby (как и в Java) указателей нет, вместо них используются ссылки на объекты, что ничуть не хуже, а иногда даже лучше.

9.3.1. Реализация двоичного дерева

В Ruby двоичное дерево можно реализовать несколькими способами. Например, хранить значения узлов можно в массиве. Но мы применим более традиционный подход, характерный для кодирования на C, только указатели заменим ссылками на объекты.

Что нужно для описания двоичного дерева? Понятно, что в каждом узле должен быть атрибут для хранения данных. Кроме того, в каждом узле должны быть атрибуты для ссылки на левое и правое поддерево.

Еще необходим способ вставить новый узел в дерево и получить хранящуюся в дереве информацию. Для этого нам потребуется два метода.

В первом дереве, которое мы рассмотрим, эти методы будут реализованы неортодоксальным способом. Позже мы расширим класс `Tree`.

В некотором смысле дерево определяется алгоритмом вставки и способом обхода. В нашем первом примере (листинг 9.1) метод `insert` будет осуществлять поиск в дереве *«в ширину»*, то есть сверху вниз и слева направо. При этом глубина дерева растет относительно медленно, и дерево всегда оказывается сбалансированным. Методу *вставки* соответствует итератор `traverse`, который обходит дерево в том же порядке.

Листинг 9.1. Вставка и обход дерева в ширину

```
class Tree

attr_accessor :left
attr_accessor :right
attr_accessor :data

def initialize(x=nil)
  @left = nil
  @right = nil
  @data = x
end

def insert(x)
  list = []
  if @data == nil
    @data = x
  elsif @left == nil
    @left = Tree.new(x)
  elsif @right == nil
    @right = Tree.new(x)
  else
    list << @left
    list << @right
    loop do
      node = list.shift
      if node.left == nil
        node.insert(x)
        break
      else
        list << node.left
      end
      if node.right == nil
        node.insert(x)
        break
      else
        list << node.right
```

```
      end
    end
  end
end

def traverse()
  list = []
  yield @data
  list << @left if @left != nil
  list << @right if @right != nil
  loop do
    break if list.empty?
    node = list.shift
    yield node.data
    list << node.left if node.left != nil
    list << node.right if node.right != nil
  end
end

end


items = [1, 2, 3, 4, 5, 6, 7]

tree = Tree.new

items.each {|x| tree.insert(x)}

tree.traverse {|x| print "#{x} "}
print "\n"

# Печатается "1 2 3 4 5 6 7 "
```

Такое дерево не слишком интересно. Но оно годится в качестве введения и фундамента, на котором можно возводить здание.

9.3.2. Сортировка с помощью двоичного дерева

Двоичное дерево позволяет эффективно реализовать сортировку произвольных данных. (Правда, если данные уже отсортированы, оно вырождается в обычный связанный список.) Понятно, что причина в том, что при каждом сравнении мы исключаем половину мест, в которые можно поместить новый узел.

Хотя в настоящее время такой способ сортировки применяется редко, знать о нем не повредит. Код в листинге 9.2 основан на предыдущем примере.

Листинг 9.2. Сортировка с помощью двоичного дерева

```
class Tree

  # Предполагается, что определения
  # взяты из предыдущего примера...
```

```
  def insert(x)
    if @data == nil
      @data = x
    elsif x <= @data
      if @left == nil
        @left = Tree.new x
      else
        @left.insert x
      end
    else
      if @right == nil
        @right = Tree.new x
      else
        @right.insert x
      end
    end
  end

  def inorder()
    @left.inorder {|y| yield y} if @left != nil
    yield @data
    @right.inorder {|y| yield y} if @right != nil
  end

  def preorder()
    yield @data
    @left.preorder {|y| yield y} if @left != nil
    @right.preorder {|y| yield y} if @right != nil
  end

  def postorder()
    @left.postorder {|y| yield y} if @left != nil
    @right.postorder {|y| yield y} if @right != nil
    yield @data
  end
end

items = [50, 20, 80, 10, 30, 70, 90, 5, 14,
         28, 41, 66, 75, 88, 96]

tree = Tree.new

items.each {|x| tree.insert(x)}

tree.inorder {|x| print x, " "}
print "\n"
tree.preorder {|x| print x, " "}
print "\n"
tree.postorder {|x| print x, " "}
```

```
print "\n"

# Печатается:
# 5 10 14 20 28 30 41 50 66 70 75 80 88 90 96
# 50 20 10 5 14 30 28 41 80 70 66 75 90 88 96
# 5 14 10 28 41 30 20 66 75 70 88 96 90 80 50
```

9.3.3. Использование двоичного дерева как справочной таблицы

Пусть дерево уже отсортировано. Тогда оно может служить прекрасной справочной таблицей; например, для поиска в сбалансированном дереве, содержащем миллион узлов, понадобится не более 20 сравнений (глубина дерева равна логарифму числа узлов по основанию 2). Чтобы поиск был осмысленным, предположим, что в каждом узле хранится не одно какое-то значение, а ключ и ассоциированные с ним данные.

Почти всегда лучше использовать в качестве справочной таблицы хэш или даже таблицу во внешней базе данных. Но реализовать такой поиск очень просто:

```
class Tree

  # Предполагается, что определения
  # взяты из предыдущего примера...

  def search(x)
    if self.data == x
      return self
    elsif x < self.data
      return left ? left.search(x) : nil
    else
      return right ? right.search(x) : nil
    end
  end

end

keys = [50, 20, 80, 10, 30, 70, 90, 5, 14,
        28, 41, 66, 75, 88, 96]

tree = Tree.new

keys.each {|x| tree.insert(x)}

s1 = tree.search(75)   # Возвращает ссылку на узел, содержащий 75...

s2 = tree.search(100)  # Возвращает nil (не найдено)
```

9.3.4. Преобразование дерева в строку или массив

С помощью тех же приемов, которые применяются для обхода дерева, мы можем преобразовать его в строку или в массив. Ниже мы выполняем обход во внутреннем порядке, хотя подошел бы и любой другой способ:

```
  # Предполагается, что определения
  # взяты из предыдущего примера...

  def to_s
    str = "["
    str << left.to_s << ", " if left
    str << data.inspect
    str << ", " << right.to_s if right
    str << "]"
  end

  def to_a
    array = []
    array << left.to_a if left
    array << data
    array << right.to_a if right
    array
  end

end

items = %w[bongo grimace monoid jewel plover nexus synergy]

tree = Tree.new
items.each {|x| tree.insert x}

str = tree.to_a * ","
# str is now "bongo,grimace,jewel,monoid,nexus,plover,synergy"
arr = tree.to_a
# arr равно:
# ["bongo",["grimace",[["jewel"],"monoid",[["nexus"],"plover",
#   ["synergy"]]]]]
```

Отметим, что глубина вложенности получающегося массива равна глубине дерева с корнем в том узле, с которого мы начали обход. Чтобы получить плоский массив, можете воспользоваться методом flatten.

9.4. Графы

Графом называется множество вершин, произвольным образом соединенных друг с другом. (Дерево – частный случай графа.) Не будем слишком углубляться в эту тему, поскольку теория и терминология весьма сложны. Очень скоро мы перешли бы от информатики в область чистой математики.

И все же у графов есть немало практических приложений. Возьмите обычную дорожную карту, на которой города соединены скоростными магистралями, или печатную плату. То и другое удобно представлять в виде графов. Компьютерную

сеть тоже можно описать в терминах теории графов, будь то локальная сеть из нескольких десятков машин или весь Интернет, насчитывающий миллионы узлов.

Под *графом* мы обычно понимаем неориентированный граф. Попросту говоря, это граф, в котором на соединительных линиях не проставлены стрелки; две вершины либо соединены, либо нет. Напротив, в *ориентированном графе* (*орграфе*) могут быть «улицы с односторонним движением»; из того, что вершина x соединена с вершиной y, не следует, что и обратное тоже верно. Наконец, во взвешенном графе ребрам можно назначать веса. Например, вес может выражать «расстояние» между вершинами. Мы ограничимся только этими основными видами графов; интересующегося читателя отсылаем к многочисленным учебникам информатики и математики.

В Ruby, как и во многих других языках, граф можно представить разными способами, например, в виде настоящей сети взаимосвязанных объектов или в виде матрицы, в которой хранятся ребра графа. Мы рассмотрим оба способа и на примерах покажем, как можно манипулировать графами.

9.4.1. Реализация графа в виде матрицы смежности

Этот пример основан на двух предыдущих. В листинге 9.3. неориентированный граф реализован в виде матрицы смежности с помощью класса `ZeroArray` (см. раздел 8.1.26). Это нужно для того, чтобы новые элементы по умолчанию получали значение 0. Кроме того, мы унаследовали классу `TriMatrix` (см. раздел 8.1.7), чтобы получить *нижнетреугольную матрицу*.

Листинг 9.3. Матрица смежности

```
class LowerMatrix < TriMatrix

  def initialize
    @store = ZArray.new
  end

end


class Graph

  def initialize(*edges)
    @store = LowerMatrix.new
    @max = 0
    for e in edges
      e[0], e[1] = e[1], e[0] if e[1] > e[0]
      @store[e[0],e[1]] = 1
      @max = [@max, e[0], e[1]].max
    end
  end

  def [](x,y)
```

```
  if x > y
    @store[x,y]
  elsif x < y
    @store[y,x]
  else
    0
  end
end

def []=(x,y,v)
  if x > y
    @store[x,y]=v
  elsif x < y
    @store[y,x]=v
  else
    0
  end
end

def edge? x,y
  x,y = y,x if x < y
  @store[x,y]==1
end

def add x,y
  @store[x,y] = 1
end

def remove x,y
  x,y = y,x if x < y
  @store[x,y] = 0
  if (degree @max) == 0
    @max -= 1
  end
end

def vmax
  @max
end

def degree x
  sum = 0
  0.upto @max do |i|
    sum += self[x,i]
  end
  sum
end

def each_vertex
  (0..@max).each {|v| yield v}
```

```
  end

  def each_edge
    for v0 in 0..@max
      for v1 in 0..v0-1
= 317 =
        yield v0,v1 if self[v0,v1]==1
      end
    end
  end

end
mygraph = Graph.new([1,0],[0,3],[2,1],[3,1],[3,2])

# Напечатать степени всех вершин: 2 3 2 3
mygraph.each_vertex {|v| puts mygraph.degree(v)}

# Напечатать список ребер
mygraph.each_edge do |a,b|
  puts "(#{a},#{b})"
end

# Удалить одно ребро
mygraph.remove 1,3

# Напечатать степени всех вершин: 2 2 2 2
mygraph.each_vertex {|v| p mygraph.degree v}
```

Отметим, что приведенная выше реализация не допускает ребер, ведущих из некоторого узла в него же. Кроме того, два узла могут быть соединены только одним ребром.

Мы позволяем задать начальный состав ребер, передавая пары в конструктор. Кроме того, можно добавлять и удалять ребра, а также проверять наличие ребра между двумя вершинами. Метод `vmax` возвращает вершину с наибольшим номером. Метод `degree` вычисляет степень указанной вершины, то есть количество исходящих из нее ребер.

Наконец, имеются два итератора `each_vertex` и `each_edge`, которые позволяют перебрать все вершины и все ребра соответственно.

9.4.2. Является ли граф связным?

Не все графы связные. Иногда нет способа «добраться из одной точки в другую», то есть между двумя вершинами нет никакого пути, составленного из ребер. Связность – это важное свойство графа, его надо уметь вычислять. В связном графе любая вершина достижима из любой другой.

Не будем объяснять принцип работы алгоритма, интересующийся читатель может найти описание в любой книге по дискретной математике. Но в листинге 9.4 приведена его реализация на Ruby.

Листинг 9.4. Вычисление связности графа

```
class Graph

  def connected?
    x = vmax
    k = [x]
    l = [x]
    for i in 0..@max
      l << i if self[x,i]==1
    end
    while !k.empty?
      y = k.shift
      # Теперь ищем все ребра (y,z)
      self.each_edge do |a,b|
        if a==y || b==y
          z = a==y ? b : a
          if !l.include? z
            l << z
            k << z
          end
        end
      end
    end
    if l.size < @max
      false
    else
      true
    end
  end

end


mygraph = Graph.new([0,1], [1,2], [2,3], [3,0], [1,3])

puts mygraph.connected?     # true

puts mygraph.euler_path?    # true

mygraph.remove 1,2
mygraph.remove 0,3
mygraph.remove 1,3

puts mygraph.connected?     # false

puts mygraph.euler_path?    # false
```

В примере упомянут метод euler_path?, с которым мы еще не встречались. Он

определен в разделе 9.4.4.

Можно было бы усовершенствовать этот алгоритм, так чтобы он находил все связные компоненты несвязного графа. Но мы не станем этого делать.

9.4.3. Есть ли в графе эйлеров цикл?

Нет такой отрасли математики,
сколь угодно абстрактной, которая со временем
не нашла бы применения в реальной жизни.

— Николай Лобачевский

Иногда нужно знать, есть ли в графе *эйлеров цикл*. Термин связан с математиком Леонардом Эйлером, который основал область топологии, занимающуюся этим вопросом. (Графы, обладающие таким свойством, называют иногда уникурсивными, поскольку их можно нарисовать, не отрывая карандаша от бумаги и не проходя дважды по одному и тому же ребру.)

В немецком городе Кенигсберг был остров посередине реки. С двумя берегами остров связывало семь мостов. Горожане хотели знать, можно ли обойти город так, чтобы побывать на каждом мосту ровно один раз и вернуться в исходную точку. В 1735 году Эйлер доказал, что это невозможно. Эта классическая задача стала первой проблемой теории графов.

Как часто бывает в жизни, решение кажется простым, когда оно найдено. Оказалось, что для существования в графе эйлерова цикла, необходимо и достаточно, чтобы все вершины имели *четную степень*. Вот короткий код, проверяющий выполнение этого свойства:

```
class Graph

  def euler_circuit?
    return false if !connected?
    for i in 0..@max
      return false if degree(i) % 2 != 0
    end
    true
  end

end

mygraph = Graph.new([1,0],[0,3],[2,1],[3,1],[3,2])

flag1 =  mygraph.euler_circuit?    # false

mygraph.remove 1,3

flag2 =  mygraph.euler_circuit?    # true
```

9.4.4. Есть ли в графе эйлеров путь?

Эйлеров путь и эйлеров цикл – разные вещи. Слово «цикл» подразумевает, что нужно вернуться в исходную точку. А наличие пути предполагает, что нужно лишь посетить каждую вершину ровно один раз. В следующем фрагменте демонстрируется это различие:

```
class Graph

  def euler_path?
    return false if !connected?
    odd=0
    each_vertex do |x|
      if degree(x) % 2 == 1
        odd += 1
      end
    end
    odd <= 2
  end

end


mygraph = Graph.new([0,1],[1,2],[1,3],[2,3],[3,0])

flag1 =  mygraph.euler_circuit?    # false
flag2 =  mygraph.euler_path?       # true
```

9.4.5. Инструменты для работы с графами в Ruby

В сообществе пользователей Ruby известно несколько таких инструментов. Самые популярные предназначены для создания графов на веб-сайтах с помощью JavaScript, но некоторые умеют создавать и изображения. К наиболее известным gem-пакетам для работы с графами на чистом Ruby относятся `gruff`, `gnuplot` и `rubyvis`. Еще одна возможность – создать документ с помощью gem-пакета rubygraphviz, а затем воспользоваться пакетом GraphViz для представления этого документа в виде изображения или Postscript-файла, допускающего печать.

Короче говоря, потребность в подобных инструментах может возникнуть. В таком случае я призываю вас написать собственный инструмент или присоединиться к какому-нибудь существующему проекту. Если работать с графами станет достаточно просто, то мы еще будет недоумевать, как могли жить без них.

9.5. Заключение

Мы познакомились с классом `Set` в Ruby, а также с несколькими примерами «доморощенных» структур данных. Мы видели, как можно создавать сложные структуры данных путем наследования существующему классу или ограниченного делегирования, когда экземпляр существующего класса инкапсулируется в новой

структуре. Мы также рассмотрели изобретательные способы хранения данных, применения различных структур данных и создания итераторов для обхода таких структур.

Мы уделили внимание стекам и очередям и способам их использования для решения задач. Также мы бросили беглый взгляд на деревья и графы.

В следующей главе мы снова займемся манипулированием данными. Но если до сих пор нас интересовало хранения объектов в основной памяти, то теперь мы обратимся к вспомогательной памяти, то есть файлам (и вводом-выводом в общем), базам данных и устойчивым объектам.

Глава 10. Ввод-вывод и хранение данных

На чистом диске можно искать бесконечно.

— Томас Б. Стил младший

Вычислительные машины хороши для вычислений. В этой тавтологии больше смысла, чем кажется на первый взгляд. Если бы программа только потребляла процессорное время да изредка обращалась к оперативной памяти, жизнь была бы куда проще.

Но от компьютера, занятого исключительно собой, мало толку. Рано или поздно придется получать информацию извне и отправлять ее во внешний мир, и вот тут-то жизнь перестает казаться медом.

Есть несколько факторов, затрудняющих ввод-вывод. Во-первых, ввод и вывод – совершенно разные вещи, но обычно мы мысленно объединяем их. Во-вторых, операции ввода-вывода столь же разнообразны, как и насекомые.

История знает такие устройства, как магнитные барабаны, перфоленты, магнитные ленты, перфокарты и телетайпы. Некоторые имели механические детали, другие были электромагнитными от начала и до конца. Одни позволяли только считывать информацию, другие только записывать, а третьи умели делать и то, и другое. Некоторые записываемые устройства позволяли стирать данные, другие – нет. Одни устройства были принципиально последовательными, другие допускали произвольный доступ. Информация на некоторых устройствах хранилась постоянно, другие были энергозависимыми. Одни устройства требовали человеческого вмешательства, другие нет. Есть устройства символьного и блочного ввода-вывода. На некоторых блочных устройствах можно хранить только блоки постоянной длины, другие допускают и переменную длину блока. Одни устройства надо периодически опрашивать, другие управляются прерываниями. Прерывания можно реализовать аппаратно, программно или смешанным образом. Есть буферизованный и небуферизованный ввод-вывод. Бывает ввод-вывод с отображением на память и канальный, а с появлением таких операционных систем, как UNIX, мы узнали об устройствах ввода-вывода, отображаемых на элементы файловой системы. Программировать ввод-вывод доводилось на машинном языке, на языке ассемблера и на языках высокого уровня. В некоторые языки механизм ввода-вывода жестко встроен, другие вообще не включают ввод-вывод в спецификацию языка. Приходилось выполнять ввод-вывод с помощью подходящего драйвера или уровня абстракции и без оного.

Если все это кажется вам хаотичным скоплением разнородных вещей, то так оно и есть. Отчасти сложность проистекает из самой природы ввода-вывода, отчасти это результат компромиссов, принятых при проектировании, а отчасти следствие наследия прошлых лет, устоявшихся традиций и особенностей различных языков и операционных систем.

Ввод-вывод в Ruby сложен, потому что он сложен в принципе. Но мы старались описать его как можно понятнее и показать, где и когда стоит применять различные приемы.

В основе системы ввода-вывода в Ruby лежит класс `IO`, который определяет поведение всех операций ввода-вывода. С ним тесно связан (и наследует ему) класс `File`. В класс `File` вложен класс `Stat`, инкапсулирующий различные сведения о файле (например, разрешения и временные штампы). Методы `stat` и `lstat` возвращают объекты типа `File::Stat`.

В модуле `FileTest` также есть методы, позволяющие опрашивать практически те же свойства. Эти методы имеются и в классе `File`, где являются методами класса. Наконец, методы ввода-вывода есть и в модуле `Kernel`, который подмешивается к классу `Object` (предку всех объектов, включая и классы). Это простые процедуры, которыми мы пользовались на протяжении всей книги, не думая о том, от имени какого объекта они вызываются. По умолчанию они настроены на стандартный ввод и стандартный вывод.

Поначалу может показаться, что это хаотическое нагромождение перекрывающейся функциональности. Но в каждый момент времени вам необходима лишь небольшая часть всей инфраструктуры.

На более высоком уровне Ruby предлагает механизмы, позволяющие сделать объекты устойчивыми. Библиотеки `Marshal`, `YAML` и `JSON` реализует простую сериализацию объектов, тогда как `CSV` и `SQLite` сохраняют данные в файлах на диске.

На самом высоком уровне возможен интерфейс с системами управления базами данных, например, PostgreSQL, MySQL или Redis. Тема внешних хранилищ настолько сложна, что ей можно было бы посвятить одну или даже несколько книг. Мы ограничимся лишь кратким введением. В некоторых случаях будут даны ссылки на документацию в сети.

10.1. Файлы и каталоги

Говоря *файл*, мы обычно, хотя и не всегда, имеем в виду файл на диске. Концепция файла в Ruby, как и в других языках, – это полезная абстракция. Говоря *каталог*, мы подразумеваем каталог или папку в смысле, принятом в UNIX и Windows.

Класс `File` тесно связан с классом `IO`, которому наследует. Класс `Dir` связан с ним не так тесно, но мы решили рассмотреть файлы и каталоги вместе, поскольку между ними имеется концептуальная связь.

10.1.1. Открытие и закрытие файлов

Метод класса `File.new`, создающий новый объект `File`, также открывает файл. Первым параметром, естественно, является имя файла.

Необязательный второй параметр называется *строкой задания режима*, он говорит, как нужно открывать файл (для чтения, для записи и т.д.). (Строка задания режима не имеет ничего общего с разрешениями.) По умолчанию предполагается режим "r", то есть только чтение. Ниже показано, как открывать файлы для чтения и записи.

```
file1 = File.new("one")       # Открыть для чтения
file2 = File.new("two", "w")  # Открыть для записи
```

Есть также разновидность метода new, принимающая три параметра. В этом случае второй параметр задает начальные разрешения для файла (обычно записывается в виде восьмеричной константы), а третий представляет собой набор флагов, объединенных союзом ИЛИ. Флаги обозначаются константами, например, File::CREAT (создать файл, если он еще не существует) и File::RDONLY (открыть только для чтения). Такая форма используется редко.

```
file = File.new("three", 0755, File::CREAT|File::WRONLY)
```

В виде любезности по отношению к операционной системе и среде исполнения всегда закрывайте открытые вами файлы. Если файл был открыт для записи, то это не просто вежливость, а способ предотвратить потерю данных. Для закрытия файла предназначен метод close:

```
out = File.new("captains.log", "w")
# Обработка файла...
out.close
```

Имеется также метод open. В простейшей форме это просто синоним new:

```
trans = File.open("transactions","w")
```

Но методу open можно также передать блок, и это более интересно. Если блок задан, то ему в качестве параметра передается открытый файл. Файл остается открытым на протяжении всего времени нахождения в блоке и автоматически закрывается при выходе из него. Пример:

```
File.open("somefile","w") do |file
  file.puts "Строка 1"
  file.puts "Строка 2"
  file.puts "Третья и последняя строка"
end
# Теперь файл закрыт
```

Это элегантный способ гарантировать закрытие файла по завершении работы с ним. Кроме того, при такой записи весь код обработки файла сосредоточен в одном месте.

Метод reopen ассоциирует с объектом-получателем новый поток. В примере ниже мы отключаем запись на стандартный вывод для ошибок, а затем снова включаем:

```
save = STDERR.dup
STDERR.reopen("/dev/null")
# Работаем молча...
STDERR.reopen(save)
```

10.1.2. Обновление файла

Чтобы открыть файл для чтения и записи, достаточно добавить знак плюс (+) в строку задания режима (см. раздел 10.1.1):

```
f1 = File.new("file1", "r+")
# Чтение/запись, от начала файла.

f2 = File.new("file2", "w+")
# Чтение/запись; усечь существующий файл или создать новый.

f3 = File.new("file3", "a+")
# Чтение/запись; перейти в конец существующего файла или создать новый.
```

10.1.3. Дописывание в конец файла

Чтобы дописать данные в конец существующего файла, нужно указать строку задания режима "a" (см. раздел 10.1.1):

```
logfile = File.open("captains_log", "a")
# Добавить строку в конец и закрыть файл.
logfile.puts "Stardate 47824.1: Our show has been canceled."
logfile.close
```

10.1.4. Прямой доступ к файлу

Для чтения из файла в произвольном порядке, а не последовательно, можно воспользоваться методом `seek`, который класс `File` наследует от `IO`. Проще всего перейти на байт в указанной позиции. Номер позиции отсчитывается от начала файла, причем самый первый байт находится в позиции 0.

```
# myfile содержит строку: abcdefghi
file = File.new("myfile")
file.seek(5)
str = file.gets                    # "fghi"
```

Если все строки в файле имеют одинаковую длину, то можно перейти сразу в начало нужной строки:

```
# Предполагается, что все строки имеют длину 20.
# Строка N начинается с байта (N-1)*20
file = File.new("fixedlines")
file.seek(5*20)                    # Шестая строка!
```

Для выполнения относительного поиска воспользуйтесь вторым параметром. Константа `IO::SEEK_CUR` означает, что смещение задано относительно текущей позиции (и может быть отрицательным):

```
file = File.new("somefile")
file.seek(55)                      # Позиция 55
file.seek(-22, IO::SEEK_CUR)       # Позиция 33
file.seek(47, IO::SEEK_CUR)        # Позиция 80
```

Можно также искать относительно конца файла, в таком случае смещение может быть только отрицательным:

```
file.seek(-20, IO::SEEK_END)       # двадцать байтов от конца файла
```

Есть еще и третья константа IO::SEEK_SET, но это значение по умолчанию (поиск относительно начала файла).

Метод tell возвращает текущее значение позиции в файле, у него есть синоним pos:

```
file.seek(20)
pos1 = file.tell        # 20
file.seek(50, IO::SEEK_CUR)
pos2 = file.pos      # 70
```

Метод rewind устанавливает указатель файла в начало. Его название восходит ко временам использования магнитных лент.

Для выполнения прямого доступа файл часто открывается в режиме обновления (для чтения и записи). Этот режим обозначается знаком + в начале строки задания режима (см. раздел 10.1.2).

10.1.5. Работа с двоичными файлами

Хотя любой файл в конечном итоге является двоичным, в разговорной речи принято называть двоичными файлы, которые человек не воспринимает как текст. Например, это может быть сжатый либо зашифрованный файл. Или файл, содержащий аудио либо видеоданные.

Если вы хотите сообщить Ruby, что нужно читать двоичные данные, добавьте символ "b" в строку задания режима при открытии файла. Все возвращаемые в таком режиме строки будут иметь кодировку ASCII-8BIT, т. е. считаются строкой байтов без какой-либо кодировки.

В двоичном режиме можно читать байты и производить над ними такие манипуляции, которые при наличии кодировки были бы недопустимы:

```
File.write("invalid", "\xFC\x80\x80 \x80\x80\xAF")
File.read("invalid", mode: "r").split(" ")
# недопустимая последовательность байтов в UTF-8
File.read("invalid", mode: "rb").split(" ")
# ["\xFC\x80", "\x80\xAF"]
```

В Windows двоичный режим означает также, что пару символов \r\n (возврат каретки и перевод строки), завершающую строку, не следует преобразовывать в один символ \n.

Еще одно важное отличие – интерпретация символа Ctrl+Z как конца файла в текстовом режиме:

```
# myfile содержит "12345\0326789\r".
# Обратите внимание на восьмеричное 032 (^Z)
File.open("myfile","rb") {|f| str = f.sysread(15) }.size  # 11
File.open("myfile","r") {|f| str = f.sysread(15) }.size   # 5
```

В следующем фрагменте показано, что на платформе Windows символ возврата каретки не преобразуется в двоичном режиме:

```
# Входной файл содержит всего одну строку: Строка 1.
file = File.open("data")
```

```
line = file.readline              # "Строка 1.\n"
puts "#{line.size} символов."     # 10 символов
file.close

file = File.open("data","rb")
line = file.readline              # "Строка 1.\r\n"
puts "#{line.size} символов."     # 11 символов
file.close
```

Отметим, что упомянутый в коде метод binmode переключает поток в двоичный режим. После переключения вернуться в текстовый режим невозможно.

```
file = File.open("data")
file.binmode
line = file.readline              # "Строка 1.\r\n"
puts "#{line.size} символов."     # 11 символов
file.close
```

При необходимости выполнить низкоуровневый ввод-вывод можно воспользоваться методами sysread и syswrite. Первый принимает в качестве параметра число подлежащих чтению байтов, второй принимает строку и возвращает число записанных байтов. (Если вы начали читать из потока методом sysread, то никакие другие методы использовать не следует. Результаты могут быть непредсказуемы.)

```
input = File.new("infile")
output = File.new("outfile")
instr = input.sysread(10);
bytes = output.syswrite("Это тест.")
```

Отметим, что метод sysread возбуждает исключение EOFError при попытке вызвать его, когда достигнут конец файла (но не в том случае, когда конец файла встретился в ходе успешной операции чтения). Оба метода возбуждают исключение SystemCallError при возникновении ошибки ввода-вывода.

При работе с двоичными данными могут оказаться полезны метод pack из класса Array и метод unpack из класса String.

10.1.6. Блокировка файлов

В тех операционных системах, которые поддерживают такую возможность, метод flock класса File блокирует или разблокирует файл. Вторым параметром может быть одна из констант File::LOCK_EX, File::LOCK_NB, File::LOCK_SH, File::LOCK_UN или их объединение с помощью оператора ИЛИ. Понятно, что многие комбинации не имеют смысла, чаще всего употребляется флаг, задающий неблокирующий режим.

```
file = File.new("somefile")

file.flock(File::LOCK_EX)     # Монопольная блокировка; никакой другой
                              # процесс не может обратиться к файлу.
file.flock(File::LOCK_UN)     # Разблокировать.

file.flock(File::LOCK_SH)     # Разделяемая блокировка (другие
                              # процессы могут сделать то же самое).
```

```
file.flock(File::LOCK_UN)     # Разблокировать.

locked = file.flock(File::LOCK_EX   File::LOCK_NB)
# Пытаемся заблокировать файл, но не приостанавливаем программу, если
# не получилось; в таком случае переменная locked будет равна false.
```

Для семейства операционных систем Windows эта функция не реализована.

10.1.7. Простой ввод-вывод

Вы уже знакомы с некоторыми методами ввода-вывода из модуля `Kernel`, мы вызывали их без указания вызывающего объекта. К ним относятся функции `gets` и `puts`, а также `print`, `printf` и `p` (последний вызывает метод объекта `inspect`, чтобы распечатать его в понятном для человека виде).

Но есть и другие методы, которые следует упомянуть для полноты. Метод `putc` выводит один символ. (Парный метод `getc` не реализован в модуле `Kernel` по техническим причинам, однако он есть у любого объекта класса `IO`). Если параметром является объект `String`, то печатается первый символ строки.

```
putc(?\n)    # Вывести символ новой строки
putc("X")    # Вывести букву X
```

Интересный вопрос: куда направляется вывод, если эти методы вызываются без указания объекта. Начнем с того, что в среде исполнения Ruby определены три глобальные константы, соответствующим трем стандартным потокам ввода-вывода, к которым мы привыкли в UNIX. Это `STDIN`, `STDOUT` и `STDERR`. Все они имеют тип `IO`.

Имеется также глобальная переменная `$stdout`, именно в нее направляется весь вывод, формируемый методами из `Kernel`. Она инициализирована значением `STDOUT`, так что данные отправляются на стандартный вывод, как и следовало ожидать. В любой момент переменной `$stdout` можно присвоить другое значение, являющееся объектом `IO`.

```
diskfile = File.new("foofile","w")
puts "Привет..."       # выводится на stdout
$stdout = diskfile
puts "Пока!"           # выводится в файл "foofile"
diskfile.close
$stdout = STDOUT       # восстановление исходного значения
puts "Это все."        # выводится на stdout
```

Помимо метода `gets`, в модуле `Kernel` есть также методы ввода `readline` и `readlines`. Первый аналогичен `gets` в том смысле, что возбуждает исключение `EOFError` при попытке читать за концом файла, а не просто возвращает `nil`. Последний эквивалентен методу `IO.readlines` (то есть считывает весь файл в память).

Откуда мы получаем ввод? Есть переменная `$stdin`, которая по умолчанию равна `STDIN`. Точно также, существует поток стандартного вывода для ошибок (`$stderr`, по умолчанию равен `STDERR`).

Еще имеется интересный глобальный объект `ARGF`, представляющий конкатенацию всех файлов, указанных в командной строке. Это не объект класса `File`, хотя

и напоминает таковой. По умолчанию ввод связан именно с этим объектом, если в командной строке задан хотя бы один файл.

```
# cat.rb
# Прочитать все файлы, а затем вывести их
puts ARGF.read
# А при таком способе более экономно расходуется память:
puts ARGF.readline until ARGF.eof?
# Пример:  ruby cat.rb file1 file2 file3
```

Чтение из стандартного ввода (STDIN) происходит в обход методов Kernel. Поэтому можно обойти (или не обходить) ARGF, как показано ниже:

```
# Прочитать строку из стандартного ввода
str1 =  STDIN.gets
# Прочитать строку из ARGF
str2 = ARGF.gets
# А теперь снова из стандартного ввода
str3 =  STDIN.gets
```

Существует возможность читать как на уровне символов, так и на уровне байтов. В случае однобайтовой кодировки разница только в том, что байт имеет тип Fixnum, а символ – это односимвольная строка:

```
c = input.getc
b = input.getbyte
input.ungetc          # Эти две операции не всегда
input.ungetbyte       # возможны.
b = input.readbyte    # Как getbyte, но может возбуждать EOFError
```

10.1.8. Буферизованный и небуферизованный ввод-вывод

В некоторых случаях Ruby осуществляет буферизацию самостоятельно. Рассмотрим следующий фрагмент:

```
print "Привет... "
sleep 10
print "Пока!\n"
```

Если запустить эту программу, то вы увидите, что сообщения «Привет» и «Пока» появляются одновременно, после завершения sleep. При этом первое сообщение не завершается символом новой строки.

Это можно исправить, вызвав метод flush для опустошения буфера вывода. В данном случае вывод идет в поток $defout (подразумеваемый по умолчанию для всех методов Kernel, которые занимаются выводом). И поведение оказывается ожидаемым, то есть первое сообщение появляется раньше второго.

```
print "Привет... "
STDOUT.flush
sleep 10
print "Пока!\n"
```

Буферизацию можно отключить (или включить) методом sync=, а метод sync позволяет узнать текущее состояние.

```
buf_flag = $defout.sync      # true
STDOUT.sync = false
buf_flag = STDOUT.sync       # false
```

Есть также еще по крайней мере один низкий уровень буферизации, который не виден. Если метод getc возвращает символ и продвигает вперед указатель файла или потока, то метод ungetc возвращает символ назад в поток.

```
ch = mystream.getc           # ?A
mystream.ungetc(?C)
ch = mystream.getc           # ?C
```

Тут следует иметь в виду три вещи. Во-первых, только что упомянутая буферизация не имеет отношения к механизму буферизации, о котором мы говорили выше в этом разделе. Иными словами, предложение sync=false не отключает ее. Во-вторых, вернуть в поток можно только один символ, при попытке вызвать метод ungetc несколько раз будет возвращен только символ, прочитанный последним. И, в-третьих, метод ungetc не работает для принципиально небуферизуемых операций (например, sysread).

10.1.9. Манипулирование правами владения и разрешениями на доступ к файлу

Вопрос о владении файлами и разрешениях сильно зависит от платформы. Как правило, в системе UNIX функций больше, чем предоставляет Ruby, а на других платформах многие возможности не реализованы.

Для определения владельца и группы файла (это целые числа), класс File::Stat предоставляет методы экземпляра uid и gid:

```
data = File.stat("somefile")
owner_id = data.uid
group_id = data.gid
```

В классе File::Stat есть также метод экземпляра mode, который возвращает текущий набор разрешений для файла.

```
perms = File.stat("somefile").mode
```

В классе File имеется метод класса и экземпляра chown, который позволяет изменить идентификаторы владельца и группы. Метод класса принимает произвольное число файлов. Если идентификатор не нужно изменять, можно передать nil или -1.

```
uid = 201
gid = 10
File.chown(uid, gid, "alpha", "beta")
f1 = File.new("delta")
f1.chown(uid, gid)
f2 = File.new("gamma")
f2.chown(nil, gid)       # Оставить идентификатор владельца без изменения
```

Разрешения можно изменить с помощью метода chmod (у него также есть два варианта: метод класса и метод экземпляра). Традиционно разрешения представляют восьмеричным числом, хотя это и не обязательно.

```
File.chmod(0644, "epsilon", "theta")
f = File.new("eta")
f.chmod(0444)
```

Процесс всегда работает от имени какого-то пользователя (возможно, root), поэтому с ним связан идентификатор пользователя (мы сейчас говорим о *действующем* идентификаторе пользователе). Часто нужно знать, имеет ли этот пользователь право читать, писать или исполнять данный файл. В классе `File::Stat` есть методы экземпляра для получения такой информации.

```
info = File.stat("/tmp/secrets")
rflag = info.readable?
wflag = info.writable?
xflag = info.executable?
```

Иногда нужно отличить действующий идентификатор пользователя от реального. На этот случай предлагаются методы экземпляра `readable_real?`, `writable_real?` и `executable_real?`.

```
info = File.stat("/tmp/secrets")
rflag2 = info.readable_real?
wflag2 = info.writable_real?
xflag2 = info.executable_real?
```

Можно сравнить владельца файла с действующим идентификатором пользователя (и идентификатором группы) текущего процесса. В классе `File::Stat` для этого есть методы `owned?` и `grpowned?`.

Отметим, что многие из этих методов можно найти также в модуле `FileTest`:

```
rflag = FileTest::readable?("pentagon_files")
# Прочие методы: writable? executable? readable_real?
#    writable_real? executable_real? owned? grpowned?
# Отсутствуют здесь: uid gid mode
```

Маска umask, ассоциированная с процессом, определяет начальные разрешения для всех созданных им файлов. Стандартные разрешения 0777 логически пересекаются (AND) с отрицанием umask, то есть биты, поднятые в маске, «маскируются» или сбрасываются. Если вам удобнее, можете представлять себе эту операцию как вычитание (без занимания). Следовательно, если задана маска 022, то все файлы создаются с разрешениями 0755.

Получить или установить маску можно с помощью метода `umask` класса `File`. Если ему передан параметр, то он становится новым значением маски (при этом метод возвращает старое значение).

```
File.umask(0237)              # Установить umask
current_umask = File.umask    # 0237
```

Некоторые биты режима файла (например, бит фиксации (sticky bit)) не имеют прямого отношения к разрешениям. Их обсуждение см. в разделе 10.1.12.

10.1.10. Получение и установка временных меток

С каждым файлом на диске связано несколько временных меток (в разных операционных системах они различны). Ruby понимает три таких метки: время модификации (когда в последний раз было изменено содержимое файла), время досту-

па (когда в последний раз файл читался) и время изменения (когда в последний раз была изменена информация о файле, хранящаяся в каталоге).

Получить эту информацию можно тремя разными способами, хотя все они дают один и тот же результат.

Методы mtime, atime и ctime класса File возвращают временные метки, не требуя предварительного открытия файла или даже создания объекта File.

```
t1 = File.mtime("somefile")
# Thu Jan 04 09:03:10 GMT-6:00 2001
t2 = File.atime("somefile")
# Tue Jan 09 10:03:34 GMT-6:00 2001
t3 = File.ctime("somefile")
# Sun Nov 26 23:48:32 GMT-6:00 2000
```

Если файл, представленный экземпляром File, уже открыт, то можно воспользоваться методами этого экземпляра.

```
myfile = File.new("somefile")
t1 = myfile.mtime
t2 = myfile.atime
t3 = myfile.ctime
```

А если имеется экземпляр класса File::Stat, то и у него есть методы, позволяющие получить ту же информацию:

```
myfile = File.new("somefile")
info = myfile.stat
t1 = info.mtime
t2 = info.atime
t3 = info.ctime
```

Отметим, что объект File::Stat возвращается методом класса (или экземпляра) stat из класса File. Метод класса lstat (или одноименный метод экземпляра) делает то же самое, но возвращает информацию о состоянии самой ссылки, а не файла, на который она ведет. Если имеется цепочка из нескольких ссылок, то метод следует по каждой из них, кроме последней.

Для изменения времени доступа и модификации применяется метод utime, которому можно передать несколько файлов. Время можно создать в виде объекта Time или числа секунд, прошедших с точки отсчета.

```
today = Time.now
yesterday = today - 86400
File.utime(today, today, "alpha")
File.utime(today, yesterday, "beta", "gamma")
```

Поскольку обе временных метки изменяются одновременно, то при желании оставить одну без изменения, ее сначала следует получить и сохранить.

```
mtime = File.mtime("delta")
File.utime(Time.now, mtime, "delta")
```

10.1.11. Проверка существования и получение размера файла

Часто необходимо знать, существует ли файл с данным именем. Это позволяет выяснить метод exist? из модуля FileTest:

```
flag = FileTest::exist?("LochNessMonster")
flag = FileTest::exists?("UFO")
# exists? является синонимом exist?
```

С вопросом о том, существует ли файл, связан другой вопрос: а есть ли в нем какие-нибудь данные? Ведь файл может существовать, но иметь нулевую длину (а это почти то же самое, что не существует).

Если нас интересует только, пуст файл или нет, то в классе `File::Stat` есть два метода экземпляра, отвечающих на этот вопрос. Метод `zero?` возвращает true, если длина файла равна нулю, и `false` в противном случае.

```
flag = File.new("somefile").stat.zero?
```

Метод `size?` возвращает либо размер файла в байтах, если он больше нуля, либо `nil` для файла нулевой длины. Не сразу понятно, почему `nil`, а не 0. Дело в том, что метод предполагалось использовать в качестве предиката, а значение истинности нуля в Ruby – `true`, тогда для `nil` оно равно `false`.

```
if File.size?("myfile")
  puts "В файле есть данные."
else
  puts "Файл пуст."
end
```

Далее возникает следующий вопрос: «Каков размер файла?» Мы уже видели, что для непустого файла метод `size?` возвращает длину. Но если мы применяем его не в качестве предиката, то значение `nil` только путает.

В классе `File` для ответа на этот вопрос есть как метод класса, так и метод экземпляра:

```
size1 = File.size("filename")   # возвращает 0, если файл filename пуст
```

Чтобы получить размер файла в блоках, а не в байтах, можно обратиться к методу `blocks` из класса `File::Stat`. Результат, конечно, зависит от операционной системы. (Метод `blksize` сообщает размер блока операционной системы.)

```
info = File.stat("somefile")
total_bytes = info.blocks * info.blksize
```

10.1.12. Опрос специальных свойств файла

У файла есть много свойств, которые можно опросить. Мы перечислим в этом разделе те встроенные методы, для которых не нашлось другого места. Почти все они являются предикатами.

Читая этот раздел (да и большую часть этой главы), помните о двух вещах. Во-первых, любую проверку, которую можно выполнить с помощью метода класса, можно произвести и обратившись к методу экземпляра любого файлового объекта. Во-вторых, функциональность класса `File` и объекта `File::Stat` (возвращаемого методом `stat` или `lstat`) сильно перекрывается. В некоторых случаях есть несколько разных способов вызвать по сути один и тот же метод. Мы не будем каждый раз приводить все варианты.

В некоторых операционных системах устройства подразделяются на блочные и символьные. Файл может ссылаться как на то, так и на другое, но не на оба сразу. Методы blockdev? и chardev? из модуля FileTest проверяют тип устройства:

```
flag1 = FileTest::chardev?("/dev/hdisk0")  # false
flag2 = FileTest::blockdev?("/dev/hdisk0") # true
```

Иногда нужно знать, ассоциирован ли данный поток с терминалом. Метод tty? класса IO (синоним isatty) дает ответ на этот вопрос:

```
flag1 = STDIN.tty?                  # true
flag2 = File.new("diskfile").isatty  # false
```

Поток может быть связан с каналом (pipe) или сокетом. В модуле FileTest есть методы для опроса этих условий:

```
flag1 = FileTest::pipe?(myfile)
flag2 = FileTest::socket?(myfile)
```

Напомним, что каталог – это частный случай файла. Поэтому нужно уметь отличать каталоги от обычных файлов, для чего предназначены следующие два метода:

```
file1 = File.new("/tmp")
file2 = File.new("/tmp/myfile")
test1 = file1.directory?        # true
test2 = file1.file?             # false
test3 = file2.directory?        # false
test4 = file2.file?             # true
```

В классе File есть также метод класса ftype, который сообщает вид потока, одноименный метод экземпляра находится в классе File::Stat. Этот метод возвращает одну из следующих строк: file, directory, blockSpecial, characterSpecial, fifo, link или socket (строка fifo относится к каналу).

```
this_kind = File.ftype("/dev/hdisk0")    # "blockSpecial"
that_kind = File.new("/tmp").stat.ftype  # "directory"
```

В маске, описывающей режим файла, можно устанавливать или сбрасывать некоторые биты. Они не имеют прямого отношения к битам, обсуждавшимся в разделе 10.1.9. Речь идет о битах *set-group-id*, *set-user-id* и *бите фиксации* (sticky bit). Для каждого из них есть соответствующий метод.

```
file = File.new("somefile")
sticky_flag = file.sticky?
setgid_flag = file.setgid?
setuid_flag = file.setuid?
```

На дисковый файл могут вести символические или физические ссылки (в тех операционных системах, где такой механизм поддерживается). Чтобы проверить, является ли файл символической ссылкой на другой файл, обратитесь к методу symlink?. Для подсчета числа физических ссылок на файл служит метод nlink (он есть только в классе File::Stat). Физическая ссылка неотличима от обычного файла, это просто файл, для которого есть несколько имен и записей в каталоге.

```
File.symlink("yourfile","myfile")            # Создать ссылку
is_sym = File.symlink?("myfile")             # true
hard_count = File.new("myfile").stat.nlink   # 0
```

Отметим попутно, что в предыдущем примере мы воспользовались методом класса symlink из класса File для создания символической ссылки.

В редких случаях может понадобиться информация о файле еще более низкого уровня. В классе File::Stat есть еще три метода экземпляра, предоставляющих такую информацию. Метод dev возвращает целое число, идентифицирующее устройство, на котором расположен файл. Метод rdev возвращает целое число, описывающее тип устройства, а для дисковых файлов метод ino возвращает номер первого *индексного узла*, занятого файлом.

```
file = File.new("diskfile")
info = file.stat
device = info.dev
devtype = info.rdev
inode = info.ino
```

10.1.13. Каналы

Ruby поддерживает разные способы читать из канала и писать в него. Метод класса IO.popen открывает канал и связывает с возвращенным объектом стандартные ввод и вывод процесса. Часто с разными концами канала работают разные потоки, но в примере ниже запись и чтение осуществляет один и тот же поток:

```
check = IO.popen("spell","r+")
check.puts("'T was brillig, and the slithy toves")
check.puts("Did gyre and gimble in the wabe.")
check.close_write
list = check.readlines
list.collect! { |x| x.chomp }
# list равно %w[brillig gimble gyre slithy toves wabe]
```

Отметим, что вызов close_write обязателен, иначе мы никогда не достигнем конца файла при чтении из канала.

Существует также блочная форма:

```
File.popen("/usr/games/fortune") do |pipe|
  quote = pipe.gets
  puts quote
  # На чистом диске можно искать бесконечно. - Том Стил
end
```

Если задана строка "-", то запускается новый экземпляр Ruby. Если при этом задан еще и блок, то он работает в двух разных процессах, как в результате разветвления (fork); блоку в процессе-потомке передается nil, а в процессе-родителе – объект IO, с которым связан стандартный ввод или стандартный вывод.

```
IO.popen("-") do |mypipe|
  if mypipe
    puts "Я родитель: pid = #{Process.pid}"
```

```
    listen = mypipe.gets
    puts listen
  else
    puts "Я потомок: pid = #{Process.pid}"
  end
end

# Печатается:
#   Я родитель: pid = 10580
#   Я потомок: pid = 10582
```

Метод pipe возвращает также два конца канала, связанных между собой. В следующем примере мы создаем два потока, один из которых передает сообщение другому (то самое сообщение, которое Сэмюэль Морзе впервые послал по телеграфу). Дополнительные сведения см. в главе 13.

```
pipe = IO.pipe
reader = pipe[0]
writer = pipe[1]

str = nil
thread1 = Thread.new(reader,writer) do |reader,writer|
  # writer.close_write
  str = reader.gets
  reader.close
end

thread2 = Thread.new(reader,writer) do |reader,writer|
  # reader.close_read
  writer.puts("What hath God wrought?")
  writer.close
end

thread1.join
thread2.join

puts str          # What hath God wrought?
```

10.1.14. Специальные операции ввода-вывода

В Ruby можно выполнять низкоуровневые операции ввода-вывода. Мы только упомянем о существовании таких методов; если вы собираетесь ими пользоваться, имейте в виду, что некоторые машинно-зависимы (различаются даже в разных версиях UNIX).

Метод ioctl принимает два аргумента: целое число, определяющее операцию, и целое число либо строку, представляющую параметр этой операции.

Метод fcntl также предназначен для низкоуровневого управления файловыми потоками системно-зависимым образом. Он принимает такие же параметры, как ioctl.

Метод select (в модуле Kernel) принимает до четырех параметров. Первый из них – массив дескрипторов для чтения, а остальные три необязательны (массив дескрипторов для записи, дескрипторов для ошибок и величина таймаута). Если на каком-то из устройств, дескрипторы которых заданы в первом массиве, оказываются новые данные для чтения или какое-то из устройств, дескрипторы которых перечислены во втором массиве, готово к выполнению записи, то метод возвращает массив из трех элементов, каждый из которых в свою очередь является массивом, в котором указаны дескрипторы тех устройств, которые готовы к выполнению ввода-вывода.

Метод syscall из модуля Kernel принимает по меньшей мере один целочисленный параметр (а всего до девяти целочисленных или строковых параметров). Первый параметр определяет выполняемую операцию ввода-вывода.

Метод fileno возвращает обычный файловый дескриптор, ассоциированный с потоком ввода-вывода. Это наименее системно-зависимый из всех перечисленных выше методов.

```
desc = $stderr.fileno      # 2
```

10.1.15. Неблокирующий ввод-вывод

«За кулисами» Ruby предпринимает согласованные меры, чтобы операции ввода-вывода не блокировали выполнение программы. В большинстве случаев для управления вводом-выводом можно пользоваться потоками – один поток может выполнить блокирующую операцию, а второй будет продолжать работу.

Но если вы все-таки хотите включить неблокирующий режим ввода-вывода, то это можно сделать с помощью методов read_nonblock и write_nonblock. Первый пользуется внутренним системным вызовом read(2), поэтому может возбуждать исключения, соответствующие обычным кодам ошибок, например Errno::EWOULDBLOCK и т. п. Он эквивалентен методу readpartial с установленным флагом неблокирующего режима (см. раздел 10.1.16).

```
string = input.read(64)     # читать 64 байта
buffer = ""
input.read(64, buffer)      # необязательный буфер
```

По достижении конца файла возбуждается исключение EOFError. Если возникло исключение EWOULDBLOCK, то в следующий раз метод можно вызывать только после того, как станут доступны данные. Например:

```
begin
       data = input.read_nonblock(256)
rescue Errno::EWOULDBLOCK
       IO.select([input])
       retry
end
```

Аналогично метод write_nonblock обращается к системному вызову write(2) (и возбуждает соответствующие исключения). Он принимает в качестве аргумента строку и возвращает количество записанных байтов (которое может оказаться

меньше длины строки). Исключение EWOULDBLOCK следует обрабатывать так же, как в случае описанного выше метода read_nonblock.

10.1.16. Применение метода readpartial

Метод readpartial упрощает ввод-вывод в поток, например, в сокет.

Параметр «максимальная длина» (max length) обязателен. Если задан параметр buffer, то он должен ссылаться на строку, в которой будут храниться данные.

```
data = sock.readpartial(128)  # Читать не более 128 байтов
```

Метод readpartial игнорирует установленный флаг неблокирующего ввода-вывода. Он может блокировать программу, но лишь при выполнении следующих трех условий: буфер объекта IO пуст, в потоке ничего нет и поток еще не достиг конца файла.

Таким образом, если в потоке есть данные, то readpartial ***не будет*** блокировать программу. Он читает не более указанного числа байтов, а если байтов оказалось меньше, то прочитает их и вернет управление.

Если в потоке нет данных, но при этом достигнут конец файла, то readpartial немедленно возбуждает исключение EOFError.

Если вызов блокирует выполнение, то метод ожидает, пока не произойдет одно из двух событий: придут новые данные или обнаружится конец файла. Если поступают данные, метод возвращает их вызывающей программе. А в случае обнаружения конца файла возбуждает исключение EOFError.

Метода sysread, вызванный в блокирующем режиме, ведет себя похоже на readpartial. Если буфер пуст, их поведение вообще идентично.

10.1.17. Манипулирование путевыми именами

Основными методами для работы с путевыми именами являются методы класса File.dirname и File.basename; они работают, как одноименные команды UNIX, то есть возвращают имя каталога и имя файла соответственно. Если вторым параметром методу basename передана строка с расширением имени файла, то это расширение исключается.

```
str = "/home/dave/podbay.rb"
dir = File.dirname(str)              # "/home/dave"
file1 = File.basename(str)           # "podbay.rb"
file2 = File.basename(str,".rb")     # "podbay"
```

Хотя это методы класса File, на самом деле они просто манипулируют строками.

Упомянем также метод File.split, которые возвращает обе компоненты (имя каталога и имя файла) в массиве из двух элементов:

```
info = File.split(str)          # ["/home/dave","podbay.rb"]
```

Метод класса expand_path преобразует относительное путевое имя в абсолютный путь. Если операционная система понимает сокращения ~ и ~user, то они тоже учитываются. Необязательный второй аргумент задает путь, начиная от которого следует производить расширение; часто в этом качестве указывают путь к текущему файлу __FILE__.

```
Dir.chdir("/home/poole/personal/docs")
abs = File.expand_path("../../misc")            # "/home/poole/misc"
abs = File.expand_path("misc", "/home/poole")   # "/home/poole/misc"
```

Если передать методу экземпляра path открытый файл, то он вернет путевое имя, по которому файл был открыт.

```
File.new("../../foobar").path   # "../../foobar"
```

Константа File::Separator равна символу, применяемому для разделения компонентов путевого имени (в Windows это обратная косая черта, а в UNIX – прямая косая черта). Имеется также синоним File::SEPARATOR.

Метод класса join использует этот разделитель для составления полного путевого имени из переданного списка компонентов:

```
path = File.join("usr","local","bin","someprog")
# path равно "usr/local/bin/someprog"
# Обратите внимание, что в начало имени разделитель не добавляется!
```

Не думайте, что методы File.join и File.split взаимно обратны. Это не так.

10.1.18. Класс Pathname

Следует знать о существовании стандартной библиотеки pathname, которая предоставляет класс Pathname. По существу, это обертка вокруг классов Dir, File и FileUtils, поэтому он комбинирует многие их функции логичным и интуитивно понятным способом.

```
path = Pathname.new("/home/hal")
file = Pathname.new("file.txt")
p2 = path + file

path.directory?       # true
path.file?            # false
p2.directory?         # false
p2.file?              # true

parts = path2.split   # [Pathname:/home/hal, Pathname:file.txt]
ext = path2.extname   # .txt
```

Как и следовало ожидать, имеется ряд вспомогательных методов. Метод root? пытается выяснить, относится ли данный путь к корневому каталогу, но его можно «обмануть», так как он просто анализирует строку, не обращаясь к файловой системе. Метод parent? возвращает путевое имя родительского каталога данного пути. Метод children возвращает непосредственных потомков каталога, заданного своим путевым именем, в их число включаются как файлы, так и каталоги, но рекурсивного спуска не производится.

```
p1 = Pathname.new("//")     # странно, но допустимо
p1.root?                    # true
p2 = Pathname.new("/home/poole")
p3 = p2.parent              # Путевое имя:/home
items = p2.children         # массив объектов Pathname (все файлы и каталоги, находящиеся
                            # непосредственно в каталоге poole)
```

Как и следовало ожидать, методы relative и absolute пытаются определить, является ли путь относительным или абсолютным (проверяя, есть ли в начале имени косая черта):

```
p1 = Pathname.new("/home/dave")
p1.absolute?                    # true
p1.relative?                    # false
```

Многие методы, например size, unlink и прочие, просто делегируют работу классам File и FileUtils; повторно функциональность не реализуется.

Дополнительную информацию о классе Pathname см. на сайте ruby-doc.org или в любом другом справочном руководстве.

10.1.19. Манипулирование файлами на уровне команд

Часто приходится манипулировать файлами так, как это деластся с помощью командной строки: копировать, удалять, переименовывать и т.д.

Многие из этих операций реализованы встроенными методами, некоторые находятся в модуле FileUtils из библиотеки fileutils. Имейте в виду, что раньше функциональность модуля FileUtils подмешивалась прямо в класс File, теперь эти методы помещены в отдельный модуль.

Для удаления файла служит метод File.delete или его синоним File.unlink:

```
File.delete("history")
File.unlink("toast")
```

Переименовать файл позволяет метод File.rename:

```
File.rename("Ceylon","SriLanka")
```

Создать ссылку на файл (физическую или символическую) позволяют методы File.link и File.symlink соответственно:

```
File.link("/etc/hosts","/etc/hostfile")     # физическая ссылка
File.symlink("/etc/hosts","/tmp/hosts")     # символическая ссылка
```

Файл можно усечь до нулевой длины (или до любой другой), воспользовавшись методом экземпляра truncate:

```
File.truncate("myfile",1000)    # Теперь не более 1000 байтов
```

Два файла можно сравнить с помощью метода compare_file. У него есть синонимы cmp и compare_stream:

```
require "fileutils"

same = FileUtils.compare_file("alpha","beta")  # true
```

Метод copy копирует файл в другое место, возможно, с переименованием. У него есть необязательный флаг, говорящий, что сообщения об ошибках нужно направлять на стандартный вывод для ошибок. Синоним – привычное для программистов UNIX имя cp.

```
require "fileutils"

# Скопировать файл epsilon в theta с протоколированием ошибок.
FileUtils.copy("epsilon","theta", true)
```

Файл можно перемещать методом move (синоним mv). Как и copy, этот метод имеет необязательный параметр, включающий вывод сообщений об ошибках.

```
require "fileutils"

FileUtils.move("/tmp/names","/etc")      # Переместить в другой каталог
FileUtils.move("colours","colors")       # Просто переименовать
```

Метод safe_unlink удаляет один или несколько файлов, предварительно пытаясь сделать их доступными для записи, чтобы избежать ошибок. Если последний параметр равен true или false, он интерпретируется как флаг, задающий режим вывода сообщений об ошибках.

```
require "fileutils"

FileUtils.safe_unlink("alpha","beta","gamma")
# Протоколировать ошибки при удалении следующих двух файлов
FileUtils.safe_unlink("delta","epsilon",true)
```

Наконец, метод install делает практически то же, что syscopy, но сначала проверяет, что целевой файл либо не существует, либо содержит такие же данные.

```
require "fileutils"

FileUtils.install("foo.so","/usr/lib")
# Существующий файл foo.so не будет переписан,
# если он не отличается от нового.
```

Дополнительную информацию о модуле FileUtils см. на сайте ruby-doc.org или в любом другом справочном руководстве.

10.1.20. Ввод символов с клавиатуры

В данном случае мы имеем в виду небуферизованный ввод, когда символ обрабатывается сразу после нажатия клавиши, не дожидаясь, пока будет введена вся строка.

Это можно сделать и в UNIX, и в Windows, но, к сожалению, совершенно по-разному.

Версия для UNIX прямолинейна. Мы переводим терминал в режим прямого ввода (raw mode) и обычно одновременно отключаем эхо-контроль.

```
def getchar
  system("stty raw -echo")  # Прямой ввода без эхо-контроля
  char = STDIN.getc
  system("stty -raw echo")  # Восстановить режим терминала
  char
end
```

На платформе Windows придется написать расширение на C. Альтернативой является использование одной из функций в библиотеке Win32API.

```
require 'Win32API'

def getchar
  char = Win32API.new("crtdll", "_getch", [], 'L').Call
end
```

Поведение в обоих случаях идентично.

10.1.21. Чтение всего файла в память

Чтобы прочитать весь файл в массив, не нужно даже его предварительно открывать. Все сделает метод IO.readlines: откроет файл, прочитает и закроет.

```
arr = IO.readlines("myfile")
lines = arr.size
puts "myfile содержит #{lines} строк."

longest = arr.collect {|x| x.length}.max
puts "Самая длинная строка содержит #{longest} символов."
```

Можно также воспользоваться методом IO.read (который возвращает одну большую строку, а не массив строк).

```
str = IO.read("myfile")
bytes = arr.size
puts "myfile содержит #{bytes} байтов."

longest=str.collect {|x| x.length}.max  # строки - перечисляемые объекты!
puts "Самая длинная строка содержит #{longest} символов."
```

Поскольку класс IO является предком File, то можно вместо этого писать File.readlines и File.read.

10.1.22. Построчное чтение из файла

Чтобы читать по одной строке из файла, можно обратиться к методу класса IO.foreach или к методу экземпляра each. В первом случае файл не нужно явно открывать.

```
# Напечатать все строки, содержащие слово "target"
IO.foreach("somefile") do |line|
  puts line if line =~ /target/
end

# Другой способ...
file = File.new("somefile")
file.each do |line|
  puts line if line =~ /target/
end
```

Отметим, что each_line – синоним each. Он также позволяет получить перечислитель:

```
lines = File.new("somefile").each_line
lines.find{|line| line =~ /target/ } # Используется как перечислитель
```

10.1.23. Побайтное и посимвольное чтение из файла

Для чтения из файла по одному байту служит метод экземпляра each_byte. Он передает в блок один байт (то есть объект типа Fixnum из диапазона 0..255):

```
a_count = 0
File.new("myfile").each_byte do |byte|
      a_count += 1 if byte == 97     # строчная a в коде ASCII
end
```

Можно читать также посимвольно (символ - это в действительности односимвольная строка). В зависимости от используемой кодировки символ может состоять из одного байта (как в ASCII) или из нескольких.

```
a_count = 0
File.new("myfile").each_char do |char|
      a_count += 1 if char == "a"
end
```

Метод each_char также возвращает обычный перечислитель.

10.1.24. Работа со строкой как с файлом

Иногда возникает необходимость рассматривать строку как файл. Что под этим понимается, зависит от конкретной задачи.

Объект определяется, прежде всего, своими методами. В следующем фрагменте показано, как к объекту source применяется итератор; на каждой итерации выводится одна строка. Можете ли вы что-нибудь сказать о типе объекта source, глядя на этот код?

```
source.each do |line|
  puts line
end
```

Это мог бы быть как файл, так и строка, содержащая внутри символы новой строки. В таких случаях строку можно трактовать как файл без всякого труда.

Класс StringIO предоставляет многие методы класса IO, которых нет у обычной строки. В нем есть метод доступа string, ссылающийся на содержимое самой строки.

```
require 'stringio'

ios = StringIO.new("abcdefghijkl\nABC\n123")

ios.seek(5)
ios.puts("xyz")
puts ios.tell              # 9
puts ios.string.inspect    # "abcdexyz\njkl\nABC\n123"

puts ios.getc              # j
ios.ungetc(?w)
puts ios.string.inspect    # "abcdexyz\nwkl\nABC\n123"

s1 = ios.gets              # "wjkl"
s2 = ios.gets              # "ABC"
```

10.1.25. Копирование потока

Метод класса copy_stream служит для копирования потока. Все данные копируются из начального объекта в конечный. Начальный и конечный объекты могут быть

объектами IO или именами файлов. Третий (необязательный) параметр – количество подлежащих копированию байтов (по умолчанию, естественно, предполагается, что нужно копировать все байты из источника). Четвертый параметр – смещение от начала источника (в байтах):

```
src = File.new("garbage.in")
dst = File.new("garbage.out")
IO.copy_stream(src, dst)

IO.copy_stream("garbage.in","garbage.out", 1000, 80)
# В конечный поток копируется 1000 байтов, начиная со смещения 80
```

10.1.26. Работа с кодировками

Эта тема рассматривается в главе 4 «Интернационализация в Ruby». Там же описывается использование кодировок при работе с объектами String и IO.

10.1.27. Чтение данных, встроенных в текст программы

Когда подростком вы учили язык BASIC, копируя программы из журналов, наверное, для удобства часто пользовались предложением DATA. Оно позволяло включать информацию прямо в текст программы, но читать ее так, будто она поступает из внешнего источника.

При желании то же самое можно сделать и в Ruby. Директива __END__ в конце программы говорит, что дальше идут встроенные данные. Их можно читать из глобальной константы DATA, которая представляет собой обычный объект IO. (Отметим, что маркер __END__ должен располагаться с начала строки.)

```
# Распечатать все строки "задом наперед"...
DATA.each_line do |line|
  puts line.reverse
end
__END__
A man, a plan, a canal... Panama!
Madam, I'm Adam.
,siht gnidaer er'uoy fI
.evisserpmi si noitacided ruoy
```

10.1.28. Чтение исходного текста программы

Если вы хотите получить доступ к исходному тексту собственной программы, то можете воспользоваться уже описанным выше трюком (см. раздел 10.1.27).

Глобальная константа DATA – это объекта класса IO, ссылающийся на данные, которые расположены после директивы __END__. Но если выполнить метод rewind, то указатель файла будет переустановлен на начало текста программы.

Следующая программа выводит собственный текст, снабжая его номерами строк. Это не очень полезно, но, быть может, вы найдете и другие применения такой техники.

```
DATA.rewind
num = 1
DATA.each_line do |line|
```

```
  puts "#{'%03d' % num}  #{line}"
  num += 1
end
__END__
```

Отметим, что наличие директивы __END__ *обязательно*, без нее к константе DATA вообще нельзя обратиться.

Другой способ прочитать исходный текст текущего файла – воспользоваться специальной переменной __FILE__. Она содержит полный путь к самому файлу, и из нее можно читать:

```
puts File.read(__FILE__)
```

10.1.29. Работа с временными файлами

Во многих случаях необходимо работать с файлами, которые по сути своей анонимны. Мы не хотим возиться с присваиванием им имен и проверять, что при этом не возникает конфликтов с существующими файлами. И помнить о том, что такие файлы нужно удалять, тоже не хочется.

Все эти проблемы решает библиотека Tempfile. Метод new (синоним open) принимает базовое имя в качестве *начального значения* и конкатенирует его с идентификатором процесса и уникальным порядковым номером. Необязательный второй параметр – имя каталога, в котором создается временный файл, по умолчанию оно равно значению первой из существующих переменных окружения TMPDIR, TMP, или TEMP, а если ни одна из них не задана, то "/tmp".

Возвращаемый объект IO можно многократно открывать и закрывать на протяжении времени работы программы, а по завершении временный файл будет автоматически удален.

У метода close есть необязательный флаг; если он равен true, то файл удаляется сразу после закрытия (не дожидаясь завершения программы). Метод path возвращает полное имя файла, если оно вам по какой-то причине понадобится.

```
require "tempfile"

temp = Tempfile.new("stuff")
name = temp.path              # "/tmp/stuff17060.0"
temp.puts "Здесь был Вася"
temp.close

# Позже...
temp.open
str = temp.gets               # "Здесь был Вася"
temp.close(true)              # Удалить СЕЙЧАС
```

10.1.30. Получение и изменение текущего каталога

Получить имя текущего каталога можно с помощью метода Dir.pwd (синоним Dir.getwd). Эти имена уже давно употребляются как сокращения от «print working directory» (печатать рабочий каталог) и «get working directory» (получить рабочий каталог).

Для изменения текущего каталога служит метод Dir.chdir. В Windows в начале строки можно указывать букву диска.

```
Dir.chdir("/var/tmp")
puts Dir.pwd          # "/var/tmp"
puts Dir.getwd        # "/var/tmp"
```

Этот метод также принимает блок в качестве параметра. Если блок задан, то текущий каталог изменяется только на время выполнения блока, а потом восстанавливается первоначальное значение:

```
Dir.chdir("/home")
Dir.chdir("/tmp") do
  puts Dir.pwd          # /tmp
  # какой-то код...
end
puts Dir.pwd            # /home
```

10.1.31. Изменение текущего корня

В большинстве систем UNIX можно изменить «представление» процесса о том, что такое корневой каталог /. Обычно это делается из соображений безопасности перед запуском небезопасной или непротестированной программы. Метод chroot делает указанный каталог новым корнем:

```
Dir.chdir("/home/guy/sandbox/tmp")
Dir.chroot("/home/guy/sandbox")
puts Dir.pwd  # "/tmp"
```

10.1.32. Обход каталога

Метод класса foreach — это итератор, который последовательно передает в блок каждый элемент каталога. Точно также ведет себя метод экземпляра each.

```
Dir.foreach("/tmp") { |entry| puts entry }

dir = Dir.new("/tmp")
dir.each  { |entry| puts entry }
```

Оба фрагмента печатают одно и то же (имена всех файлов и подкаталогов в каталоге /tmp).

10.1.33. Получение содержимого каталога

Метод класса Dir.entries возвращает массив, содержащий все элементы указанного каталога:

```
list = Dir.entries("/tmp")  # %w[. .. alpha.txt beta.doc]
```

Как видите, включаются и элементы, соответствующие текущему и родительскому каталогу. Если они вам не нужны, придется отфильтровать вручную.

10.1.34. Создание цепочки каталогов

Иногда необходимо создать глубоко вложенный каталог, причем промежуточные каталоги могут и не существовать. В UNIX мы воспользовались бы для этого командой mkdir -p.

В программе на Ruby такую операцию выполняет метод FileUtils.makedirs (из библиотеки fileutils):

```
require "fileutils"
FileUtils.makedirs("/tmp/these/dirs/need/not/exist")
```

10.1.35. Рекурсивное удаление каталога

В UNIX команда rm -rf dir удаляет все поддерево, начиная с каталога dir. Понятно, что применять ее надо с осторожностью.

В классе Pathname имеется метод rmtree, решающий ту же задачу.

```
require 'pathname'
dir = Pathname.new("/home/poole/")
dir.rmtree
```

В классе FileUtils есть метод rm_r, который делает то же самое:

```
require 'fileutils'
FileUtils.rm_r("/home/poole")
```

10.1.36. Поиск файлов и каталогов

В классе Dir имеется метод glob (синоним []), возвращающий массив файлов, имена которых соответствуют заданной маске в смысле оболочки. В простых случаях этого достаточно для поиска нужного файла в каталоге:

```
Dir.glob("*.rb")          # все файлы ruby в текущем каталоге
Dir["spec/**/*_spec.rb"] # все файлы с именами, заканчивающимися
                          # на _spec.rb в каталоге spec/
```

Для более сложных случаев имеется стандартная библиотека find, которая позволяет посетить каждый файл в указанном каталоге и всех его подкаталогах. Следующий метод находит в указанном каталоге файлы по имени (строке) либо регулярному выражению.

```
require "find"

def findfiles(dir, name)
  list = []
  Find.find(dir) do |path|
    Find.prune if [".","..".].include? path
    case name
      when String
        list << path if File.basename(path) == name
      when Regexp
        list << path if File.basename(path) =~ name
    else
      raise ArgumentError
    end
  end
  list
end

findfiles "/home/hal", "toc.txt"
```

```
# ["/home/hal/docs/toc.txt", "/home/hal/misc/toc.txt"]

findfiles "/home", /^[a-z]+.doc/
# ["/home/hal/docs/alpha.doc", "/home/guy/guide.doc",
#  "/home/bill/help/readme.doc"]
```

10.2. Доступ к данным более высокого уровня

Часто возникает необходимость сохранить данные для последующего использования, а не просто записать байты в файл. Для этого мы преобразуем объекты в последовательности байтов и обратно, этот процесс называется *сериализацией*. Существует много способов сериализации данных, мы рассмотрим только самые простые и наиболее употребительные.

Модуль `Marshal` предоставляет простые средства сохранения объектов, а на его основе построена библиотека `PStore`. Формат YAML (и соответствующая библиотека Ruby) дает еще один способ сериализации – в виде простого текста, понятного человеку.

В формате JSON можно хранить только числа, строки, массивы и хэши, зато средства для его чтения и записи имеются в любом языке. CSV-файлы применяются для обмена табличными данными между различными приложениями, в том числе электронными таблицами, например Excel.

Внешние базы данных с доступом к серверу через сокет будут рассмотрены в следующем разделе.

10.2.1. Простой маршалинг

Простейший способ сохранить объект для последующего использования – подвергнуть его *маршалингу*. Модуль `Marshal` позволяет *сериализовать* и *десериализовать* объекты Ruby, представив их в виде строк, которые уже можно записать в файл.

```
# массив элементов [composer, work, minutes]
works = [["Leonard Bernstein","Overture to Candide",11],
         ["Aaron Copland","Symphony No. 3",45],
         ["Jean Sibelius","Finlandia",20]]

# Мы хотим сохранить его для последующего использования...
File.write "store", Marshal.dump("works")

# Намного позже...
works = Marshal.load File.read("store")
```

Сохранять данные таким образом очень удобно, но потенциально и крайне опасно. Можно написать эксплойт, который, воспользовавшись загрузкой маршалированных данных, выполнит не код вашей программы, а совершенно ***произвольный*** код.

Никогда не загружайте маршалированные данные, полученные из внешнего источника, в том числе и от пользователей вашей программы. Используйте вме-

сто этого библиотеки YAML и JSON, которые позволяют безопасно читать данные, полученные из ненадежных источников. Как это делается, будет показано в следующем разделе.

У маршалинга есть также ограничения: не все объекты можно сохранить. Невозможно сериализовать объекты таких системных классов, как IO, Thread и Binding, а также объекты анонимных и синглетных классов.

Первые два байта данных, порождаемых методом Marshal.dump, — номер старшей и младшей версии.

```
Marshal.dump("foo").bytes[0..1] # [4, 8]
```

Ruby успешно загружает такие данные только в том случае, когда номера старших версий данных и метода совпадают, и номер младшей версии данных не больше младшей версии метода. Но если при вызове интерпретатора Ruby задан флаг «болтливости» (verbose или v), то версии должны совпадать точно. Номер версии метода увеличивается при изменении формата маршалинга, однако он не менялся уже много лет.

10.2.2. «Глубокое копирование» с помощью метода Marshal

В Ruby нет операции «глубокого копирования». Например, методы dup и clone хэша не копируют ключи и значения, на которые в хэше есть ссылки. Если количество вложенных ссылок на объекты достаточно велико, то операция копирования превращается в игру «собери палочки».

Ниже предлагается способ реализовать глубокое копирование с некоторыми ограничениями, обусловленными тем, что наш подход основан на использовании класса Marshal со всеми присущими ему недостатками:

```
def deep_copy(obj)
  Marshal.load(Marshal.dump(obj))
end

a = deep_copy(b)
```

10.2.3. Более сложный маршалинг

Иногда мы хотим настроить маршалинг под свои нужды. Для этого нужно создать методы marshal_load и marshal_dump. Если они существуют, то вызываются во время выполнения маршалинга, чтобы вы могли самостоятельно реализовать преобразование данных в строку и обратно.

В следующем примере человек получает 5-процентный доход на начальный капитал с момента рождения. Мы не храним ни возраст, ни текущий баланс, поскольку они являются функциями времени.

```
class Person

    attr_reader :balance, :name

    def initialize(name, birthdate, beginning)
      @name = name
```

```
      @birthdate = birthdate
      @beginning = beginning
      @age = (Time.now - @birthdate) / (365*86400)
      @balance = @beginning * (1.05 ** @age)
    end

      def age
        @age.floor
      end

      def marshal_dump
        {name: @name, birthdate: @birthdate, deposit: @deposit}
      end

      def marshal_load(data)
        initialize(data[:name], data[:birthdate], data[:deposit])
      end
end

p1 = Person.new("Rudy", Time.now - (14 * 365 * 86400), 100)
[p1.name, p1.age, p1.balance]     # ["Rudy", 14, 197.99315994394l7]

p2 = Marshal.load Marshal.dump(p1)
[p2.name, p2.age, p2.balance]     # ["Rudy", 14, 197.9931599440351]
```

При сохранении объекта этого типа атрибуты age и balance не сохраняются. А когда объект восстанавливается, они вычисляются заново. Отметим, что метод marshal_load предполагает, что объект существует; это один из немногих случаев, когда метод initialize приходится вызывать явно (обычно это делает метод new).

10.2.4. Маршалинг в формате YAML

Аббревиатура *YAML* означает «YAML Ain't Markup Language» (YAML – не язык разметки). Это не что иное, как гибкий, понятный человеку формат обмена данными между программами, возможно, написанными на разных языках.

Если загружена библиотека yaml, то в нашем распоряжении оказываются методы YAML.dump и YAML.load, очень похожие на соответственные методы из класса Marshal. Было бы поучительно сериализовать несколько объектов в формате YAML и посмотреть, что получается:

```
require 'yaml'

Person = Struct.new(:name)

puts YAML.dump("Hello, world.")
puts YAML.dump({this: "is a hash",
      with: "symbol keys and string values"})
puts YAML.dump([1, 2, 3])
```

```
puts YAML.dump Person.new("Alice")

# Выводится:
# --- Hello, world.
# ...
# ---
# :this: is a hash
# :with: symbol keys and string values
# ---
# - 1
# - 2
# - 3
# --- !ruby/struct:Person
# name: Alice
```

Любой YAML-документ начинается строкой «---», за которой следуют сериализованные объекты в понятном человеку виде.

Использовать метод `YAML.load` для загрузки строки столь же просто. А метод `YAML.load_file` идет на шаг дальше и позволяет указать имя загружаемого файла. Пусть имеется такой файл `data.yaml`:

```
---
- "Hello, world"
- 237
-
  - Jan
  - Feb
  - Mar
  - Apr
-
  just a: hash.
  This: is
```

Это четыре элемента данных, сгруппированные в единый массив. Если загрузить этот файл, то получится массив:

```
require 'yaml'
p YAML.load_file("data.yaml")
# Выводится:
# ["Hello, world", 237, ["Jan", "Feb", "Mar", "Apr"],
#  {"just a"=>"hash.", "This"=>"is"}]
```

В общем и целом, YAML – еще один способ выполнить маршалинг объектов. На верхнем уровне его можно использовать для самых разных целей. Например, человек может не только читать данные в этом формате, но и редактировать их, поэтому его естественно применять для записи конфигурационных файлов и т.п.

Поскольку библиотека YAML умеет выполнять маршалинг объектов Ruby, к ней относятся высказанные ранее предостережения. Никогда не загружайте YAML-файлы, полученные из внешнего источника. Однако метод `YAML.safe_load`

допускает загрузку не всех классов, поэтому его можно без опаски использовать для загрузки данных из ненадежных источников.

YAML позволяет и многое другое, о чем мы не можем здесь рассказать. Дополнительную информацию см. документацию по библиотеке Ruby stdlib или официальный сайт yaml.org.

10.2.5. Сохранение данных с помощью библиотеки JSON

Хотя акроним JSON означает «JavaScript Object Notation» (объектная нотация в JavaScript), в этом формате можно представить в понятном человеку виде данные, составленные из нескольких простых типов, имеющихся практически в каждом языке программирования.

В силу своей простоты и широкой распространенности JSON стал основным форматом для передачи данных между программами в Интернете.

Стандартная библиотека JSON используется почти так же, как YAML. Сериализовать можно только хэши, массивы, числа, строки, а также значения true, false и nil. Объект любого другого класса будет преобразован в строку:

```
require 'json'

data = {
       string: "Hi there",
       array: [1, 2, 3],
       boolean: true,
       object: Object.new
}

puts JSON.dump(data)
# Выводится: {"string":"Hi there","array":[1,2,3],
#             "boolean":true,"object":"#<Object:0x007fd61b890320>"}
```

Для преобразования объектов Ruby в формат JSON и обратно необходимо написать код по аналогии с рассмотренными выше методами marshal_dump и marshal_load. Существует соглашение, берущее начало в Ruby on Rails, о реализации метода as_json для преобразования объекта в типы данных, поддерживаемые JSON.

Вот как можно реализовать метод as_json для преобразования объекта Person (см. раздел 10.2.3) в формат JSON и обратно:

```
require 'json'
require 'time'

class Person
       # остальные методы не изменились...

       def as_json
          {name: @name, birthdate: @birthdate.iso8601, deposit: @deposit}
       end

       def self.from_json(json)
```

```
        data = JSON.parse(json)
        birthdate = Time.parse(data["birthdate"])
        new(data["name"], birthdate, data["deposit"])
      end
end

p1 = Person.new("Rudy", Time.now - (14 * 365 * 86400), 100)
p1.as_json    # {:name=>"Rudy", :deposit=>100,
              # :birthdate=>"2000-07-23T23:25:02-07:00"}

p2 = Person.from_json JSON.dump(p1.as_json)
[p2.name, p2.age, p2.balance] # ["Rudy", 14, 197.9931600356966]
```

Поскольку JSON не позволяет сохранять объекты типа `Time`, мы преобразуем их в строку методом `iso8601`, а затем – при создании нового объекта `Person` – восстанавливаем из этой строки объект `Time`.

Как уже было показано, библиотека JSON не поддерживает сериализацию объектов Ruby, в отличие от `Marshal` и YAML. А значит, можно без опаски вызывать метод `JSON.load` для загрузки JSON-документов из ненадежных источников, что позволяет использовать JSON для обмена данными с системами, которые вы не контролируете.

10.2.6. Работа с данными в формате CSV

CSV (значения, разделенные запятыми) – это формат, с которым вам доводилось сталкиваться, если вы работали с электронными таблицами или базами данных. К счастью, в Ruby включена библиотека CSV, написанная Джеймсом Эдвардом Греем III (James Edward Gray III).

Библиотека CSV разбирает или генерирует данные в формате CSV. О том, что же такое этот формат, нет единого мнения, но есть ряд общепринятых соглашений. По умолчанию в библиотеке действуют следующие правила:

- разделитель записей: CR + LF;
- разделитель полей: запятая (,);
- данные, содержащие символы CR, LF или запятую, заключаются в двойные кавычки;
- двойной кавычке внутри двойных кавычек должен предшествовать еще один символ двойной кавычки ("-> "");
- пустое поле в кавычках обозначает пустую строку (`данные,"",данные`);
- пустое поле без кавычек означает NULL (`данные,,данные`).

Эти правила можно изменить.

Начнем с создания файла. Чтобы вывести данные, разделенные запятыми, мы просто открываем файл для записи; метод `open` передает объект-писатель в блок. Затем с помощью оператора добавления мы добавляем массивы данных (при записи они преобразуются в формат CSV). Первая строка является заголовком.

```
require 'csv'

CSV.open("data.csv","w") do |wr|
  wr << ["name", "age", "salary"]
```

```
  wr << ["mark", "29", "34500"]
  wr << ["joe", "42", "32000"]
  wr << ["fred", "22", "22000"]
  wr << ["jake", "25", "24000"]
  wr << ["don", "32", "52000"]
end
```

В результате исполнения этого кода мы получаем такой файл data.csv:

```
"name","age","salary"
"mark",29,34500
"joe",42,32000
"fred",22,22000
"jake",25,24000
"don",32,52000
```

Другая программа может прочитать этот файл:

```
require 'csv'

CSV.open('data.csv', 'r') do |row|
  p row
end

# Выводится:
# ["name", "age", "salary"]
# ["mark", "29", "34500"]
# ["joe", "42", "32000"]
# ["fred", "22", "22000"]
# ["jake", "25", "24000"]
# ["don", "32", "52000"]
```

Этот фрагмент можно было бы записать и *без* блока, тогда метод open просто вернул бы объект-читатель. Затем можно было бы вызвать метод shift читателя (как если бы это был массив) для получения очередной строки. Но блочная форма мне представляется более естественной.

Каким бы способом ни был создан CSV-файл, его можно загрузить в Excel и другие электронные таблицы и там модифицировать.

10.2.7. SQLite3 как SQL-хранилище данных

SQLite3 – хранилище данных для тех, кто ценит ПО *с нулевым конфигурированием*. Это небольшая автономная программа, написанная на C, которая хранит всю базу данных в одном файле. Обычно она используется для небольших баз данных, но теоретически способна управиться с объемами в сотни гигабайтов.

Основное достоинство SQLite3 – тот факт, что интерфейс запросов в ней такой же, как в других полноценных СУБД, например MySQL или PostgreSQL. Это дает возможность начать проект с базой SQLite3, а затем перейти на внешний сервер БД, не переписывая код работы с базой.

Интерфейс между Ruby и SQLite3 довольно прямолинеен. Класс SQLite::Database позволяет открыть файл базы данных и выполнять SQL-запросы к ней. Ниже приведен простой пример кода.

```
require "sqlite3"

# Открыть новую базу данных
db = SQLite::Database.new("library.db")

# Создать таблицу для хранения книг
db.execute "create table books (
       title varchar(1024), author varchar(256) );"

# Вставить в таблицу записи
{
       "Robert Zubrin" => "The Case for Mars",
}.each do |author, title|
       db.execute "insert into books values (?, ?)", [title, author]
end

# Читать записи из таблицы с использованием блока
db.execute("select title,author from books") do |row|
       p row
end

# Закрыть базу данных
db.close

# Выводится:
# ["The Case for Mars", "Robert Zubrin"]
# ["Democracy in America", "Alexis de Tocqueville"]
```

Если блок не задан, то метод `execute` возвращает массив массивов, и все строки можно обойти:

```
rs = db.execute("select title,author from books")
rs.each {|row| p row }      # Тот же результат, что и выше
```

Методы библиотеки могут возбуждать различные исключения. Все они являются подклассами класса `SQLite::Exception`, так что легко перехватываются поодиночке или целой группой.

Хотя библиотека `sqlite` в достаточно мере функциональна, сама база SQLite *не полностью* реализует стандарт SQL92. Если вам нужна полная база данных SQL, обратитесь к PostgreSQL или MySQL (рассматривается в следующем разделе).

Дополнительные сведения о SQLite3 см. на сайте sqlite.org. Об интерфейсе к этой базе из Ruby см. документацию проекта в сети по адресу github.com/sparklemotion/sqlite3-ruby.

10.3. Подключение к внешним базам данных

Внешняя база данных работает в отдельном процессе, клиенты устанавливают с ней соединение через сокет. Затем клиент отправляет по соединению запросы, а база данных отвечает, возвращая результаты.

В базе данных могут храниться только простые типы, зато она обладает другими достоинствами: гибкостью, разделяемым доступом и быстродействием. Существуют библиотеки, позволяющие подключиться практически к любой существующей базе данных, в том числе к базам данных SQL, хранилищам ключей и значений и документным базам.

10.3.1. Подключение к базе данных MySQL

Интерфейс Ruby с MySQL – один из самых стабильных и полнофункциональных среди всех интерфейсов с базами данных. После установки Ruby и MySQL установите gem-пакет `mysql2` командой `gem install` или `bundle install`.

Для использования gem-пакета нужно выполнить три шага. Во-первых, загрузить его в свой скрипт, затем установить соединение с базой данных и, наконец, начать работать с таблицами. Для установления соединения следует задать обычные параметры: имя хоста, имя пользователя, пароль, имя базы данных и т.д.

Модуль включает два класса (`Mysql2::Client` и `Mysql2::Result`), описанных в файле README. Ниже мы опишем несколько наиболее употребительных методов, а дополнительную информацию всегда можно найти в документации.

Метод класса `Mysql2::Client.new` принимает несколько строковых параметров, по умолчанию равных `nil`, и возвращает объект-клиент. Наиболее полезны параметры `host`, `username`, `password`, `port` и `database`. Метод клиента `query` используется для взаимодействия с базой данных:

```
require 'mysql2'

client = Mysql2::Client.new(
   :host => "localhost",
   :username => "root"
)

# Создать базу данных
client.query("CREATE DATABASE mailing_list")

# Воспользоваться базой данных
client.query("USE mailing_list")
```

Если база данных уже существует, команду `USE` можно опустить, указав параметр `database`:

```
# Создать таблицу для хранения имен и почтовых адресов
client.query("CREATE TABLE members (
      name varchar(1024), email varchar(1024))")

# Вставить строки с описанием двух подписчиков на список рассылки
client.query <<-SQL
      INSERT INTO members VALUES
      ('John Doe', 'jdoe@rubynewbie.com'),
      ('Fred Smith', 'smithf@rubyexpert.com')
SQL
```

Вставляя данные, не забывайте экранировать значения! Злонамеренный пользователь может отправить специально сконструированные данные, которые уничтожат базу данных или даже предоставят ему права администратора. Чтобы предотвратить это, перед вставкой передавайте данные методу escape.

```
escaped = ["Bob Howard", "bofh@laundry.gov.uk"].map do |value|
      "'#{client.escape(value)}'"
end
client.query("INSERT INTO members VALUES (#{escaped.join(",")})")
```

Если в ответ на запрос был отправлен результат, то возвращается экземпляр класса Mysql2::Result, к которому подмешан модуль Enumerable. Поэтому строки результата можно перебирать методом each или любым другим, входящим в состав Enumerable.

```
# Запросить данные и вернуть каждую строку в виде хэша
client.query("SELECT * from members").each do |member|
   puts "Name: #{member["name"]}, Email: #{member["email"]}"
end
# Выводится:
# Name: John Doe, Email: jdoe@rubynewbie.com
# Name: Fred Smith, Email: smithf@rubyexpert.com
# Name: Bob Howard, Email: bofh@laundry.gov.uk
```

Поддерживаются все типы данных MySQL, а строки, числа, время и другие типы автоматически преобразуются в объекты соответствующих классов Ruby.

Объект Result отдает каждую строку результата в виде хэша. Ключи хэша по умолчанию являются строками, но можно сделать их символами, если передать методу query параметр :symbolize_keys => true.

Из полезных методов Result назовем еще count (количество возвращенных строк) и fields (массив имен столбцов результата). Метод each может также отдавать строки результата в виде массива значений, если передать ему параметр :as => :array.

```
# Запросить данные и вернуть каждую строку в виде массива
result = client.query("SELECT * FROM members")
puts "#{result.count} records"
puts result.fields.join(" - ")
result.each(:as => :array){|m| puts m.join(" - ") }
# Выводится:
# 3 records
# name - email
# John Doe - jdoe@rubynewbie.com
# Fred Smith - smithf@rubyexpert.com
# Bob Howard - bofh@laundry.gov.uk
```

Имена существующих баз данных можно получить, воспользовавшись специфичной для MySQL командой SHOW DATABASES:

```
client.query("SHOW DATABASES").to_a(:as => :array).flatten
# ["information_schema", "mailing_list", "mysql",
#  "performance_schema", "test"]
```

Наконец, для закрытия соединения с базой данных служит метод close:

```
# Закрыть соединение с базой данных
client.close
```

Как обычно, мы рассмотрели лишь малую часть доступных средств. Дополнительные сведения можно найти на сайте MySQL по адресу mysql.com и на сайте gem-пакет mysql2 по адресу github.com/brianmario/mysql2.

10.3.2. Подключение к базе данных PostreSQL

Gem-пакет pg предоставляет интерфейс для подключения к базе данных PostgreSQL, он устанавливается командой gem install или bundle install после того, как PostgreSQL и Ruby уже установлены.

Как и для других адаптеров баз данных, нужно загрузить модуль, установить соединение с базой данных и начать работу с таблицами. Для установления соединения в классе PG имеется метод connect.

У объекта соединения есть метод exec (синоним query) для отправки SQL-запросов базе данных:

```
require 'pg'

# Создать базу данных
PG.connect.exec("CREATE DATABASE pets")

# Открыть соединение с созданной базой данных
conn = PG.connect(dbname: "pets")
```

Отметим, что в отличие от MySQL, для смены базы данных необходимо создать новое соединение, а не выполнить запрос.

Отправка запроса с помощью библиотеки pg делается так же, как в любой другой библиотеке SQL:

```
# Создать таблицу и вставить в нее данные
conn.exec("CREATE TABLE pets (name varchar(255),
   species varchar(255), birthday date)")
conn.exec <<-SQL
   INSERT INTO pets VALUES
      ('Spooky', 'cat', '2008-10-03'),
      ('Spot', 'dog', '2004-06-10')
SQL
```

Экранирование вставляемых значений существенно упрощается благодаря методу exec_params, который умеет экранировать переданные ему значения:

```
# Экранировать данные при вставке
require 'date'
conn.exec_params("INSERT INTO pets VALUES ($1, $2, $3)",
   ['Maru', 'cat', Date.today.to_s])
```

Но в отличие от gem-пакета mysql2 пакет pg не преобразует результаты запроса в объекты соответствующих классов Ruby. Все результаты возвращаются в виде строк, при необходимости их нужно будет преобразовать самостоятель-

но. Ниже мы вызываем метод Date.parse для каждого присутствующего в результате поля даты:

```
# Запросить данные
res = conn.query("SELECT * FROM pets")
puts "There are #{res.count} pets."
res.each do |pet|
   name = pet["name"]
   age = (Date.today - Date.parse(pet["birthday"]))/365
   species = pet["species"]
   puts "#{name} is a #{age.floor}-year-old #{species}."
end

# Выводится:
# There are 3 pets.
# Spooky is a 5-year-old cat.
# Spot is a 10-year-old dog.
# Maru is a 0-year-old cat.
```

Метод each всегда отдает строки результата в виде хэшей, но экземпляр класса PG::Result, возвращенный после выполнения запроса, может предоставлять значения в виде массива строк, каждая из которых представляет собой массив. Из других полезных методов отметим еще count (количество возвращенных строк) и fields (массив имен столбцов результата).

Как обычно, дополнительные сведения и документацию можно найти на сайте PostgreSQL по адресу postgresql.org и на сайте gem-пакета pg по адресу bitbucket.org/ged/ruby-pg.

10.3.3. Объектно-реляционные отображения (ORM)

Традиционная реляционная база данных прекрасно справляется со своими задачами. Она эффективно выполняет произвольные запросы, о которых заранее ничего не знает. Но эта модель плохо уживается с объектной ориентированностью.

Повсеместная распространенность обеих моделей (РСУБД и ООП) и «несогласованный импеданс» между ними побудил многих людей попытаться перебросить мост. Этот программный мост получил название «объектно-реляционное отображение» (Object-Relational Mapper – ORM).

К этой задаче существуют разные подходы. У каждого есть свои достоинства и недостатки. Ниже мы бегло рассмотрим класс ActiveRecord – пожалуй, самый известный из всех.

Библиотека ActiveRecord для Ruby названа в честь предложенного Мартином Фаулером (Martin Fowler) паттерна проектирования «Active Record» (активная запись). Смысл его в том, что активная запись «обертывает строку таблицы или представления, инкапсулирует доступ к базе данных и наделяет данные логикой, присущей предметной области» (см. книгу Martin Fowler «Patterns of Enterprise Application Architecture», Addison Wesley, 2003).

В gem-пакете activerecord это делается с помощью отдельного класса Ruby для каждой таблицы базы данных. Он умеет выполнять некоторые запросы без привлечения SQL, представляет каждую строку в виде экземпляра класса и позволяет добавлять предметную логику. Имена классов соответствуют именам таблиц, и все они наследуют классу ActiveRecord::Base. Для всех классов нужно только одно соединение с базой данных.

Вот небольшой пример, демонстрирующий весь механизм в действии:

```
require 'active_record'

class Pet < ActiveRecord::Base
end

ActiveRecord::Base.establish_connection(
   :adapter => "postgresql", :database => "pets")

snoopy = Pet.new(name: "Snoopy", species: "dog")
snoopy.birthday = Date.new(1950, 10, 4)
snoopy.save

p Pet.all.map {|pet| pet.birthday.to_s }
```

Как видите, класс умеет выводить имена таблиц, предоставляет акцессоры атрибутов, основанные на именах столбцов, и выполняет запрос в объектно-ориентированном стиле. Кроме того, он предлагает один и тот же API, включающий преобразования типов данных, для любой базы данных SQL.

API библиотеки activerecord богат и сложен, о нем написаны целые книги. Для получения дополнительной информации обратитесь к любому справочному руководству.

10.3.4. Подключение к хранилищу данных Redis

Хотя Redis не является базой данных (ни SQL, ни какой-либо другой), эта программа в последнее годы стала очень популярна. Redis – это хранилище ключей и значений, в котором значения могут быть строками, хэшами, массивами, множествами и отсортированными множествами.

На роль постоянного хранилища данных Redis вряд ли подходит, но является отличным средством кэширования. В нем очень удобно сохранять результаты дорогостоящих вычислений, произведенных над постоянно хранящимися данными.

Для подключения к серверу Redis применяется gem-пакет redis. Если сервер работает на той же машине, что и клиент, никакие параметры вообще не нужны:

```
require 'redis'
r = Redis.new
```

Один из полезных типов хранимых данных – множество. Redis умеет быстро вычислять разность, пересечение и объединение нескольких множеств. При этом результат либо возвращается, либо сохраняется с другим ключом.

```
# Сохранить клички домашних животных в множестве
r.sadd("pets", "spooky")
```

```
# Получить элементы множества
r.smembers("pets") # ["spooky"]
```

Тип хэш позволяет читать несколько полей и записывать их с общим ключом, по одному или все сразу.

```
# Использовать кличку домашнего животного как ключ хэша других его
# атрибутов
r.hset("spooky", "species", "cat")
r.hset("spooky", "birthday", "2008-10-03")

# Получить одно значение из хэша
Get a single hash value
r.hget("spooky", "species") # "cat"

# Получить весь хэш
r.hgetall("spooky") # {"species"=>"cat", "birthday"=>"2008-10-03"}
```

Отсортированное множество ассоциирует с каждым значением числовую оценку и предоставляет специальные команды для увеличения и уменьшения этих оценок, а также для извлечения значения в порядке следования оценок.

```
# Использовать отсортированное множество для хранения домашних животных,
# отсортированных по весу
r.zadd("pet_weights", 6, "spooky")
r.zadd("pet_weights", 12, "spot")
r.zadd("pet_weights", 2, "maru")

# Выбрать первое значение из множества, отсортированного в порядке убывания
r.zrevrange("pet_weights", 0, 0) # => ["spot"]
```

С помощью различных предоставляемых Redis типов данных возможны и другие сложные и оптимизированные операции. Полная документация по каждому типу данных и каждой команде имеется на сайте Redis по адресу redis.io.

10.4. Заключение

В этой главе был приведен обзор ввода-вывода в Ruby. Мы рассмотрели сам класс `IO` и его подкласс `File`, а также связанные с ними классы, например, `Dir` и `Pathname`. Мы познакомились с некоторыми полезными приемами манипулирования объектами `IO` и файлами.

Мы также уделили внимание вопросам хранения данных на более высоком уровне, точнее, на внешних носителях в виде сериализованных объектов. Наконец, мы дали краткий обзор решений, которые Ruby предлагает для интерфейса с настоящими базами данных, а, кроме того, познакомились с некоторыми объектно-ориентированными подходами, упрощающими взаимодействие с ними.

Ниже мы еще вернемся к вводу-выводу в контексте сокетов и сетевого программирования. Но предварительно рассмотрим некоторые другие темы.

Глава 11. ООП и динамические механизмы в Ruby

Ни сердцу, ни сознанью беглый взгляд
Не может дать о виденном отчет.
Траве, цветам и птицам он не рад,
И в нем ничто подолгу не живет.

– Уильям Шекспир «Сонет 113»

Это необычная глава. В большинстве других глав рассматривается какой-то конкретный аспект, например строки или файлы. Но не в этой. Если расположить «пространство задачи» по одной оси системы координат, то эта глава окажется на другой оси, то есть содержит по кусочку из всех других областей. Связано это с тем, что объектно-ориентированное программирование и динамичность сами по себе являются не задачами, а парадигмами, которые могут быть применены к решению любой задачи, будь то системное администрирование, низкоуровневое сетевое программирование или разработка приложений для веб.

Поэтому значительная часть материала этой главы должна быть уже знакома любому программисту, знающему Ruby. На самом деле, все остальное в этой книге не имеет смысла без понимания изложенных здесь основ. Например, любой программист на Ruby знает, как создать подкласс.

Возникает вопрос: что включить, а без чего можно обойтись? Знает ли любой программист о методе `extend`? А о методе `instance_eval`? Очевидное одному может оказаться откровением для другого.

Мы решили отдать предпочтение полноте. Мы включили некоторые экзотические вещи, которые можно делать с помощью динамического ООП в Ruby, но не забыли и о рутинных задачах на случай, если кто-то не знаком с ними. Мы спустились до самого простого уровня, поскольку многие по-разному представляют себе, где кончается «средний» уровень. И попытались дать кое-какую дополнительную информацию даже при изложении самых базовых вопросов, чтобы оправдать включение их в эту главу. С другой стороны, тем, которые раскрываются в других частях книги, мы здесь не касались.

Еще два замечания. Во-первых, ничего магического в динамическом ООП нет. Объектная ориентированность языка Ruby и его динамическая природа прекрасно уживаются между собой, но неотъемлемой связи между ними нет. Мы расска-

зываем о том и другом в одной главе только для удобства. Во-вторых, мы затрагиваем кое-какие особенности языка, которые, строго говоря, не относятся ни к одной из двух заявленных тем. Если хотите, считайте это мелким обманом. Но надо же было поместить их *куда-то*.

11.1. Рутинные объектно-ориентированные задачи

Если вы вообще не знакомы с ООП, то эта глава вас ничему не научит. А если вы понимаете, что такое ООП в языке Ruby, то, наверное, ее и читать не стоит. Если понятия ООП не слишком свежи в памяти, просмотрите главу 1, где мы приводим их краткий обзор (или обратитесь к другой книге).

С другой стороны, большая часть материала в этой главе изложена в виде руководства и довольно элементарна. Поэтому она будет полезна начинающему и менее интересна для программиста на Ruby среднего уровня. Эта книга организована как устройство ввода-вывода с произвольной выборкой, так что можете свободно пропускать те части, которые вас не интересуют.

11.1.1. Применение нескольких конструкторов

В Ruby нет «настоящих» конструкторов, как в C++ или в Java. Сама идея, конечно, никуда не делась, поскольку объекты необходимо создавать и инициализировать, но реализация выглядит иначе.

В Ruby каждый класс имеет метод класса `new`, который вызывается для создания новых объектов. Метод `new` вызывает специальный определяемый пользователем метод `initialize`, который инициализирует атрибуты объекта, после чего `new` возвращает ссылку на новый объект.

А если мы хотим иметь несколько конструкторов? Как быть в этом случае?

Ничто не мешает завести дополнительные методы класса, которые возвращают новые объекты. В листинге 11.1 приведен искусственный пример класса для представления прямоугольника, у которого есть две длины сторон и три значения цвета. Мы создали дополнительные методы класса, предполагающие определенные умолчания для каждого параметра. (Например, квадрат – это прямоугольник, у которого все стороны равны.)

Листинг 11.1. Несколько конструкторов

```
class ColoredRectangle

   def initialize(r, g, b, s1, s2)
      @r, @g, @b, @s1, @s2 = r, g, b, s1, s2
   end

   def self.white_rect(s1, s2)
      new(0xff, 0xff, 0xff, s1, s2)
   end

   def self.gray_rect(s1, s2)
      new(0x88, 0x88, 0x88, s1, s2)
   end
```

```
    def self.colored_square(r, g, b, s)
        new(r, g, b, s, s)
    end

    def self.red_square(s)
        new(0xff, 0, 0, s, s)
    end

    def inspect
        "#@r #@g #@b #@s1 #@s2"
    end
  end

a = ColoredRectangle.new(0x88, 0xaa, 0xff, 20, 30)
b = ColoredRectangle.white_rect(15,25)
c = ColoredRectangle.red_square(40)
```

Таким образом, можно определить любое число методов, создающих объекты по различным спецификациям. Вопрос о том, уместен ли здесь термин «конструктор», мы оставим «языковым адвокатам».

11.1.2. Создание атрибутов экземпляра

Имени атрибута экземпляра в Ruby всегда предшествует знак @. Это обычная переменная в том смысле, что она начинает существовать после первого присваивания.

В ОО-языках часто создаются методы для доступа к атрибутам, чтобы обеспечить сокрытие данных. Мы хотим контролировать доступ к «внутренностям» объекта извне. Обычно для этой цели применяются методы чтения и установки (getter и setter), хотя в Ruby эта терминология не используется. Они просто читают (get) или устанавливают (set) значение атрибута.

Можно, конечно, запрограммировать такие функции «вручную», как показано ниже:

```
class Person

  def name
    @name
  end

  def name=(x)
    @name = x
  end

  def age
    @age
  end

  # ...

end
```

Но Ruby предоставляет более короткий способ. Метод attr принимает в качестве параметра символ и создает соответствующий атрибут. Кроме того, он создает одноименный метод чтения, а если необязательный второй параметр равен true, то и метод установки.

```
class Person
  attr :name, true  # Создаются @name, name, name=
  attr :age         # Создаются @age, age
end
```

Методы attr_reader, attr_writer и attr_accessor принимают в качестве параметров произвольное число символов. Первый создает только «методы чтения» (для получения значения атрибута); второй – только «методы установки», а третий – то и другое. Пример:

```
class SomeClass
  attr_reader :a1, :a2    # Создаются @a1, a1, @a2, a2
  attr_writer :b1, :b2    # Создаются @b1, b1=, @b2, b2=
  attr_accessor :c1, :c2  # Создаются @c1, c1, c1=, @c2, c2, c2=
  # ...

  def initialize
    @a1 = 237
    @a2 = nil
  end

end
```

Напомним, что для выполнения присваивания атрибуту необходимо указывать вызывающий объект, а внутри метода нужно в качестве такого объекта указывать self.

```
def update_ones(a1, b1, c1)
   self.a1 = a1
   self.b1 = b1
   self.c1 = c1
end
```

Если не указать self, то Ruby просто создаст внутри метода локальную переменную с указанным именем, а это совсем не то, что нам нужно.

Существует специальный метод, позволяющий узнать, определена ли уже переменная экземпляра:

```
obj = SomeClass.new
obj.instance_variable_defined?(:b2)   # true
obj.instance_variable_defined?(:d2)   # true
```

Повинуясь нашему желанию использовать eval только в случае острой необходимости, Ruby предоставляет также методы для чтения или присваивания значения переменной, имя которой задано в виде строки:

```
class MyClass
   attr_reader :alpha, :beta
```

```
    def initialize(a, b, g)
        @alpha, @beta, @gamma = a, b, g
    end
end

x = MyClass.new(10, 11, 12)

x.instance_variable_set("@alpha", 234)
p x.alpha                                  # 234

x.instance_variable_set("@gamma", 345)     # 345
v = x.instance_variable_get("@gamma")      # 345
```

Прежде всего, обратите внимание на знак @ в имени переменной: если его опустить, произойдет ошибка. Если это кажется вам интуитивно неочевидным, вспомните, что методы типа attr_accessor на самом деле принимают в качестве имени метода символ, поэтому-то в них и можно опускать знак @.

Может возникнуть вопрос, а не нарушает ли само существование таких методов принцип инкапсуляции? Нет, не нарушает.

Да, эти методы потенциально могут быть опасны. Применять их надо не с легким сердцем, а тщательно обдумав свои действия. Но не видя, как конкретно используется метод, нельзя сказать, нарушает он инкапсуляцию или нет.

Если он является частью хорошего дизайна, то все в порядке. Если же цель – обойти ограничения плохого дизайна, то дела плохи. Ruby сознательно дает доступ к внутренней структуре объекта тем, кому это действительно нужно; отличительный признак ответственного программиста – не использовать эту свободу во зло.

11.1.3. Более сложные конструкторы

По мере усложнения объектов у них появляется все больше атрибутов, которые необходимо инициализировать в момент создания. Соответствующий конструктор может оказаться длинным и запутанным, его параметры даже не будут помещаться на одной строке.

Чтобы справиться со сложностью, можно передать методу initialize блок (листинг 11.2). Тогда инициализация объекта выполняется в процессе вычисления этого блока.

Листинг 11.2. «Хитрый» конструктор

```
class PersonalComputer
  attr_accessor :manufacturer,
                :model, :processor, :clock,
                :ram, :disk, :monitor,
                :colors, :vres, :hres, :net

  def initialize(&block)
```

```
    instance_eval &block
  end

  # Прочие методы...
end

desktop = PersonalComputer.new do
  self.manufacturer = "Acme"
  self.model = "THX-1138"
  self.processor = "986"
  self.clock = 9.6          # ГГц
  self.ram = 16             # ГБ
  self.disk = 20            # ТБ
  self.monitor = 25         # дюймы
  self.colors = 16777216
  self.vres = 1280
  self.hres = 1600
  self.net = "T3"
end

p desktop
```

Отметим, что мы пользуемся акцессорами для присваивания атрибутам значений. Кроме того, в теле блока можно было бы сделать все что угодно. Например, вычислить одни поля по значениям других.

А если мы не хотим, чтобы у объекта были акцессоры для всех атрибутов? В таком случае можно было бы воспользоваться методом `instance_eval` и сделать методы установки защищенными – `protected`. Это предотвратит «случайное» присваивание значения атрибуту извне объекта.

```
class Library
   attr_reader :shelves

   def initialize(&block)
      instance_eval(&block)
   end

   protected

      attr_writer :shelves
end

branch = Library.new do
   self.shelves = 10
end

branch.shelves = 20
# NoMethodError: protected method `shelves=' called
branch.shelves        # 10
```

Но даже при использовании instance_eval необходимо явно вызывать метод установки от имени self. Метод установки всегда вызывается от имени явно указанного объекта, чтобы отличить его от присваивания локальной переменной.

11.1.4. Создание атрибутов и методов уровня класса

Метод или атрибут не всегда ассоциируются с конкретным экземпляром класса, они могут принадлежать самому классу. Типичным примером метода класса может служить new, он вызывается для создания новых экземпляров, а потому не может принадлежать никакому конкретному экземпляру.

Мы можем определять собственные методы класса, как показано в разделе 11.1.1. Конечно, их функциональность не ограничивается конструированием, они могут выполнять любые операции, имеющие смысл именно на уровне класса.

В следующем далеко не полном фрагменте предполагается, что мы создаем класс для проигрывания звуковых файлов. Метод play естественно реализовать как метод экземпляра, ведь можно создать много объектов, каждый из которых будет проигрывать свой файл. Но у метода detect_hardware контекст более широкий; в зависимости от реализации может оказаться, что создавать какие-либо объекты вообще не имеет смысла, если этот метод возвращает ошибку. Следовательно, его контекст – вся среда воспроизведения звука, а не конкретный звуковой файл.

```
class SoundPlayer
  MAX_SAMPLE = 192

  def self.detect_hardware
    # ...
  end

  def play
    # ...
  end

end
```

Есть еще один способ объявить этот метод класса. В следующем фрагменте делается практически то же самое:

```
class SoundPlayer
  MAX_SAMPLE = 192

  def play
    # ...
  end

end

def SoundPlayer.detect_hardware
  # ...
end
```

Единственная разница касается использования объявленных в классе констант. Если метод класса объявлен вне объявления самого класса, то эти константы оказываются вне области видимости. Например, в первом фрагменте метод detect_hardware может напрямую обращаться к константе MAX_SAMPLE, а во втором придется пользоваться нотацией SoundPlayer::MAX_SAMPLE.

Не удивительно, что, помимо методов класса, есть еще и переменные класса. Их имена начинаются с двух знаков @, а областью видимости является весь класс, а не конкретный его экземпляр.

Традиционный пример использования переменных класса – подсчет числа экземпляров этого класса. Но они могут применяться всегда, когда информации имеет смысл в контексте класса в целом, а не отдельного объекта. Другой пример приведен в листинге 11.3.

Листинг 11.3. Переменные и методы класса

```
class Metal
  @@current_temp = 70

  attr_accessor :atomic_number

  def self.current_temp=(x)
    @@current_temp = x
  end

  def self.current_temp
    @@current_temp
  end

  def liquid?
    @@current_temp >= @melting
  end

  def initialize(atnum, melt)
    @atomic_number = atnum
    @melting = melt
  end

end

aluminum = Metal.new(13, 1236)
copper = Metal.new(29, 1982)
gold = Metal.new(79, 1948)

Metal.current_temp = 1600

puts aluminum.liquid?        # true
```

```
puts copper.liquid?          # false
puts gold.liquid?            # false

Metal.current_temp = 2100

puts aluminum.liquid?        # true
puts copper.liquid?          # true
puts gold.liquid?            # true
```

Здесь переменная класса инициализируется до того, как впервые используется в методе класса. Отметим также, что мы можем обратиться к переменной класса из метода экземпляра, но обратиться к переменной экземпляра из метода класса нельзя. Немного подумав, вы поймете, что так и должно быть.

А если попытаться, что произойдет? Что если мы попробуем напечатать атрибут `@atomic_number` из метода `Metal.current_temp`? Обнаружится, что переменная вроде бы существует – никакой ошибки не возникает, – но имеет значение `nil`. В чем дело?

В том, что на самом деле мы обращаемся вовсе не к переменной экземпляра класса `Metal`, а к переменной экземпляра класса `Class`. (Напомним, что в Ruby `Class` – это класс!)

Мы столкнулись с *переменной экземпляра класса* (термин заимствован из языка Smalltalk). Дополнительные замечания на эту тему см. в разделе 11.2.4 «Создание параметрических классов».

В листинге 11.4 иллюстрируются все аспекты этой ситуации.

Листинг 11.4. Данные класса и экземпляра

```
class MyClass

  SOME_CONST = "alpha"       # Константа уровня класса

  @@var = "beta"             # Переменная класса
  @var = "gamma"             # Переменная экземпляра класса

  def initialize
    @var = "delta"           # Переменная экземпляра
  end

  def mymethod
    puts SOME_CONST          # (константа класса)
    puts @@var               # (переменная класса)
    puts @var                # (переменная экземпляра)
  end

  def self.classmethi
    puts SOME_CONST          # (константа класса)
    puts @@var               # (переменная класса)
```

```
    puts @var                # (переменная экземпляра класса)
  end

end

def MyClass.classmeth2
  puts MyClass::SOME_CONST   # (константа класса)
  # puts @@var               # ошибка: вне области видимости
  puts @var                  # (переменная экземпляра класса)
end
myobj = MyClass.new
MyClass.classmeth1           # alpha, beta, gamma
MyClass.classmeth2           # alpha, gamma
myobj.mymethod               # alpha, beta, delta
```

Следует еще сказать, что метод класса можно сделать закрытым, воспользовавшись методом `private_class_method`. Это аналог метода `private` на уровне экземпляра.

11.1.5. Наследование суперклассу

Можно унаследовать класс, воспользовавшись символом <:

```
class Boojum < Snark
  # ...
end
```

Это объявление говорит, что класс `Boojum` является подклассом класса `Snark` или – что то же самое – класс `Snark` является суперклассом класса `Boojum`. Всем известно, что каждый буюм является снарком, но не каждый снарк – буюм.

Ясно, что цель наследования – расширить или специализировать функциональность. Мы хотим получить из общего нечто более специфическое.

Попутно отметим, что во многих языках, например в C++, допускается множественное наследование (МН). В Ruby (как и в Java, и в некоторых других языках) множественного наследования нет, но наличие классов-примесей компенсирует его отсутствие, см. раздел 11.1.12.

Рассмотрим несколько более реалистичный пример. Пусть есть класс `Person` (человек), а мы хотим создать производный от него класс `Student` (студент).

Определим класс `Person` следующим образом:

```
class Person

  attr_accessor :name, :age, :sex

  def initialize(name, age, sex)
    @name, @age, @sex = name, age, sex
  end

  # ...

end
```

А класс Student – так:

```
class Student < Person

  attr_accessor :idnum, :hours

  def initialize(name, age, sex, idnum, hours)
    super(name, age, sex)
    @idnum = idnum
    @hours = hours
  end
  # ...

end

# Создать два объекта
a = Person.new("Dave Bowman", 37, "m")
b = Student.new("Franklin Poole", 36, "m", "000-13-5031", 24)
```

Посмотрим внимательно, что здесь сделано. Что за super, вызываемый из метода initialize класса Student? Это просто вызов соответствующего метода родительского класса. А раз так, то ему передается три параметра (хотя наш собственный метод initialize принимает пять).

Не всегда необходимо использовать слово super подобным образом, но часто это удобно. В конце концов, атрибуты любого класса образуют надмножество множества атрибутов его родительского класса, так почему не воспользоваться для их инициализации конструктором родительского класса?

Если говорить об истинном смысле наследования, то оно, безусловно, описывает отношение «является» . Студент ***является*** человеком, как и следовало ожидать. Сделаем еще три замечания.

- Каждый атрибут (и метод) родительского класса отражается в его потомках. Если в классе Person есть атрибут height, то класс Student унаследует его, а если родитель имеет метод say_hello, то такой метод будет и у потомка.
- Потомок может иметь дополнительные атрибуты и методы, мы это только что видели. Поэтому создание подкласса часто еще называют *расширением* суперкласса.
- Потомок может переопределять любые атрибуты и методы своего родителя.

Последнее замечание подводит нас к вопросу о том, как разрешается вызов метода. Откуда я знаю, вызывается ли метод конкретного класса или его суперкласса?

Краткий ответ таков: не знаю и не интересуюсь. Если вызывается некий метод от имени объекта класса Student, то будет вызван метод, определенный в этом классе, если он существует. А если нет, вызывается метод суперкласса и так далее вверх по иерархии наследования.

Мы говорим «и так далее», потому что у каждого класса (кроме BasicObject) есть суперкласс. Попутно отметим, что BasicObject – «чистая доска» в мире объектов, у него даже меньше методов, чем у Object.

А что если мы хотим вызвать метод суперкласса, но не из соответствующего метода подкласса? Можно сначала создать в подклассе синоним:

```
class Student    # повторное открытие класса
  # Предполагается, что в классе Person есть метод say_hello...
  alias :say_hi :say_hello

  def say_hello
    puts "Привет."
  end

  def formal_greeting
    # Поприветствовать так, как принято в суперклассе.
    say_hi
  end

end
```

У наследования есть разные тонкости, которых мы здесь касаться не будем. Общий принцип мы изложили, но не пропустите следующий раздел.

11.1.6. Опрос класса объекта

Часто возникает вопрос: «Что это за объект? Как он соотносится с данным классом?» Есть много способов получить тот или иной ответ.

Во-первых, метод экземпляра `class` всегда возвращает класс объекта. Применявшийся ранее синоним `type` объявлен устаревшим.

```
s = "Hello"
n = 237
sc = s.class    # String
nc = n.class    # Fixnum
```

Не думайте, будто методы `class` или `type` возвращают строку, представляющую имя класса. На самом деле возвращается экземпляр класса `Class`! При желании мы могли бы вызвать метод класса, определенный в этом типе, как если бы это был метод экземпляра класса `Class` (каковым он в действительности и является).

```
s2 = "some string"
var = s2.class             # String
my_str = var.new("Hi...")  # Новая строка
```

Можно сравнить такую переменную с константным именем класса и выяснить, равны ли они; можно даже использовать переменную в роли суперкласса и определить на ее основе подкласс! Запутались? Просто помните, что в Ruby `Class` – это объект, а `Object` – это класс.

Иногда нужно сравнить объект с классом, чтобы понять принадлежит ли данный объект указанному классу. Для этого служит метод `instance_of?`, например:

```
puts (5.instance_of? Fixnum)        # true
puts ("XYZZY".instance_of? Fixnum)  # false
puts ("PLUGH".instance_of? String)  # true
```

А если нужно принять во внимание еще и отношение наследования? К вашим услугам метод kind_of? (похожий на instance_of?). У него есть синоним is_a?, что вполне естественно, ибо мы описываем классическое отношение «является».

```
n = 9876543210
flag1 = n.instance_of? Bignum     # true
flag2 = n.kind_of? Bignum         # true
flag3 = n.is_a? Bignum            # true
flag3 = n.is_a? Integer           # true
flag4 = n.is_a? Numeric           # true
flag5 = n.is_a? Object            # true
flag6 = n.is_a? String            # false
flag7 = n.is_a? Array             # false
```

Ясно, что метод kind_of или is_a? более общий, чем instance_of?. Например, всякая собака – млекопитающее, но не всякое млекопитающее – собака.

Для новичков в Ruby приготовлен один сюрприз. Любой модуль, подмешиваемый в класс, становится субъектом отношения «является» для экземпляров этого класса. Например, в класс Array подмешан модуль Enumerable; это означает, что всякий массив является перечисляемым объектом.

```
x = [1, 2, 3]
flag8 = x.kind_of? Enumerable     # true
flag9 = x.is_a? Enumerable        # true
```

Для сравнения двух классов можно пользоваться также операторами сравнения. Интуитивно очевидно, что оператор «меньше» обозначает наследование суперклассу.

```
flag1 = Integer < Numeric       # true
flag2 = Integer < Object        # true
flag3 = Object == Array         # false
flag4 = IO >= File              # true
flag5 = Float < Integer         # nil
```

В любом классе обычно определен оператор «тройного равенства» ===. Выражение class === instance истинно, если экземпляр instance принадлежит классу class. Этот оператор еще называют *оператором ветвящегося равенства*, потому что он неявно используется в предложении case. Он дает возможность принимать решение в зависимости от класса выражения.

Дополнительную информацию о нем см. в разделе 11.1.7.

Упомянем еще метод respond_to. Он используется, когда нам безразлично, какому классу принадлежит объект, но мы хотим знать, реализует ли он конкретный метод. Это рудиментарный вид информации о типе. (Вообще-то можно сказать, что это самая важная информация о типе.) Методу respond_to передается символ и необязательный флаг, который говорит, нужно ли включать в рассмотрение также и закрытые методы.

```
# Искать открытые методы
if wumpus.respond_to?(:bite)
  puts "У него есть зубы!"
```

```
else
  puts "Давай-ка подразним его."
end

# Необязательный второй параметр позволяет
# просматривать и закрытые методы.

if woozle.respond_to?(:bite,true)
  puts "Вузлы кусаются!"
else
  puts "Ага, это не кусающийся вузл."
end
```

Иногда нужно знать, является ли данный класс непосредственным родителем объекта или класса. Ответ на этот вопрос дает метод `superclass` класса `Class`.

```
array_parent = Array.superclass        # Object
fn_parent = 237.class.superclass       # Integer
obj_parent = Object.superclass         # BasicObject
basic_parent = BasicObject.superclass  # nil
```

У любого класса, кроме `BasicObject`, есть суперкласс.

11.1.7. Проверка объектов на равенство

Все животные равны,
но некоторые равнее других.

– Джордж Оруэлл «Скотный двор»

При написании своих классов желательно, чтобы семантика типичных операций была такой же, как у встроенных в Ruby классов. Например, если объекты класса можно упорядочивать, то имеет смысл реализовать метод `<=>` и подмешать модуль `Comparable`. Тогда к объектам вашего класса будут применимы все обычные операции сравнения.

Однако картина перестает быть такой однозначной, когда дело доходит до сравнения объектов на равенство. В Ruby объекты реализуют пять разных методов для проверки на равенство. И в ваших классах придется реализовать хотя бы некоторые из них, поэтому рассмотрим этот вопрос подробнее.

Самым главным является метод `equal?` (унаследованный от класса `Object`), он возвращает `true`, если вызывающий объект и параметр имеют один и тот же идентификатор объекта. Это фундаментальный аспект семантики объектов, поэтому переопределять его не следует.

Самым распространенным способом проверки на равенства является старый добрый оператор `==`, который сравнивает значения вызывающего объекта и аргумента. Наверно, интуитивно это наиболее очевидный способ.

Следующим в шкале абстракции стоит метод eql? - тоже часть класса Object. (На самом деле, метод eql? реализован в модуле Kernel, который подмешивается в Object.) Как и оператор ==, этот метод сравнивает значения вызывающего объекта и аргумента, но несколько более строго. Например, разные числовые объекты при сравнении с помощью == приводятся к общему типу, но метод eql? никогда не считает объекты разных типов равными.

```
flag1 = (1 == 1.0)      # true
flag2 = (1.eql?(1.0))   # false
```

Метод eql? существует по одной-единственной причине: для сравнения значений ключей хэширования. Если вы хотите переопределить стандартное поведение Ruby при использовании объектов в качестве ключей хэша, то переопределите методы eql? и hash.

Любой объект реализует еще два метода сравнения. Метод === применяется для сравнения проверяемого значения в предложении case с каждым селектором: selector===target. Хотя правило на первый взгляд кажется сложным, на практике оно делает предложения case в Ruby интуитивно очевидными. Например, можно выполнить ветвление по классу объекта:

```
case an_object
  when String
    puts "Это строка."
  when Numeric
    puts "Это число."
  else
    puts "Это что-то совсем другое."
end
```

Эта конструкция работает, потому что в классе Module реализован метод ===, проверяющий, что параметр принадлежит тому же классу, что вызывающий объект (или одному из его предков). Поэтому, если an_object – это строка «cat», то выражение String === an_object окажется истинным, и будет выбрана первая ветвь в предложении case.

Наконец, в Ruby реализован оператор сопоставления с образцом =~. Традиционно он применяется для сопоставления строки с регулярным выражением. Но если вы найдете ему применение в других классах, то можете переопределить.

У операторов == и =~ есть противоположные формы: != и !~ соответственно. Внутри они реализованы путем обращения значения основной формы. Это означает, что если, например, вы реализовали метод ==, то метод != получаете задаром.

11.1.8. Управление доступом к методам

В Ruby объект определяется, прежде всего, своим интерфейсом: теми методами, которые он раскрывает внешнему миру. Но при написании класса часто возникает необходимость во вспомогательных методах, вызывать которые извне класса опасно. Тут-то и приходит на помощь метод private класса Module.

Использовать его можно двумя способами. Если в теле класса или модуля вы вызовете private без параметров, то все последующие методы будут закрытыми в данном классе или модуле. Если же вы передадите ему список имен методов (в виде символов), то эти и только эти методы станут закрытыми. В листинге 11.5 показаны оба варианта.

Листинг 11.5. Закрытые методы

```
class Bank
  def open_safe
    # ...
  end

  def close_safe
    # ...
  end

  private :open_safe, :close_safe

  def make_withdrawal(amount)
    if access_allowed
      open_safe
      get_cash(amount)
      close_safe
    end
  end

  # остальные методы закрытые

  private

  def get_cash
    # ...
  end

  def access_allowed
    # ...
  end
end
```

Поскольку методы из семейства attr просто определяют методы, то метод private определяет и видимость атрибутов тоже.

Реализация метода private может показаться странной, но на самом деле она весьма хитроумна. К закрытым методам нельзя обратиться, указав вызывающий объект, они вызываются только от имени неявно подразумеваемого объекта self. То есть вызвать закрытый метод из другого объекта не удастся, просто не существует способа указать объект, от имени которого этот метод вызывается. Заодно это означает, что закрытые методы доступны подклассам того класса, в котором определены, но опять же в рамках одного объекта.

Модификатор доступа protected налагает меньше ограничений. Защищенные методы доступны только экземплярам того класса, в котором определены, и его подклассов. В разделе 11.1.3 мы видели, что модификатор protected позволяет вызывать методы установки от имени self (при наличии модификатора private это невозможно).

К защищенным методам можно обращаться из других объектов (при условии, что вызывающий и вызываемый объекты принадлежат одному классу). Обычно защищенные методы применяются для определения методов доступа, чтобы два объекта одного типа могли взаимодействовать. В следующем примере объекты класса Person можно сравнивать по возрасту, но сам возраст недоступен вне класса Person. Закрытый атрибут pay_scale можно прочитать внутри класса, но извне класса он недоступен даже другим объектам того же класса:

```
class Person
   attr_reader :name, :age, :pay_scale
   protected   :age
   private     :pay_scale

   def initialize(name, age, pay_scale)
      @name, @age, @pay_scale = name, age, pay_scale
   end

   def <=>(other)
      age <=> other.age # allowed by protected
   end

   def same_rank?(other)
      pay_scale == other.pay_scale # not allowed by private
   end

   def rank
      case pay_scale
      when 1..3
         "lower"
      when 4..6
         "middle"
      when 7..9
         "high"
      end
   end
end
```

И чтобы довершить картину, модификатор public делает метод открытым. Неудивительно.

И последнее: методы, определенные вне любого класса и модуля (то есть на верхнем уровне программы), по умолчанию закрыты. Поскольку они определены в классе Object, то видимы глобально, но обращаться к ним с указанием вызывающего объекта нельзя.

11.1.9. Копирование объектов

Встроенные методы Object#clone и #dup порождают копию вызывающего объекта. Различаются они объемом копируемого контекста. Метод #dup копирует только само содержимое объекта, тогда как clone сохраняет и такие вещи, как синглетные классы, ассоциированные с объектом.

```
s1 = "cat"

def s1.upcase
  "CaT"
end

s1_dup   = s1.dup
s1_clone = s1.clone
s1                  #=> "cat"
s1_dup.upcase       #=> "CAT"  (синглетный метод не копируется)
s1_clone.upcase     #=> "CaT"  (используется синглетный метод)
```

И dup, и clone выполняют *поверхностное копирование*, то есть копируют лишь содержимое самого вызывающего объекта. Если вызывающий объект содержит ссылки на другие объекты, то последние не копируются, копия будет ссылаться на те же самые объекты. Проиллюстрируем это на примере. Объект arr2 – копия arr1, поэтому изменение элемента целиком, например arr2[2] не оказывает влияния на arr1. Но исходный массив и его копия содержат ссылку на один и тот же объект String, поэтому изменение строки через arr2 приведет к такому же изменению значения, на которое ссылается arr1.

```
arr1 = [ 1, "flipper", 3 ]
arr2 = arr1.dup

arr2[2] = 99
arr2[1][2] = 'a'

arr1              # [1, "flapper", 3]
arr2              # [1, "flapper", 99]
```

Иногда необходимо *глубокое копирование*, при котором копируется все дерево объектов с корнем в исходном объекте. В этом случае между оригиналом и копией гарантированно не будет никакой интерференции. Ruby не предоставляет встроенного метода для глубокого копирования, но есть приемы, позволяющие достичь желаемого результата.

Самый «чистый» способ потребовать, чтобы классы реализовывали метод deep_copy. Он мог бы рекурсивно обходить все объекты, на которые ссылается исходный объект, и вызывать для них метод deep_copy. Необходимо было бы еще добавить метод deep_copy во все встроенные классы Ruby, которыми вы пользуетесь.

Но есть и более быстрый способ с использованием модуля Marshal. Если вы сериализуете исходный объект, представив его в виде строки, а затем загрузите в новый объект, то этот новый объект будет копией исходного.

```
arr1 = [ 1, "flipper", 3 ]
arr2 = Marshal.load(Marshal.dump(arr1))

arr2[2] = 99
arr2[1][2] = 'a'

arr1              # [1, "flipper", 3]
arr2              # [1, "flapper", 99]
```

Обратите внимание, что изменение строки через arr2 не отразилось на строке, на которую ссылается arr1.

11.1.10. Метод initialize_copy

При копировании объекта методом dup или clone конструктор не вызывается. Копируется вся информация о состоянии.

Но что делать, если вам такое поведение не нужно? Рассмотрим пример:

```
class Document
  attr_accessor :title, :text
  attr_reader   :timestamp

  def initialize(title, text)
    @title, @text = title, text
    @timestamp = Time.now
  end
end

doc1 = Document.new("Random Stuff",File.read("somefile"))
sleep 300                          # Немного подождем...
doc2 = doc1.clone

doc1.timestamp == doc2.timestamp  # true
# Оп... временные штампы одинаковы!
```

При создании объекта Document с ним ассоциируется временная метка. При копировании объекта копируется и его временная метка. А как быть, если мы хотим запомнить время, когда было выполнено копирование?

Для этого нужно определить метод initialize_copy. Он вызывается как раз при копировании объекта. Этот метод аналогичен initialize и позволяет полностью контролировать состояние объекта.

```
class Document                          # Повторно открываем определение класса
  def initialize_copy(other)
    @timestamp = Time.now
  end
end

doc3 = Document.new("More Stuff",File.read("otherfile"))
sleep 300                               # Немного подождем...
doc4 = doc3.clone

doc3.timestamp == doc4.timestamp    # false
# Теперь временные метки правильны.
```

Отметим, что метод initialize_copy вызывается *после* того, как вся информация скопирована. Поэтому мы и опустили строку:

```
@title, @text = other.title, other.text
```

Кстати, если метод initialize_copy пуст, то поведение будет такое же, как если бы он не был определен вовсе.

11.1.11. Метод allocate

Редко, но бывает, что нужно создать объект, не вызывая его конструктор (в обход метода initialize). Например, может статься, что состояние объекта полностью определяется методами доступа к нему, тогда не нужно вызывать метод new (который вызовет initialize), разве что вы сами захотите это сделать. Представьте, что для инициализации состояния объекта вы собираете данные по частям, тогда начать следует с «пустого» объекта, а не получить все данные заранее, а потом вызвать конструктор.

Метод allocate упрощает решение этой задачи. Он возвращает «чистый», еще не инициализированный объект класса.

```
class Person
  attr_accessor :name, :age, :phone

  def initialize(n,a,p)
    @name, @age, @phone = n, a, p
  end
end

p1 = Person.new("John Smith",29,"555-1234")

p2 = Person.allocate

p p1.age    # 29
p p2.age    # nil
```

11.1.12. Модули

Для использования модулей в Ruby есть две основных причины. Первая — облегчить управление пространством имен; если поместить константы и методы в мо-

дули, то будет меньше конфликтов имен. Хранящийся таким образом метод (метод модуля) вызывается с указанием имени модуля, то есть без вызывающего объекта. Точно так же вызывается и метод класса. Увидев вызовы вида File.ctime или FileTest.exist?, мы не можем определить по контексту, что File – это класс, а FileTest – модуль.

Вторая причина более интересна: мы можем использовать модуль как примесь. Примеси – это способ реализации множественного наследования, при котором наследуется только интерфейс.

Мы уже говорили о методах модуля, а как насчет методов экземпляра? Модуль это не класс, у него не может быть экземпляров, а к методу экземпляра нельзя обратиться, не указав вызывающий объект.

Но оказывается, модуль может иметь методы экземпляра. Они становятся частью класса, который включил модуль директивой include.

```
module MyMod

  def meth1
    puts "Это метод 1"
  end

end

class MyClass

  include MyMod
  # ...
end


x = MyClass.new
x.meth1                  # Это метод 1
```

Здесь модуль MyMod подмешан к классу MyClass, а метод экземпляра meth1 унаследован. Вы видели также, как директива include употребляется на верхнем уровне программы; в таком случае модуль подмешивается к классу Object.

А что происходит с методами модуля, если таковые определены? Если вы думаете, что они становятся методами класса, то ошибаетесь. Методы модуля не подмешиваются.

Но если такое поведение желательно, то его можно реализовать с помощью нехитрого трюка. Существует метод included, который можно переопределить. Он вызывается с параметром – «целевым» классом или модулем (в который включается данный модуль). Пример приведен в листинге 11.6.

Листинг 11.6. Включение модуля с переопределенным методом included

```
module MyMod

   def self.included(klass)
      def klass.module_method
```

```
            puts "Module (class) method"
        end
    end

    def method_1
        puts "Method 1"
    end

end

class MyClass

    include MyMod

    def self.class_method
        puts "Class method"
    end

    def method_2
        puts "Method 2"
    end

end

x = MyClass.new
                          # Выводится:
MyClass.class_method      # Class method
x.method_1                # Method 1
MyClass.module_method     # Module (class) method
x.method_2                # Method 2
```

Этот пример заслуживает детального рассмотрения. Во-первых, надо понимать, что метод included вызывается в ходе выполнения include. При желании мы могли бы реализовать свой метод include, определив append_features, но такое решение встречается гораздо реже (да и вряд ли вообще желательно).

Отметим также, что внутри тела included имеется еще одно определение метода. Выглядит это необычно, но работает, поскольку вложенное определение порождает синглетный метод (уровня класса или модуля). Попытка определить таким образом метод экземпляра привела бы к ошибке Nested method error (Ошибка при определении вложенного метода).

Модуль мог бы захотеть узнать, кто был инициатором примеси. Для этого тоже можно воспользоваться методом included, потому что класс-инициатор передастся ему в качестве параметра.

Можно также подмешивать методы экземпляра модуля как методы класса с помощью метода include или extend. В листинге 11.7 приведен соответствующий пример.

Листинг 11.7. Методы экземпляра модуля становятся методами класса

```
module MyMod

  def meth3
    puts "Метод экземпляра модуля meth3"
    puts "может стать методом класса."
  end

end

class MyClass

  class << self     # Здесь self - это MyClass
    include MyMod
  end

end


MyClass.meth3

# Выводится:
#   Метод экземпляра модуля meth3
#   может стать методом класса.
Здесь полезен метод extend. Тогда пример можно записать так:
class MyClass
    extend MyMod
end
```

Мы все время говорим о методах. А как насчет переменных экземпляра? Конечно, модуль может иметь собственные данные экземпляра, но обычно так не делают. Впрочем, если вы решили, что без этого никак не обойтись, ничто вас не остановит.

Можно подмешивать модуль к объекту, а не классу (например, методом `extend`). См. по этому поводу раздел 11.2.2.

Важно понимать еще одну вещь. В вашем классе можно определить методы, которые будут вызываться из примеси. Это эффектный прием, знакомый тем, кто пользовался интерфейсами в Java.

Классический пример (с которым мы уже сталкивались ранее), – подмешивание модуля `Comparable` и определение метода `<=>`. Поскольку подмешанные методы могут вызывать метод сравнения, то мы получаем операторы `<`, `>`, `<=` и т.д.

Другой пример – подмешивание модуля `Enumerable` и определение метода `<=>` и итератора `each`. Тем самым мы получаем целый ряд полезных методов: `collect`, `sort`, `min`, `max` и `select`.

Можно также определять и собственные модули, ведущие себя подобным образом. Возможности ограничены только вашим воображением.

11.1.13. Трансформация или преобразование объектов

Иногда объект имеет нужный вид в нужное время, а иногда хочется преобразовать его во что-то другое или сделать вид, что он является чем-то, чем на самом деле не является. Всем известный пример – метод to_s.

Каждый объект можно тем или иным способом представить в виде строки. Но не каждый объект может успешно «прикинуться» строкой. Именно в этом и состоит различие между методами to_s и to_str. Рассмотрим этот вопрос подробнее.

При использовании метода puts или интерполяции в строку (в контексте #{...}) ожидается, что в качестве параметра будет передан объект String. Если это не так, объект просят преобразовать себя в String, посылая ему сообщение to_s. Именно здесь вы можете определить, как объект следует отобразить, просто реализуйте метод to_s в своем классе так, чтобы он возвращал подходящую строку.

```
class Pet

  def initialize(name)
    @name = name
  end

  # ...

  def to_s
    "Pet: #@name"
  end

end
```

Другие методы (например, оператор конкатенации строк +) не так требовательны, они ожидают получить нечто достаточно близкое к объекту String. В этом случае Мац решил, что интерпретатор не будет вызывать метод to_s для преобразования не-строковых аргументов, поскольку это могло бы привести к большому числу ошибок. Вместо этого вызывается более строгий метод to_str. Из всех встроенных классов только String и Exception реализуют to_str, и лишь String, Regexp и Marshal вызывают его. Увидев сообщение TypeError: Failed to convert xyz into String, можно смело сказать, что интерпретатор пытался вызвать to_str и потерпел неудачу.

Вы можете реализовать метод to_str и самостоятельно, например, если хотите, чтобы строку можно было конкатенировать с числом:

```
class Numeric

  def to_str
    to_s
  end

end

label = "Число " + 9      # "Число 9"
```

Аналогично обстоит дело и для массивов. Для преобразования объекта в массив служит метод to_a, а метод to_ary вызывается, когда ожидается массив и ничего другого, например, в случае множественного присваивания. Пусть есть предложение такого вида:

```
a, b, c = x
```

Если x – массив их трех элементов, оно будет работать ожидаемым образом. Но если это не массив, интерпретатор попытается вызвать метод to_ary для преобразования в массив. В принципе, это может быть даже синглетный метод (принадлежащий конкретному объекту). На само преобразование не налагается никаких ограничений, ниже приведен (нереалистичный) пример, когда строка преобразуется в массив строк:

```
class String

  def to_ary
    return self.split("")
  end

end

str = "UFO"
a, b, c = str     # ["U", "F", "O"]
```

Метод inspect реализует другое соглашение. Отладчики, утилиты типа irb и метод отладочной печати p вызывают inspect, чтобы преобразовать объект к виду, пригодному для вывода на печать. Если вы хотите, чтобы во время отладки объект раскрывал свое внутреннее устройство, то переопределите inspect.

Приведение числовых типов – еще одна ситуация, когда производится преобразование объекта. Подробнее см. раздел 5.16.

11.1.14. Классы, содержащие только данные (Struct)

Иногда нужно просто сгруппировать взаимосвязанные данные. Поначалу кажется, что для этого достаточно воспользоваться массивом или хэшем. Но такой подход хрупок – становится трудно изменить внутреннюю структуру класса или добавить методы-акцессоры. Можно было бы попытаться решить проблему, создав такой класс:

```
class Address

  attr_accessor :street, :city, :state

  def initialize(street, city, state)
    @street, @city, @state = street, city, state
  end

end

books = Address.new("411 Elm St", "Dallas", "TX")
```

Такое решение годится, но каждый раз прибегать к нему утомительно, к тому же здесь слишком много повторов. Тут-то и приходит на помощь встроенный класс Struct. Если вспомогательные методы типа attr_accessor определяют методы доступа к атрибутам, то Struct определяет целый класс, который может содержать только атрибуты. Такие классы называются *структурными шаблонами*.

```
Address = Struct.new("Address", :street, :city, :state)
books = Address.new("411 Elm St", "Dallas", "TX")
```

Зачем передавать первым параметром конструктора имя создаваемой структуры и присваивать результат константе (в данном случае Address)?

При вызове Struct.new для создания нового структурного шаблона на самом деле создается новый класс внутри самого класса Struct. Этому классу присваивается имя, переданное первым параметром, а остальные параметры становятся именами его атрибутов. При желании к вновь созданному классу можно было бы получить доступ, указав пространство имен Struct:

```
Struct.new("Address", :street, :city, :state)
books = Struct::Address.new("411 Elm St", "Dallas", "TX")
```

Если первый аргумент Struct.new – символ, то все аргументы считаются именами атрибутов, и класс в пространстве имен Struct не создается.

При создании класса Struct можно определить дополнительные методы, передав конструктору Struct.new блок. Этот блок будет вычисляться так, будто он является телом класса – практически так же, как в определении любого класса.

Создав структурный шаблон, мы вызываем его метод new для создания новых экземпляров данной конкретной структуры. Необязательно присваивать значения всем атрибутам в конструкторе. Опущенные атрибуты получат значение nil. После того как структура создана, к ее атрибутам можно обращаться с помощью обычного синтаксиса или указывая их имена в скобках в качестве индекса, как будто структура – это объект класса Hash. Более подробную информацию о классе Struct можно найти в любом справочном руководстве (например, на сайте rdoc.info).

Кстати, не рекомендуем создавать структуру с именем Tms, так как уже есть предопределенный класс Struct::Tms.

11.1.15. Замораживание объектов

Иногда необходимо воспрепятствовать изменению объекта. Это позволяет сделать метод freeze (определенный в классе Object). По существу, он превращает объект в константу.

Попытка модифицировать замороженный объект приводит к исключению TypeError. В листинге 11.8 приведено два примера.

Листинг 11.8. Замораживание объекта

```
str = "Это тест. "
str.freeze

begin
  str << " Не волнуйтесь."     # Попытка модифицировать
```

```
rescue => err
  puts "#{err.class} #{err}"
end

arr = [1, 2, 3]
arr.freeze

begin
  arr << 4                    # Попытка модифицировать
rescue => err
  puts "#{err.class} #{err}"
end

# Выводится:
#   TypeError: can't modify frozen string
#   TypeError: can't modify frozen array
```

Замораживание строк считается особым случаем. Интерпретатор создает единственный объект замороженной строки и возвращает его при каждом обращении к этой строке. Это снижает потребление памяти, если, например, некая строка возвращается из метода, который вызывается многократно:

```
str1 = "Woozle".freeze
str2 = "Woozle".freeze

str1.object_id == str2.object_id   # true
```

Но хотя замораживание и предотвращает модификацию, имейте в виду, что метод `freeze` применяется к ссылке на объект, а не к переменной! Это означает, что любая операция, приводящая к созданию нового объекта, завершится успешно. Иногда это противоречит интуиции. В примере ниже мы ожидаем, что операция `+=` не выполнится, но все работает нормально. Дело в том, что присваивание — не вызов метода. Эта операция воздействует на переменные, а не на объекты, поэтому новый объект создается беспрепятственно. Старый объект по-прежнему заморожен, но переменная ссылается уже не на него.

```
str = "counter-"
str.freeze
str += "intuitive"        # "counter-intuitive"

arr = [8, 6, 7]
arr.freeze
arr += [5, 3, 0, 9]       # [8, 6, 7, 5, 3, 0, 9]
```

Почему так происходит? Предложение `a += x` семантически эквивалентно `a = a + x`. При вычислении выражения `a + x` создается новый объект, который затем присваивается переменной `a`! Все составные операторы присваивания работают подобным образом, равно как и другие методы. Всегда задавайте себе вопрос: «Что я делаю — создаю новый объект или модифицирую существующий?» И тогда поведение `freeze` не станет для вас сюрпризом.

Существует метод frozen?, который сообщает, заморожен ли данный объект.

```
hash = { 1 => 1, 2 => 4, 3 => 9 }
hash.freeze
arr = hash.to_a
puts hash.frozen?   # true
puts arr.frozen?    # false
hash2 = hash
puts hash2.frozen?  # true
```

Как видите (на примере hash2), замораживается именно объект, а не переменная.

11.1.16. Использование метода tap в цепочке методов

Методу tap передается блок, из которого имеется доступ к значению выражения, вычисляемого другими операциями. (По-английски, слово «tap» означает прослушивание телефона или отвод от трубы).

Например, так можно распечатывать промежуточные значения в ходе их обработки, потому что блоку передается объект, от имени которого метод tap вызван:

```
a = [1, 5, 1, 4, 2, 3, 4, 3, 2, 5, 2, 1]
p a.sort.uniq.tap{|x| p x }.map {|x| x**2 + 2*x + 7 }
# [1, 2, 3, 4, 5]
# [10, 15, 22, 31, 42]
```

Наличие такого доступа полезно само по себе, но истинная мощь tap проявляется, когда нужно изменить объект внутри блока. Это позволяет программе модифицировать объект и при этом вернуть исходный объект, а не результат последней операции:

```
def feed(woozle)
   woozle.tap do |w|
      w.stomach << Steak.new
   end
end
```

Не будь здесь tap, метод feed вернул бы объект stomach (желудок), в котором теперь находится steak (стейк). А так возвращается покормленный (и, надо полагать, довольный) вузл.

Эта техника особенно полезна в методах класса, которые конструируют объект, вызывают какие-то методы нового объекта, а затем возвращают его. Удобна она и для реализации методов, которые планируется сцеплять.

11.2. Более сложные механизмы

Не все в модели ООП, реализованной в Ruby, одинаково очевидно. Что-то сложнее, что-то применяется реже. Линия раздела для каждого программиста проходит в разных местах. В этой части главы мы попытались собрать те средства, которые не так просты или не так часто встречаются в программах.

Иногда вы задаетесь вопросом, можно ли решить на Ruby ту или иную задачу. Краткий ответ таков: Ruby – богатый, динамический, объектно-ориентированный

язык с широким набором разумно ортогональных средств; если нечто можно сделать на каком-то другом языке, то, скорее всего, можно и на Ruby.

Теоретически все полные по Тьюрингу языки более или менее одинаковы. Весь смысл проектирования языков в поиске осмысленной, удобной нотации. Читателю, сомневающемуся в важности нотации, стоит попробовать написать интерпретатор LISP на языке COBOL или выполнить деление чисел, записанных римскими цифрами.

Конечно, мы не хотим сказать, что любая задача на Ruby решается элегантно или естественно. Попытайся мы высказать такое утверждение, кто-нибудь очень быстро докажет, что мы не правы.

В этом разделе мы поговорим также о реализации на Ruby различных стилей программирования, например, функционального и аспектно-ориентированного. Мы не претендуем на роль экспертов в этих областях, просто приводим мнения других. Относитесь к этому с долей скепсиса.

11.2.1. Посылка объекту явного сообщения

В статическом языке вы считаете очевидным, что имя вызываемой функции «зашито» в программу, это часть исходного текста. Динамический язык обладает в этом отношении большей гибкостью.

При любом вызове метода вы посылаете объекту сообщение. Обычно эти сообщения так же жестко «зашиты» в код, как и в статическом языке, но это необязательно. Можно написать программу, которая во время выполнения решает, какой метод вызывать. Метод `send` позволяет использовать `Symbol` для представления имени метода.

Пусть, например, имеется массив объектов, который нужно отсортировать, причем в качестве ключей сортировки хотелось бы использовать разные поля. Не проблема, можно просто написать специализированные блоки для сортировки. Но хотелось бы найти более элегантное решение, позволяющее обойтись одной процедурой, способной выполнить сортировку по любому указанному ключу. В листинге 11.9 такое решение приведено.

Листинг 11.9. Метод, название которого задается параметром

```
class Person
  attr_reader :name, :age, :height

  def initialize(name, age, height)
    @name, @age, @height = name, age, height
  end

  def inspect
    "#@name #@age #@height"
  end
end


class Array
```

```
  def map_by(sym)
    self.map {|x| x.send(sym) }
  end
end


people = []
people << Person.new("Hansel", 35, 69)
people << Person.new("Gretel", 32, 64)
people << Person.new("Ted", 36, 68)
people << Person.new("Alice", 33, 63)

p1 = people.map_by(:name)      # Hansel, Gretel, Ted, Alice
p2 = people.map_by(:age)       # 35, 32, 36, 33
p3 = people.map_by(:height)    # 69, 64, 68, 63
```

Этот код на самом деле излишен, потому что можно получить тот же результат, просто вызвав метод `map(&:name)`, но мы хотели проиллюстрировать использование метода `send`. Мы еще вернемся к этой теме в разделе 11.2.7.

Отметим еще, что синоним `__send__` делает в точности то же самое. Такое странное имя объясняется, конечно, опасением, что имя `send` уже может быть задействовано (случайно или намеренно) для определенного пользователем метода.

С методом `send` связана одна проблема: он позволяет обойти модель закрытости в Ruby (в том смысле, что закрытый метод можно вызвать косвенно, отправив объекту строку или символ методом `send`). Это не ошибка проектирования, а сознательное решение, но если вы чувствует себя увереннее, «защитившись» от такого непреднамеренного использования, то можете обратиться к методу `public_send`.

11.2.2. Специализация отдельного объекта

Я солипсист и, признаться, удивлен,
что большинство из нас таковыми не являются.

— Из письма, полученного Бертраном Расселом

В большинстве объектно-ориентированных языков все объекты одного класса ведут себя одинаково. Класс – это шаблон, порождающий объекты с одним и тем же интерфейсом при каждом вызове конструктора.

Ruby ведет себя так же, но это не конец истории. Получив объект, вы можете изменить его поведение на лету. По сути дела, вы ассоциируете с объектом частный, анонимный подкласс, все методы исходного подкласса остаются доступными, но добавляется еще и поведение, уникальное для данного объекта. Поскольку это поведение присуще лишь данному объекту, оно встречается лишь один раз. Нечто, встречающееся только один раз, называется синглетом (singleton). Так, мы

имеем синглетные методы и синглетные классы (иногда они называются также метаклассами).

Слово «синглет» может стать источником путаницы, потому что оно употребляется и в другом смысле – как название хорошо известного паттерна проектирования, описывающего класс, для которого может существовать лишь один объект. Если вас интересует такое использование, обратитесь к библиотеке singleton.

В следующем примере мы видим два объекта, оба строки. Для второго мы добавляем метод upcase, который переопределяет существующий метод с таким же именем.

```
a = "hello"
b = "goodbye"

def b.upcase        # создать синглетный метод
  gsub(/(.)(.)/) { $1.upcase + $2 }
end

puts a.upcase       # HELLO
puts b.upcase       # GoOdBye
```

Добавление синглетного метода к объекту порождает синглетный класс для данного объекта, если он еще не был создан ранее. Родителем синглетного класса является исходный класс объекта. (Можно считать, что это анонимный подкласс исходного класса.) Если вы хотите добавить к объекту несколько методов, то можете создать синглетный класс явно:

```
b = "goodbye"

class << b

  def upcase        # создать синглетный метод
    gsub(/(.)(.)/) { $1.upcase + $2 }
  end

  def upcase!
    gsub!(/(.)(.)/) { $1.upcase + $2 }
  end

end

puts b.upcase       # GoOdBye
puts b              # goodbye
b.upcase!
puts b              # GoOdBye
```

Отметим попутно, что у более «примитивных» объектов (например, Fixnum) не может быть добавленных синглетных методов. Связано это с тем, что такие объекты хранятся как непосредственные значения, а не как ссылки. Впрочем, реализа-

ция подобной функциональности планируется в будущих версиях Ruby (хотя значения так и останутся непосредственными).

Если вам приходилось разбираться в коде библиотек, то наверняка вы сталкивались с идиоматическим использованием синглетных классов. В определении класса иногда встречается такой код:

```
class SomeClass

  # Какой-то код...

  class << self
    # еще какой-то код...
  end

  # ... продолжение.

end
```

В теле определения класса слово self обозначает сам определяемый класс, поэтому создание наследующего ему синглета модифицирует класс этого класса. Можно сказать, что методы экземпляра синглетного класса извне выглядят как методы самого класса.

```
class TheClass
  class << self
    def hello
      puts "hi"
    end
  end
end

# вызвать метод класса
TheClass.hello            # hi
```

Еще одно распространенное применение такой техники – определение на уровне класса вспомогательных функций, к которым можно обращаться из других мест внутри определения класса. Например, мы хотим определить несколько функций доступа, которые преобразуют результат своей работы в строку. Можно, конечно, написать отдельно код каждой такой функции. Но есть и более элегантное решение – определить функцию уровня класса accessor_string, которая сгенерирует необходимые нам функции (как показано в листинге 11.10).

Листинг 11.10. Метод уровня класса accessor_string

```
class MyClass

  class << self

    def accessor_string(*names)
      names.each do |name|
```

```
        class_eval <<-EOF
          def #{name}
            @#{name}.to_s
          end
        EOF
      end
    end

  end

  def initialize
    @a = [ 1, 2, 3 ]
    @b = Time.now
  end

  accessor_string :a, :b

end


o = MyClass.new
puts o.a            # 123
puts o.b            # 2014-07-26 00:45:12 -0700
```

Вы наверняка сможете придумать и другие, более изобретательные применения.

Метод extend подмешивает к объекту модуль. Методы экземпляра, определенные в модуле, становятся методами экземпляра объекта. Взгляните на листинг 11.11.

Листинг 11.11. Использование метода extend

```
module Quantifier
   def two?
      2 == self.select { |x| yield x }.size
   end

   def four?
      4 == self.select { |x| yield x }.size
   end

end

list = [1, 2, 3, 4, 5]
list.extend(Quantifier)

flag1 = list.two? {|x| x > 3 }       # => true
flag2 = list.two? {|x| x >= 3 }      # => false
flag3 = list.four? {|x| x <= 4 }     # => true
flag4 = list.four? {|x| x % 2 == 0 } # => false
```

В этом примере к массиву list подмешаны методы two? и four?.

11.2.3. Вложенные классы и модули

Классы и модули можно вкладывать друг в друга произвольным образом. Программисты, приступающие к изучению Ruby, могут этого и не знать.

Основная цель этого механизма — упростить управление пространствами имен. Скажем, в класс File вложен класс Stat. Это помогает «инкапсулировать» класс Stat внутри тесно связанного с ним класса, а заодно оставляет возможность в будущем определить класс Stat, не конфликтуя с существующим (скажем, для сбора статистики).

Другой пример дает класс Struct::Tms. Любая новая структура Struct помещается в это пространство имен, не «загрязняя» расположенные выше, а Tms – в действительности тоже Struct.

Кроме того, вложенный класс можно создавать просто потому, что внешний мир не должен знать о нем или обращаться к нему. Иными словами, можно создавать целые классы, подчиняющиеся тому же принципу «сокрытия данных», которому переменные и методы экземпляра следуют на более низком уровне.

```
class BugTrackingSystem

  class Bug
    #...
  end

  #...

end

# Никто снаружи не знает о классе Bug.
```

Можно вкладывать класс в модуль, модуль в класс и т.д.

11.2.4. Создание параметрических классов

Изучи правила, потом нарушай их.

— Басё

Предположим, что нужно создать несколько классов, отличающихся только начальными значениями переменных уровня класса. Напомним, что переменная класса обычно инициализируется в самом определении класса.

```
class Terran

  @@home_planet = "Earth"

  def Terran.home_planet
```

```
    @@home_planet
  end

  def Terran.home_planet=(x)
    @@home_planet = x
  end

  #...

end
```

Все замечательно, но что если нам нужно определить несколько подобных классов. Новичок подумает: «Так я просто определю суперкласс» (листинг 11.12).

Листинг 11.12. Параметрические классы: неправильное решение

```
class IntelligentLife   # Неправильный способ решения задачи!

  @@home_planet = nil

  def IntelligentLife.home_planet
    @@home_planet
  end

  def IntelligentLife.home_planet=(x)
    @@home_planet = x
  end

  #...
end

class Terran < IntelligentLife
  @@home_planet = "Earth"
  #...
end

class Martian < IntelligentLife
  @@home_planet = "Mars"
  #...
end
```

Но это работать не будет. Вызов `Terran.home_planet` напечатает не `"Earth"`, а `"Mars"`! Почему так? Дело в том, что переменные класса — на деле не вполне переменные класса; они принадлежат не одному классу, а всей иерархии наследования. Переменная класса не копируется из родительского класса, а разделяется с родителем (и, стало быть, со всеми братьями).

Можно было бы вынести определение переменной класса в базовый класс, но тогда перестали бы работать определенные нами методы класса! Можно было бы исправить и это, перенеся определения в дочерние классы, но это губит первона-

чальную идею, ведь таким образом объявляются отдельные классы без какой бы то ни было «параметризации».

Мы предлагаем другое решение. Отложим вычисление переменной класса до момента выполнения, воспользовавшись методом class_eval. Полное решение приведено в листинге 11.13.

Листинг 11.13. Параметрические классы: улучшенное решение

```
class IntelligentLife

  def IntelligentLife.home_planet
    class_eval("@@home_planet")
  end

  def IntelligentLife.home_planet=(x)
    class_eval{"@@home_planet = #{x}"}
  end

  #...
end

class Terran < IntelligentLife
  @@home_planet = "Earth"
  #...
end

class Martian < IntelligentLife
  @@home_planet = "Mars"
  #...
end


puts Terran.home_planet            # Earth
puts Martian.home_planet           # Mars
```

Не стоит и говорить, что механизм наследования здесь по-прежнему работает. Все методы и переменные экземпляра, определенные в классе IntelligentLife, наследуются классами Terran и Martian.

В листинге 11.14 предложено, наверное, наилучшее решение. В нем используются только переменные экземпляра, а от переменных класса мы вообще отказались.

Листинг 11.14. Параметрические классы: самое лучшее решение

```
class IntelligentLife
  class << self
    attr_accessor :home_planet
  end

  #...
```

```
end

class Terran < IntelligentLife
  self.home_planet = "Earth"
  #...
end

class Martian < IntelligentLife
  self.home_planet = "Mars"
  #...
end


puts Terran.home_planet            # Earth
puts Martian.home_planet           # Mars
```

Здесь мы открываем синглетный класс и определяем метод доступа home_planet. В двух подклассах определяются собственные методы доступа и устанавливается переменная. Теперь методы доступа работают строго в своих классах.

В качестве небольшого усовершенствования добавим еще вызов метода private в синглетный класс:

```
private :home_planet=
```

Сделав метод установки закрытым, мы запретили изменять значение вне иерархии данного класса. Как всегда, private реализует «рекомендательную» защиту, которая легко обходится. Но объявление метода закрытым по крайней мере говорит, что мы не хотели, чтобы метод вызывался вне определенного контекста.

Есть и другие способы решения этой задачи. Проявите воображение.

11.2.5. Хранение кода в виде объектов Proc

Не удивительно, что Ruby предлагает несколько вариантов хранения фрагмента кода в виде объекта. В этом разделе мы рассмотрим объекты Proc и лямбда-выражения.

Встроенный класс Proc позволяет обернуть блок в объект. Объекты Proc, как и блоки, являются замыканиями, то есть запоминают контекст, в котором были определены. Метод proc — просто сокращенная запись Proc.new.

```
local = 12
myproc = Proc.new {|a| puts "Параметр равен is #{a}, local равно #{local}" }
myproc.call(99)              # Параметр равен 99, local равно 12
```

Кроме того, Ruby автоматически создает объект Proc, когда метод, последний параметр которого помечен амперсандом, вызывается с блоком в качестве параметра:

```
def take_block(x, &block)
  puts block.class
  x.times {|i| block[i, i*i] }
end

take_block(3) { |n,s| puts "#{n} в квадрате равно #{s}" }
```

В этом примере демонстрируется также применение квадратных скобок как синонима метода call. Вот что выводится в результате исполнения:

```
Proc
0 в квадрате 0
1 в квадрате 1
2 в квадрате 4
```

Объект Proc можно передавать методу, который ожидает блок, предварив имя знаком &:

```
myproc = proc { |n| print n, "... " }
(1..3).each(&myproc)                    # 1... 2... 3...
```

Возможность передавать объект Proc методу, безусловно, полезна, но если внутри Proc встретится предложение return, то произойдет выход из всего метода. Специальная разновидность объектов Proc – *лямбда-выражение*, или просто *лямбда* – позволяет вернуть управление только из блока:

```
def greet(&block)
   block.call
   "Всем доброе утро."
end

philippe_proc = Proc.new { return "Слишком рано, Филипп!" }
philippe_lambda = lambda { return "Слишком рано, Филипп!" }

p greet(philippe_proc)       # Слишком рано, Филипп!
p greet(philippe_lambda)     # Всем доброе утро.
```

Для создания лямбда-выражений применяется не только ключевое слово lambda, но и нотация ->. Но помните, что при использовании оператора -> аргументы блока указываются во внешних фигурных скобках:

```
non_stabby_lambda = lambda {|king| greet(king) }
stabby_lambda = -> (king) { stab(king) }
```

11.2.6. Хранение кода в виде объектов Method

Ruby позволяет также превратить метод в объект непосредственно с помощью метода Object#method, который создает объект класса Method как замыкание, связанное с тем объектом, из которого он был создан.

```
str = "cat"
meth = str.method(:length)

a = meth.call               #  3  (длина "cat")
str << "erpillar"
b = meth.call               # 11  (длина "caterpillar")

str = "dog"
c = meth.call               # 11  (длина "caterpillar")
```

Обратите внимание на последний вызов call! Переменная str теперь ссылается на новый объект ("dog"), но meth по-прежнему связан со старым объектом.

Метод public_method работает аналогично, но, как следует из названия, ищет только открытые методы объекта-получателя.

Чтобы получить метод, который можно будет использовать для любого экземпляра конкретного класса, можно применить метод Module#instance_method для создания объектов UnboundMethod. Прежде чем вызывать объект UnboundMethod, нужно связать его с каким-то объектом. Результатом операции связывания является объект Method, который можно вызывать как обычно:

```
umeth = String.instance_method(:length)

m1 = umeth.bind("cat")
m1.call                          # 3

m2 = umeth.bind("caterpillar")
m2.call                          # 11
```

Связывание метода с неподходящим классом приведет к ошибке, но возможность связывания с другими объектами делает объект UnboundMethod интуитивно немного более понятным, чем Method.

11.2.7. Использование символов в качестве блоков

Если параметру предшествует знак амперсанда, то Ruby считает его блоком. Как мы уже видели, можно создать объект Proc, присвоить его переменной, а затем передать в качестве блока методу, принимающему блок.

Однако в разделе 11.2.1 было сказано, что можно вызывать методы, принимающие блок, но только при условии, что им передается символ, которому предшествует знак амперсанда. Как такое возможно? Ответ кроется в методе to_proc.

Неочевидный побочный эффект передачи блока в качестве аргумента состоит в том, что если аргумент не является объектом Proc, то Ruby сделает попытку преобразовать его в такой, вызвав метод to_proc.

Изобретательные программисты поняли, что этим можно воспользоваться, чтобы упростить обращения к методу map, и начали определять метод to_proc в классе Symbol. Реализация выглядит так:

```
class Symbol
   def to_proc
      Proc.new {|obj| obj.send(self) }
   end
end

# Что позволяет вызывать map следующим образом:
%w[A B C].map(&:chr) # [65, 66, 67]
```

Этот прием оказался настолько популярным, что был включен в сам язык Ruby, и теперь им можно пользоваться везде.

11.2.8. Как работает включение модулей?

Когда модуль включается в класс, Ruby на самом деле создает прокси-класс, являющийся непосредственным родителем данного класса. Возможно, вам это по-

кажется интуитивно очевидным, возможно, нет. Все методы включаемого модуля «маскируются» методами, определенными в классе.

```
module MyMod
  def meth
    "из модуля"
  end
end

class ParentClass
  def meth
    "из родителя"
  end
end

class ChildClass < ParentClass
  include MyMod
  def meth
    "из потомка"
  end
end

x = ChildClass.new
p x.meth              # из потомка
```

Выглядит это, как настоящее наследование: все, что потомок переопределил, становится действующим определением вне зависимости от того, вызывается ли include до или после переопределения.

Вот похожий пример, в котором метод потомка вызывает super, а не просто возвращает строку. Как вы думаете, что будет возвращено?

```
# Модуль MyMod и класс ParentClass не изменились

class ChildClass < ParentClass
  include MyMod
  def meth
    "из потомка: super = " + super
  end
end

x = ChildClass.new
p x.meth              # из потомка: super = из модуля
```

Отсюда видно, что модуль действительно является новым родителем класса. А что если мы точно так же вызовем super из модуля?

```
module MyMod
  def meth
    "из модуля: super = " + super
  end
end
```

```
# ParentClass не изменился

class ChildClass < ParentClass
  include MyMod
  def meth
    "из потомка: super = " + super
  end
end

x = ChildClass.new
p x.meth       # из потомка: super = из модуля: super = из родителя
```

У модулей припасен еще один козырь в рукаве – метод prepend. Он позволяет вставить метод модуля выше метода включающего класса:

```
# MyMod и ParentClass не изменились

class ChildClass < ParentClass
   prepend MyMod
   def meth
      "из потомка: super " - super
   end
end

x = ChildClass.new
p x.meth # из модуля: super из потомка: super из родителя
```

Эта особенность Ruby позволяет модулям изменять поведение методов, даже если метод класса-потомка не вызывает super.

Вне зависимости от того, как включен модуль – методом include или prepend, – метод meth, определенный в модуле MyMod, может вызвать super только потому, что в суперклассе (точнее, хотя бы в одном из его предков) действительно есть метод meth. А что произошло бы, вызови мы этот метод при других обстоятельствах?

```
module MyMod
  def meth
    "из модуля: super = " + super
  end
end

class Foo
  include MyMod
end

x = Foo.new
x.meth
```

При выполнении этого кода мы получили бы ошибку NoMethodError (или обращение к методу method_missing, если бы таковой существовал).

11.2.9. Опознание параметров, заданных по умолчанию

Однажды Ян Макдональд (Ian Macdonald) задал в списке рассылки вопрос: «Можно ли узнать, был ли параметр задан вызывающей программой или взято значение по умолчанию?» Интересный вопрос. Не каждый день он возникает, но все равно интересно.

Было предложено по меньшей мере три решения. Самое удачное и простое нашел Нобу Накада (Nobu Nakada). Оно приведено ниже:

```
def meth(a, b=(flag=true; 345))
  puts "b равно  #{b}, a flag равно #{flag.inspect}"
end

meth(123)        # b равно 345, a flag равно true
meth(123,345)    # b равно 345, a flag равно nil
meth(123,456)    # b равно 456, a flag равно nil
```

Как видим, этот подход работает, даже если вызывающая программа явно указала значение параметра, совпадающее с подразумеваемым по умолчанию. Трюк становится очевидным, едва вы его увидите: выражение в скобках устанавливает локальную переменную flag в true, а затем возвращает значение по умолчанию 345. Это дань могуществу Ruby.

11.2.10. Делегирование или перенаправление

В Ruby есть две библиотеки, которые предлагают решение задачи о делегировании или перенаправлении вызовов методов другому объекту. Они называются delegate и forwardable; мы рассмотрим обе.

Библиотека delegate предлагает три способа решения задачи.

Класс SimpleDelegator полезен, когда объект, которому делегируется управление (делегат), может изменяться на протяжении времени жизни делегирующего объекта. Чтобы выбрать объект-делегат, используется метод __setobj__.

Однако мне этот способ представляется слишком примитивным. Поскольку я не думаю, что это существенно лучше, чем то же самое, сделанное вручную, то задерживаться на классе SimpleDelegator не стану.

Метод верхнего уровня DelegateClass принимает в качестве параметра класс, которому делегируется управление. Затем он создает новый класс, которому мы можем унаследовать. Вот пример создания класса Queue, который делегирует объекту Array:

```
require 'delegate'
class MyQueue < DelegateClass(Array)

  def initialize(arg=[])
    super(arg)
  end

  alias_method :enqueue, :push
```

```
  alias_method :dequeue, :shift
end

mq = MyQueue.new

mq.enqueue(123)
mq.enqueue(234)

p mq.dequeue        # 123
p mq.dequeue        # 234
```

Можно также унаследовать класс `Delegator` и реализовать метод `__getobj__`; именно таким образом реализован класс `SimpleDelegator`. При этом мы получаем больший контроль над делегированием.

Но если вам необходим больший контроль, то, вероятно, вы все равно осуществляете делегирование на уровне отдельных методов, а не класса в целом. Тогда лучше воспользоваться библиотекой `forwardable`. Вернемся к примеру очереди:

```
require 'forwardable'

class MyQueue
  extend Forwardable

  def initialize(obj=[])
    @queue = obj    # делегировать этому объекту
  end

  def_delegator :@queue, :push,  :enqueue
  def_delegator :@queue, :shift, :dequeue

  def_delegators :@queue, :clear, :empty?, :length, :size, :<<

  # Прочий код...
end
```

Как видно из этого примера, метод `def_delegator` ассоциирует вызов метода (скажем, `enqueue`) с объектом-делегатом `@queue` и одним из методов этого объекта (`push`). Иными словами, когда мы вызываем метод `enqueue` для объекта `MyQueue`, производится делегирование методу `push` объекта `@queue` (который обычно является массивом).

Обратите внимание, мы пишем `:@queue`, а не `:queue` или `@queue`. Объясняется это тем, как написан класс `Forwardable`; можно было бы сделать и по-другому.

Иногда нужно делегировать методы одного объекта одноименным методам другого объекта. Метод `def_delegators` позволяет задать произвольное число таких методов. Например, в примере выше показано, что вызов метода `length` объекта `MyQueue` приводит к вызову метода `length` объекта `@queue`.

В отличие от первого примера, остальные методы делегирующим объектом просто не поддерживаются. Иногда это хорошо. Ведь не хотите же вы вызы-

вать метод [ ] или [ ]= для очереди; если вы так поступаете, то очередь перестает быть очередью.

Отметим еще, что показанный выше код позволяет вызывающей программе передавать объект конструктору (для использования в качестве объекта-делегата). В полном соответствии с духом динамической типизации это означает, что я могу выбирать вид объекта, которому делегировать управление, коль скоро он поддерживает те методы, которые вызываются в программе.

Например, все приведенные ниже вызовы допустимы. (В последних двух предполагается, что предварительно было выполнено предложение `require 'thread'`.)

```
q1 = MyQueue.new                    # используется любой массив
q2 = MyQueue.new(my_array)          # используется конкретный массив
q3 = MyQueue.new(Queue.new)         # используется Queue (thread.rb)
q4 = MyQueue.new(SizedQueue.new)    # используется SizedQueue (thread.rb)
```

Так, объекты `q3` и `q4` волшебным образом становятся потокобезопасными, поскольку делегируют управление потокобезопасному объекту (если, конечно, какой-нибудь не показанный здесь код не нарушит эту гарантию).

Существует также класс `SingleForwardable`, который воздействует на один экземпляр, а не на класс в целом. Это полезно, если вы хотите, что какой-то конкретный объект делегировал управление другому объекту, а все остальные объекты того же класса этого не делали.

И последний вариант – ручное делегирование. В Ruby очень просто обернуть один объект другим, и это позволяет реализовать очередь следующим образом:

```
class MyQueue
   def initialize(obj=[])
      @queue = obj          # делегировать этому объекту
   end

   def enqueue(arg)
      @queue.push(arg)
   end

   def dequeue(arg)
      @queue.shift(arg)
   end

   %i[clear empty? length size <<].each do |name|
      define_method(name){|*args| @queue.send(name, *args) }
   end

   # Прочий код...
end
```

Как видите, в некоторых случаях использование для делегирования библиотеки не дает заметного выигрыша по сравнению с написанием собственного простого кода.

Быть может, у вас возник вопрос: «Что лучше – делегирование или наследование?» Но это не совсем правильный вопрос. В некоторых ситуациях делегирование оказывается более подходящим решением. Предположим, к примеру, что имеется класс, у которого уже есть родитель. Унаследовать еще от одного родителя мы не можем (в Ruby множественное наследование запрещено), но делегирование (в той или иной форме) вполне допустимо.

11.2.11. Автоматическое определение методов чтения и установки на уровне класса

Мы уже рассматривали методы attr_reader, attr_writer и attr_accessor, которые немного упрощают определение методов чтения и установки атрибутов экземпляра. А как быть с атрибутами уровня класса?

В Ruby нет аналогичных средств для их автоматического создания. Но можно написать нечто подобное самостоятельно. Нужно лишь открыть синглетный класс и воспользоваться в нем семейством методов attr.

Получающиеся переменные экземпляра для синглетного класса станут переменными экземпляра класса. Часто это оказывается лучше, чем переменные класса, поскольку они принадлежат данному и только данному классу, не распространяясь вверх и вниз по иерархии наследования.

```
class MyClass

  @alpha = 123                # Инициализировать @alpha

  class << self
    attr_reader :alpha          # MyClass.alpha()
    attr_writer :beta           # MyClass.beta=()
    attr_accessor :gamma        # MyClass.gamma() и
  end                           #   MyClass.gamma=()

  def MyClass.look
    puts "#@alpha, #@beta, #@gamma"
  end

  #...
end


puts MyClass.alpha            # 123
MyClass.beta = 456
MyClass.gamma = 789
puts MyClass.gamma            # 789

MyClass.look                  # 123, 456, 789
```

Как правило, класс без переменных экземпляра бесполезен. Но здесь мы их для краткости опустили.

11.2.12. Поддержка различных стилей программирования

Brother, can you paradigm?

— Граффити на здании IBM в Остине, 1989

В различных кругах популярны разные философии программирования. Часто их трудно охарактеризовать с точки зрения объектной ориентированности или динамичности, а некоторые вообще не зависят от того, является ли язык динамическим или объектно-ориентированным.

Поскольку мы отнюдь не являемся экспертами в этих вопросах, то будем полагаться в основном на чужие слова. Так что воспринимайте следующие абзацы с некоторой долей скепсиса.

Некоторые программисты предпочитают стиль ООП на основе *прототипов* (или ООП без классов). В этом мире объект не описывается как экземпляр класса, а строится с нуля. На базе такого прототипа могут создаваться другие объекты. В Ruby есть рудиментарная поддержка такого стиля программирования, поскольку допускаются синглетные методы, имеющиеся только у отдельных объектов, а метод `clone` клонирует синглеты. Интересующийся читатель может также обратить внимание на простой класс `OpenStruct` для построения объектов в духе языка Python; не забывайте также о том, как работает метод `method_missing`.

Парочка ограничений в Ruby препятствуют реализации ООП без классов. Некоторые объекты, например `Fixnum`, хранятся как непосредственные значения, а не ссылки, поэтому не могут иметь синглетных методов. В будущем это положение, вероятно, изменится, но пока невозможно предсказать, когда это произойдет.

В функциональном программировании (ФП) упор делается на вычисление выражений, а не на исполнение команд. Функциональным называется язык, поддерживающий ФП, но на этом всякая определенность заканчивается. Почти все согласятся, что Haskell – настоящий функциональный язык, а Ruby таковым, безусловно, не является.

Но в Ruby есть минимальная поддержка ФП, он располагает богатым набором методов для манипулирования массивами (списками) и поддерживает объекты `Proc`, позволяющие инкапсулировать и многократно вызывать код. Ruby также допускает сцепление методов, весьма распространенное в ФП. Предпринимались попытки создать библиотеку, которая стала бы «уровнем совместимости» с ФП, заимствуя некоторые идеи из языка Haskell. Пока эти попытки ни к чему завершенному не привели.

Интересна идея аспектно-ориентированного программирования (АОП). Это попытка рассечь модульную структуру программы. Иными словами, некоторые задачи и механизмы системы разбросаны по разным участкам кода, а не собраны в одном месте. То есть мы пытаемся придать модульность вещам, которым в традиционном объектно-ориентированном или процедурном программировании с тру-

дом поддаются «модуляризации». Взгляд на программу оказывается перпендикулярен обычному.

Разумеется, Ruby создавался без учета АОП. Но это гибкий и динамический язык, поэтому не исключено, что такой подход может быть реализован в виде библиотеки. Толику АОП привносят в Ruby метод `Module#prepend` и gem-пакет `aspector`.

Идея «проектирования по контракту» (Design by Contract – DBC) хорошо знакома поклонникам языка Eiffel, хотя и вне этого круга она тоже известна. Смысл состоит в том, что некоторый кусок кода (метод или класс) реализует контракт; чтобы код правильно работал, должны выполняться определенные предусловия, и тогда гарантируется, что после завершения работы будут выполнены некоторые постусловия. Надежность системы можно существенно повысить, введя возможность формулировать контракт явно и автоматически проверять его во время выполнения. Полезность такого подхода подкрепляется наследованием информации о контракте при расширении классов.

В язык Eiffel методология DBC встроена явно, в Ruby – нет. Однако имеются по крайней мере две работающие библиотеки, реализующие DBC, и обе достойны изучения.

Паттерны проектирования стали темой оживленных дискуссий на протяжении последних нескольких лет. Конечно, они мало зависят от конкретного языка и могут быть реализованы на самых разных языках. Но необычайная гибкость Ruby, возможно, делает их практически более полезными, чем в других средах. Хорошо известные примеры приведены в других местах; паттерн Visitor (Посетитель) реализуется стандартным итератором `each`, а многие другие паттерны входят в стандартный дистрибутив Ruby (например, библиотеки `delegator` и `singleton`).

Мы больше не используем термин «экстремальное программирование» (Extreme Programming – XP), но многие его принципы прочно вошли в нашу культуру. Эта методика поощряет (среди прочего) раннее тестирование и постоянный рефакторинг.

«Культура тестирования» не зависит от языка, но в одних языках реализуется проще, чем в других. Разумеется, мы убеждены, что в Ruby рефакторинг (ручной) проще, чем во многих языках, но это субъективное мнение. Однако, наличие библиотеки `RSpec` (и других) позволяет «поженить» Ruby и XP. Эта библиотека облегчает автономное тестирование компонентов, она функциональна богата, проста в использовании и доказала свою полезность в ходе разработки эксплуатируемых в настоящее время программ на Ruby. Мы горячо поддерживаем рекомендуемое XP раннее и частое тестирование, а тем, кто желает воплотить этот совет в Ruby, предлагаем ознакомиться с `RSpec`. (`MiniTest` – еще один отличный пакет, включающий некоторые возможности для тех, кто предпочитает стиль `Test::Unit`.)

Когда вы будете читать этот раздел, многие обсуждаемые в нем технологии нарастят мясо на костях. Как обычно, самую актуальную информацию можно найти в сети. Трудно перечислять сетевые ресурсы в печатном издании, поскольку они постоянно изменяются. Поисковая машина – ваш лучший друг.

11.3. Динамические механизмы

Скайнет осознал себя в 2:14 утра
по восточному времени 29 августа 1997 года.

— Терминатор 2. Судный День

Многие читатели имеют опыт прежний работы со статическими языками, например С. Им я адресую риторический вопрос: «Можете ли вы представите себе написанную на С функцию, которая принимает строку, рассматривает ее как имя переменной и возвращает значение этой переменной?». Нет? А как насчет того, чтобы удалить или заменить определение функции? А перехватить обращения к несуществующим функциям?

В Ruby все это возможно. К подобным возможностям нужно привыкнуть, их легко употребить во вред. Все эти идеи появились отнюдь не вчера (они стары по крайней мере так же, как язык LISP) и считаются «проверенными и доказанными» в сообществах пользователей Scheme и Smalltalk. Много динамических возможностей есть в языке Python и даже в Java, поэтому мы ожидаем, что со временем их популярность будет только расти.

11.3.1. Динамическая интерпретация кода

Глобальная функция `eval` компилирует и исполняет строку, содержащую код на Ruby. Это очень мощный (и вместе с тем опасный) механизм, поскольку позволяет строить подлежащий исполнению код во время работы программы. Например, в следующем фрагменте считываются строки вида «имя = выражение», затем каждое выражение вычисляется, а результат сохраняется в хэше, индексированном именем переменной.

```
parameters = {}

ARGF.each do |line|
  name, expr = line.split(/\s*=\s*/, 2)
  parameters[name] = eval expr
end
```

Пусть на вход подаются следующие строки:

```
a = 1
b = 2 + 3
c = `date`
```

Тогда в результате мы получим такой хэш: {"a"=>1, "b"=>5, "c"=>"Sat Jul 26 00:51:48 PDT 2014\n"}. На этом примере демонстрируется также опасность вычисления с помощью `eval` строк, содержимое которых вы не контролируете; злонамеренный пользователь может подсунуть строку `d=`rm *`` и стереть все сделанное вами за день.

В Ruby есть еще три метода, которые интерпретируют код «на лету»: class_eval, module_eval и instance_eval. Первые два — синонимы, и все они выполняют одно и то же: интерпретируют строку или блок, но при этом изменяют значение псевдопеременной self так, что она указывает на объект, от имени которого эти методы вызваны. Наверное, чаще всего метод class_eval применяется для добавления методов в класс, на который у вас имеется только ссылка.

Мы продемонстрируем это в коде метода hook_method в примере утилиты Trace в разделе 11.4.3. Другие примеры вы найдете в динамических библиотечных модулях, например, delegate.rb.

Метод eval позволяет также вычислять локальные переменные в контексте, не принадлежащем их области видимости. Мы не рекомендуем легкомысленно относиться к этой возможности, но знать, что она существует, полезно.

Ruby ассоциирует локальные переменные с блоками, с определениями высокоуровневых конструкций (класса, модуля и метода) и с верхним уровнем программы (кодом, расположенным вне любых определений). С каждой из этих областей видимости ассоциируются привязки переменных и другие внутренние детали.

Наверное, самым главным потребителем информации о привязках является программа pry — интерактивная консоль для Ruby, которая не только пользуется привязками для того, чтобы отделить собственные переменные от тех, которые принадлежат вводимой программе, но еще и добавляет метод Binding#pry, который открывает новую интерактивную консоль в контексте данной привязки. Мы еще вернемся к методу pry в главе 16 «Тестирование и отладка».

Можно инкапсулировать текущую привязку в объекте с помощью метода Kernel#binding. Тогда вы сможете передать привязку в виде второго параметра методу eval, установив контекст исполнения для интерпретируемого кода.

```
def some_ethod
  a = "local variable"
  return binding
end

the_binding = some_method
eval "a", the_binding   # "local variable"
```

Интересно, что информация о наличии блока, ассоциированного с методом, сохраняется как часть привязки, поэтому возможны такие трюки:

```
def some_method
  return binding
end

the_binding = some_method { puts "hello" }
eval "yield", the_binding                 # hello
```

11.3.2. Метод const_get

Метод const_get получает значение константы с заданным именем из модуля или класса, которому она принадлежит.

```
str = "PI"
Math.const_get(str)     # Значение равно Math::PI
```

Это способ избежать обращения к методу eval, которое иногда считается неэлегантным. Такой подход дешевле с точки зрения потребления ресурсов и безопаснее. Есть и другие аналогичные методы: instance_variable_set, instance_variable_get и define_method.

Метод const_get действительно работает быстрее, чем eval. В неформальных тестах – на 350% быстрее, хотя у вас может получиться другой результат. Но так ли это важно? Ведь в тестовой программе на 10 миллионов итераций цикла все равно ушло менее 30 секунд.

Истинная полезность метода const_get в том, что его проще читать, он более специфичен и лучше самодокументирован. Даже если бы он был всего лишь синонимом eval, все равно это стало бы большим шагом вперед.

Еще один прием описан в следующем разделе.

11.3.3. Получение класса по имени

Следующий вопрос мы встречали многократно. Пусть дана строка, содержащая имя класса, как можно создать экземпляр этого класса?

Имена классов в Ruby – константы в «глобальном» пространстве имен, то есть члены класса Object. Это означает, что правильный способ – воспользоваться методом const_get, который мы только что рассмотрели.

```
classname = "Array"
klass = Object.const_get(classname)
x = klass.new(4, 1)         # [1, 1, 1, 1]
```

Если константа определена внутри пространства имен, то достаточно задать строку, в которой это пространство имен отделено от имени константы двумя двоеточиями (как если бы вы писали прямо на Ruby):

```
class Alpha
  class Beta
    class Gamma
      FOOBAR = 237
    end
  end
end

str = "Alpha::Beta::Gamma::FOOBAR"
val = Object.const_get(str)          # 237
```

11.3.4. Метод define_method

Помимо ключевого слова def, единственный нормальный способ добавить метод в класс или объект – воспользоваться методом define_method, причем он позволяет сделать это во время выполнения.

Конечно, в Ruby практически все происходит во время выполнения. Если окружить определение метода обращениями к puts, как в примере ниже, вы это сами увидите.

```
class MyClass
  puts "до"

  def meth
    #...
  end

  puts "после"
end
```

Но внутри тела метода или в другом аналогичном месте нельзя заново открыть класс (если только это не синглетный класс). В таком случае в прежних версиях Ruby приходилось прибегать к помощи eval, теперь же у нас есть метод define_method. Он принимает символ (имя метода) и блок (тело метода).

Отметим, однако, что define_method – закрытый метод. Это означает, что обращение к нему из определения класса или метода работает нормально:

```
class MyClass
   define_method(:body_method) { puts "Тело класса." }

   def self.new_method(name, &block)
      define_method(name, &block)
   end
end

MyClass.new_method(:class_method) { puts "Метод класса." }

x = MyClass.new
x.body_method            # Печатается "Тело класса."
x.class_method           # Печатается "Метод класса."
```

Можно даже создать метод экземпляра, который будет динамически определять другие методы экземпляра:

```
class MyClass
   def new_method(name, &block)
      self.class.send(:define_method, name, &block)
   end
end

x = MyClass.new
x.new_method(:instance_method) { puts "Метод экземпляра." }
x.mymeth                 # Печатается "Метод экземпляра."
```

Здесь метод экземпляра тоже определен динамически. Изменился только способ реализации метода new_method. Обратите внимание на трюк с send, позволивший нам обойти закрытость метода define_method. Он работает, потому что метод send позволяет вызывать закрытые методы. (Некоторые сочтут это «дыркой»; как бы то ни было, пользоваться этим механизмом следует с осторожностью.)

По поводу метода define_method нужно сделать еще одно замечание. Он принимает блок, а в Ruby блок – замыкание. Это означает, что в отличие от обычного опре-

деления метода, мы запоминаем контекст, в котором метод был определен. Следующий пример практически бесполезен, но этот момент иллюстрирует:

```
class MyClass
  def self.new_method(name, &block)
    define_method(name, &block)
  end
end

a,b = 3,79

MyClass.new_method(:compute) { a*b }
x = MyClass.new
puts x.compute          # 237

a,b = 23,24
puts x.compute          # 552
```

Смысл здесь в том, что новый метод может обращаться к переменным в исходной области видимости блока, хотя сама эта область более не существует и никаким другим способом недоступна. Иногда это бывает полезно, особенно в случае метапрограммирования или при разработке графических интерфейсов, когда нужно определить методы обратного вызова, реагирующие на события.

Отметим, что замыкание имет доступ только к переменным с такими же именами. Изредка из-за этого могут возникать сложности. Ниже мы воспользовались методом define_method, чтобы предоставить доступ к переменной класса (вообще-то это следует делать не так, но для иллюстрации подойдет):

```
class SomeClass
  @@var = 999

  define_method(:peek) { @@var }
end

x = SomeClass.new
p x.peek            # 999
```

А теперь попробуем проделать этот трюк с переменной экземпляра класса:

```
class SomeClass
  @var = 999

  define_method(:peek) { @var }
end

x = SomeClass.new
p x.peek            # печатается nil
```

Мы ожидали, что будет напечатано 999, а получили nil. С другой стороны, такой код работает правильно:

```
class SomeClass
  @var = 999
```

```
  x = @var

  define_method(:peek) { x }
end

x = SomeClass.new
p x.peek      # 999
```

Так что же происходит? Да, замыкание действительно запоминает переменные в текущем контексте. Но хотя замыкание и знает о переменных в своей области видимости, контекст метода – это контекст объекта, а не самого класса.

Поскольку имя `@var` в контексте экземпляра относится к переменной экземпляра объекта, а не класса, то переменная экземпляра класса оказывается скрыта переменной экземпляра объекта, хотя последняя никогда не использовалась и технически не существует.

При работе с отдельными объектами удобной альтернативой открытию синглетного класса и определению в нем метода является метод `define_singleton_method`. Работает он аналогично `define_method`.

Хотя определять методы во время выполнения с помощью `eval` разрешено, лучше этого не делать. Во всех таких случаях может и должен использоваться метод `define_method`. Некоторые тонкости, вроде рассмотренной выше, не должны вас останавливать.

11.3.5. Получение списка определенных сущностей

API отражения в Ruby позволяет опрашивать классы и объекты во время выполнения. Рассмотрим методы, имеющиеся для этой цели в `Module`, `Class` и `Object`.

В модуле `Module` есть метод `constants`, который возвращает массив всех констант, определенных в системе (включая имена классов и модулей). Метод `nesting` возвращает массив всех вложенных модулей, видимых в данной точке программы.

Метод экземпляра `Module#ancestors` возвращает массив всех предков указанного класса или модуля.

```
list = Array.ancestors     # [Array, Enumerable, Object, Kernel, BasicObject]
```

Метод `constants` возвращает список всех констант, доступных в данном модуле. Включаются также его предки.

```
list = Math.constants # [:DomainError, :PI, :E]
```

Метод `class_variables` возвращает список всех переменных класса в данном классе и его суперклассах. Метод `included_modules` возвращает список модулей, включенных в класс.

```
class Parent
  @@var1 = nil
end
class Child < Parent
  @@var2 = nil
end
```

```
list1 = Parent.class_variables   # ["@@var1"]
list2 = Array.included_modules   # [Enumerable, Kernel]
```

Методы instance_methods и public_instance_methods класса Class – синонимы; они возвращают список открытых методов экземпляра, определенных в классе. Методы private_instance_methods и protected_instance_methods ведут себя аналогично. Любой из них принимает необязательный флаг, по умолчанию равный true; если его значение равно false, то суперклассы не учитываются, так что список получается меньше.

```
n1 = Array.instance_methods.size                 # 174
n2 = Array.public_instance_methods.size          # 174
n3 = Array.public_instance_methods(false).size   # 90
n4 = Array.private_instance_methods.size         # 84
n5 = Array.protected_instance_methods.size       # 0
```

В классе Object есть аналогичные методы, применяющиеся к экземплярам (листинг 11.15). Метод methods возвращает список всех методов, которые можно вызывать для данного объекта. Метод public_methods возвращает список открытых методов и принимает параметр, равный по умолчанию true, который говорит, нужно ли включать также методы суперклассов. Методы private_methods, protected_methods и singleton_methods тоже принимают такой параметр и возвращают именно то, что и ожидается.

Листинг 11. Отражение и переменные экземпляра

```
class SomeClass

  def initialize
    @a = 1
    @b = 2
  end

  def mymeth
    #...
  end

  protected :mymeth

end


x = SomeClass.new

def x.newmeth
  # ...
end
iv = x.instance_variables        # ["@b", "@a"]

p x.methods.size                   # 61

p x.public_methods.size            # 60
```

```
p x.public_methods(false).size     # 1

p x.private_methods.size           # 85
p x.private_methods(false).size    # 1

p x.protected_methods.size         # 1
p x.singleton_methods.size         # 1
```

11.3.6. Удаление определений

Вследствие динамичности Ruby практически все, что можно определить, можно и уничтожить. Например, это может пригодиться для того, чтобы «развязать» два куска кода в одной и той же области действия, избавляясь от переменных после того, как они были использованы. Другой повод – запретить вызовы некоторых потенциально опасных методов. Но по какой бы причине вы ни удаляли определение, делать это нужно крайне осторожно, чтобы не создать себе проблемы во время отладки.

Радикальный способ уничтожить определение – воспользоваться ключевым словом `undef` (неудивительно, что его действие противоположно действию `def`). Уничтожать можно определения методов, локальных переменных и констант на верхнем уровне. Хотя имя класса – тоже константа, удалить определение класса таким способом невозможно.

```
def asbestos
  puts "Теперь не огнеопасно"
end

tax = 0.08

PI = 3

asbestos
puts "PI=#{PI}, tax=#{tax}"

undef asbestos
undef tax
undef PI

# Любое обращение к этим трем именам теперь приведет к ошибке.
```

Внутри определения класса, можно уничтожать определения методов и констант в том же контексте, в котором они были определены. Нельзя применять `undef` внутри определения метода, а также к переменной экземпляра.

Существуют (определены в классе `Module`) также методы `remove_method` и `undef_method`. Разница между ними тонкая: `remove_method` удаляет текущее (или ближайшее) определение метода, а `undef_method` удаляет его также из всех суперклассов, не оставляя от метода даже следа. В листинге 11.16 это различие проиллюстрировано.

Листинг 11.16. Методы remove_method и undef_method

```
class Parent

  def alpha
    puts "alpha: родитель"
  end

  def beta
    puts "beta: родитель"
  end
end

class Child < Parent

  def alpha
    puts "alpha: потомок"
  end

  def beta
    puts "beta: потомок"
  end

  remove_method :alpha   # Удалить "этот" alpha
  undef_method :beta     # Удалить все beta

end


x = Child.new

x.alpha          # alpha: родитель
x.beta           # Ошибка!
Метод remove_const удаляет константу.
module Math

  remove_const :PI

end

# PI больше нет!
```

Отметим, что таким способом можно удалить и определение класса (потому что идентификатор класса – это просто константа):

```
class BriefCandle
  #...
end

out_out = BriefCandle.new
class Object
```

```
  remove_const :BriefCandle
end

BriefCandle.new        # NameError: uninitialized constant BriefCandle
out_out.class.new      # Еще один экземпляр BriefCandle
```

Такие методы, как remove_const и remove_method являются закрытыми (что и понятно). Поэтому во всех примерах они вызываются изнутри определения класса или модуля, а не снаружи.

11.3.7. Ссылки на несуществующие константы

Метод const_missing вызывается при попытке обратиться к неизвестной константе. В качестве параметра ему передается символ, ссылающийся на константу. Ситуация аналогична методу method_missing, который мы рассмотрим в разделе 11.3.8.

Чтобы перехватывать обращения к отсутствующим константам глобально, определите следующий метод в самом классе Module (это родитель класса Class).

```
class Module
  def const_missing(x)
    "из Module"
  end
end

class X
end

p X::BAR        # "из Module"
p BAR           # "из Module"
p Array::BAR    # "из Module"
```

Можно выполнить в нем любые действия: вернуть фиктивное значение константы, вычислить его и т.д. Помните класс Roman из главы 6? Воспользуемся им, чтобы трактовать любые последовательности римских цифр, как числовые константы:

```
class Module
  def const_missing(name)
    Roman.decode(name)
  end
end

year1 = MCMLCCIV        # 1974
year2 = MMXIV           # 2014
```

Если такая глобальность вам не нужна, определите этот метод на уровне конкретного класса. Тогда он будет вызываться из этого класса и его потомков.

```
class Alpha
  def self.const_missing(sym)
    "В Alpha нет #{sym}"
  end
```

В конечном итоге будет вызван method_missing, определенный в классе Object, который и возбудит исключение.

Чтобы использование метода method_missing было безопасным, всегда определяйте также метод respond_to_missing?. В этом случае Ruby сможет предоставить результаты как методу respond_to?, так и методу method. Ниже приведен пример такой техники:

```
class CommandWrapper      # повторно открыть класс

   def respond_to_missing?(method, include_all)
      system("which #{method} > /dev/null")
   end

end

cw = CommandWrapper.new
cw.respond_to?(:foo)      # false
cw.method(:echo)          # #<Method: CommandWrapper#echo>
cw.respond_to?(:echo)     # true
```

Второй параметр (include_all) определяет, должен ли метод просматривать также закрытые и защищенные методы. В данном примере ни тех, ни других нет, поэтому их можно игнорировать.

Хотя метод respond_to_missing? является логическим дополнением к method_missing, Ruby относится к этому безучастно. Можно написать только один из двух методов, если вас не смущает крайне непоследовательное поведение.

11.3.9. Повышение безопасности с помощью taint

Вследствие динамической природы Ruby в некоторых местах программы возможны бреши в защите. В качестве тривиального примера возьмем метод eval; если ему передается строка, полученная из внешнего мира, то злонамеренный пользователь может заставить программу выполнить произвольный код, например, с помощью такой неувядающей строки: system("nohup rm -rf / &"). Но даже и без злого умысла данные из внешних источников могут стать причиной серьезных проблем.

Первый рубеж обороны от таких атак в Ruby – *уровень безопасности*. Он хранится в поточно-локальной переменной и по умолчанию равен 0. Присвоив число переменной $SAFE, вы установите уровень безопасности, который ни при каких обстоятельствах нельзя понизить.

Если уровень безопасности не меньше 1, то Ruby блокирует некоторые опасные действия с помощью «запятнанных» (tainted) объектов. У любого объекта имеется флаг состояния: запятнанный или незапятнанный. Если объект происходит из «внешнего мира» (например, создан из чего-то, заданного в командной строке или в параметре URL), то он автоматически становится запятнанным. Это пятно передается и всем объектам, созданным при его участии.

Многие базовые методы (и, прежде всего, eval) ведут себя по-другому или возбуждают исключение, если им передают запятнанные данные при повышенном уровне безопасности. Рассмотрим следующие примеры использования строк:

```
str1 = "puts 'Ответ: '"
str2 = ARGV.first.dup       # "3*79" (продублирована, чтобы разморозить)
str1.tainted?               # false
str2.tainted?               # true

str1.taint                  # При желании мы можем
str2.untaint                # изменить состояние запятнанности

eval(str1)                  # Печатается: Ответ:
puts eval(str2)             # Печатается: 237

$SAFE = 1

eval(str1)                  # Возбуждается SecurityError: Insecure operation
puts eval(str2)             # Печатается: 237
```

Полный перечень изменений в поведении в зависимости от значения переменной $SAFE приведен в табл. 11.1.

Таблица 11.1. Уровни безопасности (переменная $SAFE)

	0	1	2	3
Переменные $RUBYLIB и $RUBYOPT игнорируются		x	x	x
Текущий каталог не включается в список путей поиска		x	x	x
Отключаются следующие флаги в командной строке: -e -i -l -s -x -S		x	x	x
Переменная $PATH игнорируется, если хотя бы в один указанный в ней каталог могут писать все		x	x	x
Запрещаются манипуляции с каталогом, имя которого задано в запятнанной строке		x	x	x
Метод chroot не принимает запятнанную строку		x	x	x
Методы load и require не принимают запятнанную строку (если она не обернута)		x	x	x
Запрещаются манипуляции с файлом или каналом, имя которого задано в запятнанной строке		x	x	x
Методы system и exec не принимают запятнанную строку		x	x	x
Методы glob, eval и trap не принимают запятнанную строку		x	x	x
Запрещаются любые манипуляции с каталогами и использование chroot			x	x
Запрещается загрузка файла из каталога, в который могут писать все			x	x
Запрещается загрузка файла, имя которого задано в запятнанной строке и начинается знаком ~		x	x	
Запрещается использовать следующие методы класса File: chmod chown lstat truncate flock			x	x
Запрещается использовать следующие методы класса File: stat umask			x	x

Таблица 11.1 (окончание)

	0	1	2	3
Запрещается использовать следующие методы класса IO: ioctl stat			x	x
Запрещается использовать следующие методы класса Object: fork syscall trap			x	x
Запрещается использовать следующие методы класса Process: setpgid setsid		x	x	
Запрещается использовать следующие методы класса Process: setpriority egid=		x	x	
Запрещается использовать метод trap для обработки сигналов			x	x
Все объекты создаются запятнанными				x
Объекты нельзя сделать незапятнанными				x

11.3.10. Определение чистильщиков для объектов

У классов в Ruby есть конструкторы (методы new и initialize), но нет деструкторов (методов, которые уничтожают объекты). Объясняется это тем, что в Ruby применяется алгоритм сборки мусора для удаления объектов, на которые не осталось ссылок; поэтому деструктор просто не имеет смысла.

Однако, тем, кто приходит в Ruby из таких языков, как C++, этот механизм представляется необходимым, поэтому они часто спрашивают, как написать код очистки уничтожаемых объектов. Простой ответ звучит так: невозможно сделать это надежно. Но можно написать код, который будет вызываться, когда сборщик мусора уничтожает объект.

```
a = "hello"
puts "Для строки 'hello' ид объекта равен #{a.id}"
ObjectSpace.define_finalizer(a) { |id| puts "Уничтожается #{id}" }
puts "Нечего убирать"
GC.start
a = nil
puts "Исходная строка - кандидат на роль мусора"
GC.start
```

Этот код выводит следующее:

```
Для строки 'hello' ид объекта равен 537684890
Нечего убирать
Исходная строка - кандидат на роль мусора
Уничтожается 537684890
```

Подчеркнем, что к моменту вызова чистильщика объект уже фактически уничтожен. Попытка преобразовать идентификатор в ссылку на объект с помощью метода ObjectSpace._id2ref приведет к исключению RangeError с сообщением о том, что вы пытаетесь воспользоваться уничтоженным объектом.

Имейте также в виду, что в Ruby применяется консервативный механизм сборки мусора. Нет гарантии, что любой объект будет убран до завершения программы.

Однако все это может оказаться и ненужным. В Ruby существует стиль программирования, в котором для инкапсуляции работы с ресурсами служат блоки. В конце блока ресурс освобождается, и жизнь продолжается без помощи чистильщиков. Рассмотрим, например, блочную форму метода File.open:

```
File.open("myfile.txt") do |file|
  line1 = file.gets
  # ...
end
```

Здесь в блок передается объект File, а по выходе из блока файл закрывается, причем все это делается под контролем метода open. Метод File.open ради эффективности написан на C, но на чистом Ruby он мог бы выглядеть так:

```
def File.open(name, mode = "r")
  f = os_file_open(name, mode)
  if block_given?
    begin
      yield f
    ensure
      f.close
    end
    return nil
  else
    return f
  end
end
```

Мы проверяем наличие блока. Если блок был передан, то мы вызываем его, передавая открытый файл. Делается это в контексте секции begin-end, которая гарантирует, что файл будет закрыт по выходе из блока, даже если произойдет исключение.

11.4. Интроспекция программы

Гибкость поведения Ruby во время выполнения не ограничивается рассмотренными выше динамическими возможностями. Написанный на Ruby код способен вытворять такие вещи по время выполнения программы, которые программисту на C показались бы немыслимыми.

Среди прочего можно узнать имя вызывающей функции и автоматически перечислить все определенные пользователем элементы (объекты, классы и функции). Наличие таких гибких средств исследования и изменения элементов программы во время выполнения упрощает решение многих задач.

В этом разделе мы рассмотрим несколько трюков, которые Ruby допускает во время выполнения. Ниже, в главе 16 мы увидим несколько библиотек, построенных на базе этих трюков.

11.4.1. Обход пространства объектов

Исполняющая система Ruby должна отслеживать все известные объекты (хотя бы для того, чтобы убрать мусор, когда на объект больше нет ссылок). Информацию о них можно получить с помощью метода ObjectSpace.each_object. Если задать класс или модуль в качестве параметра each_object, то будут возвращены лишь объекты указанного типа:

```
ObjectSpace.each_object do |obj|
  printf "%20s: %s\n", obj.class, obj.inspect
end

# Печатается:
# Bignum: 134168290416582806546366068851909766153
# Bignum: 9223372036854775807
```

Если вас интересует лишь количество созданных объектов каждого типа, то метод `count_objects` вернет хэш, в котором ключом является тип объекта, а значением – счетчик.

```
require 'pp'
p ObjectSpace.count_objects
# {:TOTAL=>33013, :FREE=>284, :T_OBJECT=>2145, :T_CLASS=>901,
   :T_MODULE=>32, :T_FLOAT=>4, :T_STRING=>18999, :T_REGEXP=>167,
   :T_ARRAY=>4715, :T_HASH=>325, :T_STRUCT=>2, :T_BIGNUM=>4,
   :T_FILE=>8, :T_DATA=>1518, :T_MATCH=>205, :T_COMPLEX=>1,
   :T_NODE=>3663, :T_ICLASS=>40}
```

Модуль `ObjectSpace` полезен также для определения чистильщиков объектов (см. раздел 11.3.10).

11.4.2. Просмотр стека вызовов

И вы задаете себе вопрос:
Как же я оказался здесь?

– Talking Heads,
«Once in a Lifetime»

Иногда необходимо знать, кто вызвал метод. Эта информация полезна, если, например, произошло неисправимое исключение. Метод `caller`, определенный в модуле `Kernel`, дает ответ на этот вопрос. Он возвращает массив строк, в котором первый элемент соответствует вызвавшему методу, следующий – методу, вызвавшему этот метод, и т.д.

```
def func1
  puts caller[0]
 end

def func2
  func1
end
```

11.4.3. Отслеживание изменений в определении класса или объекта

А зачем, собственно? Кому интересны изменения, которым подвергался класс?

Одна возможная причина – желание следить за состоянием выполняемой программы на Ruby. Быть может, мы реализуем графический отладчик, который должен обновлять список методов, добавляемых «на лету».

Другая причина: мы хотим вносить соответствующие изменения в другие классы. Например, мы разрабатываем модуль, который можно включить в определение любого класса. С момента включения будут трассироваться любые обращения к методам этого класса. Возможная реализация такого модуля показана в листинге 11.17.

Листинг 11.17. Трассирующий модуль

```
module Tracing

   def self.hook_method(const, meth)
      const.class_eval do
         alias_method "untraced_#{meth}", "#{meth}"
         define_method(meth) do |*args|
            puts "вызван метод #{meth} с параметрами (#{args.join(', ')})"
            send("untraced_#{meth}",*args)
         end
      end
   end

   def self.included(const)
      const.instance_methods(false).each do |m|
         hook_method(const, m)
      end

      def const.method_added(name)
         return if @disable_method_added
         puts "Метод #{name} добавлен в класс #{self}"
         @disable_method_added = true
         Tracing.hook_method(self, name)
         @disable_method_added = false
      end

      if const.is_a?(Class)
         def const.inherited(name)
            puts "Класс #{name} унаследован от #{self}"
         end
      end

      if const.is_a?(Module)
         def const.extended(name)
            puts "Класс #{name} расширил себя с помощью #{self}"
         end
```

```
        def const.included(name)
          puts "Класс #{name} включил в себя #{self}"
        end
      end

      def const.singleton_method_added(name, *args)
        return if @disable_singleton_method_added
        return if name == :singleton_method_added

        puts "Метод класса #{name} добавлен в класс #{self}"
        @disable_singleton_method_added = true
        singleton_class = (class << self; self; end)
        Tracing.hook_method(singleton_class, name)
        @disable_singleton_method_added = false
      end
   end
end
```

В этом модуле два основных метода. Первый, `trace_method`, работает прямолинейно. При добавлении метода ему сразу назначается синоним `untraced_name`. Исходный метод заменяется кодом трассировки, который выводит имя и параметры метода, а затем вызывает метод, к которому было обращение.

Обратите внимание на использование конструкции `alias_method`. Работает она почти так же, как `alias`, но только для методов (да и сама является методом, а не ключевым словом). Принимает имена методов в виде символов или строк.

Второй метод, `included`, вызывается при каждой вставке модуля в класс. Наша версия делает несколько вещей для отслеживания будущих вызовов метода и изменений в классе.

Во-первых, она вызывает метод `trace_method` для каждого метода, уже определенного в целевом классе. Это дает возможность трассировать методы, которые были определены до включения модуля `Tracing`. Затем она определяет метод класса `method_added`. В результате любой добавленный позже метод тоже будет обнаружен и протрассирован.

```
class MyClass
   def first_meth
   end
end

class MyClass
   include Tracing

   def second_meth(x, y)
   end

   # Выводится: метод second_meth добавлен в класс MyClass
end

m = MyClass.new
```

```
m.first_meth
# Выводится: вызван метод first_meth с параметрами ()
m.second_meth(1, 'cat')
# Выводится: вызван метод second_meth с параметрами (1, 'cat')
```

Далее следуют условные предложения, к которым мы вернемся чуть позже. Наконец, модуль обнаруживает добавление новых синглетных методов, для чего определяет метод `singleton_method_added` в трассирующем классе. (Напомним, что синглетный метод – то, что мы обычно называем методом класса, поскольку `Class` – это объект.)

```
class MyClass
   def self.a_class_method(options)
   end
end

MyClass.a_class_method(green: "super")

# Выводится:
# Метод класса a_class_method добавлен в класс MyClass
# Метод a_class_method вызван с параметрами ({:green=>"super"})
```

Обратный вызов `singleton_method_added` определен последним, чтобы не печаталась трассировка других методов класса, добавленных модулем `Tracing`. Отметим также, что (быть может, вопреки ожиданиям) модуль может отслеживать добавление в класс самого себя, следовательно, мы должны явно исключить его из трассировки.

В первом условном предложении определяется метод `inherited` для каждого целевого класса `Class`. Впоследствии он будет вызываться всякий раз, как этому классу наследует какой-то другой:

```
class MySubClass < MyClass
end

# Выводится: Класс MySubClass унаследован от MyClass
```

Наконец, в условном предложении для целей типа `Module` определяются обратные вызовы `included` и `extended`, чтобы можно было трассировать операции включения и расширения целевого модуля:

```
module MyModule
   include Tracing
end

class MyClass
   include MyModule
   extend MyModule
end

# Выводится:
# Класс MyClass включил в себя MyModule
# Класс MyClass расширил себя с помощью MyModule
```

11.4.4. Мониторинг выполнения программы

Программа на Ruby может следить за собственным выполнением. У этой возможности есть много применений: большая их часть связана с профилированием и отладкой и будет рассмотрена в главе 16.

В некоторых случаях библиотеки вообще не нужны, потому что задачу можно решить на Ruby, воспользовавшись интроспекцией. Такой подход резко снижает производительность (и потому в режиме эксплуатации не рекомендуется), но, тем не менее, класс `TracePoint` позволяет вызвать написанный нами код при возникновении интересующего события. В примере ниже трассируются вызовы и возвраты из методов, а глубина вложенности обозначается отступами:

```
def factorial(n)
   (1..n).inject(:*) || 1
end

@call_depth = 0

TracePoint.trace(:a_call) do |tp|
   @call_depth += 1
   print "#{tp.path}:#{sprintf("%-4d", tp.lineno)} #{" " * @depth}"
   puts "#{tp.defined_class}##{tp.method_id}"
end

TracePoint.trace(:a_return) do |tp|
   print "#{tp.path}:#{sprintf("%-4d", tp.lineno)} #{" " * @depth}"
   puts "#{tp.defined_class}##{tp.method_id} => #{tp.return_value}"
   @call_depth -= 1
end

factorial(4)
```

Здесь мы воспользовались методом `TracePoint.trace`, это просто сокращенный способ вызвать `new`, а затем `enable`. Можно также трассировать одиночный блок кода, передав его методу `TracePoint#enable`. При выполнении этой программы будет напечатано следующее:

```
factorial.rb:12   #<Class:TracePoint>#trace
factorial.rb:12   #<Class:TracePoint>#trace =>
                    #<TracePoint:0x007ffe8a893f10>
factorial.rb:1    Object#factorial
factorial.rb:2      Enumerable#inject
factorial.rb:2        Range#each
factorial.rb:2          Fixnum#*
factorial.rb:2          Fixnum#* => 2
factorial.rb:2          Fixnum#*
factorial.rb:2          Fixnum#* => 6
factorial.rb:2          Fixnum#*
factorial.rb:2          Fixnum#* => 24
factorial.rb:2        Range#each => 1..4
```

```
factorial.rb:2      Enumerable#inject => 24
factorial.rb:3    Object#factorial => 24
```

С этим методом тесно связан метод Kernel#trace_var, который вызывает указанный блок при каждом присваивании значения глобальной переменной.

Если вы хотите трассировать все вообще, то вместо того чтобы писать свои трассировщики, можете затребовать стандартную библиотеку tracer. Ниже показана программа factorial.rb:

```
def factorial(n)
   (1..n).inject(:*) || 1
end

factorial(4)
```

Имея эту программу, мы можем просто загрузить tracer из командной строки:

```
$ ruby -disable-gems -r tracer factorial.rb
#0:factorial.rb:1::-: def factorial(n)
#0:factorial.rb:5::-: factorial(4)
#0:factorial.rb:1:Object:>: def factorial(n)
#0:factorial.rb:2:Object:-:    (1..n).inject(:*) || 1
#0:factorial.rb:2:Object:-:    (1..n).inject(:*)    1
#0:factorial.rb:3:Object:<: end
```

Библиотека tracer выводит номер потока, имя файла и номер строки, имя класса, тип события и исполняемую строку исходного текста трассируемой программы. Бывают следующие типы событий: '–' – исполняется строка исходного текста, '>' – вызов, '<' – возврат, 'C' – класс, 'E' – конец. (Если вы не отключили Rubygems или затребовали еще какую-то библиотеку, то будет напечатано много дополнительных строк.)

Описанные приемы можно использовать для построения весьма развитых средств отладки и инструментовки. В главе 16 мы покажем, как пользоваться уже имеющимися инструментами, например: pry, xotrace и ruby-prof. А пока читайте документацию, экспериментируйте и исследуйте – это самый лучший способ чему-то научиться.

11.5. Заключение

В этой главе мы привели примеры использования более сложных и даже экзотических механизмов ООП, а также решения некоторых рутинных задач. Мы видели, как реализуются некоторые паттерны проектирования. Познакомились мы и с API отражения в Ruby, продемонстрировали ряд интересных следствий динамической природы Ruby и некоторые трюки, возможные в динамическом языке.

Пришло время вернуться в реальный мир. Ведь ООП – не самоцель, а средство достижения цели. Цель же – написание эффективных, безошибочных и удобных для сопровождения приложений.

В современном окружении таким приложениям часто необходим графический интерфейс. В главе 12 мы рассмотрим создание графических интерфейсов на языке Ruby.

Глава 12. Графические интерфейсы для Ruby

Нет ничего хуже
четкого образа нечеткой идеи.

– Ансел Адамс

Нет смысла отрицать, что мы живем в век графических интерфейсов пользователя (ГИП). В обозримом будущем тот или иной вид графического интерфейса станет основным способом взаимодействия с компьютерами.

Я не думаю, что командная строка не переживет следующее десятилетие, у нее есть свое место в мире. Но даже закоренелые хакеры прежних лет (предпочитающие команду `cp -R` перетаскиванию файлов мышкой) все-таки не прочь воспользоваться ГИП, когда это оправдано.

Однако у графического программирования есть свои сложности. Главная проблема, конечно, состоит в том, чтобы определить, как должен выглядеть удобный интерфейс к программе; при проектировании интерфейсов картинка *не* всегда заменяет тысячу слов. В этой книге мы не можем уделить внимание этому аспекту, все-таки наша тема – не эргономика, не эстетика и не психология.

Вторая проблема в том, что графическое программирование сложнее. Надо учитывать форму, размер, положение и поведение всех находящихся на экране элементов управления, которыми можно манипулировать с помощью мыши, клавиатуры или пальца.

Третья трудность заключается в том, что на различных платформах представления о том, что такое оконная система и как она должна быть реализована, существенно разнятся. Чтобы в полной мере оценить это расхождение, нужно иметь опыт работы в разных системах. Многие программисты пытались создать кроссплатформенные инструменты и обнаружили, что именно с нестыковкой графических интерфейсов справиться труднее всего.

Эта глава не поможет вам в разрешении указанных проблем. Максимум, что я могу сделать, – это предложить очень краткое введение в несколько популярных графических систем (в той мере, в какой они относятся к Ruby), а также несколько советов и наблюдений.

Большая часть этой главы посвящена библиотекам Shoes, Tk, GTK+ и Qt. Велики шансы на то, что у вас возникнет вопрос: «А почему здесь нет (подставьте имя своей любимой библиотеки)?»

Причин несколько. Прежде всего, ограниченность места, все же эта книга посвящена не только графическим интерфейсам. Другая причина заключается в том, что к вашей системе может быть еще не написан интерфейс из Ruby (и в таком случае мы призываем вас этим заняться). И, наконец, не все графические интерфейсы «равны». Мы попытались рассказать о самых важных и зрелых, а остальные только упомянули.

12.1. Shoes 4

Shoes – специально заточенный под Ruby графический интерфейс – в настоящее время это самая распространенная система создания графических программ на Ruby. Shoes предлагает исключительно простой и прямолинейный графический API и используется как для обучения, так и для разработки приложений. Автор первоначальной версии, _why the lucky stiff_, написал ее для обучения своему инструменту Hackety Hack, а после того как _why исчез, сопровождение проекта взял на себя Стив Клэбник (Steve Klabnik).

Последняя версия Shoes работает под управлением JRuby – интерпретатора Ruby, написанного на Java. Он исполняет Ruby-программы в виртуальной машине Java (JVM), благодаря чему они получают доступ к написанным на Java библиотекам. Хотя так было не всегда, сегодня JRuby – стабильная, зрелая программа, которая иногда работает даже быстрее, чем MRI (main Ruby interpreter – основной интерпретатор Ruby).

В настоящее время команду добровольцев, работающих над Shoes 4, возглавляют Тобиас Пфайфер (Tobias Pfeiffer) и Джейсон Р. Кларк (Jason R. Clark). Они работают над претворением в жизнь изначальной цели _why – сделать так, чтобы детям было легко и просто писать интерактивные графические программы на Ruby.

12.1.1. Начало работы с Shoes

Установка JRuby – пожалуй, самая сложная часть работы. Нужно будет установить JRuby, следуя инструкциям для вашей платформы. Подробные инструкции приведены на сайте JRuby по адресу jruby.org. Другой вариант – воспользоваться менеджером версий Ruby, который обсуждается в разделе 21.6 главы 21 «Инструменты разработки для Ruby».

После того как JRuby установлен и запущен, для установки Shoes достаточно одной команды:

```
gem install shoes -v "~> 4.0.pre"
```

Требование к номеру версии `~> 4.0` говорит Rubygems, что нужно устанавливать версию Shoes не ниже 4.0, но меньше 5.0, а суффикс `.pre` разрешает установку бета-версий.

После установки gem-пакета программы для Shoes запускаются командой `shoes`. Для выполнения программы, находящейся в файле `program.rb`, нужно написать `shoes program.rb`.

Простейшая Shoes-программа просто создает пустое окно:

```
Shoes.app
```

Чтобы изменить размер окна, нужно передать методу app аргументы height: и width:. Для создания элементов в окне приложения методу app передает блок, в котором эти элементы объявляются. Всего есть десять основных элементов, типичных для графических программ: кнопки, текст, картинки, формы, ссылки и т. п.

12.1.2. Интерактивная кнопка

Кнопки в Shoes создаются совсем просто: нужно лишь вызвать метод button, передав ему текст кнопки в первом аргументе. Если передан блок, то он будет исполняться при нажатии кнопки. Следующее приложение отображает уведомление при нажатии кнопки:

```
Shoes.app(height: 30, width: 125) do
   button "Declaim poem" do
      alert "O freddled gruntbuggly thy micturations " \
         "are to me\n As plurdled gabbleblotchits on a " \
         "lurgid bee."
   end
end
```

Метод alert отображает переданную ему строку в модальном диалоговом окне, содержащем кнопку «OK». На рис. 12.1 показано окно этого приложения и уведомление, появляющееся после нажатия кнопки.

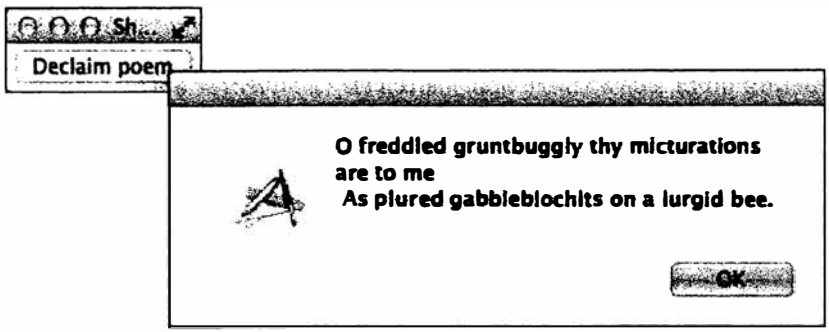


Рис. 12.1. Приложение Shoes и диалоговое окно уведомления

12.1.3. Текст и поле ввода

Для размещения текста в окне следует передать строку методу para. После строки можно задать дополнительные параметры, в частности size и stroke (изменяет цвет текста). Над этим методом есть надстройки, позволяющие быстрее записать некоторые комбинации параметров. Так, для задания текста покрупнее существуют методы banner, title, subtitle и другие, а для стилизации текста – методы em, strong, code и прочие.

Если сохранить ссылку на текстовый элемент, то впоследствии можно будет изменить его содержимое. Ниже мы создаем заголовок и кнопку, при нажатии которой его текст изменяется (см. рис. 12.2):

```
Shoes.app(height: 75, width: 225) do
   @book = title "The Urby Way"
   button "Correct spelling" do
      @book.text = "The Ruby Way"
   end
end
```

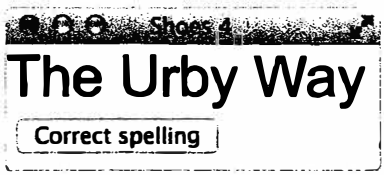


Рис. 12.2. Стилизованный текстовый заголовок в Shoes

Элемент `title` сохраняется в переменной `@book`, чтобы при нажатии кнопки текст заголовка можно было изменить. Отметим, что переменная экземпляра `@book` доступна в блоке `Shoes.app` и в блоке `button`. Оба блока вычисляются методом `instance_eval`, вызванным от имени работающего экземпляра класса `App`. В результате оба блока получают один и тот же набор переменных экземпляра вне зависимости от того, какому методу блок был передан.

Для ввода текста служат методы `edit_line` и `edit_box`, создающие соответственно однострочное и многострочное поле ввода. Если задан блок, то он будет вызван при изменении поля, а само поле передается ему в качестве аргумента. В примере ниже мы принимаем любую введенную в поле строку и отображаем ее задом наперед (см. рис. 12.3):

```
Shoes.app do
   edit_line do |line|
      @name.text = line.text.reverse
   end
   @name = para "", margin: 10
end
```

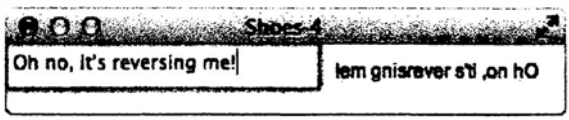


Рис. 12.3. Поле ввода и инвертированный текст

Элемент `para`, содержащий инвертированный текст, первоначально пуст (как и поле `edit_field`), но обновляется при вводе каждой буквы в поле.

12.1.4. Компоновка

Как вы, наверное, заметили (особенно если в предыдущем примере ввели длинный текст в поле edit_field), новые элементы заполняют имеющееся пространство слева направо и сверху вниз, как текст в абзаце. В Shoes такая компоновка называется *потоковой*, а поток определяет главный блок приложения Shoes.app. Другой тип компоновки называется *стопкой*. В стопке следующий элемент располагается под предыдущим.

Нетрудно догадаться, что стопки и потоки создаются путем передачи блока методам stack и flow соответственно. Комбинируя эти компоновки, легко создать окна, в которых элементы располагаются естественно, даже в несколько колонок. В следующем примере создается форма с двумя колонками, содержащая текст, поля ввода и кнопки:

```
Shoes.app width: 500, height: 240 do
   stack width: 220, margin: 10 do
      caption "Name"
      @name = edit_line

      caption "Nation"
      @nation = edit_line

      caption "Home"
      @home = edit_line

      caption "Destination"
      @destination = edit_line

      button "Compose", margin_top: 10 do
         @result.clear
         @result.append do
            caption "#{@name.text} is my name,"
            caption "and #{@nation.text} is my nation."
            caption "#{@home.text} is my dwelling place,"
            caption "#{@destination.text} my destination."
         end
      end
   end

   @result = stack width: -220, margin: 20
end
```

Ширина первой стопки фиксирована и равна 220 пикселей. Отрицательная ширина второй стопки означает, что она должна заполнять всю ширину окна, кроме 220 пикселей. Поскольку главный блок создает поток, то обе стопки размещаются бок о бок, образуя две колонки.

Вместо того чтобы писать для каждого поля edit_line блоки, которые динамически обновляют результат, мы создали кнопку, которая удаляет все элементы из правой колонки (если они там есть) и добавляет несколько текстовых элементов. Поскольку правая колонка также является стопкой, каждый элемент оказывается в отдельной строке.

В результате получается форма слева и текст справа, причем текст появляется после нажатия кнопки. Как выглядит окно программы, показано на рис. 12.4.

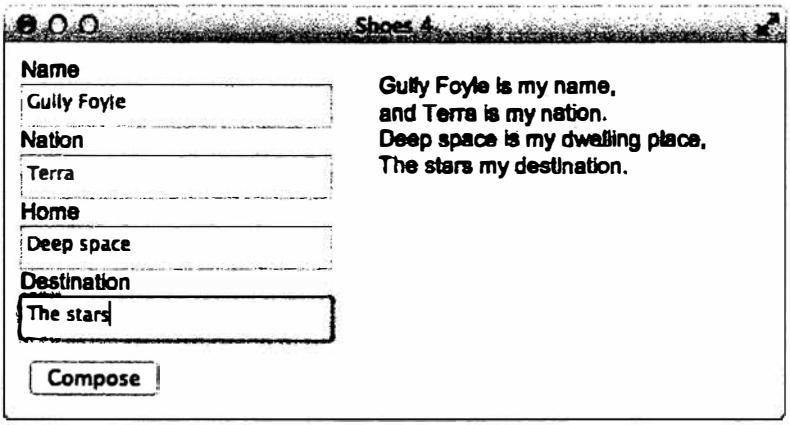


Рис. 12.4. Генератор четверостиший

12.1.5. Картинки и фигуры

Shoes умеет также отображать простые графические элементы. Для добавления изображения в окно Shoes нужно лишь вызвать метод image, указав путь к соответствующему файлу. Путь может вести на локальный файл или на изображение, размещенное в Интернете.

Для построения простых рисунков существуют методы arc, line, oval и rect. Можно строить и произвольные фигуры, воспользовавшись методом shape. Цвет обводки задается методом stroke, а цвет заливки — методом fill, причем вызывать их нужно до создания фигуры. В примере ниже мы рисуем зеленый овал, а затем черный прямоугольник, наклоненный под углом 30°. Результат показан на рис. 12.5.

```
Shoes.app width: 250, height: 125 do
   fill green
   nostroke
   oval top: 25, left: 125, width: 45, height: 65

   rotate 30
   rect top: 30, left: 30, width: 60, height: 40
end
```

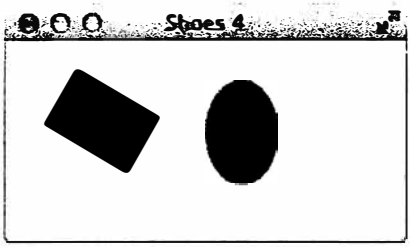


Рис. 12.5. Фигуры

Раскрашивание обводки и заливки отменяется методами nostroke и nofill соответственно, а ширина обводки задается методом strokewidth. Дополнительные сведения о рисовании фигур и операциях с ними можно найти в руководстве по Shoes на сайте shoesrb.com.

12.1.6. События

Мы уже видели, что нажатие кнопки и изменение текстового поля могут приводить к выполнению блоков. В Shoes есть еще несколько событий, которым можно сопоставить блоки:

- **keypress** – вызывается блок, которому передаются все нажимаемые клавиши;
- **motion** – вызывается при перемещении мыши, блоку передаются координаты мыши: left и top;
- **hover** – вызывается, когда мышь входит в стопку или поток;
- **click** – вызывается при щелчке мышью по элементу;
- **release** – вызывается при отпускании кнопки мыши после щелчка по элементу;
- **leave** – вызывается, когда мышь покидает стопку или поток.

Отметим, что событие keypress относится к приложению Shoes в целом, тогда как остальные – только к той стопке или потоку, с которой ассоциированы. С помощью событий мы можем перемещать фигуры в ответ на взаимодействие с программой:

```
Shoes.app do
   @circle = oval top: 5, left: 5, radius: 30

   motion do |left, top|
      if @track
         @circle.left, @circle.top = left - 15, top - 15
      else
         @circle.left = (width - left)
         @circle.top = (height - top)
      end
   end
```

```
    click { @track = true }
    release { @track = false }
end
```

В этой программе окружность двигается к противоположному краю окна, реагируя на перемещение мыши. Но если нажата кнопка мыши, то окружность точно следует за курсором. На снимке экрана показать это невозможно, поэтому попробуйте сами.

12.1.7. Прочие замечания

Программу, написанную для Shoes, можно упаковать в автономный исполняемый файл, работающий на платформах Windows, Mac OS X и Linux. Ведется работа по поддержке других графических сред, в том числе Qt и GTK, о которых речь пойдет ниже.

Хотя Shoes мало напоминает другие системы для построения ГИП, благодаря предельной простоте и «заточенности» под Ruby она стала самым популярным способом написания графических приложений на Ruby. Последние новости и дополнительную документацию, в том числе рукописное руководство и справочник по API, можно найти на сайте Shoes по адресу shoesrb.com.

12.2. Ruby/Tk

Своими корнями Tk уходит в далекий 1988 год, если считать и предварительные версии. Долгое время эта система считалась спутником языка Tcl, но вот уже несколько лет, как она используется и с другими языками, в том числе Perl и Ruby.

В 2001 году Tk являлась, наверное, самой распространенной графической библиотекой в мире Ruby. Она была первой и долгое время входила в стандартный дистрибутив Ruby. Сейчас былая популярность поугасла, но она все еще используется.

Хотя пакет Ruby/Tk включен в стандартную библиотеку Ruby, многие считают, что Tk пережила свое время; у тех, кто ценит чистые объектно-ориентированные интерфейсы, она может вызвать разочарование. Однако у нее есть и достоинства: широкая известность, переносимость и стабильность. Приведенные в этом разделе примеры работают в версиях Ruby для Windows, Mac OS X и Linux.

12.2.1. Обзор

Любое приложение Ruby/Tk должно загрузить расширение `tk`, выполнив директиву `load tk`. Далее интерфейс приложения строится поэтапно, начиная с того или иного контейнера, в который помещаются отдельные элементы управления. В конце выполняется вызов метода `Tk.mainloop`, в котором обрабатываются события: перемещения мыши, нажатия кнопок и т.д.

```
require "tk"
# Подготовка интерфейса приложения...
Tk.mainloop
```

Как и в большинстве оконных систем, графические элементы управления в Tk называются виджетами. Виджеты группируются с помощью контейнеров. Контейнер верхнего уровня называется *корневым*. Явно задавать корневой контейнер не всегда обязательно, но лучше это сделать.

Классы виджетов именуются так, как принято в мире Tk (в начале идет префикс `Tk`). Следовательно, виджету `Frame` соответствует класс `TkFrame`.

Понятно, что виджеты создаются методом `new`. Первый параметр определяет контейнер, в который помещается виджет; если он опущен, подразумевается корневой контейнер.

Дополнительные параметры виджета можно задавать двумя способами. Первый (в духе Perl) – передать хэш, содержащий названия и значения атрибутов. (Напомним, что в Ruby при передаче хэша последним или единственным параметром фигурные скобки можно опускать).

```
my_widget = TkSomewidget.new(borderwidth: 2,
  height: 40, justify: "center")
```

Другой способ – передать конструктору блок, который будет вычислен методом `instance_eval`. Внутри блока можно вызывать методы для установки атрибутов виджета (такие методы называются так же, как сами атрибуты). Имейте в виду, что блок вычисляется *в контексте виджета, а не вызывающего объекта*. Это означает, например, что к переменным экземпляра вызывающего объекта в блоке обращаться нельзя. В следующем примере создается виджет с точно такими же параметрами, как и выше:

```
my_widget = TkSomewidget.new do
              borderwidth 2
              height 40
              justify "center"
            end
```

Tk предоставляет три менеджера компоновки для управления относительным размером и расположением виджетов на экране. Первый (наиболее распространенный) – `pack`, остальные два – `grid` и `place`. Менеджер `grid` обладает богатыми возможностями, но не свободен от ошибок; менеджер `place` – самый простой из всех, он требует, чтобы были заданы абсолютные координаты всех расположенных внутри него виджетов. В примерах ниже мы будем пользоваться только менеджером `pack`.

12.2.2. Простое оконное приложение

Продемонстрируем очень простое приложение – окно, в котором выводится текущая дата. Начнем с явного создания корневого контейнера `root` и поместим в него виджет `Label`.

```
require "tk"

root = TkRoot.new() { title "Today's Date" }
str = Time.now.strftime("Today is \n%B %d, %Y")
lab = TkLabel.new(root) do
        text str
```

```
        pack("padx" => 15, "pady" => 10,
             "side" => "top")
    end
Tk.mainloop
```

Здесь мы создали корневой контейнер, сформировали строку даты и создали метку. В качестве текста, изображаемого на метке, мы задали строку str, а чтобы все выглядело аккуратно, вызвали метод pack, которому сказали, что отступ по горизонтали должен составлять 15 пикселей, а по вертикали – 10 пикселей. Текст мы попросили отцентрировать в границах метки.

На рис. 12.6 показано, как выглядит окно приложения.

Как было сказано выше, создать метку можно было бы и так:

```
lab = TkLabel.new(root) do
        text str
        pack("padx" => 15, "pady" => 10,
             "side" => "top")
      end
```

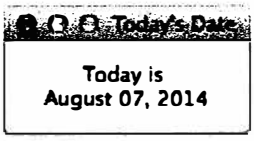


Рис. 12.6. Простое приложение Tk

Экранные единицы измерения (в примере выше мы их использовали для задания padx и pady) – по умолчанию пиксели. Можно применять и другие единицы, если добавить к числу суффикс. Тогда значение становится строковым, но поскольку Ruby/Tk на это наплевать, то и мы не станем беспокоиться. Допустимы следующие единицы измерения: сантиметры (c), миллиметры (m), дюймы (i) и пункты (p). Все показанные ниже способы задания padx правильны:

```
pack(padx: "80m")
pack(padx: "8c")
pack(padx: "3i")
pack(padx: "12p")
```

Атрибут side в данном случае не дает ничего, поскольку мы установили его значение по умолчанию. Если вы измените размер окна, то текст останется «приклеенным» к верхней части той области, которой принадлежит. Другие возможные значения side: right, left и bottom.

У метода pack есть и другие атрибуты, управляющие размещением виджетов на экране. Мы рассмотрим не все.

Атрибут fill говорит, должен ли виджет заполнять весь выделенный для него прямоугольник (по горизонтали и/или по вертикали). Допустимые значения x, y, both и none (по умолчанию none).

Атрибут anchor «скрепляет» виджет с теми или иными сторонами его прямоугольника, при этом применяется нотация «сторон света»; по умолчанию подразумевается значение center, другие допустимые значения: n, s, e, w, ne, nw, se, sw.

Атрибут in упаковывает виджет относительно контейнера, отличного от его родительского. По умолчанию, конечно, подразумевается родительский контейнер.

Атрибуты before и after позволяют произвольно задавать порядок упаковки виджетов. Это полезно, поскольку виджеты могут создаваться не в том порядке, в котором расположены на экране.

В общем, Tk обеспечивает достаточную гибкость при размещении виджетов в окне. Читайте документацию и экспериментируйте.

12.2.3. Кнопки

В любом графическом интерфейсе *кнопка* – один из наиболее употребительных виджетов. Как и следовало ожидать, в Ruby/Tk кнопка представляется классом TkButton.

В нетривиальных приложениях обычно создаются фреймы, содержащие разные виджеты. Кнопка может располагаться внутри такого фрейма.

Обычно для кнопки задаются по меньшей мере три атрибута:

- текст кнопки;
- ассоциированная с кнопкой команда (исполняемая в результате нажатия);
- способ упаковки кнопки в объемлющем контейнере.

Вот простенький пример:

```
btn_OK = TkButton.new do
  text "OK"
  command {proc { puts "Пользователь говорит OK." }}
  pack{"side" => "left"}
end
```

Здесь мы создаем новую кнопку и присваиваем объект переменной btn_OK. Конструктору передается блок, хотя при желании можно было воспользоваться и хэшем. В данном случае мы записали блок на нескольких строчках (нам так больше нравится), хотя на практике в одну строку можно «напихать» сколько угодно кода. Напомним, кстати, что блок вычисляется методом instance_eval, то есть в контексте объекта (в данном случае, вновь созданного объекта TkButton).

Текст, заданный в качестве значения атрибута text, рисуется на кнопке. Он может состоять из нескольких слов и даже строк.

Как работает метод pack, мы уже видели, ничего нового здесь нет. Стоит лишь отметить, что без вызова pack виджет не будет виден.

Интересная часть – метод command, который принимает блок и выполняет его при нажатии кнопки.

Выполняемое в этом примере действие не очень осмыслено. Когда пользователь нажимает кнопку, вызывается неграфический метод puts, который выводит строку в окно команд, из которого была запущена программа, или, быть может, в окно дополнительной консоли.

Следующий пример более содержателен. В листинге 12.1 приведено приложение, имитирующее термостат. В нем отображается то увеличивающаяся, то уменьшающаяся температура (создавая иллюзию, будто мы включаем обогрев или охлаждение). Код сопровождается комментариями.

Листинг 12.1. Имитация термостата

```
require 'tk'

# Типичные параметры компоновки...
top = { side: 'top', padx: 5, pady: 5 }
left = { side: 'left', padx: 5, pady: 5 }
bottom = { side: 'bottom', padx: 5, pady: 5 }

# Начальная температура...
temp = 74

root = TkRoot.new { title "Thermostat" }
tframe = TkFrame.new(root) { background "#606060" }
bframe = TkFrame.new(root)

tlab = TkLabel.new(tframe) do
   text temp.to_s
   font "{Helvetica} 54 {bold}"
   foreground "green"
   background "#606060"
   pack left
   end

# символ градуса
TkLabel.new(tframe) do
   text "o"
   font "{Helvetica} 14 {bold}"
   foreground "green"
   background "#606060"
   # Якорь "север" - над текстом, как обычно отображаются градусы
   pack left.update(anchor: 'n')
end

TkButton.new(bframe) do
   text " Up "
   pack left
   command do
      temp += 1
      tlab.configure(text: temp.to_s)
   end
end

TkButton.new(bframe) do
   text "Down"
```

```
    pack left
    command do
      temp -= 1
      tlab.configure(text: temp.to_s)
    end
end

tframe.pack top
bframe.pack bottom

Tk.mainloop
```

Здесь мы создали два фрейма. Верхний служит только для отображения температуры. Она измеряется по шкале Фаренгейта и для большего реализма выводится крупным шрифтом (а символ градуса отображается маленькой буквой «o», расположенной справа сверху). Нижний фрейм содержит кнопки «вверх» и «вниз».

Обратите внимание на еще не встречавшиеся атрибуты объекта `TkLabel`. Метод `font` задает гарнитуру и размер шрифта, которым выводится текст метки. Строковое значение платформенно-зависимо; то, что приведено в примере, предназначено для ОС Windows. В системах UNIX обычно указывается длинное и малопонятное имя шрифта, принятое в X Window, например: `-Adobe-Helvetica-Bold-R-Normal*-120-*-*-*-*-*-*`.

Метод `foreground` задает цвет текста. Здесь мы передаем строку `"green"` (которая в Tk имеет предопределенный смысл). Если вы хотите знать, предопределен ли тот иной цвет в Tk, то самое простое — попробовать.

Аналогично, метод `background` задает цвет фона, на котором выводится текст. В данном случае мы передаем строку в другом формате, а именно указываем красную, зеленую и синюю компоненты в шестнадцатеричном виде, как принято в языке HTML и других случаях. (Строка `"#606060"` соответствует приятному серому цвету.)

Мы не включили никакой кнопки «выхода» (чтобы не загромождать окно). Как обычно, для того чтобы закрыть приложение, достаточно щелкнуть по иконке Close в правом верхнем углу рамки окна.

Отметим использование метода `configure` в описании команд для кнопок; он изменяет текст метки по мере того, как текущая температура уменьшается или увеличивается. Мы уже говорили, что таким образом почти все атрибуты можно изменять во время выполнения, причем изменение отображается на экране незамедлительно.

Упомянем еще две операции над текстовыми кнопками. Метод `justify` принимает один параметр (`"left"`, `"right"` или `"center"`), определяющий выравнивание текста внутри кнопки (по умолчанию подразумевается `"center"`). Мы говорили, что можно отображать многострочный текст; метод `wraplength` задает номер колонки, в которой происходит перенос слова.

Стиль кнопки можно изменить методом `relief`, придав ей трехмерный вид. В качестве параметра этому методу можно передать одну из строк: `"flat"`, `"groove"`, `"raised"`, `"ridge"` (по умолчанию), `"sunken"` или `"solid"`. Методы `width` и `height` явно

задают размеры кнопки. Имеется также метод borderwidth и аналогичные. О других атрибутах (которых немало) вы можете прочесть в руководстве.

Рассмотрим еще один пример использования кнопки. На этой кнопке будет изображение, а не просто текст.

Я создал GIF-файлы с изображениями стрелок, указывающих вверх и вниз (up.gif и down.gif). Для получения ссылок на них можно воспользоваться классом TkPhotoImage, а затем передать эти ссылки в качестве параметров при создании кнопок.

```
TkButton.new(bframe) do
   image TkPhotoImage.new(file: "up.gif")
   pack left
   command do
      temp += 1
      tlab.configure(text: temp.to_s)
   end
end

TkButton.new(bframe) do
   image TkPhotoImage.new(file: "down.gif")
   pack left
   command do
      temp -= 1
      tlab.configure(text: temp.to_s)
   end
end
```

Здесь просто заменены некоторые строки в первом примере. Если не считать внешнего вида кнопок, то поведение не изменилось. На рис. 12.7 показано окно приложения.

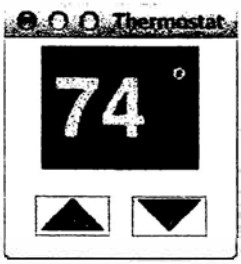


Рис. 12.7. Имитация термостата (с графическими кнопками)

12.2.4. Текстовые поля

Чтобы отобразить поле для ввода текста и манипулировать им, применяется виджет TkEntry. Как и следовало ожидать, для задания размера, цвета и поведения предусмотрены многочисленные атрибуты. Мы приведем довольно объемный пример, иллюстрирующий применение некоторых из них.

Поле ввода полезно лишь, если существует способ получить введенное в него значение. Обычно поле связывается с переменной (если быть точным, с объектом TkVariable), хотя можно воспользоваться и методом get.

Предположим, что мы разрабатываем telnet-клиент, который принимает четыре параметра: адрес хоста, номер порта (по умолчанию 23), имя пользователя и его пароль. Для красоты добавим еще две кнопки для операций «войти» и «отменить».

В представленном фрагменте используются фреймы, чтобы форма выглядела аккуратнее. Правда, написанный код не переносим, и настоящий знаток Tk с презрением отверг бы его. Но просто для сведения мы все-таки документировали этот «небрежный» подход к организации информации на экране.

Вид окна показан на рис. 12.8, а код – в листинге 12.2.

Листинг 12.2. Имитация telnet-клиента

```
require "tk"

def packing(padx, pady, side = "left", anchor = "n")
   { padx: padx, pady: pady, side: side, anchor: anchor }
end

root = TkRoot.new() { title "Telnet session" }
top  = TkFrame.new(root)
fr1  = TkFrame.new(top)
fr1a = TkFrame.new(fr1)
fr1b = TkFrame.new(fr1)
fr2  = TkFrame.new(top)
fr3  = TkFrame.new(top)
fr4  = TkFrame.new(top)

LabelPack  = packing(5, 5, "top", "w")
EntryPack  = packing(5, 2, "top")
ButtonPack = packing(15, 5, "left", "center")
FramePack  = packing(2, 2, "top")
Frame1Pack = packing(2, 2, "left")

var_host = TkVariable.new
var_port = TkVariable.new
var_user = TkVariable.new
var_pass = TkVariable.new

lab_host = TkLabel.new(fr1a) do
   text "Host name"
   pack LabelPack
end

ent_host = TkEntry.new(fr1a) do
   textvariable var_host
```

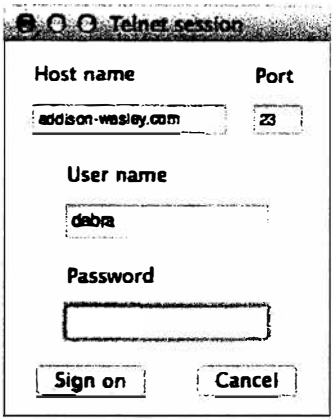


Рис. 12.8. Имитация telnet-клиента

```
   font "{Helvetica} 10"
   pack EntryPack
end

lab_port = TkLabel.new(fr1b) do
   text "Port"
   pack LabelPack
end

ent_port = TkEntry.new(fr1b) do
   width 4
   textvariable var_port
   font "{Helvetica} 10"
   pack EntryPack
end

lab_user = TkLabel.new(fr2) do
   text "User name"
   pack LabelPack
end

ent_user = TkEntry.new(fr2) do
   width 21
   font "{Helvetica} 12"
   textvariable var_user
   pack EntryPack
end

lab_pass = TkLabel.new(fr3) do
   text "Password"
   pack LabelPack
end

ent_pass = TkEntry.new(fr3) do
   width 21
   show "*"
   textvariable var_pass
   font "{Helvetica} 12"
   pack EntryPack
end

btn_signon = TkButton.new(fr4) do
   text "Sign on"
   command {}                    # Ничего не делает!
   pack ButtonPack
end

btn_cancel = TkButton.new(fr4) do
   text "Cancel"
```

```
    command { exit } # Just exits
    pack ButtonPack
end

top.pack FramePack
fr1.pack FramePack
fr2.pack FramePack
fr3.pack FramePack
fr4.pack FramePack
fr1a.pack Frame1Pack
fr1b.pack Frame1Pack

var_host.value = "addison-wesley.com"
var_user.value = "debra"
var_port.value = 23

ent_pass.focus
foo = ent_user.font

Tk.mainloop
```

Прежде всего разберемся с компоновкой виджетов. Сначала мы создали несколько фреймов, расположенных друг под другом. В самом верхнем фрейме есть два фрейма поменьше, расположенных по горизонтали.

В листинге 12.2 встречается также метод `packing`, единственная цель которого – сделать код чуточку чище. Он возвращает хэш, содержащий значения атрибутов `padx`, `pady`, `side` и `anchor`.

Объекты `TkVariable` предназначены для ассоциирования полей ввода с переменными. В классе `TkVariable` определен метод доступа `value`, который позволяет получать и устанавливать значение, хранящееся в объекте.

При создании объекта `TkEntry`, например `ent_host`, мы задаем атрибут `textvariable`, который связывает его с соответствующим объектом `TkVariable`. Иногда мы явно указываем ширину поля методом `width`; если это не сделано, то будет автоматически выбрана разумная ширина, обычно определяемая значением, которое в данный момент хранится в поле. Часто ширину подбирают методом проб и ошибок.

Шрифты задаются для полей ввода так же, как для меток. Аналогично обстоит дело и с цветами, которые в этом примере не задаются. Если шрифт пропорциональный, то два поля одинаковой ширины на экране могут оказаться различными.

Как обычно, необходимо вызвать метод `pack`. Мы немного упростили вызовы за счет использования констант.

Для поля, содержащего пароль, вызывается метод `show`, поскольку вводимое в него значение не должен видеть человек, заглядывающий через плечо. Вместо набираемых пользователем символов в таком поле будет отображаться символ, переданный методу `show` в качестве параметра (в данном случае звездочка).

Я уже сказал, что кнопки тут нужны только для красоты. Кнопка **Sign on** вообще ничего не делает, а кнопка **Cancel** завершает программу.

У полей ввода есть и другие атрибуты. Можно изменять значение из программы, не заставляя пользователя вводить его; можно задать шрифт и цвета; можно изменить характеристики курсора, указывающего место вставки, и перемещать его в нужное место. И многое, многое другое. Полное описание вы найдете в руководстве.

Раз уж мы заговорили о вводе текста, будет уместно упомянуть виджет `Text`. По сравнению с полем ввода это примерно то же, что двухместный планер по сравнению космическим челноком. Виджет `Text` спроектирован специально для ввода больших фрагментов текста, насчитывающих много строк, и по существу является основой для создания полноценного редактора. Он довольно сложен, поэтому мы не будем его здесь рассматривать, но обязательно изучите его, если планируете предлагать редактирование текста с помощью Tk.

12.2.5. Прочие виджеты

Tk содержит еще много виджетов. Упомянем некоторые из них.

Флажок обычно используется для представления полей, которые могут принимать одно из двух значений: да/нет или вкл/выкл. В Tk он называется «check button», а соответствующий ему класс – `TkCheckButton`.

Пример в листинге 12.3 – это лишь скелет, в нем даже ни одной кнопки нет. Выводятся три флажка, соответствующие курсам, на которые можно записаться (информатика, музыка и литература). На консоль выводится сообщение при каждом изменении состояния флажка.

Листинг 12.3. Флажки в Tk

```
require 'tk'

root = TkRoot.new { title "Checkbutton demo" }
top = TkFrame.new(root)

pack_opts = { side: "top", anchor: "w" }

cb1var = TkVariable.new
cb2var = TkVariable.new
cb3var = TkVariable.new

cb1 = TkCheckButton.new(top) do
  variable cb1var
  text "Информатика"
  command { puts "Button 1 = #{cb1var.value}" }
  pack pack_opts
end

cb2 = TkCheckButton.new(top) do
  variable cb2var
  text "Музыка"
```

```
  command { puts "Button 2 = #{cb2var.value}" }
  pack pack_opts
end

cb3 = TkCheckButton.new(top) do
  variable cb3var
  text "Литература"
  command { puts "Button 3 = #{cb3var.value}" }
  pack pack_opts
end

top.pack pack_opts

Tk.mainloop
```

Отметим, что переменная, ассоциированная с флажком, принимает значение 1, когда флажок отмечен, и 0 -- когда он сброшен. Эти значения можно изменить с помощью методов onvalue и offvalue. Кроме того, еще до создания флажка можно установить значение переменной и тем самым задать начальное состояние флажка.

Если по какой-то причине мы хотим закрасить флажок серым, то можем с помощью метода state установить состояние disabled. Остальные состояния -- active (отмечен) и normal (сброшен), причем последнее подразумевается по умолчанию.

Изменим пример в листинге 12.3. Пусть речь идет не о дополнительных, а о профилирующих университетских курсах. Если не считать сдвоенных курсов, то в каждый момент времени можно выбрать не более одного предмета. В таком случае нам понадобятся *переключатели* (их реализует класс TkRadioButton).

Пример в листинге 12.4 мало чем отличается от листинга 12.3. Конечно, имя класса теперь другое. Еще одно важное отличие состоит в том, что всем переключателям соответствует одна и та же переменная. Именно поэтому Tk знает, что переключатели принадлежат одной группе. В форме может быть и несколько групп переключателей, но для каждой группы выделяется по одной переменной.

Листинг 12.4. Переключатели в Tk

```
require 'tk'

root = TkRoot.new() { title "Radiobutton demo" }
top = TkFrame.new(root)

pack_opts = { side: "top", anchor: "w" }

major = TkVariable.new

b1 = TkRadioButton.new(top) do
  variable major
  text "Информатика"
  value 1
```

```
  command { puts "Major = #{major.value}" }
  pack pack_opts
end

b2 = TkRadioButton.new(top) do
  variable major
  text "Музыка"
  value 2
  command { puts "Major = #{major.value}" }
  pack pack_opts
end

b3 = TkRadioButton.new(top) do
  variable major
  text "Литература"
  value 3
  command { puts "Major = #{major.value}" }
  pack pack_opts
end

top.pack pack_opts

Tk.mainloop
```

Здесь метод value ассоциирует с каждым переключателем конкретное значение. Значения могут быть произвольны (например, строки). Мы не использовали строки, так как хотели подчеркнуть, что не существует прямой связи между текстом виджета и возвращаемым им значением.

Для настройки внешнего вида и поведения флажков и переключателей есть множество атрибутов. К примеру, метод image позволяет выводить не текстовую строку, а изображение. Применимы также обычные атрибуты форматирования и отображения виджетов; подробности см. в руководстве.

Если бы данная книга (или хотя бы эта глава) была целиком посвящена Tk, мы рассказали бы много чего еще. Но невозможно уделить внимание всем деталям, мы упомянем их лишь для того, чтобы вы знали, что имеется в вашем распоряжении.

Виджет ListBox (TkListBox) позволяет вывести список, из которого пользователь выбирает элементы. Режим выбора (метод selectmode) принимает следующие значения: single, extended, browse. Первые два режима определяют, можно ли выбрать только один или сразу несколько элементов. Режим browse аналогичен режиму single с тем отличием, что выбранный элемент можно перемещать в списке мышью. Список можно прокручивать, так что число элементов в нем не ограничено.

Tk располагает развитыми средствами для работы с меню: выпадающие меню, уединенные (tear-off) меню, каскадные подменю, клавиши быстрого выбора, переключатели в меню и многое другое. Ознакомьтесь с классами TkMenu, TkMenubar и TkMenuButton.

Пожалуй, самый «творческий» виджет – это TkCanvas, который позволяет программисту манипулировать изображением на уровне пикселей. У него есть методы для рисования линий и фигур, манипулирования цветами и загрузки изображений в различных форматах. Если вашему приложению необходима развитая графика или рисование под контролем пользователя, то этот виджет вас заинтересует.

Полоса прокрутки позволяет реализовать нестандартную прокрутку по горизонтали и по вертикали (например, синхронную прокрутку двух окон). Виджет Scale представляет собой бегунок для представления числовых значений; его можно ориентировать по горизонтали или по вертикали и использовать как для ввода, так и для вывода. О других виджетах можете прочесть в документации.

12.2.6. Дополнительные замечания

Перспективы Tk туманны (то же можно сказать и о большинстве программных систем), но в ближайшем будущем она никуда не денется. Tk (а равно и язык Tcl) по-прежнему активно разрабатываются. На момент написания этой книги пакет Ruby/Tk был основан на последней версии Tk 8.5.

Нужно еще сказать несколько слов об операционных системах. Теоретически Tk – платформенно-независимая система, и практика не далека от теории. Однако есть сообщения о том, что версия для Windows не так стабильна, как для UNIX. На всякий случай отмечу, что все приведенные в этой главе примеры были протестированы в Windows и работают, как задумано.

12.3. Ruby/GTK3

Библиотека GTK+ – побочный продукт развития графического редактора GIMP (GNU Image Manipulation Program), аббревиатура расшифровывается как GIMP Toolkit. Как UNIX и BSD, GTK+ разработана в Калифорнийском университете в Беркли.

Если вы знакомы с системой X/Motif, скажем, что GTK+ внешне похожа на нее, но не так громоздка. Библиотека GTK+ зародилась в мире UNIX и лежит в основе графического менеджера GNOME (является графическим интерфейсом для Linux), но при этом является более или менее кросс-платформенной.

12.3.1. Обзор

Ruby/GTK3 – библиотека, позволяющая приложениям, написанным на языке Ruby, обращаться к средствам библиотеки GTK+. Она создавалась в рамках проекта Ruby GNOME 2, но содержит также интерфейс к GTK2 и многим другим компонентам GTK+. GTK+ распространяется в исходных текстах на условиях лицензии GNU LGPL, поэтому может бесплатно использоваться в коммерческих приложениях.

Как и в большинстве библиотек для построения графических интерфейсов, в GTK+ есть концепции фреймов, окон, диалогов и менеджеров размещения. Она располагает богатым набором виджетов, включающим большинство стандартных,

например, метки, кнопки и текстовые поля, а также ряд более сложных: деревья и многоколонные списки.

Хотя GTK+ написана на C, спроектирована она в объектно-ориентированной манере. Поэтому Ruby/GTK3 предоставляет объектно-ориентированный API, не слишком отдаляясь от исходной реализации на C. Кроме того, Ruby/GTK3 написана вручную, а не с помощью таких генераторов кода, как SWIG. Поэтому API выдержан в духе Ruby, с использованием блоков, необязательных аргументов и т.д. Справочное руководство можно найти на сайте http://ruby-gnome2.sourceforge.jp/.

Библиотека GTK+ создана на базе других библиотек: GLib, Pango, ATK, Cairo и GDK. Она поддерживает неграфические функции (GLib), отображение интернационализированных текстов в кодировке UTF-8 (Pango), специальные возможности (Atk), графику (Cairo), низкоуровневые графические объекты (Gdk), а также множество виджетов и высокоуровневых графических объектов (Gtk).

Gem-пакет `gtk3` работает для Ruby в Linux, Windows и Mac OS X, если установлена библиотека GTK+ 3. В Mac OS X он работает только в среде XQuartz – результате переноса X Window System на OS X. Сейчас ведется работа по переносу на «родной» графический интерфейс Mac OS X, но пока эта версия далека от стабильности.

GTK+ – объектно-ориентированная библиотека, поддерживающая логически стройную иерархию виджетов. Классы `Gtk::Bin` и `Gtk::Container` весьма развиты, а комбинация менеджеров размещения `Gtk::Box` и `Gtk::Table` обеспечивает простые, но в то же время гибкие средства управления геометрией. В Ruby/GTK3 есть удобный механизм задания обработчиков сигналов.

Среди виджетов GTK+ вы найдете меню, панели инструментов, всплывающие подсказки, деревья, индикаторы хода процесса, бегунки и календари. К слабым местам текущей версии GTK+ можно отнести недостаточно богатый набор стандартных диалогов, с которыми к тому же трудно работать в модальном режиме. Есть недостатки и у стандартного многострочного текстового редактора.

Все строки, передаваемые методам Ruby/GTK3, должны быть представлены в кодировке UTF-8. Поэтому все ваши программы должны быть настроены так, чтобы преобразовывать текст в UTF-8. Дополнительные сведения о кодировках и преобразовании см. в главе 4 «Интернационализация в Ruby».

12.3.2. Простое оконное приложение

Любая программа, в которой используется Ruby/GTK3, должна содержать директиву `require "gtk3"`. Функциональность Ruby/GTK3 предоставляется посредством модулей `Gtk` и `Gdk`, поэтому имена классов GTK+ обычно начинаются с префикса `Gtk::` или `Gdk::`.

Как правило, для инициализации Ruby/GTK3 мы вызываем метод `Gtk.init`, а затем создаем окно верхнего уровня и обработчик сигнала `destroy` (который поступает, когда пользователь закрывает окно). Метод `show_all` делает окно видимым, а обращение к `Gtk.main` запускает цикл обработки событий.

Мы еще вернемся к этой теме, но сначала рассмотрим пример. Следующий код, как и рассмотренная выше программа для Tk, отображает текущую дату:

```
require "gtk3"
Gtk.init

window = Gtk::Window.new("Today's Date")
window.signal_connect("destroy") { Gtk.main_quit }
str = Time.now.strftime("Today is \n%B %d, %Y")
window.add(Gtk::Label.new(str))
window.set_default_size(200, 100)
window.show_all
Gtk.main
```

Метод `Gtk.init` инициализирует Ruby/GTK2. Главное окно (типа `Gtk::Window`) создается как окно «верхнего уровня», а указанный текст отображается в полосе заголовка. Окна верхнего уровня имеют стандартную полосу заголовка и вообще ведут себя так, как принято ожидать от главного окна приложения.

Далее создается обработчик сигнала `destroy`, который посылается при закрытии главного окна. Этот обработчик (в данном случае, один блок) просто завершает главный цикл обработки событий. В документации по Ruby/GTK3 перечислены все сигналы, которые могут поступать каждому виджету (не забудьте и о суперклассах). Обычно они генерируются в результате манипуляций с мышью и клавиатурой, срабатывания таймеров, изменений состояния окна и т.д.

В следующей строке мы добавляем метку прямо в главное окно. Размер метки вычисляется автоматически на основе длины текста.

По умолчанию размеры родительских виджетов в GTK+ устанавливаются автоматически, исходя из размеров потомков. В данном случае длина строки, отображаемой выбранным по умолчанию шрифтом, определяет размер метки, а размер главного окна выбирается так, чтобы в нем поместилась метка. Такое окно получилось бы слишком маленьким, поэтому с помощью метода `set_default_size` мы говорим, что начальный размер главного окна должен составлять 200×100 пикселей.

Затем мы вызываем метод `show_all`, чтобы сделать главное окно и всех его потомков видимыми. По умолчанию главное окно скрыто, поэтому в большинстве приложений такой вызов необходим.

Метод `Gtk.main` запускает цикл обработки событий в GTK+. Он не возвращает управления, пока приложение не завершится. В данном случае обработчик события `destroy` приводит к выходу из `Gtk.main`, после чего завершается и все приложение.

12.3.3. Кнопки

Для создания кнопки в Ruby/GTK3 предназначен класс `Gtk::Button`. В простейшем случае мы задаем обработчик события `clicked`, которое возникает, когда пользователь щелкает по кнопке.

Программа в листинге 12.5 позволяет ввести одну строку в текстовое поле и после нажатия кнопки **All Caps!** преобразует ее в верхний регистр. На рис. 12.9 показано текстовое поле до нажатия кнопки.

Листинг 12.5. Кнопки в GTK

```
require "gtk3"

class SampleWindow < Gtk::Window

  def initialize
    super("Ruby/GTK3 Sample")
    signal_connect("destroy") { Gtk.main_quit }

    entry = Gtk::Entry.new

    button = Gtk::Button.new("All Caps!")
    button.signal_connect("clicked") {
      entry.text = entry.text.upcase
    }

    box = Gtk::HBox.new
    box.add(Gtk::Label.new("Text:"))
    box.add(entry)
    box.add(button)

    add(box)
    show_all
  end
end

Gtk.init
SampleWindow.new
Gtk.main
```

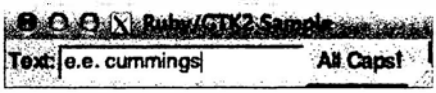


Рис. 12.9. Пример простой кнопки в GTK

В листинге 12.5 определен класс `SampleWindow`, при таком подходе класс может управлять собственным отображением и поведением (не заставляя вызывающую программу конфигурировать окно). Класс главного окна наследует `Gtk::Window`.

Как и в примере «Текущая дата», обработчик сигнала `destroy` завершает цикл обработки событий после закрытия главного окна.

Этот класс создает однострочное поле ввода (класс `Gtk::Entry`) и кнопку `Gtk::Button` с текстом `All Caps!`. С кнопкой связан обработчик события `clicked`, которое генерируется, когда пользователь *нажимает и отпускает* кнопку мыши, находясь над кнопкой.

Класс `Gtk::Window` является производным от `Gtk::Bin`, поэтому может содержать только один дочерний виджет. Чтобы поместить в окно два виджета, мы сначала

помещаем их в контейнер HBox, а уже этот контейнер делаем потомком главного окна. Виджеты, добавляемые в контейнер Gtk::HBox, по умолчанию размещаются начиная с его правой границы. Есть также контейнер Gtk::VBox, который упаковывает своих потомков по вертикали.

Как и раньше, чтобы главное окно (и все его потомки) стало видимым, необходимо вызвать метод show_all.

Обработчик события clicked вызывается при нажатии кнопки. Он получает текст, находящийся в поле ввода, преобразует его в верхний регистр и записывает обратно в поле ввода.

Собственно код приложения находится после определения класса SampleWindow. В нем всего лишь создается главное окно и запускается цикл обработки событий.

12.3.4. Текстовые поля

В библиотеке GTK+ есть класс Gtk::Entry для ввода одной строки текста, мы видели его в предыдущем примере. Существует также класс Gtk::TextView, представляющий собой мощный многострочный редактор, его мы и опишем.

Программа в листинге 12.6 создает многострочное текстовое поле и помещает в него текст. По мере изменения содержимого поля текущая длина текста отображается с помощью метки, расположенной в нижней части окна (рис. 12.10).

Листинг 12.6. Текстовый редактор в GTK

```
require "gtk3"

class TextWindow < Gtk::Window

   def initialize
      super("Text Sample")
      signal_connect("destroy") { Gtk.main_quit }
      set_default_size(200, 50)

      @text = Gtk::TextView.new
      @text.wrap_mode = :word

      @buffer = @text.buffer
      @buffer.signal_connect("changed") {
        @status.text = "Length: " + @buffer.char_count.to_s
      }
      @buffer.create_tag('notice',
                         'font' => "Times Bold Italic 18",
                         'foreground' => "red")

      @status = Gtk::Label.new

      scroller = Gtk::ScrolledWindow.new
      scroller.set_policy(:automatic, :never)
```

```
        scroller.add(@text)

        box = Gtk::Box.new(:vertical)
        box.add(scroller)
        box.add(@status)
        add(box)

        iter = @buffer.start_iter
        @buffer.insert(iter, "This is an editor")
        iter.offset = 5
        @buffer.insert(iter, "really ", "notice")

        show_all
    end
end

Gtk.init
TextWindow.new
Gtk.main
```

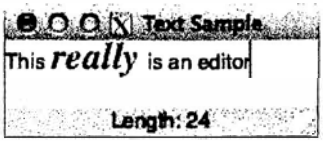


Рис. 12.10. Небольшой текстовый редактор в GTK

Структура программы такая же, как в примере с кнопкой: инициализировать Ruby/GTK3, определить класс главного окна, задать обработчик события, корректно завершающий приложение, и установить начальный размер окна. После `initialize` вызывается метод `show_all`, который делает окно видимым. В последних двух строчках создается окно и запускается цикл обработки событий.

Мы создали виджет редактора с именем `@text`. Включен режим переноса строк, по умолчанию строки разрываются без учета границы слов.

Переменная `@buffer` – это текстовый буфер для виджета `@text`. Мы установили обработчик события `changed`; он будет вызываться при вставке, удалении и изменении текста. Обработчик пользуется методом `char_count`, чтобы узнать текущую длину текста в редакторе и преобразовать ее в строку сообщения. Предложение `@status.text = text` отображает это сообщение в окне.

Далее мы конфигурируем виджет `@text` так, чтобы он показывал текст другим стилем. Для этого с помощью метода `create_tag` создается тег «notice», с которым связан шрифт «Times Bold Italic 18» и красный цвет. Класс `Gtk::TextTag` позволяет задавать и другие свойства тегов.

В данном случае мы хотим воспользовать шрифтом из семейства Times, на платформе Windows мы, скорее всего, получим какой-то вариант шрифта Times Roman. В ОС Linux/UNIX параметром должна быть стандартная для X Window

System строка задания шрифта. Система вернет шрифт, наиболее близкий к затребованному.

Метка `@status` первоначально пуста. Ее текст будет изменен позже.

GTK+ предлагает два способа добавить полосы прокрутки. Можно напрямую создать объект `Gtk::ScrollBar` и с помощью сигналов синхронизировать его с ассоциированным виджетом. Но в большинстве случаев проще воспользоваться виджетом `Gtk::ScrolledWindow`.

Виджет `Gtk::ScrolledWindow` наследует `Gtk::Bin`, поэтому может содержать только один дочерний виджет. Но этот виджет может принадлежать классу `Gtk::Box` или любому другому контейнеру, допускающему несколько потомков. Ряд виджетов GTK+, в том числе и `Gtk::TextView`, автоматически взаимодействуют с `Gtk::ScrolledWindow`, не требуя почти никакого дополнительного кода.

В данном примере мы создали виджет `Gtk::ScrolledWindow` с именем `scroller` и сконфигурировали его методом `set_policy`. Мы решили не отображать горизонтальную полосу прокрутки вовсе, а вертикальную – только тогда, когда в редакторе больше строк, чем видно в окне. Сам текстовый редактор сделан непосредственным потомком `scroller`.

Теперь надо настроить контейнер `Gtk::Vbox`, который расположит наши виджеты по вертикали. Сначала добавляется прокручиваемое окно, содержащее поле ввода, поэтому оно окажется самым верхним. Метка `@status` располагается под ним. Напоследок сам контейнер добавляется в главное окно.

В следующих четырех строчках в поле ввода добавляется текст. В первой строчке мы получаем объект `Gtk::TextIter`, соответствующий началу текста (offset = 0), и вставляем в это место строку. Поскольку до этого момента никакого текста в поле еще не было, то только сюда и можно его вставить. Затем вставляется другой кусок текста со смещением 5. В результате редактор будет содержать строку `This really is an editor`.

Поскольку мы предварительно установили обработчик события `changed`, он будет вызываться после каждого обращения к `insert`. Следовательно статус будет отображаться правильно, несмотря на то, что пользователь еще не вносил никаких изменений в текст.

12.3.5. Прочие виджеты

Даже для организации сравнительно простого графического интерфейса текстовых полей и кнопок может оказаться недостаточно. Нужны переключатели, флажки и другие виджеты. В следующем примере демонстрируются некоторые из них.

В листинге 12.7 предполагается, что пользователь хочет заказать билет на самолет. Для выбора города назначения используются классы `Gtk::treeView`, `Gtk::ListStore` и `Gtk::treeViewColumn` (многоколонный список). Флажок (класс `Gtk::CheckButton`) определяет, нужен ли обратный билет, а переключатель (класс `Gtk::RadioButton`) позволяет указать класс салона. Завершает интерфейс кнопка `Purchase` (Заказать) (рис. 12.11).

Листинг 12.7. Заказа билета на самолет

```
require "gtk3"

class TicketWindow < Gtk::Window

  def initialize
    super("Purchase Ticket")
    signal_connect("destroy") { Gtk.main_quit }

    dest_model = Gtk::ListStore.new(String, String)
    dest_view = Gtk::TreeView.new(dest_model)
    dest_column = Gtk::TreeViewColumn.new("Destination",
                    Gtk::CellRendererText.new,
                    :text => 0)
    dest_view.append_column(dest_column)
    country_column = Gtk::TreeViewColumn.new("Country",
                       Gtk::CellRendererText.new,
                       :text => 1)
    dest_view.append_column(country_column)
    dest_view.selection.set_mode(Gtk::SELECTION_SINGLE)

    [["Cairo", "Egypt"], ["New York", "USA"],
     ["Tokyo", "Japan"]].each do |destination, country|
      iter = dest_model.append
      iter[0] = destination
      iter[1] = country
    end
    dest_view.selection.signal_connect("changed") do
      @city = dest_view.selection.selected[0]
    end

    @round_trip = Gtk::CheckButton.new("Round Trip")

    purchase = Gtk::Button.new("Purchase")
    purchase.signal_connect("clicked") { cmd_purchase }

    @result = Gtk::Label.new

    @coach = Gtk::RadioButton.new("Coach class")
    @business = Gtk::RadioButton.new(@coach, "Business class")
    @first = Gtk::RadioButton.new(@coach, "First class")

    flight_box = Gtk::VBox.new
    flight_box.add(dest_view).add(@round_trip)

    seat_box = Gtk::VBox.new
    seat_box.add(@coach).add(@business).add(@first)

    top_box = Gtk::HBox.new
```

```
    top_box.add(flight_box).add(seat_box)

    main_box = Gtk::VBox.new
    main_box.add(top_box).add(purchase).add(@result)

    add(main_box)
    show_all
  end

  def cmd_purchase
    text = @city
    if @first.active?
      text += ": first class"
    elsif @business.active?
      text += ": business class"
    elsif @coach.active?
      text += ": coach"
    end
    text += ", round trip " if @round_trip.active?
    @result.text = text
  end

end

Gtk.init
TicketWindow.new
Gtk.main
```

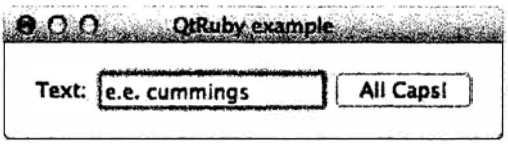


Рис. 12.11. Различные виджеты GTK

В этом приложении, как и в предыдущих примерах, создает главное окно с обработчиком события. Затем создается список с двумя колонками, дизайн которого следует паттерну Модель-Вид-Контроллер (Model-View-Controller – MVC); класс `Gtk::ListStore` (модель) имеет две колонки типа `String`.

Далее создается виджет `Gtk::TReeView`. Класс `Gtk::treeViewColumn` конфигурирует эту колонку. Первая колонка называется «Destination», а для отображения клеток применяется класс рисовальщика `Gtk::CellRendererText`. Первая колонка модели (с номером 0) (`Gtk::ListStore`) служит значением текстового свойства. Итак, рисовальщики клеток наполняют древесную модель данными. В GTK+ 2.x есть несколько готовых рисовальщиков клеток, в том числе: `Gtk::CellRendererText`, `Gtk::CellRendererPixbuf` и `Gtk::CellRendererToggle`. Далее в список добавляются три

строки данных и устанавливается обработчик события "changed", который будет вызываться, когда пользователь выберет другую строку. Этот обработчик изменит значение переменной @city, записав в нее текст из первой колонки (с номером 0) только что выбранной строки.

Затем создается простой флажок (Gtk::CheckButton) и кнопка (Gtk::Button). Обработчик события нажатия кнопки вызовет метод cmd_purchase. Метка @result первоначально пуста, но позже в нее будет записана строка, определяющая вид заказанного билета.

Три переключателя создаются как члены одной группы, то есть в любой момент может быть выбран лишь один из них. Когда пользователь щелкает по любому переключателю, ранее выбранный сбрасывается. Первым параметром конструктору переключателя передается первый переключатель из той же группы. Поэтому у конструктора первого переключателя в группе этого параметра нет, а остальным передается ссылка на первый переключатель.

Виджеты нужно организовать на экране так, чтобы пользователю было удобно. Мы воспользовались комбинацией контейнеров Gtk::HBox и Gtk::VBox. Список расположен над флажком. Все три переключателя расположены вертикально справа от списка. А кнопка помещена под всеми остальными виджетами.

Метод cmd_purchase очень прост: он строит строку, отражающую состояние всех виджетов в момент нажатия кнопки. У переключателей и флажков есть метод active?, который возвращает true, если виджет отмечен. Построенная строка записывается в метку @result и потому появляется на экране.

Во многих приложениях интерфейс содержит меню. В следующем примере показано, как можно организовать меню в Ruby/GTK3. Заодно демонстрируется применение всплывающих подсказок – мелкая деталь, способная украсить любую программу.

В листинге 12.8 создается главное окно с меню, содержащим пункт File и еще два фиктивных пункта. В меню File есть команда Exit, которая завершает приложение. Оба пункта File и Exit снабжены всплывающими подсказками.

Листинг 12.8. Пример меню в GTK

```
require "gtk3"

class MenuWindow < Gtk::Window

  def initialize
    super("Ruby/GTK3 Menu Sample")
    signal_connect("destroy") { Gtk.main_quit }

    file_exit_item = Gtk::MenuItem.new("_Exit")
    file_exit_item.signal_connect("activate") { Gtk.main_quit }

    file_menu = Gtk::Menu.new
    file_menu.add(file_exit_item)

    file_menu_item = Gtk::MenuItem.new("_File")
```

```
    file_menu_item.submenu = file_menu

    menubar = Gtk::MenuBar.new
    menubar.append(file_menu_item)
    menubar.append(Gtk::MenuItem.new("_Nothing"))
    menubar.append(Gtk::MenuItem.new("_Useless"))

    tooltips = Gtk::Tooltips.new
    tooltips.set_tip(file_exit_item, "Exit the app", "")

    box = Gtk::Box.new(:vertical)
    box.pack_start(menubar, expand: false, fill: false, padding: 0)
    box.add(Gtk::Label.new("Try the menu and tooltips!"))

    add(box)
    set_default_size(400, 100)
    show_all
  end
end

Gtk.init
MenuWindow.new
Gtk.main
```

И здесь базовая структура программы такая же, как в предыдущих примерах. В данном случае мы создаем пункт меню `Gtk::MenuItem` с именем `Exit` и задаем для него обработчик события, который завершает программу. Событие называется `activate` и генерируется, когда пользователь выбирает пункт меню.

Далее создается меню `File` и в него добавляется пункт `Exit`. Это все, что требуется для создания выпадающего меню. В конце создается пункт меню `File`, именно он и появится в полосе меню. Чтобы присоединить пункт `File` к меню `File`, мы вызываем метод `submenu=`.

Затем создается полоса меню `Gtk::MenuBar`, в которую добавляются три меню: `File`, `Nothing` и `Useless`. Что-то делает лишь первое меню, остальные приведены только для демонстрации.

Всплывающие подсказки интегрированы со всеми виджетами, в том числе с пунктами меню. Чтобы создать подсказку, нужно вызвать метод `set_tip`, которому передается текст подсказки.

Чтобы разместить полосу меню в верхней части главного окна, мы взяли `Gtk::Box` в качестве самого внешнего контейнера. В данном случае мы добавляем в него полосу меню не методом `add`, а методом `pack_start`, чтобы точнее контролировать внешний вид и положение виджета.

Первым параметром методу `pack_start` передается размещаемый виджет. Второй параметр – булев признак, показывающий, должен ли виджет занимать все доступное пространство. Отметим, что виджет при этом не растет, а обычно просто размещается в центре контейнера. Мы хотим, чтобы полоса меню располагалась сверху, поэтому передаем `false`.

Третий параметр – тоже булева величина, говорящая о том, должны ли размеры виджет быть изменены так, чтобы он занял все доступное пространство. Нам нужна лишь узкая полоса меню, поэтому мы и тут передаем `false`. Последний параметр метода `pack_start` задает отступы, то есть пустое место вокруг виджета. Нам это ни к чему, поэтому мы передаем нуль.

Большую часть главного окна занимает метка. Напоследок мы принудительно устанавливаем размер окна 400 x 100 пикселей.

12.3.6. Дополнительные замечания

Ruby/GTK3 – часть проекта Ruby-GNOME3. GNOME – пакет более высокого уровня, основанный на библиотеке GTK+, а Ruby-GNOME3 – набор привязок для библиотек, входящих в состав GNOME.

Gem-пакет `gtk3` включает следующие библиотеки:

- **Ruby/GLib.** GLib – низкоуровневая инфраструктурная библиотека. Она предоставляет структуры данных на языке C, слой, обеспечивающий переносимость, поддержку Unicode и интерфейсы для поддержки цикла обработки событий, потоков, динамической загрузки и системы объектов. Ruby/GLib – обертка библиотеки GLib. Поскольку в Ruby уже есть хорошие классы для работы со строками и списками, некоторые функции GLib не реализованы. С другой стороны, Ruby/GLib содержит ряд важных функций для преобразования между объектами на C и на Ruby. Эта библиотека необходима для всех остальных библиотек, входящих в состав Ruby/GTK.
- **Ruby/GIO.** Эта библиотека предоставляет API для работы с томами, дисками, файлами, потоками, значками и системой DNS.
- **Ruby/ATK.** Эта библиотека предоставляет набор интерфейсов для поддержки специальных средства. Приложение или набор средств разработки, поддерживающие интерфейсы ATK, могут применяться с такими инструментами, как считыватели с экрана, лупы и альтернативные устройства ввода.
- **Ruby/Pango.** Библиотека для компоновки и отрисовки текста с упором на интернационализацию и использование кодировки UTF-8. Образует основу для работы с текстами и шрифтами в GTK+ (2.0).
- **Ruby/GdkPixbuf.** Библиотека для загрузки и манипулирования изображениями. Поддерживает многочисленные графические форматы, включая JPEG, PNG, GIF и другие.
- **Ruby/GDK.** Промежуточный слой, изолирующий GTK+ от деталей оконной системы.
- **Ruby/GTK.** Основные виджеты для построения графических интерфейсов.

Помимо основных библиотек, в проект Ruby-GNOME2 включены дополнительные, в том числе добавочные виджеты, средства настройки, структурная графика, аудио, видео, работа с OpenGL 3D, HTML, XML, PDF, SVG и эмуляция терминала.

Официальная домашняя страница проекта Ruby-GNOME2 – http://ruby-gnome2.sourceforge.jp/. Там вы найдете выпущенные версии всех библиотек, ру-

ководство по установке, справочные руководства по API, учебные пособия и примеры программ. Официальный сайт проекта GNOME http://www.gnome.org/, а проекта GTK+ – http://www.gtk.org/.

12.4. QtRuby

Qt это библиотека и комплект средств разработки, созданные и распространяемые компанией Trolltech. Основной упор в Qt сделан на кросс-платформенности, единый программный интерфейс предоставляется для операционных систем Windows, Mac, и UNIX. Разработчику нужно написать код только один раз, он будет оттранслирован на всех трех платформах без модификации.

Qt распространяется на условиях одной из двух лицензий: GPL или коммерческая лицензия для разработки продуктов без раскрытия исходных текстов. Такой же политики двойного лицензирования придерживаются и другие компании, например MySQL. Она позволяет использовать библиотеку в проектах с открытыми исходными текстами, в которых предлагаемые средства находят полезное применение. Но при этом Trolltech может получать доход от продажи коммерческих лицензий клиентам, которых не устраивают ограничения GPL.

12.4.1. Обзор

Интерфейс к Qt из Ruby – результат работы многих людей, прежде всего Арно Рена (Arno Rehn) и Ричарда Дейла (Richard Dale). Распространяется он в виде gem-пакета `qtbindings`, который содержит не только обширный набор классов, относящихся в графическим интерфейсам, но и целый комплект дополнительных средств, часто необходимых программистам (например, библиотеки для работы с XML и SQL). Поддерживаются все компоненты библиотеки Qt.

Ключевой аспект Qt, а значит и QtRuby, концепция сигналов и слотов. Сигналы – это асинхронные события, возникающие, когда в приложении происходит какое-то событие, например, щелчок кнопкой мыши или ввод какого-то текста в поле. Слот – это просто метод, вызываемый в ответ на возникновение сигнала. Для связывания сигналов со слотами мы будем использовать метод `connect`.

Чтобы иметь возможность пользоваться сигналами и слотами, а также многими другими возможностями QtRuby, все наши классы будут наследовать классу `Qt::Object`. Более того, классы, используемые в графических интерфейсах, будут наследовать классу `Qt::Widget`, который, в свою очередь, является производным от `Qt::Object`.

12.4.2. Простое оконное приложение

Приложение QtRuby должно в самом начале загрузить библиотеку Qt. QtRuby раскрывает свою функциональность посредством модуля `Qt` (следовательно, имена всех классов начинаются с префикса `Qt::`). Имена всех классов в исходной библиотеке Qt начинаются с буквы *Q*, но при переходе к QtRuby эта буква опускается. Так, класс, основанный на `QWidget`, в QtRuby называется `Qt::Widget`.

```
require 'Qt'

app = Qt::Application.new(ARGV)
str = Time.now.strftime("Today is %B %d, %Y")
label = Qt::Label.new(str)
label.show
app.exec
```

Рассмотрим этот код подробнее. Вызов `Qt::Application.new` запускает приложение Qt; он инициализирует оконную систему и выполняет подготовительные действия для создания виджетов.

Затем создается объект `Qt::Label` – простейший способ показать текст пользователю. В данном случае текст инициализируется в предыдущей строчке. Следующая строчка говорит метке, что она должна отобразить себя на экране.

На последнем шаге вызов `app.exec` запускает цикл обработки событий. Он не возвращает управления, пока приложение не завершится. Обычно это происходит, когда пользователь нажимает кнопку закрытия окна.

12.4.3. Кнопки

Создание кнопки в QtRuby сводится к созданию экземпляра класса `Qt::PushButton` (см. листинг 12.9 и рис. 12.12). Обычно при нажатии кнопки нужно выполнить некоторое действие. Для этого применяется механизм событий и слотов QtRuby.

Листинг 12.9. Кнопки в QtRuby

```
require 'Qt'

class MyWidget < Qt::Widget
  slots 'buttonClickedSlot()'
  def initialize(parent = nil)
    super(parent)

    setWindowTitle("QtRuby example");
    @lineedit = Qt::LineEdit.new(self)
    @button = Qt::PushButton.new("All Caps!",self)

    connect(@button, SIGNAL('clicked()'),
            self, SLOT('buttonClickedSlot()'))

    box = Qt::HBoxLayout.new
    box.addWidget(Qt::Label.new("Text:"))
    box.addWidget(@lineedit)
    box.addWidget(@button)

    setLayout(box)
  end
```

```
  def buttonClickedSlot
    @lineedit.setText(@lineedit.text.upcase)
  end

end

app = Qt::Application.new(ARGV)
widget = MyWidget.new
widget.show
app.exec
```

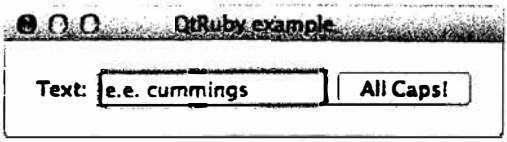


Рис. 12.12. Кнопки в Qt

В этом примере мы создали собственный класс виджета с именем `MyWidget`; он наследует классу `Qt::Widget`, являющемуся предком любого нестандартного виджета.

Перед инициализацией мы подготовили список слотов, которые будут определены в нашем классе. Слоты – это обычные методы класса, но необходимо указать их имена, чтобы во время выполнения QtRuby знала, что мы собираемся использовать их именно в качестве слотов. Метод класса `slots` принимает список строк:

```
slots 'slot1()', 'slot2()'
```

Инициализатор класса принимает аргумент `parent`, он есть почти у всех виджетов в Qt и определяет, какой виджет будет владельцем вновь создаваемого. Значение `nil` означает, что это «виджет верхнего уровня», у которого нет владельца. Концепция «владения», наверное, имеет более понятный смысл в C++; родители владеют своими детьми, то есть при уничтожении или удалении родителя удаляются и все его потомки.

Наш класс создает объект `Qt::LineEdit` для ввода текста и кнопку `Qt::PushButton` с надписью `All Caps!`. В качестве родителя каждому виджету передается `self`. Это означает, что создаваемый экземпляр `MyWidget` «усыновляет» эти виджеты.

Далее мы обращаемся к ключевой части библиотеки Qt – механизму соединения сигналов со слотами. В классе `Qt::Pushbutton` определен сигнал `clicked`, который испускается при нажатии кнопки. Этот сигнал можно соединить со слотом, в данном случае с методом `buttonClickedSlot`. Имя слота может быть любым, суффикс `Slot` мы употребили просто для наглядности.

В самом конце мы создаем экземпляр класса `Qt::HBoxLayout`. При добавлении виджетов в этот контейнер он автоматически изменяет их размеры, так что нам беспокоиться больше не о чем.

12.4.4. Текстовые поля

Как видно из листинга 12.9, в QtRuby есть класс `Qt::LineEdit` для ввода одной строки текста. Для ввода нескольких строк предназначен класс `Qt::TextEdit`.

В листинге 12.10 демонстрируется многострочное текстовое поле. Под ним расположена метка, в которой отображается текущая длина текста (рис. 12.13).

Листинг 12.10. Простой редактор в Qt

```
require 'Qt'

class MyTextWindow < Qt::Widget
  slots 'theTextChanged()'

  def initialize(parent = nil)
    super(parent)

    @textedit = Qt::TextEdit.new(self)
    @textedit.setWordWrapMode(Qt::TextOption::WordWrap)
    @textedit.setFont( Qt::Font.new("Times", 24) )

    @status = Qt::Label.new(self)

    box = Qt::VBoxLayout.new
    box.addWidget(@textedit)
    box.addWidget(@status)
    setLayout(box)

    @textedit.insertPlainText("This really is an editor")

    connect(@textedit, SIGNAL('textChanged()'),
            self, SLOT('theTextChanged()'))
  end
  def theTextChanged
    text = "Length: " + @textedit.toPlainText.length.to_s
    @status.setText(text)
  end

end

app = Qt::Application.new(ARGV)
widget = MyTextWindow.new
widget.setWindowTitle("QtRuby Text Editor")
widget.show
app.exec
```

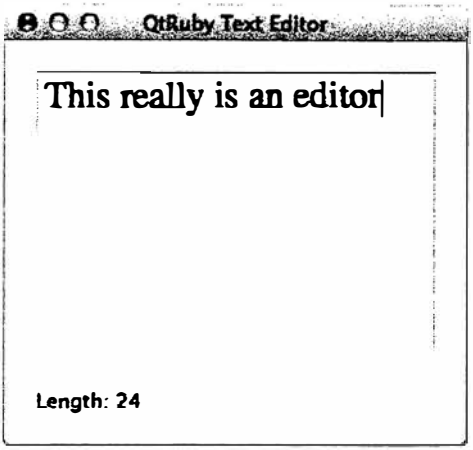


Рис. 12.13. Простой редактор в Qt

Виджет конструируется примерно так же, как в предыдущем примере. Но теперь мы создаем объект `Qt::TextEdit`, а также метку `Qt::Label` для показа текущего состояния.

Стоит отметить, что для объекта `@textedit` мы указали шрифт Times высотой 24 пункта. У каждого класса, наследующего `Qt:Widget` (в том числе и у `Qt::TextEdit`) есть свойство `font`, которое можно опросить или установить.

Затем мы создаем менеджер вертикальной компоновки (`Qt::VBoxLayout`), который будет содержать дочерние виджеты, добавляем в него виджет `@textedit` и связываем сигнал `textChanged` с определенным нами слотом `theTextChanged`.

В методе `theTextChanged` мы запрашиваем у редактора текст и получаем его длину, а затем записываем возвращенное значение в метку `@status`.

Отметим, что весь механизм сигналов и слотов работает асинхронно. После того как приложение входит в цикл обработки событий (`app.exec`), оно уже не получает управления явно. Вот почему сигналы и слоты так важны. Мы определяем события, которые нас интересуют (сигналы), и действия, которые нужно выполнить при возникновении таких событий (слоты).

12.4.5. Прочие виджеты

В библиотеке Qt есть еще много встроенных виджетов, например, переключатели, флажки и т.п. В листинге 12.11 продемонстрированы некоторые из них, а на рис. 12.14 показано, как выглядит окно приложения.

Листинг 12.11. Прочие виджеты в Qt

```
require 'Qt'

class MyWindow < Qt::Widget
  slots 'somethingClicked(QAbstractButton *)'
  def initialize(parent = nil)
```

```
    super(parent)

    groupbox = Qt::GroupBox.new("Some Radio Button",self)

    radio1 = Qt::RadioButton.new("Radio Button 1", groupbox)
    radio2 = Qt::RadioButton.new("Radio Button 2", groupbox)
    check1 = Qt::CheckBox.new("Check Box 1", groupbox)

    vbox = Qt::VBoxLayout.new
    vbox.addWidget(radio1)
    vbox.addWidget(radio2)
    vbox.addWidget(check1)
    groupbox.setLayout(vbox)

    bg = Qt::ButtonGroup.new(self)
    bg.addButton(radio1)
    bg.addButton(radio2)
    bg.addButton(check1)

    connect(bg, SIGNAL('buttonClicked(QAbstractButton *)'),
            self, SLOT('somethingClicked(QAbstractButton *)') )

    @label = Qt::Label.new(self)

    vbox = Qt::VBoxLayout.new
    vbox.addWidget(groupbox)
    vbox.addWidget(@label)
    setLayout(vbox)
  end

  def somethingClicked(who)
    @label.setText("You clicked on a " + who.className)
  end

end

app = Qt::Application.new(ARGV)
widget = MyWindow.new
widget.show
app.exec
```

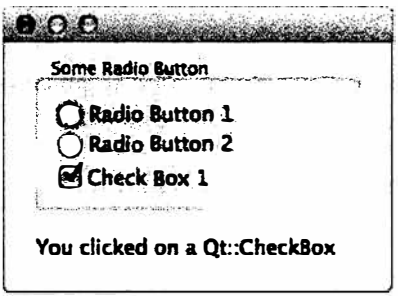


Рис. 12.14. Прочие виджеты в Qt

В этом классе мы сначала создаем объект Qt::GroupBox – контейнер с рамкой и необязательным заголовком, в который можно помещать другие виджеты. Далее создаются два переключателя Qt::RadioButtons и флажок Qt::CheckBox, а в качестве их родителя указывается ранее созданный контейнер.

Затем создается менеджер компоновки Qt::VBoxLayout, в который помещаются переключатели и флажок, после чего он связывается с групповым контейнером и начинает управлять его размещением на экране.

Следующий важный шаг – создание объекта Qt::ButtonGroup, в который помещаются флажок и переключатели. Qt::ButtonGroup предназначен для логической группировки кнопок, флажков и переключателей. На их визуальное расположение он никак не влияет, зато обеспечивает, к примеру, *взаимное исключение* (гарантирует, что только один из группы виджетов может быть отмечен). В данном случае этот объект будет источником сигнала buttonClicked, который испускается при нажатии любой кнопки в группе.

Этот сигнал отличается от виденных ранее тем, что ему сопутствует аргумент, а именно объект, по которому щелкнули мышкой. Обратите внимание на то, как синтаксис – QAbstractButton * – напоминает о C++-ных корнях Qt. В некоторых случаях употребления принятой в C++ нотации для обозначения типов параметров не избежать (хотя в будущих версиях это, возможно, и исправят).

В результате такого вызова метода connect при щелчке по любому виджету, принадлежащему группе, этот виджет будет передан слоту somethingClicked. Наконец, мы создаем метку Qt::Label, контейнер Qt::VBoxLayout и увязываем все вместе.

Внутри слота somethingClicked мы модифицируем текст метки при щелчке по любому переключателю или флажку. В данном случае выводится имя класса объекта, который испустил сигнал, приведший к вызову слота.

Если встроенных виджетов недостаточно, то Qt предоставляет мощную систему рисования для создания собственных. В листинге 12.12 приведен небольшой пример, иллюстрирующий малую часть возможностей.

Листинг 12.12. Нестандартный виджет TimerClock

```
require 'Qt'

class TimerClock < Qt::Widget

    def initialize(parent = nil)
        super(parent)

        @timer = Qt::Timer.new(self)
        connect(@timer, SIGNAL('timeout()'), self, SLOT('update()'))
        @timer.start(25)

        setWindowTitle('Stop Watch')
        resize(200, 200)
```

```
    end

    def paintEvent(e)
        fastHand = Qt::Polygon.new([Qt::Point.new(7, 8),
                                    Qt::Point.new(-7, 8),
                                    Qt::Point.new(0, -80)])

        secondHand = Qt::Polygon.new([Qt::Point.new(7, 8),
                                      Qt::Point.new(-7, 8),
                                      Qt::Point.new(0, -65)])

        secondColor = Qt::Color.new(100, 0, 100)
        fastColor = Qt::Color.new(0, 150, 150, 150)

        side = [width, height].min
        time = Qt::Time.currentTime

        painter = Qt::Painter.new(self)
        painter.renderHint = Qt::Painter::Antialiasing
        painter.translate(width() / 2, height() / 2)
        painter.scale(side / 200.0, side / 200.0)

        painter.pen = Qt::NoPen
        painter.brush = Qt::Brush.new(secondColor)

        painter.save
        painter.rotate(6.0 * time.second)
        painter.drawConvexPolygon(secondHand)
        painter.restore

        painter.pen = secondColor
        (0...12).each do |i|
            painter.drawLine(88, 0, 96, 0)
            painter.rotate(30.0)
        end
        painter.pen = Qt::NoPen
        painter.brush = Qt::Brush.new(fastColor)

        painter.save
        painter.rotate(36.0 * (time.msec / 100.0) )
        painter.drawConvexPolygon(fastHand)
        painter.restore

        painter.pen = fastColor
        (0...60).each do |j|
            if (j % 5) != 0
                painter.drawLine(92, 0, 96, 0)
            end
            painter.rotate(6.0)
        end
        painter.end
```

```
    end
end

app = Qt::Application.new(ARGV)
wid = TimerClock.new
wid.show
app.exec
```

Созданный в этом примере виджет называется `TimerClock`. В инициализаторе мы создаем объект `Qt::Timer`, который конфигурируется для периодического испускания сигнала. Его сигнал `timeout` мы соединяем со слотом `update` нашего виджета. Это встроенный слот, он заставляет виджет перерисовать себя.

Таймер запускается методом `start`. Переданный ему аргумент говорит, что таймер должен срабатывать (и испускать сигнал `timeout`) каждые 25 миллисекунд. Следовательно, слот `update` будет вызываться каждые 25 миллисекунд.

Далее определяется метод `paintEvent`. Мы переопределяем одноименный метод класса `Qt::Widget`. Когда виджет собирается перерисовать себя (то есть при срабатывании таймера), он вызывает этот метод. Переопределяя его, мы решаем, как виджет должен отображаться на экране. Код этого метода вызывает различные графические примитивы рисования.

Начиная с этого места идет сплошная геометрия. Мы создаем несколько многоугольников `Qt::Polygon`, представляющих стрелки часов. Ориентация многоугольников не имеет значения, потому что манипулировать ими мы будем позже.

Задаются значения нескольких свойств. Устанавливаются цвета `Qt::Color` обеих стрелок. Аргументами инициализатора `Qt::Color` являются значения в формате RGB с необязательной альфа-прозрачностью.

Часы должны быть квадратными, поэтому в переменную `side` (длина стороны) записывается минимум из ширины и высота виджета. Кроме того, мы запоминаем текущее время, обращаясь к методу `Qt::Time.currentTime`.

Далее создается объект `Qt::Painter`, и с его помощью мы начинаем рисовать. Задается режим сглаживания (antialiasing), чтобы на стрелках часов не было «лесенки». Начало координат помещается в центр области рисования (`painter.translate(width/2, height/2)`). Для объекта `Painter` устанавливается масштаб в предположении, что сторона квадрата составляет 200 единиц. Если размер окна изменится, то масштабирование будет произведено автоматически.

Затем выполняется последовательность операций рисования. Различные геометрические преобразования (например, поворот), сопровождаются парой вызовов `painter.save` и `painter.restore`. Метод `save` сохраняет текущие свойства объекта `Painter` в стеке, чтобы их можно было позднее восстановить.

Программа рисует обе стрелки, предварительно повернув их на нужный угол в соответствии с текущим временем. Кроме того, мы наносим риски вдоль границы циферблата.

И напоследок мы сообщаем объекту `Painter`, что рисование закончилось (вызывая метод `painter.end`). Довершают картину четыре строчки, в которых создаются объект приложения `Qt::Application` и наш виджет, а затем запускается цикл обработки событий. На рис. 12.15 показан конечный результат.

12.4.6. Дополнительные замечания

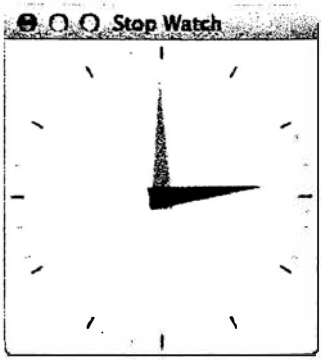


Рис. 12.15. Виджет TimerClock

Поскольку библиотека Qt написана на C++, неизбежны некоторые идиомы, отражающие ограничения этого языка. Иногда перевод на Ruby не выглядит на 100% естественным, поскольку в Ruby аналогичные вещи делаются несколько иначе. Поэтому в ряде случаев вводится избыточность, позволяющая выражать свои намерения «по-рубистски».

Например, имена методов, в которых сохранена «верблюжьяНотация», свойственная C++, можно записывать и с подчерками. Так, следующие два вызова эквивалентны:

```
Qt::Widget::minimumSizeHint
Qt::Widget::minimum_size_hint
```

Все методы установки свойств в Qt начинаются со слова set, например, Qt::Widget::setMinimumSize. В Ruby можно это слово опускать и пользоваться присваиванием, например:

```
widget.setMinimumSize(50)
widget.minimumSize = 50  # то же самое
widget.minimum_size = 50 # то же самое
```

Аналогично, в Qt имена методов, возвращающих булево значение, часто начинаются с is или has, например, Qt::Widget::isVisible. QtRuby позволяет именовать их в духе Ruby:

```
a.isVisible
a.visible?     # то же самое
```

12.5. Swing

В мире Java стандартным ГИП является Swing. Благодаря JRuby Swing можно использовать из Ruby непосредственно. Это зрелая и стабильная библиотека, которая отлично работает всюду, где есть Java.

Развернутое обсуждение Swing далеко выходит за рамки этой книги, но несколько примеров, позволяющих почувствовать, что такое Swing, мы все же приведем.

В следующем примере отображается простое окно:

```
include Java

import javax.swing.JFrame

class SimpleWindow < JFrame

   def initialize
      super "Just another window..."
      init_gui
   end
```

```
    def init_gui
        set_size 400, 300
        self.default_close_operation = JFrame::EXIT_ON_CLOSE
        self.location_relative_to = nil
        self.visible = true
    end

end

SimpleWindow.new
```

Прежде всего, мы включаем модуль Java и импортируем классы (в JRuby это, естественно, вызов метода). Наш класс `SimpleWindow` наследует `JFrame`; для его инициализации мы сначала вызываем метод инициализации родителя, а затем выполняем свой код (в данном случае вызываем метод `init_gui`), который создает окно с нужными нам параметрами.

Программисты на Java привыкли к именам вида `setSize` или `setDefaultCloseOperation`, и — пожалуйста — их можно использовать. Но, сохраняя верность стилю Ruby, мы разрешаем также синонимы – имена, написанные строчными буквами с подчерком в качестве разделителя слов (`set_size` и `set_default_close_operation`). В большинстве случаев метод с именем, скажем, `setWhatever(value)` можно записать в виде `set_whatever(value)`, а также в виде присваивания `self.whatever = value`. В последнем случае обязательно наличие `self`, чтобы отличить метод от локальной переменной.

Вот чуть более интересный пример с кнопками:

```
include Java

import javax.swing.JFrame
import javax.swing.JButton
import javax.swing.JPanel

class ButtonWindow < JFrame
    def initialize
        super "Button example..."
        init_gui
    end

    def init_gui
        win = self
        panel = JPanel.new
        self.content_pane.add panel
        panel.layout = nil
        eat = JButton.new "Eat Me"
        eat.set_bounds 50, 60, 80, 30
        eat.add_action_listener {|event| win.set_size 600, 400 }

        drink = JButton.new "Drink Me"
```

```
        drink.set_bounds 50, 90, 80, 30
        drink.add_action_listener {|event| win.set_size 200, 150 }

        panel.add eat
        panel.add drink

        self.default_close_operation = JFrame::EXIT_ON_CLOSE
        set_size 300, 200
        self.location_relative_to = nil
        self.visible = true
    end
end

ButtonWindow.new
```

Обратите внимание на использование контейнера `JPanel`, содержащего две кнопки. С каждой кнопкой ассоциирован блок, который выполняется при нажатии кнопки. В остальном этот код мало чем отличается от предшествующего.

Библиотеке Swing посвящены бесчисленные книги и пособия (как в сети, так и в печатном виде). Для получения дополнительных сведений обратитесь к любой из них.

12.6. Другие библиотеки для создания графических интерфейсов

Мы предупреждали, что вы можете и не найти своего любимого графического интерфейса. Но напоследок кратко упомянем имеющиеся альтернативы.

12.6.1. UNIX и X11

Систему *X Window System* в разговорной речи называют (не совсем корректно) просто X Windows. Вероятно, она является прародителем если не всех, то абсолютного большинства графических интерфейсов пользователя.

Пользователи всех вариантов UNIX давно уже знакомы с X (как пользователи, а то и как разработчики). К достоинствам X следует отнести широкую известность, переносимость и богатый набор возможностей. К недостаткам – сложность работы.

Не удивительно, что существуют библиотеки для работы с X из Ruby. Из-за их сложности мы не приводим документацию, а отсылаем к архиву приложений Ruby Application Archive, где вы найдете библиотеку Xlib, написанную Кадзухиро Иосида (Kazuhiro Yoshida, известный также как moriq), и Ruby/X11 Мэтью Бушара (Mathieu Bouchard, он же matju). Обе годятся для создания X-клиентов.

12.6.2. FXRuby (FOX)

Библиотека FOX (Free Objects for X) не так распространена, как Tk или GTK+, но когда-то пользовалась популярностью у программистов на Ruby. Отчасти это обусловлено великолепным интерфейсом к ней FXRuby, которым мы обязаны Лайлу

Джонсону (Lyle Johnson). Но в 2010 году он отошел от проекта и с тех пор библиотека сопровождается не так активно, как раньше. Однако она по-прежнему работоспособна. Прекрасная книга «FXRuby: Create Lean and Mean GUIs with Ruby», написанная Лайлом, все еще есть в продаже.

12.6.3. RubyMotion для iOS и Mac OS X

Каркас RubyMotion позволяет написанным на Ruby программам обращаться к платформенным графическим средствам, имеющимся в системах iOS и Mac OS X. По существу, это исполняющая среда Ruby, написанная на Objective-C, «родном» языке iOS и Mac OS X. Каркас включает также компилятор и инструментарий для комплектации и распространения Ruby-приложения в виде двоичного кода для iOS и Mac OS X. Автор RubyMotion, Лоран Сансонетти (Laurent Sansonetti), до этого создал ныне заброшенный проект MacRuby, интерпретатор Ruby, написанный на Objective-C (и тесно интегрированный с этим языком).

На момент написания этой книги были анонсированы, но еще не выпущены набор инструментов и компилятор на платформе Android. Отметим также, что RubyMotion – коммерческий продукт, он стоит 199 долларов.

12.6.4. Windows Win32 API

В Ruby есть возможность работать с полным Windows API напрямую благодаря библиотеке `WIN32API`. Один такой пример мы приведем в главе 14. По крайней мере, теоретически это позволяет Ruby-программе создавать платформенные виджеты Windows и манипулировать ими. Лично я не рекомендую такой подход, но о его существовании вы должны знать.

12.7. Заключение

В этой главе мы привели обзор различных средств разработки графических интерфейсов пользователя для Ruby. Мы познакомились с общими концепциями: циклами обработки событий, сообщениями или сигналами и т.п. Была продемонстрирована работа с различными виджетами: кнопками, флажками, переключателями, текстовыми полями и т.д.

Мы рассмотрели реализацию общих концепций на примере библиотек Tk, GTK, Qt и Shoes. Мы видели, что в каждом случае применяется своя терминология и слегка отличающиеся варианты основной парадигмы. Отметили мы и специфические средства и достоинства, присущие каждой библиотеке.

А теперь перейдем совсем к другой теме. В главе 13 мы рассмотрим работу с потоками в Ruby.

Глава 13. Потоки и параллелизм

Он тянет нить своего красноречия искуснее,
чем развивает свои доводы.

Шекспир «Бесплодные усилия любви»,
акт V, сцена 1

Одно из чудес современных компьютеров — умение делать сразу несколько вещей. Даже во времена одноядерных процессоров единственный ЦП мог переключаться между задачами достаточно быстро, чтобы можно было считать, что он занимается одновременно несколькими делами. А с появлением недорогих компьютеров, оснащенных несколькими процессорами, иллюзия стала реальностью. Так что ныне мы считаем само собой разумеющимся, что можем загружать gem-пакет и в то же время бродить по Интернету и распечатывать документ. Этой параллелизм предстает программисту в двух принципиально разных формах: процессы и потоки. Вообще говоря, процесс — это сравнительно автономная единица с собственным потоком выполнения и адресным пространством, изолированная от всех прочих процессов (за одним важным исключением — операционная система должна защищать процессы от взаимовлияния, но в то же время предоставлять механизмы совместного использования памяти).

Потоки иногда называют еще *облегченными процессами*. Это просто способ обеспечить параллельное выполнение без накладных расходов, связанных с контекстным переключением между процессами. Впрочем, общего согласия о том, что такое поток, нет, мы приводим самое распространенное определение.

Поток существует внутри процесса, имеет доступ к его адресному пространству и открытым им файлам. Потоки хороши тем, что так много разделяют со своим родительским процессом, что их создание обходится сравнительно дешево. А плохи тем, что, разделяя столь много с родительским процессом — и другими потоками в нем, — они могут мешать друг другу.

Потоки полезны, например, тогда, когда некоторые части программы могут работать независимо друг от друга. Применяются они и в тех случаях, когда приложение тратит много времени на ожидание события. Часто, пока один поток ждет, другой может выполнять полезную работу. С другой стороны, у потоков есть и недостатки. Переключение между потоками может замедлить работу отдельных задач. Иногда доступ к ресурсу принципиально должен осуществляться строго последовательно, поэтому потоки не дадут никакого выигрыша. И, нако-

нец, бывает так, что накладные расходы на синхронизацию доступа к глобальным ресурсам превышают экономию, достигаемую за счет использования нескольких потоков.

С потоками неизбежно связана некоторая потеря производительности, как в плане потребления памяти, так и в плане быстродействия. По этой и ряду других причин некоторые авторитеты вообще рекомендуют держаться подальше от многопоточного программирования. Действительно, такие программы сложны и подвержены ошибкам, которые трудно отлаживать. Но при наличии многочисленных ядер в современных процессорах увеличение скорости за счет распараллеливания задач может оказаться весьма значительным. Мы оставим читателю самому решать, когда стоит применять эту технику.

Проблемы, связанные с несинхронизированными потоками, хорошо известны. При одновременном доступе к глобальным данным со стороны нескольких потоков данные могут быть повреждены. Если один поток делает какое-то допущение о том, что успел выполнить другой поток, возможна *гонка* (race condition); обычно это приводит к «недетерминированному» коду, который дает разные результаты при каждом запуске. Наконец, существует опасность *взаимоблокировки*, когда ни один поток не может продолжить выполнение, поскольку ожидает ресурс, занятый другим потоком. Код, написанный так, что ни одна из этих проблем не возникает, называется *потокобезопасным*.

В Ruby применение потоков сопряжено с дополнительными трудностями. В основной версии Ruby (MRI, Matz's Ruby Interpreter) потоки создаются обычным образом. Но – и это очень большое *но* – интерпретатор содержит унаследованный код, не являющийся потокобезопасным. В результате разрешается исполнять только один поток в каждый момент времени. Отсюда низкие накладные расходы, но и крайне низкий уровень параллелизма. В MRI несколько потоков полезны только в том случае, когда программа большую часть времени ожидает завершения ввода-вывода.

Совсем не так обстоит дело в интерпретаторах Rubinius и JRuby. JRuby – это реализация Ruby для виртуальной машины Java Virtual Machine, а Rubinius – экспериментальный интерпретатор, в котором сделана попытка реализовать Ruby на самом Ruby – в той мере, в какой это возможно. Потоки в JRuby опираются на механизмы многопоточности, предоставляемые самой платформой Java, тогда как в Rubinius используются потоки операционной системы, как в MRI. Но в отличие от MRI, Rubinius и JRuby не занимаются искусственной синхронизацией потоков. Это означает, что в интерпретаторах, отличных от MRI, потоки *могут* выполняться одновременно, задействуя все имеющиеся на машине процессоры.

Для правильной работы с потоками необходимо знать о методах синхронизации, которые позволяют контролировать доступ к переменным и ресурсам, защищать критические секции программы и избегать взаимоблокировок. Мы рассмотрим их в этой главе и проиллюстрируем на примерах.

13.1. Создание потоков и манипулирование ими

К числу основных операций над потоками относятся создание потока, передача ему входной информации и получение результатов, останов потока и т.д. Можно получить список запущенных потоков, опросить состояние потока и выполнить ряд других проверок.

13.1.1. Создание потоков

В любой программе есть по меньшей мере один поток – начальный поток управления, который мы называем «главным». Программа становится многопоточной, когда запускает дополнительные потоки.

Создать поток в Ruby просто. Достаточно вызвать метод new и присоединить блок, который будет исполняться в потоке:

```
thread = Thread.new do
  # Предложения, исполняемые в потоке...
end
```

Синонимом new является метод fork, поэтому для создания потока можно было бы написать Thread.fork вместо Thread.new. Вне зависимости от имени возвращается экземпляр класса Thread. С его помощью можно управлять потоком.

А если нужно передать потоку параметры? Достаточно передать их методу Thread.new:

```
thread2 = Thread.new(99, 100) do |a, b|
   # начальное значение параметра a равно 99
   # начальное значение параметра b равно 100
end
```

Таким образом, блок в этом примере принимает два параметра, a и b, которые в начальный момент равны соответственно 99 и 100.

Поскольку в основе потоков лежат обычные блоки кода, поток может обращаться к переменным из той области видимости, в которой был создан. Это значит, что следующий поток может изменять значения x и y, поскольку эти переменные находились в области видимости, когда создавался ассоциированный с потоком блок.

```
x = 1
y = 2

thread3 = Thread.new(99, 100) do |a, b|
   # x и y видимы и могут быть изменены!
   x = 10
   y = 11
end

puts x, y
```

Опасность таких действий таится в ответе на вопрос: *когда*. Когда именно изменяются значения x и y? Мы можем быть уверены, что это происходит после начала

выполнения потока thread3, но это и все, что можно сказать. Начав работать, поток живет своей жизнью, и если не предпринять каких-то мер по синхронизации, вы никогда не узнаете, как быстро он продвигается вперед. Из-за этой непредсказуемости невозможно сказать, что будет напечатано в примере выше. Если thread3 завершается очень быстро, то, возможно, будут напечатаны значения 10 и 11. Наоборот, если thread3 задержался на старте, то могут быть напечатаны 1 и 2. Существует даже шанс, хотя и небольшой, что мы увидим значение 1, а сразу за ним 11, – если thread3 и главный поток идут ноздря в ноздрю. В реальных программах мы всеми силами стремимся избежать такого неопределенного поведения.

13.1.2. Доступ к поточно-локальным переменным

Так как же все-таки программировать потоки, которые могли бы разделять данные с прочими частями программы, не портя переменные вне своей области видимости? К счастью, в Ruby для этой цели предусмотрен специальный механизм. Хитрость в том, чтобы рассматривать объект Thread как хэш и читать и изменять значения по ключу. К таким поточно-локальным данным можно обращаться из любого места внутри *и* вне потока, получив тем самым удобный механизм совместного использования. Ниже приведен пример работы с поточно-локальными данными:

```
thread = Thread.new do
  t = Thread.current
  t[:var1] = "Это строка"
  t[:var2] = 365
end

sleep 1         # Пусть поток немного побездельничает

# Доступ к локальным данным потока извне...

x = thread[:var1]                    # "Это строка"
y = thread[:var2]                    # 365

has_var2 = thread.key?("var2")# true
has_var3 = thread.key?("var3")# false
```

Как видим, поток может инициализировать собственные локальные данные; для этого нужно вызвать метод Thread.current, который возвращает текущий экземпляр Thread, а затем рассматривать этот объект как хэш. В классе Thread имеется метод экземпляра key?, который сообщает, было ли присвоено значение указанному ключу. Еще одна приятная особенность поточно-локальных данных – тот факт, что они не исчезают даже после того, как создавший их поток завершил работу.

Следует понимать, что хотя потоки обращаются со своими поточно-локальными данными как с хэшем, на самом деле это не настоящий хэш: у него нет большинства привычных методов Enumerable, в частности each и find. Но еще важнее то, что

потоки очень разборчивы в части допустимых ключей поточно-локальных данных. Ключами могут быть только символы и строки, причем строки неявно преобразуются в символы. Это видно из следующего примера:

```
thread = Thread.new do
   t = Thread.current
   t["started_as_a_string"] = 100
   t[:started_as_a_symbol] = 101
end

sleep 1          # Пусть поток немного побездельничает

a = thread[:started_as_a_string]    # 100
b = thread["started_as_a_symbol"]   # 101
```

Важно отметить, что поточно-локальные данные решают проблему *где*, предоставляя место для размещения данных, но не решают проблему *когда*: без синхронизации у нас нет ни малейшего представления о том, когда поток соберется что-то сделать со своими локальными данными. Распространенное решение – «вернуть» поточно-локальные данные, когда поток завершает выполнение. В этом случае никакой синхронизации не нужно, потому что поток уже завершился.

13.1.3. Опрос и изменение состояния потока

В классе `Thread` есть несколько полезных методов класса. Метод `list` возвращает массив «живых» потоков, метод `main` возвращает ссылку на главный поток программы, который породил все остальные, а метод `current` позволяет потоку идентифицировать самого себя.

```
t1 = Thread.new { sleep 100 }

t2 = Thread.new do
  if Thread.current == Thread.main
    puts "Это главный поток."    # НЕ печатается
  end
  1.upto(1000) { sleep 0.1 }
end

count = Thread.list.size         # 3

if Thread.list.include?(Thread.main)
  puts "Главный поток жив."      # Печатается всегда!
end

if Thread.current == Thread.main
  puts "Я главный поток."        # Здесь печатается...
end
```

Методы `exit`, `pass`, `start`, `stop` и `kill` служат для управления выполнением потоков (как изнутри, так и извне):

```
# В главном потоке...
Thread.kill(t1)                  # Завершить этот поток
Thread.pass                      # Уступить свой квант
t3 = Thread.new do
  sleep 20
  Thread.exit                    # Выйти из потока
  puts "Так не бывает!"          # Никогда не выполняется
end

Thread.kill(t2)                  # Завершить t2

# Выйти из главного потока (все остальные тоже завершаются)
Thread.exit
```

Отметим, что не существует метода экземпляра stop, поэтому поток может приостановить собственное выполнение, но не выполнение другого потока.

Существуют различные методы для опроса состояния потока. Метод экземпляра alive? сообщает, является ли данный поток «живым» (не завершил выполнение), а метод stop? – находится ли он в состоянии «приостановлен». Отметим, что методы alive? и stop? – не полные противоположности: оба возвращают true, если поток спит, потому что спящий поток 1) еще жив и 2) в данный момент ничего не делает.

```
count = 0
t1 = Thread.new { loop { count += 1 } }
t2 = Thread.new { Thread.stop }

sleep 1                  # Пусть поток немного побездельничает

flags = [t1.alive?,      # true
         t1.stop?,       # false
         t2.alive?,      # true
         t2.stop?]       # true
```

Получить состояние потока позволяет метод status. Множество возвращаемых им значений напоминает сборную солянку: если поток выполняется, метод возвращает значение "run", если приостановлен, спит или ожидает завершения ввода-вывода – то "sleep". Если поток нормально завершился, возвращается false, а если трагически погиб в результате исключения, то nil. Все это иллюстрируется в следующем примере:

```
t1 = Thread.new { loop {} }
t2 = Thread.new { sleep 5 }
t3 = Thread.new { Thread.stop }
t4 = Thread.new { Thread.exit }
t5 = Thread.new { raise "exception" }

sleep 1                  # Пусть поток немного побездельничает

s1 = t1.status           # "run"
```

```
s2 = t2.status          # "sleep"
s3 = t3.status          # "sleep"
s4 = t4.status          # false
s5 = t5.status          # nil
```

Потоки знают о глобальной переменной $SAFE, но на самом деле она вовсе не такая глобальная, как кажется, потому что у каждого потока она своя. Выглядит противоречиво, но стоит ли жаловаться, если она позволяет разным потокам работать с разным уровнем безопасности? Поскольку переменная $SAFE отнюдь не является глобальной, когда речь заходит о потоках, у метода экземпляра Thread имеется метод safe_level, который возвращает уровень безопасности данного потока.

```
t1 = Thread.new { $SAFE = 1; sleep 5 }
t2 = Thread.new { $SAFE = 3; sleep 5 }
sleep 1
lev0 = Thread.main.safe_level    # 0
lev1 = t1.safe_level             # 1
lev2 = t2.safe_level             # 3
```

У потоков в Ruby имеется числовой приоритет: чем он больше, тем чаще планировщик выделяет потоку процессор. Узнать и изменить приоритет потока позволяет акцессор priority:

```
t1 = Thread.new { loop { sleep 1 } }
t2 = Thread.new { loop { sleep 1 } }
t2.priority = 3    # Установить для потока t2 приоритет 3
p1 = t1.priority   # 0
p2 = t2.priority   # 3
```

Специальный метод pass позволяет передать управление планировщику. Иными словами, поток просто уступает свой временной квант, но не останавливается и не засыпает.

```
t1 = Thread.new do
   Thread.pass
   puts "Первый поток"
end

t2 = Thread.new do
   puts "Второй поток"
end

sleep 3              # Дать возможность потокам переключиться
```

В этом искусственном примере второй поток, скорее всего, напечатает сообщение раньше первого. Если убрать вызов pass, то, наверное, первый поток – запущенный раньше – выиграл бы состязание. Но, конечно, для несинхронизированных потоков никаких гарантий не дается. Метод pass и приоритеты потоков – это лишь ***пожелания*** планировщику, для синхронизации потоков они не предназначены.

Выполнение приостановленного потока можно возобновить методами методами run или wakeup:

```
t1 = Thread.new do
  Thread.stop
  puts "Здесь есть изумруд."
end

t2 = Thread.new do
  Thread.stop
  puts "Вы находитесь в точке Y2."
end

sleep 0.5              # Пусть потоки запустятся

t1.wakeup
t2.run
```

Между этими методами есть тонкое различие. Метод `wakeup` изменяет состояние потока, так что он становится готовым к выполнению, но не запускает его немедленно. Метод же `run` пробуждает поток и сразу же планирует его выполнение. В данном случае мы получили следующий результат:

```
Вы находитесь в точке Y2.
Здесь есть изумруд.
```

Повторим еще раз — в зависимости от причуд планировщика и фазы луны строки могли бы быть напечатаны в другом порядке. Ни в коем случае не следует реализовывать синхронизацию потоков на основе методов `stop`, `wakeup` и `run`.

Метод экземпляра `raise` возбуждает исключение в потоке, от имени которого вызван. Его необязательно вызывать в том потоке, которому адресовано исключение.

```
factorial1000 = Thread.new do
  begin
    prod = 1
    1.upto(1000) {|n| prod *= n }
    puts "1000! = #{prod}"
  rescue
    # Ничего не делать...
  end
end

sleep 0.01                 # На вашей машине значение может быть иным.
if factorial1000.alive?
  factorial1000.raise("Стоп!")
  puts "Вычисление было прервано!"
else
  puts "Вычисление успешно завершено."
end
```

Поток, запущенный в этом примере, пытается вычислить факториал 1000. Если для этого не хватило одной сотой секунды, то главный поток потеряет терпе-

ние и завершит его. Следовательно, на относительно медленной машине будет напечатано сообщение «Вычисление было прервано!».

13.1.4. Назначение рандеву (и получение возвращенного значения)

Иногда главный поток хочет дождаться завершения другого потока. Для этой цели предназначен метод join:

```
t1 = Thread.new { do_something_long() }

do_something_brief()
t1.join              # Ждать завершения t1
```

Типичная ошибка – запустить несколько потоков, а затем позволить главному потоку завершиться. Беда в том, что вместе с главным потоком Ruby завершает и весь процесс. Например, следующий код никогда не напечатал бы окончательный ответ, не будь в конце вызова join:

```
meaning_of_life = Thread.new do
  puts "Смысл жизни заключается в..."
  sleep 2
  puts 42
end

meaning_of_life.join     # Дождаться завершения потока
```

Существует полезная идиома – вызывать метод join, чтобы дождаться завершения всех остальных «живых» потоков.

```
Thread.list.each { |t| t.join if t != Thread.current }
```

Обратите внимание на проверку того, что поток не является текущим, перед тем как вызывать join. Если поток, пусть даже главный, попытается вызвать join для себя самого, произойдет ошибка.

Если два потока попытаются одновременно вызвать join друг для друга, возникнет взаимоблокировка. Интерпретатор обнаружит это и возбудит исключение.

```
thr = Thread.new { sleep 1; Thread.main.join }

thr.join             # Взаимоблокировка!
```

Как мы уже видели, с потоком связан блок. Всякий, кто хоть немного знаком с Ruby, понимает, что блок может возвращать значение. Следовательно, и сам поток может возвращать значение. Метод value неявно вызывает join и ждет, пока указанный поток завершится, а потом возвращает значение последнего вычисленного в потоке выражения.

```
max = 10000
t = Thread.new do
   sleep 0.2          # Имитировать глубокое раздумье
   42
end

puts "Секретное значение равно #{t.value}"
```

13.1.5. Обработка исключений

Что произойдет, если в потоке возникнет исключение? При нормальных обстоятельствах исключение, возникшее в дополнительном потоке, не возбуждается в главном. Ниже приведен пример потока, возбуждающего исключение:

```
t1 = Thread.new do
   puts "Hello"
   sleep 1
   raise "какое-то исключение"
end

t2 = Thread.new do
   sleep 2
   puts "Привет из другого потока"
end

sleep 3
puts "Конец"
```

В этом пример оба потока печатают свой привет, а главный поток возвещает о конце.

Исключение, возникшее внутри потока, не возбуждается, пока для этого потока не будет вызван метод `join` или `value`. Проверить, что поток завершился с ошибкой, и известить об этом, должен какой-то другой поток. Ниже показано, как безопасно перехватить исключение, возбужденное в потоке:

```
t1 = Thread.new do
   raise "О нет!"
   puts "Это никогда не печатается"
end

begin
   t1.status        # nil означает, что произошло исключение
   t1.join
rescue => e
   puts "Поток возбудил исключение #{e.class}: #{e.message}"
end
```

В этом примере будет напечатано сообщение «Поток возбудил исключение: О нет!». Важно, чтобы исключения в потоках возбуждались в известной точке, чтобы их можно было перехватить или возбудить повторно, не прерывая критическую секцию в каком-то другом потоке.

При отладке многопоточной программы иногда бывает полезен флаг `abort_on_exception`. Если он установлен в `true` (в конкретном потоке или глобально в классе `Thread`), то неперехваченное исключение завершает *все* работающие потоки.

Попробуем выполнить самый первый пример, присвоив флагу `abort_on_exception` значение `true` в потоке `t1`:

```
t1 = Thread.new do
   puts "Hello"
   sleep 1
   raise "какое-то исключение"
end

t1.abort_on_exception = true    # Ах, какой фейерверк!

t2 = Thread.new do
   sleep 2
   puts "Привет из другого потока"
end

sleep 3
puts "Конец"
```

В данном случае печатается только один «Привет», потому что неперехваченное исключение в потоке t1 завершает как главный поток, так и t2.

Метод t1.abort_on_exception влияет на поведение только одного потока, тогда как метод класса Thread.abort_on_exception завершает все потоки, если *хотя бы в одном* из них возникло исключение.

Отметим, что эту возможность следует применять только во время разработки или отладки, потому что она эквивалентна вызову Thread.list.each(&:kill). Но по иронии судьбы метод kill как раз не является потокобезопасным. Используйте kill или abort_on_exception только в ходе отладки или для завершения потоков, относительно которых вы твердо уверены, что это безопасно. Возможно, принудительно завершаемый поток удерживает блокировку или еще не выполнил код очистки в блоке ensure, то и другое может оставить программу в ошибочном и не допускающем восстановления состоянии.

13.1.6. Группы потоков

Группа потоков – это механизм управления логически связанными потоками. По умолчанию все потоки принадлежат группе Default (это константа класса). Но если создать новую группу, то в нее можно будет помещать потоки. В любой момент времени поток может принадлежать только одной группе. Если поток помещается в группу, то он автоматически удаляется из той группы, которой принадлежал ранее.

Метод класса ThreadGroup.new создает новую группу потоков, а метод экземпляра add помещает поток в группу.

```
t1 = Thread.new("file1") { sleep(1) }
t2 = Thread.new("file2") { sleep(2) }

threads = ThreadGroup.new
threads.add t1
threads.add t2
```

Метод экземпляра list возвращает массив всех потоков, принадлежащих данной группе.

```
# Подсчитать все "живые" потоки в группе this_group
count = 0
this_group.list.each {|x| count += 1 if x.alive? }
if count < this_group.list.size
  puts "Некоторые потоки в группе this_group уже скончались."
else
  puts "Все потоки в группе this_group живы."
end
```

У объектов класса ThreadGroup имеется также метод со странным названием enclose, который, как правило, не дает помещать в группу новые потоки:

```
tg = ThreadGroup.new
tg.enclose                    # Запереть эту группу
tg.add Thread.new {sleep 1}   # Ба-бах!
```

Оговорка «как правило» необходима, потому что если новый поток запущен из потока, уже находящегося в запертой группе, то он так в этой группе и останется. У группы потоков есть метод экземпляра enclosed?, который возвращает true, если группа заперта.

В класс ThreadGroup можно добавить немало полезных методов. В примере ниже показаны методы для возобновления всех потоков, принадлежащих группе, для группового ожидания потоков (с помощью join) и для группового завершения потоков:

```
class ThreadGroup

  def wakeup
    list.each { |t| t.wakeup }
  end

  def join
    list.each { |t| t.join if t != Thread.current }
  end

  def kill
    list.each { |t| t.kill! }
  end

end
```

Следует иметь в виду, что когда поток завершается, он автоматически удаляется из группы. Так что из того, что вы поместили поток в группу, вовсе не следует, что он по-прежнему будет пребывать там через десять минут или хотя бы через десять секунд.

13.2. Синхронизация потоков

Почему необходима синхронизация? Потому что из-за «чередования» операций доступ к переменным и другим сущностям может осуществляться в порядке, который не удается установить путем чтения исходного текста отдельных потоков.

Два и более потоков, обращающихся к одной и той же переменной, могут взаимодействовать между собой непредвиденными способами, и отлаживать такую программу очень трудно.

Рассмотрим простой пример:

```
def new_value(i)
      i + 1
end

x = 0

t1 = Thread.new do
      1.upto(1000000) { x = new_value(x) }
end

t2 = Thread.new do
      1.upto(1000000) { x = new_value(x) }
end

t1.join
t2.join
puts x
```

Сначала переменная x равна 0. Затем мы запускаем два потока и каждый увеличивает ее значение на единицу миллион раз, вызывая метод new_value. В конце печатается значение x. Логика подсказывает, что должно быть напечатано 2 000 000. Однако иногда печатается только один миллион!

При выполнении этой программы под управлением JRuby результат оказался еще более неожиданным: 1143345 в одном прогоне, 1077403 во втором и 1158422 в третьем. Уж не наткнулись ли мы на чудовищную ошибку в Ruby?

Вовсе нет. Мы предполагали, что инкремент целого числа – атомарная (неделимая) операция. Но это не так. Рассмотрим последовательность выполнения приведенной выше программы. Поместим поток t1 слева, а поток t2 справа. Каждый квант времени занимает одну строчку и предполагается, что к моменту, когда был сделан этот мгновенный снимок, переменная x имела значение 123.

```
t1                                  t2
_____________________________       _____________________________

Прочитать значение x (123)

                                    Прочитать значение x (123)
Увеличить значение на 1 (124)

                                    Увеличить значение на 1 (124)
Записать 124 в x

                                    Записать 124 в x
```

Ясно, что каждый поток увеличивает на 1 то значение, которое видит. Но не менее ясно и то, что после увеличения на 1 *обоими* потоками x оказалась равно всего 124. Мы сознательно ухудшили ситуацию в этом примере, введя метод new_value и тем самым увеличив интервал времени между чтением x и записью увеличенно-

го значения. Даже крохотная задержка, связанная с единственным вызовом функции, может кардинально уменьшить число правильных операций инкремента.

Тот факт, что MRI иногда дает правильный результат, может привести невнимательного наблюдателя к выводу, что этот код потокобезопасен. Проблему маскирует побочный эффект наличия в MRI глобальной блокировки интерпретатора (GIL), из-за которой в каждый момент времени может выполняться только один поток. Но, как видите, даже GIL не помогает MRI надежно решить эту проблему синхронизации — результат все равно может оказаться неверным.

И это лишь самая простая из проблем, возникающих в связи с синхронизацией. Для решения более сложных, характеризуемых тем, что программа «почти всегда работает», приходится прилагать серьезные усилия, это предмет изучения специалистами в области теоретический информатики и математики.

13.2.1. Простая синхронизация

Простейший способ синхронизации дает метод `Thread.exclusive`. Он определяет *критическую секцию* в программе. Когда поток входит в критическую секцию программы, гарантируется, что никакой другой поток не войдет в нее, пока первый не выйдет.

Использовать метод `Thread.exclusive` просто: нужно лишь передать ему блок. В следующем примере мы переработали код предыдущего, воспользовавшись методом `Thread.exclusive` для задания критической секции, которая защищает уязвимые участки программы.

```
def new_value(i)
   i + 1
end

x = 0

t1 = Thread.new do
   1.upto(1000000) do
      Thread.exclusive { x = new_value(x) }
   end
end

t2 = Thread.new do
   1.upto(1000000) do
      Thread.exclusive { x = new_value(x) }
   end
end

t1.join
t2.join
puts x
```

И хотя мы почти ничего не изменили в коде, последовательность выполнения изменилась. Вызов `Thread.exclusive` не дает потокам наступать друг другу на пятки.

```
t1                                    t2
--------------------------------      --------------------------------

Прочитать значение x (123)
Увеличить значение на 1 (124)
Записать 124 в x
                                      Прочитать значение x (124)
                                      Увеличить значение на 1 (125)
                                      Записать 124 в x
```

На практике с методом Thread.exclusive связан ряд проблем, и самая главная – широта его воздействия. Вызывая Thread.exclusive, мы блокируем *почти все* остальные потоки, включая и ни в чем не повинные, которые и не собирались изменять наши драгоценные данные.

Слово «почти» подводит еще к одной проблеме: хотя эффект Thread.exclusive распространяет очень широко, этот метод все не является воздухонепроницаемым. При некоторых условиях другой поток может работать, несмотря на поставленную Thread.exclusive блокировку. Например, если запустить второй поток изнутри критической секции, то он *будет выполняться*.

Поэтому использование Thread.exclusive лучше ограничить лишь такими простыми ситуациями, как в примере выше. По счастью, в Ruby есть и другие, более гибкие, средства синхронизации.

13.2.2. Синхронизация доступа с помощью мьютекса

Одно из них – *мьютекс* (сокращение от «mutual exclusion» – взаимное исключение). Чтобы увидеть мьютекс в действии, переработаем предыдущий пример с увеличением счетчика.

```
require 'thread'

def new_value(i)
   i + 1
end

x = 0
mutex = Mutex.new

t1 = Thread.new do
   1.upto(1000000) do
      mutex.lock
      x = new_value(x)
      mutex.unlock
   end
end

t2 = Thread.new do
   1.upto(1000000) do
      mutex.lock
      x = new_value(x)
```

```
      mutex.unlock
   end
end

t1.join
t2.join
puts x
```

Можно сказать, что объекты класса Mutex реализуют синхронизацию с узкой областью действия: мьютекс гарантирует, что в каждый момент времени только один поток может успешно вызвать метод lock. Если два потока попытаются одновременно вызвать lock, то неудачник будет приостановлен до тех пор, пока более удачливый соперник – сумевший захватить блокировку – не вызовет unlock. Поскольку действие мьютекса распространяется только на потоки, пытающиеся захватить именно его, то мы получаем синхронизацию, не приостанавливая все потоки, кроме одного.

Помимо метода lock, в классе Mutex есть также метод try_lock. Он отличается от lock тем, что если мьютекс уже захвачен другим потоком, то он не дожидается освобождения, а сразу возвращает false:

```
require 'thread'

mutex = Mutex.new
t1 = Thread.new do
   mutex.lock
   sleep 10
end

sleep 1

t2 = Thread.new do
   if mutex.try_lock
      puts "Захватил"
   else
      puts "Не смог захватить"        # Печатается сразу же
   end
end
```

Эта возможность полезна, если поток не хочет приостанавливаться. Есть также метод synchronize, который ставит блокировку:

```
x = 0
mutex = Mutex.new

t1 = Thread.new do
   1.upto(1000000) do
      mutex.synchronize { x = new_value(x) }
   end
end
```

Наконец, существует библиотека mutex_m, где определен модуль Mutex_m, который можно подмешивать к классу (или использовать для расширения объекта).

У такого расширенного объекта будут все методы мьютекса, так что он сам может выступать в роли мьютекса.

```
require 'mutex_m'

class MyClass
  include Mutex_m

  # Теперь любой объект класса MyClass может вызывать
  # методы lock, unlock, synchronize ...
  # Внешние объекты также могут вызывать эти
  # методы для объекта MyClass.
end
```

13.2.3. Встроенные классы очередей

В библиотеке thread.rb есть два класса, которые иногда бывают полезны. Класс Queue реализует потокобезопасную очередь, доступ к обоим концам которой синхронизирован. Это означает, что разные потоки могут, ничего не опасаясь, работать с такой очередью. Класс SizedQueue отличается от предыдущего тем, что позволяет ограничить размер очереди (число элементов в ней).

Оба класса имеют практически один и тот же набор методов, поскольку SizedQueue наследует Queue. Правда, в подклассе определен еще акцессор max, позволяющий получить и установить максимальный размер очереди.

```
buff = SizedQueue.new(25)
upper1 = buff.max             # 25
# Увеличить размер очереди...
buff.max = 50
upper2 = buff.max             # 50
```

В листинге 13.1 приведено решение задачи о производителе и потребителе. Для производителя задержка (аргумент sleep) чуть больше, чем для потребителя, чтобы единицы продукции «накапливались».

Листинг 13.1. Задача о производителе и потребителе

```
require 'thread'

buffer = SizedQueue.new(2)

producer = Thread.new do
  item = 0
  loop do
    sleep(rand * 0.1)
    puts "Производитель произвел #{item}"
    buffer.enq item
    item += 1
  end
end
```

```
consumer = Thread.new do
  loop do
    sleep((rand 0.1)+0.09
    item = buffer.deq
    puts "Потребитель потребил #{item}"
    puts "  ожидает = #{buffer.num_waiting}"
  end
end

sleep 10   # Работать 10 секунд , потом завершить оба потока
```

Чтобы поместить элемент в очередь и извлечь из нее, рекомендуется применять соответственно методы enq и deq. Можно было бы для помещения в очередь пользоваться также методом push, а для извлечения – методами pop и shift, но их названия не так мнемоничны в применении к очередям.

Метод empty? проверяет, пуста ли очередь, а метод clear опустошает ее. Метод size (и его синоним length) возвращает число элементов в очереди.

```
# Предполагается, что другие потоки не мешают ...

buff = Queue.new
buff.enq "one"
buff.enq "two"
buff.enq "three"
n1 = buff.size              # 3
flag1 = buff.empty?         # false
buff.clear
n2 = buff.size              # 0
flag2 = buff.empty?         # true
```

Метод num_waiting возвращает число потоков, ожидающих доступа к очереди. Если размер очереди не ограничен, то это потоки, ожидающие возможности удалить элементы; для объекта SizedQueue включаются также потоки, ожидающие возможности добавить элементы в очередь.

Необязательный параметр non_block метода deq в классе Queue по умолчанию равен false. Если же он равен true, по при попытке извлечь элемент из пустой очереди он не блокирует поток, а возбуждает исключение ThreadError.

13.2.4. Условные переменные

Да зовите моих скрипачей, трубачей.

– Веселый король
(детский стишок, перевод Маршака)

Условная переменная – это, по существу, очередь потоков. Они используются в сочетании с мьютексами для лучшего управления синхронизацией потоков. Условная переменная всегда ассоциируется с каким-то мьютексом. Ее назначение –

освободить мьютекс *до тех пор, пока не начнет выполняться определенное условия*. Представьте себе ситуацию, когда поток захватил мьютекс, но не готов продолжать выполнение. Тогда он может заснуть под контролем условной переменной, ожидая, что будет разбужен, когда условие станет истинным.

Важно понимать, что пока поток ждет условную переменную, мьютекс свободен, поэтому другие потоки могут получить доступ к защищенному им ресурсу. А как только другой поток сигнализирует этой переменной, ожидающий поток пробуждается и пытается вновь захватить мьютекс.

Рассмотрим несколько искусственный пример в духе задачи об обедающих философах. Представьте себе, что вокруг стола сидят три скрипача, ожидающих своей очереди поиграть. Но у них есть всего две скрипки и один смычок. Понятно, что скрипач сможет играть, только если одновременно завладеет одной из скрипок и смычком.

Мы поддерживаем счетчики свободных скрипков и смысков. Когда скрипач хочет получить скрипку и смычок, он должен ждать их освобождения. В программе ниже мы защитили проверку условия мьютексом и под его защитой ждем скрипку и смычок порознь. Если скрипка или смычок заняты, поток засыпает. Он не владеет мьютексом до тех пор, пока другой поток не просигнализирует о том, что ресурс свободен. В этот момент первый поток просыпается и снова захватывает мьютекс. Код представлен в листинге 13.2.

Листинг 13.2. Три скрипача

```
require 'thread'

@music  = Mutex.new
@violin = ConditionVariable.new
@bow    = ConditionVariable.new

@violins_free = 2
@bows_free    = 1

def musician(n)
  loop do
    sleep rand(0)
    @music.synchronize do
      @violin.wait(@music) while @violins_free == 0
      @violins_free -= 1
      puts "#{n} владеет скрипкой"
      puts "скрипок #@violins_free, смычков #@bows_free"

      @bow.wait(@music) while @bows_free == 0
      @bows_free -= 1
      puts "#{n} владеет смычком"
      puts "скрипок #@violins_free, смычков #@bows_free"
    end
    sleep rand(0)
```

```
    puts "#{n}:  (...играет...)"
    sleep rand(3)
    puts "#{n}: Я закончил."

    @music.synchronize do
      @violins_free += 1
      @violin.signal if @violins_free == 1
      @bows_free += 1
      @bow.signal if @bows_free == 1
    end
  end
end

threads = []
3.times {|i| threads << Thread.new { musician(i) } }

threads.each {|t| t.join }
```

Мы полагаем, что это решение никогда не приводит к взаимоблокировке, хотя доказать этого не сумели. Но интересно отметить, что описанный алгоритм не справедливый. В наших тестах оказалось, что первый скрипач играет чаще двух остальных, а второй чаще третьего. Выяснение причин такого поведения и его исправление мы оставляем читателю в качестве интересного упражнения.

13.2.5. Другие способы синхронизации

Еще один механизм синхронизации — это *монитор*, который в Ruby реализован в библиотеке `monitor.rb`. Это более развитый по сравнению с мьютексом механизм, основное отличие состоит в том, что захваты одного и того же мьютекса не могут быть вложенными, а монитора — могут.

Как и испанская инквизиция, вложенные блокировки приходят, когда их никто не ждет. В самом деле, вряд ли кто-нибудь станет писать такой код:

```
@mutex = Mutex.new

@mutex.synchronize do
  @mutex.synchronize do
    #...
  end
end
```

Но это не значит, что вложенные блокировки вообще невозможны. Что, если обращение к `synchronize` находится в рекурсивном методе? Или метод захватывает мьютекс, а потом, не чуя за собой вины, вызывает какой-то другой метод?

```
def some_method
   @mutex = Mutex.new

   @mutex.synchronize do
      #...
      some_other_method
   end
```

```
end

def some_other_method
   @mutex.synchronize do          # Взаимоблокировка!
   #...
   end
end
```

В обоих случаях – по крайней мере, при использовании мьютекса – результатом будет взаимоблокировка с последующим не красящим программу исключением. Но поскольку мониторы допускают вложенные блокировки, следующий код работает безо всяких проблем:

```
def some_method
   @monitor = Monitor.new

   @monitor.synchronize do
      #...
      some_other_method
   end
end

def some_other_method
   @monitor.synchronize do         # Никаких проблем!!
   #...
   end
end
```

Как и мьютексы, мониторы Ruby бывают двух видов: класс `Monitor`, с которым мы только что познакомились, и модуль с педантично выбранным именем `MonitorMixin`, который позволяет использовать в качестве монитора объект любого класса:

```
class MyMonitor
   include MonitorMixin
end
```

Класс `ConditionVariable` в библиотеке `monitor.rb` дополнен по сравнению с определением в библиотеке `thread`. У него есть методы `wait_until` и `wait_while`, которые блокируют поток в ожидании выполнения условия. Кроме того, возможен таймаут при ожидании, поскольку у метода `wait` имеется параметр `timeout`, равный количеству секунд (по умолчанию `nil`).

Поскольку примеры работы с потоками у нас кончаются, то в листинге 13.3 мы предлагаем реализацию классов `Queue` и `SizedQueue` с помощью монитора. Код приводится с разрешения автора, Шуго Маэда (Shugo Maeda).

Листинг 13.3. Реализация класса `Queue` с помощью монитора

```
# Немного модифицированная версия кода, написанного Шуго Маэда

require 'monitor'

class OurQueue
```

```
  def initialize
    @que = []
    @monitor = Monitor.new
    @empty_cond = @monitor.new_cond
  end

  def enq(obj)
    @monitor.synchronize do
      @que.push(obj)
      @empty_cond.signal
    end
  end

  def deq
    @monitor.synchronize do
      while @que.empty?
        @empty_cond.wait
      end
      return @que.shift
    end
  end

   def size
      @que.size
   end
end

class OurSizedQueue < OurQueue
  attr :max

  def initialize(max)
    super()
    @max = max
    @full_cond = @monitor.new_cond
  end

  def enq(obj)
    @monitor.synchronize do
      while @que.length >= @max
        @full_cond.wait
      end
      super(obj)
    end
  end

  def deq
    @monitor.synchronize do
      obj = super
```

```
        @full_cond.signal
      end
      return obj
    end
  end

  def max=(max)
    @monitor.synchronize do
      @max = max
      @full_cond.broadcast
    end
  end
end
```

Еще один вариант синхронизации потоков (двухфазную блокировку со счетчиком) предлагает библиотека sync.rb. В ней определен модуль Sync_m, который можно применять вместе с ключевыми словами include и extend (как и Mutex_m). Этот модуль содержит методы sync_locked?, sync_shared?, sync_exclusive?, sync_lock, sync_unlock и sync_try_lock.

13.2.6. Таймаут при выполнении операций

Часто встречается ситуация, когда на выполнение операции отводится определенное максимальное время. Подобная возможность очень полезна, в частности, в сетевых приложениях, где ответ от сервера может и не прийти.

Библиотека timeout.rb предлагает решение этой проблемы на основе потоков (см. листинг 13.4). С методом timeout ассоциирован выполняемый блок. Если истечет заданное число секунд, метод возбуждает исключение Timeout::Error, которое можно перехватить с помощью rescue.

Листинг 13.4. Пример таймаута

```
require 'timeout'

flag = false
answer = nil

begin
  timeout(5) do
    puts "Хочу печенье!"
    answer = gets.chomp
    flag = true
  end
rescue Timeout::Error
  flag = false
end

if flag
  if answer == "cookie"
```

```
    puts "Спасибо! Хрум, хрум ..."
  else
    puts "Это же не печенье!"
    exit
  end
else
  puts "Эй, слишком медленно!"
  exit
end

puts "До встречи..."
```

Но у таймаутов в Ruby есть два подвоха. Во-первых, они не являются потокобезопасными. Таймаут реализован следующим образом: создать новый поток, который следит за завершением текущего, и принудительно снять текущий поток, если он не завершится в течение указанного времени. Но в разделе 13.1.5 мы уже говорили, что принудительное снятие потоков не является потокобезопасной операцией, поскольку может оставить программы в неопределенном состоянии.

Чтобы добиться безопасного эффекта таймаута, нужно написать обрабатывающий поток таким образом, чтобы он периодически проверял, не пора ли завершаться. Тогда при завершении поток сможет выполнить необходимую очистку контролируемым образом. В примере ниже программа вычисляет столько простых чисел, сколь успеет за 5 секунд, и не нуждается во внешнем потоке, который завершит вычисление принудительно:

```
require 'prime'
primes = []
generator = Prime.each
start = Time.now
while Time.now < (start + 5)
   10.times { primes << generator.next }
end

puts "Работала #{Time.now - start} секунд."
puts "Найдено простых чисел: #{primes.size}, последнее - #{primes.last}"
```

Во время тестирования эта программа напечатала «Работала 5.005569 секунд. Найдено простых чисел: 2124010, последнее – 34603729». На вашей машине результат может быть другим, но в любом случае видно, как можно следить за истечением времени, не прибегая к принудительному снятию работающего потока.

Во-вторых, нужно помнить, что исключение `Timeout::Error` не перехватывается простым предложением `rescue` без аргументов. Вызов `rescue` без аргументов перехватывает исключения класса `StandardError` и его подклассов, а `Timeout::Error` подклассом `StandardError` не является.

Если в вашей программе возможны исключения таймаута, то обязательно перехватывайте исключения обоих видов. Это относится, в частности, к любому использованию библиотеки `Net::HTTP`, в которой библиотека `Timeout` применяется, чтобы прервать обработку запросов, выполняющихся слишком долго.

13.2.7. Ожидание события

Часто один или несколько потоков следят за «внешним миром», а остальные выполняют полезную работу. Все примеры в этом разделе надуманные, но общий принцип они все же иллюстрируют.

В следующем примере прикладную задачу решают три потока. Четвертый поток каждые пять секунд просыпается, проверяет глобальную переменную $flag и, когда видит, что флаг поднят, пробуждает еще два потока. Это освобождает три рабочих потока от необходимости напрямую общаться с двумя другими и, возможно, от многочисленных попыток разбудить их.

```
$flag = false
work1 = Thread.new { job1() }
work2 = Thread.new { job2() }
work3 = Thread.new { job3() }

thread4 = Thread.new { Thread.stop; job4() }
thread5 = Thread.new { Thread.stop; job5() }

watcher = Thread.new do
  loop do
    sleep 1
    if $flag
      thread4.wakeup
      thread5.wakeup
      Thread.exit
    end
  end
end
```

Если в какой-то момент выполнения любого из методов job, переменная $flag станет равной true, то в течение одной секунды после этого потоки thread4 и thread5 гарантированно запустятся. После этого поток watcher завершается.

В следующем примере мы ждем создания файла. Каждую секунду проверяется его существование, и как только файл появится, мы запускаем новый поток. Тем временем остальные потоки занимаются своим делом. На самом деле, ниже мы наблюдаем за тремя разными файлами.

```
def process_file(filename)
   puts "обрабатывается файл #{filename}"
end

def waitfor(filename)
  loop do
    if File.exist? filename
      file_processor = Thread.new { process_file(filename) }
      Thread.exit
    else
      sleep 1
```

```
    end
  end
end

waiter1 = Thread.new { waitfor("Godot") }
waiter2 = Thread.new { waitfor("Guffman") }
headwaiter = Thread.new { waitfor("head") }

# Основной поток занимается другими делами...
```

Есть много других ситуаций, когда поток должен ожидать внешнего события, например, в сетевых приложениях так бывает, когда сервер на другом конце соединения работает медленно или ненадежно.

13.2.8. Параллельный поиск в коллекции

С помощью потоков нетрудно реализовать одновременную работу над альтернативными решениями. Для иллюстрации напишем программу, которая ищет максимальное число в нескольких массивах с ограничением по времени. Метод threaded_max принимает в качестве аргументов лимит времени, а также массив числовых массивов, в которых производится поиск. Он возвращает наибольшее число, которое сумеет найти за отведенное время.

Чтобы как можно больше успеть, наш код не просматривает все массивы поочередно, а запускает по одному потоку для каждого массива:

```
require 'thread'

def threaded_max(interval, collections)
   threads = []

   collections.each do |col|
      threads << Thread.new do
         me = Thread.current
         me[:result] = col.first
         col.each do |n|
            me[:result] = n if n > me[:result]
         end
      end
   end

   sleep(interval)

   threads.each {|t| t.kill }
   results = threads.map {|t| t[:result]}

   results.compact.max          # Может быть nil
end

collections = [
```

```
    [ 1, 25, 3, 7, 42, 64, 55 ],
    [ 3, 77, 1, 2, 3, 5, 7, 9, 11, 13, 102, 67, 2, 1],
    [ 3, 33, 7, 44, 77, 92, 10, 11]]

biggest = threaded_max(0.5, collections)
```

Сделаем несколько замечаний. Во-первых, для простоты мы предположили, что все массивы не пусты.

Во-вторых, каждый поток сообщает о результате своей работы в поточно-локальной переменной `:result`. Эта переменная обновляется на каждом шаге, потому что главный поток в методе `threaded_max` подождет столько времени, сколько ему сказали, а затем убьет все рабочие потоки и вернет наибольшее число, найденное к этому моменту. Наконец, теоретически `threaded_max` может вернуть `nil`: какое бы длительное время мы не задали, есть шанс, что потоки даже не приступят к работе к тому моменту, когда у главного потока истощится терпение (маловероятно, но все же возможно).

Действительно ли использование нескольких потоков ускоряет работы? Трудно сказать. Ответ зависит от операционной системы, а также от количества просматриваемых массивов.

13.2.9. Параллельное рекурсивное удаление

Забавы ради, напишем программу, которая будет удалять дерево каталогов, причем сделаем ее параллельной. Идея в том, что обнаружив очередной подкаталог, мы запускаем новый поток, который будет обходить его и удалять содержимое.

Созданные в ходе работы программы потоки хранятся в массиве `threads`. Поскольку это локальная переменная, у каждого потока будет собственная копия массива. Раз к ней может обращаться всего один поток, то синхронизировать доступ не нужно.

Отметим также, что в блок потока передается полное имя файла `fullname`, чтобы не нужно было беспокоиться по поводу того, что поток обращается к переменной, которую кто-то еще изменяет. Поток делает для себя локальную копию `fn` этой переменной.

После того как обход всего каталога завершен, мы должны дождаться завершения всех созданных потоков и только потом удалять сам обработанный каталог:

```
def delete_all(dir)
  threads = []
  Dir.foreach(dir) do |e|
    next if [".",".."].include? e          # Пропустить . и ..
    fullname = dir + "/" + e
    if FileTest.directory?(fullname)
      threads << Thread.new(fullname) {|fn| delete_all(fn) }
    else
      File.delete(fullname)
      puts "удален: #{fullname}"
    end
  end
```

```
  threads.each { |t| t.join }
  puts "удаляется каталог #{dir}"
  Dir.delete(dir)
end

delete_all("/tmp/stuff")
```

Будет ли работать такая программа быстрее, чем ее вариант без потоков? В наших тестах получалось по-разному. Возможно, это зависит от операционной системы и структуры конкретного каталога – глубины, количества файлов и т.д.

13.3. Волокна и кооперативная многозадачность

Помимо полноценных потоков операционной системы, Ruby предоставляет еще *волокна*, которые можно описать как урезанные потоки или как блоки со сверхъестественными способностями.

С волокнами не связан поток ОС, но они могут содержать блок кода, который запоминает свое состояние, может быть приостановлен и возобновлен и способен отдавать результаты. Для иллюстрации начнем с создания волокна методом `Fiber.new`:

```
fiber = Fiber.new do
   x = 2
   Fiber.yield x
   x = x * 2
   Fiber.yield x
   x * 2
end
```

Метод `Fiber.new` не приводит к выполнению какого-либо кода в блоке волокна. Чтобы шестеренки закрутились, нужно вызвать метод `resume`. Вызов `resume` заставляет код в блоке выполняться до тех пор, пока не будет достигнут конец блока или не вызван метод `Fiber.yield`. Поскольку в нашем волокне вызов `Fiber.yield` встречается во второй строчке блока, то первый вызов `resume` доработает до второй строчки. Метод `resume` возвращает значение, переданное методу `Fiber.yield`, в данном случае 2:

```
answer1 = fiber.resume      # answer1 равно 2
```

До сих пор все это сильно напоминает лямбда-выражения, но дальше станет интереснее. Если вызвать `resume` второй или третий раз, то волокно продолжит работу с того места, где остановилось:

```
answer2 = fiber.resume      # должно быть 4
answer3 = fiber.resume      # должно быть 8
```

По достижении конца блока волокно поступит так, как принято в таком случае, – вернет последнее вычисленное в блоке значение.

С учетом сказанного волокно можно рассматривать как блок с возможностью продолжения: встретив вызов `Fiber.yield`, волокно запоминает, где находится,

и дает возможность продолжить работу с этого места. Продолжать можно до тех пор, пока не будет достигнут конец блока; если вызвать resume после этого, возникнет исключение FiberError.

Как мы уже сказали, волокно можно считать блоком с возможностью продолжения. Но можно взглянуть на него и как на поток, планируемый вручную. Волокно не работает более-менее непрерывно в фоновом режиме, а исполняется в потоке, вызвавшем resume, пока не завершится или не встретит очередной вызов Fiber.yield. Другими словами, если потоки реализуют вытесняющую многозадачность, то волокна – кооперативную.

Помимо основных методов Fiber.new и yield, можно получить дополнительный бонус, загрузив библиотеку 'fiber'. В этом случае становится доступным метод Fiber.current, который возвращает текущий исполняемый экземпляр волокна, а также метод экземпляра alive?, который сообщает, живо ли еще волокно. Наконец, библиотека fiber наделяет каждый объект волокна методом экземпляра transfer, который позволяет передавать управление от одного волокна другому (чем достигается по-настоящему кооперативное поведение).

Чтобы закрыть эту тему, перепишем пример поиска в нескольких массивах с использованием волокон:

```
require 'fiber'

class MaxFiber < Fiber
   attr_accessor :result
end

def max_by_fiber(interval, collections)
   fibers = []
   collections.each_with_index do |numbers|
      fibers << MaxFiber.new do
         me = Fiber.current
         me.result = numbers[0]
         numbers.each_with_index do |n, i|
            me.result = n if n > me.result
            Fiber.yield me.result if (i+1) % 3 == 0
         end
         me.result
      end
   end

   start_time = Time.now
   while fibers.any? &:alive?
      break if Time.now - start_time > interval
      fibers.each {|f| puts f.resume if f.alive? }
   end

   values = fibers.map &:result
```

```
    values.compact.max || 0
end

collections = [
      [ 1, 25, 3, 7, 42, 64, 55 ],
      [ 3, 77],
      [ 3, 33, 7, 44, 77, 102, 92, 10, 11],
      [1, 2, 3, 4, 5, 6, 7, 8, 9, 77, 2, 3 ]]

biggest = max_by_fiber(0.5, collections)
```

Поскольку мы работаем с волокнами, а не с потоками, истинного параллелизма здесь нет. Но волокна дают элегантный способ циклически перебирать массивы, запоминая, в каком месте каждого массива мы остановились.

Возможность волокон приостанавливать выполнение наиболее полезна, когда необходимо сделать паузу перед вычислением очередного элемента. Вероятно, вы обратили внимание на сходство с работой объектов `Enumerator`, и это неудивительно, потому что перечислители на самом деле реализованы с помощью волокон. Методы `next`, `take` и все остальные просто вызывают метод `resume` ассоциированного с перечислителем волокна столько раз, сколько необходимо для порождения запрошенных результатов.

13.4. Заключение

Как было сказано в этой главе, поддержка многопоточности в Ruby опирается на потоки ОС. Однако из-за наличия GIL в интерпретаторе MRI в каждый момент времени может работать только один поток. В интерпретаторах JRuby и Rubinius такого ограничения нет.

Из этой главы вы могли составить представление о том, как трудно писать многопоточные программы и как имеющиеся в Ruby примитивы синхронизации – мьютексы, мониторы, условные переменные и очереди – помогают координировать работу потоков. Наконец, мы бегло обсудили волокна, которые в Ruby являются «почти, но не совсем потоками».

В главе 14 мы перейдем от обсуждения техники программирования к решению конкретных задач. Мы обсудим, как Ruby применяется для написания рутинных скриптов и решения задач системного администрирования.

Глава 14. Скрипты и системное администрирование

И сказал главный программист:
«Даже если программа состоит всего из трех строчек,
когда-то ее придется сопровождать».

– Джеффри Джеймс «Тао программирования»

Программистам часто приходится писать небольшие скрипты для запуска внешних программ и работы с операционной системой на достаточно высоком уровне. Особенно это относится к ОС UNIX, где для повседневной работы составляются многочисленные скрипты на языке интерпретатора команд (shell).

Программисту, поднаторевшему в Ruby, этот язык покажется очень удобным для написания небольших полезных скриптов. На самом деле, когда Мац задумывал Ruby, взяв за образец Perl, одной из его целей было упростить написание небольших скриптов для решения полезных задач.

Нередко для этой цели можно воспользоваться каким-то более традиционным языком. Преимущества Ruby в его универсальности, богатстве функций и объектной ориентированности. Предполагая, что найдутся люди, желающие использовать Ruby для взаимодействия с операционной системой на таком уровне, мы продемонстрируем несколько полезных приемов.

Многое из того, что *можно было бы* включить в эту главу, вошло в другие главы. Обратите, в частности, внимание на главу 10, в которой рассматривается ввод-вывод и атрибуты файлов; эта информация часто бывает полезна при написании скриптов.

14.1. Запуск внешних программ

Никакой язык не может использоваться в качестве «клея», если он не позволяет запускать внешние программы. В Ruby для этого есть несколько способов.

Не могу не сказать о том, что перед тем как запускать внешнюю программу, неплохо понять, что она делает. Я имею в виду вирусы и другие потенциально разрушительные программы. Не запускайте произвольную командную строку, особенно поступившую из внешнего источника. Это касается не только приложений, ориентированных на веб.

14.1.1. Методы system и exec

Метод system (из модуля Kernel) эквивалентен одноименной функции из библиотеки языка C. Он выполняет указанную команду в отдельной оболочке.

```
system("date")
# Вывод направляется, как обычно, на stdout...
```

Дополнительные параметры, если они заданы, рассматриваются как список аргументов; в большинстве случаев аргументы можно задавать и в командной строке, эффект будет тот же. Разница лишь в том, что алгоритм расширения имени файла применяется только к первой из переданных строк.

```
system("rm", "/tmp/file1")
system("rm /tmp/file2")
# Оба варианта годятся.

# А тут есть различие...
system("echo *")     # Печатается список всех файлов
system("echo","*")   # Печатается звездочка (расширение
                     # имени файла не производится)

# Более сложные командные строки тоже работают.
system("ls -l | head -n 1")
```

Отмечу, что если требуется запомнить выведенную программой информацию (например, в переменной), то system – не лучший способ. См. следующий раздел.

Упомяну еще метод exec. Он ведет себя аналогично system с тем отличием, что новый процесс замещает текущий. Поэтому код, следующий за exec, исполняться не будет.

```
puts "Содержимое каталога:"
exec("ls", "-l")

puts "Эта строка никогда не исполняется!"
```

14.1.2. Запоминание вывода программы

Простейший способ запомнить информацию, выведенную программой, – заключить команду в *обратные кавычки*, например:

```
listing = `ls -l`  # Одна строка будет содержать несколько строчек (lines)
now = `date`       # "Mon Mar 12 16:50:11 CST 2001"
```

Обобщенный ограничитель %x вызывает оператор обратных кавычек (который в действительности является методом модуля Kernel). Работает он точно так же:

```
listing = %x(ls -l)
now = %x(date)
```

Применение %x бывает полезно, когда подлежащая исполнению строка содержит такие символы, как одиночные и двойные кавычки.

Поскольку обратные кавычки – это на самом деле метод (в некотором смысле), то его можно переопределить. Изменим его так, чтобы он возвращал не одну стро-

ку, а массив строк (конечно, при этом мы сохраним синоним старого метода, чтобы его можно было вызвать).

```
alias old_execute `

def `(cmd)
  out = old_execute(cmd)  # Вызвать исходный метод обратной кавычки
  out.split("\n")         # Вернуть массив строк!
end

entries = `ls -l /tmp`
num = entries.size                    # 95

first3lines = %x(ls -l | head -n 3)
how_many = first3lines.size           # 3
```

Как видите, при таком определении изменяется также поведение ограничителя `%x`.

В следующем примере мы добавили в конец команды конструкцию интерпретатора команд, которая перенаправляет стандартный вывод для ошибок в стандартный вывод:

```
alias old_execute `

def `(cmd)
  old_execute(cmd + " 2>&1")
end

entries = `ls -l /tmp/foobar`
# "/tmp/foobar: No such file or directory\n"
```

Есть, конечно, и много других способов изменить стандартное поведение обратных кавычек.

14.1.3. Манипулирование процессами

В этом разделе мы обсудим манипулирование процессами, хотя создание нового процесса необязательно связано с запуском внешней программы. Основной способ создания нового процесса – это метод `fork`, название которого в соответствии с традицией UNIX подразумевает разветвление пути исполнения, напоминая развилку на дороге. (Отметим, однако, что Ruby не поддерживает метод `fork` на платформе Windows.)

Метод `fork`, находящийся в модуле `Kernel` (а также в модуле `Process`), не следует путать с одноименным методом экземпляра в классе `Thread`.

Существует два способа вызвать метод `fork`. Первый похож на то, как это обычно делается в UNIX, – вызвать и проверить возвращенное значение. Если оно равно `nil`, значит, мы находимся в дочернем процессе, в противном случае – в родительском. Родительскому процессу возвращается идентификатор дочернего процесса (*pid*).

```
pid = fork
if (pid == nil)
  puts "Ага, я, должно быть, потомок."
  puts "Так и буду себя вести."
else
  puts "Я родитель."
  puts "Пора отказаться от детских штучек."
end
```

В этом не слишком реалистичном примере выводимые строки могут чередоваться, а может случиться и так, что строки, выведенные родителем, появятся раньше. Но сейчас это несущественно.

Следует также отметить, что процесс-потомок может пережить своего родителя. Для потоков в Ruby это не так, но системные процессы – совсем другое дело.

Во втором варианте вызова метод fork принимает блок. Заключенный в блок код выполняется в контексте дочернего процесса. Так, предыдущий вариант можно было бы переписать следующим образом:

```
fork do
  puts "Ага, я, должно быть, потомок."
  puts "Так и буду себя вести."
end

puts "Я родитель."
puts "Пора отказаться от детских штучек."
```

Конечно, pid по-прежнему возвращается, мы просто не показали его.

Чтобы дождаться завершения процесса, мы можем вызвать метод wait из модуля Process. Он ждет завершения любого потомка и возвращает его идентификатор. Метод wait2 ведет себя аналогично, только возвращает массив, содержащий pid и сдвинутый влево код завершения.

```
pid1 = fork { sleep 5; exit 3 }
pid2 = fork { sleep 2; exit 3 }

Process.wait    # Возвращает pid2
Process.wait2   # Возвращает [pid1,768]
```

Pid, конечно, все равно возвращается. Просто мы это не показали.

Чтобы дождаться завершения процесса, применяется метод wait из модуля Process. Он ждет завершения любого потомка и возвращает его идентификатор. Метод wait2 ведет себя аналогично, но возвращает массив, содержащий два значения: pid и объект Process::Status, в котором хранится pid и код выхода:

```
pid1 = fork { sleep 2; exit 3 }
pid2 = fork { sleep 1; exit 3 }

pid2_again = Process.wait       # Возвращается pid2
pid1_and_status = Process.wait2 # Возвращается [pid1,
                                #      #<Process::Status exit 3>]
```

Чтобы дождаться завершения конкретного потомка, пользуйтесь методами waitpid и waitpid2 соответственно.

```
pid3 = fork { sleep 2; exit 3 }
pid4 = fork { sleep 1; exit 3 }

sleep 3                  # Дать потомкам время завершиться

pid4_again = Process.waitpid(pid4, Process::WNOHANG)
pid3_array = Process.waitpid2(pid3, Process::WNOHANG)

# pid3_array равен [pid3, #<Process::Status exit 3>]
```

Если второй параметр не задан, то вызов может блокировать программу (если такого потомка не существует). Второй параметр можно с помощью ИЛИ объединить с флагом Process::WUNTRACED, чтобы перехватывать остановленные процессы. Этот параметр системно-зависим, поэкспериментируйте.

Методу exit можно передать true, false или целое число. В соответствии с принятым в UNIX соглашением, код состояния 0 означает успешное завершение, а любой другой – ошибку. Поэтому при передаче true процесс завершится с кодом состояния 0, а при передаче false – с кодом состояния 1. Целое число интерпретируется как сам код состояния.

Метод exit! немедленно завершает процесс (не вызывая зарегистрированных обработчиков). Если задан целочисленный аргумент, то он возвращается в качестве кода завершения, по умолчанию подразумевается значение 1 (не 0).

```
pid1 = fork { exit! }      # Вернуть код завершения -1
pid2 = fork { exit! 0 }    # Вернуть код завершения 0
```

Методы pid и ppid возвращают соответственно идентификатор текущего и родительского процессов.

```
proc1 = Process.pid
fork do
  if Process.ppid == proc1
    puts "proc1 - мой родитель"  # Печатается это сообщение
  else
    puts "Что происходит?"
  end
end
```

Метод kill служит для отправки процессу сигнала, как это понимается в UNIX. Первый параметр может быть целым числом, именем POSIX-сигнала с префиксом SIG или именем сигнала без префикса. Второй параметр – идентификатор процесса-получателя; если он равен нулю, подразумевается текущий процесс.

```
Process.kill(1,pid1)          # Послать сигнал 1 процессу pid1
Process.kill("HUP",pid2)      # Послать SIGHUP процессу pid2
Process.kill("SIGHUP",pid2)   # Послать SIGHUP процессу pid3
Process.kill("SIGHUP",0)      # Послать SIGHUP самому себе
```

Для обработки сигналов применяется метод Kernel.trap. Обычно он принимает номер или имя сигнала и подлежащий выполнению блок.

```
trap(1) do
   puts "ОЙ!"
   puts "Перехвачен сигнал 1"
end

Process.kill(1,0)  # Послать самому себе
```

О применениях метода trap в более сложных ситуациях читайте в документации по Ruby и UNIX.

В модуле Process есть также методы для опроса и установки таких атрибутов процесса, как идентификатор пользователя, действующий идентификатор пользователя, приоритет и т.д. Дополнительную информацию см. в справочном руководстве по Ruby.

14.1.4. Стандартный ввод и вывод

В главе 10 мы видели, как работают методы IO.popen и IO.pipe, но существует еще небольшая библиотека, которая иногда бывает удобна.

В библиотеке Open3.rb есть метод popen3, который возвращает массив из трех объектов IO. Они соответствуют стандартному вводу, стандартному выводу и стандартному выводу для ошибок того процесса, который был запущен методом popen3. Вот пример:

```
require "open3"

filenames = %w[ file1 file2 this that another one_more ]
output, errout = [], []

Open3.popen3("xargs", "ls", "-l") do |inp, out, err|
      filenames.each { |f| inp.puts f }    # Писать в stdin процесса
      inp.close                            # Закрывать обязательно!

      output = out.readlines               # Читать из stdout
      errout = err.readlines               # Читать также из stderr
end

puts "Послано #{filenames.size} строк входных данных."
puts "Получено #{output.size} строк из stdout"
puts "и #{errout.size} строк из stderr."
```

В этом искусственном примере мы выполняем команду ls -l для каждого из заданных имен файлов и по отдельности перехватываем стандартный вывод и стандартный вывод для ошибок. Отметим, что вызов close необходим, чтобы порожденный процесс увидел конец файла. Также отметим, что в библиотеке Open3 используется метод fork, не реализованный на платформе Windows; для этой платформы придется пользоваться библиотекой win32-open3 (ее написали и поддерживают Дэниэль Бергер (Daniel Berger) и Парк Хисоб (Park Heesob)).

См. также раздел 14.3.

14.2. Флаги и аргументы в командной строке

Слухи о кончине командной строки сильно преувеличены. Хоть мы и живем в век графических интерфейсов, ежедневно тысячи и тысячи программистов по тем или иным причинам обращаются к командным утилитам.

Мы уже говорили, что корнями своими Ruby уходит в UNIX. Но даже в Windows существует понятие командной строки, и, честно говоря, мы не думаем, что в обозримом будущем она исчезнет.

На этом уровне для управления работой программы во время ее запуска применяются аргументы и флаги. О них мы и поговорим ниже.

14.2.1. Константа ARGV

Глобальная константа ARGV представляет список аргументов, переданных в командной строке. По сути дела, это массив.

```
n = ARGV.size
argstr = '"' + ARGV*"," + '"'
puts "Мне было передано аргументов: #{n}..."
puts "Вот они: #{argstr}"
puts "Заметьте, что ARGV[0] = #{ARGV[0]}"
```

Если запустить эту программу с аргументами red green blue, то она напечатает:

```
Мне было передано аргументов: 3...
Вот они: "red,green,blue"
Заметьте, что ARGV[0] = red
```

В некоторых языках передается также количество аргументов, но в Ruby это ни к чему, так как эта информация – часть массива.

Привычных к старым соглашениям программистов может смутить также тот факт, что нулевой элемент массива – это настоящий аргумент (а не, скажем, имя скрипта). Нумерация аргументов начинается с нуля, а не с единицы, как в языке C и в различных языках оболочки.

14.2.2. Константа ARGF

Глобальная константа ARGF представляет псевдофайл, получающийся в результате конкатенации всех имен файлов, заданных в командной строке. Во многих отношениях она ведет себя так же, как объект IO.

Когда в программе встречается «голый» метод ввода (без указания вызывающего объекта), обычно имеется в виду метод, подмешанный из модуля Kernel (например, gets и readlines). Если в командной строке не задано ни одного файла, то по умолчанию источником ввода является объект STDIN. Но если файлы заданы, то данные читаются из них. Понятно, что конец файла достигается в конце последнего из указанных файлов.

Если хотите, можете обращаться к ARGF явно:

```
# Скопировать все файлы на stdout
puts ARGF.readlines
```

Быть может, вопреки ожиданиям, признак конца файла устанавливается после каждого файла. Так, предыдущий код выведет все файлы, а следующий – только первый файл:

```
puts ARGF.gets until ARGF.eof?
```

Является ли это ошибкой или нет, предоставим судить вам. Впрочем, сюрпризы могут быть и приятными. Входные данные – это не просто поток байтов, мы можем применять к ARGF операции seek и rewind, как если бы это был «настоящий файл».

С константой ARGF ассоциирован метод file, он возвращает объект IO, соответствующий файлу, обрабатываемому в данный момент. Естественно, возвращаемое значение изменяется по мере перехода от одного файла к другому.

А если мы не хотим интерпретировать имена аргументов в командной строке как имена файлов? Тогда не надо обращаться к методам ввода без указания вызывающего объекта. Если вы хотите читать из стандартного ввода, укажите в качестве такого объекта STDIN, и все будет работать правильно.

14.2.3. Разбор флагов в командной строке

История библиотек для разбора командной строки в Ruby имеет давние традиции, сегодня для этой цели используется библиотека OptionParser. По существу, это предметно-ориентированный язык (DSL), описывающий, как следует разбирать аргументы программы.

Пожалуй, лучше всего объяснить это на примере. Пусть имеется программа с такими флагами: -h, или -help для печати справки, -f, или -file для задания имени файла и -l, или -lines для прекращения печати после вывода заданного числа строк (по умолчанию 100). Начать можно было бы так:

```
require 'optparse'

args = {lines: 100}

OptionParser.new do |opts|
   opts.banner = "Порядок вызова: tool [options] COMMAND"

   opts.on("-f", "--file FILE") do |file|
      args[:file] = file
   end

   opts.on("-l", "--lines [LINES]", Integer,
      "Сколько строк выводить (по умолчанию 100)"
   ) do |lines|
      args[:lines] = lines
   end

   opts.on_tail("-h", "--help", "Показать эту справку") do
      puts opts
      exit
   end
```

```
end.parse!

p args
p ARGV.first
```

Если сохранить этот код в файле `tool.rb` и выполнить его, то будет напечатано то, что и следовало ожидать:

```
$ ruby tool.rb -h
Порядок вызова: tool [options] COMMAND
        -f, -file FILE
        -l, -lines [LINES]   Сколько строк выводить (по умолчанию 100)
        -h, -help            Показать эту справку

$ ruby tool.rb -file book.txt
{:lines=>100, :file=>"book.txt"}
[]

$ ruby tool.rb -f book.txt -lines 10 print
{:lines=>10, :file=>"book.txt"}
["print"]
```

Как видно из этого примера, использование `OptionParser` сводится главным образом к созданию объекта, и большая часть работы происходит в методе `on`. Идея в том, что каждый вызов `on` описывает один распознаваемый анализатором флаг.

Методу `on` нужно сообщить как минимум две вещи. Во-первых, задается имя флага – короткое (например, `-f`) или длинное (например, `-file`). Если во второй строке имя аргумента заключено в квадратные скобки, то `OptionParser` понимает, что аргумент необязателен и может быть опущен.

На примере параметра `lines` мы видим, что можно задать тип и описание флага. Тогда его строковое представление, заданное в командной строке, будет преобразовано в указанный тип, а описание напечатано в справке.

Во-вторых, необходимо задать блок кода, который анализатор выполнит, распознав соответствующий флаг. В этом блоке можно сделать что-то непосредственно, как в случае флага `-h`, а можно просто сохранить данные на будущее, как в случае флагов `-file` и `-lines`.

В конечном итоге вызывается метод `parse!`, и анализатор разбирает содержимое `ARGV`, удаляет из него аргументы, которые распознал как флаги, и выполняет для них блоки. После этого в хэше `args` будут находиться переданные программе флаги, а в массиве `ARGV` останутся только нераспознанные аргументы.

Есть немало gem-пакетов, предлагающих другие подходы к разбору командной строки. Можете взять `highline`, `slop`, `cocaine`, `thor` или любую другую библиотеку, отвечающую вашим целям.

14.3. Библиотека Shell

Не всегда Ruby удобен в качестве скриптового языка. Например, в языке bash для запуска внешней программы достаточно просто указать ее имя, безо всякого дополнительного синтаксиса.

Оборотной стороной мощи и гибкости Ruby является более сложный синтаксис. Кроме того, функциональность разнесена по различным классам, модулям и библиотекам.

Это стало основанием для создания библиотеки Shell, которая упрощает, к примеру, организацию конвейеров команд и перенаправление вывода в файл. Кроме того, она сводит воедино функциональность из разных источников, скрывая ее за интерфейсом объекта Shell. (На платформе Windows эта библиотека работает не во всех случаях.)

14.3.1. Использование библиотеки Shell для перенаправления ввода-вывода

В классе Shell для создания объектов есть два метода: new и cd. Первый создает объект, ассоциированный с текущим каталогом, второй – объект, для которого рабочим будет указанный каталог.

```
require "shell"

sh1 = Shell.new             # Работать в текущем каталоге
sh2 = Shell.cd("/tmp/hal")  # Работать в каталоге /tmp/hal
```

В библиотеке Shell определено несколько встроенных команд (например, echo, cat и tee) в виде методов. Они всегда возвращают объекты класса Filter (как и определяемые пользователем команды, с которыми мы вскоре познакомимся).

Класс Filter понимает, что такое перенаправление ввода-вывода. В нем определены методы (или операторы) <, > и |, которые ведут себя примерно так, как мы ожидаем по многолетнему опыту написания shell-скриптов.

Если методу перенаправления передать в качестве параметра строку, то она будет считаться именем файла. Если же параметром является объект IO, то он используется для операций ввода-вывода. Примеры:

```
sh = Shell.new

# Вывести файл motd на stdout
sh.cat("/etc/motd") > STDOUT

# Напечатать его еще раз
(sh.cat < "/etc/motd") > STDOUT
(sh.echo "Это тест") > "myfile.txt"

# Добавить строку в конец файла /etc/motd
sh.echo("Hello, world!") >> "/etc/motd"

# Вывести два файла на stdout и продублировать (tee) вывод в третий файл
(sh.cat "file1" "file2") | (tee "file3") > STDOUT
```

Отметим, что у оператора > высокий приоритет. Скобки, которые вы видите в этом примере, в большинстве случаев обязательны. Вот два примера правильного использования и один неправильный:

```
# Интерпретатор Ruby понимает такую конструкцию...
sh.cat("myfile.txt") > STDOUT

# ...и такую тоже.
(sh.cat "myfile.txt") > STDOUT

# TypeError! (ошибка связана с приоритетами)
sh.cat "myfile.txt" > STDOUT
```

Отметим еще, что можно «инсталлировать» системные команды по своему выбору. Для этого служит метод `def_system_command`. Ниже определяются два метода: `ls` и `ll`, которые выводят список файлов в текущем каталоге (в коротком и длинном формате).

```
# Имя метода совпадает с именем команды...
# необходим только один параметр
Shell.def_system_command "ls"

# А здесь должно быть два параметра
Shell.def_system_command "ll", "ls -l"

sh = Shell.new
sh.ls > STDOUT   # Короткий формат
sh.ll > STDOUT   # Длинный формат
```

Вы, наверное, обратили внимание на то, что в большинстве случаев мы явно отправляем вывод объекту `STDOUT`. Связано это с тем, что объект `Shell` автоматически вывод команд никуда не направляет. Он просто ассоциирует его с объектом `Filter`, который уже может быть связан с файлом или с объектом `IO`.

14.3.2. Дополнительные замечания по поводу библиотеки Shell

Метод `transact` исполняет блок в контексте объекта `shell`. Таким образом, допустима следующая сокращенная запись:

```
sh = Shell.new
sh.transact do
  echo("Строка данных") > "somefile.txt"
  cat("somefile.txt","otherfile.txt") > "thirdfile"
  cat("thirdfile") | tee("file4") > STDOUT
end
```

Итератор `foreach` принимает в качестве параметра файл или каталог. Если это файл, то он перебирает все его строки, а если каталог – все имена файлов в нем.

```
sh = Shell.new

# Напечатать все строки файла  /tmp/foo
sh.foreach("/tmp/foo") {|l| puts l }

# Вывести список файлов в каталоге /tmp
sh.foreach("/tmp") {|f| puts f }
```

Метод pushdir запоминает текущий каталог, а метод popdir делает последний запомненный каталог текущим. У них есть синонимы pushd и popd. Метод pwd возвращает текущий рабочий каталог, его синонимы – getwd, cwd и dir.

```
sh = Shell.cd "/home"

puts sh.pwd      # /home
sh.pushd "/tmp"
puts sh.pwd      # /tmp

sh.popd
puts sh.pwd      # /home
```

Для удобства в класс Shell импортируются методы из различных источников, в том числе из класса File, модуля FileTest и библиотеки ftools.rb. Это избавляет от необходимости выполнять require, include, создавать объекты, квалифицировать вызовы методов и т.д.

```
sh = Shell.new
flag1 = sh.exist? "myfile"          # Проверить существование файла
sh.delete "somefile"                # Удалить файл
sh.move "/tmp/foo", "/tmp/bar"      # Переместить файл
```

У библиотеки Shell есть и другие возможности, которые мы здесь не рассматриваем. Дополнительную информацию ищите в документации.

14.4. Переменные окружения

Иногда необходимо обращаться к переменным окружения, которые являются связующим звеном между программой и внешним миром. Переменные окружения – это просто метки, связанные с некоторым текстом (обычно небольшим), в них хранятся, к примеру, пути к файлам, имена пользователей и т.п.

Переменные окружения широко применяются в ОС UNIX. Система Windows (а еще раньше MS-DOS) позаимствовала эту идею у UNIX, поэтому приведенные ниже примеры будут работать на обеих платформах.

14.4.1. Чтение и установка переменных окружения

Глобальная константа ENV – это хэш, с помощью которого можно читать и изменять переменные окружения. В примере ниже мы читаем значение переменной PATH (в Windows вместо двоеточия нужно употреблять точку с запятой):

```
mypath = ENV["PATH"]
# А теперь получим массив...
dirs = mypath.split(":")
```

А вот пример установки переменной. Новый процесс мы создали, чтобы проиллюстрировать две вещи. Во-первых, дочерний процесс наследует переменные окружения от своего родителя. Во-вторых, значение переменной окружения, установленное в дочернем процессе, родителю не видно.

```
ENV["alpha"] = "123"
ENV["beta"]  = "456"
puts "Родитель: alpha = #{ENV['alpha']}"
puts "Родитель: beta  = #{ENV['beta']}"

fork do   # Код потомка...
  x = ENV["alpha"]
  ENV["beta"] = "789"
  y = ENV["beta"]
  puts " Потомок: alpha = #{x}"
  puts " Потомок: beta  = #{y}"
end

Process.wait
a = ENV["alpha"]
b = ENV["beta"]
puts "Родитель: alpha = #{a}"
puts "Родитель: beta  = #{b}"
```

Программа выводит следующие строки:

```
Родитель: alpha = 123
Родитель: beta  = 456
 Потомок: alpha = 123
 Потомок: beta  = 789
Родитель: alpha = 123
Родитель: beta  = 456
```

Это следствие того факта, что родитель ничего не знает о переменных окружения своих потомков. Поскольку программа на Ruby обычно исполняется в подоболочке, то после ее завершения все сделанные изменения переменных окружения *не* будут видны в текущей оболочке.

14.4.2. Хранение переменных окружения в виде массива или хэша

Важно понимать, что объект `ENV` – не настоящий хэш, а лишь выглядит как таковой. Например, мы не можем вызвать для него метод `invert`; будет возбуждено исключение `NameError`, поскольку такого метода не существует. Причина такой реализации в том, что существует тесная связь между объектом `ENV` и операционной системой; любое изменение хранящихся в нем значений отражается на состоянии ОС, а такое поведение с помощью простого хэша не смоделируешь.

Однако имеется метод `to_hash`, который вернет настоящий хэш, отражающий текущее состояние:

```
envhash = ENV.to_hash
val2var = envhash.invert
```

Получив такой хэш, мы можем преобразовать его к любому другому виду (например, в массив):

```
envarr = ENV.to_hash.to_a
```

Обратное присваивание объекту `ENV` недопустимо, но при необходимости можно пойти обходным путем:

```
envhash = ENV.to_hash
# Выполняем произвольные операции... и записываем обратно в ENV.
envhash.each {|k,v| ENV[k] = v }
```

14.5. Работа с файлами, каталогами и деревьями

При выполнении рутинных задач приходится много работать с файлами и каталогами, в том числе с целыми иерархиями каталогов. Немало материала на эту тему вошло в главу 4 «Интернационализация в Ruby», но кое-какие важные моменты мы хотим осветить здесь.

Поскольку ввод-вывод – вещь системно-зависимая, то для различных систем приходится применять разные приемы. Если сомневаетесь, экспериментируйте.

14.5.1. Несколько слов о текстовых фильтрах

Многие инструменты, которыми мы постоянно пользуемся (как поставляемые производителем, так и разрабатываемые собственными силами), – просто текстовые *фильтры*. Иными словами, они принимают на входе текст, каким-то образом преобразуют его и выводят. Классическими примерами текстовых фильтров в UNIX служат, в частности, программы `sort` и `uniq`.

Иногда файл настолько мал, что целиком помещается в памяти. В этом случае возможны такие виды обработки, которые по-другому было бы сложно реализовать.

```
lines = File.open(filename){|f| f.readlines }
# Какие-то операции...
lines.each {|x| puts x }
```

Бывает, что нужно обрабатывать файл построчно.

```
File.open(filename) do |file|
   file.each_line do |line|
      # Какие-то операции...
      puts line
   end
end
```

Наконец, не забывайте, что все имена файлов, указанные в командной строке, автоматически собираются в объект `ARGF`, представляющий конкатенацию всех выходных данных (см. раздел 14.2.2). Мы можем вызывать, к примеру, метод `ARGF.readlines`, как если бы `ARGF` был объектом класса `IO`. Вся выходная информация будет, как обычно, направлена на стандартный вывод.

14.5.2. Копирование дерева каталогов (с символическими ссылками)

Пусть нужно скопировать целое дерево каталогов в новое место. Сделать это можно по-разному, но если в дереве есть символические ссылки, задача усложняется.

В листинге 14.1 приведено рекурсивное решение. Оно достаточно дружелюбно – контролирует входные данные и выводит информацию о порядке запуска.

Листинг 14.1. Копирование дерева каталогов

```
require "fileutils"

def recurse(src, dst)
  Dir.mkdir(dst)
  Dir.foreach(src) do |e|
    # Пропустить . и ..
    next if [".","..."].include? e
    fullname = src + "/" + e
    newname = fullname.sub(Regexp.new(Regexp.escape(src)),dst)
    if File.directory?(fullname)
      recurse(fullname,newname)
    elsif File.symlink?(fullname)
      linkname = `ls -l #{fullname}`.sub(/.* -> /,"").chomp
      newlink = linkname.dup
      n = newlink.index($oldname)
      next if n == nil
      n2 = n + $oldname.length - 1
      newlink[n..n2] = $newname
      newlink.sub!(/\/\//,"/")
      # newlink = linkname.sub(Regexp.new(Regexp.escape(src)),dst)
        File.symlink(newlink, newname)
    elsif File.file?(fullname)
      FileUtils.copy(fullname, newname)
    else
      puts "??? :  #{fullname}"
    end
  end
end


# "Главная программа"

if ARGV.size != 2
  puts "Порядок вызова: copytree oldname newname"
  exit
end

oldname = ARGV[0]
newname = ARGV[1]

if !File.directory?(oldname)
  puts "Ошибка: первый параметр должен быть именем существующего каталога."
  exit
end

if File.exist?(newname)
  puts "Ошибка: #{newname} уже существует."
```

```
  exit
end

oldname = File.expand_path(oldname)
newname = File.expand_path(newname)

$oldname=oldname
$newname=newname

recurse(oldname, newname)
```

В современных вариантах UNIX, например Mac OS X, команда `cp -R` сохраняет символические ссылки, но в более старых это не так. Программа, показанная в листинге 14.1, была написана для решения этой практической задачи.

14.5.3. Удаление файлов по времени модификации и другим критериям

Предположим, мы хотим удалить самые старые файлы из какого-то каталога. В нем могут, к примеру, храниться временные файлы, журналы, кэш браузера и т. п.

Представленная ниже небольшая программа удаляет файлы, которые в последний раз модифицировались раньше указанного момента (заданного в виде объекта `Time`):

```
def delete_older(dir, time)
  Dir.chdir(dir) do
    Dir.foreach(".") do |entry|
      # Каталоги не обрабатываются
      next if File.stat(entry).directory?
      # Используем время модификации
      if File.mtime(entry) < time
        File.unlink(entry)
      end
    end
  end
end

delete_older("/tmp",Time.local(2001,3,29,18,38,0))
```

Неплохо, но можно обобщить. Создадим метод `delete_if`, который принимает блок, возвращающий значение `true` или `false`. И будем удалять те и только те файлы, которые удовлетворяют заданному критерию.

```
def delete_if(dir)
  Dir.chdir(dir) do
    Dir.foreach(".") do |entry|
      # Каталоги не обрабатываются
      next if File.stat(entry).directory?
      if yield entry
```

```
        File.unlink(entry)
      end
    end
  end
end

# Удалить файлы длиннее 3000 байтов
delete_if("/tmp") { |f| File.size(f) > 3000 }

# Удалить файлы с расширениями LOG и BAK
delete_if("/tmp") { |f| f =~ /(log|bak)$/i }
```

14.5.4. Вычисление свободного места на диске

Пусть нужно узнать, сколько байтов свободно на некотором устройстве. В следующем примере это делается по-простому, путем запуска системной утилиты:

```
def freespace(device=".")
   lines = %x(df -k #{device}).split("\n")
   n = (lines.last.split[3].to_f / 1024 / 1024).round(2)
end

puts freespace("/")        # 48.7
```

Для Windows имеется несколько более элегантное решение (предложено Дэниэлем Бергером):

```
require 'Win32API'

GetDiskFreeSpaceEx = Win32API.new('kernel32', 'GetDiskFreeSpaceEx',
                                  'PPPP', 'I')

def freespace(dir=".")
  total_bytes = [0].pack('Q')
  total_free  = [0].pack('Q')
  GetDiskFreeSpaceEx.call(dir, 0, total_bytes, total_free)

  total_bytes = total_bytes.unpack('Q').first
  total_free = total_free.unpack('Q').first
end

puts freespace("C:")       # 5340389376
```

14.6. Различные скрипты

Приведем еще несколько примеров. Не претендуя на оригинальность, мы отнесли их к категории «разное».

14.6.1. Распространение программ на Ruby

Иногда требуется распространить свою Ruby-программу, чтобы ей могли пользоваться другие люди. Если предполагаемые пользователи работают в Mac OS X,

вам повезло, потому что Apple включила Ruby в дистрибутив Mac OS X (на момент написания этой книги включалась версия 2.0.0).

Для пользователей Windows на сайте rubyinstaller.org имеется инсталлятор Ruby, тоже версии 2.0.0. Пользователи Linux могут установить Ruby с помощью менеджера пакетов.

Однако имейте в виду, что во всех этих случаях устанавливается не последняя версия Ruby (последней на момент написания книги была версия 2.1.2, а готовые установщики есть в основном для версии 2.0.0). О том, как установить Ruby в среде разработки см. раздел 21.6 в главе 21 «Инструменты разработки для Ruby».

Но можно не просить пользователей установить на свою машину Ruby, а воспользоваться инструментом Omnibus (созданным разработчиками платформы управления конфигурацией Chef), который позволяет построить автономный пакет, включающий как сам интерпретатор Ruby, так и все необходимые gem-пакеты.

В Сети есть и другие подобные программы, помимо Omnibus. У каждой свои плюсы и минусы, так что я рекомендую изучить вопрос, прежде чем принимать решение.

14.6.2. Подача входных данных Ruby по конвейеру

Поскольку интерпретатор Ruby – это однопроходный транслятор, то можно подать ему на вход некий код и выполнить его. Это может оказаться полезным, когда обстоятельства вынуждают вас работать на традиционном скриптовом языке типа bash, но для каких-то сложных задач вы хотите применить Ruby.

В листинге 14.2 представлен bash-скрипт, который вызывает Ruby (посредством вложенного документа) для вычисления интервала в секундах между двумя моментами времени. Ruby-программа печатает на стандартный вывод одно значение, которое перехватывается вызывающим скриптом.

Листинг 14.2. bash-скрипт, вызывающий Ruby

```
# Для вычисления разницы в секундах между двумя моментами временами
# bash вызывает Ruby...

export time1="2007-04-02 15:56:12"
export time2="2007-12-08 12:03:19"

cat <<EOF | ruby
require "time"

time1 = ENV["time1"]
time2 = ENV["time2"]

t1 = Time.parse(time1)
t2 = Time.parse(time2)

diff = t2 - t1
```

```
puts diff
EOF

echo "Прошло секунд = " $elapsed
```

В данном случае оба исходных значения передаются в виде переменных окружения (которые необходимо экспортировать). Строки, читающие эти значения, можно было бы записать так:

```
time1="$time1"  # Включить переменные оболочки непосредственно
time2="$time2"  #   в строку...
```

Но возникающие при этом проблемы очевидны. Очень трудно понять, имеется ли в виду переменная bash или глобальная переменная Ruby. Возможна также путаница при экранировании и расстановке кавычек.

Флаг -e позволяет создавать однострочные Ruby-скрипты. Вот пример обращения строки:

```
#!/usr/bin/bash
string="Francis Bacon"
reversed=$(ruby -e "puts '$string'.reverse")
echo $reversed   # "nocaB sicnarF"
```

На самом деле, в Ruby предусмотрено несколько средств, удобных для написания однострочных скриптов. Чтобы автоматически выполнять указанный код для каждой строки входных данных, используется флаг -n. Следующий код инвертирует все поданные на вход строки:

```
$ echo -e "Knowledge\nis\npower\n" | ruby -ne 'print $_.reverse'
egdelwonK
si
rewop
```

Чтобы еще упростить жизнь, имеется флаг -p, который работает, как -n, но вдобавок после обработки каждой строки включает предложение print:

```
$ echo -e "France\nis\nBacon\n"   ruby -pe '$_.reverse! '
ecnarF
si
nocaB
```

Знатоки UNIX заметят, что awk использовался подобным образом с незапамятных времен.

14.6.3. Работает ли Ruby в интерактивном режиме?

Чтобы узнать, работает ли программа в интерактивном режиме, нужно проверить стандартный ввод. Метод isatty? возвращает true, если устройство интерактивное, а не диск или сокет. (Для Windows этот метод не реализован.)

```
if STDIN.isatty?
  puts "Привет! Я вижу, вы печатаете"
  puts "на клавиатуре."
```

```
else
  puts "Входные данные поступают не с клавиатуры."
end
```

14.6.4. Определение текущей платформы или операционной системы

Если программа хочет знать, в какой операционной системе исполняется, то может опросить ключ 'host_os' в глобальном хэше RbConfig::CONFIG. В ответ будет возвращена загадочная строка (что-то вроде darwin13.3.0, cygwin или solaris2), содержащая информацию о платформе, для которой был собран интерпретатор Ruby.

Ruby работает в основном в системах Mac OS X (Darwin), вариантах UNIX вроде Linux и Solaris или в Windows (XP, Vista, 7 и 8). Поэтому различить платформы можно с помощью очень простого регулярного выражения:

```
def os_family
   case RbConfig::CONFIG['host_os']
   when /(mingw|mswin|windows)/i
      "windows"
   when /cygwin/i
      "cygwin"
   when /(darwin|mac os)/i
      "osx"
   when /(linux|bsd|aix|solaris)/i
      "unix"
   end
end
```

Конечно, это весьма неуклюжий способ определения ОС. Даже если вы правильно определите семейство ОС, отсюда еще не следует, что нужная вам функциональность имеется (или отсутствует).

14.6.5. Модуль Etc

Модуль Etc получает различную информацию из файлов /etc/passwd и /etc/group. Понятно, что полезен он только на платформе UNIX.

Метод getlogin возвращает имя пользователя, от имени которого запущена программа. Если он завершается неудачно, может помочь метод getpwuid (принимающий в качестве необязательного параметра идентификатор пользователя uid).

```
require 'etc'

myself = Etc.getlogin                # Это я!
root_name = Etc.getpwuid(0).name     # Имя суперпользователя

# Без параметров getpwuid вызывает
# getuid ...
me2 = Etc.getpwuid.name              # Это снова я!
```

Метод getpwnam возвращает структуру passwd, которая содержит поля name, dir (начальный каталог), shell (начальный интерпретатор команд) и другие.

```
rootshell = getpwnam("root").shell   # /sbin/sh
```

Методы getgrgid и getgrnam ведут себя аналогично, но по отношению к группам. Они возвращают структуру group, содержащую имя группы и т.д.

Итератор passwd обходит все записи в файле /etc/passwd. Запись передается в блок в виде структуры passwd.

```
require 'etc'

all_users = []
passwd { |entry| all_users << entry.name }
```

Имеется также итератор group для обхода записей в файле /etc/group.

14.7. Заключение

На этом мы завершаем обсуждение применения Ruby для решения рутинных задач автоматизации. Мы видели, как передавать в программу и получать от нее информацию в виде переменных окружения и с помощью стандартного ввода-вывода. Мы познакомились с типичными операциями «склеивания», позволяющими разным программам взаимодействовать. Рассмотрели мы и различные уровни взаимодействия с операционной системой.

Поскольку значительная часть изложенного материала системно-зависима, я призываю вас экспериментировать. Между платформами Mac OS X, Windows и UNIX имеются серьезные различия. Есть они и между разными операционными системами, относящимися к одному семейству.

Следующая тема, которую мы рассмотрим, тоже весьма широка. Речь пойдет о работе с данными в разных форматах, от графических до XML.

Глава 15. Ruby и форматы данных

Ваша информация, сэр, — говорит Библиотекарь.
— Сможешь увязать эту информацию
с утилитой «ВЫ ЗДЕСЬ»? — говорит Хиро.
Я посмотрю, что можно сделать, сэр.
Форматы представляются совместимыми.

— Нил Стивенсон «Лавина»

В программировании так уж получается, что по мере усложнения какой-то информации для ее описания вырабатывается отдельный «миниязык», а чаще даже *несколько* таких языков. Мы называем их *форматами файлов* или данных.

Любой из вас припомнит сотни примеров форматов файлов. Это и графические форматы типа JPG, GIF и PNG, и форматы документов (DOC и PDF), и «универсальные» форматы (CSV, XML или JSON) и бесчисленные форматы, разработанные отдельными компаниями, многие из которых являются просто вариациями на тему хранения данных в виде таблицы с фиксированной шириной колонок, столь популярного в античные времена (я имею в виду 1960-е годы).

Один из самых простых и наиболее употребительных форматов данных — простой текст. Файлы, состоящие из неструктурированного множества байтов, представленных в кодировке ASCII (или ее надмножеств типа ISO 8859-1 и UTF-8), за прошедшие десятилетия доказали свою полезность как самой надежной и доступной формы хранения данных.

Но даже на такой простой формат можно наложить ту или иную структуру (отсюда и популярность XML и JSON). Бывают также чисто двоичные (например, Protocol Buffers) и двоично-текстовые форматы (например, RTF). Выбирая между двоичным и текстовым форматом, отдают предпочтение либо удобочитаемости, либо скорости; какой формат лучше, зависит от конкретных обстоятельств.

Но в каком бы формате данные ни хранились, рано или поздно их придется читать, разбирать и снова сохранять. В этой главе мы рассмотрим лишь несколько самых распространенных форматов файлов; в одной книге невозможно охватить все употребляемые форматы. Если вы хотите разбирать файлы в таких форматах, как vCard, iCal и прочие, то придется поискать соответствующие библиотеки или, быть может, написать свою собственную.

15.1. Разбор JSON

Формат JSON (JavaScript Object Notation) проектировался, чтобы в понятно ловеку виде, допускающем массивы, хэши, числа и строки, сериализовать да в виде простого текста для обмена между программами, написанными на л языке. Созданный в противовес многословности XML, формат JSON, с одно роны, гораздо лаконичнее, если измерять в терминах количества знаков, а, с гой, беднее в плане множества поддерживаемых типов.

В разделе 10.2.5 главы 10 «Ввод-вывод и хранение данных» мы уже вст лись с использованием JSON для хранения данных в Ruby. В этом разделе м говорим о том, как разбирать данные в этом формате и манипулировать им точником таких данных может быть, например, API какого-то сайта. Но сн рассмотрим преобразование строки или файла, содержащего JSON-данные, Ruby:

```
require 'json'

data = JSON.parse('{"students":[
   {"name":"Alice","grade":4},
   {"name":"Bob","grade":3}
]}')
p data["students"].first.values_at("name", "grade")
# ["Alice", 4]
```

То, что в JSON называется «объектом», почти точно соответствует хэшу в л ле Ruby. Кроме того, в JSON поддерживаются еще несколько хорошо знак нам типов: массивы, строки и числа. Все данные в это формате представлены ко в кодировке UTF-8. Помимо вышеперечисленных, можно еще представит чения `true`, `false` и `null` (соответствует `nil` в Ruby).

15.1.1. Обход JSON-данных

После того как JSON-данные разобраны, можно обойти образовавшиеся вло ные хэши и массивы и извлечь из них нужные данные. Для иллюстрации воз данные, которые возвращает открытый API сайта GitHub:

```
require 'json'
require 'open-uri'
require 'pp'

json = open("https://api.github.com/repos/ruby/ruby/contributors")
users = JSON.parse(json)

pp users.first
# {"login"=>"nobu",
#  "id"=>16700,
#  "url"=>"https://api.github.com/users/nobu",
#  "html_url"=>"https://github.com/nobu",
#  [...прочие атрибуты...]
```

```
#  "type"=>"User",
#  "site_admin"=>false,
#  "contributions"=>9850}

users.sort_by!{|u| -u["contributions"] }
puts users[0...10].map{|u| u["login"] }.join(", ")
# nobu, akr, nurse, unak, eban, koi, drbrain, knu, kosaki, mame
```

Здесь мы воспользовались библиотекой open-uri. Подробнее она рассматривается в главе 18 «Сетевое программирование», а пока достаточно знать, что она позволяет использовать метод open для URI – во многом так же, как для обычного файла.

С помощью open-uri мы скачиваем строку, которая содержит JSON-массив всех, кто принимает участие в разработке основного интерпретатора Ruby. (Многие авторы не имеют учетных записей на сайте GitHub и потому не включены в этот список.) Разбор JSON-данных возвращает массив хэшей, в каждом из которых находится информация об одном авторе. Среди ключей этого хэша мы видим, в частности, login, id, url и contributions.

Затем мы с помощью библиотеки красивой печати «prettyprint» распечатываем хэш атрибутов для первого автора (pp печатает все атрибуты, но мы для краткости часть из них опустили). Далее этот список сортируется в порядке убывания количества вкладов автора, т. е. первым будет хэш автора, сделавшего больше всех записей в репозиторий исходного кода. Наконец, из отсортированных таким образом хэшей с информацией об авторах извлекаются только их логины и распечатываются через запятую.

Реальные приложения обычно похожи на этот пример. Мы пишем код для получения данных из источника, разбираем их, представляя в виде хэша или массива Ruby, а затем ищем в нем элементы, содержащие интересующие нас данные.

15.1.2. Типы данных, не представимые в JSON

В главе 10 мы уже видели, что в формате JSON нельзя представить объекты Ruby. В значительной мере работа с JSON сводится к написанию кода для преобразования объектов в хэши, содержащие только строки и числа, и обратно. Есть библиотеки, например Oj, которые предлагают автоматические средства для такого преобразования.

Большинство объектов нетрудно представить в виде хэша, содержащего строки и числа, но так бывает не всегда. Так, напрямую невозможно представить в формате JSON символы, экземпляры классов Time и Date и денежные величины с точностью до цента.

Для символов все просто – нужно лишь преобразовать символ в JSON-строку. Для объектов типа Time придется воспользоваться методом iso8601 или to_i, а затем обратить процедуру, применив метод parse или at. Для денежных величин десятичное число нужно будет сохранить в строковом формате. (Согласно спецификации JSON, любое число рассматривается как число с плавающей точкой. Поэтому

нельзя предполагать, что число содержит сотые доли точно, обычно оно оказывается чуть больше или чуть меньше.)

В конечном итоге, алгоритм сохранения таких данных в формате JSON сводится к следующему правилу: преобразовать в строку или в число с плавающей точкой, а затем разобрать эту строку или число и восстановить исходный объект.

15.1.3. Другие библиотеки для работы с JSON

Помимо стандартной библиотеки `json`, существует еще ряд библиотек, предлагающих дополнительную функциональность.

Иногда JSON-документ оказывается таким большим, что не помещается в память, и его приходится читать из файла или из сети потоком. В таких случаях полезен gem-пакет `json-stream`. В случае потокового разбора можно выполнить заданный блок кода всякий раз, как встречается начало или конец документа, массива или объекта либо после обнаружения указанного ключа или значения. Дополнительные сведения см. в документации по gem-пакету `json-stream`.

Еще одно типичное применение JSON в веб – использование долговременного HTTP-соединения для отправки потока событий, в котором каждое событие представлено в виде отдельного JSON-документа. Наверное, самый известный пример такого рода – Twitter API, но и многие другие API построены по аналогичному образцу. Gem-пакет `yajl-ruby` специально предназначен для этого случая. Он умеет разбирать поток JSON-объектов, поступающий от объекта `IO`, а также сам отправлять HTTP-запросы; при этом по завершении разбора JSON-документа вызывается заданный блок.

Наконец, упомянем gem-пакет `Oj`, написанный на C и оптимизированный для максимально быстрого кодирования и обратного разбора. Правда, встречаются тонкие несовместимости с другими кодировщиками JSON, поэтому обязательно тестируйте. Но в любом случае производительность этой библиотеки поистине впечатляет, она работает от двух до пяти раз быстрее всех остальных библиотек, относящихся к JSON.

15.2. Разбор XML (и HTML)

XML (*eXtensible Markup Language*) – язык разметки, основанный на использовании тегов. HTML, язык гипертекстовой разметки, очень похож на него (на самом деле, общим предком XML и HTML является язык SGML). XML и HTML достигли огромной популярности в 1990-е годы и до сих пор активно применяются в инструментах разработки, для хранения и передачи данных, а также повсеместно в веб.

В XML и HTML у каждого элемента есть имя, а весь документ имеет иерархическую структуру. И хотя оба формата очень многословны, всё записывается простым текстом, и при желании человек может это прочитать. Еще одно достоинство XML по сравнению с форматами 1970-х и 1980-х годов – возможность иметь данные переменной длины, а не требовать, чтобы каждое поле занимало фиксированное число байтов.

Тридцать-сорок лет назад XML был бы невозможен из-за ограничений на объем памяти. Но если бы он все же появился тогда, то знаменитая проблема 2000 года (хотя, как оказалось, проблема-то и яйца выеденного не стоила) вообще не возникла бы. Причиной-то было то, что в унаследованных системах данные хранились в формате с фиксированной длиной. Так что, несмотря на некоторые недостатки, у XML есть сферы применения.

В Ruby для чтения, записи и манипулирования данными в форматах XML и HTML чаще всего используется gem-пакет Nokogiri. Он предоставляет интерфейс (употребляется также термин *привязка*) из Ruby к библиотеке libXML2, написанной на C. В пакете Nokogiri есть два API: документо-ориентированный и потоковый. Мы рассмотрим оба подхода.

15.2.1. Разбор документа

В случае документо-ориентированного подхода файл разбирается целиком и результатом разбора является древовидная структура. Эту структуру можно обойти, как мы обходили хэши, образующиеся при разборе JSON. Но в отличие от хэшей, для документов в Nokogiri определен язык запросов, позволяющий выбирать определенное подмножество элементов. Nokogiri поддерживает два таких языка: CSS-селекторы и XPath.

Для разбора HTML-документа библиотека Nokogiri предоставляет метод parse, являющийся методом класса Nokogiri::HTML. Обход и операции с результирующим документом производятся так же, как для XML-документов, поэтому все примеры ниже относятся к XML.

В примерах мы будем использовать следующий простой XML-файл, описывающий часть личной библиотеки.

Листинг 15.1. Файл books.xml

```
<library shelf="Recent Acquisitions">
   <section name="Ruby">
      <book isbn="0321714636">
         <title>The Ruby Way</title>
         <author>Hal Fulton</author>
         <author>André Arko</author>
         <description>Third edition. The book you are now reading.
                     Ain't recursion grand?
         </description>
      </book>
   </section>
   <section name="Space">
      <book isbn="0684835509">
         <title>The Case for Mars</title>
         <author>Robert Zubrin</author>
         <description>Pushing toward a second home for the human
                     race.
         </description>
```

```
      </book>
      <book isbn="074325631X">
        <title>First Man: The Life of Neil A. Armstrong</title>
        <author>James R. Hansen</author>
        <description>Definitive biography of the first man on
                     the moon.
        </description>
      </book>
   </section>
</library>
```

Сначала разберем эти XML-данные как документ. Первым делом затребуем библиотеку nokogiri, а затем приступим к разбору:

```
require 'nokogiri'
doc = Nokogiri::XML.parse File.read("books.xml")

doc.root.name      # библиотека
doc.root["shelf"]  # недавние приобретения
```

Метод root возвращает экземпляр класса Nokogiri::XML::Element, представляющий «корневой» тег, внутри которого содержится весь XML-документ. Все атрибуты тега (заданные внутри угловых скобок) можно прочитать с помощью метода [], точно так же, как при доступе к хэшу. Затем мы обходим документ и выбираем некоторые элементы:

```
books = doc.css("section book")     # a NodeSet, напоминает массив
books = doc.xpath("//section/book") # тот же самый NodeSet

books.each do |book|
   title = book.elements.first.text
   authors = book.css("author").map(&:text)
   puts "#{title} (ISBN #{book["isbn"]})"
   puts " by #{authors.join(' and ')}"
end
# Выводится:
# The Ruby Way (ISBN 0672328844)
#      by Hal Fulton and André Arko
# The Case for Mars (ISBN 0684835509)
#      by Robert Zubrin
# First Man: The Life of Neil A. Armstrong (ISBN 074325631X)
#      by James R. Hansen
```

С помощью методов css и xpath мы можем выбрать только те теги book, которые находятся внутри тегов section. Подробные сведения о CSS-селекторах и синтаксисе языке XPath см. на каком-нибудь сайте с документацией, например developer.mozilla.org. Оба метода возвращают объект NodeSet, который во многих отношениях ведет себя как массив. У этого объекта имеются методы each и size, а также метод [] для доступа по индексу.

Отобрав теги book, мы затем распечатываем текст из первого элемента (тега title) и используем метод css для получения тегов author, вложенных в тег book. Наконец, мы читаем атрибут "isbn" каждой книги с помощью метода [] и распечатываем информацию о книгах. Теперь посмотрим на теги section:

```
doc.root.elements.map{|e| e["name"] } # ["Ruby", "Space"]

space = doc.at_css("section[name=Space]")          # Element
space = doc.at_xpath("//section[@name='Space']")  # тот же самый Element

books.include?(space.elements.first) # true
```

У каждого элемента имеется метод elements, который возвращает список содержащихся в нем элементов. Читая атрибут "name", мы узнаем имена в тегах section. Затем с помощью методов at_css и at_xpath находим тег section, в котором атрибут "name" равен "Space". Методы вида at_ возвращают только первый из элементов, соответствующих указанному селектору. Наконец, мы удостоверяемся, что элементы, возвращенные разными методами, действительно являются одним и тем же объектом Ruby, потому что первый элемент book в разделе section с именем "Space" также входит в набор отобранных ранее книг.

Существуют различные способы выбрать одни и те же элементы, например: elements, css и xpath. Можно также воспользоваться любым методом из модуля Enumerable, например find или select, для выборки элементов с помощью блока.

Над элементами можно производить различные операции. Для каждого из рассмотренных выше методов, например text или [], имеется акцессор записи, позволяющий добавлять, удалять и изменять атрибуты и содержимое элементов. Модифицированный XML или HTML-документ можно записать в виде строки с помощью метода to_xml или to_html соответственно.

Дополнительные сведения об использовании библиотеки Nokogiri для разбора XML и HTML-документов см. в полном описании API на сайте nokogiri.org.

15.2.2. Потоковый разбор

Потоковый разбор основан на принципе «разбирай по мере поступления», он полезен, когда документ очень велик или имеются ограничения на доступную память. В этом случае документ разбирается по мере чтения с диска, а файл целиком никогда в памяти не хранится. После распознавания каждого элемента вызываются написанные пользователем методы или блоки *обратного вызова*. Таким образом, файл обрабатывается постепенно.

Обработаем тот же самый XML-файл потоком (на практике мы вряд ли стали бы это делать, потому что файл невелик). Есть разные варианты этой методики, в листинге 15.2 показан один из них. Идея в том, чтобы определить класс *прослушивателя*, методы которого будет вызывать анализатор. При разборе XML ваш класс должен быть подклассом Nokogiri::XML::SAX::Document, а при разборе HTML – подклассом Nokogiri::HTML::SAX::Document.

Листинг 15.2. Потоковый разбор

```
require 'nokogiri'
class Listener < Nokogiri::XML::SAX::Document
   def start_element(name, attrs = [])
      case name
      when "book"
         isbn = attrs.find{|k,v| k == "isbn" }.last
         puts "Новая книга с ISBN #{isbn}"
      when "title", "author", "description"
         print "#{name.capitalize}: "
      else
      end
   end

   def characters(string)
      return unless string =~ /\w/ # игнорировать строку из одних пробелов
      print string.tr("\n", " ").squeeze(" ")
   end

   def end_element(name)
      print "\n" if %w[book title author description].include?(name)
   end

   def end_document
      puts "Конец документа."
   end
end

xml = File.read("books.xml")
Nokogiri::XML::SAX::Parser.new(Listener.new).parse(xml)
```

Класс `Nokogiri::XML::SAX::Document` предоставляется пустые методы обратного вызова, которые мы переопределяем в нашем подклассе `Listener`. Например, встретив открывающий тег, анализатор вызывает метод `start_element`, передавая ему имя тега и все его атрибуты. В этом методе мы на основе имени тега определяем, что делать: если это тег `book`, то мы сообщаем о том, что обнаружена новая книга и печатаем ее номер ISBN. Если это тег `title`, `author` или `description`, то мы распечатываем только имя тега. Для остальных тегов не делаем ничего.

В методе `characters` первое предложение возвращает управление, ничего не печатая, если строка не содержит ни одного символа, являющегося частью «слова». В противном случае мы печатаем текст внутри тега, заменяя знаки новой строки пробелами, а несколько повторяющихся пробелов – одним.

Метод `end_element` печатает знак новой строки по завершении любого из тегов, содержащих информацию о книге, т. е. после печати всего текста внутри этого тега. Мы проверяем, что завершился один из тегов, перечисленных в списке, потому что не хотим переходить на новую строку, встретив завершающий тег `section` или `library`. Метод `end_document` работает аналогично, но вызывается только один раз.

Хотя код в листинге 15.2 несколько искусственный, применение его к файлу books.xml порождает хорошо организованную (и, хочется надеяться, полезную) распечатку, показанную в листинге 15.3.

Листинг 15.3. Результат работы программы потокового разбора

```
Новая книга с ISBN 0321714636
Title: The Ruby Way
Author: Hal Fulton
Author: André Arko
Description: Third edition. The book you are now reading.
Ain't recursion grand?

Новая книга с ISBN 0684835509
Title: The Case for Mars
Author: Robert Zubrin
Description: Pushing toward a second home for the human race.

Новая книга с ISBN 074325631X
Title: First Man: The Life of Neil A. Armstrong
Author: James R. Hansen
Description: Definitive biography of the first man on the moon.

Конец документа.
```

Дополнительные сведения об использовании библиотеки Nokogiri для разбора документов в формате XML и HTML см. в описании API на сайте nokogiri.org.

15.3. RSS и Atom

По мере роста веб разработчики столкнулись с необходимостью объявлять о различных событиях, изменениях да и просто о новых материалах на сайте. В конечном итоге выкристаллизовалось и решение — публиковать *ленту новостей*, или *новостной канал* сайта в формате XML-документа, структурированного в соответствии со стандартом RSS или Atom.

Стандарт RSS (Really Simple Syndication) появился на заре развития Интернета, после чего разработчики дополняли его разными интересными (и иногда противоречащими друг другу) функциями. В результате потребовался новый стандарт, каковым и стал Atom – самый развитый и широко используемый в наши дни стандарт новостных каналов.

В основе RSS лежит язык XML, так что его *можно* было бы разбирать как обычный XML-документ. Однако поскольку формат RSS специализирован, он заслуживает и специального анализатора. К тому же, непоследовательность RSS уже стала легендой, и не редки программы, порождающие «кривой» RSS, с трудом поддающийся разбору.

Это неудобство осложняется еще и наличием нескольких несовместимых версией стандарта RSS, из которых чаще всего встречаются версии 0.9, 1.0 и 2.0. О вер-

сиях RSS мы хотели бы знать не больше, чем о том, как делают колбасу, – эти неаппетитные детали интересны только специалистам.

15.3.1. Разбор новостной ленты

Рассмотрим обработку документов в форматах RSS и Atom с помощью стандартной библиотеки Ruby rss, которая прозрачно поддерживает не только все три версии RSS, но и Atom. В примере ниже мы запрашиваем у предоставляемой НАСА службы «Астрономическая картина дня» ленту новостей, а затем печатаем заголовки всех новостей:

```
require 'rss'
require 'open-uri'

xml = open("http://apod.nasa.gov/apod.rss").read
feed = RSS::Parser.parse(xml, false)

puts feed.channel.description
feed.items.each_with_index do |item, i|
   puts "#{i + 1}. #{item.title.strip}"
end
```

Обратите внимание, что анализатор RSS извлекает *канал* новостной ленты; наша программа затем печатает заголовок канала. Существует также список элементов (его возвращает акцессор items), который можно рассматривать как перечень статей. Мы получаем весь список и печатаем заголовок каждой статьи.

Разумеется, конкретные данные день ото дня меняются, но вот что получилось у меня в результате запуска этой программы:

```
Astronomy Picture of the Day
1. Star Trails Over Indonesia
2. Jupiter and Venus from Earth
3. No X rays from SN 2014J
4. Perseid in Moonlight
5. Surreal Moon
6. Rings Around the Ring Nebula
7. Collapse in Hebes Chasma on Mars
```

Прежде чем двигаться дальше, я хотел бы оказать любезность поставщикам каналов. Программами, подобными приведенной выше, следует пользоваться с осторожностью, так как они потребляют ресурсы сервера поставщика. В любом реальном приложении, например в агрегаторе каналов, следует прибегать к кэшированию. Вот простой (но работающий) пример кэширования:

```
unless File.exist?("apod.rss")
   File.write("apod.rss", open("http://apod.nasa.gov/apod.rss"))
end

xml = File.read("apod.rss")
```

Здесь мы просто читаем ленту из файла, а если ее еще не существует, то записываем в файл. Для реализации более полезной схемы кэширования мы, наверное, проверяли бы дату создания кэшированного файла и вновь запрашивали бы ленту, если возраст файла превышает некоторый порог. Кроме того, мы использовали бы HTTP-заголовки `If-Modified-Since` или `If-None-Match`. Однако такая система уже выходит за рамки простого примера.

Ленты в формате Atom разбираются тем же самым методом `RSS::Parser.parse`. Библиотека `rss` читает содержимое ленты и сама определяет, каким анализатором пользоваться. Единственное существенное различие состоит в том, что в формате Atom отсутствует атрибут `channel`. Вместо этого нужно искать атрибуты `title` и `author` в самой ленте.

15.3.2. Создание новостных лент

Есть также возможность генерировать документы в формате RSS или Atom с помощью метода `RSS::Maker` из стандартной библиотеки RSS. В примере ниже создается небольшая лента в формате Atom для гипотетического сайта:

```
require 'rss'

feed = RSS::Maker.make("atom") do |f|
   f.channel.title = "Feed Your Head"
   f.channel.id = "http://nosuchplace.org/home/"
   f.channel.author = "Y.T."
   f.channel.logo = "http://nosuchplace.org/images/headshot.jpg"
   f.channel.updated = Time.now

   f.items.new_item do |i|
      i.title = "Once again, here we are"
      i.link = "http://nosuchplace.org/articles/once_again/"
      i.description = "Don't you feel more like you do now than usual?"
      i.updated = Time.parse("2014-08-17 10:23AM")
end
```

Объект `Maker` создается в инициализаторе блока, подобном обсуждавшемуся в разделе 11.1.3 главы 11 «ООП и динамические механизмы в Ruby». В блоке мы устанавливаем атрибуты ленты `title`, `id`, `author` и временную метку `updated` – все они являются обязательными.

Затем мы вызываем метод `new_item`, который создает новую статью и добавляет ее в ленту, снова с помощью блока. У каждой статьи должны быть атрибуты `title`, `description` (или `summary`), `link` (ссылка на URL, содержащий полный текст статьи) и `updated` (временная метка). Необязательный атрибут `content` позволяет включить полный текст непосредственно в статью.

Стандартная библиотека RSS очень неплоха, но есть и другие библиотеки для разбора и генерации новостных лент в форматах RSS и Atom. Если для ваших целей стандартной библиотеки недостаточно, зайдите на сайт ruby-toolbox.com или поищите в Интернете что-нибудь более подходящее.

15.4. Обработка изображений с помощью RMagick

По мере того как компьютеры, планшеты и смартфоны становятся основным способом потребления мультимедийного контента, умение программно обрабатывать графические данные приобретает все большую значимость. Программистам нужны средства для манипулирования изображениями, представленными в различных сложных форматах на разных платформах. На языке Ruby это лучше всего делать с помощью библиотеки RMagick, которую написал Тим Хантер (Tim Hunter).

Gem-пакет RMagick – это интерфейс из Ruby к библиотеке ImageMagick. Устанавливается он командой `gem install rmagick`, но для работы нужно сначала установить саму библиотеку ImageMagick. Если вы работаете в Linux, то, вероятно, она уже имеется, а на платформе Mac OS X ее можно установить с помощью приложения Homebrew. Если понадобится помощь, то начать лучше всего с сайта ImageMagick по адресу http://imagemagick.org.

Поскольку RMagick – лишь интерфейс, то спрашивать, какие графические форматы она поддерживает, – все равно, что спрашивать, какие форматы поддерживает базовая библиотека. Все наиболее распространенные, в частности, JPG, GIF, PNG, TIFF наряду с десятками других. То же относится и к операциям, поддерживаемым RMagick. Они ограничены лишь возможностями базовой библиотеки, поскольку RMagick дублирует весь ее API, адаптируя его к возможностям Ruby: символам, блокам и т. д.

Кстати говоря, ImageMagick API поистине огромен. Ни этой главы, ни даже всей книги целиком не хватило бы для рассмотрения всех его деталей. В следующих разделах мы дадим лишь общее представление об RMagick, а полную информацию вы можете найти на сайте проекта (http://www.imagemagick.org/RMagick/doc/).

15.4.1. Типичные графические задачи

Одна из самых простых и распространенных задач, связанных с графическим файлом, – получение характеристик изображения (ширина и высота в пикселях и т.д.). Посмотрим, как можно извлечь эти метаданные.

На рис. 15.1 приведены два простых изображения, на которые мы будем ссылаться в этом и последующих примерах. Первое (smallpic.jpg) – просто абстрактная картинка, созданная в графическом редакторе; в ней присутствуют несколько оттенков серого цвета, а также прямые и кривые линии. Второе – фотография старенького автомобиля, которую я сделал в 2002 году в сельском районе Мексики. Для книги оба изображения переведены в черно-белый формат. В листинге 15.4 показано, как извлечь из соответствующих файлов необходимую информацию.

Листинг 15.4. Получение информации об изображении

```
require 'rmagick'

def show_info(fname)
  img = Magick::Image::read(fname).first
```

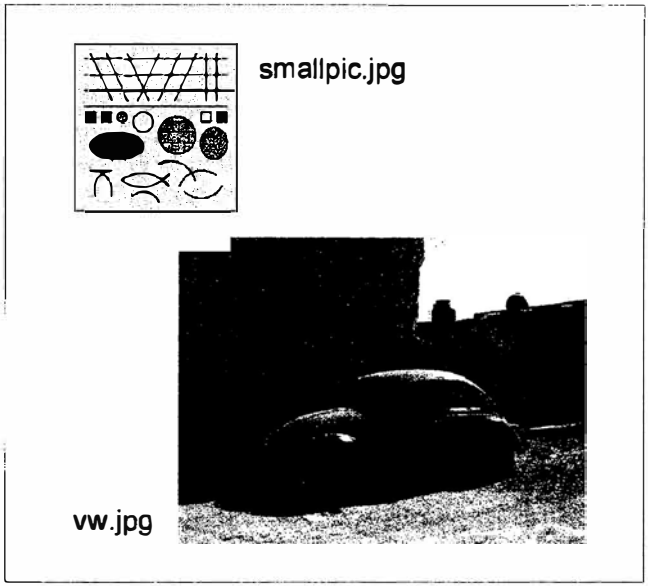


Рис. 15.1. Два примера изображений

```
  fmt = img.format
  w,h = img.columns, img.rows
  dep = img.depth
  nc  = img.number_colors
  nb  = img.filesize
  xr  = img.x_resolution
  yr  = img.y_resolution
  res = Magick::PixelsPerInchResolution ? "дюйм" : "см"

  puts <<-EOF
  Файл:  #{fname}
  Формат:#{fmt}
  Размеры:  #{w}x#{h} пикселей
  Цветов:#{nc}
  Длина файла: #{nb} байтов
  Разрешение:  #{xr}/#{yr} пикселей на #{res}
  EOF
  puts
end

show_info("smallpic.jpg")
show_info("vw.jpg")
```

Вот результат работы этой программы:

```
Файл:  smallpic.jpg
Формат:  JPEG
```

```
Размеры: 257x264 пикселей
Цветов:  248
Длина файла:  19116 байтов
Разрешение: 72.0/72.0 пикселей на дюйм

Файл: vw.jpg
Формат:  JPEG
Размеры: 640x480 пикселей
Цветов:  256
Длина файла:  55892 байтов
Разрешение: 72.0/72.0 пикселей на дюйм
2.0 pixels per inch
```

Разберемся, как именно работает эта программа. Для чтения файла мы вызываем метод `Magick::Image::read`. Поскольку один файл (например, анимированный GIF) может содержать несколько изображений, эта операция возвращает массив изображений (мы получаем лишь первое, вызывая метод `first`). Для чтения файла можно также воспользоваться методом `Magick::ImageList.new`.

У объекта, представляющего изображение, есть ряд методов чтения: `format` (название формата изображения), `filesize`, `depth` и другие. Не так очевидно, что для получения ширины и высоты изображения служат методы `columns` и `rows` соответственно (поскольку изображение представляется в виде прямоугольной таблицы пикселей). Разрешение представляется двумя числами, так как может быть разным по вертикали и горизонтали.

Можно получить и другие метаданные об изображении. Подробнее об этом вы можете прочитать в онлайновой документации по RMagick.

Часто возникает необходимость перевести изображение из одного формата в другой. В RMagick это проще всего сделать, прочитав изображение из файла в одном из поддерживаемых форматов и записав его в другой файл. Новый формат определяется расширением имени файла. Понятно, что «за кулисами» при этом происходит преобразование данных. Пример:

```
img = Magick::Image.read("smallpic.jpg")
img.write("smallpic.gif")   # Преобразовать в формат GIF
```

Иногда нужно изменить размер изображения (сделать его больше или меньше). Для этого обычно применяется один из трех методов: `thumbnail`, `resize` и `scale`. Все они принимают либо число с плавающей точкой (коэффициент масштабирования), либо два числа (новые размеры в пикселях). Различия между этими методами продемонстрированы в листинге 15.5.

Листинг 15.5. Три способа масштабирования изображения

```
require 'rmagick'

img = Magick::ImageList.new("vw.jpg")

# Метод thumbnail предназначен для получения миниатюрной версии большого
# изображения. Он самый быстрый из всех, особенно если нужно получить очень
```

```
# маленькое изображение.

pic1 = img.thumbnail(0.2)       # Уменьшить до 20%
pic2 = img.thumbnail(64,48)     # Новый размер 64x48 пикселей

# resize работает со средней скоростью, он изменяет размеры изображения,
# сохраняя пропорции. Необязательные третий и четвертый
# параметры интерпретируются как фильтр и размывание
# соответственно. По умолчанию подразумевается фильтр LanczosFilter и
# коэффициент размывания 1.0

pic3 = img.resize(0.40)         # Уменьшить до 40%
pic4 = img.resize(320,240)      # Новый размер 320x240
pic5 = img.resize(300,200,Magick::LanczosFilter,0.92)

# Метод scale самый медленный, он изменяет ширину и высоту независимо
#(при необходимости искажая пропорции).

pic8 = img.scale(0.60)          # Уменьшить до 60%
pic9 = img.scale(400,300)       # Новый размер 400x300
```

Изображения можно подвергать и многим другим преобразованиям. Некоторые просты и понятны, другие гораздо сложнее. В следующем разделе мы рассмотрим несколько интересных преобразований и специальных эффектов.

15.4.2. Специальные эффекты и преобразования

Над изображением можно выполнять следующие операции: зеркально отразить, инвертировать цвета, повернуть, растянуть, перекрасить и т.д. В RMagick есть десятки методов для выполнения подобных операций, причем многие из них еще и настраиваются путем задания параметров.

В листинге 15.6 показано 12 различных эффектов. Чтобы уменьшить размер программы, метод `example` принимает имя файла, символ, соответствующий методу, и имя нового файла; он читает файл, вызывает метод и записывает результат в новый файл. Сами методы (скажем, `do_rotate`) по большей части просты, они получают изображение и вызывают его метод экземпляра (а возвращают результат преобразования).

Листинг 15.6. Двенадцать специальных эффектов и преобразований

```
require 'rmagick'

def do_flip(img)
  img.flip
end

def do_rotate(img)
  img.rotate(45)
end
```

```
def do_implode(img)
  img = img.implode(0.65)
end

def do_resize(img)
  img.resize(120,240)
end

def do_text(img)
  text = Magick::Draw.new
  text.annotate(img, 0, 0, 0, 100, "HELLO") do
    self.gravity = Magick::SouthGravity
    self.pointsize = 72
    self.stroke = 'black'
    self.fill = '#FAFAFA'
    self.font_weight = Magick::BoldWeight
    self.font_stretch = Magick::UltraCondensedStretch
  end
  img
end
def do_emboss(img)
  img.emboss
end


def do_spread(img)
  img.spread(10)
end

def do_motion(img)
  img.motion_blur(0,30,170)
end

def do_oil(img)
  img.oil_paint(10)
end

def do_charcoal(img)
  img.charcoal
end
```

```
  img.affine_transform(spin_xform)          # Применить преобразование
end

def example(old_file, meth, new_file)
  img = Magick::ImageList.new(old_file)
  new_img = send(meth,img)
  new_img.write(new_file)
end

example("smallpic.jpg", :do_flip,    "flipped.jpg")
example("smallpic.jpg", :do_rotate,  "rotated.jpg")
example("smallpic.jpg", :do_resize,  "resized.jpg")
example("smallpic.jpg", :do_implode, "imploded.jpg")
example("smallpic.jpg", :do_text,    "withtext.jpg")
example("smallpic.jpg", :do_emboss,  "embossed.jpg")

example("vw.jpg", :do_spread,   "vw_spread.jpg")
example("vw.jpg", :do_motion,   "vw_motion.jpg")
example("vw.jpg", :do_oil,      "vw_oil.jpg")
example("vw.jpg", :do_charcoal, "vw_char.jpg")
example("vw.jpg", :do_vignette, "vw_vig.jpg")
example("vw.jpg", :do_affine,   "vw_spin.jpg")
```

Мы продемонстрировали методы flip, rotate, implode, resize, annotate и другие. Результаты представлены на рис. 15.2.

О том, какие еще существуют трансформации изображений, читайте в онлайновой документации по адресу http://www.imagemagick.org/RMagick/doc/.

15.4.3. API рисования

В RMagick имеется развитый API для рисования линий, многоугольников и различных кривых. Он поддерживает заливку, полупрозрачность, задание цветов, выбор шрифтов, вращение, растяжение и другие операции.

Полное описание этого API выходит за рамки книги, но чтобы получить представление об имеющихся возможностях, рассмотрим простой пример.

В листинге 15.7 приведена программа, которая рисует на заданном фоне сетку, а поверх нее несколько закрашенных геометрических фигур. Черно-белое изображение, получившееся в результате, показано на рис. 15.3.

Листинг 15.7. Простая программа рисования

```
require 'rmagick'

img = Magick::ImageList.new
img.new_image(500, 500)

purplish = "#ff55ff"
yuck = "#5fff62"
bleah = "#3333ff"
```

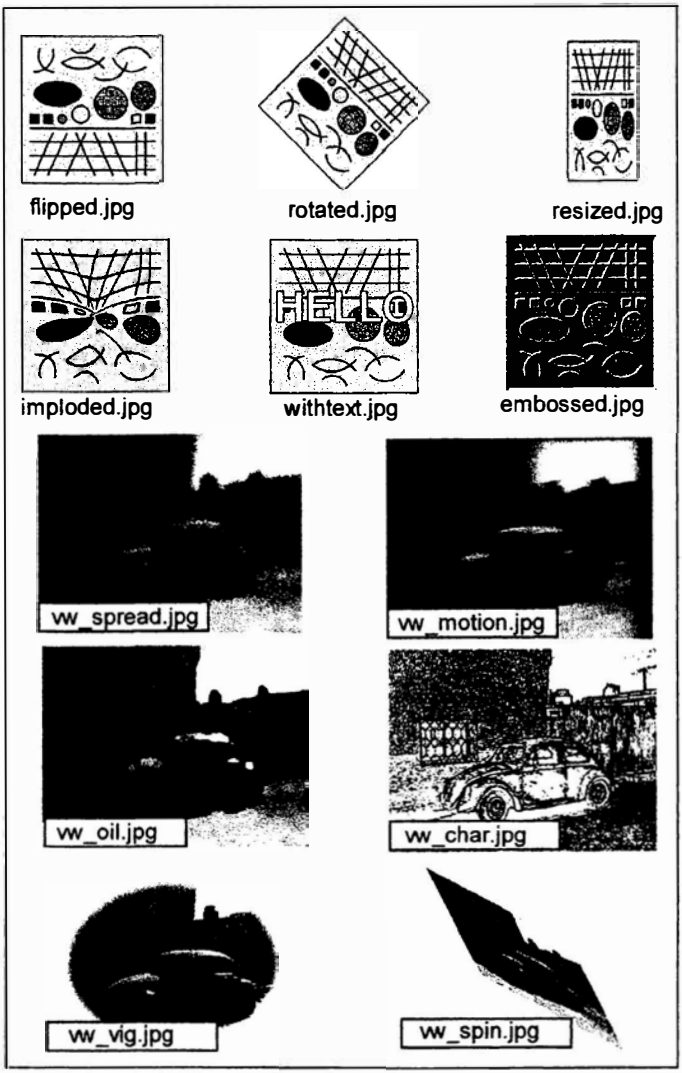


Рис. 15.2. Двенадцать специальных эффектов и преобразований

```
line = Magick::Draw.new
50.step(450,50) do |n|
  line.line(n,50, n,450)  # вертикальная прямая
  line.draw(img)
  line.line(50,n, 450,n)  # горизонтальная прямая
  line.draw(img)
end

# Нарисовать окружность
cir = Magick::Draw.new
```

```
cir.fill(purplish)
cir.stroke('black').stroke_width(1)
cir.circle(250,200, 250,310)
cir.draw(img)

rect = Magick::Draw.new
rect.stroke('black').stroke_width(1)
rect.fill(yuck)
rect.rectangle(340,380,237,110)
rect.draw(img)

tri = Magick::Draw.new
tri.stroke('black').stroke_width(1)
tri.fill(bleah)
tri.polygon(90,320,160,370,390,120)
tri.draw(img)

img = img.quantize(256,Magick::GRAYColorspace)

img.write("drawing.gif")
```

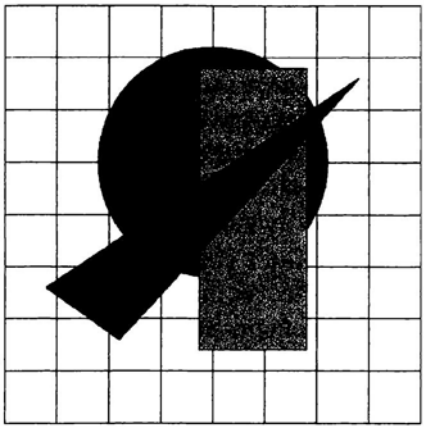


Рис. 15.3. Простая программа рисования

Рассмотрим эту программу подробнее. Сначала мы создаем «пустое» изображение методом `ImageList.new`, а потом вызываем для возвращенного объекта метод `new_image`. Можно считать, что мы получили «чистый холст» заданного размера (500×500 пикселей).

Для удобства определим несколько цветов с понятными именами, например: `purplish` и `yuck`. Цвета определяются так же, как в HTML. Базовая библиотека ImageMagick сама распознает много названий цветов, например, «red» и «black»; если сомневаетесь, пробуйте, или задавайте цвета в шестнадцатеричном виде.

Затем мы создаем объект рисования `line`; это объект Ruby, соответствующий графическому объекту, который мы видим на экране. Переменную иногда называют `gc` или как-то похоже (от «graphics context» – графический контекст), но нам кажется естественным употребить имя, отражающее природу объекта.

Далее вызывается метод `line` объекта рисования, по два раза на каждой итерации цикла. Взглянув на то, как изменяются координаты, вы поймете, что на каждой итерации рисуется одна горизонтальная и одна вертикальная прямая.

После каждого обращения к `line` мы вызываем метод `draw` того же объекта и передаем ему ссылку на изображение. Именно на этом шаге графический объект помещается на холст.

Лично меня обращения вида `shape.draw(image)` немного путают. В общем случае вызов любого метода выглядит так:

```
big_thing.operation(little_thing)
# Например: dog.wag(tail) (собака.вилять(хвост))
```

Но методы RMagick записываются, скорее, в виде:

```
little_thing.operation(big_thing)
# Продолжая аналогию: tail.wag(dog) (хвост.вилять(собака))
```

Впрочем, эта идиома достаточно распространена, особенно в программах рисования и графических интерфейсах. И в классической объектно-ориентированной идеологии это вполне оправдано: фигура знает, как нарисовать себя, а, стало быть, должна иметь метод `draw`. А он должен знать, где рисовать, поэтому ему нужно передать ссылку на холст (или что-то подобное).

Но, возможно, вас не мучает вопрос, от имени какого объекта следует вызывать метод. Тем лучше.

Покончив с сеткой, мы переходим к рисованию фигур. Метод `circle` принимает в качестве параметров центр окружности и какую-нибудь *точку* на ней (радиус не передается!). Метод `rectangle` еще проще; для рисования прямоугольника нужно задать координаты левого верхнего угла (первые два параметра) и координаты правого нижнего угла (последние два параметра). Треугольник же является частным случаем многоугольника, мы задаем координаты всех его вершин, а замыкающий отрезок (из конечной точки в начальную) рисуется автоматически.

У каждого графического объекта есть еще несколько методов. Взгляните на этот «сцепленный» вызов:

```
shape.stroke('black').stroke_width(1)
```

Это что-то вроде пера, которое рисует черными чернилами линию толщиной в один пиксель. Цвет обводки часто имеет значение, особенно если мы хотим закрасить фигуру.

Конечно, у каждой из трех этих фигур есть еще метод `fill`, при вызове которого указывается цвет заливки. (Имеются также более сложные способы заливки, например, штриховкой, с наложением тени и т.д.) Метод `fill` заменяет цвет внутренних пикселей фигуры указанным, ориентируясь на цвет границы, чтобы отличить внутреннюю часть от внешней.

API рисования содержит также методы для задания полупрозрачности, пространственных преобразований и многого другого. Есть методы для анализа, рисования и манипулирования текстовыми строками. Существует даже специальный RVG API (Ruby Vector Graphics – векторная графика в Ruby), совместимый с рекомендацией консорциума W3C по масштабируемой векторной графике (SVG).

Мы не можем привести здесь документацию по всем этим бесчисленным возможностям. Дополнительную информацию см. на сайте http://www.imagemagick.org/RMagick/doc/.

15.5. Создание документов в формате PDF с помощью библиотеки Prawn

Формат Portable Document Format (PDF), первоначально получивший известность благодаря программе Adobe Acrobat Reader, уже много лет популярен. Он доказал свою полезность для распространения «готовых к печати» качественно

отформатированных документов, независимых от программного обеспечения обработки текстов и операционной системы.

В Ruby для создания PDF-документов чаще всего применяется библиотека Prawn, созданная Грегори Брауном и многими другими авторами. Мы рассмотрим, как работает Prawn, а затем воспользуемся ей для создания PDF-документа.

15.5.1. Основные концепции и приемы

В основе Prawn лежит класс `Prawn::Document`, который позволяет создавать PDF-документы, применяя два или три стиля кодирования, например:

```
require "prawn"

doc = Prawn::Document.new           # Начать новый документ
doc.text "Lorem ipsum dolor..."     # Добавить текст
doc.render_file "my_first.pdf"      # Записать в файл
```

Если задать параметр блока, то объект-генератор передается явно:

```
Prawn::Document.generate("ericblair.pdf") do |doc|
   doc.text "It was a bright cold day in April, "
   doc.text "and the clocks were striking thirteen."
end
```

Эти формы по существу эквивалентны. Какую выбрать, зависит от личных предпочтений и удобства в конкретной ситуации.

Теперь поговорим о *страничных координатах*. Начало координат в PDF находится в левом нижнем углу страницы, как принято в математике. *Ограничивающий прямоугольник* – это воображаемый прямоугольник внутри страницы. Существует один ограничивающий прямоугольник по умолчанию, который называется *граничным прямоугольником* (margin box) и служит контейнером для всего, что находится на странице.

Начальное положение *курсора* в Prawn находится в верхней части страницы, несмотря на то, что начало координат располагается внизу. При добавлении текста в документ курсор сдвигается (его можно сдвинуть и вручную с помощью таких методов, как `move_cursor_to`, `move_up` и `move_down`). Метод `cursor` возвращает текущую позицию курсора.

«Стандартной» единицей измерения в Prawn является *пункт* (1/72 дюйма). Загрузив пакет `'prawn/measurement_extensions'`, вы сможете использовать и другие единицы измерения.

Prawn располагает большим набором процедур обработки текста, а также графическими примитивами для рисования прямых и кривых линий и геометрических фигур.

15.5.2. Пример документа

Рассмотрим несколько искусственный пример. Программа, показанная в листинге 15.8, делит страницу на четыре прямоугольника и каждый заполняет по-разному. Результат показан на рис. 15.4.

Листинг 15.8. pdf-demo.rb

```
require 'prawn'

# На основе кода, предложенного Брэдом Эдигером

class DemoDocument
   def initialize
      @pdf = Prawn::Document.new
   end

   def render_file(file)
      render
      @pdf.render_file(file)
   end

   def render
      side = @pdf.bounds.width / 2.0
      box(0,    0,    side, side) { star }
      box(side, 0,    side, side) { heart }
      box(0,    side, side, side) { ruby }
      box(side, side, side, side) { misc_text }
   end

   private

   # Выполнить указанный блок в ограничивающем прямоугольнике, отделенном
   # от родителя отступом размером 'padding' пунктов.
   def inset(padding)
      left = @pdf.bounds.left + padding
      top = @pdf.bounds.top - padding
      @pdf.bounding_box([left, top],
         width: @pdf.bounds.width - 2*padding,
         height: @pdf.bounds.height - 2*padding) { yield }
      end

   # Нарисовать прямоугольник заданного размера с левым верхним углом в точке
   # (x, y) и вызвать yield, чтобы можно было выполнить в нем рисование.
   def box(x, y, w, h)
      @pdf.bounding_box([x, @pdf.bounds.top - y], width: w, height: h) do
         @pdf.stroke_bounds
         inset(10) { yield }
      end
   end

   def star
      reps = 15
      size = 0.24 * @pdf.bounds.width
      radius = 0.26 * @pdf.bounds.width
```

```
    center_x = @pdf.bounds.width / 2.0
    center_y = @pdf.bounds.height / 2.0
    reps.times do |i|
      @pdf.rotate i * 360.0 / reps, origin: [center_x, center_y] do
        edge = center_y + radius
        @pdf.draw_text ")", size: size, at: [center_x, edge]
      end
    end
end

def ruby
    @pdf.image "ruby.png",
             at: [0, @pdf.cursor],
          width: @pdf.bounds.width,
           height: @pdf.bounds.height
end

def heart
    10.times do |i|
      inset(i*10) do
        box = @pdf.bounds
        center = box.width / 2.0
        cusp_y = 0.6 * box.top

        k = center * Prawn::Graphics::KAPPA
        @pdf.stroke_color(0, 0, 0, 100-(i*10))
        @pdf.stroke do
          # Нарисовать сердце, состоящее из двух кривых Безье
          paths = [[0, 0.9*center], [box.right, 1.1*center]]
          paths.each do |side, midside|
            @pdf.move_to [center, cusp_y]
            @pdf.curve_to [side, cusp_y],
              bounds: [[center, cusp_y + k], [side, cusp_y + k]]
            @pdf.curve_to [center, box.bottom],
              bounds: [[side, 0.6 * cusp_y], [midside, box.bottom]]
          end
        end
      end
    end

    # переустановить цвет обводки
    @pdf.stroke_color 0, 0, 0, 100
end

def misc_text
    first_lines = <<-EOF
      Call me Ishmael. Somewhere in la Mancha, in a place whose
      name I do not care to remember, a gentleman lived not long
```

```
        ago, one of those who has a lance and ancient shield on a
        shelf and keeps a skinny nag and a greyhound for racing.
        The sky above the port was the color of television, tuned to
        a dead channel. It was a pleasure to burn. Granted: I am an
        inmate of a mental hospital; my keeper is watching me, he
        never lets me out of his sight; there's a peephole in the
        door, and my keeper's eye is the shade of brown that can
        never see through a blue-eyed type like me. Whether I shall
        turn out to be the hero of my own life, or whether that
        station will be held by anybody else, these pages must show.
        I have never begun a novel with more misgiving.
      EOF
      first_lines.gsub!(/\n/, " ")
      first_lines.gsub!(/ +/, " ")
      @pdf.text first_lines
    end
end

DemoDocument.new.render_file("demo.pdf")
```

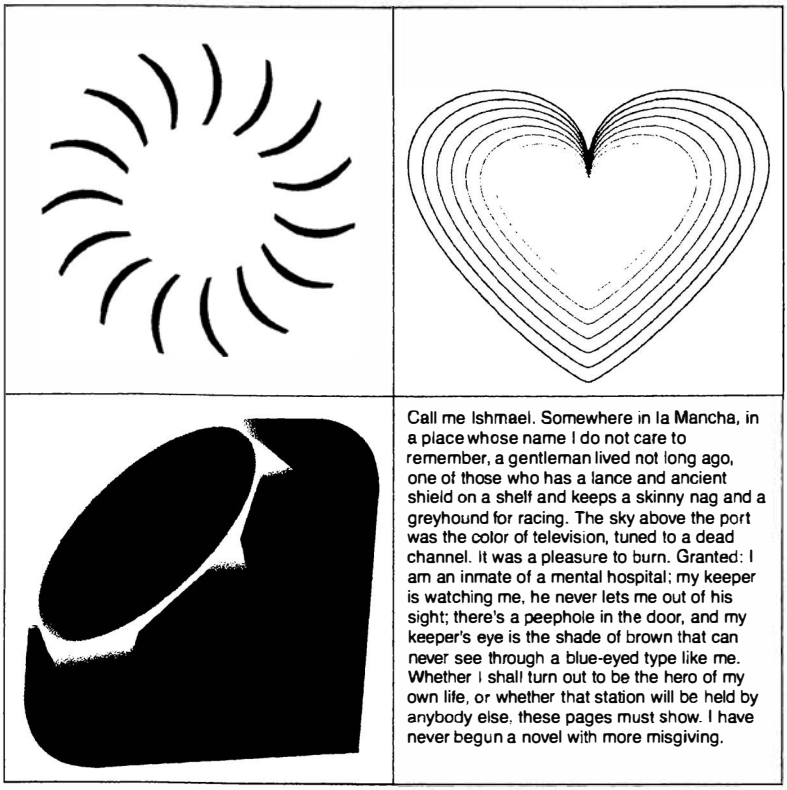


Рис. 15.4. Результат работы pdf-demo.rb

В двух верхних квадрантах демонстрируется использование API рисования. В левом верхнем нарисована звездообразная фигура, состоящая из последовательности дуг одинакового радиуса, но с разными центрами. В правом верхнем мы видим более сложный рисунок сердца, составленного из кривых Безье.

В левый нижний квадрант мы поместили PNG-изображение, воспользовавшись методом `image`. Наконец, в правом нижнем квадранте размещен текст, не выходящий за пределы ограничивающего прямоугольника.

Но это лишь малая толика возможностей Prawn API. Его полное описание можно найти в справочной документации по адресу prawnpdf.org.

15.6. Заключение

В этой главе мы рассмотрели разбор текста в формате JSON и операции с ним, а также использование библиотеки Nokogiri для разбора XML-документов двумя способами: потоковым и документо-ориентированным. Мы также видели, как с помощью библиотеки `rss` разбирать новостные ленты в форматах RSS и Atom, основанных на XML.

Мы также видели, как можно читать и обрабатывать графические изображения разного формата с помощью библиотеки RMagick. Рассмотрели мы и API рисования, позволяющий включать в изображение произвольный текст и геометрические фигуры. Наконец, мы показали, как с помощью библиотеки PDF::Writer можно программно создавать сложные PDF-документы высокого качества.

Следующая глава посвящена совсем другой теме. Речь пойдет об эффективном тестировании и отладке написанных на Ruby программ.

Глава 16. Тестирование и отладка

Неполадки в блоке AE-35. В ближайшие семьдесят два часа блок может отказать.

А. Кларк «Космическая Одиссея 2001 года»

Тестирование – вещь важная. Все компетентные программисты об этом знают, хотя не всегда этот вопрос стоит для них на первом месте.

Конечно, исчерпывающее тестирование, как правило, невозможно. Программа сколько-нибудь заметного размера на протяжении своего жизненного цикла обязательно преподнесет сюрпризы. Максимум, что мы можем сделать, – тестировать тщательно и избирательно, стараясь проверить как можно больше.

Исторически сложилось так, что программисты не всегда тестируют, как положено. Объясняют это обычно тем, что тесты трудно готовить и прогонять, что вся процедура требует ручного вмешательства или что это отнимает слишком много времени.

В 1990-е годы в сообществе программистов стала распространяться «культура тестирования». Термин *экстремальное программирование* сейчас уже вышел из моды, но стоящие за ним идеи по-прежнему актуальны. Концепции *разработки через тестирование* (TDD) и *разработки, управляемой поведением* (BDD), прочно овладели умами разработчиков по всему миру.

Являетесь ли вы твердокаменным приверженцем идеологии «тестируй с самого начала», не так существенно. Важно, что *любой* человек может воспользоваться инструментами, которые позволяют автоматизировать тестирование, упростив написание и прогон тестов.

Инструменты тестирования писать на Ruby проще в силу динамичности и гибкости языка. Не менее легко и (посмею ли сказать?) *приятно* ими пользоваться. Внес изменение в программу, а потом смотришь, как все тесты успешно доходят до конца, – положительно в этом что-то есть.

Помимо этих инструментов, в Ruby есть еще немало программ и библиотек для отладки, профилирования и испытания различных путей исполнения. Эта глава посвящена обзору имеющихся средств.

Мы начнем с очень популярного инструмента RSpec, а затем рассмотрим программу Minitest, предлагающую классический подход к тестированию. Кроме того, мы поговорим о средстве приемочного тестирования Cucumber и дадим беглый обзор других инструментов.

16.1. Тестирование с помощью RSpec

В последние годы широкое распространение получила идея разработки, управляемой поведением (behavior-driven development BDD): спецификация поведения системы и есть естественная основа для ее тестирования.

RSpec, плод трудов Дэвида Хелимски (David Chelimsky) и других, пожалуй, является самым употребительным каркасом BDD для Ruby. При его проектировании за образец была взята система JBehave для Java.

Для иллюстрации этого и прочих каркасов тестирования возьмем показанный ниже фрагмент кода (надеюсь, курс математики за 7 класс еще не полностью выветрился из вашей головы).

Метод quadratic (мы поместили его в файл quadratic.rb) решает квадратное уравнение. Он принимает три коэффициента квадратного трехчлена и возвращает массив, содержащий 0, 1 или 2 вещественных корня. В качестве дополнительной функциональности мы включили еще четвертый параметр – булев флаг, показывающий, разрешены ли комплексные корни:

```
require 'complex'

def quadratic(a, b, c, complex=false)
   raise ArgumentError unless [a, b, c].all? {|x| Numeric === x }
   discr = b*b - 4*a*c
   if (discr > 0) || (complex && discr < 0)
      val = Math.sqrt(discr)
      r1 = (-b + val)/(2*a.to_f)
      r2 = (-b - val)/(2*a.to_f)
      return [r1, r2]
   elsif discr.zero?
      r1 = -b/(2*a.to_f)
      return [r1]
   else                        # если complex равно false
      return []
   end

end
```

Для тестирования этого метода мы соберем сравнительно простой набор тестов RSpec, которые будем прогонять с помощью gem-пакета rspec. Обычно эти тесты находятся в каталоге spec, а имена содержащих их файлов оканчиваются суффиксом _spec и имеют расширение .rb.

Вообще говоря, один spec-файл тестирует какую-то часть функциональности. Как правило, такая часть соответствует одному классу, но может быть как меньше, так и больше класса.

Типичный spec-файл имеет такой вид:

```
# spec/foobar_spec.rb
require 'foobar'

RSpec.describe Foobar do
```

```
  describe "some_method" do
    it "returns something when called" do
      # Здесь должен быть тестовый код...
    end
  end
end
```

Установив gem-пакет rspec, мы можем выполнить этот spec-файл, запустив исполнитель:

```
$ rspec spec/foobar_spec.rb
```

Программа rspec читает параметры командной строки из файла .rspec. По умолчанию этот файл содержит только параметр --require spec_helper. Файл spec/spec_helper.rb – это место для настройки RSpec, а также общего кода, который понадобится во время тестирования. Его наличие необязательно.

Сам spec-файл должен затребовать класс или иной код, который предстоит протестировать. В наших примерах предполагается, что тестируемый код находится в каталоге lib. RSpec (или файл spec_helper) должен добавить каталог lib в переменную Ruby $LOAD_PATH, чтобы находящиеся там файлы можно было затребовать без указания полного пути, как в примере выше.

Метод describe принимает параметр, который обычно является именем класса, но может быть и описательной строкой. Блок содержит все тесты (и может также содержать вложенные блоки describe).

Теперь обратимся к тестам метода решения квадратного уравнения. Содержимое файла quadratic_spec.rb показано в листинге 16.1.

Листинг 16.1. Тестирование quadratic.rb с помощью RSpec

```
require 'quadratic'

describe "Quadratic equation solver" do
  it "can take integers as arguments" do
    expect(quadratic(1, 2, 1)).to eq([-1.0])
  end

  it "can take floats as arguments" do
    expect(quadratic(1.0, 2.0, 1.0)).to eq([-1.0])
  end

  it "returns an empty solution set when appropriate" do
    expect(quadratic(2, 3, 9)).to eq([])
  end

  it "honors the 'complex' Boolean argument" do
    solution = quadratic(1, -2, 2, true)
    expect(solution).to eq([Complex(1.0,1.0), Complex(1.0,-1.0)])
  end

  it "raises ArgumentError when for non-numeric coefficients" do
```

```
    expect { quadratic(3, 4, "foo") }.to raise_error(ArgumentError)
  end
end
```

Любой тест представляет собой вызов метода `it`; он назван так для того, чтобы получалась иллюзия предложения на английском языке, в котором «it» – подлежащее. Строковый параметр довершает эту иллюзию, а блок содержит код соответствующего теста.

Тест может содержать произвольный код, но обязательно должен включать хотя бы одно предложение, в котором производится проверка (технически оно называется *выражением ожидания*).

Типичное выражение ожидания состоит из метода `expect`, ожидания и сопоставителя. Существует два метода ожидания: `to` и `to_not`.

В самой распространенной форме такое выражение просто проверяет, что одно равно другому: `expect(result).to eq(value)`.

Сопоставителей существует много, и можно определять свои собственные. Ниже показаны наиболее употребительные (во всех случаях результат равен `true`):

```
expect(5).to eq(5)
expect(5).to_not eq(6)
expect(5).to_not be_nil
expect(5).to be_truthy
expect(5).to_not eq(false)
expect(nil).to be_nil
expect("").to be_empty   # "be_xxx" вызывает метод the "xxx?"
expect([]).to be_empty   # аргумента expect
expect([1,2,3]).to include(2)
expect([1,2]).to be_an(Array)
```

Для тестирования исключений код следует поместить в блок, который передается методу `expect`:

```
expect { x = 1 / 0 }.to raise_error
```

Метод `raise_error` может принимать в качестве параметра класс исключения и строковое значение. Оба параметра необязательны, обычно сравнение со строкой считается перебором.

RSpec этим отнюдь не ограничивается. Для более сложных тестов могут потребоваться методы `before`, `after` и `let` (для инициализации, очистки и прочих вещей). Эти и другие детали описаны в полном руководстве.

Понятно, что под капотом RSpec вершится разного рода «черная магия». Если по каким-то причинам этот каркас вам не по вкусу, присмотритесь к пакету `minitest`, основанному на более традиционном «автономном тестировании».

16.2. Тестирование с помощью Minitest

Gem-пакет `minitest`, написанный Райаном Дэвисом (Ryan Davis), позволяет тестировать Ruby-код в классическом автономном стиле. Вполне возможно, что он уже входит в ваш дистрибутив Ruby, а, если нет, установите его командой `gem install minitest`.

В классе Minitest::Test применяется отражение для анализа тестового кода. В его подклассах все методы с именами, начинающимися строкой test, исполняются как тесты:

```
require 'minitest/autorun'

class TestMyThing < Minitest::Test

   def test_that_it_works
      # ...
   end

   def test_it_doesnt_do_the_wrong_thing
      # ...
   end

   # ...
end
```

Категорически не рекомендуется и, пожалуй, даже неправильно при тестировании поведения опираться на порядок выполнения тестов. Не должно быть так, что тесты проходят, только если выполняются в том порядке, в каком написаны, – следите за этим. Minitest специально прогоняет тесты в случайном порядке при каждом запуске.

Кроме того, было бы неплохо сопровождать каждый тест однострочным комментарием, в котором описывается его назначение, особенно если тест сложный или содержит скрытые тонкости. В общем случае, у каждого теста должна быть одна и только одна цель.

Если для теста требуется определенная настройка окружения, то можно создать методы класса setup и teardown. Эти методы вызываются до и после каждого прогона *каждого* теста, хотя интуитивно это, возможно, и неочевидно.

А если настройка занимает много времени? Было бы непрактично выполнять ее для каждого теста, но и поместить ее в первый тестовый метод тоже нельзя (потому что тесты прогоняются в случайном порядке). Чтобы произвести настройку только один раз, перед всеми тестами, можно поместить соответствующий код в тело класса, до тестовых методов (или даже после самого класса).

А как быть, если требуется произвести очистку после выполнения все тестов? «Самый лучший» способ – переопределить метод класса run, включив в него эту функциональность. И раз уж мы заговорили об этом, добавим метод класса, выполняющий настройку до прогона тестов. См. пример в листинге 16.2.

Листинг 16.2. Инициализация и очистка

```
require 'minitest/autorun'

class MyTest < Minitest::Test

   def self.setup
```

```
    # ...
  end

  def self.teardown
    # ...
  end

  def self.run(*)
    self.setup
    super   # выполнить каждый тестовый метод, как обычно
    self.teardown
  end

  def setup
    # ...
  end

  def teardown
    # ...
  end

  def test_it_works
    # ...
  end

  def test_it_is_not_broken
    # ...
  end

  # ...
end
```

Вряд ли вам придется прибегать к этой технике часто, но в тех случаях, когда это необходимо, она исключительно полезна.

Ну а что же поместить внутрь каждого теста? Нужен какой-то способ определить, завершился тест успешно или неудачно. Для этого предусмотрены *утверждения*.

Простейшее утверждение – метод `assert`. Его первый параметр – подлежащее проверке выражение, а необязательный второй параметр – сообщение. Если значением первого параметра является `true` (т. е. все, кроме `false` и `nil`), значит, все хорошо. В противном случае считается, что тест не прошел, и печатается сообщение (если оно было задано).

Ниже приведены другие методы утверждения (в комментариях описано, для чего они нужны). Отметим, что «ожидаемое» значение всегда предшествует «фактическому»; это существенно, если вы пользуетесь сообщениями об ошибках, подразумеваемыми по умолчанию, и не хотите, чтобы их смысл был прямо противоположен реальному положению вещей:

```
assert_equal(expected, actual)     # assert(expected == actual)
refute_equal(expected, actual)     # assert(expected != actual)
assert_match(regex, string)        # assert(regex =~ string)
refute_match(regex, string)        # assert(regex !~ string)
assert_nil(object)                 # assert(object.nil?)
refute_nil(object)                 # assert(!object.nil?)
```

Некоторые утверждения связаны с объектной ориентированностью (и для них существуют противоположные варианты, начинающиеся словом refute – опровергать):

```
assert_instance_of(klass, obj)     # assert(obj.instance_of? klass)
assert_kind_of(klass, obj)         # assert(obj.kind_of? klass)
assert_respond_to(obj, meth)       # assert(obj.respond_to? meth)
```

Некоторые методы относятся к исключениям и символам, обозначающим возбуждаемое исключение. У них нет варианта refute, потому что если предложения throw и raise не встречались, то код просто выполняется, как обычно. Поскольку если эти предложения все же встречаются, то код не может выполняться, как обычно, оба метода принимают блок:

```
assert_throws(symbol) { ... }      # проверяет, что возбуждено исключение,
                                   # обозначаемое указанным символом
assert_raises(exception) { ... }   # проверяет, что возбуждено исключение
```

Для коллекций и других объектов типа Enumerable определено несколько специальных методов:

```
assert_empty(coll)                 # assert(coll.empty?)
assert_includes(coll, obj)         # assert(coll.includes?(obj))
```

Есть и другие, но этих хватает для большинства потребностей. Дополнительные сведения смотрите в онлайновой документации по Minitest.

Существует также метод flunk, который всегда завершается неудачно. Это просто заглушка для еще ненаписанных тестов.

Вооружившись знаниями об утверждениях, протестируем наш метод решения квадратных уравнений. Тестовые файлы поместим в каталог test. Соглашение об именовании аналогично используемому в RSpec: имя файла должно завершаться строкой _test.rb (см. листинг 16.3).

Листинг 16.3. Тестирование файла quadratic.rb с помощью Minitest::Test

```
require 'minitest/autorun'
require 'quadratic'

class QuadraticTest < MiniTest::Unit::TestCase
   def test_integers
      assert_equal [-1], quadratic(1, 2, 1)
   end

   def test_floats
      assert_equal [-1.0], quadratic(1.0, 2.0, 1.0)
```

```
    end
    def test_no_real_solutions
        assert_equal quadratic(2, 3, 9), []
    end

    def test_complex_solutions
        actual = quadratic(1, -2, 2, true)
        assert_equal actual, [Complex(1.0,1.0), Complex(1.0,-1.0)]
    end

    def test_bad_args
        assert_raises(ArgumentError) { quadratic(3, 4, "foo") }
    end
end
```

Если запустить тестовый файл, не предпринимая никаких специальных мер, то по умолчанию будет выполнен консольный исполнитель тестов, подразумеваемый по умолчанию. Это вызовет ностальгическое воспоминание о старой доброй технологии 1970-х годов – Minitest выводит сообщения такого вида:

```
Run options: -seed 7759

# Running tests:



Finished tests in 0.000702s, 7122.5071 tests/s,
       7122.5071 assertions/s.

5 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

Если вы поклонник RSpec, то Minitest можно использовать аналогично. Основное отличие заключается в том, что в RSpec применяется форма `expect(subject).to something`, а в Minitest – более прямолинейная форма `subject.must_something`. Пример приведен в листинге 16.4.

Листинг 16.4. Тестирование файла quadratic.rb с помощью Minitest::Spec

```
require 'minitest/autorun'
require 'quadratic'

describe "Quadratic equation solver" do
    it "can take integers as arguments" do
        quadratic(1, 2, 1).must_equal([-1.0])
    end

    it "can take floats as arguments" do
        quadratic(1.0, 2.0, 1.0).must_equal([-1.0])
    end

    it "returns an empty solution set when appropriate" do
```

```
      quadratic(2, 3, 9).must_equal([])
   end

   it "honors the 'complex' Boolean argument" do
      actual = quadratic(1, -2, 2, true)
      expected = [Complex(1.0,1.0), Complex(1.0,-1.0)]
      actual.must_equal expected
   end

   it "raises ArgumentError when for non-numeric coefficients" do
      lambda { quadratic(3, 4, "foo") }.must_raise ArgumentError
   end
end
```

Прогон этих тестов в стиле спецификаций дает те же результаты, что и в предыдущем случае.

Помимо всего вышеизложенного, существуют десятки расширений Minitest, которые добавляют или изменяют функциональность. Некоторые из них описаны в прилагаемом к Minitest файле readme, другие можете поискать на сайте Rubygems.org или GitHub.

16.3. Тестирование с помощью Cucumber

Cucumber – еще один каркас для BDD-тестирования. В нем используется очень простой и гибкий «язык», или нотация под названием Gherkin, близкий к разговорному английскому.

В каких ситуациях использовать Cucumber, дело вкуса. Он увеличивает накладные расходы и требует дополнительного сопровождения тестов, зато объясняет, что именно тестируется, на языке, понятном заказчикам или руководителям, не имеющим технической подготовки. Идеальный случай – когда в написании спецификаций принимают участие стороны, не обремененные техническими познаниями.

В других случаях, особенно когда все участники технически подкованы, гораздо проще писать тесты (включая интеграционные и приемочные) прямо в виде кода.

Вряд ли пример с решением квадратного уравнения здесь уместен, но все же рассмотрим его. Для начала создадим каталог `features` и в нем подкаталог `step_definitions`. В каталоге `features` может находиться много файлов, в листинге 16.5 показан только один из них – файл `first.feature`.

Листинг 16.5. Файл описания функции для quadratic.rb

```
Feature: Check all behaviors of the quadratic equation solver

Scenario: Real roots
   Given coefficients that should yield one real root
   Then the solver returns an array of one Float r1
   And r1 is a valid solution.
```

```
    Given coefficients that should yield two real roots
    Then the solver returns an array of two Floats, r1 and r2
    And r1 and r2 are both valid solutions.

Scenario: No real roots, and complex flag is off
    Given coefficients that should yield no real roots
    And the complex flag is missing or false
    Then the solver returns an empty array.

Scenario: No real roots, and complex flag is on
    Given coefficients that should yield no real roots
    And the complex flag is true
    Then the solver returns an array of two complex roots, r1 and r2
    And r1 and r2 are both valid solutions.
```

Если теперь запустить Cucumber, то он сделает разумную и полезную вещь: создаст «заготовку», которой можно воспользоваться для написания *определений шагов*, т. е. реализаций на Ruby тестов, описанных в виде спецификации на английском языке. Эту заготовку можно скопировать в файл (в данном случае step_definitions/first.rb) и модифицировать его по ходу дела. Тесты помечены признаком pending, чтобы они не «падали» из-за того, что код еще не написан.

Вот пример созданной заготовки:

```
Given(/^coefficients that should yield one real root$/) do
    pending # express the regexp above with the code you wish you had
end

Then(/^the solver returns an array of one Float r(\d+)$/) do |arg1|
    pending # express the regexp above with the code you wish you had
end

Then(/^r(\d+) is a valid solution\.$/) do |arg1|
    pending # express the regexp above with the code you wish you had
end
```

Обратите внимание на то, как используются регулярные выражения, чтобы связать англоязычную фразу (на языке Gherkin) с соответствующим Ruby-кодом. Ниже показано, как я наполнил эту заготовку реальным кодом (в файле first.rb):

```
Given(/^coefficients that should yield one real root$/) do
    @result = quadratic(1, 2, 1)
end

Then(/^the solver returns an array of one Float r(\d+)$/) do |arg1|
    expect(@result.size).to eq(1)
    expect(@result.first).to be_a(Float)
end

Then(/^r(\d+) is a valid solution\.$/) do |arg1|
    expect(@result).to eq([-1.0])
end
```

Теперь эти три простеньких теста проходят. Написав аналогичным образом определения остальных шагов, мы получим спецификацию, которая, с одной стороны, написана на понятном английском языке, а, с другой, пригодна для проверки правильности кода.

Cucumber – очень мощная система, обладающая множеством полезных функций. Более полное описание можно найти в онлайновой документации, а также в книге Matt Wynne, Aslak Hellesoy «The Cucumber Book».

16.4. Работа с отладчиком byebug

Самая популярная библиотека для отладки программ на Ruby – gem-пакет `byebug`. Он реализован на основе механизма `TracePoint`, рассмотренного в разделе 11.4.4 главы 11 «ООП и динамические механизмы в Ruby». В версиях Ruby младше 2.0 аналогичную функциональность предлагал gem-пакет `debugger`.

Честно говоря, применение отладчиков самих по себе мало что дает в Ruby. Лично я использую Byebug посредством инструмента `pry`, который мы рассмотрим в следующем разделе. Но давайте сначала поглядим, как работать с отладчиком, а затем уже перейдем к Pry и имеющимся в нем дополнительным возможностям.

Чтобы запустить Byebug, просто введите в командной строке слово `byebug` вместо `ruby`:

```
byebug myfile.rb
```

Получив приглашение `(byebug)`, можно вводить команды, например, `step` для входа внутрь метода, `list` для печати текста всей программы или ее части и т. д. Ниже перечислены наиболее употребительные команды:

- **continue (c)** – продолжить выполнение до указанной строки;
- **break (b)** – поставить точку останова или получить их список;
- **delete** – удалить некоторые или все точки останова;
- **catch** – поставить точку перехвата или получить их список;
- **step (s)** – войти внутрь метода;
- **next (n)** – дойти до следующей строки (выполнить метод без захода внутрь);
- **where (w)** – напечатать трассировку текущего состояния стека;
- **help (h)** – получить справку (напечатать список всех команд);
- **quit (q)** – выйти из отладчика.

В листинге 16.6 приведен код простой программы (по правде говоря, слишком простой, чтобы ее отлаживать).

Листинг 16.6. Простая программа для демонстрации отладки

```
STDOUT.sync = true

def palindrome?(word)
   word == word.reverse
end
```

```
def signature(w)
   w.split("").sort.join
end

def anagrams?(w1,w2)
   signature(w1) == signature(w2)
end

print "Введите слово: "
w1 = gets.chomp

print "Введите еще одно слово: "
w2 = gets.chomp

verb = palindrome?(w1) ? "является" : "не является"
puts "'#{w1}' #{verb} палиндромом."

verb = palindrome?(w2) ? "является" : "не является"
puts "'#{w2}' #{verb} палиндромом."

verb = anagrams?(w1,w2) ? "являются" : "не являются"
puts "'#{w1}' и '#{w2}' #{verb} анаграммами."
```

В листинге 16.7 показан весь сеанс отладки, хотя некоторые строки программы ради краткости опущены. Читая распечатку, помните, что консоль используется как для отладки, так и для ввода-вывода.

Листинг 16.7. Простой сеанс отладки

```
$ byebug simple.rb

[1, 10] in simple.rb
=> 1: STDOUT.sync = true

(byebug) b Object#palindrome?
Created breakpoint 1 at Object::palindrome?
(byebug) b Object#anagrams?
Created breakpoint 2 at Object::anagrams?
(byebug) info b
Num Enb What
1   y   at Object:palindrome?
2   y   at Object:anagrams?
(byebug) c 16
Введите слово:
Stopped by breakpoint 3 at simple.rb:16

[11, 20] in simple.rb
   15:     print "Введите слово: "
=> 16:     w1 = gets.chomp
```

```
   17:

(byebug) c 19
live
Введите еще одно слово:
[14, 23] in simple.rb
   18:     print "Введите еще одно слово: "
=> 19:     w2 = gets.chomp

(byebug) n
evil

[16, 25] in simple.rb
=> 21:     verb = palindrome?(w1) ? "является" : "не является"
   22:     puts "'#{w1}' #{verb} палиндромом."

(byebug) c
Stopped by breakpoint 1 at Object:palindrome?

[1, 10] in simple.rb
=> 3:      def palindrome?(word)
   4:        word == word.reverse

(byebug)

[1, 10] in simple.rb
   3:     def palindrome?(word)
=> 4:       word == word.reverse
   5:     end

(byebug)
Next went up a frame because previous frame finished

[17, 26] in simple.rb
   21:     verb = palindrome?(w1) ? "является" : "не является"
=> 22:     puts "'#{w1}' #{verb} палиндромом."

(byebug) c
'live' не является палиндромом.
'evil' не является палиндромом.

Stopped by breakpoint 2 at Object:anagrams?

[6, 15] in simple.rb
=> 11:     def anagrams?(w1,w2)
   12:       signature(w1) == signature(w2)
   13:     end

(byebug) c
'live' и 'evil' являются анаграммами.
```

Отметим, что последнюю команду можно повторить, просто нажав клавишу Enter. Имейте в виду, что если программа, исполняемая под управлением отладчика, затребует библиотеки, то в начале, возможно, придется «перешагнуть» довольно много предложений. Я рекомендую сразу перейти к интересующей вас строке, указав в команде `continue` ее номер.

Отладчик распознает еще много команд. Можно распечатать стек вызовов и перемещаться по нему в обоих направлениях. Можно «наблюдать» за выражением и автоматически останавливать программу, когда его значение изменяется. Можно добавлять выражения в список «показа». Можно отлаживать многопоточную программу и переключаться между потоками.

Все эти возможности довольно плохо документированы. Я рекомендую при работе с отладчиком почаще пользоваться командой `help` и продвигаться методом проб и ошибок.

Существуют также графические отладчики. Если они вас интересуют, обратитесь к главе 21 «Инструменты разработки для Ruby», где обсуждаются интегрированные среды разработки (IDE).

16.5. Отладка с помощью pry

Всякий программист на Ruby знаком с программой irb. Но многие считают, что недавно появившийся инструмент `pry` гораздо лучше. Для установки Pry добавьте его в Gemfile проекта или выполните команду `gem install pry`.

Как и irb, `pry` основан на принципе REPL (read-eval-print-loop – цикл чтения-вычисления-печати). Однако он включает ряд дополнительных возможностей, а также систему расширений, которая позволяет любому желающему подключать свои модули. Существуют расширения, которые интегрируют `pry` с `byebug` и `debugger`, так что вы получаете всю функциональность отладчика плюс улучшенный механизм REPL, встроенный в `pry`.

Рассмотрим, как можно остановить работающую программу и исследовать ее состояние в отладчике. Внесем в листинг 16.6 два изменения: в начале добавим `require 'pry'`, а где-то ниже обращение к методу `binding.pry`:

```
require 'pry'

# ...
w2 = gets.chomp

binding.pry

# ...
```

Вот несколько наиболее важных команд (это выдержка из оперативной справки):

```
cd          Сменить контекст (объект или область видимости).
find-method Рекурсивно искать метод в классе, модуле или в текущем пространстве имен.
ls          Показать список переменных и методов в текущей области видимости.
```

```
whereami    Показать код вокруг текущего контекста.
wtf?        Показать обратную трассировку последнего исключения.
show-doc    Вывести документацию по методу или классу.
show-source Показать исходный код метода или класса.
reload-code Перезагрузить исходный файл, содержащий код указанного объекта.
```

А теперь запустим. Ниже показано, как мы входим в среду pry и оглядываемся вокруг:

```
$ ruby myprog.rb
Введите слово: parental
Введите еще одно слово: prenatal

From: /Users/Hal/rubyway/ch16/myprog.rb @ line 23 :

       18:      w1 = gets.chomp
       19:
       20:      print "Введите слово: "
       21:      w2 = gets.chomp
       22:
    => 23:      binding.pry
       24:
       25:     verb = palindrome?(w1) ? "является" : "не является"
       26:     puts "'#{w1}' #{verb} палиндромом."
       27:
       28:     verb = palindrome?(w2) ? "является" : "не является"

[1] pry(main)> w1
=> "parental"
[2] pry(main)> w2
=> "prenatal"
[3] pry(main)> palindrome?(w1)
=> false
[4] pry(main)> anagrams?(w1, w2)
=> true
[5] pry(main)> exit
'parental' не является палиндромом.
'prenatal' не является палиндромом.
'parental' и 'prenatal' являются анаграммами.
```

Хотя сама программа pry не является отладчиком в традиционном смысле слова, в нее можно встроить отладчик, и она обладает весьма интересными функциями, которых у обычных отладчиков нет. Дополнительные сведения об этих функциях можно найти в главе 21 (и, как всегда, в актуальной онлайновой документации).

16.6. Измерение производительности

Я не люблю уделять слишком много внимания оптимизации скорости. В общем случае нужно правильно выбрать алгоритм и придерживаться здравого смысла.

Конечно, быстродействие имеет значение. Иногда даже очень большое. Однако начинать думать об этом на *слишком раннем* этапе цикла разработки – ошибка.

Как говорится, «преждевременная оптимизация – источник всех зол». (Эту мысль впервые высказал Хоар, а потом подтвердил Кнут.) Или, перефразируя, «сначала пусть работает правильно, а уж потом быстро». На уровне отдельного приложения эта рекомендация обычно оказывается хорошим эвристическим правилом (хотя для больших систем, она, быть может, и не так актуальна).

Я бы еще добавил: «*Не оптимизируйте, пока не измерите*».

Это не такое уж серьезное ограничение. Просто не приступайте к переработке ради скорости, пока не ответите на два вопроса: «Действительно ли программа работает медленно? Какие именно ее части снижают производительность?»

Второй вопрос важнее, чем кажется на первый взгляд. Программисты часто уверены, что и так знают, на что программа тратит большую часть времени, но специальные исследования убедительно свидетельствуют о том, что в среднем эти догадки имеют очень мало общего с действительностью. «Теоретическая» оптимизация для большинства из нас – плохая идея.

Нам нужны объективные измерения. Профилировщик нужен.

В комплект поставки Ruby входит профилировщик `profile`. Для его вызова достаточно включить библиотеку:

```
ruby -rprofile myprog.rb
```

Рассмотрим программу в листинге 16.6. Она открывает файл `/usr/share/dict/words` и ищет в нем анаграммы. Затем смотрит, у каких слов наибольшее количество анаграмм и распечатывает их.

Листинг 16.8. Поиск анаграмм в словаре

```
words = File.readlines("/usr/share/dict/words")
words.map! {|x| x.chomp }

hash = {}
words.each do  word|
  key = word.split("").sort.join
  hash[key] ||= []
  hash[key] << word
end

sizes = hash.values.map {|v| v.size }
most = sizes.max
list = hash.find_all {|k,v| v.size == most }

puts "Ни у одного слова нет более #{most-1} анаграмм."
list.each do |key,val|
  anagrams = val.sort
  first = anagrams.shift
  puts "Слово #{first} имеет #{most-1} анаграмм:"
  anagrams.each {|a| puts "   #{a}" }
end

num = 0
```

```
whereami    Показать код вокруг текущего контекста.
wtf?        Показать обратную трассировку последнего исключения.
show-doc    Вывести документацию по методу или классу.
show-source Показать исходный код метода или класса.
reload-code Перезагрузить исходный файл, содержащий код указанного объекта.
```

А теперь запустим. Ниже показано, как мы входим в среду pry и оглядываемся вокруг:

```
$ ruby myprog.rb
Введите слово: parental
Введите еще одно слово: prenatal

From: /Users/Hal/rubyway/ch16/myprog.rb @ line 23 :

       18:      w1 = gets.chomp
       19:
       20:      print "Введите слово: "
       21:      w2 = gets.chomp
       22:
    => 23:      binding.pry
       24:
       25:     verb = palindrome?(w1) ? "является" : "не является"
       26:     puts "'#{w1}' #{verb} палиндромом."
       27:
       28:     verb = palindrome?(w2) ? "является" : "не является"

[1] pry(main)> w1
=> "parental"
[2] pry(main)> w2
=> "prenatal"
[3] pry(main)> palindrome?(w1)
=> false
[4] pry(main)> anagrams?(w1, w2)
=> true
[5] pry(main)> exit
'parental' не является палиндромом.
'prenatal' не является палиндромом.
'parental' и 'prenatal' являются анаграммами.
```

Хотя сама программа pry не является отладчиком в традиционном смысле слова, в нее можно встроить отладчик, и она обладает весьма интересными функциями, которых у обычных отладчиков нет. Дополнительные сведения об этих функциях можно найти в главе 21 (и, как всегда, в актуальной онлайновой документации).

16.6. Измерение производительности

Я не люблю уделять слишком много внимания оптимизации скорости. В общем случае нужно правильно выбрать алгоритм и придерживаться здравого смысла.

Конечно, быстродействие имеет значение. Иногда даже очень большое. Однако начинать думать об этом на *слишком раннем* этапе цикла разработки – ошибка.

Как говорится, «преждевременная оптимизация – источник всех зол». (Эту мысль впервые высказал Хоар, а потом подтвердил Кнут.) Или, перефразируя, «сначала пусть работает правильно, а уж потом быстро». На уровне отдельного приложения эта рекомендация обычно оказывается хорошим эвристическим правилом (хотя для больших систем, она, быть может, и не так актуальна).

Я бы еще добавил: «*Не оптимизируйте, пока не измерите*».

Это не такое уж серьезное ограничение. Просто не приступайте к переработке ради скорости, пока не ответите на два вопроса: «Действительно ли программа работает медленно? Какие именно ее части снижают производительность?»

Второй вопрос важнее, чем кажется на первый взгляд. Программисты часто уверены, что и так знают, на что программа тратит большую часть времени, но специальные исследования убедительно свидетельствуют о том, что в среднем эти догадки имеют очень мало общего с действительностью. «Теоретическая» оптимизация для большинства из нас – плохая идея.

Нам нужны объективные измерения. Профилировщик нужен.

В комплект поставки Ruby входит профилировщик `profile`. Для его вызова достаточно включить библиотеку:

```
ruby -rprofile myprog.rb
```

Рассмотрим программу в листинге 16.6. Она открывает файл `/usr/share/dict/words` и ищет в нем анаграммы. Затем смотрит, у каких слов наибольшее количество анаграмм и распечатывает их.

Листинг 16.8. Поиск анаграмм в словаре

```
words = File.readlines("/usr/share/dict/words")
words.map! {|x| x.chomp }

hash = {}
words.each do |word|
  key = word.split("").sort.join
  hash[key] ||= []
  hash[key] << word
end

sizes = hash.values.map {|v| v.size }
most = sizes.max
list = hash.find_all {|k,v| v.size == most }

puts "Ни у одного слова нет более #{most-1} анаграмм."
list.each do |key,val|
  anagrams = val.sort
  first = anagrams.shift
  puts "Слово #{first} имеет #{most-1} анаграмм:"
  anagrams.each {|a| puts "   #{a}" }
end

num = 0
```

```
hash.keys.each do |key|
  n = hash[key].size
  num += n if n > 1
end

puts
puts "Всего слов в словаре: #{words.size},"
puts "из них имеют анаграммы: #{num}."
```

Наверняка вам интересно, какие получились результаты. Вот какие:

Ни у одного слова нет более 14 анаграмм.

```
Слово alerts имеет 14 анаграмм:
   alters
   artels
   estral
   laster
   lastre
   rastle
   ratels
   relast
   resalt
   salter
   slater
   staler
   stelar
   talers

Всего слов в словаре: 483523,
из них имеют анаграммы: 79537.
```

На моем компьютере этот файл содержит более 483 000 слов, и программа работала чуть меньше 18 секунд. Как вы думаете, на что ушло это время? Попробуем выяснить. Профилировщик выдал более 100 строк, отсортированных в порядке убывания времени. Мы покажем только первые 20:

```
%   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
42.78   190.93    190.93       15 12728.67 23647.33  Array#each
10.78   239.04     48.11  1404333     0.03     0.04  Hash#[]
 7.04   270.48     31.44        2 15720.00 25575.00  Hash#each
 5.66   295.73     25.25   483523     0.05     0.05  String#split
 5.55   320.51     24.78  1311730     0.02     0.02  Array#size
 3.64   336.76     16.25        1 16250.00 25710.00  Array#map
 3.24   351.23     14.47   483524     0.03     0.03  Array#sort
 3.12   365.14     13.91   437243     0.03     0.03  Fixnum#==
 3.04   378.72     13.58   483526     0.03     0.03  Array#join
 2.97   391.98     13.26   437244     0.03     0.03  Hash#default
 2.59   403.53     11.55   437626     0.03     0.03  Hash#[]=
 2.43   414.38     10.85   483568     0.02     0.02  Array#<<
```

```
 2.29   424.59     10.21         1 10210.00 13430.00  Array#map!
 1.94   433.23      8.64    437242     0.02     0.02  Fixnum#<=>
 1.86   441.54      8.31    437244     0.02     0.02  Fixnum#>
 0.72   444.76      3.22    483524     0.01     0.01  String#chomp!
 0.11   445.26      0.50         4   125.00   125.00  Hash#keys
 0.11   445.73      0.47         1   470.00   470.00  Hash#values
 0.06   446.00      0.27         1   270.00   270.00  IO#readlines
 0.05   446.22      0.22     33257     0.01     0.01  Fixnum#+
```

Видно, что больше всего времени программа тратит в методе `Array#each`. Это понятно, ведь цикл выполняется для каждого слова, и на каждой итерации делает довольно много. Среднее значение в данном случае сбивает с толку, поскольку почти все время уходит на первый вызов each, а остальные 14 (см. `anagrams.each`) выполняются очень быстро.

Мы также видим, что `Hash#[]` – дорогая операция (главным образом, потому что часто выполняется); на 1.4 миллиона вызовов было потрачено почти 11 секунд.

Обратите внимание, что метод `readlines` оказался чуть ли не в самом конце списка. Эта программа тратит время не на ввод-вывод, а на вычисления. На чтение файла ушло всего-то четверть секунды.

Но этот пример не показывает, в чем истинная ценность профилирования. В программе нет ни методов, ни классов. На практике вы, скорее всего, увидите свои методы среди системных. И тогда будете точно знать, какие из ваших методов находятся в числе первых 20 «пожирателей времени».

Надо ясно понимать, что профилировщик Ruby (видно, по иронии судьбы) работает медленно. Он подключается к программе во многих местах и следит за ее выполнением на низком уровне (причем сам написан на чистом Ruby). Так что не удивляйтесь, если ваша программа в ходе сеанса профилирования будет работать на несколько порядков медленнее. В нашем примере она работала 7 минут 40 секунд (460 секунд), то есть в 25 раз медленнее обычного.

Помимо профилировщика, есть еще один низкоуровневый инструмент – стандартная библиотека `benchmark`, которая тоже полезна для измерения производительности.

Один из способов ее применения – вызвать метод `Benchmark.measure` и передать ему блок.

```
require 'benchmark'

file = "/usr/share/dict/words"
result = Benchmark.measure { File.readlines(file) }
puts result

# Выводится:    0.350000   0.070000   0.420000 (  0.418825)
```

Этот метод выводит следующую информацию:

- время, затраченное процессором в режиме пользователя (в секундах);
- время, затраченное процессором в режиме ядра (в секундах);
- полное затраченное время – сумма вышеупомянутых величин;
- время работы программы (по часам).

Для сравнения производительности отдельных участков удобен метод Benchmark.bm. Передайте ему блок, а он сам передаст блоку объект формирования отчета. Если вызвать этот объект, передав ему метку и блок, то он выведет метку, а за ней временные характеристики блока. Пример:

```
require 'benchmark'

n = 200_000
s1 = ""
s2 = ""
s3 = ""

Benchmark.bm do |rep|
  rep.report("str <<     ") { n.times { s1 << "x" } }
  rep.report("str.insert ") { n.times { s3.insert(-1,"x") } }
  rep.report("str +=     ") { n.times { s2 += "x" } }
end
```

Здесь мы сравниваем три способа добавить символ в конец строки, дающие один и тот же результат. Чтобы получить более точные цифры, каждая операция выполняется 200 000 раз. Вот что получилось:

```
                user          system        total          real
str <<      0.180000       0.000000     0.180000     (  0.174697)
str.insert  0.200000       0.000000     0.200000     (  0.200479)
str +=     15.250000      13.120000    28.370000     ( 28.375998)
```

Обратите внимание, что последний вариант на два порядка медленнее остальных. Почему? Какой урок можно извлечь отсюда?

Вы можете предположить, что оператор + почему-то работает медленно, но дело не в этом. Это единственный из трех способов, который не работает с одним и тем же объектом, а каждый раз создает новый.

Стало быть, вывод такой: создание объекта – дорогая операция. Библиотека Benchmark может преподать много подобных уроков, но я все же рекомендую сначала заняться высокоуровневым профилированием.

В пакете Minitest тоже есть средства эталонного тестирования (библиотека minitest/benchmark), а в RSpec – флаг -profile, при наличии которого выводится информация о самых медленных спецификациях.

16.7. Объекты форматированной печати

Метод inspect (и вызывающий его метод p) предназначен для вывода объектов в виде, понятном человеку. В этом смысле он является связующим звеном между тестированием и отладкой, поэтому рассмотрение его в этой главе оправдано.

Проблема в том, что результат, формируемый методом p, бывает трудно читать. Из-за этого и появилась библиотека pp, добавляющая одноименный метод.

Рассмотрим следующий искусственный пример объекта my_obj:

```
class MyClass

  attr_accessor :alpha, :beta, :gamma

  def initialize(a,b,c)
    @alpha, @beta, @gamma = a, b, c
  end

end

x = MyClass.new(2, 3, 4)
y = MyClass.new(5, 6, 7)
z = MyClass.new(7, 8, 9)

my_obj = { x => y, z => [:p, :q] }

p my_obj
Вызов метода p печатает следующее:
{#<MyClass:0xb7eed86c @beta=3, @alpha=2,
 @gamma=4>=>#<MyClass:0xb7eed72c @beta=6, @alpha=5, @gamma=7>,
 #<MyClass:0xb7eed704 @beta=8, @alpha=7, @gamma=9>=>[:p, :q]}
```

Все правильно и в общем-то даже читаемо. Но не красиво. А давайте затребуем библиотеку pp и воспользуемся предоставляемым ей методом pp:

```
require 'pp'

# ...

pp my_obj
```

Теперь вывод приобретает такой вид:

```
{#<MyClass:0xb7f7a050 @alpha=7, @beta=8, @gamma=9>=>[:p, :q],
 #<MyClass:0xb7f7a1b8 @alpha=2, @beta=3, @gamma=4>=>
  #<MyClass:0xb7f7a078 @alpha=5, @beta=6, @gamma=7>}
```

Мы получили хотя бы пробелы и разбиение на строки. Уже лучше. Но можно пойти еще дальше. Предположим, что в классе MyClass определен специальный метод pretty_print:

```
class MyClass

  def pretty_print(printer)
    printer.text "MyClass(#@alpha, #@beta, #@gamma)"
  end

end
```

Аргумент printer передается вызывающей программой (или методом pp). Это аккумулятор текста, являющийся экземпляром класса PP; мы вызываем его метод text и передаем ему текстовое представление self. Вот что получается в результате:

```
{MyClass(7, 8, 9)=>[:p, :q], MyClass(2, 3, 4)=>MyClass(5, 6, 7)}
```

Разумеется, можно настроить поведение по своему вкусу. Можно, например, печатать переменные экземпляра на разных строчках с отступами.

На самом деле, в библиотеке pp есть много средств для подготовки ваших классов к совместной работе с методом pp. Методы object_group, seplist, breakable и прочие позволяют управлять расстановкой запятых, разбиением на строки и другими способами форматирования. Дополнительную информацию можно найти в документации по pp.

16.8. О том, что осталось за кадром

В этой книге мы можем охватить лишь части экосистемы Ruby. По ходу дела я буду вскользь упоминать дополнительные средства. Кроме того, материал на эту тему есть в главе 21.

Я глубоко верю в инструменты для измерения покрытия кода. *Невероятно* полезно знать, какие части программы не исполнялись. Это дает возможность найти места, не покрытые автономными тестами.

В последние годы самым популярным инструментом такого рода стала программа simple-cov; она пришла на смену rcov и устанавливается в виде gem-пакета. Во время выполнения она создает каталог coverage и помещает в него ряд HTML-файлов. Файл index.html содержит общие сведения и ссылки на все исходные тексты, в которых подсвечены строки, выполнявшиеся хотя бы один раз.

Для Ruby имеется несколько средств непрерывной интеграции; самое популярное из них, не нуждающееся во внешнем хостинге, – Jenkins. Наиболее популярной размещаемой на хостинге службой для проектов с открытым исходным кодом является Travis CI. Для закрытых проектов можно упомянуть Travis CI, Circle CI, Semaphore и Codeship. Поиск в Интернете подскажет, что для вас оптимально.

Для Ruby есть также немало инструментов статического анализа кода. Я пользуюсь программой flog, которая помогает находить «корявый» или излишне сложный код. Программа flay ищет структурно похожие фрагменты, которые, возможно, имеет смысл подвергнуть рефакторингу (в соответствии с принципом DRY – «Не повторяйся»). Программа heckle особенно полезна (и может привести в уныние), так как отыскивает слабые места в ваших тестах (она модифицирует код и проверяет, привело ли это к падению тестов). Размещаемая на хостинге служба Code Climate автоматически запускает эти средства контроля качества, а также средства измерения покрытия кода и аудита безопасности.

С точки зрения стиля программирования на Ruby полезен инструмент rubocop. Однако лично я не согласен с одним или двумя правилами, встроенными в него по умолчанию.

16.9. Заключение

В этой главе мы рассмотрели некоторые подходы к тестированию (особенно к автономному тестированию). Конкретно мы познакомились с пакетами Minitest, RSpec и Cucumber.

Мы бросили беглый взгляд на отладчик Ruby. Мы также видели, как с помощью программы pry можно войти в интерактивный цикл REPL. Наконец, мы остановились на профилировании и замере временных характеристик программы на Ruby.

Предположим, что вы работаете над программой, которую собираетесь предложить обществу. Что вы делаете после завершения тестирования? Пришло время задуматься над оформлением дистрибутивного пакета и способами распространения своего детища. Этим мы и займемся в следующей главе.

Глава 17. Создание пакетов и распространение программ

Все больше и больше продуктов
и в первую очередь аспирин – выпускаются
в упаковке, защищенной до такой степени,
что потребитель уже и воспользоваться ими
не может.

-- Дэйв Бэрри

Эта глава посвящена вопросу о том, как проявлять уважение к программистам, работающим с вашим кодом. Если никто (включая и вас) с вашим кодом не работает или вы не хотите проявлять вежливость к кому-то (включая и себя), можете пропустить ее.

Есть две вещи, о которых типичный программист не желает задумываться: документация и процедура установки. Я настоятельно советую думать и о том, и о другом. Даже если никто, кроме вас, не увидит вашего кода, пройдет каких-то несколько месяцев, и вы напрочь забудете тонкости и детали своей программы. Прилагайте максимум усилий, чтобы облегчить жизнь других разработчиков, ведь в будущем вы станете одним из них.

Чтобы быть вежливым по отношению к другим разработчикам, нужно немногое: сделать так, чтобы ваш код было просто использовать, чтобы его было просто модифицировать и чтобы его было легко понять. Для решения этих задач к вашим услугам инструменты Rubygems, Bundler и RDoc. В этой главе мы узнаем, как ими пользоваться.

17.1. Библиотеки и система Rubygems

Поделиться написанным на Ruby кодом с другим разработчиком на удивление просто: нужно лишь отдать ему копию содержащего код файла. Он загрузит файл командой `require` и сможет воспользоваться вашим кодом. Проблемы возникают, когда вы хотите обновить свой код или отдать его в распоряжение всего сообщества. Для их решения создана система Rubygems и сайт `rubygems.org`.

Система `Rubygems` является частью дистрибутива Ruby, начиная с версии 1.9, и составляет основу распространения Ruby-кода. Она позволяет собрать gem-пакет, состоящий из единственного сжатого файла. Имена gem-пакетов имеют

расширение .gem, они содержат как код некоторой библиотеки, так и описывающие ее метаданные, включая информацию об авторе и номере версии.

Сайт Rubygems.org основан сообществом для распространения открытых gem-пакетов. Каждому gem-пакету отведена страница, содержащая сведения об его авторах и версиях. Если автор включил ссылки на домашнюю страницу пакета, его исходный код и онлайновую документацию, то они тоже будут показаны. Кто угодно может создать gem-пакет и поместить его на rubygems.org, чтобы все могли им бесплатно пользоваться.

17.1.1. Работа с RubyGems

Исполняемая единица в Rubygems называется gem-пакетом. Система располагает подкомандами, и для каждой подкоманды определены свои параметры. Основные функции очень просты. Чтобы установить gem-пакет rake, нужно выполнить команду gem install rake. Rubygems сначала поищет gem-файл в текущем каталоге, а затем обратится к сайту rubygems.org. Если gem-пакет найден, то система установит его, так чтобы он был доступен любой Ruby-программе. По умолчанию устанавливается последняя версия пакета, но если, к примеру, вы хотите установить версию rake с номером 10.0.1, то можете ввести команду gem install rake -v 10.0.1. Чтобы узнать о дополнительных возможностях установки gem-пакетов, наберите команду gem help install.

Иногда gem-пакет зависит от других пакетов. В этом случае Rubygems автоматически найдет и установит зависимости, так что на вашей машине окажутся все пакеты, необходимые для работы запрошенного.

Откуда вы узнаете, что искать? На сервере rubygems.org перечислены все когда-либо созданные gem-пакеты, т. е. имена всех пакетов. Можно вместо этого поискать на rubygems.org пакет по имени командой gem list -r NAME.

Но ни один из этих вариантов не годится, если вы не знаете имени gem-пакета. На сайте ruby-toolbox.com имеется очень удобный каталог, где gem-пакеты сгруппированы по своему назначению, а не по имени. Там же имеется информация о популярности и ходе разработки каждого пакета, она позволит решить, какой из пакетов со сходной функциональностью выбрать.

Получить список установленных gem-пакетов позволяет команда gem list. Нетрудно догадаться, что для удаления пакета служит команда gem uninstall NAME.

Познакомившись с тем, как найти и установить gem-пакеты, посмотрим, как создать свой собственный.

17.1.2. Создание gem-пакетов

Для построения gem-пакета необходимо создать файл с расширением .gemspec и поместить в него определенный код. Ниже приведен пример такого файла для gem-пакета drummer:

```
# drummer.gemspec
Gem::Specification.new do |spec|
   spec.name    = "Drummer"
   spec.version = "1.0.2"
   spec.authors = ["H. Thoreau"]
```

```
    spec.email = ["cabin@waldenpond.net"]
    spec.description = %q{A Ruby library for those who march to a
different drummer.}
    spec.summary  = %q{Drum different}
    spec.homepage = "http://waldenpond.com/drummer"
    spec.license  = "MIT"

    spec.files  = Dir["./**/*"]
    spec.executables = Dir["./bin/*"]
    spec.test_files  = Dir["./spec/**/*"]
    spec.require_paths = ["lib"]

    spec.add_development_dependency "rake"
    spec.add_runtime_dependency "activerecord", "~> 4.1.0"
end
```

Файл не требует особых пояснений, но, если что-то непонятно, не стоит расстраиваться. После того как gemspec-файл создан, для построения пакета нужно выполнить команду `gem build drummer.gemspec`. А для загрузки на сайт rubygems.org – команду `gem push drummer-1.0.2.gem`.

Для рассмотрения Rubygems во всех подробностях потребовалась бы целая книга, так что остановимся на этом. Дополнительные сведения о Rubygems, в том числе о команде gem и gemspec-файлах, обратитесь к руководствам на сайте http://guides.rubygems.org.

17.2. Управление зависимостями с помощью Bundler

Итак, собирать и распространять отдельные библиотеки в виде gem-пакетов мы научились, но тут возникает новая проблема. Gem-пакеты должны содержать небольшие библиотеки с четко определенным назначением. Сколько-нибудь полезному приложению, скорее всего, понадобится несколько пакетов. Как вести учет gem-пакетов, необходимых в крупном проекте? Что если мы захотим взять более свежую версию уже используемого пакета? Ответ на эти и другие вопросы дает менеджер зависимостей Bundler.

До появления Bundler в проектах на Ruby не было возможности объявлять, какие пакеты необходимы. Чтобы развернуть проект на новой машине, нужно было прочитать в файле readme о том, какие gem-пакеты нужны, и оставалось лишь надеяться, что автор ничего не забыл и что с момента написания readme в эти пакеты не было внесено несовместимых изменений. Bundler следит за всеми используемыми в проекте gem-пакетами и их версиями и позволяет одной командой установить любой нужный пакет.

Но до появления Bundler трудности возникали не только с отслеживанием пакетов. При переходе на новую версию пакета приложение могло перестать работать, если эта версия зависела от других версий остальных используемых пакетов. Bundler анализирует все пакеты, от которых зависит проект, и находит множество совместимых версий.

Подготовить новый проект к использованию Bundler очень просто: нужно лишь создать файл Gemfile и записать в него список необходимых gem-пакетов. В Gemfile должен быть хотя бы один вызов метода source, но может быть и несколько. После них обычно идет список обращений к методу gem. В приведенном ниже Gemfile показаны примеры наиболее употребительных обращений:

```
source "https://rubygems.org" # Download gems from rubygems.org

gem 'red'                 # Зависимость от gem-пакета "red"
gem 'green', '1.2.1'      # Gem-пакет "green" - только версия 1.2.1
gem 'blue', '>= 1.0'      # Версия 1.0 или старше gem-пакета "blue"
gem 'yellow', '~> 1.1'    # "yellow" 1.1 ли старше, но меньше 2.0
gem 'purple', '~> 1.1.1' # Как минимум, 1.1.1, но меньше 1.2
```

Назначение Gemfile – хранить список gem-пакетов, необходимых вашему проекту, вместе с указанием версий. Если проект содержит Gemfile, то для установки всех необходимых пакетов достаточно выполнить команду bundle install.

Автоматически создается также файл Gemfile.lock. В отличие от Gemfile, в нем хранятся имена и номера версий всех установленных gem-пакетов, а также их зависимости, зависимости зависимостей и т. д.

В состав исходного кода проекта включайте как Gemfile, так и Gemfile.lock. Это позволит другим разработчикам установить сразу все зависимости командой bundle install.

Поскольку в каждом проекте используются свои версии gem-пакетов, нередко возникает ситуация, когда в двух проектах указаны разные версии общеупотребительных пакетов, например rake и rspec. Bundler может воспрепятствовать использованию неподходящих версий пакета в текущем проекте, но для этого пакет нужно сначала загрузить.

Чтобы в данном проекте использовалась правильная версия gem-пакета, поставьте перед именем команды bundle exec. Так, вместо rake spec нужно выполнить bundle exec rake spec. Тогда гарантируется, что будет взята именно та версия rake, которая нужна данному проекту.

Поскольку писать каждый раз bundle exec быстро надоедает, имеет смысл определить для этой команды псевдоним оболочки, например b. Есть и альтернатива – Bundler может помещать специфичные для проекта исполняемые файлы в каталог bin, например bin/rake. Дополнительные сведения смотрите в разделе документации по Bundler, посвященном binstub. Еще один прием – выполнить команду bundle exec bash и тем самым запустить новую оболочку, в которой окружение уже настроено правильно.

Если ваше приложение представляет собой Ruby-файл, а не исполняемую программу, то оно должно затребовать Bundler до всех остальных gem-пакетов. Проще всего это сделать таким образом:

```
require 'bundler/setup'
# здесь можно затребовать другие библиотеки...
```

Приложения для Rails также вызывают метод Bundler.require из файла boot.rb. Тогда все gem-пакеты, перечисленные в Gemfile, затребуются автоматически. За-

грузка большого числа пакетов может занять много времени, и это типичная причина длительных задержек в момент запуска приложения. Авторы Bundler рекомендуют избегать вызова метода `Bundler.require` и помещать предложения `require` там, где они действительно нужны.

Актуальную и полную информацию о Bundler можно найти в документации на сайте http://bundler.io.

17.2.1. Семантическое версионирование

Наверное, вы недоумеваете, зачем Gemfile позволяет указывать диапазон версий, например `< 2.5` или `>= 3.1.4`. Такие требования к версии дают возможность обновлять gem-пакеты без редактирования Gemfile. Если в Gemfile приложения имеется команда `gem "drummer", "~> 1.0"`, то для перехода с версии 1.0.2 на новую версию 1.0.3 достаточно выполнить команду `bundle update drummer`.

Правда, обновление gem-пакетов таит в себе опасность – что, если новая версия работает иначе, чем старая? Тогда приложение может «поломаться». Чтобы как-то снизить этот риск, многие разработчики пакетов прибегают к схеме «семантического версионирования». Звучит загадочно, но, по существу, эта схема основана на трех принципах, по одному для каждого числа в номере версии пакета.

Первое число, «основной» номер версии, говорит о фундаментальных изменениях. Если основной номер увеличивается, значит, надо быть готовым к изменению кода собственной программы, в которой этот gem-пакет используется. И наоборот, если в свой gem-пакет вы внесли несовместимые с предыдущей версией изменения, обязательно увеличьте основной номер версии.

Второе число называется «дополнительным» номером версии и говорит о добавлении новых возможностей без изменения существующих. Gem-пакет продолжает предоставлять все, что было раньше, и кое-что сверх того.

Последнее, третье, число называется «добавочным» (tiny) номером версии. Если оно равно нулю, то его иногда опускают, например: версия 2.2. Увеличение добавочного номера означает, что исправлена какая-то ошибка. Риск при переходе на версию с новым добавочным номером минимален, потому что в ней не появилось ни нового кода, ни несовместимых изменений.

Вообще говоря, семантическое версионирование считается наиболее полезной схемой нумерации версий. Даже для самого языка Ruby оно применяется, начиная с версии 2.1.0. Однако имейте в виду, что не все gem-пакеты придерживаются этой схемы.

Хотя семантическое версионирование показывает, насколько сильно изменился gem-пакет при переходе от одной версии к другой, оно все же не дает полного решения проблемы обновления пакетов. Тем не менее, скрупулезное соблюдение требований к номерам версий и использование команды `bundle update` делает обновление пакетов если не совсем беспроблемным, то уж точно менее болезненным делом.

17.2.2. Загрузка зависимостей из Git

Помимо управления gem-пакетами и их версиями, Bundler может использовать пакеты непосредственно из репозиториев git, в которых хранится их исходный

код. Поскольку git позволяет создавать собственную копию репозитория, с помощью Bundler очень просто создать ветвь репозитория пакета, исправить ошибку в нем и протестировать ее. Затем можно использовать исправленную версию из собственного репозитория, до тех пор пока владелец gem-пакета не примет ваше исправление и не положит новую версию на сайт rubygems.org.

Следующий Gemfile означает, что нужен gem-пакет с именем "rake", и мы хотим взять его непосредственно из репозитория Джима Вейриха в Github:

```
gem "rake", git: "https://github.com/jimweirich/rake.git",
   branch: "master"
```

Команда `bundle install` создаст клон репозитория git с данным URL и извлечет указанную ветвь или версию.

17.2.3. Создание gem-пакетов с помощью Bundler

Хотя основное назначение Bundler – управление установкой gem-пакетов, в нем есть несколько весьма удобных команд, реализующих надстройки над Rubygems, которые позволяют максимально удобно создавать и выпускать пакеты.

Чтобы создать пакет «drummer» с помощью Bundler, нужно просто выполнить команду `bundle gem drummer`. В результате будет создан каталог `drummer`, а в нем Ruby-файл с исходным кодом и файл `.gemspec`, в который можно записать название и описание. Выпуск gem-пакета также упрощен. Номер версии хранится в файле `lib/drummer/version.rb`. После его обновления для отправки пакета на rubygems.org достаточно выполнить команду `rake release`.

Разрабатывая собственный код, обращайте внимание на части, которые можно использовать повторно и будьте готовы оформить их в виде отдельного gem-пакета. Таким образом, нетрудно создать частный пакет, используемый только в ваших проектах (мы поговорим об этом в следующем разделе). Со временем, если обнаружится, что частные gem-пакеты достаточно полезны, их можно будет разместить на сайте rubygems.org. Огромное количество опубликованных бесплатных пакетов с открытым исходным кодом – одно из самых весомых преимуществ сообщества Ruby.

17.2.4. Частные gem-пакеты

По мере развития и роста проекта или компании, скорее всего, накапливаются частные gem-пакеты для внутреннего пользования. Они содержат код, общий для нескольких приложений или служб, но не предназначенный для опубликования. И хотя в общем случае лучше бы вынести повторно используемые библиотеки и раскрыть их код, в любом достаточно крупном проекте неизбежно образуются частные gem-пакеты.

На первых порах управлять частными пакетами можно с помощью git, поскольку репозиторий Git легко закрыть, так что установить пакеты смогут только авторизованные пользователи. Но если число таких пакетов становится слишком велико, то пора задуматься о закрытом gem-сервере. Есть службы, например Gemnasium, которые предлагают организовать закрытый gem-сервер за несколько

долларов в месяц, а также проекты типы Geminabox, которые позволяют установить собственный gem-сервер в любом удобном для вас месте.

17.3. Программа RDoc

Программа RDoc, которую написал Дэйв Томас и поддерживает Эрик Ходел, входит в состав дистрибутива Ruby. Она получает на входе Ruby-код и создает каталог doc, содержащий HTML-файлы с документацией по классам, методам, константам и прочим элементам исходного кода.

Замечательной особенностью RDoc является то, что он пытается вывести нечто полезное, *даже если в исходном тексте вообще нет комментариев.* Для этого она анализирует текст программы и собирает информацию обо всех классах, модулях, константах, методах и т. д. Тем самым можно получить более-менее полезный HTML-файл из исходного текста, не содержащего никакой внутренней документации. Если раньше не пробовали, попробуйте сейчас.

Но это еще не все. RDoc также пытается ассоциировать найденные комментарии с конкретными частями программы. Общее правило таково: блочный комментарий, предшествующий определению (скажем, класса или метода), считается описанием этого определения.

Если просто вызвать RDoc для какого-нибудь исходного текста на Ruby, то будет создан каталог doc, в который помещаются все выходные файлы (стандартный шаблон сам по себе неплох, но есть и другие). Откройте в браузере файл index.html и изучите его.

В листинге 17.1. приведен простой (почти ничего не содержащий) исходный файл. Все определенные в нем методы пусты. Но RDoc даже в таком случае формирует симпатичную страницу документации (рис. 17.1).

Листинг 17.1. Простой исходный файл

```
require 'foo'

# Внешний класс, MyClass
class MyClass
  CONST = 237

  # Внутренний класс MyClass::Alpha...

  class Alpha

    # Класс The MyClass::Alpha::Beta...

    class Beta
      # Метод класса Beta mymeth1
      def mymeth1
      end
    end
```

```
    # Метод класса Alpha mymeth2
    def mymeth2
    end
  end
  # Инициализировать объект
  def initialize(a,b,c)
  end

  # Создать объект со значениями по умолчанию

 def self.create
  end

  # И метод экземпляра

 def do_something
  end
end
```

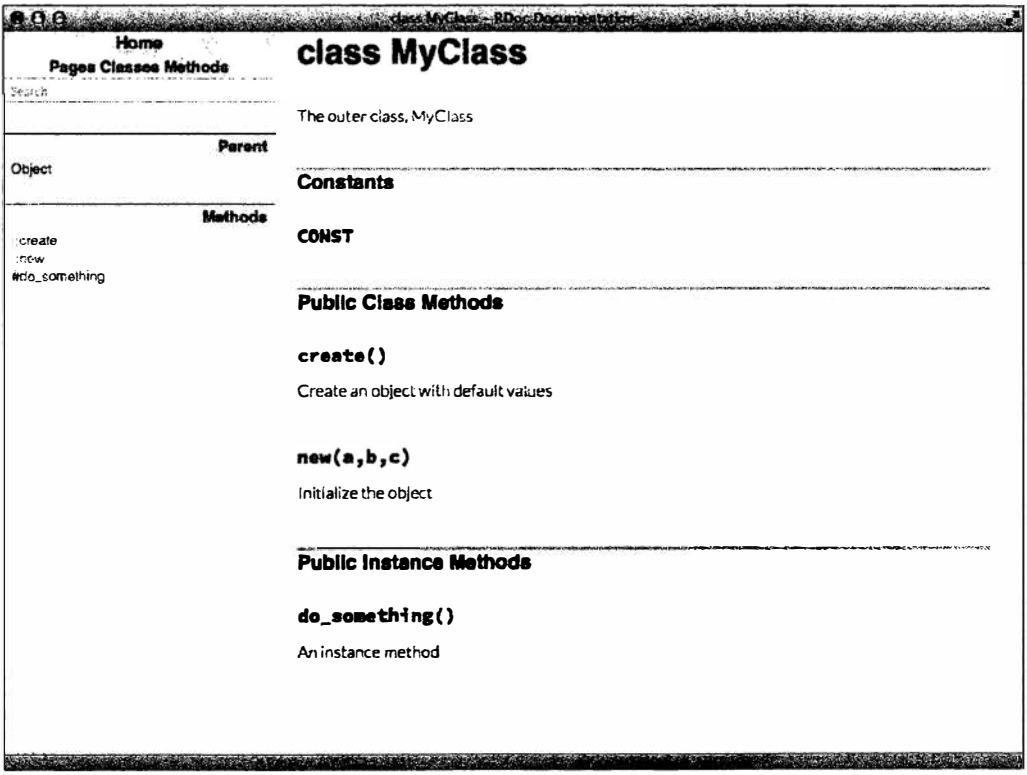


Рис. 17.1. Выходной файл, формируемый программой RDoc по исходному тексту в листинге 17.1

В этом разделе мы обсудим еще две полезные функции. Имя каждого метода является ссылкой, при щелчке по которой открывается исходный текст метода. При изучении библиотеки это оказывается исключительно полезно – документация API ссылается на сам код.

Кроме того, когда программа RDoc распознает URL, она помещает в выходной файл гиперссылку. По умолчанию, текст гиперссылки совпадает с самим URL, но это можно изменить. Если поместить в фигурных скобках какой-нибудь описательный текст, а за ним URL в квадратных скобках (`{descriptive text}[myurl]`), то этот текст и станет содержимым ссылки. Если текст состоит из одного слова, фигурные скобки можно опустить.

17.3.1. Простая разметка

На случай, если вы захотите улучшить форматирование документации, RDoc располагает собственным механизмом разметки, поэтому можно включать в исходный текст информацию о форматировании. Правила языка разметки выбраны так, что текст в редакторе выглядит «естественно», но вместе с тем может быть легко преобразован в HTML.

В листинге 17.2 приведено несколько примеров разметки, дополнительную информацию см. в книге «Programming Ruby» или в документации по RDoc API. На рис. 17.2 показано, во что преобразуется текст в листинге 17.2 (нижний фрейм).

Листинг 17.2. Пример разметки для RDoc

```
# This block comment will not appear in the output.
# Rdoc only processes a single block comment before
# each piece of code. The empty line after this
# block separates it from the block below.

# This block comment will be detected and
# included in the rdoc output.
#
# Here are some formatting tricks.
#
# Boldface, italics, and "code" (without spaces):
# This is *bold*, this is _italic_, and this is +code+.
#
# With spaces:
#
# This is a bold phrase. Have you read Intruder
# in the Dust? Don't forget to require thread
# at the top.
#
# = First level heading
# == Second level heading
# === Third level heading
#
# Here's a horizontal rule:
```

```
# --
#
# Here's a list:
# - item one
# - item two
# - item three

class MarkupDocumentation
   # This block will not appear because the class after
   # it has been marked with a :nodoc: directive.
   class NotDocumented # :nodoc:
   end
end

# This block comment will not show up in in the output.
# Rdoc only processes blocks immediately before code,
# and this comment block is after the only code in this
# listing.
This is a bold phrase. Have you read Intruder
in the Dust? Don't forget to require thread
at the top.

= First level heading
== Second level heading
=== Third level heading

Here's a horizontal rule:
---

Here's a list:
- item one
- item two
- item three

=end
=begin
This block comment is untagged and will not show up in
rdoc output. Also, I'm not putting blank lines between
the comments, as this will terminate the comments until
some real program source is seen. If this comment had
been before the previous one, processing would have
stopped here until program text appeared.
=end
```

В листинге 17.2 приведено несколько правил разбора комментариев, принятых в RDoc. Не все они интуитивно очевидны. Считается, что пустые строки завершают блок комментариев, даже если вслед за пустой строкой сразу идет еще один такой блок.

Внутри блока комментариев, начинающегося со знака #, можно отключить копирование текста в выходной файл, вставив строку #-- (а следующая такая строка вновь включает копирование). Ведь не все комментарии должны быть частью пользовательской документации.

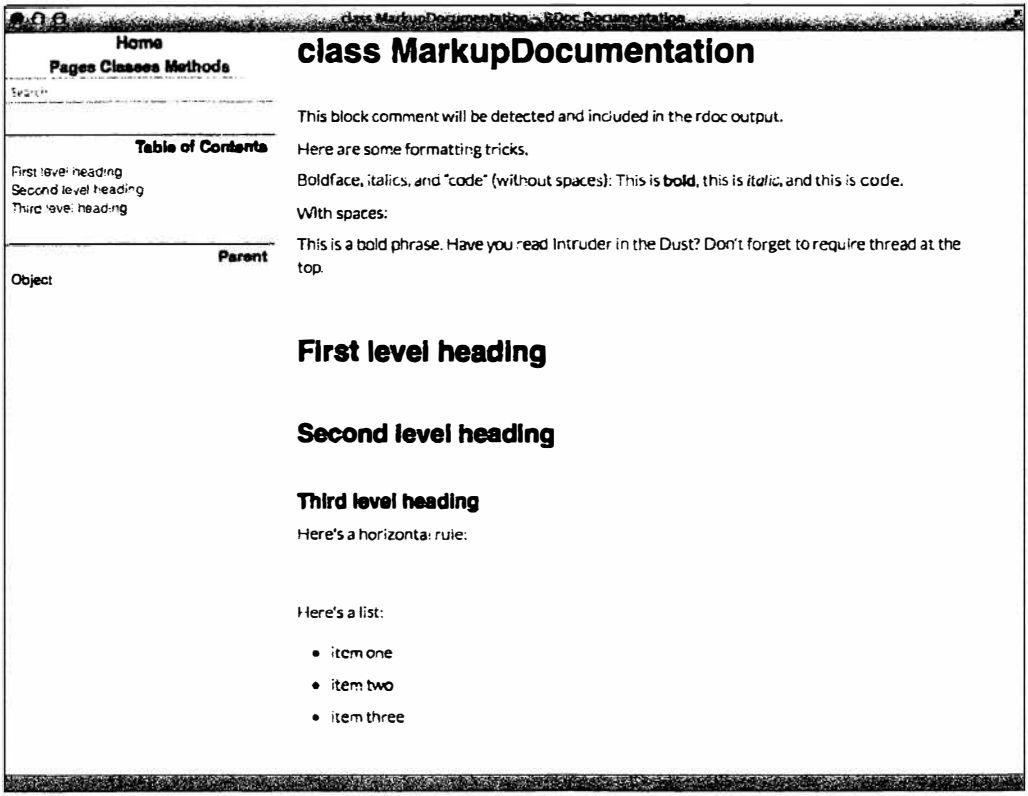


Рис. 17.2. Результат работы RDoc для примера из листинга 17.2

Наконец, внутри блочных комментариев могут встречаться теги, изменяющие сгенерированный RDoc HTML-файл. Ниже приведены наиболее употребительные:

- **:include:** – включить в документацию содержимое указанного файла. Величины отступов будут скорректированы, чтобы не портить внешний вид.
- **:title:** – задать заголовок документа.
- **:main:** – задать начальную страницу документации.

17.3.2. Создание улучшенной документации с помощью Yard

Хотя RDoc – замечательный инструмент (к тому же, включенный в дистрибутив Ruby), существует еще одно средство для подготовки документации с дополнительными возможностями: Yay! A Ruby Documentation, его еще называют YARD. У YARD по сравнению с RDoc есть два основных преимущества: динамический

предварительный просмотр документации по мере ее ввода и включение подробной информации о типах и назначении всех аргументов и возвращаемого значения каждого метода. На рис. 17.3 приведен пример документа, формируемого YARD при запуске для кода из листинга 17.1. Дополнительные сведения о YARD можно найти на сайте http://yardoc.org.

17.4. Заключение

В этой главе мы познакомились с библиотечной системой RubyGems и менеджером зависимостей Bundler. Мы узнали, как искать, создавать и публиквать gem-пакеты, которыми могут воспользоваться другие разработчики. Мы также обсудили основы документирования проектов с помощью программ RDoc и YARD. В следующей главе мы снова переключим передачу и поговорим об интересной и сложной предметной области: сетевом программировании.

Глава 18. Сетевое программирование

Не следует недооценивать пропускную способность забитого лентами фургона, мчащегося по скоростной трассе.

Эндрю С. Таненбаум

Когда торговец говорит вам слово *networking* (социальные связи), он, скорее всего, желает всучить свою визитную карточку. Но в устах программиста это слово обозначает электронное взаимодействие физически удаленных машин, неважно, находятся они в разных углах комнаты, в разных районах города или в разных частях света.

Для программистов сеть чаще всего ассоциируется с набором протоколов TCP/IP – тем языком, на котором неслышно беседуют миллионы машин, подключенных к сети Интернет. Несколько слов об этом наборе, перед тем как перейти к конкретным примерам.

Концептуально сетевое взаимодействие принято представлять в виде различных уровней (или слоев) абстракции. Самый нижний – *канальный уровень*, на котором происходит аппаратное взаимодействие, о нем мы говорить не будем. Сразу над ним расположен *сетевой уровень*, который отвечает за перемещение пакетов в сети, это епархия протокола IP (Internet Protocol). Еще выше находится *транспортный уровень*, на котором расположились протоколы TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). Далее мы видим *прикладной уровень*, это мир telnet, FTP, протоколов электронной почти и т. д. и т. п.

Можно обмениваться данными непосредственно по протоколу IP, но обычно так не поступают. Чаще нас интересуют протоколы TCP и UDP.

Протокол TCP обеспечивает надежную связь между двумя компьютерами (хостами). Он упаковывает данные в пакеты и распаковывает их, подтверждает получение пакетов, управляет таймаутами и т.д. Поскольку протокол надежный, приложению нет нужды беспокоиться о том, получил ли удаленный хост посланные ему данные.

Протокол UDP гораздо проще, он просто отправляет пакеты (датаграммы) удаленному хосту, как будто это двоичные почтовые открытки. Нет никакой гарантии, что данные будут получены, поэтому протокол называется ненадежным (а, следовательно, приложению придется озаботиться дополнительными деталями).

Ruby поддерживает сетевое программирование на низком уровне (главным образом, по протоколам TCP и UDP), а также и на более высоких, в том числе по протоколам telnet, FTP, SMTP и т. д. На рис. 18.1 представлена иерархия классов, из которой видно, как организована поддержка сетевого программирования в Ruby.

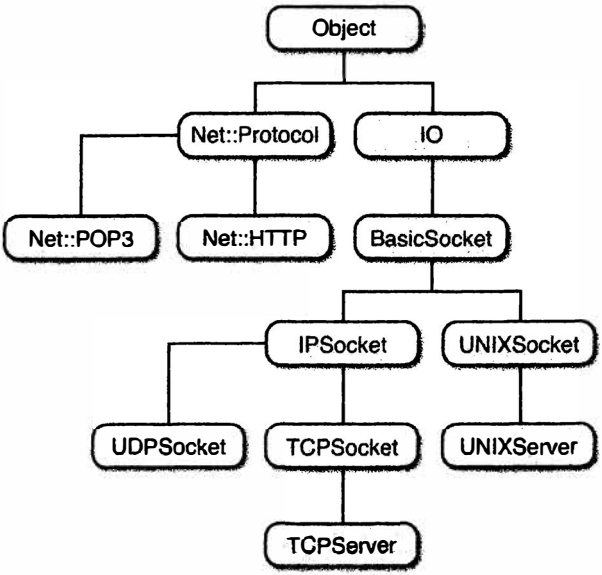


Рис. 18.1. Часть иерархии наследования для поддержки сетевого программирования в Ruby

Отметим, что большая часть этих классов прямо или косвенно наследует классу `IO`. Следовательно, мы можем пользоваться уже знакомыми методами этого класса. Помимо этих классов, в стандартную библиотеку входят `Net::IMAP`, `Net::POPMail`, `Net::SMTP`, `Net::Telnet` и `Net::FTP`, все они наследуют непосредственно классу `Object`.

Попытка документировать все функции всех показанных классов завела бы нас далеко за рамки этой книги. Я лишь покажу, как можно применять их к решению конкретных задач, сопровождая примеры краткими пояснениями. Полный перечень всех методов вы можете найти к справочном руководстве на сайте rdoc.info.

Ряд важных областей применения в этой главе вообще не рассматривается, поэтому сразу упомянем о них. Класс `Net::Telnet` упоминается только в связи с NTP-серверами в разделе 18.2.2; этот класс может быть полезен не только для реализации собственного telnet-клиента, но и для автоматизации всех задач, поддерживающих интерфейс по протоколу telnet.

Библиотека `Net::FTP` также не рассматривается. В общем случае автоматизировать обмен по протоколу FTP несложно и с помощью уже имеющихся клиентов, так что необходимость в этом классе возникает реже, чем в прочих.

Класс Net::Protocol, являющийся родительским для классов HTTP, POP3 и SMTP полезен скорее для разработки новых сетевых протоколов, но эта тема в данной книге не обсуждается.

На этом завершим краткий обзор и приступим к рассмотрению низкоуровневого сетевого программирования.

18.1. Сетевые серверы

Жизнь *сервера* проходит в ожидании входных сообщений и ответах на них. Не исключено, что для формирования ответа требуется серьезная обработка, например обращение к базе данных, но с точки зрения сетевого взаимодействия сервер просто принимает запросы и отправляет ответы.

Но даже это можно организовать разными способами. Сервер может в каждый момент времени обслуживать только один запрос или иметь несколько потоков. Первый подход проще реализовать, зато у второго есть преимущества, когда много клиентов одновременно обращаются с запросами.

Можно представить себе сервер, единственное назначение которого состоит в том, чтобы облегчить общение между клиентами. Классические примеры – чат-серверы, игровые серверы и файлообменные сети.

18.1.1. Простой сервер: время дня

Рассмотрим самый простой сервер, который только можно себе представить. Предположим, что некоторая машина располагает такими точными часами, что ее можно использовать в качестве стандарта времени. Такие серверы, конечно, существуют, но взаимодействуют не по тому тривиальному протоколу, который мы обсудим ниже. (В разделе 18.2.2 приведен пример обращения к подобному серверу по протоколу telnet.)

В нашем примере все запросы обслуживаются в порядке поступления однопоточным сервером. Когда приходит запрос от клиента, мы возвращаем строку, содержащую текущее время. Ниже приведен код сервера:

```
require "socket"

server = UDPSocket.open       # Применяется протокол UDP...
server.bind nil, 12321

loop do
  text, sender = server.recvfrom(1)
  server.send("#{Time.now}\n", 0, sender[3], sender[1])
end
```

А это код клиента:

```
require "socket"
require "timeout"

socket = UDPSocket.new
```

```
socket.connect("localhost", 12321)

socket.send("", 0)
timeout(10) do
  time = socket.gets
  puts "The time is #{time}"
end
```

Чтобы сделать запрос, клиент посылает пустой пакет. Поскольку протокол UDP ненадежен, то, не получив ответа в течение некоторого времени, мы завершаем работу по таймауту.

В следующем примере такой же сервер реализован на базе протокола TCP. Он прослушивает порт 12321; запросы к этому порту можно посылать с помощью программы telnet (или клиента, код которого приведен ниже).

```
require "socket"

server = TCPServer.new(12321)

loop do
  session = server.accept
  session.puts Time.new
  session.close
end
```

Обратите внимание, как просто использовать класс `TCPServer`. Обращение к методу `accept` блокирует программу, приостанавливая ее работу до получения запроса на подключение от клиента. Когда такой запрос приходит, метод возвращает подключенный сокет, который можно использовать в коде сервера.

Вот TCP-версия клиента:

```
require "socket"

session = TCPSocket.new("localhost", 12321)
puts "The time is #{session.gets.chomp}"
session.close
```

18.1.2. Реализация многопоточного сервера

Если показанный выше сервер одновременно получает несколько запросов, то следующий должен будет ждать в очереди, пока не закончится обслуживание предыдущего. В этом случае лучше обрабатывать каждый запрос в отдельном потоке.

Ниже показана реализация сервера текущего времени, с которым мы познакомились в предыдущем разделе. Он работает по протоколу TCP и создает новый поток для каждого запроса.

```
require "socket"

server = TCPServer.new(12321)

loop do
```

```
    Thread.new(server.accept) do |session|
      session.puts Time.new
      session.close
    end
end
```

Многопоточность позволяет достичь высокого параллелизма. Вызывать метод join не нужно, поскольку сервер исполняет бесконечный цикл, пока его не остановят вручную.

Код клиента, конечно, остался тем же самым. С точки зрения клиента, поведение сервера не изменилось (разве что он стал более надежным).

18.1.3. Пример: сервер для игры в шахматы по сети

Не всегда нашей конечной целью является взаимодействие с самим сервером. Иногда сервер – всего лишь средство для соединения клиентов между собой. В качестве примера можно привести файлообменные сети, столь популярные 10–15 лет назад. Другой пример – чат-серверы и разного рода игровые серверы.

Давайте напишем скелет шахматного сервера. Мы не имеем в виду программу, которая будет играть в шахматы с клиентом. Нет, наша задача – связать клиентов друг с другом, чтобы они могли затем играть без вмешательства сервера.

Предупреждаю, что ради простоты показанная ниже программа ничего не знает о шахматах. Логика игры просто заглушена, чтобы можно было сосредоточиться на сетевых аспектах.

Для установления соединения между клиентом и сервером будем использовать протокол TCP. Можно было бы остановиться и на UDP, но этот протокол ненадежен, и нам пришлось бы самостоятельно обрабатывать повторную передачу данных и прием пакетов, пришедших не по порядку.

Клиент может передать два поля: свое имя и имя желательного партнера. Для идентификации противника условимся записывать его имя в виде user:hostname; мы употребили двоеточие вместо напрашивающегося знака @, чтобы не вызывать ассоциаций с электронным адресом, каковым эта строка не является.

Когда от клиента приходит запрос, сервер сохраняет сведения о клиенте у себя в списке. Если поступили запросы от *обоих* клиентов, сервер посылает каждому из них сообщение, теперь у каждого клиента достаточно информации для установления связи с партнером.

Есть еще вопрос о выборе цвета фигур. Оба партнера должны как-то договориться о том, кто каким цветом будет играть. Для простоты предположим, что цвет назначает сервер. Первый обратившийся клиент будет играть белыми (и, стало быть, ходить первым), второй – черными.

Уточним. Компьютеры, которые первоначально было клиентами, начиная с этого момент общаются друг с другом напрямую, следовательно, один из них становится сервером. Но на эту семантическую тонкость я не буду обращать внимание.

Поскольку клиенты посылают запросы и ответы попеременно и сеанс связи включает много таких обменов, то будем пользоваться протоколом TCP. Следо-

вательно, клиент, который на самом деле играет роль «сервера», создает объект TCPServer, а клиент на другом конце - объект TCPSocket. Будем предполагать, что номер порта для одноранговой обмена данными заранее известен, как и номер порта, по которому клиент обращается к серверу (разумеется, эти порты различны).

Мы только что описали простой протокол прикладного уровня. Его можно было бы сделать и более хитроумным.

Сначала рассмотрим код сервера (листинг 18.1). Чтобы его было проще запускать из командной строки, создадим поток, который завершит сервер при нажатии клавиши Enter. Сервер многопоточный, он может одновременно обслуживать нескольких клиентов. Данные о пользователях защищены мьютексом, ведь теоретически несколько потоков могут одновременно попытаться добавить новую запись в список.

Листинг 18.1. Шахматный сервер

```
require "thread"
require "socket"

PORT = 12000

# Выход при нажатии клавиши Enter
waiter = Thread.new do
  puts "Нажмите Enter для завершения сервера."
  gets
  exit
end

$mutex = Mutex.new
$list = {}

def match?(p1, p2)
  return false unless @list[p1] && @list[p2]
  @list[p1][0] == p2 && @list[p2][0] == p1
end

def handle_client(sess, msg, addr, port, ipname)
  cmd, player1, player2 = msg.split

  # Примечание: от клиента мы получаем данные в виде user:hostname,
  # но храним их в виде user:address
  player1 << ":#{addr}"              # Добавить IP-адрес клиента

  user2, host2 = player2.split(":")
  host2 = ipname if host2 == nil
  player2 = user2 + ":" + IPSocket.getaddress(host2)

  if cmd != "login"
```

```
    puts "Ошибка протокола: клиент послал сообщение #{msg}"
  end

  @mutex.synchronize do
    $list[player1] = [player2, addr, port, ipname, sess]
  end

  if match?(player1, player2)
    notify_clients(player1, player2)
  end

def notify_clients(player1, player2)
  # Имена теперь переставлены: если мы попали сюда, значит
  # player2 зарегистрировался первым.
  p1, p2 = @mutex.synchronize do
    [@list.delete(player1), @list.delete(player2)]
  end

  p1name = player1.split(":")[0]
  p2name = player2.split(":")[0]

  # ID игрока = name:ip:color
  # Цвет: 0=белые, 1=черные
  p1id = "#{p1name}:#{p1[3]}:1"
  p2id = "#{p2name}:#{p2[3]}:0"

  sess2 = p2[4]
  sess2.puts p1id
  sess2.close

  sleep 0.2       # дадим серверу запуститься
  sess1 = p1[4]
  sess1.puts p2id
  sess1.close
end

# В этом простом примере завершаемся, не обрабатывая ошибки в потоках
Thread.abort_on_exception = true

server = TCPServer.new("0.0.0.0", PORT)
loop do
  Thread.new(session) do |sess|
    text = sess.gets
    print "Получено: #{text}"  # Чтобы знать, что сервер получил
    domain, port, ipname, ipaddr = sess.peeraddr
    handle_client sess, text, ipaddr, port, ipname
    sleep 1
  end
end
```

Метод handle_client сохраняет информацию о клиенте. Если запись о таком клиенте уже существует, то каждому клиенту посылается сообщение о том, где находится другой партнер. На этом обязанности сервера кончаются.

Клиент (листинг 18.2) оформлен в виде единственной программы. При первом запуске она становится TCP-сервером, а при втором – TCP-клиентом. Честно говоря, решение о том, что сервер будет играть белыми, совершенно произвольно. Вполне можно было бы реализовать приложение так, чтобы цвет не зависел от таких деталей.

Листинг 18.2. Шахматный клиент

```
require "socket"
require "timeout"

ChessServer     = '10.0.1.7'  # Заменить этот IP-адрес
ChessServerPort = 12000
PeerPort        = 12001
White, Black    = 0, 1

def draw_board(board)
  puts <<-EOF
    +--------------------------------+
    | Заглушка! Шахматная доска...   |
    +--------------------------------+
  EOF
end

def analyze_move(who, move, num, board)
  # Заглушка - черные всегда выигрывают на четвертом ходу
  if who == BLACK and num == 4
    move << "  Мат!"
  end
  true  # Еще одна заглушка - любой ход считается допустимым.
end

def get_move(who, move, num, board)
  ok = false
  until ok do
    print "\nВаш ход: "
    move = STDIN.gets.chomp
    ok = analyze_move(who, move, num, board)
    puts "Недопустимый ход" unless ok
  end
  sock.puts move
  move
end

def my_move(who, lastmove, num, board, opponent, sock)
```

```
  move = get_move(who, lastmove, num, board)
  sock.puts move
  draw_board(board)

  case move
  when "resign"
    puts "\nВы сдались. #{opponent} выиграл."
    true
  when /Checkmate/
    puts "\nВы поставили мат #{opponent}!"
    true
  else
   false
end

def other_move(who, move, num, board, opponent, sock)
  move = sock.gets.chomp
  puts "\nПротивник: #{move}"
  draw_board(board)

  case move
  when "resign"
    puts "\n#{opponent} сдался... вы выиграли!"
    true
  when /Checkmate/
    puts "\n#{opponent} поставил вам мат."
    true
  else
    false
  end
end

if ARGV[0]
  myself = ARGV[0]
else
  print "Ваше имя? "
  myself = STDIN.gets.chomp
end

if ARGV[1]
  opponent_id = ARGV[1]
else
  print "Ваш противник? "
  opponent_id = STDIN.gets.chomp
end

opponent = opponent_id.split(":")[0]  # Удалить имя хоста

# Обратиться к серверу
```

```
socket = TCPSocket.new(ChessServer, ChessServerPort)

response = nil

socket.puts "login #{myself} #{opponent_id}"
socket.flush
response = socket.gets.chomp
name, ipname, color = response.split ":"
color = color.to_i

if color == Black                       # Цвет фигур другого игрока
  puts "\nУстанавливается соединение..."

  server = TCPServer.new(PeerPort)
  session = server.accept

  str = nil
  begin
    timeout(30) do
      str = session.gets.chomp
      if str != "ready"
        raise "Ошибка протокола: получено сообщение о готовности #{str}"
      end
    end
  rescue TimeoutError
    raise "Не получено сообщение о готовности от противника."
  end

  puts "Ваш противник #{opponent}... у вас белые.\n"

  who = White
  move = nil
  board = nil               # В этом примере не используется
  num = 0
  draw_board(board)         # Нарисовать начальное положение для белых

  loop do
    num += 1
    won = my_move(who, move, num, board, opponent, session)
    break if won
    lost = other_move(who, move, num, board, opponent, session)
    break if lost
    end
  end
else                        # Мы играем черными
  puts "\nУстанавливается соединение..."

  socket = TCPSocket.new(ipname, PeerPort)
```

```
  socket.puts "ready"

  puts "Ваш противник #{opponent}... у вас черные.\n"

  who = Black
  move = nil
  board = nil               # В этом примере не используется
  num = 0
  draw_board(board)         # Нарисовать начальное положение

  loop do
    num += 1
    lost = other_move(who, move, num, board, opponent, socket)
    break if lost
    won = my_move(who, move, num, board, opponent, socket)
    break if won
  end

  socket.close
end
```

Я определил это протокол так, что черные посылают белым сообщение «ready», чтобы партнер знал о готовности начать игру. Затем белые делают первый ход. Ход посылается черным, чтобы клиент мог нарисовать такую же позицию на доске, как у другого игрока.

Повторю, это приложение ничего не знает о шахматах. Вместо проверки допустимости хода вставлена заглушка; проверка выполняется локально, то есть на той стороне, где делается ход. Никакой реальной проверки нет, заглушка всегда говорит, что ход допустим. Кроме того, мы хотим, чтобы имитация игры завершалась после нескольких ходов, поэтому мы написали программу так, что черные всегда выигрывают на четвертом ходу. Победа обозначается строкой «Checkmate!» в конце хода. Эта строка печатается на экране соперника и служит признаком выхода из цикла.

Помимо «традиционной» шахматной нотации (например, «P-K4»), существует еще «алгебраическая» нотация, которую многие предпочитают. Но написанный код вообще не имеет представления о том, какой нотацией мы пользуемся.

Поскольку это было несложно сделать, мы позволяем игроку в любой момент сдаться. Рисование доски тоже заглушено. Желающие могут реализовать грубый рисунок, выполненный ASCII-символами.

Метод `my_move` всегда относится к локальной стороне, метод `other_move` – к удаленной. В листинге 18.3 приведен протокол сеанса. Действия клиентов нарисованы друг против друга.

```
% ruby chess.rb Hal
Capablanca:deepthought.org


Устанавливается соединение...
Ваш противник Capablanca... у вас белые.
```

```
% ruby chess.rb
Hal:deepdoodoo.org


Устанавливается соединение...
Ваш противник Hal... у вас черные.
```

```
+------------------------------+
| Заглушка! Шахматная доска... |
+------------------------------+

Ваш ход: N-QB3
+------------------------------+
| Заглушка! Шахматная доска... |
+------------------------------+

Противник: P-K4
+------------------------------+
| Заглушка! Шахматная доска... |
+------------------------------+

Ваш ход: P-K4
+------------------------------+
| Заглушка! Шахматная доска... |
+------------------------------+

Противник: B-QB4
+------------------------------+
| Заглушка! Шахматная доска... |
+------------------------------+

Ваш ход: B-QB4
+------------------------------+
| Заглушка! Шахматная доска... |
+------------------------------+

Противник: Q-KR5
+------------------------------+
| Заглушка! Шахматная доска... |
+------------------------------+

Ваш ход: N-KB3
+------------------------------+
| Заглушка! Шахматная доска... |
+------------------------------+

Противник: QxP  Checkmate!
+------------------------------+
| Заглушка! Шахматная доска... |
+------------------------------+

Capablanca поставил вам мат.
```

```
+------------------------------+
| Заглушка! Шахматная доска... |
+------------------------------+

Противник: N-QB3
+------------------------------+
| Заглушка! Шахматная доска...
+------------------------------

Ваш ход: P-K4
+------------------------------
| Заглушка! Шахматная доска...
+------------------------------

Противник: P-K4
+------------------------------
| Заглушка! Шахматная доска...
+------------------------------

Ваш ход: B-QB4
+------------------------------+
| Заглушка! Шахматная доска... |
+------------------------------+

Противник: B-QB4
+------------------------------+
| Заглушка! Шахматная доска... |
+------------------------------+

Ваш ход: Q-KR5
+------------------------------+
| Заглушка! Шахматная доска... |
+------------------------------+

Противник: N-KB3
+------------------------------
| Заглушка! Шахматная доска...
+------------------------------

Ваш ход: QxP
+------------------------------+
| Заглушка! Шахматная доска... |
+------------------------------+

Вы поставили мат Hal!
```

18.2. Сетевые клиенты

Иногда сервер пользуется хорошо известным протоколом. В таком случае нам надо лишь спроектировать клиент, который общается с сервером на понятном тому языке.

В разделе 18.1 мы видели, что это можно сделать с помощью протоколов TCP или UDP. Но чаще применяются протоколы более высокого уровня, например HTTP или SNMP. Рассмотрим несколько примеров.

18.2.1. Получение истинно случайных чисел из веб

Всякий, кто пытается сгенерировать случайное число, пользуясь детерминированными средствами, безусловно, живет во грехе.

— Джон фон Нейман

В модуле Kernel есть функция rand, которая возвращает случайное число, но вот беда – число-то не является истинно случайным. Если вы математик, криптограф или еще какой-нибудь педант, то назовете эту функцию генератором *псевдослучайных* чисел, поскольку она пользуется алгебраическими методами для детерминированного порождения последовательности чисел. Случайному наблюдателю эти числа представляются случайными и даже обладают необходимыми статистическими свойствами, но рано или поздно последовательность начнет повторяться. Мы можем даже намеренно (или случайно) повторить ее, задав ту же самую затравку.

Но природные процессы считаются истинно случайными. Поэтому при розыгрыше призов в лотерее счастливчики определяются лототроном, который хаотично выбрасывает шары. Другие источники случайности – радиоактивный распад или атмосферный шум.

Есть источники случайных чисел и в веб. Один из них – сайт www.random.org, которым и станет предметом следующего примера.

Программа в листинге 18.4 моделирует подбрасывание пяти обычных (шестигранных) костей. Конечно, игровые фанаты могли бы увеличить число граней до 10 или 20, но тогда стало бы сложно рисовать ASCII-картинки.

Листинг 18.4. Случайное бросание костей

```
require 'net/http'

URL = "http://www.random.org/integers/"

def get_random_numbers(count=1, min=0, max=99)
   uri = URI.parse(URL)
   uri.query = URI.encode_www_form(
      col: 1, base: 10, format: "plain", rnd: "new",
      num: count, min: min, max: max
   )
   response = Net::HTTP.get_response(uri)
   case response
```

```
    when Net::HTTPOK
      response.body.lines.map(&:to_i)
    else
      []
    end
end

DICE_LINES = [
  "+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ ",
  "|     | |  *  | | *   | | * * | | * * | | * * | ",
  "|  *  | |     | |  *  | |     | |  *  | | * * | ",
  "|     | |  *  | |   * | | * * | | * * | | * * | ",
  "+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ "
]

DIE_WIDTH = DICE_LINES[0].length/6

def draw_dice(values)
  DICE_LINES.each do |line|
    for v in values
      print line[(v-1)*DIE_WIDTH, DIE_WIDTH]
      print " "
    end
    puts
  end
end

draw_dice(get_random_numbers(5, 1, 6))
```

Здесь мы воспользовались классом Net::HTTP для прямого взаимодействия с веб-сервером. Считайте, что эта программа – узкоспециализированный браузер. Мы формируем URL и пытаемся установить соединение; когда оно будет установлено, мы получаем ответ, возможно, содержащий некие данные. Если код ответа показывает, что ошибок не было, то можно разобрать полученные данные. Предполагается, что исключения будут обработаны вызывающей программой.

Посмотрим на вариацию этой идеи. Что если бы мы захотели использовать эти случайные числа в каком-нибудь приложении? Поскольку обслуживающая программа на стороне сервера позволяет указать количество возвращаемых чисел, то было бы логично сохранить их в буфере. Поскольку при обращении к удаленному серверу задержки неизбежны, то нужно сразу заполнить буфер во избежание лишних запросов по сети.

В листинге 18.5 эта мысль реализована. Буфер заполняется отдельным потоком и совместно используется всеми экземплярами класса. Размер буфера и «нижняя отметка» (@slack) настраиваются; какие значения задать в реальной программе, зависит от величины задержки при обращении к серверу и от того, как часто приложение выбирает случайное число из буфера.

Листинг 18.5. Генератор случайных чисел с буферизацией

```
require "net/http"
require "thread"

class TrueRandom

   URL = "http://www.random.org/integers/"

   def initialize(min = 0, max = 1, buffsize = 1000, slack = 300)
      @buffer = SizedQueue.new(buffsize)
      @min, @max, @slack = min, max, slack
      @thread = Thread.new { fillbuffer }
   end

   def fillbuffer
      count = @buffer.max - @buffer.size

      uri = URI.parse(URL)
      uri.query = URI.encode_www_form(
         col: 1, base: 10, format: "plain", rnd: "new",
         num: count, min: @min, max: @max
      )

      Net::HTTP.get(uri).lines.each do |line|
         @buffer.push line.to_i
      end
   end

   def rand
      if @buffer.size < @slack && !@thread.alive?
         @thread = Thread.new { fillbuffer }
      end

      @buffer.pop
   end

end

t = TrueRandom.new(1, 6, 1000, 300)
count = Hash.new(0)

10000.times do |n|
   count[t.rand] += 1
end

p count

# При одном прогоне:
# {4=>1692, 5=>1677, 1=>1678, 6=>1635, 2=>1626, 3=>1692}
```

18.2.2. Запрос к официальному серверу времени

Как и обещали, приведем программу для обращения к сетевому NTP-серверу (NTP – Network Time Protocol), в которой класс TCPSocket используется для подключения к публичному серверу и чтения данных о текущем времени:

```
require "socket"

resp = TCPSocket.new("time.nist.gov", 13).read
time = ts.split(" ")[2] + " UTC"
remote = Time.parse(time)

puts "Локальное: #{Time.now.utc.strftime("%H:%M:%S")}"
puts "Удаленное: #{remote.strftime("%H:%M:%S")}"
```

Мы устанавливаем соединение и читаем данные из сокета. Ответ сервера содержит дату юлианского календаря в формате YY-MM-DD, время в формате HH:MM:SS, количество дней до или после даты перехода на летнее время и прочие сведения. Мы читаем ответ целиком, выделяем из него время и печатаем его вместе с временем на локальной машине для сравнения.

Отметим, что в этом примере мы не пользуемся протоколом NTP – более сложным и предоставляющим дополнительные данные (например, задержку ответов). Вместо него мы работаем по гораздо более простому протоколу DAYTIME. Полная реализацию для NTP заняла бы слишком много места (впрочем, всего-то три страницы кода). При желании можете ознакомиться с ней, скачав gem-пакет net-ntp.

18.2.3. Взаимодействие с POP-сервером

Многие серверы электронной почты пользуются почтовым протоколом (Post Office Protocol – POP). Имеющийся в Ruby класс POP3 позволяет просматривать заголовки и тела всех сообщений, хранящихся для вас на сервере, и обрабатывать их, как вы сочтете нужным. После обработки сообщения можно удалить.

Для создания объекта класса Net::POP3 нужно указать доменное имя или IP-адрес сервера, номер порта по умолчанию равен 110. Соединение устанавливается только после вызова метода start (которому передается имя и пароль пользователя).

Вызов метода mails созданного объекта возвращает массив объектов класса POPMail. (Имеется также итератор each для перебора этих объектов.)

Объект POPMail соответствует одному почтовому сообщению. Метод header получает заголовки сообщения, а метод all – заголовки и тело (у метода all, как мы вскоре увидим, есть и другие применения).

Фрагмент кода стоит тысячи слов. Вот пример обращения к серверу с последующей распечаткой темы каждого сообщения:

```
require "net/pop"

pop = Net::POP3.new("pop.fakedomain.org")
pop.start("gandalf", "mellon")      # имя и пароль пользователя
pop.mails.each do |msg
  puts msg.header.grep /^Subject: /
end
```

Метод delete удаляет сообщение с сервера. (Некоторые серверы требуют, чтобы POP-соединение было закрыто методом finish, только тогда результат удаления становится необратимым.) Вот простейший пример фильтра спама:

```
require "net/pop"

pop = Net::POP3.new("pop.fakedomain.org")
pop.start("gandalf", "mellon")     # имя и пароль пользователя
pop.mails.each do |msg|
  if msg.all =~ /.*make money fast.*/
    msg.delete
  end
end
pop.finish
```

Отметим, что при вызове метода start можно также задавать блок. По аналогии с методом File.open, в этом случае открывается соединение, исполняется блок, а затем соединение закрывается.

Метод all также можно вызывать с блоком. В блоке просто перебираются все строки сообщения, как если бы мы вызвали итератор each для строки, возвращенной методом all.

```
# Напечатать все строки в обратном порядке... полезная штука!
msg.all { |line| print line.reverse }
# То же самое...
msg.all.each { |line| print line.reverse }
```

Методу all можно также передать объект. В таком случае для каждой строчки (line) в полученной строке (string) будет вызван оператор конкатенации (<<). Поскольку в различных объектах он может быть определен по-разному, то в результате такого обращения возможны самые разные действия:

```
arr = []          # Пустой массив
str = "Mail: "    # String
out = $stdout     # Объект IO

msg.all(arr)      # Построить массив строчек
msg.all(str)      # Конкатенировать с str
msg.all(out)      # Вывести на stdout
```

Наконец, покажем еще, как вернуть только тело сообщения, игнорируя все заголовки.

```
module Net
  class POPMail
    def body
      # Пропустить байты заголовка
      self.all[self.header.size..-1]
    end
  end
end
```

Этот модуль не обладает всеми свойствами `all`, но его можно расширить. Оставляю это вам в качестве упражнения.

Если вы предпочитаете протокол IMAP, а не POP3, обратитесь к разделу 18.2.5.

18.2.4. Отправка почты по протоколу SMTP

Это понял бы и пятилетний ребенок.
Дайте мне пятилетнего ребенка.

— Гроучо Маркс

Название «простой протокол электронной почты» (Simple Mail Transfer Protocol SMTP) не вполне правильно. Если он и «простой», то только по сравнению с более сложными протоколами.

Конечно, библиотека `smtp` скрывает от программиста большую часть деталей протокола. Но, на наш взгляд, эта библиотека интуитивно не вполне очевидна и, пожалуй, слишком сложна (надеемся, что в будущем это изменится). В этом разделе мы приведем несколько примеров, чтобы помочь вам освоиться.

В классе `Net::SMTP` есть два метода класса: `new` и `start`. Метод `new` принимает два параметра: имя сервера (по умолчанию localhost) и номер порта (по умолчанию 25).

Метод `start` принимает следующие параметры:

- *server* – доменное имя или IP-адрес SMTP-сервера, по умолчанию `"localhost"`;
- *port* – номер порта, по умолчанию 25;
- *domain* – доменное имя отправителя, по умолчанию `ENV["HOSTNAME"]`;
- *account* – имя пользователя, по умолчанию `nil`;
- *password* – пароль, по умолчанию `nil`;
- *authtype* – тип авторизации, по умолчанию `:cram_md5`.

Обычно большую часть этих параметров можно не задавать.

Если метод `start` вызывается «нормально» (без блока), то он возвращает объект класса `SMTP`. Если же блок задан, то этот объект передается прямо в блок.

У объекта `SMTP` есть метод экземпляра `sendmail`, который обычно и занимается всеми деталями отправки сообщения. Он принимает три параметра:

- *source* – строка или массив (или любой объект, у которого есть итератор `each`, возвращающий на каждой итерации одну строку);
- *sender* – строка, записываемая в поле «from» сообщения;
- *recipients* – строка или массив строк, описывающие одного или нескольких получателей.

Вот пример отправки сообщения с помощью методов класса:

```
require 'net/smtp'

msg = <<EOF
Subject: Разное
```

```
... пришла пора
Подумать о делах:
О башмаках, о сургуче,
Капусте, королях.
И почему, как суп в котле,
Кипит вода в морях.
EOF

Net::SMTP.start("smtp-server.fake.com") do |smtp|
   smtp.sendmail msg, 'walrus@fake1.com', 'alice@fake2.com'
end
```

Поскольку в начале строки находится слово `Subject:`, то получатель сообщения увидит тему `Разное`.

Имеется также метод экземпляра `start`, который ведет себя практически так же, как метод класса. Поскольку почтовый сервер определен в методе `new`, то задавать его еще и в методе `start` не нужно. Поэтому этот параметр пропускается, а остальные не отличаются от параметров, передаваемых методу класса. Следовательно, сообщение можно послать и с помощью объекта `SMTP`:

```
require 'net/smtp'

msg = <<EOF
Subject: Ясно и логично
"С другой стороны, - добавил Тарарам, -
если все так и было, то все именно так и было.
Если же все было бы так, то все не могло бы быть
не так. Но поскольку все было не совсем так, все
было совершенно не так. Ясно и логично!"
EOF

smtp = Net::SMTP.new("smtp-server.fake.com")
smtp.start
smtp.sendmail msg, 'tweedledee@fake1.com', 'alice@fake2.com'
```

Если вы еще не запутались, добавим, что метод экземпляра может принимать также блок:

```
require 'net/smtp'

msg = <<EOF
Subject: Моби Дик
Зовите меня Измаил.
EOF

addressees = ['reader1@fake2.com', 'reader2@fake3.com']

smtp = Net::SMTP.new("smtp-server.fake.com")
smtp.start do |obj|
  obj.sendmail msg, 'narrator@fake1.com', addressees
end
```

Как видно из примера, объект, переданный в блок (obj), не обязан называться так же, как объект, от имени которого вызывается метод (smtp). Воспользуюсь также случаем подчеркнуть, что несколько получателей можно представить в виде массива строк.

Существует еще метод экземпляра со странным названием ready. Он похож на sendmail, но есть и важные различия. Задаются только отправитель и получатели, тело же сообщения конструируется с помощью объекта adapter класса Net::NetPrivate::WriteAdapter, у которого есть методы write и append. Адаптер передается в блок, где может использоваться произвольным образом[1]:

```
require "net/smtp"

smtp = Net::SMTP.new("smtp-server.fake1.com")
smtp.start

smtp.ready("t.s.eliot@fake1.com", "reader@fake2.com") do |obj|
  obj.write "Пошли вдвоем, пожалуй.\r\n"
  obj.write "Уж вечер небо навзничь распяло,\r\n"
  obj.write "Как пациента под ножом наркоз...\r\n"
end
```

Отметим, что пары символов «возврат каретки», «перевод строки» обязательны (если вы хотите разбить сообщение на строчки). Читатели, знакомые с деталями протокола, обратят внимание на то, что сообщение «завершается» (добавляется точка и слово «QUIT») без нашего участия.

Можно вместо метода write воспользоваться оператором конкатенации:

```
smtp.ready("t.s.eliot@fake1.com", "reader@fake2.com") do |obj|
  obj << "В гостиной разговаривают тени\r\n"
  obj << "О Микеланджело Буонаротти.\r\n"
end
```

И еще одно небольшое усовершенствование: мы добавим метод puts, который вставит в сообщение символы перехода на новую строку:

```
class Net::NetPrivate::WriteAdapter
  def puts(args)
    args << "\r\n"
    self.write(*args)
  end
end
```

Новый метод позволяет формировать сообщение и так:

```
smtp.ready("t.s.eliot@fake1.com", "reader@fake2.com") do |obj|
  obj.puts "Мы были призваны в глухую глубину,"
  obj.puts "В мир дев морских, в волшебную страну,"
  obj.puts "Но нас окликнули - и мы пошли ко дну."
end
```

---

[1] В тексте приводится отрывок из стихотворения Т. Элиота «Песнь любви Дж. Альфреда Пруфрока», перевод В. Топорова.

Если всего изложенного вам не хватает, поэкспериментируйте самостоятельно. А если соберетесь написать новый интерфейс к протоколу SMTP, не стесняйтесь.

18.2.5. Взаимодействие с IMAP-сервером

Протокол IMAP нельзя назвать вершиной совершенства, но во многих отношениях он превосходит POP3. Сообщения могут храниться на сервере сколь угодно долго (с индивидуальными пометками «прочитано» и «не прочитано»). Для хранения сообщений можно организовать иерархию папок. Этих возможностей уже достаточно для того, чтобы считать протокол IMAP более развитым, чем POP3.

Для взаимодействия с IMAP-сервером предназначена стандартная библиотека net/imap. Естественно, вы должны сначала установить соединение с сервером, а затем идентифицировать себя с помощью имени и пароля:

```
require 'net/imap'

host = "imap.hogwarts.edu"
user, pass = "lupin", "riddikulus"

imap = Net::IMAP.new(host)
begin
  imap.login(user, pass)
  # Или иначе:
  # imap.authenticate("LOGIN", user, pass)
rescue Net::IMAP::NoResponseError
  abort "Не удалось аутентифицировать пользователя #{user}"
end

# Продолжаем работу...

imap.logout   # разорвать соединение
```

Установив соединение, можно проверить почтовый ящик методом examine; по умолчанию почтовый ящик в IMAP называется INBOX. Метод responses возвращает информацию из почтового ящика в виде хэша массивов (наиболее интересные данные находятся в последнем элементе массива). Показанный ниже код показывает общее число сообщений в почтовом ящике ("EXISTS") и число непрочитанных сообщений ("RECENT"):

```
imap.examine("INBOX")
total = imap.responses["EXISTS"].last     # всего сообщений
recent = imap.responses["RECENT"].last    # непрочитанных сообщений
imap.close                                # закрыть почтовый ящик
```

Отметим, что метод examine позволяет только читать содержимое почтового ящика. Если нужно удалить сообщения или произвести какие-то другие изменения, пользуйтесь методом select.

Почтовые ящики в протоколе IMAP организованы иерархически, как имена путей в UNIX. Для манипулирования почтовыми ящиками предусмотрены методы create, delete и rename:

```
imap.create("lists")
imap.create("lists/ruby")
imap.create("lists/rails")
imap.create("lists/foobar")

# Уничтожить последний созданный ящик:
imap.delete("lists/foobar")
```

Имеются также методы list (получить список всех почтовых ящиков) и lsub (получить список «активных» ящиков, на которые вы «подписались»). Метод status возвращает информацию о состоянии ящика.

Метод search находит сообщения, удовлетворяющие заданному критерию, а метод fetch возвращает запрошенное сообщение:

```
msgs = imap.search("TO","lupin")
msgs.each do |mid|
  env = imap.fetch(mid, "ENVELOPE")[0].attr["ENVELOPE"]
  puts "От #{env.from[0].name}     #{env.subject}"
end
```

Команда fetch в предыдущем примере выглядит так сложно, потому что возвращает массив хэшей. Сам конверт тоже представляет собой сложную структуру; некоторые методы доступа к нему возвращают составные объекты, другие – просто строки.

В протоколе IMAP есть понятия UID (уникального идентификатора) и порядкового номера сообщения. Обычно методы типа fetch обращаются к сообщениям по номерам, но есть и варианты (например, uid_fetch) для обращения по UID. У нас нет места объяснять, почему нужны обе системы идентификации, но если вы собираетесь серьезно работать с IMAP, то должны понимать различие между ними (и никогда не путать одну с другой).

Библиотека net/imap располагает разнообразными средствами для работы с почтовыми ящиками, сообщениями, вложениями и т.д. Дополнительные сведения см. в справочной документации.

18.2.6. Кодирование и декодирование вложений

Для вложения в почтовое сообщение или в сообщение, отправляемое в конференцию, файл обычно кодируется. Как правило, применяется кодировка base64, для работы с которой служит метод pack с аргументом m:

```
bin = File.read("new.gif")
str = [bin].pack("m")        # str закодирована
orig = str.unpack("m")[0]    # orig == bin
```

Старые почтовые клиенты работали с кодировкой uuencode / uudecode. В этом случае вложение просто добавляется в конец текста сообщения и ограничивается строками begin и end, причем в строке begin указываются также разрешения на доступ к файлу (которые можно и проигнорировать) и имя файла. Аргумент u метода pack позволяет представить строку в кодировке uuencode. Пример:

```
# Предположим, что mailtext содержит текст сообщения

filename = "new.gif"
bin = File.read(filename)
encoded = [bin].pack("u")

mailtext << "begin 644 #{filename}"
mailtext << encoded
mailtext << "end"
# ...
```

На принимающей стороне мы должны извлечь закодированную информацию и декодировать ее методом unpack:

```
# ...
# Предположим, что 'attached' содержит закодированные данные
# (включая строки begin и end)

lines = attached.split("\n")
filename = /begin \d\d\d (.*)/.scan(lines[0]).first.first
encoded = lines[1..-2].join("\n")
decoded = encoded.unpack("u")       # Все готово к записи в файл
```

Современные почтовые клиенты работают с почтой в формате MIME; даже текстовая часть сообщения обернута в конверт (хотя клиент удаляет все заголовки, прежде чем показать сообщение пользователю).

Подробное рассмотрение формата MIME заняло бы слишком много места, да и не относится к рассматриваемой теме. Но в следующем простом примере показано, как можно закодировать и отправить сообщение, содержащее текстовую часть и двоичное вложение. Двоичные куски обычно представлены в кодировке base64:

```
require 'net/smtp'

def text_plus_attachment(subject,body,filename)
  marker = "MIME_boundary"
  middle = "--#{marker}\n"
  ending = "--#{middle}--\n"
  content = "Content-Type: Multipart/Related; " +
            "boundary=#{marker}; " +
            "typw=text/plain"
  head1 = <<-EOF
MIME-Version: 1.0
#{content}
Subject: #{subject}
  EOF
  binary = File.read(filename)
  encoded = [binary].pack("m")    # base64
  head2 = <<EOF
Content-Description: "#{filename}"
Content-Type: image/gif; name="#{filename}"
```

```
Content-Transfer-Encoding: Base64
Content-Disposition: attachment; filename="#{filename}"

EOF

  # Возвращаем...
  head1 + middle + body + middle + head2 + encoded + ending
end

domain  = "someserver.com"
smtp    = "smtp.#{domain}"
user, pass = "elgar","enigma"

body = <<EOF
Это мое сообщение. Особо
говорить не о чем. Я вложил
небольшой GIF-файл.

        -- Боб
EOF
mailtext = text_plus_attachment("Привет...",body,"new.gif")
Net::SMTP.start(smtp, 25, domain, user, pass, :plain) do |mailer|
  mailer.sendmail(mailtext, 'fromthisguy@wherever.com',
                  ['destination@elsewhere.com'])
end
```

18.2.7. Пример: шлюз между почтой и конференциями

В онлайновых сообществах общение происходит разными способами. К наиболее распространенным относятся списки рассылки и конференции (новостные группы).

Но не каждый хочет подписываться на список рассылки и ежедневно получать десятки сообщений; кто-то предпочитает время от времени заходить в конференцию и просматривать новые сообщения. С другой стороны, есть люди, которым система Usenet кажется слишком медлительной, они хотели вы видеть сообщение, пока еще электроны не успели остыть.

Таким образом, мы имеем ситуацию, когда в сравнительно небольшом закрытом списке рассылки рассматриваются те же темы, что в немодерируемой конференции, открытой всему миру. В конце концов кому-то пришла в голову мысль организовать зеркало – шлюз между обеими системами.

Подобный шлюз подходит не к любой ситуации, но в случае списка рассылки Ruby он существовал с 2001 по 2011 год. Сообщения из конференции копировались в список, а сообщения, отправляемые в список рассылки, направлялись также и в конференцию.

Эта задача была решена Дэйвом Томасом (конечно, на Ruby), и с его любезного разрешения мы приводим код в листингах 18.6 и 18.7.

Но сначала небольшое вступление. Мы уже немного познакомились с тем, как отправлять и получать электронную почту, но как быть с конференциями Usenet?

Доступ к конференциям обеспечивает протокол NNTP (Network News Transfer Protocol – сетевой протокол передачи новостей). Кстати, создал его Ларри Уолл, который позже подарил нам язык Perl.

В Ruby нет «стандартной» библиотеки для работы с NNTP, но есть несколько gem-пакетов, предлагающих функциональность NNTP. В примерах ниже мы пользуемся библиотекой, которую написал один японский программист (известный нам только по псевдониму greentea).

В библиотеке nntp.rb определен модуль NNTP, содержащий класс NNTPIO. В этом классе имеются, в частности, методы экземпляра connect, get_head, get_body и post. Чтобы получить сообщения, необходимо установить соединение с сервером и в цикле вызывать методы get_head и get_body (мы, правда, немного упрощаем). Чтобы отправить сообщение, нужно сконструировать его заголовки, соединиться с сервером и вызвать метод post.

В приведенных ниже программах используется библиотека smtp, с которой мы уже познакомились. В оригинальной версии кода производится также протоколирование хода процесса и ошибок, но для простоты мы эти фрагменты опустили.

Файл params.rb нужен обеим программам. В нем описаны параметры, управляющие всем процессом зеркалирования: имена серверов, имена пользователей и т.д. Ниже приведен пример, который вы можете изменить самостоятельно. (Все доменные имена, содержащие слово «fake», очевидно, фиктивные.)

```
# Различные параметры, необходимые шлюзу между почтой и конференциями

module Params
  NEWS_SERVER = "usenet.fake1.org"       # имя новостного сервера
  NEWSGROUP   = "comp.lang.ruby"         # зеркалируемая конференция
  LOOP_FLAG   = "X-rubymirror: yes"      #  чтобы избежать циклов
  LAST_NEWS_FILE = "/tmp/m2n/last_news"  # номер последнего прочитанного сообщения
  SMTP_SERVER = "localhost"              # имя хоста для исходящей почты

  MAIL_SENDER = "myself@fake2.org"       # От чьего имени посылать почту
  # (Для списков, на которые подписываются, это должно быть имя
  # зарегистрированного участника списка.)

  MAILING_LIST = "list@fake3.org"        # Адрес списка рассылки
end
```

Модуль Params содержит лишь константы, нужные обеим программам. Большая их часть не нуждается в объяснениях, упомянем лишь парочку. Во-первых, константа LAST_NEWS_FILE содержит путь к файлу, в котором хранится идентификатор последнего прочитанного из конференции сообщения, эта «информация о состоянии» позволяет избежать дублирования или пропуска сообщений.

Константа LOOP_FLAG определяет строку, которой помечаются сообщения, уже прошедшие через шлюз. Тем самым мы препятствуем возникновению бесконечной рекурсии, а заодно гневу возмущенных обитателей сети, получивших тысячи копий одного и того же сообщения.

Возникает вопрос: «А как вообще почта поступает в программу mail2news?» Ведь она, похоже, читает из стандартного ввода. Автор рекомендует следующую настройку: сначала в файле .forward программы sendmail перенаправить всю входящую почту на программу procmail. Файл .procmail конфигурируется так, чтобы извлекать сообщения, приходящие из списка рассылки, и по конвейеру направлять их программе mail2news. Уточнить детали можно в документации, сопровождающей приложение RubyMirror (в архиве Ruby Application Archive). Если вы работаете не в UNIX, то придется изобрести собственную схему конфигурирования.

Ну а все остальное расскажет сам код, приведенный в листингах 18.6 и 18.7.

Листинг 18.6. Перенаправление почты в конференцию

```
# mail2news: Принимает почтовое сообщение и отправляет
# его в конференцию

require "nntp"
include NNTP

require "params"

# Прочитать сообщение, выделив из него
# заголовок и тело. Пропускаются только
# определенные заголовки.

HEADERS = %w{From Subject References Message-ID
             Content-Type Content-Transfer-Encoding Date}

allowed_headers = Regexp.new(%{^(#{HEADERS.join("|")}):})

# Прочитать заголовок. Допускаются только
# некоторые заголовки. Добавить строки Newsgroups
# и X-rubymirror.

head = "Newsgroups: #{Params::NEWSGROUP}\n"
subject = "unknown"
while line = gets
  exit if line =~ /^#{Params::LOOP_FLAG}/o # такого не должно быть
  break if line =~ /^\s*$/
  next if line =~ /^\s/
  next unless line =~ allowed_headers

  # вырезать префикс [ruby-talk:nnnn] из темы, прежде чем
  # отправлять в конференцию
  if line =~ /^Subject:\s*(.*)/
    subject = $1

    # Следующий код вырезает специальный номер ruby-talk
    # из начала сообщения в списке рассылки, перед тем
```

```
    # как отправлять его новостному серверу.

    line.sub!(/\[ruby-talk:(\d+)\]\s*/, '')
    subject = "[#$1] #{line}"
    head << "X-ruby-talk: #$1\n"
  end
  head << line
end

head << "#{Params::LOOP_FLAG}\n"


body = ""
while line = gets
  body << line
end

msg = head + "\n" + body
msg.gsub!(/\r?\n/, "\r\n")

nntp = NNTPIO.new(Params::NEWS_SERVER)
raise "Failed to connect" unless nntp.connect
nntp.post(msg)
```

Листинг 18.7. Перенаправление конференции в почту

```
##
# Простой скрипт для зеркалирования трафика из
# конференции comp.lang.ruby в список рассылки ruby-talk
#
# Вызывается периодически (скажем, каждые 20 минут).
# Запрашивает у новостного сервера все сообщения с номером,
# большим номера последнего сообщения, полученного в
# прошлый раз. Если таковые есть, то читает сообщения,
# отправляет их в список рассылки и запоминает номер
# последнего.

require 'nntp'
require 'net/smtp'
require 'params'

include NNTP

##
# Отправить сообщения в список рассылки. Сообщение должно
# быть отправлено участником списка, хотя в строке From:
# может стоять любой допустимый адрес.
#

def send_mail(head, body)
```

```
  smtp = Net::SMTP.new
  smtp.start(Params::SMTP_SERVER)
  smtp.ready(Params::MAIL_SENDER, Params::MAILING_LIST) do |a|
    a.write head
    a.write "#{Params::LOOP_FLAG}\r\n"
    a.write "\r\n"
    a.write body
  end
end

##
# Запоминаем идентификатор последнего прочитанного из конференции
# сообщения.

begin
  last_news = File.open(Params::LAST_NEWS_FILE) {|f| f.read} .to_i
rescue
  last_news = nil
end

##
# Соединяемся с новостным сервером и получаем номера сообщений
# из конференции comp.lang.ruby.
#
nntp = NNTPIO.new(Params::NEWS_SERVER)
raise "Failed to connect" unless nntp.connect
count, first, last = nntp.set_group(Params::NEWSGROUP)

##
# Если номер последнего сообщения не был запомнен раньше,
# сделаем это сейчас.

if not last_news
  last_news = last
end

##
# Перейти к последнему прочитанному ранее сообщению и
# попытаться получить следующие за ним. Это может привести
# к исключению, если сообщения с указанным номером не
# существует, но мы не обращаем на это внимания.

begin
  nntp.set_stat(last_news)
rescue
end

##
# Читаем все имеющиеся сообщения и отправляем каждое
```

```
# в список рассылки.

new_last = last_news

begin
  loop do
    nntp.set_next
    head = ""
    body = ""
    new_last, = nntp.get_head do |line|
      head << line
    end

    # Не посылать сообщения, которые программа mail2news
    # уже отправляла в конференцию ранее (иначе зациклимся).
    next if head =~ %r{^X-rubymirror:}

    nntp.get_body do |line|
      body << line
    end

    send_mail(head, body)
  end
rescue
end

##
# И записать в файл новую отметку.

File.open(Params::LAST_NEWS_FILE, "w") do |f|
  f.puts  new_last
end unless new_last == last_news
```

18.2.8. Получение веб-страницы с известным URL

Пусть нам нужно получить HTML-документ из веб. Возможно, вы хотите проверить контрольную сумму и узнать, не изменился ли документ, чтобы послать автоматическое уведомление. А, быть может, вы пишете собственный браузер, тогда это первый шаг на пути длиной в тысячу километров.

```
require "net/http"

begin
   uri = URI(http://www.space.com/index.html)
   res = Net::HTTP.get(uri)
rescue => err
   puts "Ошибка: #{err.class}: #{err.message}"
   exit
```

```
end

puts "Получено строк: #{res.body.lines.size} " \
     "и байтов: #{res.body.size}"
# Обработать...
```

Сначала мы создаем объект класса HTTP, указывая доменное имя и номер порта сервера (обычно используется порт 80). Затем выполняется операция get, которая возвращает ответ по протоколу HTTP и вместе с ним строку данных. В примере выше мы не проверяем ответ, но если возникла ошибка, то перехватываем ее и выходим.

Если мы благополучно миновали предложение rescue, то можем ожидать, что содержимое страницы находится в строке data. Мы можем обработать ее, как сочтем нужным.

Что может пойти не так, какие ошибки мы перехватываем? Несколько. Может не существовать или быть недоступным сервер с указанным именем; указанный адрес может быть перенаправлен на другую страницу (эту ситуацию мы не обрабатываем); может быть возвращена пресловутая ошибка 404 (указанный документ не найден). Обработку таких ошибок мы оставляем вам.

Следующий раздел также окажется полезным. В нем мы покажем несколько более простой способ решения данной задачи.

18.2.9. Библиотека Open-URI

Библиотеку Open-URI написал Танака Акира (Tanaka Akira). Ее цель – унифицировать работу с сетевыми ресурсами из программы, предоставив интуитивно очевидный и простой интерфейс.

По существу она является оберткой вокруг библиотек net/http, net/https и net/ftp, и предоставляет метод open, которому можно передать произвольный URI. Пример из предыдущего раздела можно было бы переписать следующим образом:

```
require 'open-uri'

site = open("http://www.marsdrive.com/")
data = site.read
puts "Получено строк: #{data.split.size}, байтов: #{data.size}"
```

Объект, возвращаемый методом open (f в примере выше), – не просто файл. У него есть также методы из модуля OpenURI::Meta, поэтому мы можем получить метаданные:

```
uri = f.base_uri            # объект URI с собственными методами доступа
ct  = f.content_type        # "text/html"
cs  = f.charset             # "utf-8"
ce  = f.content_encoding    # []
```

Библиотека позволяет задать и дополнительные заголовочные поля, передавая методу open хэш. Она также способна работать через прокси-серверы и обладает рядом других полезных функций. В некоторых случаях этой библиотеки недостаточно, например, если необходимо разбирать заголовки HTTP, буферизовать

очень большой скачиваемый файл, отправлять куки и т.д. Дополнительные сведения можно найти в онлайновой документации на сайте rdoc.info.

18.3. Заключение

Эта глава содержит введение в сетевое программирование на низком уровне. В частности, приведены простые примеры серверов и клиентов. Мы видели, как написать клиент для существующего сервера, созданного не нами.

Мы рассмотрели также протоколы более высокого уровня, как, например, POP и IMAP для получения почты. Аналогично мы говорили о протоколе отправки почты SMTP. Попутно мы продемонстрировали способ кодирования и декодирования вложений в почтовые сообщения. В контексте разработки шлюза между списком рассылки и конференциями мы упомянули о протоколе NNTP.

Пришло время внимательнее познакомиться с более узким вопросом, относящимся к данной теме. В настоящее время один из самых важных видов сетевого программирования – разработка для веб, которой и посвящена следующая глава.

Глава 19. Ruby и веб-приложения

Как ловко мы сплетаем сеть...!

— Сэр Вальтер Скотт, «Мармион»

За последние лет десять Ruby завоевал невероятную популярность (кто-то может даже сказать, что незаслуженно). Хотя Ruby – универсальный язык, на котором можно писать любые программы, своей славой он обязан, прежде всего, одной-единственной библиотеке – Ruby on Rails. Созданная Дэвидом Хейнемейером Хансоном и впервые выпущенная в 2004 году, Rails (как ее раньше называли) позволяет создавать функционально насыщенные интерактивные веб-приложения с невиданной дотоле скоростью.

В этой главе мы узнаем, как отвечать на HTTP-запросы от браузера или другого веб-клиента (например, мобильного приложения или какой-то веб-службы). Мы кратко расскажем о достоинствах таких веб-каркасов, как Rails и Sinatra, и рассмотрим многочисленные инструменты, которые интегрируются с этими каркасами, чтобы упростить работу с хранилищами данных, HTML, CSS и JavaScript. Наконец, мы остановимся на том, как разрабатывать на Ruby свой API и потреблять API, написанные другими людьми. И еще поговорим о том, как без особых усилий создавать сложные статические сайты.

Ну а начнем с HTTP-серверов.

19.1. HTTP-серверы

Доля HTTP-серверов среди используемых сегодня программ очень высока. Они предоставляют интерфейсы к средствам коммуникации, документам, финансам, системам планирования путешествий, да чуть ли не ко всем сторонам современной жизни. Но каким бы сложным ни было приложение, в его основе всегда лежит очень простой протокол HTTP. По существу, HTTP сводится к нескольким строкам обычного текста, описывающим запросы и ответы, посылаемые через TCP-сокет.

19.1.1. Простой HTTP-сервер

Чтобы продемонстрировать простоту HTTP, напишем на Ruby HTTP-сервер, к которому можно обратиться из любого браузера. Попробуйте запустить программу, показанную в листинге 19.1, а потом открыть URL `http://localhost:3000` в своем браузере.

Листинг 19.1 Простой HTTP-сервер

```
require 'socket'
server = TCPServer.new 3000

loop do
   # Принять запрос на подключение
   socket = server.accept

   # Напечатать запрос, заканчивающийся пустой строкой
   puts line = socket.readline until line == "\r\n"

   # Сообщить браузеру, что он общается с HTTP-сервером
   socket.write "HTTP/1.1 200 OK\r\n"

   # Сообщить браузеру, что длина тела составляет 52 байта
   socket.write "Content-Length: 51\r\n"

   # Пустая строка, отделяющая заголовки от тела
   socket.write "\r\n"

   # Вывести HTML-код, составляющий тело
   socket.write "<html><body>"
   socket.write "<h1>Hello from Ruby!</h1>"
   socket.write "</body></html>\n"

   # Закрыть соединение
   socket.close
end
```

Здесь мы принимаем входящие запросы на подключение по протоколу TCP. Затем печатаем каждую строку, отправленную браузером, пока не встретим пустую строку. Далее мы отправляем *HTTP-ответ* – простой текст в определенном формате, понятном браузеру.

Ответ состоит из кода состояния, заголовков и тела. В нашем случае посылается код состояния 200, означающий, что ни запрос, ни ответ не содержат ошибок. Есть много других кодов состояния, в том числе перенаправление на другой адрес (301 и 302), отсутствие данных по указанному адресу (404) и даже ошибка сервера (500). Полный перечень можно найти в сетевых справочниках, например в википедии или на сайте httpstatus.es.

Вслед за кодом состояния мы посылаем один заголовок, сообщающий браузеру, сколько байтов будет в следующем далее теле. Могут быть и другие заголовки, в которых указывается, например, какого рода информация содержится в теле (HTML, JSON, XML или даже видео или аудио), является ли тело сжатым и т. д.

Далее мы посылаем пустую строку, означающую, что заголовки кончились и начинается тело. Вероятно, вы обратили внимание, что строка заканчивается не одним символом перевода строки `\n`, а парой `\r\n` (возврат каретки и перевод строки). Одиночный символ перевода строки применяется для завершения строк в си-

стемах UNIX и Mac OS X, а пара символов – в системе Windows и некоторых других, а также в запросах и ответах по протоколу HTTP.

Если теперь запустить сервер и перейти по адресу `localhost:3000` в браузере, то появится строка «Hello from Ruby!», напечатанная крупным полужирным шрифтом. А на экране терминала вы увидите текст, который браузер посылает серверу. На моей машине было напечатано:

```
GET / HTTP/1.1
Host: localhost:3000
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
  */*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-us
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_4)
AppleWebKit/537.78.2 (KHTML, like Gecko)
Version/7.0.6 Safari/537.78.2
Cache-Control: max-age=0
Connection: keep-alive
```

Это и есть HTTP-запрос. Как видите, он очень похож на HTTP-ответ, сформированный нами в листинге 19.1. Запрос начинается строкой объявления, в данном случае это запрос `GET` к пути `/`. Заголовок `Host` определяет доменное имя, связанное с данным запросом, – в нашем случае `localhost`. Различные заголовки `Accept` говорят, что браузер желал бы получить ответ в формате HTML, XHTML или XML, что ответ может быть сжатым и должен быть на английском языке. Заголовок `User-Agent` сообщает, что запрос был отправлен из браузера Safari версии 7.0.6.

19.1.2. Rack и веб-серверы

Любой HTTP-сервер принимает запросы, подобные рассмотренному выше, и отправляет ответы так, как это сделали мы. В 2007 году Кристиан Нойкирхен (Christian Neukirchen) написал библиотеку Rack, которая предоставляет единый API для всех написанных на Ruby HTTP-серверов и веб-приложений.

Имейте в виду, что Rack предназначена не для высокоуровневой разработки, а для создания инструментов, библиотек и каркасов. Кроме того, она рассчитана на опытных программистов, а пользователи каркаса или библиотеки обычно не видят Rack API. Если разобраться в назначении и способах работы с Rack, то в ваших руках окажется мощнейший инструмент.

На мой взгляд, Rack – просто гениальное изобретение. За несколько лет, прошедших с момента ее появления, написанные на ее основе программы обработали много триллионов запросов.

Рассмотрим некоторые детали работы этой библиотеки. HTTP-запрос, как минимум, включает набор параметров, организованный в виде подобия хэша. Возвращаемый ответ состоит из трех частей: код состояния, набор заголовков и тело.

Допустим, что у объекта имеется метод `call`, который возвращает массив такого формата. Любой класс, следующий описанным правилам, можно считать Rack-приложением. Вот простейший пример такого класса:

```
class MyRackApp
   def call(env)
      [200, {'Content-type' => 'text/plain'}, ["Welcome to Rack!"]]
   end
end
```

Отметим, что тело само является массивом, а не просто строкой. Объясняется это соглашением, по которому тело должно отвечать на вызов метода each. (На самом деле, простая строка тоже сгодилась бы, если бы она отвечала на вызов each, как было в старых версиях Ruby.)

Ну и как запустить такое приложение? Способов несколько. В состав Rack входит утилита rackup. По сути дела, это исполнитель приложений, умеющий работать с файлами с расширением .ru. Ниже приведен пример такого файла, который просто вызывает приложение, передавая методу run объект в качестве параметра.

```
# Предполагается приведенное выше определение
app = MyRackApp.new
run app
```

По соглашению, Rack-приложение конфигурируется с помощью написанного на Ruby файла config.ru. Пользуясь несложным Rack API, мы можем сильно упростить HTTP-сервер. В приведенном ниже фрагменте тот же самый HTTP-ответ посылается средствами Rack. Обратите внимание, насколько лаконичнее стрелочка, обозначающая лямбда-выражение:

```
run -> (env) { [
   200, {"Content-Type" => "text/html"}, ["<html><body>",
      "<h1>Hello from Ruby!</h1>", "</body></html>"]
] }
```

Любое Rack-приложение должно отвечать на обращения к методу call, который вызывается один раз для каждого запроса. Аргумент env, передаваемый методу call, содержит информацию о запросе: глагол HTTP, путь, заголовки, IP-адрес клиента и многое другое.

Поскольку любой Proc-объект умеет отвечать на вызов call, простейшим Rack-приложением является лямбда, как в примере выше. Каждое Rack-приложение должно возвращать Rack-ответ: массив из трех элементов, содержащий код состояния (число), заголовки ответа (в виде хэша) и тело (в виде объекта, отвечающего на вызовы each; в нашем случае это массив).

Важная концепция, благодаря которой библиотека Rack и является столь полезной, – идея *ПО промежуточного уровня*. Можно считать, что это некая программа, расположенная между браузером (или клиентом) и сервером. Она обрабатывает запросы, посылаемые клиентами серверу, и ответы сервера клиентам.

При этом компоненты промежуточного уровня можно более-менее произвольно «сцеплять». Каждый следующий компонент каким-то образом преобразует поток данных. Можно рассматривать это как аналог конвейера (|) в UNIX.

Существуют соглашения и приемы написания компонентов промежуточного уровня, но здесь мы их рассматривать не будем. Отметим лишь, что имеется великое множество готовых компонентов – как входящих в состав Rack, так и написан-

ных сторонними разработчиками. Они реализуют, например, HTTP-кэширование, фильтрацию спама, проверку работоспособности отправкой периодического сигнала-пульса, аналитику Google и многое другое.

Если вы собираетесь всерьез использовать Rack, то должны познакомиться с классом `Rack::Builder`; это предметно-ориентированный язык для сборки Rack-приложений из отдельных кусочков. Утилита rackup преобразует `ru`-файлы в объект `Builder` «за кулисами», но, если требуется большая гибкость и дополнительные возможности, то вы можете сделать это и самостоятельно.

Для запуска Rack-приложения можно использовать любой из многочисленных совместимых с Rack HTTP-серверов, написанных на Ruby. Этим вы избавите себя от необходимости писать и поддерживать TCP-сервер, корректно разбирать HTTP-запросы и форматировать HTTP-ответы. Если никакой другой сервер не установлен, то Rack по умолчанию возьмет сервер Webrick, входящий в состав дистрибутива Ruby, но для производственных приложений он не годится. Самый популярный написанный на Ruby веб-сервер – Unicorn, он оптимизирован под однопоточные приложения для интерпретатора MRI. Дополнительные сведения о Unicorn можно найти в документации на сайте `unicorn.bogomips.org`. Лично я предпочитаю веб-сервер Puma, который поддерживает многопоточные приложения, а также интерпретаторы MRI, JRuby и Rubinius. Дополнительную информацию о нем см. на сайте `puma.io`.

Обратите также внимание на сообщения, печатаемые на консоли сервером Puma и библиотекой Rack. Сервер выводит одну строку для каждого запроса, которая включает IP-адрес отправителя, дату и время, глагол HTTP, версию протокола HTTP, возвращенный код состояния и время, потраченное на обработку запроса. Вот что печатается на консоли после запуска Puma и отправки одного запроса:

```
Puma starting in single mode...
* Version 2.9.0 (ruby 2.1.2-p95), codename: Team High Five
* Min threads: 0, max threads: 16
* Environment: development
* Listening on tcp://0.0.0.0:9292
Use Ctrl-C to stop
127.0.0.1 - - [02/Sep/2014 16:32:26] "GET / HTTP/1.1" 200 - 0.0001
```

При работе с Rack можно получить всю ту информацию, которую мы получаем при самостоятельном чтении запросов из TCP-сокета, но усилий для этого приходится прилагать не в пример меньше. Достаточно просто прочитать из хэша `env` значение того или иного ключа. Над Rack и его унифицированным серверным API надстроены различные каркасы приложений и, в частности, Rails.

19.2. Каркасы приложений

Вы наверняка понимаете, что веб-приложение очень быстро может стать чрезмерно сложным и громоздким, даже при использовании абстракций, предоставляемых Rack. И вот здесь-то вступают в игру каркасы для построения веб-приложений типа Rails и Sinatra. Веб-каркас предлагает, как минимум, дополнительный уро-

вень абстракции, который позволяет вызывать различные методы в зависимости от глагола HTTP, пути и других условий.

Ruby on Rails – каркас, в который «все включено». В него входит командная утилита для создания и запуска Rails-приложений и многочисленные библиотеки, отвечающие за такие аспекты, как хранение данных, отрисовка HTML-шаблонов, управление CSS и JavaScript и многие другие.

Каркас Sinatra гораздо меньше по объему, чуть ли не единственная его функциональность – отображение между URL и Ruby-кодом. Все дополнительные возможности предоставляются другими библиотеками (на самом деле, во многих приложениях Sinatra используются части Rails). В этом разделе мы приведем примеры использования Rails и Sinatra, а в оставшейся части главы рассмотрим дополнительные инструменты и библиотеки, интегрирующиеся с Rails (и допускающие использование из Sinatra).

Существуют и другие каркасы веб-приложений для Ruby, в том числе почтенный Ramaze, совсем новенький Lotus и т. д. Если вас интересуют различные способы построения веб-каркасов, определенно стоит изучить несколько вариантов. Если же ваши интересы ограничиваются только разработкой собственно веб-приложений, то я рекомендую пользоваться Rails. Это самый популярный каркас с максимальным количеством пользователей, для него существует масса примеров, решений и пособий, так что вам точно будет куда обратиться за помощью в случае необходимости.

Получив запрос, каркас веб-приложений должен первым делом решить, какой код вызвать для его обработки. Это обычно называется маршрутизацией, соответствующие механизмы есть и в Rails, и в Sinatra, хотя устроены они по-разному. Сначала рассмотрим Sinatra, а потом перейдем к Rails.

Примеры из этой главы были протестированы для версий Sinatra 1.4.5 и Rails 4.1.5 работающих с версией Ruby 2.1.2. Если с тех пор вышли новые версии, то ситуация может измениться. При необходимости вы можете установить указанные версии и выполнить примеры.

19.2.1. Маршрутизация в Sinatra

Gem-пакет Sinatra надстроен над библиотекой Rack и добавляет к ней сравнительно немного функциональности. Приложения Sinatra отображают указанные в запросе глаголы HTTP и пути на соответствующие блоки кода. В листинге 19.2 создается приложение Sinatra, эквивалентное показанному ранее.

Впрочем, мы пошли немного дальше и добавили еще один путь. Не будь Sinatra, путь пришлось бы анализировать самостоятельно и выполнять разный код в зависимости от того, что указано в запросе. Sinatra делает это автоматически, мы должны лишь определить соответствие между комбинацией глагола и пути и написанным на Ruby кодом.

Листинг 19.2. Приложение Sinatra

```
require 'sinatra'

get "/" do
  <<-HTML
```

```
      <html>
        <body>
          <h1>Hello from Ruby!</h1>
        </body>
      </html>
  HTML
end

get "/world" do
  <<-HTML
      <html>
        <body>
          <h1>Hello, World!</h1>
        </body>
      </html>
  HTML
end
```

Чтобы выполнить этот пример, сохраните показанный в листинге 19.2 код в файле `sinatra.rb`. При необходимости установите gem-пакет Sinatra командой `gem install sinatra`, после чего запустите сервер командой `ruby sinatra.rb`. Теперь, перейдя в браузере по адресу `http://localhost:4567`, вы увидите первоначальное сообщение, а перейдя по адресу `http://localhost:4567/world` – сообщение «Hello, World!».

На консоль приложение Sinatra выводит почти то же самое, что приложение Rack. Единственное мелкое отличие – вторая строка журнала, относящаяся к запросу к URL `"/world"`.

```
Puma 2.9.0 starting...
* Min threads: 0, max threads: 16
* Environment: development
* Listening on tcp://localhost:4567
== Sinatra/1.4.5 has taken the stage on 4567 for development with
backup from Puma
127.0.0.1 - - [02/Sep/2014 17:36:33] "GET / HTTP/1.1" 200 51 0.0047
127.0.0.1 - - [02/Sep/2014 17:36:38] "GET /world HTTP/1.1" 200 48
0.0004
```

19.2.2. Маршрутизация в Rails

Теперь рассмотрим, как тот же самый пример реализуется в Rails. Сам gem-пакет Rails содержит совсем немного кода, т. к. Rails объединяет в единое целое другие библиотеки, содержащие четко очерченную функциональность. Класс `ActionController`, входящий в состав gem-пакета `action_pack`, подходит к задаче маршрутизации иначе, чем в Sinatra. В Rails вызывается код, находящийся внутри *контроллера*, а в файле `routes.rb` указывается, какой метод какого контроллера соответствует данному пути.

Реализуем наш пример в виде небольшого приложения Rails. Для начала установите каркас Rails и все его компоненты командой `gem install rails`, если не сделали это раньше. Затем выполните команду `rails new hello`. В результате будет создан

каталог hello, содержащий заготовку приложения Rails. На данный момент большая часть созданных файлов нас не интересует, а если вы хотите углубиться в тему, почитайте руководство «Getting Started with Rails» на сайте guides.rubyonrails.org.

Перейдите в каталог вновь созданного приложения командой cd hello и выполните команду bin/rails generate controller hello. Будет создано несколько файлов, из которых нам интересен только app/controllers/hello_controller.rb. Откройте его в редакторе и введите такой код:

```
class HelloController < ApplicationController
   def index
      render inline: "<h1>Hello from Ruby!</h1>"
   end

   def world
      render inline: "<h1>Hello, World!</h1>"
   end
end
```

Мы добавили в класс HelloController два метода – index и world. Имена могут быть любыми, с путем они связываются при настройке маршрутизации. Чтобы добавить маршруты, нужно отредактировать файл config/routes.rb. В этом заранее сгенерированном файле содержатся подробные пояснения по поводу маршрутов разного вида, но для нашего примера достаточно таких четырех строк:

```
Rails.application.routes.draw do
   get "/" => "hello#index"
   get "/world" => "hello#world"
end
```

Объявление маршрута в Rails состоит из вызова метода, имя которого совпадает с маршрутизируемым глаголом HTTP. Этому методу передается хэш, в котором ключом является путь, а значением – имя контроллера и метода в нем в формате name#index. Как нетрудно догадаться, первый из показанных выше маршрутов означает, что запрос к пути "/" должен обслуживаться методом index контроллера hello, а запрос к пути "/world" методом world того же контроллера.

Чтобы выполнить этот пример, введите команду bin/rails server. Вы уже, наверное, поняли, что команда rails – портал ко всем типичным задачам, встречающимся при создании веб-приложений для Rails. Когда сервер запустится, перейдите в браузере по адресу http://localhost:3000 и посмотрите, что возвращает наш пример. Для просмотра второго варианта перейдите по адресу http://localhost:3000/world.

Вывод в окне терминала теперь выглядит по-другому: для каждого запроса печатается несколько строк. В них представлены те же элементы, что и в предыдущих примерах: глагол HTTP, указанный в запросе путь, IP-адрес отправителя, возвращенный код состояния и время обработки запроса:

```
=> Booting WEBrick
=> Rails 4.1.5 application starting in development on http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
```

```
[2014-09-02 17:29:32] INFO WEBrick 1.3.1
[2014-09-02 17:29:32] INFO ruby 2.1.2 (2014-05-08) [x86_64-darwin13.0]
[2014-09-02 17:29:32] INFO WEBrick::HTTPServer#start: pid=20372 port=3000

Started GET "/" for 127.0.0.1 at 2014-09-02 17:29:32 -0700
Processing by HelloController#index as HTML
   Rendered inline template (0.6ms)
Completed 200 OK in 10ms (Views: 3.2ms | ActiveRecord: 0.0ms)

Started GET "/world" for 127.0.0.1 at 2014-09-02 17:29:39 -0700
Processing by HelloController#world as HTML
   Rendered inline template (0.2ms)
Completed 200 OK in 1ms (Views: 0.4ms | ActiveRecord: 0.0ms)
```

Но вторая и третья строки содержат новую информацию: имя контроллера и метода, вызванного в ответ на запрос, в данном случае `HelloController#index` и `#world`. Мы видим также, что для генерации ответа отрисовывался *встроенный шаблон* (inline template). Шаблоны мы будем обсуждать в разделе 19.4.

Помимо пути, в HTTP-запросе могут быть указаны *параметры* – наборы ключей и значений, формируемые браузером обычно на основе данных, введенных пользователем. И в Sinatra, и в Rails параметры доступны через хэш `params`. В этот хэш элементы попадают либо из URL, либо из *тела запроса*, если запрос отправлен с помощью глагола `POST`. С помощью параметров мы можем сделать наши примеры *динамичными*, т. е. результат будет зависеть от отправленных клиентом данных.

19.2.3. Параметры в Sinatra

Добавим в приложение Sinatra еще пример, в котором *строка запроса* будет содержать один параметр, следующий после знака `?`. Откройте сохраненный ранее файл `sinatra.rb` (листинг 19.3) и допишите в конец такой код:

```
get "/hello/:name" do
   body = "<html><body><h1>Hello"
   body << " from " << CGI.h(params[:from]) if params[:from]
   body << ", " << CGI.h(params[:name])
   body << "!</h1></html></body>"
end
```

Здесь наличие параметра в URL обозначается двоеточием, как символ в Ruby. Теперь Sinatra знает, что при разборе URL нужно распознавать пути вида `/hello/Julia` или `/hello/Tim`. Та часть пути, которая соответствует параметру `:name`, помещается в хэш `params` с ключом `:name`. Другой используемый параметр, `from`, не является частью пути. Это означает, что он должен встречаться в строке запроса, например: `/hello/Julia?from=Ruby`.

Любой параметр, который включается в возвращаемый HTML-код, мы экранируем с помощью метода `CGI.h`. Метод `h` – синоним `escape_html`. Он принимает на входе строку и возвращает строку, в которой все специальные символы HTML экранированы. Это абсолютно необходимо с точки зрения безопасности, поэто-

му так нужно поступать всякий раз, как приложение включает в HTML-страницу текст, полученный от пользователя.

Модифицировав файл, запустите сервер командой `ruby sinatra.rb` и проверьте, что получилось. Поэкспериментируйте с различными URL, задавайте разные значения `name` и `from` – и следите за результатом. Обратите внимание, что если параметр `from` отсутствует, то в ответе не сообщается, от кого пришел привет, а если отсутствует параметр `name` (например, путь содержит только `/hello/`), то возвращается сообщение об ошибке.

19.2.4. Параметры в Rails

Обработка параметров в Rails и в Sinatra очень похожа. Добавьте в файл `app/controllers/hello_controller.rb` метод `hello` (определение метода должно располагаться внутри блока `class/end`).

```
def name
   response = "<h1>Hello"
   response << " from " << h(params[:from]) if params[:from]
   response << ", " << h(params[:name])
   response << "!</h1>"
   render inline: response
end
```

Затем откройте файл `config/routes.rb` и добавьте следующую строку перед последним `end`:

```
get "/hello/:name" => "hello#name"
```

Как и раньше, имя метода (`name`) не имеет значения, но, с моей точки зрения, проще запомнить, что путь `/hello/:name` ведет на контроллер `Hello` и метод `name`. Снова запустите сервер командой `bin/rails server` и перейдите в браузере по адресу `http://localhost:3000/hello/Zelda?from=Ruby`. Вы увидите тот же ответ, что и раньше.

Обратите внимание, что на консоли появилась новая строчка, содержащая параметры запроса. Sinatra не протоколирует эту информацию для каждого запроса, а показывает только путь. Rails протоколирует и путь, и параметры:

```
Parameters: {"name"=>"Zelda"}
Parameters: {"from"=>"Ruby", "name"=>"Zelda"}
```

Ни Sinatra, ни Rails сказанным отнюдь не исчерпываются, но погружаться слишком глубоко мы не можем из-за нехватки места. Дополнительные сведения о Sinatra можно найти на сайте `sinatrarb.com` или в книге Alan Harris, Konstantin Haase «Sinatra: Up and Running».

Мы еще немного разовьем пример для Rails в следующих разделах, а полную документацию вы можете найти в Интернете по адресу `rubyonrails.org/documentation` или в книге Obie Fernandez, Kevin Faustino «The Rails 4 Way».

Познакомившись с основами маршрутизации и обработки параметров в Sinatra и Rails, перейдем к сохранению данных с помощью библиотеки ActiveRecord для Rails.

19.3. Хранение данных

В веб-приложениях сохранять данные труднее, чем в других программах Ruby, потому что HTTP, по определению, является протоколом без сохранения состояния. Это означает, что от HTTP-сервера не требуется сохранять какую-либо информацию между запросами. Причины технические и обусловлены, в основном, простым соображением: одно и то же приложение может работать на нескольких машинах, но получаемый клиентом ответ не должен зависеть от того, какой машиной обслужен запрос.

Это позволяет одновременно обрабатывать гораздо больше запросов (поскольку следующий запрос может быть направлен любому серверу, а, значит, работа распараллеливается), но также означает, что никакой сервер не вправе сохранять какую-либо информацию (например, учетные данные пользователя) в переменных или в локальных файлах. Переменные доступны только внутри одного процесса Ruby, а файл – только на одной машине, так что ни то, ни другое не годится.

Решение проблемы – использовать *базу данных* или *хранилище данных*. В разделе 10.3 мы рассматривали работу в PostgreSQL, MySQL и Redis. Чем повторять уже сказанное, мы лучше воспользуемся библиотекой ActiveRecord, которая позволяет создавать, читать, обновлять и удалять данные из базы с помощью объектов, не прибегая напрямую к SQL-запросам.

19.3.1. Базы данных

Библиотека ActiveRecord умеет подключаться к различным базам данных SQL, включая PostgreSQL, MySQL, Sqlite3 и даже Oracle и Microsoft SQL. Каждая таблица базы данных представлена классом, названным по имени таблицы, который наследует классу `ActiveRecord::Base`. Каждая строка таблицы представлена объектом этого класса, имеющим методы чтения и записи для каждого столбца.

В этом разделе мы наделим наше приложение Rails способностью запоминать имена всех, кто его приветствовал, вместе с информацией о том, откуда был отправлен привет, если в запросе присутствует параметр `from`. В следующем разделе «Генерация HTML» мы добавим форму с текстовыми полями, содержащими сведения о том, кому и откуда посылается привет.

Прежде всего, необходимо сгенерировать *модель* с именем `Greeting`. Классы модели наследуют `ActiveRecord::Base` и работают с одноименной таблицей базы данных. Для создания модели (и файла, который добавит одноименную таблицу в базу данных, если его выполнить) введите команду `bin/rails generate resource greeting to:string from:string`. Будет сгенерировано несколько файлов, но нас в данный момент интересуют только два.

Первый файл находится в каталоге `db/migrations`, его имя начинается временной меткой и заканчивается строкой `create_greetings.rb`. Вот что должно быть в этом файле:

```
class CreateGreetings < ActiveRecord::Migration
  def change
    create_table :greetings do |t|
```

```
            t.string :to
            t.string :from
            t.timestamps
        end
    end
end
```

Теперь выполните команду `bin/rake db:migrate`. Она создаст базу данных SQL для приложения, а также файл миграции с кодом для создания таблицы `greetings`, содержащей столбцы `to` и `from`. В обоих столбцах будут храниться строки. После успешного завершения этой команды можно изменить метод `name` в файле `app/controllers/hello_controller.rb`. Сделайте его таким:

```
def name
    greeting = Greeting.create!(to: params[:name],
                                from: params[:from])
    render inline: "<h1>" << CGI.h(greeting.to_s) << "!</h1>"
end
```

Здесь класс `Greeting` используется для создания новой строки в таблице базы данных, причем в столбцы `to` и `from` записываются значения соответственных параметров из хэша `params`. Затем для создания ответа вызывается метод экземпляра `to_s`.

Обычно не рекомендуется создавать (а тем более удалять!) записи в базе данных в ответ на запрос `GET`. Поисковые роботы, обшаривающие Интернет, периодически – сотни раз в сутки – отправляют запросы `GET` на все URL, о которых знают. В любом реальном приложении изменять данные следует только в ответ на запросы типа `POST`, `PUT`, `PATCH` и `DELETE`. В этом примере мы создаем запись в базе в обработчике `GET`-запроса только простоты ради.

Сохраните изменения, выполните команду `bin/rails server` и зайдите на один из ранее посещавшихся URL – вы увидите странный ответ вида `#<Greeting: 0x007fead16c1df8>!`. Это совсем не то, что мы хотели бы увидеть, поэтому надо написать собственный метод `to_s` в классе `Greeting`, который вернет настоящее приветствие. Для этого откройте файл `app/models/greeting.rb` и введите такой код:

```
class Greeting < ActiveRecord::Base
    def to_s
        s = "Hello"
        s << " from #{from}" if from
        s << ", #{to}"
    end
end
```

В этом методе повторно реализована уже знакомая нам логика, только теперь она является частью модели, а не контроллера. Поскольку за HTML-экранирование строки отвечает контроллер, нам здесь ничего экранировать не надо. Изменив файл `greeting.rb`, снова выполните команду `bin/rails server` – появится давно знакомое нам приветствие.

Научившись сохранять все когда-либо показанные приветствия, создадим новый контроллер, который будет отображать все ранее отправленные приветствия.

Отметим, что эти данные не пропадут, даже если остановить и снова запустить сервер Rails, потому что они сохранены в базе данных Sqlite3.

Чтобы создать контроллер для отображения старых приветствий, выполните команду `bin/rails generate controller greetings index`. Она добавит строку `get 'greetings/index'` в файл маршрутов и создаст файл `app/controllers/greetings_controller.rb`. Маршрут `get 'greetings/index'` – краткая запись маршрута `get 'greetings' => 'greetings#index'`. Затем откройте файл контроллера и введите код формирования ответа, содержащего список всех приветствий:

```
class GreetingsController < ApplicationController
   def index
      greetings = Greeting.all
      body = "<h1>Greetings</h1>\n<ol>\n"
      greetings.each do |greeting|
         body << "<li>" << CGI.h(greeting.to_s) << "</li>\n"
      end
      body << "</ol>"
      render inline: body
   end
end
```

В этом действии метод `all` загружает все строки из таблицы `greetings`, преобразует каждую строку в объект класса `Greeting` и возвращает массив таких объектов. С помощью метода `each` мы обходим массив и включаем каждое приветствие в ответ. Сохранив этот файл, запустите сервер командой `bin/rails server` и перейдите в браузере по адресу `http://localhost:3000/greetings`.

В зависимости от того, сколько экспериментов вы провели после создания модели Greeting, список может оказаться коротким или длинным. В любом случае не поленитесь отправить новое приветствие, а затем снова выведите список и убедитесь, что приветствие добавлено.

На этом заканчивается наше краткое знакомство с библиотекой ActiveRecord. Эта тема настолько сложна и обширна, что сколько-нибудь подробно осветить ее у нас не хватит места, ей можно посвятить целые книги (и они уже написаны). Дополнительные сведения об ActiveRecord ищите в руководствах на сайте `guides.rubyonrails.org` и в книге «The Rails 4 Way».

19.3.2. Хранилища данных

ActiveRecord – лишь одно из средств объектно-реляционного отображения (ORM), написанных на Ruby. Любой ORM представляет собой библиотеку, которая превращает хранимые данные в объекты Ruby, позволяет изменять эти объекты, а затем отображает их обратно на хранимые данные. Существуют ORM для многих хранилищ данных, не основанных на SQL, в том числе: Mongoid для MongoDB, Couchrest для CouchDB, redis-objects для Redis.

В большинстве случаев для веб-приложений лучше использовать популярные базы данных SQL с открытым исходным кодом, например PostgreSQL и MySQL. Поэтому ради экономии места мы не станем приводить примеры работы с NoSQL-

хранилищами. Однако горячо рекомендуем заняться самостоятельными исследованиями возможностей, предоставляемых другими хранилищами.

19.4. Генерация HTML

Возможно, вы пришли к выводу, особенно глядя на примеры выше, что создание HTML-ответов путем конкатенации строк выглядит не слишком элегантно. (Если вы не знаете, как работать с HTML, то, наверное, стоит отложить чтение этого раздела и сначала познакомиться с основами HTML. Соответствующее справочное руководство на сайте Mozilla Developer Network, находящееся по адресу `developer.mozilla.org/docs/Web/HTML`, – вполне подходящая отправная точка.)

Чем длиннее HTML-код ответа, тем труднее формировать его, пользуясь только строками. Поэтому в большинстве веб-каркасов (в том числе, в Rails) для этой цели применяются *шаблоны*.

Шаблон – это файл, написанный на специальном языке, который по запросу преобразуется в HTML-код. Часто шаблоны содержат Ruby-код, который вставляет в HTML-разметку значения переменных. Шаблоны могут быть вложенными – тогда результат отрисовки одного шаблона становится частью другого.

Обычно при использовании шаблонов создается один или несколько *макетов* (layout), содержащих HTML-код верхнего и нижнего колонтитула сайта. Макет будет повторяться на каждой странице, а в него вставляется шаблон, отрисованный конкретным контроллером. Наконец, небольшие фрагменты (например, повторяющиеся в цикле) можно выделить в *подшаблоны* (partial). В каждом шаблоне используется только один язык шаблонов, но разные макеты, шаблоны и подшаблоны могут быть написаны на разных языках.

Самым распространенным языком шаблонов является ERB. Он используется в Rails по умолчанию и является частью стандартной библиотеки Ruby. Выглядит он очень похоже на HTML (или, точнее сказать, на PHP) и предназначен для того, чтобы максимально упростить вставку строк Ruby в HTML-разметку.

19.4.1. ERB

Работать с ERB не просто, а очень просто. Вот пример оформления решения математической задачи:

```
require 'erb'
puts ERB.new("два плюс два равно <%= 2 + 2 %>").result
# Выводится: два плюс два равно 4
```

Код на языке ERB состоит из текста и *erb-тегов* вида `<%= code %>` или `<% code %>`. Если присутствует знак равенства, то значение выражения включается в текст. Если знака равенства нет, то выражение вычисляется, но в текст ничего не вставляется. Теги без знака равенства используются, главным образом, вместе с предложениями `if`, `each` и соответствующими `end`.

Использовать ERB в Rails тоже очень просто. Он даже подразумевается по умолчанию. Если действие контроллера ничего не отрисовывает самостоятель-

но, то Rails ищет шаблон с таким же именем, как у действия, в каталоге с таким же именем, как у контроллера, и возвращает результат отрисовки этого шаблона.

Для демонстрации перепишем наш первый пример с использованием шаблона, а не контроллера. Прежде всего, нужно удалить все содержимое метода index в классе HelloController, оставив только пустое определение:

```
def index
end
```

Затем напишем шаблон, которым должен воспользоваться Rails. Для этого создайте файл app/views/hello/index.html.erb и поместите в него ставшее уже привычным сообщение:

```
<h1>Hello from Ruby!</h1>
```

Сохранив файл, можете запустить приложение командой bin/rails server и перейти по адресу http://localhost:3000 – вы увидите все то же сообщение. На консоли будет написано, что Rails отрисовал шаблон hello/index:

```
Rendered hello/index.html.erb within layouts/application (0.7ms)
```

Теперь преобразуем в шаблон список приветствий, созданный в предыдущем разделе. Перепишите метод index в файле app/controllers/greetings_controller.rb следующим образом:

```
def index
   @greetings = Greeting.all
end
```

В Rails переменные экземпляра, устанавливаемые в методах контроллера, доступны и в шаблоне при его отрисовке. Мы воспользуемся этим фактом и изменим сгенерированный ранее шаблон app/views/greetings/index.html.erb. Введите в него такой ERB-код:

```
<h1>Greetings</h1>
<ol>
<% @greetings.each do |greeting| %>
   <li><%= h(greeting.to_s) %></li>
<% end %>
</ol>
```

Этот шаблон почти совпадает с кодом, написанным ранее для метода index. Основное отличие состоит в том, что HTML-код пишется непосредственно, без участия строковых переменных. ERB берет на себя заботу о замене ERB-тегов результатами исполнения находящегося внутри них кода на Ruby.

Для использования в шаблонах предназначены некоторые вспомогательные методы, например метод h, который делает то же самое, что рассмотренный выше метод CGI.h. Отметим, что даже внутри шаблона текст приветствия нужно подвергать HTML-экранированию, потому что он поступил от пользователя.

Изменив контроллер и шаблон, перезапустите сервер Rails и перейдите в браузере по адресу http://localhost:3000/greetings. Вы увидите тот же самый список приветствий, что и раньше, но метод контроллера теперь свелся к одной строчке.

Теперь в нашем примере присутствуют все три части парадигмы MVC (модель-представление-контроллер), поддерживаемой Rails. Сохранение данных реализуется путем взаимодействия с классом модели, логика обработки запроса — контроллером, который передает данные представлению, отвечающему за их отрисовку в виде HTML. Отрисованная HTML-разметка возвращается в составе тела HTTP-ответа, после чего обработка запроса завершается.

Инструментарий, применяемый для работы с шаблонами в Rails, гораздо сложнее, чем мы в состоянии описать на страницах этой книги. Нам пора переходить к Haml, еще одной популярной шаблонной системе, а вы обязательно изучите разнообразные возможности отрисовки с помощью макетов, шаблонов и подшаблонов, предоставляемых Rails. Дополнительные сведения см. в руководствах по шаблонам на сайте `guides.rubyonrails.org` и, конечно, в книге «The Rails 4 Way».

19.4.2. Haml

Библиотека Haml – еще один язык шаблонов, сильно отличающийся от ERB. Вместо включения тегов в HTML-разметку в Haml HTML-теги объявляются преимущественно с помощью пунктуации. Благодаря синтаксически значимым отступам удается отказаться от закрывающих тегов, угловые скобки тоже не нужны.

Чтобы можно было воспользоваться языком Haml в нашем примере Rails-приложения, добавьте в `Gemfile` строку `gem "haml"`. Затем выполните команду `bundle install`, которая добавит Haml в состав gem-пакетов, загружаемых при запуске приложения Rails. Дополнительные сведения о Bundler см. на сайте `bundler.io`.

Для иллюстрации различий между Haml и ERB перепишем шаблон списка приветствий на Haml. Прежде всего нужно переименовать файл шаблона `index.html.erb` в каталоге `app/views/greetings` в `index.html.haml`. Первое расширение показывает, файл какого типа будет сгенерирован, второе – какая шаблонная система должна использоваться для генерации.

Откройте файл `app/view/greetings/index.html.haml` и замените его содержимое кодом на Haml:

```
%h1 Greetings
%ol
  - @greetings.each do |greeting|
    %li= h(greeting.to_s)
```

Как видите, ни закрывающие HTML-теги, ни строки `end` не нужны. Знак `%` в начале строки означает, что это HTML-тег, а все следующее за ним – содержимое этого тега. Знак `=` означает, что строка содержит Ruby-код и внутрь тега должны быть включены результаты исполнения этого кода, а не дословное содержимое строки. Два знака равенства `==` после тега означают, что эта строчка содержит строку Ruby, допускающую обычную интерполяцию посредством конструкции `#{}`.

Многие пишущие на Ruby веб-разработчики находят Haml эстетически более совершенным и считают, что писать на нем гораздо быстрее, чем на ERB, благодаря меньшему количеству повторений. Но если для написания большого числа

тегов Haml, безусловно, удобен, то этого никак не скажешь о страницах, содержащих преимущественно текст. В таких случаях использовать ERB обычно проще.

Haml предоставляет развитый API с многочисленными возможностями и является привлекательной альтернативой ERB. Дополнительные сведения можно найти на сайте `haml.info`.

19.4.3. Другие шаблонные системы

ERB и Haml, безусловно, самые популярные шаблонные системы в Ruby, но отнюдь не единственные. В других системах приняты другие подходы к генерации HTML. Slim стремится по максимуму сократить объем ввода. Написанный на Slim шаблон насчитывает даже меньше символов, чем написанный на Haml.

Движок Liquid, разработанный для генератора Интернет-магазинов Shopify, допускает только ограниченный набор синтаксических конструкций, например предложения `if` и циклы `each`. Это делает безопаснее шаблоны, разрабатываемые пользователями, потому что чем меньше кода, тем меньше шансов стать жертвой какого-нибудь эксплойта.

Библиотека Mustache идет еще дальше и вообще запрещает использовать логику в шаблонах. Код должен быть выполнен до отрисовки – в объекте, который готовит данные для шаблона.

Если ни ERB, ни Haml вам не по душе, оглядитесь вокруг! Есть все шансы, что найдется шаблонная система, отвечающая вашим потребностям или вкусам. Ну а если нет, то хочу заметить, что написание движка шаблонов – освященная временем традиция у рубистов, так что вперед и с песней – ваяйте свой собственный.

19.5. Конвейер активов

Если вы проводите много времени, бродя по Интернету, то, наверное, заметили две вещи, которые встречаются практически на каждом современном сайте, но в нашем приложении пока отсутствуют: каскадные таблицы стилей (CSS) для визуальной стилизации и JavaScript для программирования взаимодействия с пользователем.

Rails предоставляет унифицированную систему управления CSS и JavaScript. Во время разработки очень полезно иметь доступ к каждому CSS и JavaScript-файлу – для экспериментов, отладки и т. п. Но на «боевом» сайте гораздо эффективнее объединить все таблицы стилей в один CSS-файл, а весь клиентский код – в один JavaScript-файл. Это уменьшает число HTTP-запросов и ускоряет загрузку страниц.

Механизм автоматизации этого процесса включен в Rails под названием *конвейера активов* (asset pipeline). Любой файл в каталоге `app/assets` обрабатывается конвейером активов; на этапе обработки файлы сохраняют автономность, а на этапе эксплуатации объединяются. Из объединенного файла удаляются комментарии, пробелы и вообще все, что можно удалить. Задача – сократить размер до минимума.

Подобно отрисовке шаблонов, активы, написанные на одном языке, можно транслировать на другой. В первом нашем примере исходным будет язык SCSS, а конечным – CSS.

19.5.1. CSS и Sass

Sass, который авторы иногда называют «Syntactically Awesome Style Sheets» (синтаксически прелестные таблицы стилей), – это процессор языка SCSS, являющегося надмножеством CSS. (Если вы еще незнакомы с CSS, то этот раздел вряд ли принесет вам много пользы. На сайте Mozilla Developer Network имеются справочные и учебные материалы по CSS по адресу `https://developer.mozilla.org/docs/Web/CSS`.)

Поскольку SCSS – надмножество CSS, то любой CSS-код одновременно является корректным кодом на SCSS, но SCSS предлагает и многое сверх того. Важнейшие дополнения – переменные, вложенность, подтаблицы, наследование и математические операторы.

Gem-пакет `sass-rails` был автоматически включен в наше приложение `hello`, так что мы можем прямо сейчас изменить визуальное представление примера, добавив что-нибудь в существующий SCSS-файл. Откройте файл `app/assets/stylesheets/greetings.css.scss`. В конец файла добавьте строки, которые изменят внешний вид списка приветствий:

```
body {
   font-family: "Avenir", "Helvetia Neue", Arial;
}

h1 {
   text-shadow: 2px 2px 5px rgba(150, 150, 150, 1);
}

ol {
   margin-left: 15px;
   li {
      margin-bottom: 10px;
   }
}
```

Сохранив файл, проверьте, что сервер Rails работает, и перезагрузите страницу по адресу `http://localhost:3000`. Список приветствий будет выглядеть по-другому. Под заголовком «Greetings» появится светлая тень, а каждое приветствие будет отделено от края страницы и от других приветствий промежутком.

Код на SCSS, как и на CSS, состоит из *селекторов* и *правил*. Селекторы определяют, к каким HTML-тегам применяются правила, а правила описывают визуальные характеристики. В примере SCSS выше мы изменили шрифт, используемый внутри тега `body` (а это вся видимая часть страницы). Затем мы задали для тега `h1` серую тень, которая выделяет и «приподнимает» такой текст на странице. И в последнем наборе правил мы задаем левое поле шириной 15 пикселей для каждого нумерованного списка (тег `ol`), а для каждого элемента такого списка (тег `li`) – нижнее поле высотой 10 пикселей.

Изменения простые, но ясно показывают всю процедуру работы с таблицами стилей. Файлы, находящиеся в каталоге `app/assets/stylesheets`, транслируют-

ся в CSS, если необходимо (на основе того же соглашения о расширениях файлов, что и для шаблонов). Затем результирующие CSS-файлы отправляются клиентскому браузеру, который с их помощью изменяет визуальное представление HTML-разметки, пришедшей в теле ответа.

Крис Эпштейн (Chris Eppstein), один из авторов Sass, разработал также Compass – библиотеку предопределенных правил, которые можно импортировать в SCSS-файлы. В Compass включен большой набор общеупотребительных правил, относящихся к межбраузерной совместимости, управлению макетом и типографике, а также правила, которые должны войти в стандарт CSS3, но пока поддерживаются не всеми браузерами. Дополнительные сведения о Compass см. на сайте `compass-style.org`.

Даже без Compass процессор Sass – полезный и эффективный инструмент, который пригодится любому разработчику, который занимается написанием и сопровождением CSS-стилей. Для получения дополнительных сведений о Sass начните с посещения сайта `sass-lang.com`. Отметим также, что существуют и другие препроцессоры CSS, помимо Sass, в частности Less и Stylus. Если Sass вам почему-то не нравится, быть может, альтернативы покажутся более привлекательными.

Разобравшись с тем, как в Rails обрабатываются таблицы стилей, перейдем к JavaScript.

19.5.2. JavaScript и CoffeeScript

С годами веб-приложения стали гораздо более сложными, функционально насыщенными и интерактивными. Из хорошо известных примеров так называемых «обогащенных клиентов» можно назвать Gmail и Google Docs, Microsoft Office 365 и iWork для iCloud от Apple. Во всех них активно используется JavaScript-код для создания сайта, который, работая в браузере, имитирует внешний вид и функциональные возможности сложного персонального приложения для Windows или Mac OS X.

Для некоторых сайтов наличие такой развитой интерактивности – необходимое требование. Реагируя на эту тенденцию, появилось множество JavaScript-каркасов, помогающих создавать приложения, характеризуемые большей интерактивностью. Одни каркасы ориентированы больше на богатство возможностей, другие – на простоту работы. В качестве примеров назовем Ember, Angular, React, Backbone и даже Batman.js, но отметим, что вообще-то их десятки и, похоже, каждую неделю появляется новый.

Конвейер активов позволяет включить в приложение Rails одну или несколько JavaScript-библиотек. Для некоторых (например, Ember и Angular) приложение Rails играет роль JSON API, которое поставляет данные, необходимые JavaScript-приложению, работающему внутри браузера. Для других (например, React и Backbone) приложение Rails может отдавать данные в формате HTML, JSON или том и другом в зависимости от того, как написан код на Ruby и JavaScript. Более подробное обсуждение различных JSON API приведено в следующем разделе.

Так же, как для SCSS имеется транслятор на CSS, так и для JavaScript существует популярный язык, транслируемый на понятный браузерам JavaScript. Язык CoffeeScript добавляет многое из того, чего, по мнению авторов, недостает JavaScript, в том числе итераторы, исключение почти всех скобок, синтаксис создания функций и другое.

Для демонстрации использования CoffeeScript в приложении Rails добавим код включения нового приветствия в список при каждом щелчке по заголовку «Greetings». Откройте файл app/assets/javascripts/greetings.js.coffee и добавьте в конец такие строки:

```
$('body').ready ->
   $('h1').click () ->
      $.ajax "/hello/Hilda?from=Rails",
         success: (data) ->
            item = $.parseHTML('<li>' + $(data).text() + '</li>')
            $('ol').append item
```

JavaScript-код, получившийся в результате трансляции этого кода на CoffeeScript, можно просмотреть в браузере, перейдя по адресу http://localhost:3000/assets/greetings.js, ниже он воспроизводится:

```
(function() {
   $('body').ready(function() {
      return $('h1').click(function(e) {
         return $.ajax("/hello/Hilda?from=Rails", {
            success: function(data) {
               var item;
               item = $.parseHTML('<li>' + $(data).text() + '</li>');
               return $('ol').append(item);
            }
         });
      });
   });
}).call(this);
```

Здесь мы пользуемся JavaScript-библиотекой jQuery (доступ к которой открывает переменная $), чтобы найти тег body, а затем предоставляем функцию, которая выполняется один раз, когда браузер закончит отрисовку и тело будет «готово». В этой функции мы находим тег h1 и передаем функцию, которая вызывается при щелчке по этому тегу. В ней предоставляемый jQuery метод ajax запрашивает конкретное приветствие, обращаясь к URL /hello, который мы реализовали раньше. Если обработка запроса завершилась успешно, то создается новый тег li, содержащий текст из тела ответа, и этот тег добавляется в конец нумерованного списка ol. Явно этого не видно из кода, но если Ajax-запрос завершается неудачно, то ничего не происходит. Для обработки ошибок следовало бы предоставить функцию с ключом failure.

Это очень простой пример совместного использования CoffeeScript и jQuery, но из него легко составить представление о том, чего можно достичь в приложе-

нии Rails с помощью конвейера активов. Есть много веб-приложений, которые начинали с таких же скромных шагов, а затем разрослись в гигантские монолитные приложения Rails, приносящие тысячи и даже миллионы долларов дохода.

Как видно даже из этого тривиального примера, CoffeeScript-код короче, проще и чище эквивалентного JavaScript-кода. Это различие проявляется тем нагляднее, чем больше объем кода – JavaScript оказывается заметно длиннее. Но у CoffeeScript есть и недостатки. И самый существенный – тот факт, что CoffeeScript добавляет к JavaScript собственный цикл for и систему наследования на основе классов.

Вы спросите, почему это недостаток? Потому что цикл for и наследование на основе классов запланированы к включению в следующую версию языка, известную под названием ECMAScript 6. ES6 (общепринятое сокращение) стремится устранить многие недостатки, из-за которых и был создан CoffeeScript, но способом, совместимым с существующим языком JavaScript, а не радикально отличающимся от него.

Как часто бывает в подобных ситуациях, существуют компромиссы, склоняющиеся как к одному, так и к другому подходу. Лично я перестал пользоваться CoffeeScript, когда понял, что он не обеспечит совместимость со следующей версией JavaScript. Для других скорость разработки (а, быть может, отсутствие лишних скобок) имеют большую ценность, для них CoffeeScript – несомненное благо.

Желающие узнать больше о CoffeeScript могут зайти на сайт `coffeescript.org`. Что же касается изучения JavaScript, то я рекомендую справочные руководства и учебные пособия на сайте Mozilla Developer по адресу `developer.mozilla.org/docs/Web/JavaScript`.

19.6. Предоставление веб-служб по протоколу HTTP

Наконец, настало время поговорить о том, как программы в Интернете могут взаимодействовать друг с другом, а не с человеком. Аббревиатура API расшифровывается как *application programming interface* (программный интерфейс приложений), но обычно под этим понимают интерфейс, посредством которого программы могут общаться между собой.

В недобрые старые дни сетевого программирования, говоря «сетевой API», имели в виду написание клиентских и серверных программ с использованием сокетов – типа рассмотренных в главе 18. Но в последнее время смысл термина изменился, и теперь все чаще подразумевают отправку HTTP-запросов из программы некоторому HTTP-серверу, предоставляющему документированный набор URL-адресов.

Входящий в стандартную библиотеку Ruby класс `Net::HTTP` представляет собой полнофункциональный клиент для взаимодействия с такими HTTP API. У него есть ряд неудобств и причуд, поэтому существуют другие gem-пакеты, предлагающие более удобный API, в том числе http, excon и typhoeus.

Но пользоваться HTTP API сравнительно просто и вообще без всяких сторонних библиотек, особенно если прибегнуть к помощи стандартной библиотеки `open-uri`:

```
require 'open-uri'
response = open("http://mywebstore.com/products?page=5")
```

После этого вызов метода `response.read` вернет тело HTTP-ответа в виде строки. К заголовкам можно обратиться с помощью метода `[]`, например, `response["Content-Type"]` возвращает значение заголовка Content-Type в виде строки.

Библиотеки для работы с HTTP позволят вам без труда взаимодействовать с другими сайтами из своей Ruby-программы.

19.6.1. Использование JSON в API

В 1990-х и начале 2000-х годов для кодирования данных, передаваемых в запросах и ответах, определенных в API, чаще всего использовался формат XML (eXtensible Markup Language). Но из-за своей сложности и громоздкости XML вышел из моды. Ему на смену пришел лаконичный, минималистский стандарт сериализации JSON.

Аббревиатура JSON означает *JavaScript Object Notation* (объектная нотация JavaScript), но этот формат стал стандартом ISO и широко используется для передачи данных между программами, написанными практически на любом языке. При разработке любого веб-приложения, скорее всего, придется взаимодействовать с HTTP API, возвращающем ответы в формате JSON.

В стандартный дистрибутив Ruby входит библиотека `JSON`, которая умеет преобразовывать данные в формате JSON в хэши Ruby и, наоборот, представлять хэши Ruby в формате JSON. Существуют и другие библиотеки, предоставляющие дополнительные возможности. Например, Yajil умеет разбирать поток JSON-данных, а Oj осуществляет разбор быстрее, потому что написана на C, а не на Ruby.

Если бы ответ, полученный в предыдущем примере, содержал массив товаров, представленный в формате JSON, то мы могли бы преобразовать его в хэш Ruby и получить доступ к массиву следующим образом:

```
require 'json'
products = JSON.parse(response.body)["products"]
# products теперь является массивом хэшей, полученным
# в результате разбора JSON-строки, которую вернул API
```

Этот хэш можно подать на вход любой задаче, и в результате программа будет работать с данными, полученными от другого сервера. Реальный пример взаимодействия с применением JSON API см. в разделе 15.2.

19.6.2. REST API и его вариации

Архитектурный стиль REST (REpresentational State Transfer – передача представимых состояний) в 2000 году определил Рой Филдинг (Roy Fielding) как стиль проектирования программного обеспечения, обеспечивающий взаимодействие с ним предсказуемыми способами. Идеи, положенные в основу REST, оказали значительное влияние на проектирование протокола HTTP версии 1.1, который был наделен возможностями, практически не используемыми веб-разработчиками и браузерами. Лишь много лет спустя философия REST была частично воспринята Rails, где реализованы условные URL и стандартные спосо-

бы выполнения действий создания, чтения, обновления и удаления (Create, Read, Update, Delete – CRUD).

Rails отображает операции CRUD на различные «глаголы» HTTP, или действия, допускаемые стандартом HTTP. Созданию соответствует глагол `POST`, чтению – `GET`, обновлению – `PATCH`, а удалению – `DELETE`.

Гипотетический REST API для взаимодействия с каталогом книг мог бы выглядеть следующим образом:

```
GET /books          # Вернуть список всех книг
POST /books         # Создать новую книгу
GET /books/1        # Вернуть информацию о книге 1
PATCH /books/1      # Обновить информацию о книге 1
DELETE /books/1     # Удалить информацию о книге 1
```

API, спроектированные с соблюдением этих принципов, настолько хорошо стандартизованы, что существуют написанные на Ruby библиотеки, для взаимодействия с ними, например: rest-client, httparty и faraday.

Хотя появление Rails привело к активному использованию глаголов HTTP в веб-приложениях, то, что называется словом «REST» в Rails, на самом деле проще и значительно уступает в функциональности основополагающим идеям Филдинга. Те разработчики, которые стремятся воплотить в жизнь изначальную идею REST, называют ее «гипермедиа», чтобы отличить от урезанной версии, встроенной в Rails и другие каркасы.

Я всячески рекомендую разработчикам веб-служб (да и всем интересующимся) побольше почитать о REST – как в изначальной форме, так и в той, что принята в Rails. Если вы хотите узнать о дальнейшем развитии идей REST в том виде, как их видел Филдинг, ищите в Интернете по ключевому слову «hypermedia».

19.7. Генерация статических сайтов

Хотя популярность Ruby принесли динамические веб-приложения, этот язык нередко применяется и для генерации статических сайтов. Сайт называется статическим, если нет никакого веб-приложения, обрабатывающего входящие запросы, а веб-сервер просто находит на диске файл, соответствующий указанному в запросе пути, и возвращает его содержимое в теле ответа.

В принципе, это может быть даже веб-сервер, написанный на Ruby, например Puma или Unicorn. Но обычно статические сайты обслуживаются каким-то широко известным сервером, оптимизированным для раздачи статических файлов, скажем, Apache или Nginx. Строго говоря, «статический сайт» может содержать интерактивные или динамические элементы, реализованные на JavaScript. Важно лишь, что в состав сайта входят только неизменные HTML, CSS и JavaScript-файлы.

В этом разделе мы создадим статический сайт, аналогичный примеру «Hello» для Rails. Поскольку на статическом сайте в ответ на запрос не запускается никакой код, то мы ограничимся несколькими конкретными приветствиями, а не станем генерировать новое приветствие для каждого запроса.

19.7.1. Middleman

Генератор статических сайтов Middleman организует окружение, очень похожее на имеющееся в приложении Rails, только результатом его работы является набор файлов, которые может отдавать любой HTTP-сервер. Чтобы воспользоваться Middleman, сначала установите gem-пакет командой `gem install middleman`. В состав пакета входит утилита `middleman`, предоставляющая несколько команд для управления созданием статических сайтов. Для получения полного списка команд введите `middleman help`.

Установив Middleman, создадим новый проект командой `middleman init hello_world`. В результате будет создан каталог `hello_world`, а в нем файлы, составляющий сайт (почти пустой). Чтобы посмотреть, как выглядит сайт, перейдите в этот каталог командой `cd`, выполните команду `bundle exec middleman server`, которая запустит сервер предварительного просмотра, а затем перейдите в браузере по адресу `http://localhost:4567`. Как и сервер Rails, сервер Middleman позволяет быстро оценивать результаты внесенных в сайт изменений на этапе разработки.

Закончив работу над сайтом, мы можем остановить сервер Middleman и выполнить команду `middleman build`. Она создаст окончательный статический сайт и поместит все файлы в подкаталог `public` внутри каталога проекта Middleman. Сгенерированный сайт можно развернуть на рабочем сервере, просто скопировав файлы из каталога `public` на нужный HTTP-сервер.

Поскольку мы хотим, чтобы наш сайт показывал приветствие, изменим файл `source/indexhtml.erb`, поместив в него следующий текст:

```
title: Hello from Middleman!
<h1>Hellc from Middleman!</h1>
```

Сохраните файл и перезагрузите страницу `http://localhost:4567`, чтобы посмотреть, что получилось. Поскольку Middleman создает статические сайты, в URL мы можем указывать только пути, соответствующие физическим файлам. Чтобы добавить приветствие, размещенное по адресу `/world`, мы должны создать файл. Сначала создайте новый каталог командой `mkdir source/world`. Затем создайте в нем файл `source/world/index.html.erb` и поместите в него сообщение:

```
<h1>Hello, World!</h1>
```

Теперь, перейдя по адресу `http://localhost:4567/world`, вы увидите это приветствие. Наверное, у вас возник вопрос, зачем мы создали каталог `world` и поместили в него шаблон `index.html.erb`. Причина – в желании иметь красивые URL-адреса.

Серверы, раздающие статические файлы, ищут файл по имени, включающему и расширение. Если в запросе указан путь, не содержащий расширение, например `/world`, то сервер будет просматривать соответствующий каталог. Если он найдет в нем *индексный файл*, то автоматически вернет его. Именно поэтому, поместив индексный файл в каталог `world`, мы смогли увидеть приветствие, перейдя по адресу `/world`. Если бы мы просто создали файл `world.html.erb`, то смогли бы увидеть приветствие, лишь указав путь `/world.html`.

Статические сайты, сгенерированные Middleman, предоставляют, в основном, те же утилиты и вспомогательные средства, что и приложения Rails, в том числе

вспомогательные функции в шаблонах, полный конвейер активов, включая интеграцию с Sass, Compass и CoffeeScript, и многое другое. Полный перечень функций см. в документации по Middleman на сайте middlemanapp.com.

19.7.2. Другие генераторы статических сайтов

Jekyll – еще один популярный генератор статических сайтов, написанный на Ruby. В нем для написания шаблонов используется язык Liquid, а содержимое страниц записывается на языке Markdown. Автором Jekyll является один из сооснователей GitHub, и на GitHub он используется для генерации статических сайтов размещенных там проектов. Поскольку GitHub предлагает хостинг для проектов с открытым исходным кодом, на нем размещено много сайтов с документацией новых gem-пакетов – и все они сгенерированы Jekyll. Дополнительные сведения о Jekyll можно найти на сайте jekyll.com. О размещении статических сайтов на GitHub читайте по адресу pages.github.com.

Octopress – это инструменты и набор расширений Jekyll, предназначенный для быстрого создания блога, публикуемого в виде статического сайта. Несмотря на то, что Octopress надстроен над Jekyll, он содержит все необходимое для того, чтобы сразу же начать написание статей для блога. В комплект входит шаблон блога, таблицы стилей, поддержка комментариев в стиле Disqus, синтаксическая подсветка фрагментов кода, включенных в статьи, и великое множество подключаемых модулей, наделяющих Octopress дополнительными возможностями. Дополнительные сведения об Octopress см. на сайте octopress.org.

Генератор статических сайтов Nanoc не столь популярен, как Middleman или Jekyll, зато является самым первым написанным на Ruby инструментом такого рода. Первая версия вышла в 2007 году, но и сейчас, в 2014, он активно сопровождается. В нем принят свой подход к генерации сайтов, выраженный в анонсе: «годится для построения всего – от небольшого личного блога до сайта крупной корпорации». Обратите внимание на Nanoc, если ищете альтернативы или просто хотите узнать о проекте с открытым исходным кодом, который процветает уже много лет. Сайт Nanoc находится по адресу nanoc.ws.

19.8. Заключение

В этой главе мы рассмотрели низкоуровневые детали программирования HTTP и реализовали свой минимальный HTTP-сервер. Мы узнали о каркасах и инструментах, упрощающих создание веб-приложений:Puma, Rails, Sinatra, ActiveRecord, ERB, Haml, Sass и Less. Наконец, мы поговорили о способах взаимодействия между веб-приложениями и службами с помощью HTTP API и познакомились с генераторами статических сайтов Middleman и Jekyll.

Теперь обратимся к иному гораздо более простому и абстрактному способу сетевого программирования. Следующая глава посвящена распределенной обработке с применением Ruby.

Глава 20. Распределенный Ruby

Меньше – это больше.

– Робер Браунинг
«Андреа дель Сарто»

За последние 25 лет было изобретено немало технологий, поддерживающих распределенные вычисления: различные варианты RPC, а также COM, CORBA, DCE и Java RMI.

Одни проще, другие сложнее, но в принципе все делают одно и то же – предоставляют относительно прозрачный способ связи между находящимися в сети объектами так, чтобы с удаленными объектами можно было работать, как с локальными.

Зачем это вообще может понадобиться? Причин много. Например, чтобы распределить некоторую вычислительную задачу между многими процессорами. Примером может служить программа SETI@home, которая использует ваш ПК для обработки небольших объемов данных в поисках внеземного разума (кстати, эта программа не является проектом института SETI). Другой пример – привлечение широких масс к взлому шифра RSA129 (эта попытка увенчалась успехом несколько лет назад). Существует очень много задач, которые можно разбить на небольшие части, пригодные для распределенного решения.

Или допустим, вы хотите предоставить интерфейс к некоей службе, не раскрывая исходных текстов. Часто это делается с помощью веб-приложений, но из-за отсутствия состояния в протоколе HTTP это не всегда удобно (есть и другие недостатки). Механизм распределенного программирования позволяет решать подобные задачи более естественно.

В мире Ruby ответом на этот вызов стала программа `drb`, написанная Масатоси Секи (Masatoshi Seki) (иначе ее называют `DRb`). Существуют и другие способы распределенной обработки на Ruby, но `drb`, пожалуй, самый легкий. В ней нет сложных служб разрешения имен, как в CORBA. Это всего лишь простая и удобная библиотека, предоставляющая все необходимую функциональность. В этой главе мы рассмотрим основы работы как с ней самой, так и с надстроенной над ней системой Rinda.

20.1. Обзор: библиотека drb

Приложение `drb` состоит из двух основных частей: серверной и клиентской. Грубую границу между ними можно провести следующим образом:

Сервер:

- запускает TCPServer и начинает прослушивать порт;
- привязывает объект к экземпляру сервера drb;
- принимает запросы на подключение от клиентов и отвечает на их сообщения;
- дополнительно может предоставлять контроль доступа (безопасность).

Клиент:

- устанавливает соединение с сервером;
- привязывает локальный объект к удаленному экземпляру сервера;
- посылает сообщения серверу и получает ответы.

Метод класса start_service отвечает за запуск TCP-сервера, прослушивающего указанный порт. Он принимает два параметра: URI (универсальный идентификатор ресурса), задающий порт (если он равен nil, то порт выбирается динамически), и объект, к которому мы хотим привязаться. Этот объект будет доступен удаленному клиенту, который сможет вызывать его методы, как если бы объект был локальным.

```
require "drb"

myobj = MyServer.new
DRb.start_service("druby://:1234", myobj)   # Порт 1234

# ...
```

Если порт выбирается динамически, то для получения полного URI, включающего и номер порта, можно воспользоваться методом класса uri.

```
DRb.start_service(nil, myobj)
myURI = DRb.uri                  # "druby://hal9000:2001"
```

Поскольку drb – многопоточная программа, любое серверное приложение, должно выполнять join в потоке сервера (чтобы не дать приложению завершиться преждевременно и тем самым уничтожить выполняющийся поток).

```
# Предотвратить преждевременный выход
DRb.thread.join
```

На стороне клиента мы вызываем метод start_service без параметров и с помощью класса DRbObject создаем локальный объект, соответствующий удаленному. Обычно первым параметром методу DRbObject.new передается nil.

```
require "drb"

DRb.start_service
obj = DRbObject.new(nil, "druby://hal9000:2001")

# Сообщения, передаваемые obj, перенаправляются
# удаленному объекту на стороне сервера...
```

Следует подчеркнуть, что на стороне сервера привязка осуществляется к единственному объекту, который должен отвечать на все получаемые запросы. Если клиентов несколько, то объект должен быть безопасным относительно потоков, чтобы не оказаться в некорректном состоянии. (Для совсем простых или узкоспециализированных приложений это может быть и необязательно.)

Мы не можем вдаваться в технические детали. Но имейте в виду, что если клиент читает или изменяет внутреннее состояние удаленного объекта, то при наличии нескольких клиентов возможна интерференция. Во избежание таких неприятностей мы рекомендуем применять механизмы синхронизации, например, класс `Mutex`. (Подробнее о потоках и синхронизации см. главу 13.)

Скажем хотя бы несколько слов о безопасности. Ведь не всегда желательно, чтобы с вашим сервером мог соединяться кто угодно. Помешать им пытаться вы не можете, зато можете сделать такие попытки безуспешными.

В программе `drb` есть понятие списка контроля доступа (ACL). Это не что иное, как списки клиентов (или категорий клиентов), которым явно разрешен (или запрещен) доступ.

Приведем пример. Для создания нового списка ACL мы воспользуемся классом `ACL`, которому передадим один или два параметра. Второй (необязательный) параметр метода `ACL.new` служит для ответа на вопрос: «Мы запрещаем доступ всем клиентам, кроме некоторых, или, наоборот, разрешаем доступ всем клиентам, кроме некоторых?» По умолчанию принимается первый вариант, который обозначается константой `DENY_ALLOW`, равной 0. Второй режим обозначается `ALLOW_DENY`, равной 1.

Первый параметр `ACL.new` – обычный массив строк, которые идут парами. Первая строка в паре должна быть равна `"deny"` или `"allow"`, вторая описывает одного клиента или группу клиентов (по имени или по адресу):

```
require "drb/acl"
acl = ACL.new( %w[ deny all
                   allow 192.168.0.*
                   allow 210.251.121.214
                   allow localhost] )
```

Первая пара в этом примере, строго говоря, излишня, но проясняет смысл всей конструкции.

А как используются ACL? Метод `install_acl` приводит ACL в действие. Его необходимо вызывать перед обращением к `start_service`, иначе он не возымеет эффекта.

```
# Продолжение примера...

DRb.install_acl(acl)
DRb.start_service(nil, some_object)
# ...
```

Теперь, после запуска службы любой неавторизованный запрос на соединение приведет к исключению `RuntimeError`.

Это, конечно, не все, что можно сказать о библиотеке drb. Но для обзора вполне достаточно. В следующем разделе мы рассмотрим простой drb-сервер и drb-клиент, близкие к реальным программам. А затем поговорим о программах Rinda и Ring.

20.2. Пример: эмуляция биржевой ленты

В этом примере сервер публикует в сети биржевые котировки акций. К серверу может обратиться любой клиент, желающий узнать, сколько сейчас стоит его пакет.

Но мы добавили одну тонкость. Не желая следить за малейшими колебаниями цен, мы реализовали модуль Observer, который позволяет подписаться на информационный канал. Клиент следит за поступающими сведениями и предупреждает нас, когда изменение цены превысит заданный порог.

Сначала рассмотрим модуль DrbObservable. Это прямолинейная реализация паттерна Observer (Наблюдатель), описанного в замечательной книге Гамма, Хелм, Джонсон и Влиссидес «Паттерны проектирования». Еще этот паттерн называют «Издатель-Подписчик».

В листинге 20.1 наблюдатель определен как объект, отвечающий на вызов метода update. Сервер добавляет наблюдателей по их просьбе и посылает им уведомления, обращаясь к методу notify_observers.

Листинг 20.1. Модуль DrbObservable

```
module DRbObservable

  def add_observer(observer)
    @observer_peers ||= []
    unless observer.respond_to? :update
      raise NameError, "наблюдатель должен отвечать на вызов 'update'"
    end
    @observer_peers.push observer
  end

  def delete_observer(observer)
    @observer_peers.delete observer if defined? @observer_peers
  end

  def notify_observers(*arg)
    observer_peers.dup.each do |peer|
      peer.update(*arg) rescue delete_observer(i)
    end if observer_peers.any?
  end
end
```

Сервер (он же канал) в листинге 20.2 эмулирует биржевые котировки с помощью последовательности псевдослучайных чисел (простите мою иронию, но это очень точно соответствует характеру рынка). Символ, идентифицирующий ком-

панию, – всего лишь косметическое украшение, никакого реального смысла в этой программе он не имеет. При каждом изменении цены посылается уведомление всем наблюдателям.

Листинг 20.2. Канал биржевых котировок (drb-сервер)

```
require "drb"
require "drb_observer"

# Генерировать случайные котировки
class MockPrice

  MIN = 75
  RANGE = 50

  def initialize(symbol)
    @price = RANGE / 2
  end

  def price
    @price += (rand() - 0.5)*RANGE
    if @price < 0
      @price = -@price
    elsif @price >= RANGE
      @price = 2*RANGE - @price
    end
    MIN + @price
  end
end

class Ticker # Периодически получать котировку акций
  include DRbObservable

  def initialize(price_feed)
    @feed = price_feed
    Thread.new { run }
  end

  def run
    lastPrice = nil
    loop do
      price = @feed.price
      print "Текущая котировка: #{price}\n"
      if price != lastPrice
        lastPrice = price
        notify_observers(Time.now, price)
      end
      sleep 1
    end
```

```
  end
end

ticker = Ticker.new(MockPrice.new("MSFT"))

DRb.start_service('druby://localhost:9001', ticker)
puts 'Нажмите [return] для завершения.'
DRb.thread.join
```

Как и следовало ожидать, клиент (листинг 20.3) начинает с установления соединения с сервером. Он получает ссылку на объект показа котировок и устанавливает верхний и нижний пороги изменения цены. Затем клиент выводит сообщение пользователю всякий раз, как цена выйдет за пределы указанного диапазона.

Листинг 20.3. Наблюдатель биржевых котировок (drb-клиент)

```
require "drb"

class Warner
  include DRbUndumped

  def initialize(ticker, limit)
    @limit = limit
    ticker.add_observer(self) # любой объект Warner является наблюдателем
  end
end

class WarnLow < Warner
  def update(time, price)       # обратный вызов наблюдателя
    if price < @limit
      print "--- #{time.to_s}: Цена ниже #@limit: #{price}\n"
    end
  end
end

class WarnHigh < Warner
  def update(time, price)       # обратный вызов наблюдателя
    if price > @limit
      print "+++ #{time.to_s}: Цена выше #@limit: #{price}\n"
    end
  end
end

DRb.start_service
ticker = DRbObject.new(nil, "druby://localhost:9001")

WarnLow.new(ticker, 90)
WarnHigh.new(ticker, 110)

puts 'Нажмите [return] для завершения.'
gets
```

Модуль DRbUndumped (см. листинге 20.3) следует включать в любой объект, который не нужно подвергать маршалингу. Самого присутствия этого модуля в числе предков объекта достаточно, чтобы drb не пытался применять к нему маршалинг. Вот исходный текст этого модуля целиком:

```
module DRbUndumped
  def _dump(dummy)
    raise TypeError, "can't dump"
  end
end
```

Приложение из этого раздела достаточно содержательно, и в то же время в нем легко разобраться. Есть и другие подходы к решению подобных задач. Но показанный нами демонстрирует простоту и элегантность распределенного Ruby.

20.3. Rinda: пространство кортежей в Ruby

Термин *пространство кортежей* появился в 1985 году, а сама идея еще старше. *Кортежем* называется массив или вектор, состоящий из элементов данных (как строка в таблице базы данных). Пространство кортежей – это большое объектное пространство, наполненное кортежами, нечто вроде «информационного супа».

Пока реализация пространства кортежей кажется ничем не примечательной. Но все становится гораздо интереснее, стоит лишь осознать, что к нему могут обращаться многие клиенты, и доступ должен синхронизироваться. Короче говоря, это распределенная сущность; любой клиент может читать из пространства кортежей или писать в него, то есть его можно рассматривать как большое распределенное хранилище или даже способ коммуникации.

Первой реализацией пространства кортежей был проект Linda – исследование в области параллельного программирования, выполненное в Йельском университете в 1980-х годах. Реализация на языке Ruby (конечно, на основе библиотеки drb), естественно, называется *Rinda*.

Кортеж в Rinda может быть массивом или хэшем. На хэш налагается дополнительное ограничение: все ключи должны быть строками. Вот несколько примеров простых кортежей:

```
t1 = [:add, 5, 9]
t2 = [:name, :add_service, Adder.new, nil]
t3 = { 'type' => 'add', 'value_1' => 5, 'value_2' => 9 }
```

Количество элементов в кортеже не ограничено. Элемент кортежа может быть произвольным объектом; это работает, потому что drb умеет выполнять маршалинг и демаршалинг объектов Ruby. (Конечно, необходимо либо включить модуль DRbUndumped, либо сделать определения классов доступными серверу).

Пространство объектов создается методом new:

```
require 'rinda/tuplespace'

ts = Rinda::TupleSpace.new
# ...
```

Поэтому сервер выглядит так:

```
require 'rinda/tuplespace'

ts = Rinda::TupleSpace.new
DRb.start_service("druby://somehost:9000", ts)
DRb.thread.join   # Нажатие Control-C останавливает сервер
```

А клиент – так:

```
require 'rinda/tuplespace'

DRb.start_service
ts = DRbObject.new(nil, "druby://somehost:9000")
# ...
```

К пространству кортежей в Rinda применимы пять операций: read, read_all, write, take и notify. Операция чтения read позволяет получить один кортеж. Но способ идентификации кортежа не вполне очевиден: необходимо задать кортеж, соответствующий искомому, при этом nil соответствует любому значению.

```
t1 = ts.read [:sum,nil]          # может извлечь, например, [:sum, 14]
```

Обычно операция read блокирует выполнение программы (для синхронизации). Чтобы быстро проверить существование кортежа, можно выполнить неблокирующее чтение, задав нулевой таймаут:

```
t2 = ts.read [:result,nil],0   # возбуждает исключение, если кортеж не существует
```

Если мы точно знаем или предполагаем, что образцу будет соответствовать более одного кортежа, то можно воспользоваться методом read_all, который возвращает массив:

```
tuples = ts.read_all [:foo, nil, nil]
tuples.each do |t|
  # ...
end
```

У метода read_all нет второго параметра. Он всегда блокирует программу, если не найдено ни одного подходящего кортежа.

Операция take – это чтение, за которым следует неявное удаление. Иными словами, метод take удаляет кортеж из пространства кортежей и возвращает его вызывающей программе:

```
t = ts.take [:sum, nil]    # кортежа больше нет в пространстве кортежей
```

Может возникнуть вопрос, почему не существует явного способа удаления. Надо полагать, что этой цели служит метод take.

Метод write помещает кортеж в пространство кортежей. Второй параметр говорит, сколько секунд кортеж может существовать, прежде чем система сочтет, что срок его хранения истек. (По умолчанию его значение равно nil, то есть срок хранения не ограничен.)

```
ts.write [:add, 5, 9]          # Хранить "вечно"
ts.write [:foo, "Bar"], 10     # Хранить 10 секунд
```

Здесь уместно будет сказать несколько слов о синхронизации. Предположим, что два клиента пытаются примерно в одно и то же время забрать (take) один и тот же кортеж. Одному это удастся, а другой будет заблокирован. Если первый клиент затем изменит кортеж и запишет (write) его обратно в хранилище, то второй получит модифицированную версию. Можно считать, что операция «обновления» – это последовательность take и write, которая не приводит к потере данных (при условии, что пространство кортежей в каждый момент времени предоставляет кортеж только одному клиенту).

Метод notify позволяет следить за пространством кортежей и получать уведомления, когда над интересующим вас кортежем была выполнена какая-то операция. Этот метод возвращает объект NotifyTemplateEntry и может наблюдать на операциями четырех видов:

- операции write;
- операции take;
- операции удаления (когда истекает срок хранения кортежа);
- операции закрытия (когда истекает срок хранения объекта NotifyTemplateEntry).

Поскольку операция чтения ничего не изменяет, то система не поддерживает уведомлений о чтениях. В листинге 20.4 приведен пример использования notify.

Листинг 20.4. Уведомление в системе Rinda

```
require 'rinda/tuplespace'

ts = Rinda::TupleSpace.new

alberts = ts.notify "write", ["Albert", nil]
martins = ts.notify "take",  ["Martin", nil]

thr1 = Thread.new do
  alberts.each {|op,t| puts "#{op}: #{t.join(' ')}" }
end

thr2 = Thread.new do
  martins.each {|op,t| puts "#{op}: #{t.join(' ')}" }
end

sleep 1

ts.write ["Martin", "Luther"]
ts.write ["Albert", "Einstein"]
ts.write ["Martin", "Fowler"]
ts.write ["Albert", "Schweitzer"]
ts.write ["Martin", "Scorsese"]
ts.take  ["Martin", "Luther"]

# Выводится:
```

```
# write: Albert Einstein
# write: Albert Schweitzer
# take: Martin Luther
```

Мы видели, что read и другие операции пользуются шаблонами для сопоставления с образцами (и этим напоминают регулярные выражения). Мы уже знаем, что nil выступает в роли метасимвола, но можно указать и класс; ему будет соответствовать любой экземпляр этого класса.

```
tem1 = ["X", Integer]     # соответствует ["X",5], но не ["X","Files"]
tem2 = ["X", NilClass]    # соответствует литералу nil в кортеже
```

Кроме того, разрешается определять собственный оператор ветвящегося равенства (===), если вы хотите проводить сопоставление особым способом. В противном случае для сравнения будет использован стандартный оператор =.

Время жизни кортежа можно задать в момент записи. В сочетании с величинами таймаутов для различных операций над кортежами это позволяет ограничить время выполнения простых и более сложных манипуляций.

Тот факт, что у кортежа может быть конечный срок хранения, означает заодно, что после истечения этого срока кортеж можно обновить с помощью специально написанного объекта. В библиотеке имеется готовый класс SimpleRenewer, который каждые 180 секунд обращается к drb-серверу, создавшему кортеж. Если сервер не отвечает, то кортеж удаляется. Но не пытайтесь программировать обновление, пока не освоитесь с парадигмой пространства кортежей.

В листинге 20.5 приведен еще один пример работы с пространством кортежей. Он решает ту же задачу о производителе и потребителе, которая была рассмотрена в главе 13.

Листинг 20.5. Задача о производителе и потребителе

```
require 'rinda/tuplespace'

ts = Rinda::TupleSpace.new

producer = Thread.new do
  item = 0
  loop do
    sleep rand(0)
    puts "Производитель произвел ##{item}"
ts.write ["Item",item]
    item += 1
  end
end

consumer = Thread.new do
  loop do
    sleep rand(0)
    tuple = ts.take ["Item", nil]
word, item = tuple
    puts "Потребитель потребил ##{item}"
```

```
  end
end

sleep 60   # Работать одну минуту, потом завершиться и завершить потоки
```

20.4. Обнаружение служб в распределенном Ruby

Методика *обнаружения служб* может оказаться полезной, когда имеется много локально работающих служб, поскольку дает возможность находить службу по имени. Если же число служб невелико и их местонахождение точно известно, то особого смысла в автоматическом обнаружении нет.

Раз уж вы продолжили чтение, то, наверное, хотите знать, как работает механизм обнаружения служб. Такую возможность предоставляет библиотека `Rinda::Ring` (естественно, основанная на системе Rinda). В чем-то она похожа на службу DNS, это центральная служба регистрации, в которой хранится информация (в виде пространства кортежей) о `drb`-процессах. Службы `drb` могут по протоколу UDP найти ближайший сервер регистрации, объявить о своем присутствии или найти другие работающие поблизости службы.

Класс `Rinda::RingServer` реализует сервер регистрации. Он управляет пространством имен, в котором хранится информация о местонахождении других drb-служб. RingServer прослушивает сеть в ожидании широковещательных UDP-пакетов с запросами о местонахождении сервера. В ответ на такой запрос он устанавливает соединение (посредством `drb`) с отправившей его службой. Пример:

```
require 'rinda/ring'
require 'rinda/tuplespace'

DRb.start_service
Rinda::RingServer.new(Rinda::TupleSpace.new)
DRb.thread.join
```

Класс `Rinda::RingProvider` регистрирует службу, объявляя о ее присутствии серверу RingServer. При этом сообщается о типе службы и о фронтальном объекте, предоставляющем эту службу, а также передается описательная информация. В следующем примере мы создаем простую службу `Adder`, которая складывает два числа, а потом объявляем о ней всему миру:

```
require 'rinda/ring'

class Adder
  include DRbUndumped

  def add(val1, val2)
    return val1 + val2
  end
end

adder = Adder.new
```

```
DRb.start_service(nil, adder)
Rinda::RingProvider.new(:adder, adder, 'Simple Adder')
DRb.thread.join
```

Класс Rinda::RingFinger (названный так по аналогии с имеющейся в UNIX командой finger) применяется для обнаружения сервера RingServer. Он посылает широковещательный UDP-пакет и ждет ответа от сервера. Пространство кортежей первого ответившего сервера RingServer становится доступно с помощью метода primary. Все прочие обнаруженные серверы доступны с помощью метода to_a. Возвращенное пространство имен можно использовать для поиска объявленных служб:

```
require 'rinda/ring'

DRb.start_service
rs = Rinda::RingFinger.primary
every_space = [rs] + Rinda::RingFinger.to_a
svc = every_space.find_all do |ts|
  ts.read([:name, :adder, nil, nil], 0) rescue nil
end
```

20.5. Заключение

Эта глава содержит введение в распределенный Ruby. Мы познакомились с тем, как служба запускается и взаимодействует с клиентами, а также рассмотрели вопросы безопасности.

Мы также видели, что система Rinda может выступать в роли простого распределенного хранилища объектов, обеспечивающего синхронизацию доступа. Наконец, мы показали, как можно использовать библиотеку Rinda::Ring для обнаружения drb-служб.

На этом рассмотрение распределенного Ruby заканчивается. Переходим к следующей теме – инструментам разработки на языке Ruby, в частности, программе rake, оболочке irb и другим.

Глава 21. Инструменты разработки для Ruby

Человек — это животное,
создающее инструменты.

Бенджамин Франклин

Среда разработки включает не только интерпретатор. У каждого хорошего разработчика под рукой всегда есть целый набор средств, облегчающих жизнь. Некоторые из них зависят от языка, другие – нет.

Самый важный инструмент – редактор. Основная работа программиста – манипулирование текстом, поэтому выбор редактора (и ваше умение им пользоваться) оказывает огромное влияние на производительность труда. Немаловажно также наличие в нем поддержки конкретного языка или средств настройки. Мы дадим очень краткий обзор распространенных редакторов.

Другие инструменты помогают в составлении документации, установке библиотек, отладке и т.д. Мы уже познакомились с библиотекой отладки Byebug (не являющейся автономным приложением) в главе 16, а в главе 17 рассказали о системах Rubygems, Bundler и RDoc. Поэтому здесь мы их касаться не будем.

Мы начнем эту главу с программы `rake`, варианта утилиты UNIX `make` для Ruby. Далее будут рассмотрены интерактивные оболочки Ruby `irb` и `pry`, а также программа `ri` (командная утилита для чтения документации). И завершим кратким обсуждением менеджеров версий Ruby.

21.1. Программа Rake

Утилита `rake` – это вариация на тему давно известной в UNIX программы `make`. Но вместо сложного и трудного для использования синтаксиса Makefile, который все мы знаем и терпеть не можем, в rake используется код на самом Ruby. Программу написал Джим Вайрих, это первый (насколько мне известно) пример формальной реализации языка DSL (domain-specific language – язык предметной области) на Ruby.

Есть два варианта написания названия: Rake и `rake`. Первый – это название инструмента, второй – имя самого исполняемого файла. На мой взгляд, различие несущественное.

Rake (на дизайн которой несомненно оказала большое влияние `make`) позволяет определять задачи и зависимости между ними, а затем одним движением выпол-

нять саму задачу и все, от которых она зависит. Терминология та же, что в make: цели, действия, зависимости и правила.

Применениям Rake нет числа. Вы можете пользоваться ей для сборки проектов, написанных на C, C++ или Java (или на любом другом языке). Годится она и для генерирования документации с помощью RDoc, развертывания программ, обновления проекта на сайте RubyForge и множества других задач.

Не удивительно, что на вход Rake подается командный файл (rake-файл), который по умолчанию называется Rakefile. Если вы хотите назвать его иначе, укажите имя с помощью флага -f или --rakefile:

```
$ rake             # искать сначала 'rakefile', потом 'Rakefile'
$ rake -f myfile   # использовать файл с именем 'myfile'
```

Основная «единица работы» в Rake – *задача*; именуются они посредством символов Ruby. Предполагается, что в каждом rake-файле есть задача по умолчанию :default, она будет выполняться, если явно не указано другое имя.

```
$ rake             # выполнить задачу по умолчанию
$ rake mytask      # выполнить задачу с именем 'mytask'
```

Внутри rake-файла мы объявляем и создаем задачи с помощью метода task, передавая ему символ и блок:

```
task :mytask do
  # ...
end
```

То, что находится в блоке, называется *действием*. Оно выполняется, когда мы запускаем задачу по имени. В действии может находиться произвольный код на Ruby. Для типичных операций предусмотрены готовые методы. Так, метод sh (имя которого напоминает об интерпретаторе команд в UNIX) запускает системную команду. Методы cp, mv и rm предназначены соответственно для копирования, перемещения и удаления файлов. (Как и make, Rake беззастенчиво выдает свое происхождение от UNIX). Полный перечень вспомогательных команд можно найти в документации по Rake API на сайте rdoc.info или ruby-doc.org.

При желании можете заключать блок в фигурные скобки, но обычно в этом случае интерпретатор Ruby ожидает, что параметры будут заключены в круглые скобки.

```
task(:mytask) { do_something }
```

Рассмотрим более конкретный пример. Предположим, что имеется программа myprog.c, написанная на C, и еще два связанных с ней исходных файла (каждый со своим заголовочным файлом). Иными словами, у нас есть пять исходных файлов:

```
myprog.c
sub1.c
sub1.h
sub2.c
sub2.h
```

Мы хотим собрать из них исполняемый файл myprog. Эта процедура состоит из нескольких шагов: откомпилировать все файлы с расширением .c, а затем скомпоновать получившиеся в результате файлы с расширением .o. Начнем с метода file, который определяет зависимости файлов:

```
file "myprog.o" => ["myprog.c"]
file "sub1.o" => ["sub1.c", "sub1.h"]
file "sub2.o" => ["sub2.c", "sub2.h"]
file "myprog" => ["sub1.o", "sub2.o"]
```

Отметим, что метод file принимает хэш, который ассоциирует имя файла с массивом имен файлов, от которых он зависит.

Теперь перейдем к построению двоичных файлов. Расширим написанный выше код. Если после вызова метода file указать блок, то мы сможем ассоциировать с файлом набор действий, которые необходимо выполнить для создания этого файла:

```
file "myprog.o" => ["myprog.c"] do
  sh "cc -c -o myprog.o myprog.c"
end

file "sub1.o" => ["sub1.c", "sub1.h"] do
  sh "cc -c -o sub1.o sub1.c"
end

file "sub2.o" => ["sub2.c", "sub2.h"] do
  sh "cc -c -o sub2.o sub2.c"
end

file "myprog" => ["sub1.o", "sub2.o"] do
  sh "cc -o myprog myprog.o sub1.o sub2.o"
end
```

Здесь имеется некоторое дублирование, но от него можно избавиться. В Rake есть специальный механизм, который называется FileList; он понимает метасимволы (шаблоны) и позволяет работать сразу с несколькими файлами. В данном случае поместим все файлы с расширением .c в список SRC. Константа типа FileList ведет себя, как массив:

```
FileList["*.c"]
```

Таким образом, для определения действий можно использовать цикл each, как показано ниже. И обратите внимание, что зависимости здесь не упоминаются, Rake самостоятельно учтет эту информацию, если она была задана в другом месте.

```
FileList["*.c"].each do |src|
   obj = src.sub(/.c$/,".o")
   file(obj) { sh "cc -c -o #{obj} #{src}" }
end
```

Однако проще пользоваться правилами. Это еще один механизм Rake, естественно, позаимствованный у make:

```
rule '.o' => '.c' do |target|
  sh "cc -c -o #{target.name} #{target.source}"
end
```

Тут мы наблюдаем небольшое волшебство. Rake устанавливает атрибут source, подставляя расширение имени файла из хэша (в данном случае .o заменяется на .c).

Продолжим сеанс магии. Если затребовать библиотеку rake/clean, то появляются константы CLEAN и CLOBBER (первоначально пустые) и задачи :clean и :clobber. По сложившейся традиции clean удаляет временные файлы, а clobber – еще и собранный исполняемый файл.

Для этих напоминающих массив констант определен метод include, принимающий маску имени файлов; это неявное использование FileList.

Теперь наш Rakefile принимает такой вид:

```
require 'rake/clean'

CLEAN.include("*.o")
CLOBBER.include("myprog")

SRC = FileList['*.c']
OBJ = SRC.ext('o')

rule '.o' => '.c' do |t|
  sh "cc -c -o #{t.name} #{t.source}"
end

file "hello" => OBJ do
  sh "cc -o hello #{OBJ}"
end

file "myprog.o" => ["myprog.c"]
file "sub1.o" => ["sub1.c", "sub1.h"]
file "sub2.o" => ["sub2.c", "sub2.h"]

task :default => ["myprog"]
```

Обратите внимание, что мы не задавали задачи «clean» и «clobber» явно. Кроме того, отметим что «clobber» неявно включает операцию «clean». И, наконец, мы определили задачу default для удобства запуска Rakefile, теперь можно не указывать явно имя задачи, осуществляющей компиляцию и сборку.

У программы Rake есть несколько параметров командной строки. Иногда желательно протестировать Rakefile, не выполняя никаких (потенциально опасных) операций; для этого служит флаг -n или --dry-run. Флаг -T выводит список всех целей в Rakefile. Имеются также флаги, управляющие поиском библиотек, трассировкой, протоколированием и т.д.

Программа Rake сложнее, чем я описал в этом разделе (особенно это касается правил). Популярное средство развертывания Capistrano основано на Rake. Кро-

ме того, в большинстве gem-пакетов Rake в той или иной мере используется для автоматизации повторяющихся задач. Дополнительные сведения о том, как решить конкретную интересующую вас проблему, можно найти в документации по Rake (на сайте rdoc.info или ruby-doc.org), а также в различных сетевых форумах и в учебных пособиях.

21.2. Оболочка irb

Утилита `irb` (интерактивный Ruby) уже много лет как включена в дистрибутив Ruby. Можно считать, что это «испытательный стенд» или «площадка для игр», на которой вы опробуете новые приемы и идеи.

Работать с `irb` просто. После запуска вы получаете приглашение и можете вводить произвольные выражения Ruby. Выражение вычисляется и печатается результат. Вот пример короткого сеанса:

```
$ irb
irb(main):001:0> "cell" + "o"
=> "cello"
irb(main):002:0> 3*79
=> 237
irb(main):003:0> Dir.entries(".").size
=> 17
irb(main):004:0> rand
=> 0.850757389880155
irb(main):005:0> rand
=> 0.679879756672551
irb(main):006:0> defined? foo
=> nil
irb(main):007:0> defined? Object
=> "constant"
irb(main):008:0> quit
$
```

Конечно, это больше, чем калькулятор. При желании можно ввести произвольный код на Ruby:

```
[hal@localhost ch21]$ irb
irb(main):001:0> require 'mathn'
=> true
irb(main):002:0> gen = Prime.instance
=> #<Prime:0x000001019e62e8>
```

При задании флага `-r` выполняется `require`, поэтому можно включать код из внешнего файла. Предположим, что вы написали такой файл:

```
# File: foo.rb
class MyClass

  attr_accessor :alpha, :beta

  def initialize(a, b)
```

```
    @alpha, @beta = a, b
  end

end

obj1 = MyClass.new(23,34)
obj2 = MyClass.new("abc","xyz")
```

Тогда можно сделать следующее:

```
$ irb -rfoo
irb(main):001:0> obj = MyClass.new(88,99)
=> #<MyClass:0x00000102830c90 @a=88, @b=89>
```

Отметим, что хотя обращаться к сущностям, определенным в файле (например, к константе `MyClass`) мы можем, это не относится к локальным переменным. Локальные переменные доступны только в самом файле, `require` (выполненное хоть внутри, хоть вне `irb`) доступа к ним не дает.

Новичков часто смущает информация, выводимая `irb`:

```
$ irb -rfoo
irb(main):001:0> puts "hello"
hello
=> nil
```

А что тут делает `nil`? Разумеется, это значение, возвращенное методом `puts`.

Имейте в виду, что `irb` настраивается в широких пределах. При запуске она читает все инициализационные файлы, которые может найти, в следующем порядке:

- файл `~/.irbrc`;
- файл `.irbrc`;
- файл `irb.rc`;
- файл `_irbrc`.

Инициализационные файлы пишутся на Ruby. В них можно настраивать текст приглашения и многое другое. Подробно эти файлы обсуждаются в книге Dave Thomas, Andy Hunt «Programming Ruby» (ее еще называют «киркомотыгой» по изображению на обложке). Ниже мы сделаем лишь несколько замечаний.

Если ваша версия Ruby собрана с поддержкой библиотеки GNU `readline` (обычно так и есть), то вы можете перемещаться по истории команд с помощью клавиш со стрелками вверх и вниз. Еще важнее, что работает механизм автозавершения по клавише Tab: когда вы набираете начало идентификатора, а затем нажимаете Tab, `irb` пытается дописать имя идентификатора за вас.

Чтобы включить механизм автозавершения, добавьте в файл `.irbrc` следующий фрагмент:

```
IRB.conf[:AUTO_INDENT] = true
IRB.conf[:USE_READLINE] = true
IRB.conf[:LOAD_MODULES] ||= []
IRB.conf[:LOAD_MODULES] |= ['irb/completion']
```

В файле .irbrc может содержаться произвольный код. Например, я часто пользуюсь описанным ниже методом. Для краткости он назван sm (сокращение от «show methods»), а цель – вывести (в алфавитном порядке) список всех методов, которые можно вызывать для данного объекта, за исключением тех, которые он унаследовал от своих предков:

```
def sm(obj)
  list = obj.methods
  anc = obj.class.ancestors - [obj.class]
  anc.each {|a| list -= a.instance_methods }
  list.sort
end
```

Вот пример его использования:

```
irb(main):001:0> str = "hello"
=> "hello"
irb(main):002:0> sm str
=> [:%, :*, :+, :<<, :[], :[]=, :ascii_only?, :b, :bytes,
    :bytesize, :byteslice, :capitalize, :capitalize!, :casecmp,
    :center, :chars, :chomp, :chomp!, :chop, :chop!, :chr, :clear,
    :codepoints, :concat, :count, :crypt, :delete, :delete!,
    :downcase, :downcase!, :dump, :each_byte, :each_char,
    :each_codepoint, :each_line, :empty?, :encode, :encode!,
    :encoding, :end_with?, :force_encoding, :getbyte, :gsub,
    :gsub!, :hex, :include?, :index, :insert, :intern, :length,
    :lines, :ljust, :lstrip, :lstrip!, :match, :next, :next!, :oct,
    :ord, :partition, :prepend, :replace, :reverse, :reverse!,
    :rindex, :rjust, :rpartition, :rstrip, :rstrip!, :scan, :scrub,
    :scrub!, :setbyte, :size, :slice, :slice!, :split, :squeeze,
    :squeeze!, :start_with?, :strip, :strip!, :sub, :sub!, :succ,
    :succ!, :sum, :swapcase, :swapcase!, :to_c, :to_f, :to_i,
    :to_r, :to_str, :to_sym, :tr, :tr!, :tr_s, :tr_s!, :unpack,
    :upcase, :upcase!, :upto, :valid_encoding?]
irb(main):003:0> sm String
=> [:allocate, :new, :superclass, :try_convert]
irb(main):004:0> sm 123
=> [:&, :*, :**, :+, :-, :/, :<<, :>>, :[], :^, :bit_length, :size, :to_f, :|, :~]
```

Программа irb позволяет запускать подсеансы внутри сеанса, хотя это используется и нечасто. Можно запустить несколько сеансов и переключаться между ними, у каждого сеанса собственный набор привязок.

Может быть, вам это и не кажется полезным, но можно, например, задать объект в сочетании с подкомандой irb. Тогда контекстом подсеанса станет этот объект, псевдопеременная self будет ссылаться на него, он же станет областью видимости и т.д.:

```
$ irb
irb(main):001:0> t0 = Time.now
=> 2014-08-29 19:01:42 -0700
```

```
irb(main):002:0> irb t0
irb#1(2014-08-29 19:01:42 -0700):001:0> strftime("%a %b %c")
=> "Fri Aug Fri Aug 29 19:01:42 2014"
irb#1(2014-08-29 19:01:42 -0700):002:0> to_i
=> 1409364102
irb#1(2014-08-29 19:01:42 -0700):003:0> self + 1000
=> 2014-08-29 19:18:22 -0700
irb#1(2014-08-29 19:01:42 -0700):004:0> wday
=> 5
irb#1(2014-08-29 19:01:42 -0700):005:0> class
SyntaxError: (irb#1):5: syntax error, unexpected end-of-input
irb#1(2014-08-29 19:01:42 -0700):006:0> self.class
=> Time
irb#1(2014-08-29 19:01:42 -0700):007:0> quit
=> #<IRB::Irb: @context=#<IRB::Context:0x007ff9430437d0>,
  @signal_status=:IN_EVAL, @scanner=#<RubyLex:0x007ff94303b238>>
irb(main):003:0> quit
$
```

Иногда для изучения результатов выполнения Ruby-кода полезна библиотека xmpfilter, входящая в состав gem-пакета rcodetools. Она принимает предложения на Ruby, исполняет их и помещает возвращаемое значение в комментарий.

У irb есть еще одна приятная особенность, о которой стоит знать. Понятно, что irb умеет анализировать написанный на Ruby код, но лексический анализатор несложно использовать и в других приложениях. Вот простой пример программы, которая открывает саму себя и анализирует собственный код, выводя отсортированный список всех идентификаторов и констант:

```
require 'irb/ruby-lex'

file = File.new(__FILE__)

parse = RubyLex.new # (file)
parse.set_input(file)

idents = []

loop do
  token =  parse.token
  break if token.nil?
  if token.is_a? RubyToken::TkIDENTIFIER or
     token.is_a? RubyToken::TkCONSTANT
    idents << token.name
  end
end

p idents.uniq.sort

# Выводится:
```

```
# ["File", "RubyLex", "RubyToken", "TkCONSTANT", "TkIDENTIFIER",
#  "file", "idents", "loop", "name", "new", "p", "parse",
#  "require", "set_input", "sort", "token", "uniq"]
```

Насколько мне известно, этот механизм не документирован на английском языке. Но если вам необходим лексический анализатор Ruby, то можете изучить исходный текст и адаптировать его под свои нужды.

21.3. Основы pry

Программа pry не входит в стандартный дистрибутив Ruby, но пользуется огромной популярностью. Можете считать, что это «улучшенная irb».

Ее использование для отладки описано в главе 16. А здесь мы рассмотрим, как pry помогает организовать интерактивную разработку.

Поскольку pry – сложный инструмент с широкими возможностями, здесь мы можем затронуть лишь самые азы. Как обычно, все подробности в документации.

В pry встроены весьма развитые средства работы с клавиатурой. Разумеется, клавиши со стрелками позволяют перемещаться по истории команд, но существуют также команды для просмотра и редактирования буфера ввода, чего irb не поддерживает.

Вот лишь некоторые относящиеся сюда команды:

```
show-input              Показать весь буфер ввода
amend-line n newtext    Заменить строку n текстом newtext
!                       Выйти из цикла чтения и опустошить буфер ввода
hist -tail k            Показать строку истории с номером k
hist -replay m..n       Воспроизвести строки истории с номерами от m до n
edit                    Открыть буфер ввода во внешнем редакторе
```

Команду edit можно также использовать для редактирования класса, метода и даже просто файла, содержащего исполняемый в данный момент код (edit -c). После закрытия редактора pry перезагрузит измененный файл, перезаписав существующие определения класса и метода взятыми из файла.

Как и в irb, знак подчеркивания (_) позволяет получить значение последнего вычисленного выражения. Чтобы вычислить выражение, не отображая это значение, можно поставить в конце выражения точку с запятой; это полезно, например, когда результатом вычисления является очень длинная строка. Наконец, ввод и вывод хранятся соответственно в переменных _in_ и _out_:

```
_           Значение последнего вычисленного выражения
expr;       Вычислить выражение, но не печатать значение
_in_        Ввод с консоли
_out_       Вывод на консоль
```

Чтобы передать команду операционной системе, нужно поставить перед ней точку. Разрешается также интерполировать в команду Ruby-код:

```
.ps aux                 Выполнить команду "ps aux"
.ps aux | grep #{prog} Выполнить ps aux и искать grep'ом значение переменной prog
```

Существует также режим оболочки. По сути дела, это означает добавление текущего каталога в приглашение pry и разрешение производить автозавершение имени файла (при определенных обстоятельствах).

Команда show-source показывает исходный код, в том числе код самой программы pry, написанных на C расширений и т. д. Ее поведение можно модифицировать с помощью задания параметров.

Команда show-doc выводит документацию. Она умеет показывать информацию о классах, методах, частях ядра Ruby и т. д. С ней тесно связаны команды stat и ri (последняя вызывает программу ri, которая будет описана в следующем разделе).

Наконец, команда help выводит информацию о возможностях и порядке вызова pry. После онлайновых руководств и пособий help – лучший источник знаний об этом замечательном инструменте.

21.4. Утилита ri

Сокращение ri означает «Ruby index» или нечто подобное. Это командная утилита для чтения документации, предоставляющая информацию о классах, методах, модулях и т.д. Пример:

```
(из ядра ruby)
Implementation from Enumerator
  e.with_index(offset = 0) {|(*args), idx| ... }
  e.with_index(offset = 0)


Iterates the given block for each element with an index, which
starts from offset. If no block is given, returns a new Enumerator
that includes the index, starting from offset

offset
  the starting index to use
```

По умолчанию при установке gem-пакетов с помощью Rubygems генерируется также документация в формате ri, которую затем можно просмотреть в ri.

21.5. Поддержка со стороны редакторов

Любой современный редактор должен проявлять дружелюбие по отношению к программистам. Память о двадцатом веке уходит в прошлое, и мы принимаем как должное, что редакторы изменяют свое поведение в зависимости от типа редактируемого файла. Мы ожидаем синтаксической подсветки, автоматического формирования отступов и прочего в том же духе.

Программист, пишущий на Ruby, не будет разочарован имеющимся инструментарием. Многие возможности стандартно включены в редактор, другие приходится настраивать. Кратко рассмотрим, что имеется в нашем распоряжении.

Два прекрасных графических редактора – SublimeText 3 ($30, имеются версии для Windows, Mac OS X и Linux) и TextMate 2 (бесплатный, только для Mac OS X).

Оба хорошо поддерживают Ruby и расширяются путем добавления команд и текстовых фрагментов для повышения эффективности кодирования. TextMate в последние годы немного растерял былую популярность, отчасти потому, что между версиями 1 и 2 прошло несколько лет, но тем не менее остается востребованным. У SublimeText много поклонников, и его популярность только растет.

Третий редактор – RubyMine, основанный на Eclipse IDE для Java. Разработчикам, которые переходят с Java на Ruby, он может показаться более знакомым, чем альтернативы. Это не просто редактор, а полноценная IDE – пожалуй, лучшая (и последняя из выживших). Это коммерческий продукт, но он стоит своих денег, если вам нужны мощные средства редактирования, а также интеграция с различными инструментами и документацией.

Хотя в последние десятилетия популярность графических редакторов резко возросла, среди программистов по-прежнему широко распространены Vim и Emacs. Дадим их краткий обзор.

21.5.1. Vim

Пользователям vim средства работы с Ruby доступны благодаря подключаемому модулю `vim-ruby`. Это набор конфигурационных файлов, поддерживающий синтаксическую подсветку, автоматическое формирование отступов и автозавершение кода. Кроме того, он позволяет вызывать интерпретатор Ruby прямо из редактора (подключаемые модули компиляции).

`vim-ruby` включен в дистрибутив Vim, начиная с версии 7. Но может оказаться, что эти возможности не активированы (возможно, в целях обратной совместимости). Тогда нужно их активировать, добавив в файл `.vimrc` следующие строки:

```
set nocompatible
syntax on
filetype on
filetype indent on
filetype plugin on
```

Кроме того, стоит установить несколько великолепных подключаемых к Vim модулей, написанных Тимом Поупом (Tim Pope). Их можно найти в GitHub на его странице по адресу https://github.com/tpope:

- **pathogen** – менеджер подключаемых к Vim модулей;
- **unimpaired** – удобные команды с использованием квадратных скобок;
- **surround** – служит для управления скобками, кавычками, тегами XML и прочим;
- **fugitive** – интеграция Git с Vim;
- **rails** – улучшенная навигация по файлам Rails и синтаксическая подсветка;
- **dispatch** – диспетчер сборки и тестирования.

Если вы используете Vim для редактирования Ruby-кода, рекомендую познакомиться с этими и другими подключаемыми модулями для Ruby.

В принципе, есть много причин не любить Vim, одна из самых основательных – язык vimscript. Хорошая новость – теперь есть возможность программировать поведение Vim на Ruby. И плохая – она неважно документирована. Если хотите

узнать подробнее, рекомендую для начала набрать команду :help ruby в Vim. Дополнительная информация имеется также в описании VimRubyInterface. Наконец, поищите на сайте http://vim/org; может быть, там появилось что-то новенькое.

21.5.2. Emacs

Еще один популярнейший в мире редактор – emacs. Вообще-то называть его редактором не вполне правильно. Это, скорее, миниатюрная операционная система, которая заодно позволяет и редактировать тексты. Одно из главных достоинств emacs – поразительная расширяемость, пользователь программирует его поведение на диалекте языка Lisp (elisp). Язык elisp мощнее, чем vimscript, но читать написанные на нем программы не менее сложно (правда, по другой причине).

Начиная с версии 23, emacs поставляется со встроенной поддержкой Ruby в форме режима Ruby Mode. Последние версии Ruby Mode включены и в дистрибутив Ruby. Я сам не работаю с emacs, но очень уважаю тех, кто им пользуется. Поэтому больше мне сказать нечего. За дополнительными сведениями отсылаю вас на Emacs Wiki, можете также поискать в Интернете по фразе «emacs ruby».

21.6. Менеджеры версий Ruby

В последние годы разработаны инструменты, помогающие устанавливать различные версии Ruby и переключаться между ними. Управление gem-пакетами относится сюда же, потому что версии пакетов могут меняться вместе с версиями Ruby (а иногда и чаще).

Ruby Version Manager (rvm) – весьма полезное дополнение к инструментарию, появившееся в 2010 году, а то и раньше. Я постоянно использую его в паре с Bundler. Но мы рассмотрим также программы rbenv и chruby.

21.6.1. Работа с rvm

Первый менеджер версий называется просто – Ruby Version Manager, или RVM. Он позволяет устанавливать несколько версий Ruby, переключаться между ними и назначать версию по умолчанию для машины в целом или для отдельного проекта. О том, как установить и использовать RVM, можно прочитать на сайте http://rvm.io.

Главное, что нужно понимать относительно rvm, – тот факт, что он никак не взаимодействует с «системной установкой Ruby». Он управляет лишь теми версиями, которые установил сам.

Ниже перечислено несколько команд:

```
rvm install ruby-2.0.0    Установить и откомпилировать версию Ruby 2.0.0 и управлять ей
rvm uninstall ruby-1.9.2  Удалить версию 1.9.2
rvm use ruby-1.9.4        Сделать "текущей" версию 1.9.4
rvm use system            Переключиться на неуправляемую системную версию Ruby
rvm list                  Показать список всех установленных версий Ruby
rvm current               Показать сведения о текущем окружении Ruby
```

В rvm имеется понятие *набора gem-пакетов* (gemset), это группа взаимосвязанных gem-пакетов, имеющих отношение к текущему проекту или окружению. Но эта функциональность, пожалуй, лучше реализована в Bundler.

По умолчанию подразумевается «обычный» интерпретатор (MRI, «Matz's Ruby Interpreter»). Но поддерживаются и другие интерпретаторы, в том числе JRuby, Rubinius, REE и MacRuby. Дополнительные сведения см. на сайте RVM (http://rvm.io).

Принятый в RVM подход охватывает все, вносятся изменения в работу оболочки, Rubygems и Bundler. Многие считают, что это слишком сложно, чревато ошибками и вообще – вмешательство в окружение ОС недопустимо. Были предложены более простые альтернативы. Самые популярные из них – `rbenv` и `chruby`. Их мы и рассмотрим.

21.6.2. Работа с rbenv

Программа `rbenv` поддерживает использование нескольких версий Ruby. Ее лозунг – «Делай только одну вещь, но просто и хорошо». Поэтому она не занимается установкой Ruby, не имеет средств управления наборами gem-пакетов и слабо зависит от оболочки ОС.

Те, кто пользуется `rbenv`, часто применяют для установки `ruby-build`, а для управления gem-пакетами – `bundler`. Эти инструменты практически не зависят друг от друга.

`rbenv` определяет, какой интерпретатор Ruby использовать, читая три источника: переменную окружения `RBENV_VERSION`, локальный файл `.ruby-version` и глобальный файл `~/.rbenv/version`. Файл `.ruby-version` может находиться в каталоге относительно самого приложения или относительно текущего каталога.

Если в системе установлена программа `ruby-build`, то команда `rbenv install` вызовет ее для установки версии Ruby. Это разные пакеты, которые могут работать совместно.

Ниже приведено несколько команд `rbenv`:

```
rbenv versions          Вывести список всех версий Ruby
rbenv local ruby-2.0.0  Установить версию Ruby для данного каталога
rbenv global ruby-1.8.7 Установить глобальную версию Ruby
rbenv shell jruby-1.7.1 Установить зависящую от оболочки версию Ruby
rbenv rehash            Выполняется после установки gem-пакетов или версий Ruby,
                        чтобы соответствующие команды вступили в силу
rbenv which rake        Показать полный путь к команде
rbenv whence rackup     Вывести список всех версий Ruby, в которых
                        есть указанная команда
```

Отметим, что зависящая от оболочки версия имеет больший приоритет, чем привязанная к каталогу локальная версия, и обе они приоритетнее глобальной версии. Если устанавливается глобальная или локальная версия, то `rbenv` записывает номер версии в локальный или глобальный файл с именем, начинающимся точкой. Если устанавливается зависящая от оболочки версия, то `rbenv` устанавливает переменную окружения.

Команда `rehash` «освежает» знания оболочки об установленных версиях Ruby и gem-пакетов. Выполняйте ее всегда после установки новой версии Ruby или нового пакета командой `gem install`.

Отметим, что rbenv и rvm несовместимы. Если вы установили программу rvm, то перед установкой rbenv ее необходимо удалить.

21.6.3. Работа с chruby

chruby – самый минималистский инструмент управления версиями. На момент написания этой книги его основу составляли менее 100 строк кода на bash. Он умеет устанавливать версии Ruby с помощью программы ruby-install (того же автора) или ruby-build (созданной автором rbenv).

Chruby можно установить непосредственно с GitHub или с помощью менеджера пакетов типа Homebrew или rpm. Подробные инструкции по установке имеются в репозитории chruby на GitHub.

Автор стремился сделать программу интуитивно понятной. Она устанавливает переменные окружения PATH, GEM_HOME, GEM_PATH и RUBY_ROOT. По умолчанию подразумевается системная установка Ruby и поддерживается приблизительное сравнение с названиями версиями. Имеется факультативная поддержка автоматического переключения активной версии Ruby с помощью файлов .ruby-version, о существовании которых знают также rvm и rbenv.

chruby не подменяет команду cd функцией оболочки (как rvm) и не замещает команду ruby и все команды из установленных gem-пакетов скриптами-обертками на языке оболочки (как rbenv). Она не требует прав на запись в установочный каталог Ruby для установки gem-пакетов и не настаивает на том, чтобы версии Ruby устанавливались в ваш домашний каталог.

Порядок вызова Chruby следующий:

```
chruby              Вывести список всех установленных версий Ruby
chruby 2.1.0        Переключиться на версию Ruby 2.1.0
chruby system       Переключиться на системную установку Ruby
```

chruby поддерживает оболочки bash и zsh. Она допускает переход от rvm или rbenv с помощью переменной RUBIES. Кроме того, она предлагает ограниченную поддержку взаимодействия с такими инструментами, как capistrano, chef, puppet и некоторыми другими.

Дополнительные сведения можно найти в онлайновой документации по chruby. А в репозитории chruby на GitHub (github.com/postmodern/chruby) имеется файл readme и wiki – они могут служить неплохой отправной точкой.

21.7. Заключение

В этой главе мы рассмотрели ряд инструментов, облегчающих жизнь программисту на Ruby. К числу общеупотребительных относятся irb, rvm и другие. Мы также упомянули редакторы и IDE с поддержкой Ruby.

Но самый главный инструмент – не программы, а люди, то есть все сообщество пользователей Ruby. Я призываю вас искать группы пользователей в сети и в реальной жизни: на Google и на Stack Overflow, в IRC (обратите особое внимание на сайт Freenode) и на конференциях, посвященных Ruby (местных и международных). Мы всегда готовы прийти вам на помощь.

Глава 22. Сообщество пользователей Ruby

Кто не способен жить в обществе
или не имеет такой потребности,
поскольку самодостаточен,
либо животное, либо бог.

— Аристотель «Политика»

Уже было сказано, что одна из лучших черт Ruby – сообщество его пользователей. Конечно, это дело вкуса, вы составите собственное мнение. Моя цель в настоящей главе – представить вашему вниманию некоторые «посиделки»: источники новостей и новых знаний, форумы как в сети, так и вне нее, а также места, где «рубисты» встречаются в киберпространстве и в реальной жизни.

Эта глава намеренно сделано очень краткой. Значительная часть приведенной в ней информации стабильна, но ведь в жизни все меняется. Если сомневаетесь, воспользуйтесь поисковой машиной.

22.1. Ресурсы в веб

Основной сайт Ruby – `ruby-lang.org`, на нем всегда можно найти самую свежую версию Ruby, а также сведения о выпусках.

Если вас интересует документация, не пропустите сайт ruby-doc.org, который поддерживает Джеймс Бритт. На нем собраны документы, сгенерированные программой RDoc для системных и стандартных библиотек. На сайте `rdoc.info` представлена сгенерированная программой Yard документация не только по самому языку Ruby, но и по всем имеющимся gem-пакетам и относящимся к Ruby репозиториям на GitHub.

Файл с говорящим названием `iwanttolearnruby.com`, созданный Амандой Вагенер (Amanda Wagener), содержит полный список посвященных Ruby ресурсов, который можно сортировать по типу или уровню ресурса. Он будет полезен разработчикам любой квалификации.

В различных местах сети существует много других ресурсов по Ruby, нетрудно найти книги, учебные пособия, статьи и т. п.

22.2. Новостные группы и списки рассылки

Список рассылки `ruby-talk` – наверное, старейший англоязычный форум для программистов на Ruby. На сайте ruby-talk.org размещена вся история этого списка,

и по ней можно производить поиск (там же находится несколько родственных списков, в частности ruby-core и ruby-math).

Список рассылки Ruby Weekly (rubyweekly.com) еженедельно отправляет одно письмо с подборкой интересных сообщений, видео, событий и предложений работы.

Ruby Rogues (rubyrogues.com) – продолжительный еженедельный подкаст, в котором публикуются дискуссии и интервью с приглашенными гостями. Каждый эпизод представлен в виде аудиозаписи или ее стенограммы на сайте, а в архивах за последние несколько лет можно найти различные темы, интересные рубистам.

Кроме того, Ruby Rogues ведет популярный и полезный форум Parley. Участие в нем стоит 10 долларов в год, но эта сумма гарантирует, что участники действительно заинтересованы в обсуждениях, и качество дискуссий обычно довольно высокое.

22.3. Извещения об ошибках и предложения новых функций

Язык Ruby продолжает развиваться. К достоинствам языка следует отнести и то, что изменяется он медленно и обдуманно.

Но Ruby, конечно, не достиг совершенства. Для сбора отзывов, предложений новых функций и извещений об ошибках группа разработки ядра Ruby пользуется системой отслеживания проблем по адресу bugs.ruby-lang.org, основанной на программе Redmine с открытым исходным кодом (естественно, написанной на Ruby).

Прежде чем подавать новый запрос, убедитесь, что никто не опередил вас. Языком Ruby пользуется так много людей во всем мире, что об ошибке с большой вероятностью уже сообщали и, возможно, она уже даже исправлена. Если же вы все-таки обнаружили новую проблему, прочитайте инструкцию по подаче извещений об ошибках по адресу bugs.ruby-lang.org/projects/ruby/wiki/HowToReport.

Никто не гарантирует, что ошибка будет исправлена или что ваш запрос будет одобрен. Большинство разработчиков ядра Ruby тратят свое личное время, а окончательное решение о новых функциях принимает Мац. Но чем тщательнее вы обдумаете свое предложение заранее, тем больше шансов, что оно будет принято всерьез. Список рассылки ruby-talk – тоже подходящее место для обсуждения ошибок и идей.

22.4. Каналы IRC

Можно побиться об заклад, что в любое время суток десятки «рубистов» болтают в IRC-чатах. Такие серверы принадлежат freenode.net, зайдите на этот сайт и поищите ближайший к вам. Поиск в Веб поможет вам найти подходящий IRC-клиент для любой платформы.

В каналах #ruby-lang и #ruby-on-rails довольно интенсивный трафик. Это международный форум, так что кого-нибудь вы там встретите в любое время суток.

Языком международного общения де факто является английский, но кто-нибудь наверняка подскажет вам каналы, где говорят на вашем языке.

Соблюдайте общепринятые в IRC правила общения. Главное, не «флудите», посылая фрагменты кода, одна-две строки на сообщение – это максимум. При необходимости пользуйтесь службой хранения фрагментов `gist.github.com` или аналогичной.

22.5. Конференции по Ruby

Первая международная конференция по Ruby (RubyConf) состоялась во Флориде в 2001 году, затем города проведения менялись. Чтобы узнать, где и когда состоится следующая конференция, зайдите на сайт rubyconf.org Обычно на конференции приезжает и сам Мац, и еще несколько японских рубистов, вообще участники прибывают со всех континентов. (Если вы работаете на антарктической научной станции, не забудьте взять отпуск и посетить очередную конференцию.)

Некоммерческая организация RubyCentral со штаб-квартирой в США, организующая RubyConf, взялась также за организацию еще более крупной ежегодной конференции RailsConf, которая впервые была проведена в Чикаго в 2006 году. Чтобы узнать о месте и времени проведения следующей конференции, подать заявку на выступление с докладом или зарегистрироваться, зайдите на сайт railsconf.org.

Региональные и национальные конференции проводятся и в других местах по всему миру. Первая европейская конференция по Ruby (EuRuKo) состоялась в Карлсруэ, Германия, в 2003 году. Там обычно бывает меньше народу, но если вы живете в Европе, то добираться удобнее. В 2013 году я (Хэл) выступал с докладом на конференции RubyConf в бразильском городе Сан-Паулу и обнаружил, что это весьма оживленное мероприятие с сотнями участников. В Токио каждый год проводится национальная конференция RubyKaigi, где я (Андрэ) выступал в 2014 году.

Популярность Ruby возросла настолько, что ему уже были посвящены выступления на конференциях OOPSLA (Object-Oriented Programming, Systems, Languages, and Applications – объектно-ориентированное программирование, системы, языки и приложения) и OSCON (Open Source Convention – конвенция о программном обеспечении с открытыми исходными текстами), и не только. То ли еще будет.

Популярность каркаса Ruby on Rails привела к организации посвященных только ему конференций. Первая международная конференция по Rails состоялась в Чикаго в июне 2006 года, другие уже на подходе. Свежую информацию ищите на сайте railsconf.org.

В апреле 2006 года в городе Санта-Клара в Кремниевой долине состоялась первая региональная конференция по Ruby в США, и она оказалась отнюдь не последней. Начиная с 2007 года в Остине, штат Техас, проводится конференция Lone Star Ruby. Из популярных конференций упомянем еще Madison+Ruby (madisonpl.us/ruby), Gotham RubyConf (goruco.com), Golden Gate RubyConf (gogaruco.com) и MountainWest RubyConf (mtnwestrubyconf.org), но это далеко не все.

22.6. Локальные группы пользователей Ruby

В США и по всему миру возникают многочисленные локальные группы пользователей. Обычно они называются как-то вроде *Городrb*, что напоминает имя файла, содержащего Ruby-программу (кстати, .rb означает еще и Ruby Brigade). Некоторые группы неформальны, другие – более структурированы. Крупные, активные группы есть в Сиэттле (штат Вашингтон), Остине (штат Техас), Асакуса (Токио), Амстердаме (Нидерланды) и в Нью-Йорке. Многие группы организуют встречи на сайте meetup.com, загляните на него или просто поищите в Интернет ближайшую активную группу поблизости от вас.

22.7. Заключение

Вы дошли до конца этого объемистого тома. В противовес моим ожиданиям, читатели говорили, что прочли первое издание от корки до корки, а некоторые даже выучили Ruby по этой книге (хотя я уделил очень мало времени изложению основ).

Мне не важно, читали вы книгу последовательно или случайно наткнулись на этот абзац. Как бы то ни было, вы дошли до конца, с чем вас и поздравляю.

Но программирование осваивают не по книгам, а применяя полученные из книг знания на практике. Поэтому я призываю вас заняться тем, чем и положено заниматься программисту: идите и пишите программы. Только так вы приобретете настоящий опыт. А если что-то не получается и в книгах нет ответа, обратитесь к сообществу. Наверняка найдутся люди, которые сумеют вам помочь (и люди, которым поможете вы).

Кто входит в «сообщество пользователей Ruby»? Ну, раз вы читаете этот текст, то и вы в том числе. От имени прочих членов приветствую вас и призываю принять активное участие в жизни сообщества.

Предметный указатель

Символы

\', 92
$defout, поток ввода-вывода, 302
$MATCH, переменная, подстановка строк, 103
$stdout, переменная, 301
$& переменная, подстановка строк, 103
__END__, директива, 317
/m (модификатор многострочности), регулярные выражения, 87
%Q, конструкция, строки, 92
%U, спецификатор, определения дня недели, 219
%W, спецификатор, определения дня недели, 219
> (больше)
  >> оператор сдвига вправо, 186
| (вертикальная черта)
  конвейер, 495
  ||, оператор, 76
? (вопросительный знак)
  в идентификаторах, 74
  ?>, конструкция, регулярные выражения, 137
! (восклицательный знак), в идентификаторах, 74
- (дефис), классы символов, 134
= (знак равенства)
  =end, маркер, встроенная документация, 47
  === (оператор отношения), 78, 87, 347
  == (оператор сравнения на равенство), 78, 97, 349
^, каре
  классы символов, 134
  оператор исключающего или, поразрядные операции над числами, 185
  строковые параметры, 113
[], квадратные скобки
  математические множества, создание, 274
  строки, 92
*, квантор, 127
( ) (круглые скобки)
  методы, 88
  строки, 92
< (меньше)
  << оператор сдвига влево, 186
  <=> оператор сравнения, 88, 97
` (обратная кавычка), 48, 479
&, оператор
  математические множества, 274
  поразрядные операции над числами, 185
&&, оператор, 76
**, оператор возведения в степень, 168
<<, оператор дописывания в конец строки, 106
+, оператор над множествами, 274
_ (подчерк), числа, 167
_ (подчерка), идентификаторы, 46, 74
@@, префикс (переменные), 76
% (процент), метод
  класс String, 99
  преобразование системы счисления, 187
  форматирование чисел, 173
(решетка)
  комментарии, 47
  методы экземпляра, 81
  #{}, синтаксическая конструкция, строки, 108
  строки, 74
< >, угловые скобки, 92
{} (фигурные скобки)
  блоки, 83
  строки, 92

A

abort_on_exception, флаг, потоки, 457
accessor_string, вспомогательная функция уровня класса, 366
ACL, список контроля доступа, 616
ActiveRecord, библиотека, 599
add, метод, математические множества, 275

alias_method, метод, 66, 399
alive?, метод экземпляра, потоки, 453
allocate, метод, неинициализированные объекты, 354
all?, квантор, 264
any?, квантор, 264
append_features, метод, 355
ARGF, глобальная константа, 301, 484
ArgumentError, исключение, 58
Array#each, метод, измерение производительности, 541
ASCII, кодировка, интернационализация, 146
asctime, метод, форматирование даты и времени, 224
atan2, функция, 191
atime, метод, временные метки файла, 305
Atom, формат новостных каналов, 507
attr, метод, создание атрибутов экземпляра, 338

B

base64, кодирование и декодирование, 118
basename, метод, путевые имена файлов, 311
begin-end, блоки
  ensure, 59
  исключения, 59
begin, ключевое слово, 75
Benchmark.bm, метод, измерение производительности, 542
Benchmark.measure, метод, измерение производительности, 541
BigDecimal, класс, 174
Bignum, 173
binmode, метод, двоичные файлы, 300
blksize, метод, размер файла, 306
blockdev?, метод, 307
blocks, метод, 306
break, ключевое слово, циклы, 56
break, предложение, 79
Bundler, управление зависимостями, 548
Byebug, отладочная библиотека, 534

C

calculate, метод, 69
caller, метод, 397
capitalize, метод, класс String, 100
casecmp, метод, класс String, 100
case, предложение, 53, 77
  break, 79
  ветви, 79
  ключевое слово then, 74
  оператор отношения (===), 78
center, метод, форматирование строк, 99
chardev?, метод, 307
children, метод, 312
chmod, метод, 303
chomp!, метод, 51
chop, метод, 107
chroot, метод, 319
chruby, программа управления версиями, 639
classify, метод, множества, 275
class_variables, метод, 387
clone, метод
  глубокое копирование, 322
  копирование объектов, 352
close, метод, временные файлы, 318
CoffeeScript и JavaScript, 607
column_size, метод, матрицы, 179
Comparable, модуль, сравнение дат, 221
compare_file, метод, 313
Complex, класс, 180
constants, метод, получение списка констант, 387
const_get, метод
  динамическое создание экземпляра класса, 384
  получение значений констант класса или модуля, 383
const_missing, метод, 391
copy, метод, 313
count, метод, строки, 113
covector, метод, 179
crc32, метод, строки, 116
crypt, метод, шифрование строк, 112
CSS (каскадные таблицы стилей), 606
CSV, формат данных, 326
ctime, метод
  временные метки файлов, 305
  форматирование даты и времени, 224
Cucumber, средство тестирования, 532
current, метод, идентификация потока, 452
cweek, акцессор, определение недели в году, 219

D

DateTime, класс, преобразования, 223
Date, класс
  cweek, акцессор, 219
  контроль даты и времени, 218

методы
Date.leap?, 220
gregorian_leap?, 220
julian_leap?, 220
преобразование в DateTime, 223
deep_copy, метод, 353
def_delegator, метод, вызовы методов, 377
defined?, ключевое слово, 70
define_method, добавление метода в класс/объект, 384
Deflate, класс, сжатие строк, 113
delete, метод
удаление символов из строки, 114
файлы, 313
deq, метод, потоки, 465
dev, метод, получение низкоуровневой информации о файле, 308
dhms2sec, функция, прибавление интервала к моменту времени, 222
Dir.chdir, метод, каталоги файлов, 319
Dir.entries, метод, получение содержимого каталога, 319
Dir.getwd, метод, каталоги файлов, 318
dirname, метод, путевые имена файлов, 311
Dir.pwd, метод, каталоги файлов, 318
divide, метод, множества, 276
do-end, блоки, 67
downcase, метод, класс String, 99
do, ключевое слово, 74
drb. *См.* Распределенный Ruby
dump, метод, 115
dup, метод, 322, 352

E

each_byte, итератор, строки, 95
each, итератор
конструкция for, 82
строки, 94
циклы, 57
each, метод
обход каталога, 319
чтение файла, 315
else, ветвь предложения, 53, 54, 58
emacs, 637
empty?, метод, потоки, 465
end, ключевое слово, 75
enq, метод, потоки, 465
ensure, часть блока begin-end, 59
equal?, метод, сравнение объектов на равенство, 348
err, объект-исключение, 58
Etc, модуль, 497
eval, функция, 382
executable_real?, метод, 304
exec, метод, запуск внешней программы, 479
exist?, метод, проверка существования файла, 305
exit, метод, потоки, 452
exit!, метод, управление процессами, 482
extend, метод
добавление функций из модуля, 62
подмешивание модуля к объекту, 367

F

fail, синоним, 82
FalseClass, 75
false, значение, 75, 76
fcntl, метод, 309
File
Stat, объект
blksize, метод, 306
blocks, метод, 306
dev, метод, 308
executable_real?, метод, 304
gid, метод, 303
grpowned?, метод, 304
ino, метод, 308
mode, метод, 303
owned?, метод, 304
rdev, метод, 308
readable_real?, метод, 304
uid, метод, 303
writable_real?, метод, 304
права владения файлом, 303
File::Separator, объект, путевое имя файла, 312
fileno, метод, 310
FileTest, модуль
blockdev?, метод, 307
chardev?, метод, 307
exist?, метод, 305
size?, метод, 306
symlink?, метод, 307
zero?, метод, 306
FileUtils, библиотека, makedirs, метод, 320
Filter, класс, перенаправление ввода-вывода, 487
Fixnum, 76, 167, 173
Float, метод, преобразование строки в число, 110
flock, метод, 300
foreach, метод, обход каталога, 319

fork, метод, манипулирование процессами, 480
forwardable, библиотека, 377
for, синтаксическая конструкция, 82
ftype, метод, 307

G

getmyvar, метод, 63
Gherkin, нотационный язык, 532
gid, метод, права владения файлом , 303
Golden Gate RubyConf, 642
Gotham RubyConf, 642
gregorian_leap?, метод, 220
grpowned?, метод, 304
gsub, метод, подстановка строк, 103

H

hook_method, метод, 399

I

if, модификатор, 51
if, предложение, 53
  elsif, ключевое слово, 74
  then, ключевое слово, 74
IMAP, протокол, 578
included_modules, метод, получение списка включенных модулей, 387
include?, метод, поиск в строке, 103
index, метод, поиск в строке, 103
Inflate, класс, сжатие строк, 113
initialize_copy, метод, 353
initialize, метод, 83, 339
inner_product, метод, 180
ino, метод, получени индексного узла файла, 308
insert, метод, двоичные деревья, 283
inspect, метод, преобразование к виду для печати, 359
install, метод, 314
instance_methods, метод, 388
instance_of?, метод, 346
Integer, метод, преобразование строки в число, 110
integrate, метод, 189
intersection, метод, 274
ioctl, метод, 309
IP-адрес, сопоставление с регулярным выражением, 139
irb, интерактивный Ruby, 630
  завершение по клавише Tab, 631
IRC-каналы, 641
is_a?, метод, 347

J

Jekyll, генератор статических сайтов, 613
join, метод
  потоки, 456
  путевые имена файлов, 312
JRuby
  многопоточность, 449
  установка, 404
JSON (JavaScript Object Notation)
  библиотеки, 502
  использование в API, 610
  разбор, 500
  сохранение данных, 325
julian_leap?, метод, 220

K

kill, метод
  манипулирование процессами, 482
  потоки, 452
kind_of?, метод, 71, 347

L

length, метод, строки, 94
list, метод, потоки, 458
ljust, метод, форматирование строк, 99
lstrip, метод, строки, 108

M

MacRuby, проект, 447
Madison+Ruby, конференция, 642
main, метод, потоки, 452
makedirs, метод, создание цепочки каталогов, 320
Marshal, модуль, 321, 353
  clone, метод, глубокое копирование, 322
  _dump, метод, нестандартный маршалинг, 322
  dup, метод, 322
  _load, метод, нестандартный маршалинг, 322
MatchData, объект, 131
mathn, библиотека, 181
  простые числа, 182
  разложение на простые множители, 181

Math.sqrt, 187
Math, модуль, управление пространством имен, 62
method_added, метод, 399
method_missing, метод, 392
Method, класс, 372
Middleman, генератор статических сайтов, 612
MIME, формат почтового сообщения, 580
Minitest, средство тестирования, 527
mktime, метод, 213
MySQL, интерфейс с, 329
myvar=, метод, 63
m, модификатор многострочности, 136

N

Nanoc, генератор статических сайтов, 613
nil, определение операторов, 82
Nokogiri, разбор XML и HTML, 503

O

ObjectSpace, модуль, 72
Octopress, генератор статических сайтов, 613
Onigmo, движок регулярных выражений, 122
Open-URI, библиотека, 587

P

Parley, форум, 641
Pathname, класс, 312
pid, метод, 482
popdir, метод, 489
POP, протокол, 573
PostgreSQL, база данных, 331
ppid, метод, 482
Prawn и PDF-документы, 518
  граничный прямоугольник, 519
  единицы измерения, 519
  курсор, 519
  ограничивающий прямоугольник, 519
  основные концепции и приемы, 519
  страничные координаты, 519
priority, метод, потоки, 454
Proc, объект, 371
pry, утилита, 634
Puma, веб-сервер, 593
pushdir, метод, 489

Q

QtRuby, 435
  TimerClock, виджет, 441
  переключатели, 441
  простое оконное приложение, 435
  текстовые поля, 438
  флажки, 441

R

Rack, библиотека, 591
Rails, каркас, 594
  ERB и HTML, 602
  Haml, 604
  JavaScript и CoffeeScript, 607
  Sass и CSS, 606
  конвейер активов, 605
  маршрутизация, 595
  параметры, 598
raise, метод, потоки, 455
rake, утилита
  действия, 627
  задания, 627
Rational, класс, 175
RDoc, простая разметка, 554
RDoc, документирование кода, 552
readpartial, метод, 311
Redis, хранилище данных, 333
redo, ключевое слово, циклы, 57
rescue, ключевое слов, исключения, 58
REST, архитектурный стиль, 610
retry, ключевое слово, исключения, 59
reverse, метод, строки, 114
Rinda::RingFinger, класс, 625
Rinda::RingProvider, класс, 624
Rinda::RingServer, класс, 624
Rinda, пространства кортежей, 620
ri (Ruby index), утилита, 635
rjust, метод, форматирование строк, 99
RMagick, библиотека
  API рисования, 515
  обработка изображений
    масштабирование, 512
    получение информации, 510
    специальные эффекты и трансформации, 513
RSpec, средство тестирования, 525
RSS, формат новостных каналов, 507
rstrip, метод, строки, 108
Rubygems
  зависимости, 547

создание gem-пакетов, 547
установка gem-пакетов, 547
Ruby/GTK3, библиотека, 423
Ruby-GNOME2, 434
кнопки, 425
меню, 432
переключатели, 432
простое оконное приложение, 424
текстовые поля, 427
флажки, 429
RubyMine, графический текстовый редактор, 636
Ruby/Tk, 410
кнопки, 413
переключатели, 421
простое оконное приложение, 411
списки, 422
текстовые поля, 416
флажки, 420
холст, 423
rvm (Ruby Version Manager), 637

S

safe_level, метод, 454
safe_unlink, метод, 314
SCSS и Sass, 606
select, метод, 310
self, псевдопеременная, 64
setmyvar, метод, 63
Shell, библиотека
Filter, класс, 487
popdir, метод, 489
pushdir, метод, 489
transact, метод, 488
перенаправление ввода-вывода, 487
Shoes 4, 404
ввод текста, 405
генератор четверостиший, 408
графика, 408
кнопки, 405
компоновка, 407
сетевые ресурсы, 410
события, 409
установка, 404
SimpleDelegator, класс, 376
SingleForwardable, класс, 378
singleton_method_added, метод, 400
singleton_methods, метод, 388
SizedQueue, класс, 464, 465
SMTP, протокол, 575
start, метод, потоки, 452
STDERR, константа, 301
STDIN, константа, 301
STDOUT, константа, 301
stop, метод, потоки, 452
StringIO, библиотека, 316
StringScanner, класс, 98
Struct, класс, 360
super, ключевое слово, 66
swapcase, метод, класс String, 100
sync.rb, библиотека, потоки, 470
syscall, метод, 310

T

Tempfile, библиотека, 318
then, ключевое слово, 74
Time, класс
Comparable, модуль, 221
методы
mktime, 213
strftime, 214
Time.at, 217
Time.gm, 214
Time.utc, 214
to_a, 214
wday, 214
определение високосности года, 220
часовые пояса, 220
to_ary, метод, 359
to_a, метод, дата и время, 214
to_f, метод, преобразование строки в число, 110
to_int, метод, неявные преобразования, 184
to_i, метод, преобразование строки в число, 110
to_set, метод, 274
to_str, метод, 104
to_s, метод
преобразование системы счисления, 187
преобразования строк, 104
transact, метод, 488
trap, метод, 483
TrueClass, 75
truncate, метод, 313

U

UCS-2, кодировка, 149
uid, метод, права владения файлом , 303
UnboundMethod, объект, 373
undef_method, метод, удаление определений, 389

Unicode
  байт, определение, 148
  глиф, определение, 148
  графема, определение, 148
  кодовая позиция, определение, 148
  семейство кодировок, 149
  строки
    нормализация, 152
    упорядочение, 156
  терминология, 148
union, метод, множества, 274
unpack, метод
  двоичные файлы, 300
upcase, метод, класс String, 99
upto, метод, генерирование последовательности строк, 115
UTF-16, кодировка, 149

V

vim, редактор, 636

X

XML, разбор, 502
X Window System, 446

Y

YAML (YAML Ain't Markup Language), 158, 323
YARD (Yay! A Ruby Documentation), 556
yield, ключевое слово, 81

Z

ZenTest. *См.* Тестирование, ZenTest
Zlib, библиотека, 113

А

Абстрактные классы, 44, 66
Акцессоры, 87
  cweek, определение недели в году, 219
  zone, определение часового пояса, 220
Аргументы
  ARGF, 484
  ARGV, 484
  метода, 67
Аспектно-ориентированное программирование (AOP), 380
Атрибуты
  акцессоры, 87
  и локальные переменные, 77
  класса, 41
  методы установки, 87
  методы чтения, 87
  определение, 40
  экземпляра, создание, 338

Б

Базы данных, внешние соединения
  MySQL, 329
  ORM, 332
Байты
  интерпретация, 147
  порядок следования, 188
Библиотеки
  ActiveRecord, 599
  Byebug, 534
  Haml, 604
  ImageMagick, 510
  Mustache, 605
  Net
    FTP, 559
    HTTP, 609
  Oj, 501
  Open-URI, 587
  OptionParser, 485
  pp, форматирование печати, 542
  Rack, 591
  RSS, 509
  Ruby/GTK3, 434
  Shell, 488
  yajl-ruby, 502
  для работы с JSON, 502
  для работы с графами, 293
Блоки
  do-end, 67
  Kernel#trace_var, метод, 402
  в фигурных скобках, 67, 83
  метод initialize, 339
  передача итератору, 83, 87
  передача методам, 67
  синтаксис, 74
Булевы типы, 75

В

Ввод-вывод
  $stdout, переменная, 301
  ARGF, переменная, 301
  fcntl, метод, 309
  fileno, метод, 310

ioctl, метод, 309
readpartial, метод, 311
select, метод, 310
STDERR, константа, 301
STDIN, константа, 301
STDOUT, константа, 301
syscall, метод, 310
буферизованный и небуферизованный, 302
каналы, 308
каталоги. *См.* Каталоги
перенаправление, 487
программы
чтение встроенных данных, 317
чтение исходного текста, 317
файлы. *См.* Файлы
Ввод символов с клавиатуры, 314
Ветвящегося равенства оператор (===), 53, 347
Взаимоблокировка, 449
Високосные секунды, 217
Високосный год, проверка на, 220
Висячие указатели, 72
Вложения, кодирование и декодирование, 579
Внешние программы, запуск
манипулирование процессами, 480
метод exec, 479
метод system, 479
перехват выводимой информации, 479
стандартный ввод-вывод, 483
Возведение в степень, 168
Волокна, 475
Временные метки, 304
Временные файлы, 318
Встроенная документация, 74
Встроенные данные, чтение, 317
Встроенные документы, 93
Выражения
включение в строку, 108
интерполяция, 47
ориентация на, 85
синтаксические ошибки, 73

Г

Глиф
определение, 148
составные и монолитные формы, 152
Глобальные константы, 301
Глобальные методы, синонимы, 80
Глобальные переменные, 46
$stdout, 301
примеры, 47
синонимы, 80
Глубокое копирование, 322, 352
Град, единица измерения углов, 190
Графема, определение, 148
Графический интерфейс пользователя
QtRuby. *См.* QtRuby
Ruby/GTK3. *См.* Ruby/GTK3
Ruby/Tk. *См.* Ruby/Tk
Shoes 4. *См.* Shoes 4
Swing, 444
Графы
матрица смежности, 288
связность, 290
эйлеров путь, 293
эйлеров цикл, 292

Д

Дата и время
mktime, метод, 213
strftime, метод, 214
Time.gm, метод, 214
Time.utc, метод, 214
to_a, метод, 214
wday, метод экземпляра, 214
високосные секунды, 217
всеобщее скоординированное время (UTC), 212, 225
вывод на печать, 224
вычисление n-го дня недели в месяце, 215
вычисление интервала, 222
извлечение из строки, 223
контроль, 218
летнее время, 212
определение даты Пасхи, 215
определение дня недели, 214
определение недели в году, 219
определение порядкового номера дня в году, 218
определение текущего времени, 213
определение числа дней в месяце, 225
преобразование к точке отсчета, 217
преобразования, 223
прибавление интервала, 221
разбиение месяца на недели, 226
секунд в дни, часы, минуты, 216
сопоставление с помощью регулярных выражений, 141
сравнение, 221
среднее время по Гринвичу, 212, 225

точка отсчета, 213, 223
форматирование, 224
часовые пояса
определение, 220
преобразование, 225
юлианская дата, 218
Двоичные деревья
реализация, 282
сортировка, 284
справочные таблицы, 286
Действия, утилита rake, 627
Делегирование
вызовы методов, 376
и наследование, 379
Деревья, 282
двоичные
реализация, 282
сортировка, 284
справочные таблицы, 286
преобразование в массив, 286
преобразование в строку, 286
Деструкторы, 40
Динамическая типизация, 88
Динамические средства
const_get, метод, 383
const_missing, метод, 391
define_method, метод, 384
динамическая интерпретация кода, 382
классы
отслеживание изменений, 398
получение списка закрытых методов, 388
получение списка защищенных методов, 388
получение списка модулей, 387
получение списка открытых методов, 388
получение списка переменных, 387
получение списка предков, 387
константы, получение списка, 387
методы
const_missing, 391
вызовы несуществующих, 392
добавление в класс или объект, 384
модули
отслеживание добавлений, 399
получение списка, 387
объекты, чистильщики, 395
определения, удаление, 389
просмотр стека вызовов, 397
функция eval, 382
Дисперсия (наборы данных), 193
Документация
PDF и Prawn, 518
RDoc, 552
Дубликаты, удаление из строки, 114
Дуговой градус, 190

З

Зависимости
Bundler, управление зависимостями
загрузка из Git, 550
семантическое версионирование, 550
создание gem-пакетов, 551
Gemfile, 549
Gemfile.lock, 549
частные gem-пакеты, 551
Заглядывание вперед, регулярные выражения, 128
Замораживание объектов, 360
Замыкание, 84

И

Извещения об ошибках, 641
Изменяемые объекты, 61
Именованные соответствия, 133
Индексная переменная, модификация, 77
Инкапсуляция, определение, 40
Интернационализация, Unicode, кодировки, 149
Интернационализация, 145
UCS-2, кодировка, 149
Unicode
UTF-8, кодировка, 149
терминология, 148
UTF-8, кодировка.
См. Интернационализация, строки
UTF-16, кодировка, 149
строки
интерпретация байтов, 147
нормализация, 152
упорядочение, 156
Интерполяция, 47
Исключения
блоки begin-end, 58
ветвь else, 58
ключевое слово retry, 59
переменная err, 58
предложение raise, 57
предложение rescue, 58
Итераторы
each
конструкция for, 82

строки, 94
each_byte, строки, 95
retry, 83
блоки, 83, 87
вызовы методов, 83
циклы, 57

К

Канальный уровень, сетевое программирование, 558
Каталоги
изменение корня, 319
изменение текущего, 318
обход, 319
получение содержимого, 319
рекурсивное удаление, 320
цепочка, 319
Кванторы, регулярные выражения, 126
*, 127
Классы
Complex, 180
Date
акцессор cweek, 219
контроль даты и времени, 218
преобразование в DateTime, 223
Deflate, сжатие строк, 113
DelegateClass, метод, 376
FalseClass, 75
File
atime, метод, 305
basename, метод, 311
binmode, метод, 300
chmod, метод, 303
chown, метод, 303
ctime, метод, 305
delete, метод, 313
dirname, метод, 311
flock, метод, 300
join, метод, 312
link, метод, 313
mtime, метод, 305
new, метод, 296
open, метод, 297
path, метод, 312
putc, метод, 301
rename, метод, 313
rewind, метод, 299
seek, метод, 298
split, метод, 311
symlink, метод, 313
tell, метод, 299
truncate, метод, 313
umask, метод, 304
utime, метод, 305
Inflate, сжатие строк, 113
Module, 64, 71
Pathname, 312
Rational, 175
self, псевдопеременная, 64
SimpleDelegator, 376
SingleForwardable, 378
SizedQueue, потоки, 464, 465
String. *См.* String, класс
StringScanner, 98
Struct, 360
Time. *См.* Time, класс
TrueClass, 75
абстрактные, 44, 66
вложенные, 368
закрытые методы, 65
защищенные методы, 65
имена, 62
конкретный тип, 44
конструкторы, 395
методы
добавление, 384
закрытые, 344
объявление, 341
отслеживание добавлений, 399
получение списка, 388
создание, 341
модули, 44
включение, 373
получение списка, 387
наследование, 41, 66
объекты, 82
опрос типа, 346
создание, 60
определение, 40, 62
открытые методы, 65
отслеживание изменений в определении, 398
параметрические, создание, 368
переменные, 75
получение списка, 387
префикс @@, 76
создание, 342
получение значения константы, 383
производные, 42
синглетные, 89
содержащие только данные, 359
список предков, 387
структурные шаблоны, 360

суперклассы, наследование, 344
является, отношение, 42
Классы символов, регулярные выражения, 134
Ключевые слова
begin, 75
break, 56
defined?, 70
do, 74
elsif, 74
end, 75
next, 57
redo, 57
then, 74
как имена методов, 74
список, 46
Кнопки
Shoes 4, 405
Swing, 445
Кнопки (графический интерфейс)
QtRuby, 436
Ruby/GTK3, 425
Ruby/Tk, 413
Кодировки (Unicode), 149
Кодовая позиция
определение, 148
составные и монолитные формы, 152
Командная строка, аргументы, ARGV[0], 76
Комментарии, 47
Комплексные числа, 180
конвейер активов
JavaScript и CoffeeScript, 607
Sass и CSS, 606
Конкретные типы, 44
Константы, 46
изменение значения, 81
отсутствующие, 391
получение списка, 387
примеры, 47
удаление, 390
числовые, 47
Конструкторы, 40, 395
Конференции по Ruby, 642
Концевой маркер, встроенные документы, 93
Кооперативная многозадачность, 475
Копирование
глубокое, 322, 352
дерева каталогов, 491
объектов, 352
поверхностное, 352
файлов, 313
Коэффициент корреляции, 193

Л

Левенштейна расстояние, строки, 117
Летнее время, 212
Логарифмы, 187, 191
Локализация, 145
Локальные переменные, 46
и атрибуты объекта, 77
примеры, 47

М

Маршалинг
YAML, 323
глубокое копирование, 322
нестандартный, 322
простой, 321
Массивы. *См. также* Хэши
ассоциативные. *См.* Хэши
битовые, 186
деревья и, 286
и переменные окружения, 490
метод pack, двоичные файлы, 300
параметров, сворачивание, 67
преобразование
объекта в, 359
строки в, 359
примеры, 48
строк, 48
разложение строки в, 95
чтение файла в, 315
Матрица смежности, 288
Матрицы, 176, 288
Медиана, наборы данных, 192
Меню, создание, Ruby/GTK3, 432
Методы
accessor_string, 366
add, множества, форматирование даты и времени, 275
alias_method, 66, 399
alive?, потоки, 453
allocate, неинициализированные объекты, 354
append_features, 355
asctime, форматирование даты и времени, 224
atime, 305
attr, создание атрибутов экземпляра, 338
basename, 311
binmode, двоичные файлы, 300

blksize, 306
blockdev?, 307
blocks, 306
caller, определение вызвавшего метода, 397
capitalize, класс String, 100
casecmp, класс String, 100
center, форматирование строк, 99
chardev?, 307
children, 312
chmod, 303
chomp, строки, 107
chop, строки, 107
chroot, 319
classify, множества, 275
class_variables , 387
clone, копирование объектов, 352
close, временные файлы, 318
column_size, матрицы, 179
compare_file, 313
constants, 387
const_get, 383
const_missing, 391
copy, 313
count, строки, 113
covector, 179
crc32, строки, 116
crypt, шифрование строк, 112
ctime, форматирование даты
и времени, 224
ctime, временные метки файла, 305
current, идентификация потока, 452
Date.leap?, 220
deep_copy, 353
def_delegator, 377
define_method, добавление метода
в класс/объект, 384
DelegateClass, 376
delete
  удаление символов из строки, 114
  файлы, 313
deq, потоки, 465
dev, низкоуровневая информация
о файле, 308
Dir.chdir, 319
Dir.entries, 319
Dir.getws, 318
dirname, 311
Dir.pwd, 318
divide, множества, 276
downcase, класс String, 99
dump, печать специальных символов, 115
dup, копирование объектов, 352
each
  обход каталога, 319
  чтение файла, 315
empty?, потоки, 465
enq, потоки, 465
equal?, сравнение объектов
на равенство, 348
exist, проверка существования файла, 305
exit, потоки, 452
extend, 62, 367
fcntl, 309
fileno, 310
Float, преобразование строки в число, 110
flock, 300
foreach, обход каталога, 319
ftype, 307
getmyvar, 63
gregorian_leap?, 220
gsub, подстановка в строку, 103
hook_method, 399
included_modules, 387
include, поиск в строке, 103
index, поиск в строке, 103
initialize, 83, 339
initialize_copy, 353
inner_product, 180
ino, 308
insert, двоичные деревья, 283
inspect, 359
install, 314
instance_methods, 388
instance_of?, 346
Integer, преобразование строки
в число, 110
integrate, 189
intersection, множества, 274
IO
  построчное чтение из файла, 315
  чтение файла в память, 315
ioctl, 309
IO.popen, 308
is_a?, 347
join
  потоки, 456
  путевые имена файлов, 312
julian_leap?, 220
Kernel#trace_var, 402
kill, потоки, 452
kind_of?, 347
length, строки, 94
link, 313
list, потоки, 458

ljust, форматирование строк, 99
load, 62
lstrip, 108
main, главный поток программы, 452
makedirs, 320
method_added, 399
method_missing, 392
mktime, 213
Module#ancestors, 387
move, 314
mtime, 305
myvar=, 63
num_waiting, 465
ObjectSpace.each_object, 396
offset, 132
open, 297
pack, двоичные файлы, 300
polar, 180
pop, потоки, 465
prime_division, 181
printf, 187
private, 65, 350
private_class_method, 344
private_instance_methods, 388
private_methods, 388
protected, 65, 351
protected_instance_methods, 388
protected_methods, 388
public, 65, 352
public_instance_methods, 388
public_methods, 388
push, потоки, 465
putc, 301
puts, 105
raise, потоки, 455
rdev, 308
readpartial, 311
Regexp.compile, 124
Regexp.escape, 125
Regexp.last_match, 103, 131
remove_const, 390
remove_method, 389
rename, 313
require, 62
respond_to?, 71, 347
reverse, строки, 114
rewind, прямой доступ к файлу, 299
rindex, поиск в строке, 103
rjust, форматирование строк, 99
rmtree, рекурсивное удаление каталога, 320
root?, 312
rot13, кодирование строк, 111
row_size, vfnhbws, 179
rstrip, 108
run, возобновление потока, 454
safe_level, 454
safe_unlink, 314
scanf, 111
scan, сопоставление с регулярным выражением, 98, 104
seek, прямой доступ к файлу, 298
select, 310
setmyvar, 63
shift, потоки, 465
similar?, расстояние Левенштейна, 118
singleton_method_added, 400
singleton_methods, 388
size?, 306
split
  путевые имена файлов, 311
  разбиение строки на лексемы, 97
sprintf
  преобразование системы счисления, 187
  форматирование строк, 98
squeeze, удаление дубликатов, 114
start, потоки, 452
status, потоки, 453
stop, потоки, 452
stop?, потоки, 453
strftime
  определение дня недели, 214
  определение недели в году, 219
  печать часов, минут и секунд, 221
  преобразование часовых поясов, 225
  форматирование даты и времени, 224
strip, 107
sub, подстановка строк, 102
succ, генерирование последовательности строк, 115
swapcase, класс String, 100
symlink, 313
sync, буферизация ввода-вывода, 302
syscall, 310
ThreadGroup.new, создание группы потоков, 458
Time.at, 217
Time.gm, 214
Time.local, 213
Time.utc, 214
to_ary, 359
to_f, преобразование строки в число , 110
to_int, неявные преобразования, 184
to_i, преобразование строки в число, 110

to_s, 104, 187
to_set, 274
to_str, 104, 358
truncate, 313
tty?, 307
type, 71
umask, 304
undef_method, 389
ungetc, 303
union, множества, 274
unpack, двоичные файлы, 300
upto, генерирование последовательности строк, 115
utime, 305
value, потоки, 456
wakeup, возобновление потока, 454
zero, матрицы, 178
Zlib, методы сжатия, 113
вызовы
  делегирование, 376
  перенаправление, 376
  скобки, 73
глобальные, синонимы, 80
как операторы, 76
класса
  закрытые, 344
  объявление, 341
  создание, 341
определение, 40
отслеживание добавлений, 399
отсутствующие, 72
передача блока, 67
переопределение, 42, 66
синглетные, 68, 89
синонимы, создание, 66
сцепление, 67
управление доступом, 349
установки, 82, 337
хэш, передача, 73
чтения, 337
экземпляра, 62
  executable_real?, 304
  gid, 303
  grpowned?, 304
  mode, 303
  owned?, 304
  readable_real?, 304
  stat, 305
  to_a, 214
  uid, 303
  wday, 214
  writable_real?, 304
  zero?, 306
Множества математические
  более сложные операции, 275
  перечисляемые объекты, преобразование, 274
  простые операции, 274
  создание, 274
Мода, наборы данных, 192
Модификаторы, регулярные выражение, 123
Модули, 44
  append_features, метод, 355
  Math, управление пространством имен, 62
  включенные в класс, получение списка, 387
  вложенные, 368
  добавление функций, 62
  методы экземпляра, 62
  определение, 61
  отслеживание добавления методов, 399
  подмешивание к объектам, 367
  получение списка констант, 383
  примеси, 355
Мониторинг выполнения программы, 401
Мультимодальные наборы данных, 192

Н

Наборы данных
  дисперсия, 193
  медиана, 192
  мода, 192
  среднее, 191
Наследование, 41
  и делегирование, 379
  множественное, 42
  отношение \, 345
  полиморфизм, 43
  ромбовидное, 42
  символ <, 83
Незапоминаемые группы, 131
Неявные преобразования
  строк, 104
  чисел, 183
Новостные группы (Ruby), 640

О

Обратные ссылки, регулярные выражения, 130
Объектно-реляционные отображения (ORM), 332
ООП (объектно-ориентированное программирование)
  Struct, класс, 360

UnboundMethod, объект, 373
АОП, 380
атрибуты экземпляра, создание, 337
без классов, 380
вложенные классы, 368
замораживание объектов, 360
классы
включение модулей, 373
опрос, 346
содержащие только данные, 359
методы
делегирование, 376
перенаправление, 376
управление доступом, 349
методы класса
закрытые, 344
объявление, 341
создание, 341
методы чтения и установки уровня класса, 379
модули
append_features, метод, 355
включение в классы, 373
методы экземпляра, 355
подмешивание к объекту, 357, 367
примеси, 355
на основе прототипов, 380
наследование, 344
несколько конструкторов, 336
объекты
Proc, 371
копирование, 352
неинициализированные, 354
специализация, 364
параметры, заданные по умолчанию, 376
паттерны проектирования, 381
переменные класса, создание, 342
преобразование
объекта в массив, 359
объекта в представление для печати, 359
объекта в строку, 358
строки в массив, 359
проектирование по контракту, 381
сложные конструкторы, 339
функциональное программирование, 380
экстремальное программирование, 381
Операторы
&&, 76
||, 76
деления по модулю, 75
диапазона, 80
для объекта nil, 82
как методы, 76
отношения, 87
перегрузка, 66, 82
перечень, 49
повтора строк, 108
приоритет, 76
разрешения области видимости, 82
расширения массива, 89
составного присваивания, 81
сравнения, 88, 97
тернарный, 74, 87
Операционная система, определение, 497
Определения
исполняемые, 82
удаление, 389
Отладка
Byebug, отладочная библиотека, 534
Pry, как средство отладки, 537
извещения об ошибках, 641
инспекция объекта, 359
многопоточных программ, 457
Отражение, 70, 387, 388
Очереди, 277, 281

П

Параметры
заданные по умолчанию, 376
массивы, сворачивание, 67
передача потоку, 450
Партизанское латание, 90
Паттерны проектирования, 381
Перевод, локализация, 158
Переключатели (графический интерфейс)
QtRuby, 436
Ruby/GTK3, 432
Ruby/Tk, 421
Переключатели, создание
QtRuby, 436
Ruby/GTK3, 432
Ruby/Tk, 421
Переменные
$MATCH, подстановка строк, 103
$&, подстановка строк, 103
@@, префикс, 76
err, исключения, 58
глобальные, 46
$stdout, 301
примеры, 47
синонимы, 80
интерполяция, 47
класса, 46

получение списка, 387
создание, 342
локальные, 46
и атрибуты объекта, 77
и имена методов, 77
примеры, 47
массивы, 48
обмен значений, 81
объявление, 75
окружения, 489
хранение в виде массива или хэша, 490
чтение и установка, 489
потока, 450
присваивание значений, 81
псевдопеременные, 75
self, 64
примеры, 47
условные, потоки, 465
хэши, 49
экземпляра, 46
класса, 85
получение списка, 387
примеры, 47
экземпляра класса, 90, 343
Переменные окружения, 489
хранение в виде массива или хэша, 490
чтение и установка, 489
Переменные экземпляра, экземпляра класса, 343
Перенаправление вызовов методов, 376
Перечислители, 266
подсчет, 269
преобразование в массив или множество, 266
сравнение, 269
Подшаблоны, 602
Потоки, 448
JRuby, 449
Rubinius, 449
взаимоблокировка, 449
вложенные блокировки, 467
возбуждение исключений, 455
возобновление, 454
главный
ожидание других, 456
получение ссылки, 452
группы
возобновление, 459
добавление потока, 458
завершение, 459
получение списка потоков, 458
создание, 458
идентификация, 452
монитор, 467
мьютексы, 462
обработка исключений, 457
опрос уровня безопасности, 454
параллельное рекурсивное удаление, 474
параллельный поиск в коллекции, 473
передача параметров, 450
перечисление, 452
планировщик, передача управления, 454
приоритеты, 454
приостановка, 452
синхронизация, 459
очереди, предопределенные классы, 464
события, ожидание, 472
создание, 450
состояние, 453
таймаут, 470
условные переменные, 465
Поэтический режим, 88
Предки класса, получение списка, 387
Преобразование
объекта в массив, 359
объекта в представление для печати, 359
объекта в строку, 358
строки в массив, 359
Приведение типов, 184
Прикладной уровень (сетевое программирование), 558
Приложения (веб)
HTML
ERB, 602
Haml, библиотека, 604
Liquid, шаблоны, 605
Mustache, шаблоны, 605
макеты, 602
подшаблоны, 602
шаблоны, 602
конвейер активов
JavaScript и CoffeeScript, 607
Sass и CSS, 606
базы данных, 599
хранилища данных, 601
Примеси, 355
определение, 61
Присваивания операторы составные, 81
Проблема ромбовидного наследования, 42
Проектирование по контракту, 381
Производительность, измерение, 539
Benchmark.bm, метод, 542
Benchmark.measure, метод, 541
Производные классы, 42

Пропуски, синтаксические ошибки, 73
Пространства кортежей, 620
Псевдопеременные, 75
  self, 64
  примеры, 47
Путевые имена файлов, 311

P

Разработки средства
  irb, утилита, 630
  Pry, 634
  rake, утилита, 626
  ri, утилита, 635
  менеджеры версий, 637
  редакторы, 635
Распределенный Ruby (drb), 614
  клиент, 615
  обнаружение сервисов, 624
  пространства кортежей (Rinda), 620
  сервер, 615
  списки контроля доступа, 616
Расширения массива оператор, 89
Регулярные выражения
  Onigmo, движок, 122
  внутренние модификаторы, 137
  внутренние подвыражения, 137
  заглядывание вперед, 128
  именованные соответствия, 133
  кванторы, 126
  классы символов, 134
  компиляция, 124
  литеральные, 125
  модификаторы, 123
    многострочности, 87
  обнаружение повторяющихся слов, 142
  обобщенные, 135
  обратные ссылки, 130
  оглядывание назад, 129
  простые примеры, 122
  рекурсивные, 138
  римские цифры, 140
  синтаксис, 122
  соответствия нулевой длины, 87
  сопоставление с IP-адресом, 139
  сопоставление с датой и временем, 141
  сопоставление с номером версии, 143
  сопоставление с номером телефона, 143
  сопоставление со строкой, 98
  сопоставление с парой
  ключ-значение, 140
  сопоставление с почтовым индексом, 143
  сопоставление с символом конца
  строки, 136
  сопоставление с числовыми
  константами, 141
  экранирование специальных
  символов, 125
  якоря, 125
Редакторы emacs, 637

C

Сборка мусора, алгоритм пометки
и удаления, 395
Сетевой уровень, сетевое
программирование, 558
Сжатие строк, 113
Символические ссылки, 307
Синглеты, 365
  Singleton, паттерн, 89
  классы, 89
  методы, 89
  определение, 68
  создание, 83
Синонимы
  fail, 82
  kind_of?, 71
  глобальных методов, 80
  глобальных переменных, 80
  методов, создание, 66
Система счисления, преобразование, 186
Случайные числа, генерирование, 195
Списки рассылки, посвященные Ruby, 640
Сравнение файлов, 313
Сравнения оператор (<=>), 88, 97
Среднее, наборы данных, 191
Стек, 277
  и рекурсия, 279
  обнаружение несбалансированных
  скобок в выражениях, 278
  строгая реализация, 278
Стек вызовов, 397
Строки
  base64-кодирование, 118
  each_byte, итератор, 95
  each, итератор, 94
  %q, конструкция, 92
  StringScanner, класс, 98
  в двойных кавычках, 92
  в одиночных кавычках, 92
  встроенные документы, 93
  вычисление 32-разрядного CRC, 115
  генерирование последовательности, 115

данные, разделенные запятыми, 109
длина, 94
извлечение даты и времени, 223
как объекты IO, 99
как файлы, 316
конкатенация с числом, 358
массивы, 48, 95
нормализация, 152
обращение, 114
оператор <<, 106
парные ограничители, 92
перехват вывода команды, 48
повтор, 108
подстановка, 102
подстроки
вычленение и замена, 100
подсчет числа символов, 113
поиск, 103
преобразования
в массив, 359
в число, 110
объектов в, 358
явные и неявные, 104
разбиение на лексемы, 97
расстояние Левенштейна, 117
регистр, определение, 100
сжатие, 113
символы, преобразование в строку, 61
сопоставление с регулярным выражением, 98
специализированное сравнение, 96
табуляторы, 119
убирание лишних пропусков, 107
удаление хвостовых символов, 106
форматирование, 98, 172
цитирование текста, 120
шифрование
crypt, 112
rot13, 111
Структурные шаблоны, 360
Сцепление вызовов, 67

Т

Текстовые поля
QtRuby, 438
Ruby/GTK3, 427
Ruby/Tk, 416
Текущее время, определение, 213
Тождественная матрица, 177
Точка отсчета, 213
более ранние даты, 223
контроль даты и времени, 218
микросекунды, 217
преобразование, 217
Транслитерация, 155
Транспортный уровень, сетевое программирование, 558
Трема, определение, 152
Тригонометрические функции, 190

У

Удаление
каталогов рекурсивное, 320
констант, 390
определений, 389
файлов, 313, 493
Унимодальные наборы данных, 192
Упорядочение строк, 156
Управляющие конструкции
redo, ключевое слово, 57
until, 54
while, 54
ветвление, 53
циклы
break, ключевое слово, 56
next, ключевое слово, 57
итераторы, 57
модуль Kernel, 56
с предварительной проверкой условия, 54
Условные переменные, потоки, 465
Утверждения, 529

Ф

Файлы
umask, 304
ассоциирован ли с терминалом, 307
блокировка, 300
блочные устройства, 307
временные, 318
временные метки, 304
дописывание в конец, 298
идентификатор пользователя, 304
информация низкого уровня, 308
копирование, 313
обновление, 298
открытие и закрытие, 296
переименование, 313
перемещение, 314
получение индексного узла, 308

получение информации об устройстве, 308
получение размера, 306
посимвольный вывод, 301
построчное чтение, 315
права владения, 303
проверка на пустоту, 306
проверка существования, 305
прямой доступ, 298
путевые имена, 311
разрешения, 303
символические ссылки, 307
сравнение, 313
строки как, 316
считывание в память, 315
удаление, 313, 493
усечение, 313
установка, 314
физические ссылки, 307
Фильтры текстовые, 491
Флажки
QtRuby, 439
Ruby/GTK3, 429
Ruby/Tk, 420
Форматирование
даты и времени, 224
строк, 98
чисел, 172
Форматы данных
Atom, 507
HTML
потоковый разбор, 505
разбор документов, 502
JSON, 500
библиотеки, 502
обход данных, 500
PDF, 518
RMagick
API рисования, 515
масштабирование изображения, 512
получение информации об изображении, 510
специальные эффекты и трансформации, 513
RSS, 507
XML, 502
примеры, 499
Форматы файлов. *См.* Форматы данных
Формы множественного числа, 162
Функции
atan2, 191
dtms2sec, прибавление интервала к моменту времени, 222
eval, динамическая интерпретация кода, 382
sec2dhms, вычисление разности между моментами времени, 222
кэширование, memoize, 195
Функциональное программирование, 380

X

Хэш. *См. также* Массивы
константа, 49
передача методу, 73
переменная, 49
переменные окружения, хранение в виде, 490

Ц

Цепочка каталогов, создание, 319
Циклы. *См.* Управляющие конструкции, циклы

Ч

Числа
BigDecimal, класс, 174
Fixnum, 167
библиотека mathn, 181
простые числа, 182
разложение на простые множители, 181
битовые массивы, 186
большие целые, 173
возведение в степень, 168
вычисление определенного интеграла, 189
деление, 168
извлечение корней, 187
комплексные, 180
конкатенация со строкой, 358
коэффициент корреляции, 193
кэширование результатов вычисления, 195
логарифмы, 187
матрицы, 176
подчерки, 167
поразрядные операции, 185
порядок байтов, 188
представление в Ruby, 167

преобразование в строку, 110, 187
преобразование системы счисления, 186
приведение типов, 184
простые, 182
разложение на простые множители, НОД и НОК, 181
рациональные, 175
римские, 140
случайные, генерирование, 195
с плавающей точкой
округление, 169
создание, 168
сравнение, 171
тригонометрические функции, 190
форматирование
вставка разделителей, 173
для вывода, 172
явные и неявные преобразования, 183
Чистильщики, 395

Ш

Шифрование строк
crypt, 112
rot13, 111

Э

Эйлеров путь, графы, 293
Эйлеров цикл, графы, 292
Электронная почта
вложения, кодирование и декодирование, 579
отправка по протоколу SMTP, 575
шлюз в конференции, 581

Я

Явные преобразования, строки, 104
Якоря, 125