Фишерман Л. В.



Git

ПРАКТИЧЕСКОЕ РУКОВОДСТВО

Управление и контроль версий в разработке программного обеспечения

НиТ
издательство

Фишерман Л. В.

Git

Практическое руководство

Управление и контроль версий в разработке программного обеспечения

"Наука и Техника"

Санкт-Петербург

УДК 004.42
ББК 32.973

Фишерман Л. В.

Git. Практическое руководство. Управление и контроль версий в разработке программного обеспечения — СПб.: Наука и Техника, 2021. — 304 с., ил.

ISBN 978-5-94387-547-2

Git в настоящее время нужен практически всем программистам, которые занимаются разработкой программного обеспечения. Git — это система управления версиями, с помощью которой вы сможете вести и контролировать разработку нескольких версий одного приложения, осуществлять совместную разработку одного приложения несколькими разработчиками (учитывать изменения, которые делаются на том или ином шаге разработки тем или иным разработчиком). С помощью системы Git у вас будет полная иерархия всех версий программного кода разрабатываемого приложения.

Данная книга представляет собой подробное практическое руководство по Git, в котором описывается Git и приводится разбор конкретных ситуаций и применений, например, как изменения из одной ветки разработки включить в другую ветку, но не все. Изложение начинается с самых азов, никакой предварительной подготовки не требуется: по ходу изложения даются все необходимые определения и пояснения. Лучший выбор, чтобы освоить Git и максимально быстро начать его применять на практике.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-5-94387-547-2

9 78- 5- 94387- 547- 2

Контактные телефоны издательства:
(812) 412 70 26

Официальный сайт: www.nit.com.ru

© Фишерман Л. В.
© Наука и Техника (оригинал-макет)

Содержание

ГЛАВА 1. ВВЕДЕНИЕ В GIT 9

1.1. НЕОБХОДИМОСТЬ РЕЗЕРВНОГО КОПИРОВАНИЯ 10

1.2. ЭТАПЫ РАЗВИТИЯ 11

1.3. ЧТО В ИМЕНИ ТВОЕМ? 12

ГЛАВА 2. УСТАНОВКА GIT 13

2.1. ИСПОЛЬЗОВАНИЕ ДВОИЧНЫХ ДИСТРИБУТИВОВ LINUX 14

Debian/Ubuntu 15

Другие дистрибутивы 16

2.2. ПОЛУЧЕНИЕ ИСХОДНОГО КОДА 17

2.3. СБОРКА И УСТАНОВКА 18

2.4. УСТАНОВКА GIT В WINDOWS 20

Установка пакета Cygwin Git 21

Установка msysGit 22

ГЛАВА 3. НАЧАЛО ИСПОЛЬЗОВАНИЯ 25

3.1. КОМАНДНАЯ СТРОКА GIT 26

3.2. БЫСТРОЕ ВВЕДЕНИЕ В GIT 29

Создание начального репозитария 29

Добавление файлов в ваш репозитарий 30

Неясные сообщения об ошибках 32

Настройка автора коммита (фиксации) 32

Вносим другой коммит (фиксацию) 33

Просмотр ваших фиксаций 33

Просмотр разницы между коммитами (фиксациями) 35

Удаление и переименование файлов в вашем репозитарии 36

Копирование вашего репозитария 37

3.3. ФАЙЛЫ КОНФИГУРАЦИИ 38

Настройка псевдонимов 40

Резюме 41

ГЛАВА 4. БАЗОВЫЕ ПОНЯТИЯ GIT 42

4.1. БАЗОВЫЕ ПОНЯТИЯ 43
Репозитарии 44
Типы объектов Git 44
Индекс 46
Глобально уникальные идентификаторы 47
Git отслеживает контент 47
Путь по сравнению с содержанием 48
Pack-файлы 49

4.2. ИЗОБРАЖЕНИЯ ХРАНИЛИЩА ОБЪЕКТОВ 50

4.3. ПОНЯТИЯ GIT В ДЕЙСТВИИ 53
Внутри каталога .git 53
Объекты, хэши, блобы 55
Откуда мы знаем, что хэш SHA1 уникален? 55
Файлы и деревья 56
Примечание относительно использования SHA1 57
Иерархия деревьев 59
Фиксации 60
Теги 62

ГЛАВА 5. УПРАВЛЕНИЕ ФАЙЛАМИ. ИНДЕКС 65

5.1. ВСЕ ОБ ИНДЕКСЕ 67

5.2. КЛАССИФИКАЦИЯ ФАЙЛОВ В GIT 68

5.3. ИСПОЛЬЗОВАНИЕ GIT ADD 70

5.4. НЕКОТОРЫЕ ЗАМЕЧАНИЯ ОТНОСИТЕЛЬНО ИСПОЛЬЗОВАНИЕ GIT COMMIT 73

5.5. ИСПОЛЬЗОВАНИЕ GIT RM 75

5.6. ИСПОЛЬЗОВАНИЕ GIT MV 78

5.7. ЗАМЕЧАНИЕ ОТНОСИТЕЛЬНО ОТСЛЕЖИВАНИЯ ПЕРЕИМЕНОВАНИЙ. 80

5.8. ПРОБЛЕМЫ С ОТСЛЕЖИВАНИЕМ ПЕРЕИМЕНОВАНИЯ 81

5.9. ФАЙЛ .GITINGNORE 82

ГЛАВА 6. КОММИТЫ (ФИКСАЦИИ) 85

6.1. АТОМАРНЫЕ НАБОРЫ ИЗМЕНЕНИЙ 87

6.2. ИДЕНТИФИКАЦИЯ ФИКСАЦИЙ 88

Абсолютные имена фиксации 89
Ссылки и символьные ссылки 90
Относительные имена фиксаций 92

6.3. ИСТОРИЯ ФИКСАЦИЙ 95

Просмотр старых фиксаций 95
Графы фиксации 98
Использование gitk для просмотра графа фиксации 102
Диапазоны фиксаций 103

6.4. ДОСТИЖИМОСТЬ В ГРАФАХ 104

6.5. ПОИСК ФИКСАЦИЙ 108

Использование git bisect 108
Использование git blame 114
Использование Pickaxe 115

ГЛАВА 7. ВЕТКИ В GIT 117

7.1. ПРИЧИНЫ ИСПОЛЬЗОВАНИЯ ВЕТОК 118

7.2. ВЕТКА ИЛИ ТЕГ? 119

Имена веток 120
Правила именования веток 120

7.3. ИСПОЛЬЗОВАНИЕ ВЕТОК 121

7.4. СОЗДАНИЕ ВЕТОК 123

7.5. ВЫВОД ИМЕН ВЕТОК 124

7.6. ПРОСМОТР ВЕТОК 125

7.7. ПЕРЕКЛЮЧЕНИЕ ВЕТОК И ВЫГРУЗКА ФАЙЛОВ 128

Базовое переключение между ветками 128
Переключение веток при имеющихся незафиксированных изменениях 129
Объединение изменений в другую ветку 131
Создание новой ветки и переключение на нее 133
Отсоединение головы ветки 135

7.8. УДАЛЕНИЕ ВЕТОК 136

ГЛАВА 8. РАЗЛИЧИЯ В GIT 139

8.1. ФОРМЫ КОМАНДЫ GIT DIFF 141

8.2. ПРОСТОЙ ПРИМЕР GIT DIFF 144

8.3. КОМАНДА GIT DIFF И ДИАПАЗОНЫ ФИКСАЦИЙ 148

8.4. КОМАНДА GIT DIFF С ОГРАНИЧЕНИЕМ ПУТИ 151

Сравнение как Subversion (SVN) и Git получают разницы 154

ГЛАВА 9. СЛИЯНИЯ 157

9.1. ПРИМЕРЫ СЛИЯНИЯ 159

Подготовка к слиянию 159

Объединение двух веток 160

Слияние с конфликтом 162

9.2. РАБОТА С КОНФЛИКТАМИ СЛИЯНИЯ 167

Обнаружение конфликтных файлов 168

Исследование конфликтов 169

git diff с конфликтами 170

git log с конфликтами 173

Как Git отслеживает конфликты 175

Завершение разрешения конфликта 177

Отмена или перезапуск слияния 179

9.3. СТРАТЕГИИ СЛИЯНИЯ 179

Вырожденные слияния 183

Обычные слияния 185

Рекурсивные слияния 185

Слияния осьминога 186

Специальные слияния 187

9.4. ПРИМЕНЕНИЕ СТРАТЕГИЙ СЛИЯНИЯ 187

Использование стратегий ours и subtree 189

Драйверы слияния 190

Как Git думает о слияниях 191

Слияния и объектная модель Git 191

Слияние squash 192
Почему не выполнять слияние каждого изменения одно за другим? 193

ГЛАВА 10. ИЗМЕНЕНИЕ КОММИТОВ (ФИКСАЦИЙ)... 195

10.1. ЗАЧЕМ ПРОИЗВОДИТЬ ИЗМЕНЕНИЯ ИСТОРИИ 197

10.2. ИСПОЛЬЗОВАНИЕ КОМАНДЫ GIT RESET 199

10.3. ИСПОЛЬЗОВАНИЕ КОМАНДЫ GIT CHERRY-PICK 209

10.4. ИСПОЛЬЗОВАНИЕ КОМАНДЫ GIT REVERT 211

10.5. КОМАНДЫ RESET, REVERT И CHECKOUT 212

10.6. ИЗМЕНЕНИЕ ПОСЛЕДНЕЙ ФИКСАЦИИ 214

10.7. ПЕРЕБАЗИРОВАНИЕ ФИКСАЦИЙ 217
Использование команды git rebase -i 220
Сравнение rebase и merge 225

ГЛАВА 11. STASH И REFLOG 233

11.1. STASH 234

11.2. REFLOG 246

ГЛАВА 12. УДАЛЕННЫЕ РЕПОЗИТАРИИ 253

12.1. ПОНЯТИЯ РЕПОЗИТАРИЯ 256
Чистый репозитарий и репозитарий разработки 256
Клоны репозитариев 257
Удаленные 258
Ветки отслеживания 260

12.2. ССЫЛКИ НА ДРУГИЕ РЕПОЗИТАРИИ 261
Ссылки на удаленные репозитарии 262
Refspec 264

12.3. ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ УДАЛЕННЫХ РЕПОЗИТАРИЕВ 267
Создание авторитетного репозитария 268
Создание вашей собственной удаленной origin 270
Разработка в вашем репозитарии 273
Передача ваших изменений 274

Добавление нового разработчика 275
Получение обновлений репозитария 278

12.4. УДАЛЕННАЯ КОНФИГУРАЦИЯ 285
Использование команды git remote 285
Использование git config 287
Редактирование файла конфигурации вручную 288
Несколько удаленных репозитариев 288

12.5. РАБОТА С ВЕТКАМИ ОТСЛЕЖИВАНИЯ 289
Создание веток отслеживания 289
Впереди и сзади 293

12.6. ДОБАВЛЕНИЕ И УДАЛЕНИЕ УДАЛЕННЫХ ВЕТОК 295

12.7. ЧИСТЫЕ (BARE) РЕПОЗИТАРИИ И КОМАНДА GIT PUSH 297

Глава 1.

Введение в Git



1.1. Необходимость резервного копирования

Ни один осторожный и творческий человек не начнет проект в наше время без четкой стратегии резервного копирования. Данные эфемерны и могут быть легко потеряны, например, в результате ошибочного изменения или отказа диска. Очень разумно поддерживать оперативный архив всей своей работы.

Для проектов, содержащих текст и код, стратегия резервного копирования обычно включает управление версиями. Каждый разработчик в день может создать несколько версий кода, а представьте, что над проектом работает целая команда. Учитывая его основную роль, управление версиями является наиболее эффективным, когда оно адаптировано в соответствии с привычками разработчика и целями команды проекта.

Утилита, которая управляет и отслеживает разные версии программного обеспечения, обычно называется системой контроля версий (version/revision control system, VCS), менеджером исходного кода (source code manager, SCM) и еще несколько вариантов, построенных путем перестановки слов «версия», «код», «управление», «контент» и «система». Хотя авто-

ры и пользователи той или иной утилиты могут доказывать, что их утилита - лучшая, назначение каждой системы то же: разработка и обслуживание репозитария контента, предоставление доступа к более старым редакциям кода по дате, запись всех изменений в журнал. В этой книге мы будем использовать термин система контроля версий (VCS, version control system) для всех утилит подобного рода.

В данной книге рассматривается утилита Git - мощный и гибкий инструмент управления версиями. Git был создан Линусом Торвальдсом для поддержки разработки ядра Linux, но с тех пор он оказался полезен огромному числу других проектов.

1.2. Этапы развития

Первая версия Git появилась в апреле 2005 года, а именно 7 апреля (сообщения были переведены на русский язык):

```
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date: Thu Apr 7 15:13:13 2005 -0700
Первая версия «git», менеджера данных из ада
```

Немного позже была выполнена первая фиксация (коммит) ядра Linux:

```
commit 1da177e4c3f41524e886b7f1b8a0c1fc7321cac2
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date: Sat Apr 16 15:20:36 2005 -0700
Linux-2.6.12-rc2
Начальная сборка репозитария git. Я не волнуюсь относительно
полной истории, даже при том, что она у нас есть. Мы можем
создать отдельный «исторический» архив Git позже, если мы
захотим. Тем временем - это 3.2 Гб дискового пространства
- объем, который сделал бы ранние дни Git излишне сложными,
когда у нас нет хорошей инфраструктуры для него.
Давайте разорвем его!
```

Да, одна фиксация (коммит) представляла собой объем всего ядра Linux. Она состояла из:

```
17291 files changed, 6718755 insertions(+), 0 deletions(-)
```

Да, вы все правильно поняли: 6.7 миллионов строк кода!

Это было всего три минуты спустя, когда первый патч с использованием Git был применен к ядру. Убедившись, что он работает, Линус объявил о нем 20 апреля 2005 года в списке рассылки ядра Linux (Linux Kernel Mailing List).

Приблизительно два месяца спустя с использованием Git была выпущена версия 2.6.12 ядра Linux.

1.3. Что в имени твоем?

Сам Линус рационализирует имя «Git», утверждая, что «Я - эгоцентричный ублюдок и называю все свои проекты в свою честь. Сначала Linux, теперь Git». Ведь Linux - это гибрид Linus и Minix. А в случае с Git Линус использовал британский термин «git», на сленге означающий «гнилой человек», «мерзавец». Почему - знает только он.

С тех пор другие пользователи Git предложили некоторую альтернативу и возможно более приемлемые интерпретации. Наиболее популярной интерпретацией является Global Information Traker.

Глава 2.

Установка Git



На момент написания этих строк Git по умолчанию не установлен ни в одном дистрибутиве GNU/Linux или любой другой операционной системе. Поэтому прежде, чем вы сможете использовать Git, вы должны установить его. Действия, необходимые для установки Git, зависят от производителя и версии вашей операционной системы. В этой главе вы узнаете, как установить Git в Linux, Microsoft Windows и в Cygwin.

2.1. Использование двоичных дистрибутивов Linux

Много поставщиков Linux предоставляют предварительно скомпилированные, двоичные пакеты для упрощения установки новых приложений, инструментов и утилит. У каждого пакета есть свои зависимости, и диспетчер пакетов при установке пакета устанавливает также и пакеты, от которых зависит устанавливаемый пакет. Все автоматизировано и пользователю нужно только указать, какой пакет он хочет установить.

Debian/Ubuntu

На большинстве систем Debian и Ubuntu Git поставляется в виде набора пакетов, где каждый пакет может быть установлен независимо, в зависимости от ваших потребностей. До версии 12.04 основной пакет назывался git-core, а, начиная с выпуска 12.04, он называется просто git, а пакет с документацией называется git-doc. Также доступны дополнительные пакеты:

git-arch

git-cvs

git-svn

Данные пакеты понадобятся для переноса проекта из Arch, CVS и SVN в Git соответственно. Установите один или более из этих пакетов:

git-gui

gitk

gitweb

Если вы предпочитаете просматривать свои репозитарии с помощью графического приложения или в вашем браузере, установите эти пакеты. Пакет git-gui - это графический интерфейс для Git, основанный на библиотеке Tcl/Tk. Пакет gittk - другой Git-браузер, тоже написанный на Tcl/Tk, но с лучшей визуализацией истории проекта. Пакет gitweb - веб-интерфейс для Git-репозитария, написанный на Perl.

git-email

Это очень важный компонент, если вы хотите отправлять Git-патчи посредством электронной почты, что является часто используемой практикой в некоторых проектах.

`git-daemon-run`

Установите этот пакет, чтобы предоставить общий доступ к вашему репозитарию. Он установит службу, позволяющую «расшарить» ваши репозитарии посредством анонимных запросов загрузки.

Поскольку дистрибутивы значительно варьируются, список пакетов Git в вашем дистрибутиве может немного отличаться. Пакеты git-doc и git-email настоятельно рекомендуются.

Следующая команда установит наиболее важные пакеты Git (вам нужны права root для запуска apt-get):

```
$ sudo apt-get install git git-doc gitweb \
git-gui gitk git-email git-svn
```

Другие дистрибутивы

Чтобы установить Git в других дистрибутивах, найдите соответствующий пакет или пакеты и используйте родной менеджер пакетов дистрибутива для установки программного обеспечения. Например, в системах Gentoo используется emerge:

```
$ sudo emerge dev-util/git
```

В Fedora используйте **yum**:

```
$ sudo yum install git
```

Обратите внимание, что пакет git в Fedora практически аналогичен пакету git в Debian. Список других пакетов Git, которые вы найдете в Fedora (платформа i386):

git.i386:

Основные утилиты Git

git-all.i386:

Мета-пакет для установки всех утилит Git

git-arch.i386:

Git-утилиты для импорта репозитариев Arch

git-cvs.i386:

Git-утилиты для импорта репозитариев CVS

git-daemon.i386:

Демон протокола Git

git-debuginfo.i386:

Информация отладки для пакета git

git-email.i386:

Утилиты Git для отправки email

git-gui.i386:

Графический интерфейс для Git

git-svn.i386:

Git-утилиты для импорта репозитариев SVN

gitk.i386:

Визуализатор дерева изменений Git

Снова, будьте внимательны: некоторые дистрибутивы, подобно Debian, могут разделить Git на много разных пакетов. Если в вашей системе не поддерживается определенная команда Git, значит, вы должны установить дополнительный пакет.

Обязательно убедитесь, что пакеты вашего дистрибутива Git достаточно актуальны. После установки Git в вашей системе выполните команду `git --version`. Если ваши сотрудники используют более современную версию Git, вам нужно будет заменить скомпилированные пакеты Git вашего дистрибутива собственной сборкой. Проконсультируйтесь с документацией по своему диспетчеру пакетов, чтобы узнать, как удалить ранее установленные пакеты, а потом перейдите к следующему разделу, чтобы узнать, как установить Git из исходного кода.

2.2. Получение исходного кода

Если вы предпочитаете загрузить код Git из его канонического источника или же вам нужна самая последняя версия Git, посетите основной репозитарий Git. На момент написания этих строк этот репозитарий находится на

сайте http://git.kernel.org в каталоге pub/software/scm. В этой книге описывается версия Git 2.28.0, но, возможно, уже доступна более новая версия. Список всех доступных версий можно найти по адресу:

http://code.google.com/p/git-core/downloads/list (версии до 1.9.0)

и

https://git-scm.com/downloads

Чтобы начать сборку, загрузите исходный код версии 2.29.0 (или более поздней) и распакуйте его.

2.3. Сборка и установка

Процесс сборки Git подобен любому другому программному обеспечению OpenSource. Просто настройте его, введите make и установите его.

Если в вашей системе установлены надлежащие библиотеки и сформирована устойчивая среда сборки, установка Git из исходного кода - оптимальное решение. С другой стороны, если на вашей машине отсутствует компилятор и необходимые для сборки Git библиотеки, а также, если вы никогда не собирали сложное приложение из исходного кода, рассмотрите сборку Git только в качестве крайней мере, когда других вариантов нет. Git гибко настраиваемый, но его сборка никогда не была легким процессом.

Для продолжения сборки откройте файл INSTALL, содержащийся в пакете исходного кода Git. В нем перечислены несколько внешних зависимостей, включая библиотеки zlib, openssl и libcurl.

Некоторые необходимые библиотеки и пакеты немного неясны или принадлежат большему пакету. Рассмотрим три подсказки для стабильного дистрибутива Debian:

- **curl-config** - небольшая утилита, позволяющая извлечь информацию о локальной инсталляции curl, ее можно найти в пакете libcurl4-openssl-dev.
- Заголовочный файл **expat.h** можно взять из пакета libexpat1-dev.
- Утилита **msgfmt** принадлежит пакету gettext.

Поскольку компиляцию из источников считают работой разработчика, обычных двоичных версия установленных библиотек не достаточно. Вместо них вам нужны dev-версиях, поскольку они также предоставляют заголовочные файлы, необходимые во время компиляции.

Если вы не можете определить местоположение некоторых из этих пакетов или не можете найти необходимую библиотеку в своей системе, Make-файл (Makefile) и параметры конфигурации предлагают альтернативы. Например, если у вас отсутствует библиотека expat, вы можете установить опцию NO_EXPAT в Make-файле. Однако, тогда в вашей сборке будут отсутствовать некоторые функции, для выполнения которых была нужна библиотека expat. Например, вы будете не в состоянии поместить изменения в удаленный репозитарий с использованием протоколов HTTPS и HTTP.

Другие параметры конфигурации файла Makefile поддерживают портирование на различные платформы и дистрибутивы. Например, в нем есть несколько флагов, относящихся к операционной системе Mac OS X Dawrin.

Как только ваша система и параметры сборки готовы, остальное все просто. По умолчанию Git устанавливается в ваш домашний каталог в подкаталоги ~/bin, ~/lib и ~/share. Это полезно, если вы используете Git персонально, но никак не подходит, если вы используете Git коллективно.

Введите следующие команды для сборки и установки Git в ваш домашний каталог:

```
$ cd git-2.29.0
$ ./configure
$ make all
$ make install
```

Если вы хотите установить Git в альтернативное расположение, например, в /usr/local, добавьте префикс --prefix=/usr/local к команде ./configure. Для продолжения выполните make install не как обычный пользователь, а как root:

```
$ cd git-2.29.0
$ ./configure --prefix=/usr/local
$ make all
$ sudo make install
```

Для установки документации Git выполните эти команды:

```
$ cd git-2.29.0
$ make all doc
$ sudo make install install-doc
```

Для полной сборки документации необходимо несколько библиотек. В качестве альтернативы можно использовать пакеты со страницами руководства (man) и HTML страницы, которые можно установить отдельно от git (можно собрать git из исходников, а документацию установить из пакетов).

Сборка из исходного кода включает все подпакеты Git и команды, в том числе **git-email** и **gitk**. Вам не нужно собирать или устанавливать эти утилиты отдельно.

2.4. Установка Git в Windows

Существует два полных пакета Git для Windows: Cygwin-ориентированная версия Git и «родная» версия, называемая msysGit.

Изначально поддерживалась Cygwin-версия, а msysGit был экспериментальным и нестабильным. Но на момент печати этой книги обе версии работали стабильно и обладали примерно одинаковым функционалом. Наиболее важное исключение: msysGit пока не полностью поддерживает git-svn. Если вам необходимо взаимодействие между Git и SVN, вам нужео использовать Cygwin-версию, в противном случае вы можете использовать любую версию.

Если вы не уверены, какая версия вам нужна, вот некоторые эмпирические правила:

- Если вы уже используете Cygwin в Windows, используйте Git для Cygwin, поскольку он будет взаимодействовать с вашей установкой Cygwin, к которой вы привыкли. Например, имена файлов Cygwin-стиля будут работать в Git. Перенаправление ввода/Вывода программы будут всегда работать точно, как и ожидалось.
- Если вы не используете Cygwin, проще установить msysGit, потому что у него есть свой собственный инсталлятор.
- Если вы хотите интегрировать Git с оболочкой Windows Explorer, установите msysGit. Если вы хотите эту функцию, но предпочитаете использовать Cygwin, установите оба пакета.

Если вы все равно не определились, какой пакет вам нужен, установите msysGit. Только убедитесь, что вы получили последнюю версию.

Установка пакета Cygwin Git

Для установки Cygwin запустите программу установки setup.exe, получить которую можно по адресу: **http://cygwin.com/**

После запуска setup.exe используйте параметры по умолчанию для большинства опций, пока вы не доберетесь до списка пакетов, доступных для установки. Пакеты Git будут в категории **devel**, как показано на рис. 2.2.



Рис. 2.1. Установка Cygwin

После выбора пакетов для установки нажмите **Next** несколько раз, пока установка Cygwin не будет завершена. После этого запустите **Cygwin Bash Shell** или **Cygwin Terminal** (в зависимости от версии Cygwin) из вашего меню Пуск, которая должна содержать программную группу **Cygwin** (рис. 2.3).

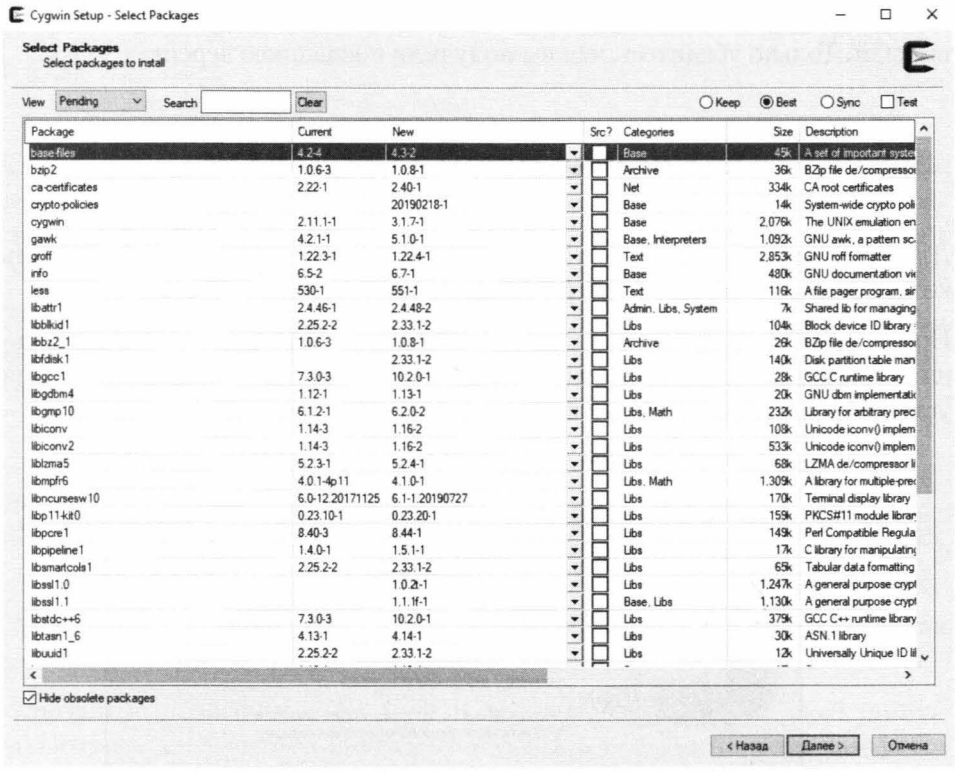


Рис. 2.2. Установка Cygwin

В качестве альтернативы вы можете собрать вашу собственную копию Git из исходного кода в Windows, если ваша конфигурация Cygwin содержит утилиты компилятора вроде gcc и make следуя тем же инструкциям, что и в Linux.

Установка msysGit

Пакет msysGit очень просто установить в Windows, поскольку этот пакет содержит все свои зависимости. В нем даже есть команды Secure Shell (SSH) для создания ключей, генерирующихся для управления доступом к репозиториям. Также msysGit интегрируется с оболочкой Windows (Проводником).

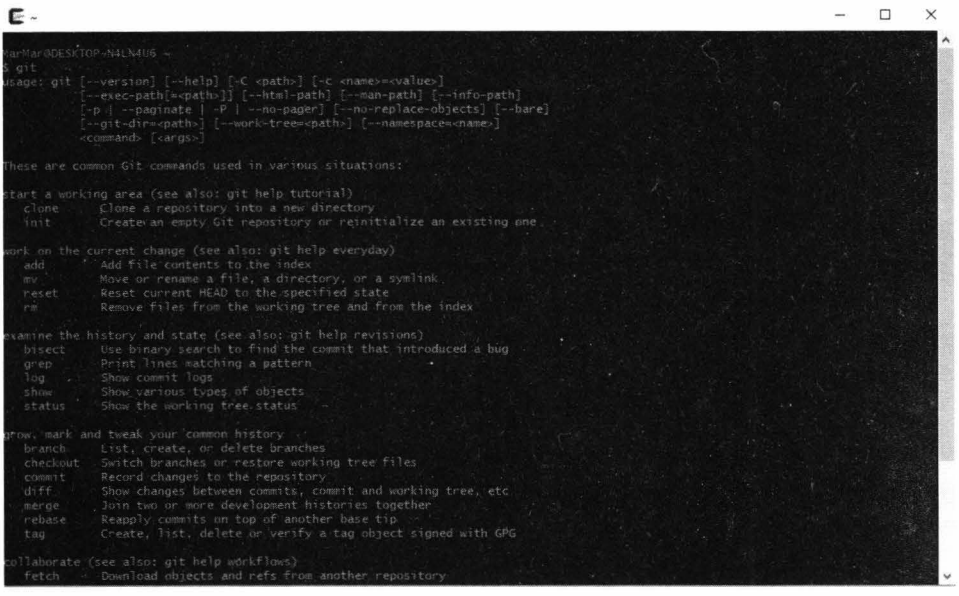


Рис. 2.3. Оболочка Cygwin

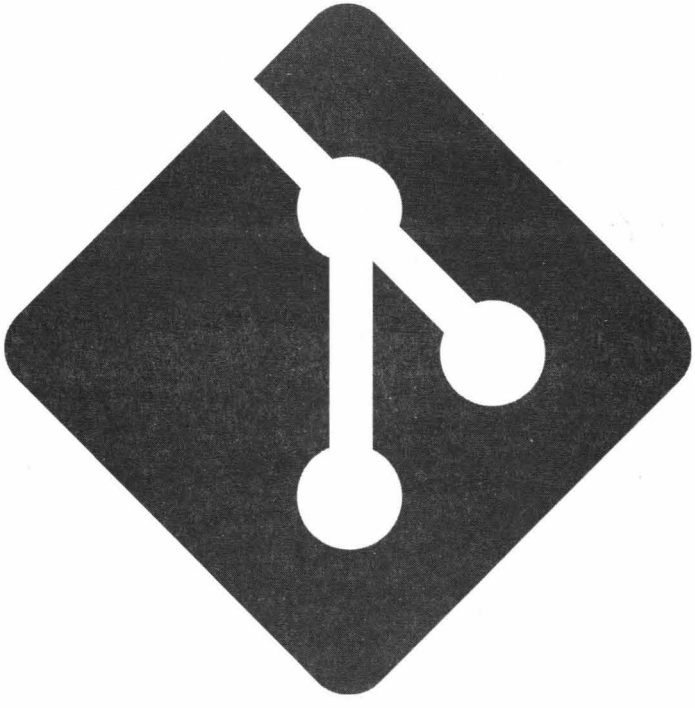
Первым делом загрузите последнюю версию инсталлятора по адресу **https://gitforwindows.org/** Файл, который вам нужно получить, называется примерно так Git-2.28.0-64.exe.

После загрузки инсталлятора, запустите его. В процессе установки инсталлятор спросит вас, как обрабатывать окончания строк - в Windows-стиле или Unix-стиле. Если вся ваша команда работает в Windows, выберите Windows-стиль, если же ваша команда в основном использует Unix, выберите Unix-стиль.

Нажмите **Next** несколько раз. Лучший способ запустить msysGit - через Проводник Windows.

Дополнительно также будет установлен ярлык **Git Bash** (командная строка, позволяющая вводить команды Git) в меню **Пуск** в программную группу **Git**. Поскольку большинство из примеров в этой книге используют командную строку, этот ярлык вам очень пригодится.

Все примеры в этой книге работают одинаково, что в Linux, что в Windows, но есть один нюанс: msysGit для Windows использует старые имена команд Git, как будет описано в разделе «Командная строка Git» главы 3. В msysGit нужно вводить git-add для git add.



Глава 3.

Начало использования



Git управляет изменениями. Git - одна из систем управления версиями. Множество принципов, например, понятие фиксации, журнала изменений, репозитария, являются одинаковыми для всех инструментов подобного рода. Но Git предлагает много новинок. Некоторые понятия и методы других систем управления версиями могут работать по-другому в Git или не работать вообще. Независимо от наличия вашего опыта работы с подобными системами, данная книга учит вас работать с Git.

Давайте начнем!

3.1. Командная строка Git

Git очень просто использовать. Просто введите git без всяких аргументов. Git выведет свои параметры и наиболее общие подкоманды.

```
$ git

git [--version] [--exec-path[=GIT_EXEC_PATH]]
[-p|--paginate|--no-pager] [--bare] [--git-dir=GIT_DIR]
```

```
[--work-tree=GIT_WORK_TREE] [--help] COMMAND [ARGS]
```

Далее приведены наиболее часто используемые команды git:

```
add Добавляет содержимое файла в индекс
bisect Выполняет бинарный поиск ошибки по истории фиксаций
branch Выводит, создает или удаляет ветки
checkout Проверяет и переключает на ветку
clone Клонирует репозиторий в новый каталог
commit Записывает изменения в каталог (фиксация)
diff Показывает изменения между фиксациями, рабочими
деревьями и т.д.
fetch Загружает объекты и ссылки из другого репозитария
grep Выводит строки, соответствующие шаблону
init Создает пустой репозитарий git или переинициализирует
существующий
log Показывает журналы фиксации
merge Объединяет две или больше истории разработки
mv Перемещает или переименовывает файл, каталог или
символическую ссылку
pull Получает изменения из удаленного репозитария и
объединяет их с локальным репозитарием или локальной веткой
push Загружает изменения из вашего локального репозитория в
удаленный
rebase Строит ровную линию фиксаций
reset Сбрасывает текущий HEAD в указанное состояние
rm Удаляет файлы из рабочего дерева и из индекса
show Показывает различные типы объектов
status Показывает состояние рабочего дерева
tag Создает, выводит, удаляет или проверяет тег объекта,
подписанного с GPG
```

Чтобы вывести полный список подкоманд git, введите:

```
git help --all
```

Как видно из подсказки использования, имеется множество полезных опций для git. Большинство опций, показанных как [ARGS], применяются к определенным подкомандам.

Например, опция --version применяется к команде git и производит вывод номера версии:

```
$ git --version
git version 2.29.0.windows.1
```

А опция --amend, наоборот, относится к подкоманде commit:

```
$ git commit --amend
```

В некоторых случаях придется указывать обе формы опция, например:

```
$ git --git-dir=project.git repack -d
```

Получить справку по каждой подкоманде git можно следующими способами:

```
git help subcommand, git --help subcommand или git subcommand --help
```

Раньше у Git был набор простых, отличных, автономных команд, разработанных согласно философии «Unix toolkit: небольшие и независимые инструменты. Каждая команда начиналась со строки git и содержала дефис, после которого следовало название выполнимого действия, например, git-commit и git-log. Однако на данный момент считается, что должна быть одна единственная исполнимая программа git, а выполняемые действия должны передаваться в качестве параметров, например, git commit и git log. Нужно отметить, что формы команд git commit и git-commit идентичны.

> **Примечание. Онлайн-документация по Git доступна по адресу http://www.kernel.org/pub/software/scm/git/docs/**

Команды git понимают «короткие» и «длинные» команды. Например, следующие две команды git commit эквиваленты:

```
$ git commit -m "Fixed a typo."
$ git commit --message="Fixed a typo."
```

В короткой форме используется один дефис, а длинная форма использует два. Некоторые опции существуют только в одной форме.

Наконец, вы можете разделить параметры из списка аргументов, используя «пустое двойное тире». Например, используйте двойное тире, чтобы отде-

лить часть управления командной строки от списка операндов, например, от имен файлов:

```
$ git diff -w master origin -- tools/Makefile
```

Двойное тире нужно использовать, чтобы разделить и явно идентифицировать имена файлов, если иначе они могли бы быть приняты за другую часть команды. Например, если у вас есть тег «main.c» и файл «main.c», вы получите различное поведение:

```
# Проверка тега с именем «main.c»
$ git checkout main.c
# Проверка файла с именем «main.c»
$ git checkout -- main.c
```

3.2. Быстрое введение в Git

Чтобы увидеть git в действии, давайте создадим новый репозитарий и добавим в него некоторое содержимое, а после чего сделаем несколько версий этого содержимого.

Существует две фундаментальных техники установки репозитария Git. Вы можете создать его с нуля, а потом заполнить его содержимым, или же скопировать, или как говорят в мире Git, склонировать, существующий репозитарий. Проще начать с пустого репозитария, поэтому давайте сделаем это.

Создание начального репозитария

Чтобы смоделировать типичную ситуацию, давайте создадим репозитарий для вашего персонального сайта в каталоге ~/public_html.

Если у вас нет вашего личного сайта в ~/public_html, создайте этот каталог и поместите в него файл index.html с любой строкой:

```
$ mkdir ~/public_html
$ cd ~/public_html
$ echo 'Мой сайт жив!' > index.html
```

Чтобы превратить ~/public_html или любой другой каталог в репозитарий Git, просто запустите git init:

```
$ git init
Initialized empty Git repository in .git/
```

Программе git все равно, начинаете ли вы с полностью пустого каталога или с каталога, полного файлов. Процесс преобразования каталога в репозитарий Git будет одинаковым.

Чтобы отметить, что ваш каталог является репозитарием Git, команда git init создает скрытый каталог с названием .git, находящийся на верхнем уровне вашего проекта. Содержимое и назначение файлов из этого каталога описано в главе 4.

Все остальное в каталоге ~/public_html останется нетронутым. Git считает этот каталог рабочим каталогом вашего проекта, в котором вы изменяете свои файлы. Git интересуют только файлы из скрытого каталога .git.

Добавление файлов в ваш репозитарий

Команда **git init** создает новый репозитарий Git. В самом начале каждый репозитарий Git будет пустым. Чтобы управлять контентом, вы должны явно добавить его в репозитарий. Такой осознанный файл позволяет разделить рабочие файлы от важных файлов.

Команда `git add <файл>` используется для добавления файла <файл> в репозитарий:

```
$ git add index.html
```

> **Примечание. Если в вашем каталоге есть несколько файлов, не нужно добавлять все их вручную, пусть за вас это сделает git. Чтобы добавить в репозитарий все файлы в каталоге и во всех подкаталогах, используйте команду git add . (одна точка в Unix означает текущий каталог).**

После добавления файла Git знает, что файл index.html принадлежит репозитарию. Но Git просто подготовил файл, это еще не все. В Git разделены

операции добавления и фиксации. Это сделано для лучшей производительности - вы только представьте, сколько времени займет обновление репозитария при каждом добавлении или удалении файла. Вместо этого Git предлагает раздельные операции, что особенно удобно в пакетном режиме.

Команда **git status** покажет промежуточное состояние файла index.html:

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
# (use "git rm --cached <file>..." to unstage)
#
# new file: index.html
```

Данная команда сообщает, что при следующей фиксации в репозитарий будет добавлен новый файл index.html.

Помимо актуальных изменений в каталоге и его файлах при каждой фиксации Git записывает различные метаданные, включая сообщение журнала и автора изменения. Полная команда фиксации выглядит так:

```
$ git commit -m «Начальное содержимое public_html» \
--author="Федя Колобков <kolobok@lobok.com>"
Created initial commit 9da581d: Initial contents of public_html
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 index.html
```

Вы можете предоставить сообщение журнала из командной строки, но обычно удобнее создать его с помощью интерактивного текстового редактора. Чтобы Git открывал ваш любимый текстовый редактор при фиксации изменений, установите переменную окружения GIT_EDITOR:

```
# В tcsh
$ setenv GIT_EDITOR emacs
# В bash
$ export GIT_EDITOR=vim
```

После фиксации добавления нового файла в репозитарий, команда git status покажет, что нет ничего, что требовало ли фиксации:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Git также сообщает вам, что ваш рабочий каталог чист. А это означает, что в вашем рабочем каталоге нет неизвестных или измененных файлов, которые отличаются от тех, которые находятся в репозитарии.

Неясные сообщения об ошибках

Git пытается определить автора каждого изменения. Если вы не сконфигурировали свое имя и e-mail так, чтобы они были доступны в Git, это может стать причиной некоторых предупреждений. Но не нужно паниковать, когда вы видите примерно такие сообщения:

```
You don't exist. Go away!
Your parents must have hated you!
Your sysadmin must hate you!
```

Они просто означают, что Git не может определить ваше реальное имя по какой-то причине. Проблема может быть исправлена путем установки вашего имени и e-mail, что будет показано ниже».

Настройка автора коммита (фиксации)

Перед коммитом (фиксацией) вам нужно установить некоторые параметры конфигурации и переменные окружения. Как минимум, Git должен знать ваше имя и ваш e-mail. Вы можете указывать эти данные при каждой команде фиксации (git commit), как было показано ранее, но это очень неудобно. Вместо этого лучше всего сохранить данную информацию в файле конфигурации, используя команду **git config**:

```
$ git config user.name "Федя Колобков"
$ git config user.email kolobok@lobok.com
```

Вы также можете сообщить Git свое имя и свой e-mail, установив переменные окружения GIT_AUTHOR_NAME и GIT_AUTHOR_EMAIL. Если эти переменные окружения установлены, они перезапишут все параметры, установленные в файле конфигурации.

Вносим другой коммит (фиксацию)

Чтобы продемонстрировать еще несколько функций Git, давайте внесем некоторые изменения и создадим сложную историю изменений внутри репозитария.

Откройте index.html, конвертируйте его в HTML, сохраните изменения и выполните фиксацию изменений:

```
$ cd ~/public_html

# редактируем index.html

$ cat index.html
<html>
<body>
Мой сайт жив!
</body>
</html>

$ git commit index.html
```

Когда откроется редактор, введите описание фиксации, например, «Файл конвертирован в HTML». После чего в репозитарии появится две версии файла index.html.

Просмотр ваших фиксаций

Как только в вашем репозитарии появилось несколько фиксаций, вы можете исследовать их различными способами. Некоторые команды Git показывают последовательность отдельных фиксаций, другие - сводку об отдельной фиксации, а остальные показывают полную информацию о любой фиксации в каталоге.

Команда **git log** показывает историю отдельных фиксаций в репозитарии:

```
$ git log
commit ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6
Author: Федя Колобков <kolobok@lobok.com>
Date: Wed Apr 2 16:47:42 2020 -0500

Файл конвертирован в HTML

commit 9da581d910c9c4ac93557ca4859e767f5caf5169
Author: Федя Колобков <kolobok@lobok.com>
Date: Thu Mar 13 22:38:13 2020 -0500

Начальное содержимое public_html
```

Записи выводятся в порядке от самой последней до самой старой. Каждая запись показывает имя и e-mail автора фиксации, дату фиксации и сообщение, добавленное в журнал при фиксации. Идентификатор фиксации объяснен в главе 4, а пока вам просто нужно знать, что он уникальный.

Для получения более подробной информации о конкретной фиксации, используйте команду **git show**:

```
$ git show 9da581d910c9c4ac93557ca4859e767f5caf5169
commit 9da581d910c9c4ac93557ca4859e767f5caf5169
Author: Федя Колобков <kolobok@lobok.com>
Date: Thu Mar 13 22:38:13 2020 -0500
Initial contents of public_html
diff --git a/index.html b/index.html
new file mode 100644
index 0000000..34217e9
--- /dev/null
+++ b/index.html
@@ -0,0 +1 @@
+Мой сайт жив!
```

Если вы запустите git show без указания идентификатора фиксации, вы получите информацию о последней фиксации.

Другая команда, **show-branch**, выводит краткую сводку для текущей ветки разработки:

```
$ git show-branch --more=10
[master] Файл конвертирован в HTML
```

```
[master^] Начальное содержимое public_html
```

Параметр --more=10 указывает, что нужно вывести не более 10 версий, но в нашем случае пока существуют только две версии. Имя master - это имя по умолчанию для ветки. Ветки подробно описаны в главе 7, там же будет дополнительно описана команда git show-branch.

Просмотр разницы между коммитами (фиксациями)

Чтобы увидеть разницу между двумя версиями index.html, вам понадобятся идентификаторы этих двух фиксаций, которые нужно передать команде **git diff**:

```
$ git diff 9da581d910c9c4ac93557ca4859e767f5caf5169 \
ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6
diff --git a/index.html b/index.html
index 34217e9..8638631 100644
--- a/index.html
+++ b/index.html
@@ -1 +1,5 @@
+<html>
+<body>
Мой сайт жив!
+</body>
+</html>
```

Этот вывод должен выглядеть знакомым. Он напоминает вывод команды diff. Первая версия, 9da581d910c9c4ac93557ca4859e767f5caf5169, является более ранней версией содержимого, а версия с именем ec232cddfb94e0dfd5b 5855af8ded7f5eb5c90d6 является более новой. Таким образом, знак «плюс» (+) предшествует каждой строке нового содержимого.

Вы испугались, глядя на эти шестнадцатеричные идентификаторы? Не волнуйтесь, Git предлагает более лаконичные способы идентифицировать фиксации без необходимости указания таких сложных чисел.

Удаление и переименование файлов в вашем репозитарии

Удаление файла из репозитарии аналогично добавлению файла, но вместо команды **git add** используется команда **git rm**. Представим, что в нашем каталоге веб-сайта есть файл poem.html, который больше не нужен.

```
$ cd ~/public_html
$ ls
index.html poem.html

$ git rm poem.html
rm 'poem.html'

$ git commit -m "Удаляем поэму"
Created commit 364a708: Remove a poem
0 files changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 poem.html
```

Как и добавление, удаление состоит из двух этапов. Сначала git rm удаляет файл из репозитария, а затем git commit фиксирует изменения в репозитарии. Опция -m позволяет указать описание фиксации, например, «Удаляем поэму». Вы можете опустить эту опцию, чтобы задать сообщение в вашем любимом текстовом редакторе.

Для переименования файла можно использовать комбинацию git rm и git add или же, что намного быстрее, использовать команду **git mv**:

```
$ mv foo.html bar.html
$ git rm foo.html
rm 'foo.html'
$ git add bar.html
```

В данном случае мы сначала переименовываем файл foo.html в bar.html и удаляем файл foo.html, как из репозитария, так и с файловой системы, после чего добавляем файл bar.html в репозитарий.

Эту же операцию можно выполнить с помощью одной команды:

```
$ git mv foo.html bar.html
```

После всего этого нам нужно фиксировать изменения:

```
$ git commit -m «Переименование foo в bar»
Created commit 8805821: Moved foo to bar
1 files changed, 0 insertions(+), 0 deletions(-)
rename foo.html => bar.html (100%)
```

Git обрабатывает операции перемещения файла иначе, чем другие системы, используя механизм, основанный на базе подобия содержания между двумя версиями файлов. Особенности механизма описаны в главе 5.

Копирование вашего репозитария

Ранее мы создали наш начальный репозитарий в каталоге ~/public_html. Теперь мы можем создать полную копию (или клон) этого репозитария командой **git clone**.

Сейчас мы создадим копию нашего репозитария в нашем домашнем каталоге, она будет называться my_website:

```
$ cd ~
$ git clone public_html my_website
```

Хотя эти два репозитария Git теперь содержат те же объекты, файлы и каталоги, есть некоторые тонкие различия. Исследовать эти различия можно командами:

```
$ ls -lsa public_html my_website
$ diff -r public_html my_website
```

На локальной файловой системе, подобно этой, использование **git clone** создаст копию репозитария подобно командам cp -a или rsync. Однако Git поддерживает богатый набор любых других источников, в том числе сетевые репозитарии. Эти формы источников и их использование будут объяснены в главе 12.

Как только вы клонируете репозитарий, вы сможете изменять, делать новые фиксации, исследовать журналы и историю и т.д. Это полностью новый репозитарий со своей полной историей.

3.3. Файлы конфигурации

Конфигурационные файлы Git - это простые текстовые файлы в стиле ini-файлов. В них записываются различные установки, используемые многими Git-командами. Некоторые установки представляю собой сугубо личные предпочтения (например, color.pager), другие жизненно важны для правильного функционирования репозитария (core.repositoryformatversion), а остальные управляют поведением команд (например, gc.auto).

Подобно многим другим утилитам Git поддерживает иерархию конфигурационных файлов. В порядке уменьшения приоритета приведены его конфигурационные файлы:

- **.git/config** - специфичные для репозитария параметры конфигурации, которыми управляет опция --file. У этих настроек наивысший приоритет.
- **~/.gitconfig** - специфичные для пользователя параметры конфигурации, которыми управляет опция --global.
- **/etc/gitconfig** - системные параметры конфигурации, изменить которые можно опцией --system, если у вас есть надлежащие права доступа Unix (нужно право на запись этого файла). Данные настройки обладают наименьшим приоритетом.

В зависимости от вашей инсталляции, системные настройки могут быть где-то еще, возможно, в /usr/local/etc/config, или могут полностью отсутствовать.

Например, для установки имени и e-mail пользователя, произведшего фиксацию, нужно указать значения параметров user.name и user.email в вашем файле $HOME/.gitconfig. Для этого используется команда git config --global:

```
$ git config --global user.name "Федя Колобков"
$ git config --global user.email "kolobok@lobok.com"
```

Можно установить специфичные для репозитария имя пользователя и адрес электронной почты, которые переопределят глобальные установки, для этого просто опустите флаг --global:

```
$ git config user.name "Федя Колобков"
$ git config user.email "kolobok@special-project.example.org"
```

Используйте git config -l для вывода всех переменных, найденных во всем наборе файлов конфигурации:

```
# Создаем пустой репозитарий
$ mkdir /tmp/new
$ cd /tmp/new
$ git init

# Устанавливаем некоторые переменные
$ git config --global user.name "Федя Колобков"
$ git config --global user.email "kolobok@lobok.com"
$ git config user.email "kolobok@special-project.example.org"

$ git config -l
user.name=Федя Колобков
user.email=kolobok@lobok.com
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
user.email=kolobok@special-project.example.org
```

Поскольку файлы конфигурации - это обычные текстовые файлы, вы можете просмотреть их содержимое командой **cat** и отредактировать в любимом текстовом редакторе:

```
# Посмотрим на параметры репозитария
$ cat .git/config
[core]
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
[user]
email = kolobok@special-project.example.org
```

Или, если вы используете ОС от Microsoft, у вас будут некоторые изменения. Скорее всего, ваш конфигурационный файл будет выглядеть примерно так:

```
[core]
repositoryformatversion = 0
filemode = true
bare = true
logallrefupdates = true
symlinks = false
ignorecase = true
hideDotFiles = dotGitOnly
```

Большинство из этих изменений - из-за разницы в характеристиках файловой системы. Отключены символические ссылки, включено игнорирование регистра символов, другие настройки для скрытых файлов.

Используйте опцию --unset для удаления настройки:

```
$ git config --unset --global user.email
```

Для одной и той же цели существуют несколько параметров конфигурации и переменных окружения. Например, редактор, который будет вызван при создании сообщения журнала фиксации, можно задать с помощью:

- Переменной окружения GIT_EDITOR
- Опции конфигурации core.editor
- Переменной окружения VISUAL
- Переменной окружения EDITOR
- Команды vi

Есть несколько сотен параметров конфигурации. По мере чтения книги я буду обращать ваше внимание на самые важные из них. На странице руководства (man) по команде **git config** вы найдете более подробный (и все же не полный) список параметров конфигурации.

Настройка псевдонимов

Новичкам пригодится совет по настройке псевдонимов командной строки. Часто команды Git довольно сложны, поэтому для самых часто используемых команд вы можете создать псевдонимы:

```
$ git config --global alias.show-graph \
'log --graph --abbrev-commit --pretty=oneline'
```

В этом примере я создал псевдоним show-graph и сделал его доступным в любом созданном мной репозитарии. Теперь, когда я использую команду **git show-graph** будет выполнена длинная команда **git log** со всеми заданными опциями.

Резюме

Конечно, у вас осталось много вопросов без ответа даже после прочтения этой главы. Например, как Git хранит каждую версию файла? Что на самом деле представляет фиксация? Что такое ответвление? Что означает master? Хорошие вопросы. Следующая глава определяет некоторую терминологию и знакомит вас с определенными понятиями Git.

Глава 4.

Базовые понятия Git



4.1. Базовые понятия

В предыдущей главе было рассмотрено типичное приложение Git. Скорее всего, после ее прочтения у вас появилось гораздо больше вопросов, чем было до того. Какие данные Git хранит для каждой фиксации? Каково назначение каталога .git? Почему ID фиксации напоминает мусор? Нужно ли мне обращать внимание на него?

Если вы использовали другую VCS, например, SVN или CVS, команды, представленные во второй главе, наверняка покажутся вам знакомыми. На самом деле, Git служит для той же цели и предоставляет все операции, которые вы ожидаете увидеть в современной VCS. Однако, Git некоторые понятия Git отличаются, о чем мы и поговорим.

В этой главе мы выясним, чем отличается Git от других VCS, исследуя ключевые компоненты его архитектуры и некоторые важные понятия. В этой главе мы сфокусируемся на основах взаимодействия с одним репозиторием, а глава 12 объясняет, как работать с несколькими соединенными репозиториями.

Репозитарии

Репозитарий Git - просто база данных, содержащая всю информацию, необходимую для управления версиями и историей проекта. В Git, как и в большинстве других систем управления версиями, репозитарий хранит полную копию всего проекта на протяжении всей его жизни. Однако, в отличие от большинства других VCS, репозитарий Git не только предоставляет полную рабочую копию всех файлов в репозитарии, но также и копию самого репозитария, с которым работает.

В каждом репозитарии Git обслуживает ряд значений конфигурации. Вы уже видели некоторые из них (имя пользователя и его e-mail) в предыдущей главе. В отличие от данных файла и других метаданных репозитария, параметры конфигурации не переходят из одного репозитария в другой при клонировании. Вместо этого Git исследует конфигурацию и информацию об установке на основе пользователя, сайта, репозитария.

В репозитарии Git управляет двумя основными структурами данных - хранилищем объектов и индексом. Все эти данные репозитария хранятся в корне вашего рабочего каталога в скрытом подкаталоге с именем .git.

Хранилище объектов разработано для эффективного копирования при операции клонирования как часть механизма, полностью поддерживающего распределенный VCS. Индекс - это переходная информация, персональная для репозитария. Индекс может быть создан или изменен по требованию в случае необходимости.

В следующих двух разделах подробно рассмотрены хранилище объектов и индекс.

Типы объектов Git

Сердце репозитария Git - это хранилище объектов. Оно содержит ваши исходные файлы и все сообщения журналов, информацию об авторе, даты и другую информацию, необходимые для сборки любой версии или ветки проекта.

В хранилище объектов Git помещает объекты четырех типов: блобы (от англ. BLOB, Binary Large Object), деревья, фиксации и теги.

Блобы

Блобы — каждая версия файла представлена как BLOB. BLOB - это аббревиатура для «Binary Large Object», то есть большой двоичный объект. Этот термин часто используется в информатике для обозначения некоторой переменной или файла, которая (который) может содержать любые данные и внутренняя структура которой (которого) игнорируется программой. Блоб содержит данные файлы, но не содержит какие-либо метаданные о файле, даже его имя.

Деревья

Дерево представляет один уровень информации каталога. Оно записывает идентификаторы блобов, имена путей и немного метаданных для всех файлов в каталоге. Оно также может рекурсивно ссылаться на другие поддеревья. Деревья используются для построения полной иерархии файлов и подкаталогов.

Фиксации (Коммиты)

Объекты фиксации (коммиты) хранят метаданные для каждого изменения, произошедшего в репозитарии, включая имя автора, дату фиксации и сообщение журнала. Общепринятым сленгом является использование слова коммит для обозначения фиксации. Однако, в рамках данной книги мы будем использовать русский эквивалент, так как мы с вами говорим по-русски.

Каждая фиксация указывает на объект дерева, который захватывает в одном полном снимке состояние репозитария на момент осуществления фиксации. У начальной (корневой) фиксации нет родителя. У большинства обычных фиксаций есть одна родительская фиксация, хотя в главе 9 мы объясним, как фиксация может ссылаться на несколько родительских фиксаций.

Теги

Теги — объект тега назначает человекочитаемое имя определенному объекту, обычно фиксации. Хотя 9da581d910c9c4ac93557ca4859e767f5caf5169 позволяет точно идентифицировать фиксацию, для человека более удобным идентификатором является что-то вроде Ver-1.0-Alpha.

Со временем вся информация в хранилище объектов изменяется и растет, отслеживая и моделируя ваши правки проекта, добавления и удаления файлов. Чтобы эффективно использовать дисковое пространство и пропускную способность, Git сжимает объекты в pack-файлы, которые также помещаются в хранилище объектов.

Индекс

Индекс - временный и динамический двоичный файл, который описывает структуру каталогов всего репозитария. В частности, индекс получает версию полной структуры проекта в некоторый момент времени. Состояние проекта может быть представлено фиксацией и деревом.

Одна из ключевых отличительных функций Git - то, что он позволяет вам изменить содержание индекса четко определенными действиями. Индекс позволяет разделение между шагами поэтапной разработки и поддержкой тех изменений.

Вот как это работает. Как разработчик, вы выполняете команды Git, чтобы подготовить изменения в индексе. Изменения обычно включают добавление, удаление или редактирование файлов. Индекс записывает эти изменения и хранит их, пока вы не зафиксируете их. Вы можете также удалить или заменить изменения в индексе. Таким образом, индекс - это как мост между двумя сложными состояниями репозитария.

Как будет показано в главе 9, индекс играет важную роль в слияниях, позволяя управлять несколькими версиями одного и того же файла.

Ассоциативные имена

Хранилище объектов Git организовано и реализовано как ассоциативная система хранения. В частности каждому объекту в хранилище присваивается уникальное имя, применяя алгоритм SHA1 к содержанию объекта, что приводит к значению хэш-функции SHA1. Поскольку значение хэш-функции вычисляется на основании содержания объекта, полагают, что оно уникально для определенного содержания. Следовательно, у каждого объекта будет уникальное имя в базе объектов. Крошечное изменение в файле приведет к изменению хэша SHA1, в итоге новая версия файла будет индексироваться отдельно.

Значения SHA1 - 160-разрядные значения, которые обычно представляются как 40-разрядное шестнадцатеричное число, такое как 9da581d910c9c4a c93557ca4859e767f5caf5169. Иногда во время отображения значения SHA1 сокращаются до меньшего уникального префикса. Пользователи Git называют эти числа SHA1, хэш-код и идентификатор объекта - все эти понятия взаимозаменяемы.

Глобально уникальные идентификаторы

Важная характеристика вычисления хеша SHA1 - то, что он всегда вычисляет тот же ID для идентичного содержания, независимо от того, где это содержание находится. Другими словами, одно и то же содержание файла в разных каталогах и даже на разных машинах даст одно и то же значение хэша SHA1. Таким образом, идентификатор SHA1 - это глобально уникальный идентификатор.

Благодаря глобально уникальным идентификаторам мы можем через Интернет сравнивать блобы или файлы произвольного размера путем простого сравнения их идентификаторов. Нам не нужно пересылать файлы по сети, чтобы определить, идентичны ли они.

Git отслеживает контент

Git - это нечто больше, чем просто VCS. Git - это *система отслеживания контента*. Отслеживания контента в Git реализовано двумя способами, которые существенно отличаются от почти всех других систем управления версиями.

Во-первых, хранилище объектов Git основано на вычислении хэша содержимого объекта, а не на имени файла или каталога. Таким образом, когда Git помещает файл в хранилище, он это делает на основании данных хэша, а не на основании имени файла. Фактически, Git вообще не отслеживает имена файлов и каталогов, которые связаны с файлами вторичными способами. Git вместо имен файлов отслеживает их содержимое.

Если у двух отдельных файлов есть одинаковое содержание, независимо от того, хранятся ли они в одном каталоге или разным, Git хранит только одну копию этого содержания в виде блоба в хранилище объекта. Git вычисляет хэш-код каждого файла исключительно по его содержанию. Оба файла в проекте с одинаковым содержанием, независимо от того, где они расположены, используют один и тот же объект содержания (блоб).

Если один из этих двух файлов будет изменен, Git вычислит новый хэш SHA1 для него и добавит новый блоб в хранилище объектов. Исходный блоб останется в хранилище и будет использован для файла, который остался без изменений.

Во-вторых, внутренняя база данных Git хранит каждую версию каждого файла, а не разницу между ними. Поскольку Git использует хеш полного содержания файла как имя для этого файла, он должен оперировать с полной копией файла. Он не может основывать свою работу на части содержания файла или на разнице между версиями файла.

Типичное пользовательское представление файла в виде версий и изменений между версиями является обычным артефактом. Git вычисляет эту историю как ряд изменений между различными блобами вместо того, чтобы хранить имя файла и набор различий между версиями.

Путь по сравнению с содержанием

Как и в случае с другими VCS, Git должен вести явный список файлов, которые формируют содержание репозитария. Однако это не требует, чтобы внутренне Git основывался на именах файлов. Действительно, Git обрабатывает имя файла только как часть данные, которые отличны от содержания этого файла. В результате индекс отделяется от данных в традиционном смысле базы данных. Посмотрите на таблицу 4.1, которая сравнивает Git с другими известными системами.

Таблица 4.1. Сравнение баз данных

Система	Механизм индекса	Хранилище данных
Традиционная база данных	Индексно-Последовательный Метод Доступа (ISAM)	Записи данных
Файловая система UNIX	Каталоги (/путь/к/файлу)	Блоки данных
Git	.git/objects/hash, содержимое объекта дерева	Объекты блоб, объекты дерева

Названия файлов и каталогов происходят от базовой файловой системы, но Git действительно не заботится об именах. Он просто записывает каждый путь и удостоверяется, что он может точно воспроизвести файлы и каталоги из содержимого, которое индексировано значением хэша.

Физический формат данных не моделируется после пользовательской структуры каталога. Вместо этого есть абсолютно другая структура, которая может, тем не менее, воспроизвести оригинальную разметку пользователя. Внутренняя структура Git - более эффективная структура данных для своих собственных внутренних операций и средств хранения.

Когда Git нужно создать рабочий каталог, он говорит файловой системе: «Привет! У меня есть большой блоб данных, которые я хочу поместить в файл с именем /каталог/файл. Сделай это как посчитаешь нужным». На что файловая система отвечает: «Я распознала строку, являющуюся набором имен подкаталогов, и я знаю, куда поместить твои блоб- данные! Спасибо!».

Pack-файлы

Проницательный читатель может заметить: «Очень неэффективно хранить полное содержимое каждой версии каждого файла. Даже если содержимое сжато, все равно неэффективно хранить содержимое разных версий каждого файла. Почему, если добавляется всего одна строка в файл, Git сохраняет полное содержимое файла?».

Давайте попробуем разобраться. Git использует очень эффективный механизм хранения, называемый pack-файлом. Для создания упакованного файла Git сначала определяет местоположение файлов, содержание которых очень подобно и хранит полное содержание для одного из них. Затем он вычисляет различия или дельты между подобными файлы и хранит просто различия. Например, если вам нужно просто изменить или добавить одну строку в файл, Git может сохранить полную, более новую версия файла и затем записать изменение одной строки как дельта и тоже сохранить ее в пакете.

Хранение полной версии файла и дельт, необходимых для создания других версий подобных файлов - далеко не новый прием. Это, по существу, тот же механизм, которые другие VCS, такие как RCS, использовали на протяжении многих десятилетий.

Git очень умно упаковывает файл. Поскольку Git основывается на контенте файла, ему действительно все равно, принадлежат ли дельты, которые он вычисляет между двумя файлами двум версиям одного и того же файла или нет. То есть Git может взять два любых файла и вычислить дельты между ними, если он считает, что они могли бы быть достаточно подобными, чтобы быть хорошо сжатыми. Таким образом, Git обладает тщательно продуманным алгоритмом, чтобы найти потенциальных кандидатов дельты по всему репозитарию.

Упакованные файлы хранятся в хранилище объектов вместе с другими объектами. Они также используются для передачи данных репозитария по сети.

4.2. Изображения хранилища объектов

Давайте посмотрим как объекты Git работают вместе, чтобы сформировать полную систему. Объект блоб - это «низ» структуры данных, они ни на что не ссылаются, а на них ссылаются только объекты дерева. В изображениях, приведенных ниже, каждый блоб представлен в виде прямоугольника.

Объекты дерева указывают на блобы и, возможно, на другие деревья. На любой данный объект дерева могут указывать несколько разных объектов фиксации. Каждое дерево представлено треугольником.

Звезда представляет фиксацию. Фиксация указывает на одно конкретное дерево, которое введено в репозитарий фиксацией.

Каждый тег представлен параллелограммом. Один тег может указывать на одну фиксацию. Ветвление - не является фундаментальным объектом Git, но все же оно играет важную роль в именовании фиксаций. Каждое ветвление изображено как округленный прямоугольник.

Рис. 4.1 собирает все части вместе. Эта диаграмма показывает состояние репозитария после единственной, начальной фиксации добавления двух файлов. Оба файла находятся в каталоге верхнего уровня. Ветка называется master, а тег с именем V1.0 указывает фиксацию с ID 1112.

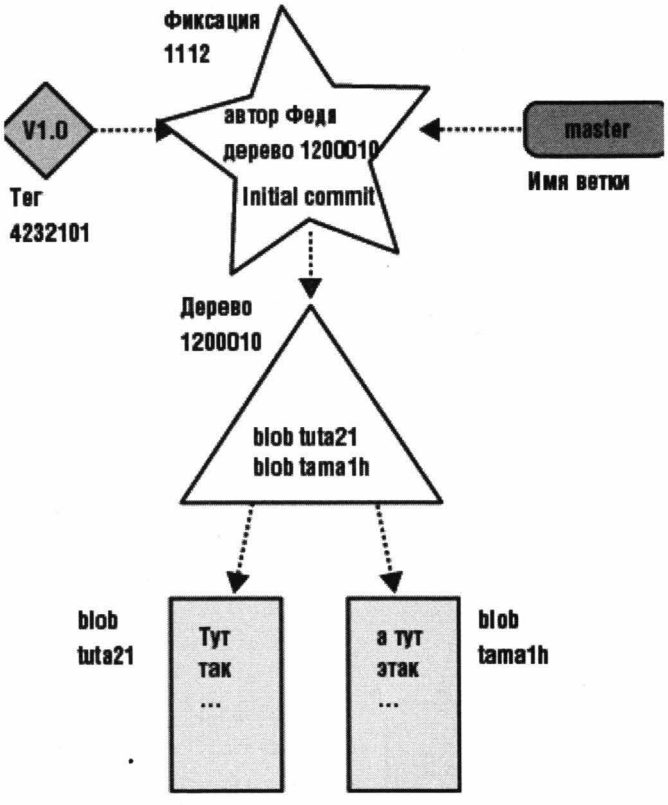


Рис. 4.1. Объекты Git

Теперь давайте сделаем вещи более сложными. Давайте оставим эти два исходных файла как есть и добавим новый подкаталогом с одним файлом в нет. После этого хранилище объектов будет напоминать представленное на рис. 4.2.

Как и в предыдущем случае, новая фиксация добавила один связанный объект дерева, чтобы представить общее состояние структуры файлов и каталогов. Поскольку каталог верхнего уровня был изменен добавлением нового подкаталога, содержимое объекта дерева верхнего уровня также было изменено, поэтому Git представляет новое дерево, bb10254.

Однако блобы tuta21 и tama1h не изменились во второй фиксации. Git понимает, что поскольку идентификаторы не изменились, то они могут совместно использоваться новым деревом bb10254, нет необходимости копировать эти объекты в новое дерево.

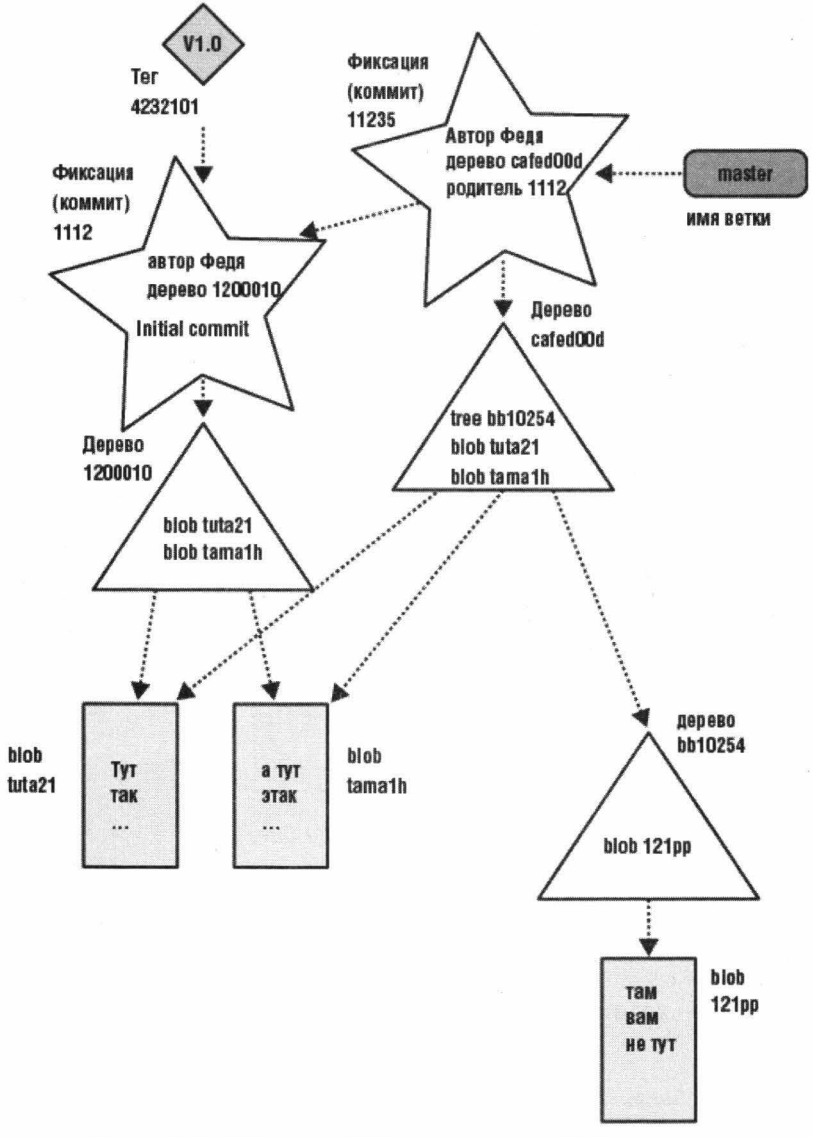


Рис. 4.2. Объекты Git после второй фиксации

Обратите внимание на направление стрелок между фиксациями. Каждая стрелка указывает обратно на своего родителя или родителей. Часто это выглядит запутывающим, поскольку состояние репозитария традиционно изображается в противоположном направлении: как поток данных от родительской фиксации до дочерних фиксаций.

В главе 6 мы подробно еще вернемся к этим изображениям, чтобы показать, как создается история репозитария.

4.3. Понятия Git в действии

Теперь, когда вы знакомы с некоторыми понятиями, давайте посмотрим, как они работают в самом репозитарии. Давайте создадим новый репозитарий и посмотрим на него внутренние файлы.

Внутри каталога .git

Для начала инициализируем пустой репозитарий, используя **git init**, а затем запустим команду **find**, чтобы посмотреть, что же было создано.

```
$ mkdir /tmp/hello
$ cd /tmp/hello
$ git init
Initialized empty Git repository in /tmp/hello/.git/

# Выводим все файлы в текущем каталоге
$ find .
.
./.git
./.git/hooks
./.git/hooks/commit-msg.sample
./.git/hooks/applypatch-msg.sample
./.git/hooks/pre-applypatch.sample
./.git/hooks/post-commit.sample
./.git/hooks/pre-rebase.sample
./.git/hooks/post-receive.sample
./.git/hooks/prepare-commit-msg.sample
./.git/hooks/post-update.sample
./.git/hooks/pre-commit.sample
./.git/hooks/update.sample
./.git/refs
./.git/refs/heads
./.git/refs/tags
./.git/config
./.git/objects
```

```
./.git/objects/pack
./.git/objects/info
./.git/description
./.git/HEAD
./.git/branches
./.git/info
./.git/info/exclude
```

Как видите, в каталоге .git много чего есть. Файлы, выведенные на экран, основаны на каталоге шаблона, который вы можете при желании изменить. В зависимости от используемой версии Git содержимое этого каталога может немного изменяться. Например, более старые версии Git добавляют «расширение» .sample к файлам в каталоге .git/hooks.

В целом, вам не нужно ни просматривать, ни управлять файлами из каталога .git. Эти «скрытые» файлы считаются частью инфраструктуры или конфигурации Git. Конечно, в Git есть набор команд по управлению этими скрытыми файлами, но вы будете редко их использовать.

Изначально каталог .git/objects (каталог для всех объектов Git) пуст, за исключением нескольких заполнителей:

```
$ find .git/objects
.git/objects
.git/objects/pack
.git/objects/info
```

Теперь давайте осторожно создадим простой объект:

```
$ echo "hello world" > hello.txt
$ git add hello.txt
```

Если вы введете «hello world» точно так же (без изменений в регистре символов или интервале между ними), ваш каталог объектов будет выглядеть примерно так:

```
$ find .git/objects
.git/objects
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

Все это выглядит довольно таинственным. Но на самом деле все просто и в следующем разделе вы поймете почему.

Объекты, хэши, блобы

Когда Git создает объект для hello.txt, он не беспокоится об имени файла hello.txt. Его интересует только то, что в файле: последовательность из 12 байтов - строка «hello world» и символ новой строки. Git выполняет несколько операций на этом блобе, вычисляет его хеш SHA1 и помещает в хранилище объектов как файл, в качестве имени файла используется вычисленное значение хэша.

Откуда мы знаем, что хэш SHA1 уникален?

Есть чрезвычайно небольшой шанс, что два различных блоба приводят к тому же хэшу SHA1. Когда это происходит, эту ситуацию называют коллизией. Однако, коллизия SHA1 так маловероятна, что практически никогда не происходит.

SHA1 - "криптографически безопасный хэш". До недавнего времени не было никакого известного способа (кроме слепой удачи) специально вызвать коллизию. Но ведь коллизия могла произойти непроизвольно? Давайте посмотрим.

Учитывая 160 бит, у вас есть 2^{160} или 10^{48} (1 с 48 нулями после) возможных SHA1 хэшей. Это число просто огромно. Даже если бы вы наняли триллион человек, чтобы произвести триллион новых уникальных блобов в секунду в течение триллиона лет, у вас было бы приблизительно 10^{43} блоба.

Если бы вы прохэшировали 280 случайных блобов, вы бы могли найти коллизию. Не верите нам? Пойдите почитайте Брюса Шнайера.

Хэш в этом случае - 3b18e512dba79e4c8300dd08aeb37f8e728b8dad. 160 битов хэша SHA1 соответствуют 20 байтам, а при отображении на экране в шестнадцатеричном виде - 40 байтов. Итак, наш контент сохранен как .git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad.

Git вставляет / после первых двух разрядов, чтобы повысить эффективность файловой системы. Производительность некоторых файловых систем заметно падает, если поместить в один каталог слишком много файлов. А если превратит первый SHA1 байт в каталог, то это самый простой способ

создать фиксированное разделение пространства имен для всех возможных объектов с равным распределением.

Чтобы показать, что Git действительно ничего не сделал с содержанием файла (это все еще строка «hello world»), вы можете использовать хэш файла, чтобы получить доступ к содержимому в любой момент:

```
$ git cat-file -p 3b18e512dba79e4c8300dd08aeb37f8e728b8dad
hello world
```

Примечание. Git также знает, что 40 символов немного рискованно, чтобы их вводить вручную, поэтому он предоставил команду для поиска объектов по префиксу хэша:

$ git rev-parse 3b18e512d

3b18e512dba79e4c8300dd08aeb37f8e728b8dad

Файлы и деревья

Теперь, когда блоб «hello world» безопасно помещен в хранилище объектов, что произошло с его именем файла? Git был бы не очень полезен, если он не мог бы найти файлы по имени.

Как было упомянуто выше, Git отслеживает пути файлов через другой вид объектов - деревья. Когда вы используете **git add**, Git сздает объект для содержания каждого добавленного вами файла, но он не создает объект для вашего дерева сразу же. Вместо этого он обновляет индекс. Индекс хранится в .git/index и используется для отслеживания пути файла и соответствующих блобов. Каждый раз, когда вы выполняете команды вроде git add, git rm или git mv, Git обновляет индекс, устанавливая новую информацию блоба и пути файла.

Создать объект дерева из вашего текущего индекса можно с помощью низкоуровневой команды git write-tree.

Для просмотра содержимого индекса введите следующую команду (на момент ее ввода индекс содержал только один файл - hello.txt):

```
$ git ls-files -s
100644 3b18e512dba79e4c8300dd08aeb37f8e728b8dad 0 hello.txt
```

Здесь мы видим, что файл hello.txt соответствует блобу 3b18e5.... Далее, давайте получим состояние индекса и сохраним его как объект дерева:

```
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60
$ find .git/objects
.git/objects
.git/objects/68
.git/objects/68/aba62e560c0ebc3396e8ae9335232cd93a3f60
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

Теперь у нас есть два объекта: объект «hello world» с ID 3b18e5 и новый объект, объект дерева, с ID 68aba6. Как видите, имя объекта соответствует каталогу и файлу в каталоге .git/objects.

Но как выглядит само дерево? Поскольку дерево - это тоже объект, подобно блобу, вы можете использовать ту же команду для его просмотра:

```
$ git cat-file -p 68aba6
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad hello.
txt
```

Содержимое объекта просто интерпретировать. Первое число - 100644 - представляет атрибуты файла в восьмеричной системе, если вы работали с Unix, то вы с ними знакомы. Далее идет имя (3b18e5) объекта, а hello.txt - имя файла, связанное с блобом.

Примечание относительно использования SHA1

Перед более подробным рассмотрением содержимого объекта дерева, давайте посмотрим на очень важную функцию SHA1-хэшей:

```
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60
$ git write-tree
```

```
68aba62e560c0ebc3396e8ae9335232cd93a3f60
```

Каждый раз, когда Вы вычисляете другой объект дерева для того же индекса, хэш SHA1 остается неизменным. Git не должен воссоздать новый объект дерева. Если вы вводите эти команды на своем компьютере, то вы должны увидеть те же хэши, что и приведенные в этой книге.

Хэш-функция - истинная функция в математическом смысле: для заданного ввода она всегда производит один и тот же вывод.

Это чрезвычайно важно. Например, если вы создаете то же самое содержание как другой разработчик, независимо от того, где или когда или как вы оба работаете, идентичный хэш - доказательство полной идентичности содержимого.

Но задержитесь на секунду. Разве SHA1 хэши не являются уникальными? Что произошло с триллионами людей с триллионами блобов в секунду, которые никогда не произведут одну единственную коллизию? Это частый источник недоразумений среди новых пользователей Git. Внимательно продолжайте читать далее, поскольку если вы сможете понять эту разницу, тогда все остальное в этой главе - просто.

Идентичные хэши SHA1 в этом случае *не считаются коллизией*. Коллизия - это если два разных объекта производят один и тот же хэш. Здесь же мы создали два отдельных экземпляра одного и того же содержимого, поэтому хэш одинаковый (у одного и того же контента всегда будет одинаковый хэш).

Git зависит от другого последствия хеш-функции SHA1: не имеет значения, *как* вы получили дерево, названное 68aba62e560c0ebc3396e8ae9335232cd93 a3f60. Если оно есть у вас, вы можете быть полностью уверены, что это - тот же древовидный объект, который, скажем, есть у другого читателя этой книги. Блоб, возможно, создал дерево, комбинируя фиксации A и B от Дженни и фиксации C от Сергея, тогда как вы получили фиксацию A от Сью и обновление от Лакшми, которое комбинирует фиксации B и C. Результат - тот же, и это существенно упрощает распределенную разработку.

Если вас попросили найти объект 68aba62e560c0ebc3396e8ae9335232cd93a 3f60 и вы можете найти именно этот объект, поскольку SHA1 - криптографический хэш и вы можете быть уверены, что нашли именно те данные, по которым был создан хэш.

Также истинно: если вы нашли объект с определенным хэшем в вашем хранилище объектов, вы можете быть уверены, что у вас нет копии этого объекта. Хэш таким образом является надежной меткой или именем для объекта.

Но Git также полагается на что-то более важное, чем просто заключение. Рассмотрим самую последнюю фиксацию (или связанный с ней объект дерева). Поскольку она содержит, как часть ее контента, хэш ее родительский фиксаций и ее дерево содержит хэш всех его поддеревьев и блобов. А это означает, что хэш исходной фиксации однозначно определяет состояние целой структуры данных, которая основана на той фиксации.

Наконец, импликации нашего требования в предыдущем абзаце приводят к мощному использованию хеш-функции: она предоставляет эффективный способ сравнения двух объектов, даже очень больших и сложных структур данных без передачи этих объектов полностью.

Иерархия деревьев

Хорошо иметь информацию относительно одного файла, как было показано в предыдущем разделе, но проекты обычно более сложны и содержат глубоко вложенные каталоги, которые со временем перестраиваются и перемещаются. Давайте посмотрим, как Git обработает создание нового каталога, в который мы поместим полную копию файла hello.txt:

```
$ pwd
/tmp/hello
$ mkdir subdir
$ cp hello.txt subdir/
$ git add subdir/hello.txt
$ git write-tree
492413269336d21fac079d4a4672e55d5d2147ac
$ git cat-file -p 4924132693
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad hello.
txt
040000 tree 68aba62e560c0ebc3396e8ae9335232cd93a3f60 subdir
```

В новом корневом дереве теперь есть два элемента: исходный файл hello.txt и новый подкаталог subdir, который выводится как *tree*, а не как *blob*.

Что же тут необычного? Посмотрите на имя объекта subdir. Это наш старый друг - 68aba62e560c0ebc3396e8ae9335232cd93a3f60!

Новое дерево для subdir содержит только один файл, hello.txt и этот файл содержит старый контент - строку «hello world». Поэтому дерево subdir в глазах Git выглядит так же, как и корневой каталог.

Теперь давайте посмотрим на каталог .git/objects и посмотрим, на что повлияло это новое изменение:

```
$ find .git/objects
.git/objects
.git/objects/49
.git/objects/49/2413269336d21fac079d4a4672e55d5d2147ac
.git/objects/68
.git/objects/68/aba62e560c0ebc3396e8ae9335232cd93a3f60
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

У нас все еще есть три *уникальных* объекта: блоб со строкой «hello world»; дерево, содержащее hello.txt, в котором есть текст «hello world» плюс новая строка; второе дерево, которое содержит *другую* ссылку на hello.txt в первом дереве.

Фиксации

Следующий объект для обсуждения - *фиксации*. Теперь, когда hello.txt был добавлен с помощью **git add** и был произведен объект дерева командой **git writer-tree**, вы можете создать объект фиксации, используя следующие команды:

```
$ echo -n "Commit a file that says hello\n" \
| git commit-tree 492413269336d21fac079d4a4672e55d5d2147ac
3ede4622cc241bcb09683af36360e7413b9ddf6c
```

Результат будет примерно таким:

```
$ git cat-file -p 3ede462
tree 492413269336d21fac079d4a4672e55d5d2147ac
author Федя Колобков <kolobok@lobok.com> 1220233277 -0500
committer Федя Колобков <kolobok@lobok.com> 1220233277 -0500
```

```
Commit a file that says hello
```

Если вы выполняете примеры из этой книге на своем компьютере, вы, вероятно, обнаружили, что у объекта фиксации, сгенерированного вами, будет другое имя, которое отличается от приведенного в книге. Если вы до сих пор понимали все написанное, причина должна быть очевидной: это не та фиксация. Ведь фиксация содержит ваше имя и время создания фиксации, а эти данные в вашем случае будут отличаться. С другой стороны, у вашей фиксации есть то же дерево, что и в примерах.

Это то, почему объекты фиксации отделяются от их объектов дерева: разные фиксации часто относятся к одному и тому же дереву. Когда это происходит, Git достаточно умен, чтобы передать только новый объект фиксации, который обычно крошечный, вместо копирования всего дерева и всех блоб-объектов, которые, наверняка, гораздо больше.

В реальной жизни вы можете (и должны) вызвать команды **git write-tree** и **git commit-tree**, и потом просто использовать команду git commit. Вам не нужно помнить все команды, чтобы быть счастливым пользователем Git.

Основной объект фиксации довольно прост и содержит следующее:

- Имя объекта дерева, который фактически идентифицирует связанные файлы.
- Имя человека, создавшего новую версию (author) и время создания этой версии.
- Имя человека, который поместил новую версию (commiter) в репозитарий и время фиксации.
- Описание версии (сообщение о фиксации).

По умолчанию *author* и *commiter* - это один и тот же человек. Но существуют ситуации, когда это разные люди.

Примечание. Вы можете использовать команду git show --pretty=fuller для просмотра дополнительной информации о заданной фиксации.

Объекты фиксации также хранятся в виде структуры графа, хотя эта структура полностью отличается от структур, которые используются объектами дерева. Когда вы производите новую фиксацию, вы можете указать одну

или больше родительских фиксаций. Подробно о фиксациях мы поговорим в главе 6.

Теги

Наконец, мы добрались до последнего объекта, которым управляет Git - тег. Хотя в Git реализован тег только одного вида, на самом деле существует два вида тегов - легковесный и аннотированный.

Легковесный тег - это просто ссылка на объект фиксации и обычно он частный для репозитария. Эти теги не создают постоянный объект в хранилище объектов. Аннотированный тег более существенный и создает объект, содержащий сообщение, предоставленное вами и может содержать цифровую подпись, созданную с помощью GPG-ключа согласно RFC4880.

Git обрабатывает обрабатывает как легковесные, так и аннотированные теги, но по умолчанию большинство тегов Git работают только с аннотированными тегами, поскольку считают их «постоянными» объектами.

Создать аннотированный тег с сообщением можно командой **git tag**:

```
$ git tag -m "Tag version 1.0" V1.0 3ede462
```

Увидеть объект тега можно командой git cat-file -p, но как узнать хэш SHA1 объекта тега? Чтобы найти его, используйте совет из «Объекты, хэши и блобы»:

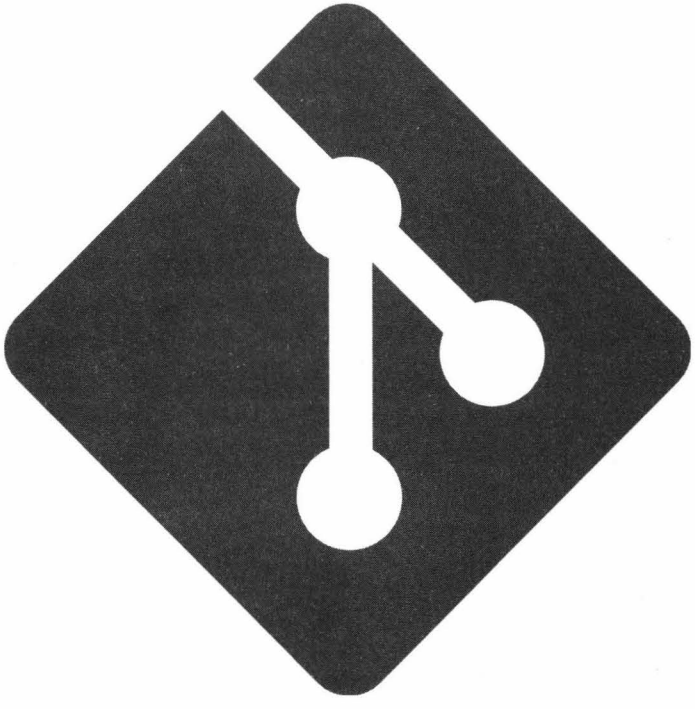
```
$ git rev-parse V1.0
6b608c1093943939ae78348117dd18b1ba151c6a
$ git cat-file -p 6b608c
object 3ede4622cc241bcb09683af36360e7413b9ddf6c
type commit
tag V1.0
tagger Федя Колобков <kolobok@lobok.com> Sun Oct 26 17:07:15 2020 -0500
Tag version 1.0
```

В дополнение к сообщению журнала и информации об авторе тег ссылается на объект фиксации (3ede462).

Git обычно тегирует объект фиксации, указывающий на объект дерева, охватывающий общее состояние всей иерархии файлов и каталогов в вашем репозитарии.

Вспомните рис. 4.1: тег V1.0 указывает на фиксацию с именем 1492, которая в свою очеред указывает на дерево 8675309, а оно уже охватывает много разных файлов. Таким образом, тег одновременно применяется ко всем файлам того дерева.

Поведение Git не похоже на другие CVS, где тег применяется отдельно к каждому файлу, а затем на основании коллекции этих тегированных файлов воссоздается целая тегированная версия. Другие CVS позволяют вам перемещать теги отдельных файлов, а Git требует создания новой фиксации, которая охватывает изменение состояния файла, тег которого был перемещен.



Глава 5.

Управление файлами. Индекс



Когда ваш проект на попечении системы контроля версий, вы редактируете файлы проекта в своем рабочем каталоге, а потом передаете свои изменения в ваш репозитарий для сохранности. Git работает также, но в нем есть промежуточный уровень между рабочим каталогом и репозитарием - индекс. Индекс используется для подготовки, или организации, изменений. Когда вы управляете своим кодом с помощью Git, вы редактируете изменения в своем рабочем каталоге, накапливаете их в своем индексе, а затем фиксируете то, что накопилось в индексе как один набор изменений.

Вы можете думать об индексе, как о ряде намеченных или предполагаемых модификаций. Вы добавляете, удаляете, перемещаете или редактируете файлы до точки фиксации, которая реализовывает накопленные файлы в репозитарии. Большая часть важной работы фактически предшествует шагу фиксации.

Примечание. Помните, что фиксация - двухступенчатый процесс. Сначала вы подготавливаете изменения, а потом вы фиксируете изменения. Изменение, найденное в рабочем каталоге, но не в индексе, не подготовлено и не может фиксироваться.

Для удобства Git позволяет вам комбинировать эти два шага, когда вам нужно добавить или изменить файл:

$ git commit index.html

Но если вы переместили или удалили файл, у вас не будет такой роскоши. Вам нужно будет выполнить два действия:

$ git rm index.html

$ git commit

Эта глава объясняет, как управлять индексом. Она описывает, как добавлять и удалять файлы из вашего репозитария, как переименовать файл и т.д.

5.1. Все об индексе

Линус Торвальдс утверждал в списке рассылки Git, что вы не можете осознать всю мощь Git без понимания назначения индекса.

Индекс Git не содержит названий файла, он просто отслеживает то, что вы хотите фиксировать. Когда вы выполняете фиксацию, Git проверяет индекс, а не ваш рабочий каталог (полностью фиксация будет рассмотрена в главе 6).

Несмотря на то, что многие высокоуровневые команды Git разработаны, чтобы скрыть детали индекса, все еще важно помнить об индексе и его состоянии.

Вы можете запросить состояние индекса в любое время командой **git status**. Она явно отображает файлы, подготовленные для фиксации (находящиеся в индексе). Также вы можете узнать о внутреннем состоянии Git командами вроде git ls-files.

Вероятно, вам пригодится команда **git diff** (подробно она описана в главе 8). Эта команда может вывести на экран два различных набора изменений: git diff выводит изменения, которые есть в рабочем каталоге, но которых нет в индексе; git diff -cached выводит изменения, которые уже подготовле-

ны (находятся в индексе) и будут помещены в репозитарий при следующей фиксации.

5.2. Классификация файлов в Git

Git классифицирует ваши файлы на три группы: отслеживаемые, игнорируемые и неотслеживаемые.

Отслеживаемые

Отслеживаемым называется файл, уже находящийся в репозитарии или в индексе. Чтобы добавить файл somefile в эту группу, выполните команду **git add somefile**.

Игнорируемые

Игнорируемый файл должен быть явно объявлен невидимым или игнорируемым в репозитарии (даже при том, что он может присутствовать в вашем рабочем каталоге). В вашем проекте может быть много игнорируемых файлов: ваши личные заметки, сообщения, временные файлы, вывод компилятора и большинство файлов, сгенерированных автоматически во время сборки проекта. Git по умолчанию ведет список игнорируемых файлов, но вы можете настроить свой репозитарий для распознания других игнорируемых файлов. Чуть позже будет показано, как это сделать.

Неотслеживаемые

Неотслеживаемым является любой файл, не относящийся к любой из предыдущих двух категорий. Git просматривает весь набор файлов в вашем ра-

бочем каталоге и вычитает из него отслеживаемые и игнорируемые файлы, в результате получается список неотслеживаемых файлов.

Давайте исследуем разные категории файлов, создав совершенно новый рабочий каталог и репозитарий:

```
$ cd /tmp/my_stuff
$ git init

$ git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to
track)

$ echo "Новые данные" > data

$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
# (use "git add <file>..." to include in what will be
committed)
#
# data
nothing added to commit but untracked files present (use
"git add" to track)
```

Изначально нет никаких отслеживаемых или игнорируемых файлов, поэтому набор неотслеживаемых файлов пуст. Как только вы создадите данные, **git status** покажет один неотслеживаемый файл.

Редакторы и окружения сборки часто оставляют временные файлы среди вашего исходного кода. Такие файлы обычно не должны отслеживаться, как файлы исходного кода. Чтобы Git игнорировал файлы в каталоге, просто добавьте имя этого файла в специальный файл .gitignore.

```
# Вручную создадим файл, который будет игнорироваться
$ touch main.o
```

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
# (use "git add <file>..." to include in what will be
committed)
#
# data
# main.o

$ echo main.o > .gitignore

$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
# (use "git add <file>..." to include in what will be
committed)
#
# .gitignore
# data
```

Как видите, файл main.o был проигнорирован, но **git status** показал новый неотслеживаемый файл .gitignore. Хотя этот файл имеет специальное назначение в Git, он обрабатывается как любой другой обычный файл в пределах вашего репозитария. Пока .gitignore не будет добавлен, Git будет рассматривать его как неотслеживаемый.

Следующие разделы демонстрируют несколько способов изменения статуса отслеживаемого файла, а именно добавление файла в индекс и удаление его из индекса.

5.3. Использование git add

Команда **git add** организует файл, после следующей фиксации (git commit) такой файл будет добавлен в репозитарий. В терминологии Git файл будет

неотслеживаемым, пока он не будет добавлен командой git add, что изменит его статус на отслеживаемый. Если команде git add передать имя каталога, все файлы и подкаталоги этого каталога будут добавлены рекурсивно.

Давайте продолжить пример из предыдущего раздела:

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
# (use "git add <file>..." to include in what will be
committed)
#
# .gitignore
# data
# Track both new files.

$ git add data .gitignore

$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
# (use "git rm --cached <file>..." to unstage)
#
# new file: .gitignore
# new file: data
#
```

Первая команда **git status** показывает, что у нас есть два неотслеживаемых и напоминает, что для того, чтобы сделать файл отслеживаемым, нужно просто использовать команду git add. После выполнения команды git add оба файла (data и .ignore) будут отслеживаться и будут подготовлены для помещения в репозитарий при следующей фиксации.

В терминах объектной модели Git добавление файла командой git add означает копирование файла в хранилище объектов и его индексирование. Организацию файла (git add) так же называют «кэшированием файла» или «помещением файла в индекс».

Вы можете использовать команду **git ls-files** для определения хэш-кодов для организованных файлов:

```
$ git ls-files --stage
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0 .gitignore
100644 534469f67ae5ce72a7a274faf30dee3c2ea1746d 0 data
```

Большинство ежедневных изменений в вашем репозитарии, вероятно, будут простыми правками. После редактирования и перед фиксацией изменений нужно запустить git add для обновления индекса, чтобы занести в него последнюю и самую высокую версию вашего файла. Если вы это не сделаете, у вас будут две разные версии файла: одна из хранилища объектов, на которую ссылается индекс, и другая - в вашем рабочем каталоге.

Чтобы продолжить пример, давайте изменим данные файлы так, чтобы они отличались от тех, которые есть в индексе. После сего командой git hash-object <файл> вычислим и отобразим хэш новой версии файла.

```
$ git ls-files --stage
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0
.gitignore
100644 534469f67ae5ce72a7a274faf30dee3c2ea1746d 0 data
# отредактируйте файл "data"...
$ cat data
Новые данные
Еще немного данных
$ git hash-object data
e476983f39f6e4f453f0fe4a859410f63b58b500
After the file is amended, the previous version of the file
in the object store and index
has SHA1 534469f67ae5ce72a7a274faf30dee3c2ea1746d. However,
the updated version
of the file has SHA1
e476983f39f6e4f453f0fe4a859410f63b58b500. Let's update the
index to contain the new version of the file:
$ git add data
$ git ls-files --stage
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0
.gitignore
100644 e476983f39f6e4f453f0fe4a859410f63b58b500 0 data
```

Теперь индекс содержит обновленную версию файла. Снова, «**данные** файла были подготовлены (для фиксации)» или, проще говоря, «**данные** файла были *помещены в индекс*». Последняя фраза не очень точная, но зато более понятная.

Опция **--interactive** для команд git add или git commit может быть полезна, чтобы узнать, какие файлы вы бы хотели подготовить для фиксации.

5.4. Некоторые замечания относительно использование git commit

Использование git commit -all

Опция *-a* или *--all* команды **git commit** заставляет Git автоматически подготавливать все неподготовленные, прослеженные изменения файла, в том числе удаление отслеживаемых файлов из рабочей копии, перед осуществлением фиксации.

Давайте посмотрим, как это работает, установив несколько файлов с разными характеристиками подготовки:

```
# Устанавливаем тестовый репозитарий
$ mkdir /tmp/commit-all-example
$ cd /tmp/commit-all-example
$ git init
Initialized empty Git repository in /tmp/commit-all-
example/.git/
$ echo something >> ready
$ echo somthing else >> notyet
$ git add ready notyet
$ git commit -m "Setup"
[master (root-commit) 71774a1] Setup
2 files changed, 2 insertions(+), 0 deletions(-)
create mode 100644 notyet
create mode 100644 ready
# Изменяем файл «ready» и добавляем командой «git add» его
# в индекс
```

```
$ git add ready
# Изменяем файл «notyet»
# редактируем «notyet»
# Добавляем новый файл в подкаталог, но не добавляем его в
репозитарий
$ mkdir subdir
$ echo Nope >> subdir/new
```

Используем **git status**, чтобы просмотреть изменения, требующие фиксации:

```
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# modified: ready
#
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
#
# modified: notyet
#
# Untracked files:
# (use "git add <file>..." to include in what will be
committed)
#
# subdir/
```

Здесь индекс подготовлен для фиксации только одного файла - с именем **ready**, поскольку только этот файл был подготовлен (добавлен).

Однако, если вы запустите команду git commit -all, Git рекурсивно обойдет весь репозитарий, организует все известные, измененные файлы. В этом случае, когда ваш редактор представит шаблон сообщения фиксации, он должен указать, что измененный и известный файл notyet тоже будет фиксироваться:

```
# Please enter the commit message for your changes.
# (Comment lines starting with '#' will not be included)
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
```

```
# modified: notyet
# modified: ready
#
# Untracked files:
# (use "git add <file>..." to include in what will be
committed)
#
# subdir/
```

Наконец, поскольку каталог с именем subdir/ новый и в нем не находится ни один из файлов, даже опция --all не заставит его фиксироваться:

```
Created commit db7de5f: Some --all thing.
2 files changed, 2 insertions(+), 0 deletions(-)
```

Git рекурсивно обойдет репозитарий в поисках измененных и файлов. Полностью новый каталог subdir/ и всего его файлы не станут частью фиксации.

Написания сообщений журнала фиксации

Если в командной строке вы явно не указываете сообщение журнала, Git запустит редактор и предложит вам написать его. Будет запущен установленный в вашей конфигурации редактор. Как установить редактор, было показано в главе 3.

Если вы выйдете из редактора без сохранения, Git будет использовать пустое сообщение журнала. Если вы уже сохранили сообщение, но еще не вышли из редактора, вы можете удалить сообщение и опять сохранить - тогда Git тоже сохранит пустое сообщение фиксации.

5.5. Использование git rm

Команда **git rm**, как и ожидается, обратна для git add. Она удаляет файл из репозитария и рабочего каталога. Однако удаление файла может быть более проблематичным (если что-то пойдет не так), чем добавление файла. Поэтому Git относится к удалению файла с большей осторожностью.

Git может удалить файл только из индекса или одновременно из индекса и рабочего каталога. Git не может удалить файл только из рабочего каталога, для этого используется команда операционной системы **rm**.

Удаление файла из каталога и из индекса не удаляет историю файла из репозитария. Любая версия файла до момента удаления хранится в хранилище объектов и не будет оттуда удалена.

Давайте представим, что у нас есть некоторый файл, который мы еще не добавили в репозитарий (не выполнили команду git add), и мы пытаемся удалить его командой **git rm**:

```
$ echo «Random stuff» > oops
# Не можем удалить файл, не добавленный в репозитарий
# Нужно использовать команду «rm oops»
$ git rm oops
fatal: pathspec 'oops' did not match any files
```

Теперь давайте добавим файл командой **git add**, а затем выполним команду **get status**:

```
# Добавляем файл
$ git add oops
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
# (use "git rm --cached <file>..." to unstage)
#
# new file: .gitignore
# new file: data
# new file: oops
#
```

Чтобы конвертировать файл из подготовленного в неподготовленный, используйте команду git rm --cached:

```
$ git ls-files --stage
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0
.gitignore
100644 e476983f39f6e4f453f0fe4a859410f63b58b500 0 data
```

```
100644 fcd87b055f261557434fa9956e6ce29433a5cd1c 0 oops
$ git rm --cached oops
rm 'oops'
$ git ls-files --stage
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0
.gitignore
100644 e476983f39f6e4f453f0fe4a859410f63b58b500 0 data
```

Обратите внимание: git rm --cached удаляет файл только из индекса, но оставляет его в рабочем каталоге, в то время как команда **git rm** удаляет файл, как из индекса, так и с рабочего каталога.

> **Примечание. Использование команды git rm --cached делает файл неотслеживаемым, в то время как его копия остается в рабочем каталоге. Это довольно опасно, так как вы можете забыть о нем, и файл больше никогда не будет отслеживаемым. Будьте осторожны!**

Если вы хотите удалить файл, как только он был зафиксирован, просто отправьте запрос через команду git rm <файл>:

```
$ git commit -m "Add some files"
Created initial commit 5b22108: Add some files
2 files changed, 3 insertions(+), 0 deletions(-)
create mode 100644 .gitignore
create mode 100644 data
$ git rm data
rm 'data'
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# deleted: data
#
```

Перед удалением файла Git проверяет, соответствует ли версия файла в рабочем каталоге последней версии в текущем ответвлении (в версии, которую команды Git называют HEAD). Эта проверка устраняет случайную потерю любых изменений, которые, возможно, были сделаны в файле.

Примечание. Используйте команду git rm -f для принудительного удаления вашего файла. В этом случае файл будет удален, даже если он был изменен с момента последней фиксации.

Если вам нужно сохранить файл, который вы случайно удалили, просто добавьте его снова:

```
$ git add data
fatal: pathspec 'data' did not match any files
```

Ошибочка вышла! Git ведь удалил и рабочую копию тоже! Но не волнуйтесь, VCS содержит отличный механизм восстановления старых версий файлов:

```
$ git checkout HEAD -- data
$ cat data
Новые данные
Еще немного новых данных
$ git status
# On branch master
nothing to commit (working directory clean)
```

5.6. Использование git mv

Предположим, что вам нужно удалить или переименовать файл. Вы можете использовать комбинацию команд git rm (для удаления старого файла) и git add (для добавления нового файла). Или же вы можете использовать непосредственно команду **git mv**. Представим, что в нашем репозитарии есть файл с именем stuff и вы хотите переименовать его в newstuff. Следующие две последовательности действий являются эквивалентными:

$ mv stuff newstuff

$ git rm stuff

$ git add newstuff

или

$ git mv stuff newstuff

В обоих случаях Git удалит путь stuff из индекса, добавит новый путь newstuff, сохраняя оригинальное содержимое stuff в хранилище объектов и реассоциирует это содержимое с путем newstuff.

Файл data мы уже восстановили, теперь давайте его переименуем в mydata:

```
$ git mv data mydata
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# renamed: data -> mydata
#

$ git commit -m "Переместил данные в хранилище"
Created commit ec7d888: Переместил данные в хранилище
1 files changed, 0 insertions(+), 0 deletions(-)
rename data => mydata (100%)
```

Если вы проверите историю файла, вы можете быть немного удивлены, когда увидите, что Git, очевидно, специально потерял историю исходного файла и помнит, только, что данные были переименованы:

```
$ git log mydata
commit ec7d888b6492370a8ef43f56162a2a4686aea3b4
Author: Федя Колобков <kolobok@lobok.com>
Date: Sun Nov 2 19:01:20 2020 -0600
Moved data to mydata
```

Git все еще помнит всю историю, но отображает только то, что касается определенного имени файла, указанного в команде. Опция --follow просит Git отследить журнал и найти всю историю, связанную с контентом:

```
$ git log --follow mydata
commit ec7d888b6492370a8ef43f56162a2a4686aea3b4
Author: Федя Колобков <kolobok@lobok.com>
Date: Sun Nov 2 19:01:20 2020 -0600

Moved data to mydata

commit 5b22108820b6638a86bf57145a136f3a7ab71818
```

```
Author: Федя Колобков <kolobok@lobok.com>
Date: Sun Nov 2 18:38:28 2020 -0600

Add some files
```

Одна из классических проблем многих VCS заключается в том, что после переименования файла невозможно отследить его историю. В Git эта проблема решена.

5.7. Замечание относительно отслеживания переименований

Давайте немного подробнее рассмотрим отслеживание переименований файла.

SVN, как пример традиционного управления версиями, производит большую работу по отслеживанию во время переименования/перемещения файла, поскольку она отслеживает только разницу между файлами. Если вы перемещаете файл, то это, по сути, то же, что и удаление всех строк из старого файла, и их добавление в новый файл. Но передавать все содержание файла каждый раз, когда вы делаете простое переименование, очень неэффективно. Подумайте о том, что будет, если нужно переименовать целый подкаталог с тысячами файлов.

Чтобы облегчить эту ситуацию, SVN явно отслеживает каждое переименование. Если вы хотите переименовать hello.txt в subdir/hello.txt, вы должны использовать команду **svn mv** вместо команды svn rm и svd add. Иначе SVN никак не поймет, что это переименование и ему придется пойти по неэффективному пути удаления/добавления, что и было описано выше.

Затем, учитывая эту исключительную функцию отслеживания переименования, SVN нуждается в специальном протоколе, чтобы сказать его клиентам: «переместите файл hello.txt в subdir/hello.txt». Кроме того, каждый клиент SVN должен убедиться, что выполнил эту работу правильно.

Git, с другой стороны, не отслеживает переименование. Вы можете переместить или скопировать hello.txt куда угодно, но это влияет только на объекты дерева. Помните, что объекты дерева хранят отношения между содержимым, тогда как само содержимое хранится в блобах. Посмотрев на разницу между двумя деревьями, становится очевидным, что блоб с именем 3b18e5 переместился в новое места.

В этой ситуации, как и во многих других, система хранения Git, основанная на хэше, упрощает много вещей по сравнению с другой RCS.

5.8. Проблемы с отслеживанием переименования

Отслеживание переименования файла порождает постоянные дебаты среди разработчиков VCS.

Простое переименование - объект разногласия. Аргументы становятся еще более весомыми, когда изменяется и имя, и содержимое файла. Тогда сценарий переговоров переходит от практического к философскому. Что это: переименование или новый файл (раз у него другое содержимое и другое имя)? Насколько новый файл подобен старому? Если вы применяете чей-то патч, который удаляет файл и воссоздает подобный в другом месте, как это обрабатывать? Что произойдет, если файл переименован двумя различными способами на двух разных ветках? Какая тактика менее подвержена ошибкам: используемая в Git или в SVN?

В реальной жизни, похоже, что система отслеживания переименований, используемая в Git, очень хороша, поскольку есть много способов переименовать файл и человек может просто забыть уведомить SVN об этом. Но помните, что нет идеальной системы для обработки переименований, к сожалению.

5.9. Файл .gitingnore

Ранее в этой главе было показано, как использовать файл .gitignore для игнорирования файла main.o. Чтобы проигнорировать любой файл, просто добавьте его имя в файл .gitignore, который находится в этом же каталоге. Вы также можете игнорировать файлы где угодно, добавив его в файл .gitignore, который находится в корневом каталоге вашего репозитария.

Но Git предоставляет более богатый механизм. Файл .gitignore может содержать список шаблонов имен файлов, указывающий, какие файлы нужно игнорировать. Формат .gitignore следующий:

- Пустые строки игнорируются, как и строки, начинающиеся с решетки (#). Такие строки можно использовать для комментариев, однако символ # не представляет комментарий, если он не является первым в строке.
- Обычные имена файлов соответствуют файлу в любом каталоге с указанным именем.
- Имя каталога отмечается с помощью слеша (/). Это правило соответствует любому каталогу или любому подкаталогу, но не соответствует файлу или символической ссылке.
- Шаблон может содержать маски оболочки, такие как звездочка (*). Звездочка может соответствовать единственному имени файла или каталога. Также звездочка может быть частью шаблона, включающего наклонные черты для обозначения имен каталогов, например, debug/32bit/*.o.
- Восклицательный знак (!) инвертирует смысл шаблона оставшейся части строки. Дополнительно, любой файл, исключенный предшествующим образцом, но соответствующий этому правилу инверсии, будет включен. У инверсии более высокий приоритет.

Кроме того, Git позволяет вам создавать файл .gitignore в любом каталоге вашего репозитария. Каждый такой файл влияет на свой каталог и все подкаталоги. Правила .gitignore каскадные: вы можете переопределить правила в каталоге более высокого уровня, включив инвертированный шаблон (с использованием ! в начале правила) в одном из подкаталогов.

Приоритет игнорирования следующий:

- Шаблоны, определенные в командной строке

- Шаблоны, прочитанные из файла .gitignore, находящего в том же каталоге
- Шаблоны в родительских каталогах. Шаблоны, находящиеся в текущем каталоге, будут перезаписывать шаблоны родительского каталога, следовательно, шаблоны более близкого родителя перезапишут шаблоны родителя более высокого уровня
- Шаблоны из файла .git/info/exclude
- Шаблоны из файла, указанного переменной конфигурации core. excludefile.

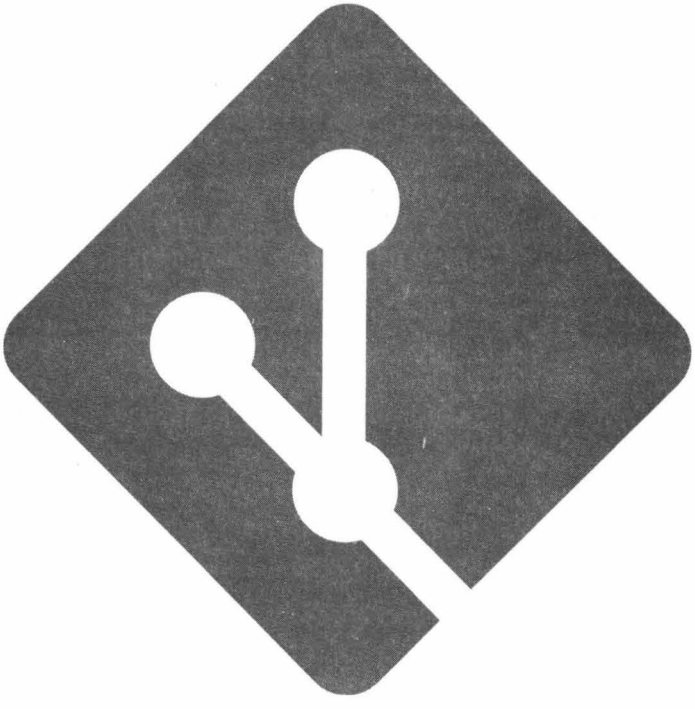
Поскольку.gitignore обрабатывается как обычный файл в вашем репозитарии, он копируется во время операций клонирования и применяется ко всем копиям вашего репозитария.

Если образец исключения, так или иначе определенный для одного вашего репозитария, не должен применяться к какому-нибудь клону этого репозитария, то шаблоны должны находиться в файле .git/info/exclude, поскольку он не копируется во время операций клонирования. Формат его шаблонов полностью совпадает с форматом файла .gitignore.

Рассмотрим другой сценарий. Нам нужно исключить файлы *.o, сгенерированные компилятором из исходного кода. Для игнорирования этих файлов поместите шаблон *.o в файл .gitignore самого верхнего уровня. Но что если в определенном каталоге у вас есть определенный .o файл, который нужно отследить? Тогда у вас может быть примерно эта конфигурация:

```
$ cd my_package
$ cat .gitignore
*.o
$ cd my_package/vendor_files
$ cat .gitignore
!driver.o
```

Данная конфигурация игнорирует все .o файлы в репозитарии, но Git будет отслеживать одно исключение - файл driver.o в подкаталоге vendor_files.



Глава 6.

Коммиты (фиксации)



В Git коммит (или *фиксация)* используется для записи изменений в репозитарий.

На первый взгляд кажется, что фиксация Git не отличается от фиксации в другой VCS. Но на самом деле фиксация Git работает своим собственным, уникальным образом.

Когда происходит фиксация, Git записывает снимок индекса и помещает этот снимок в хранилище объектов (подготовка индекса для фиксации была рассмотрена в главе 5). Данный снимок не содержит копию каждого файла и каталога в индексе, поскольку бы такая стратегия потребовала бы огромных затрат дискового пространства. Вместо этого Git сравнивает текущее состояние индекса с предыдущим и получает список измененных/добавленных/удаленных файлов и каталогов. Git создает новые блобы для каждого измененного файла и новые деревья для каждого измененного каталога. Если же файл/каталог не изменился, используются уже имеющиеся блоб/дерево.

Снимки фиксации объединяются в цепочку, в которой каждый новый снимок указывает на своего предшественника. Со временем последовательности изменений могут быть представлены как серия фиксаций.

Вам может показаться очень «дорогим» (в плане системных ресурсов) сравнивать индекс с некоторым предшествующим состоянием. Но все же про-

цесс удивительно быстр, потому что у каждого объекта Git есть SHA1-хэш. Если у двух объектов, даже у двух поддеревьев, один и тот же SHA1-хэш, значит, объекты идентичны. Git может избежать ряда рекурсивных сравнений, сокращая поддеревья, у которых то же самое содержимое.

Есть взаимно-однозначное соответствие между рядом изменений в репозитарии и фиксацией. Фиксация - единственный способ представления изменений в репозитарии, и любое изменение в репозитарии должно быть представлено фиксацией. Этот мандат обеспечивает отслеживаемость. Ни при каком обстоятельстве данные репозитария не должны изменяться без записи изменения! Просто вообразите хаос, если содержимое в главном репозитарии было изменено и при этом небыло бы никакой записи о том, как это произошло, кто внес изменения или почему.

Хотя фиксации чаще всего производятся явно, то есть разработчиком, Git может самостоятельно произвести фиксацию. В главе 9 мы поговорим о таких неявных фиксациях при слиянии репозитариев.

Когда именно фиксировать изменения - в значительной степени ваше дело и зависит от ваших предпочтений или стиля разработки. В целом вы должны выполнить фиксацию в четко определенные моменты времени, когда ваша разработка достигает определенного этапа, например, когда было выполнено тестирование, когда все идут домой после рабочего дня и еще по огромному числу других причин.

Git нормально относится к частым фиксациям и обеспечивает богатый набор команд для управления ними. Позже вы увидите, как несколько фиксаций, каждая с небольшими, но четко определенными изменениями, могут привести к лучшей организации изменений и более простому манипулированию наборами патча.

6.1. Атомарные наборы изменений

Каждая фиксация Git представляет единственный, атомарный набор изменений относительно предыдущего состояния. Независимо от числа каталогов, файлов, строк или байтов, которые изменяются фиксацией, или применяются все изменения, или ни одно.

С точки зрения базовой объектной модели атомарность просто целесообразна: снимок фиксации представляет общий набор измененных файлов и каталогов. Он должен представлять одно состояние дерева или другое, а набор изменений между этими двумя снимками состояний представляет полное преобразование дерево-дерево (разницы между фиксациями описаны в главе 8).

Рассмотрим поток операций по перемещению функции из одного файла в другой. Если вы выполняете удаление одной фиксацией, а добавление - другой фиксации, то останется небольшой «семантический разрыв» в истории вашего репозитария, вовремя которого пропала функция. Две фиксации в обратном порядке тоже проблематичны. В любом случае перед первой фиксацией и после второй ваш код семантически непротиворечивый, но после первой фиксации код является дефектным.

Данную проблему можно решить с помощью атомарной фиксации, которая одновременно удаляет и добавляет функцию

Однако атомарная фиксация одновременно удаляет и добавляет функцию без всякого семантического разрыва в истории репозитария. О том, как лучше организовать ваши фиксации, будет написано в главе 10.

Git все равно, почему изменяются файлы. То есть содержание изменений не имеет значения. Как разработчик, вы можете переместить функцию из одного файла в другой и ожидать, что эта операция будет обработана как одно унитарное перемещение. Но, альтернативно, вы можете фиксировать удаление и чуть позже - добавление. При этом Git будет все равно, ведь он не имеет никакого отношения к семантике того, что находится в файлах.

Атомарность позволяет вам структурировать свои фиксации и в конечном счете вы можете быть уверенными, что Git не оставил ваш репозитарий в некотором переходном состоянии между одним снимком фиксации и другим.

6.2. Идентификация фиксаций

Программируете ли вы самостоятельно или в составе команды очень важно идентифицировать отдельные фиксации. Например, для создания ветвле-

ния вы должны выбрать фиксацию, с которой можно начать; для сравнения двух вариантов кода нужно указать две фиксации; для редактирования истории фиксации нужно предоставить набор фиксаций. В Git вы можете обратиться к каждой фиксации по явной или неявной ссылки.

Вы уже видели явные ссылки и несколько неявных ссылок. Уникальный, 40-разрядный шестнадцатеричный SHA1-идентификатор - это явная ссылка, а HEAD, которая всегда указывает на наиболее последнюю фиксацию, это пример неявной ссылки. Однако, ни одна из ссылок не является удобно. К счастью, Git обеспечивает много разных механизмов именования фиксаций, каждый из которых более или менее полезен в той или иной ситуации.

Например, при обсуждении определенной фиксации с коллегой, работающим над теми же данными, но в распределенной в среде, лучше использовать имя фиксации, которое гарантировано будет одинаковым в обоих репозиториях. С другой стороны, при работе со своим локальным репозитарием проще использовать относительное имя.

Абсолютные имена фиксации

Самое строгое имя фиксации - ее идентификатор хэша. ID хэша - это абсолютное имя, которое может относиться только к одной фиксации. Не имеет значение, где в репозитарии находится эта фиксация, ID хэша всегда уникально и идентифицирует одну и ту же фиксацию.

Каждый идентификатор фиксации *глобально* уникален, не только для одного репозитария, а для всех репозитариев. Например, если разработчик пишет вам и ссылается на определенный ID фиксации в его репозитарии и при этом в своем репозитарии вы находите фиксацию с таким же ID, это значит, что у вас обоих есть одинаковая фиксация с одинаковым содержимым.

Поскольку 40-разрядное шестнадцатеричное SHA1-число утомительно и подвержено ошибкам, Git позволяет вам сокращать это число до уникального префикса в пределах базы данных объектов репозитария. Рассмотрим пример из собственного репозитария Git:

```
$ git log -1 --pretty=oneline HEAD
1fbb58b4153e90eda08c2b022ee32d90729582e6 Merge git://repo.
or.cz/git-gui
```

```
$ git log -1 --pretty=oneline 1fbb
error: short SHA1 1fbb is ambiguous.
fatal: ambiguous argument '1fbb': unknown revision or path
    not in the working tree.

Use '--' to separate paths from revisions

$ git log -1 --pretty=oneline 1fbb58
1fbb58b4153e90eda08c2b022ee32d90729582e6 Merge git://repo.
or.cz/git-gui
```

Несмотря на то, что имя тега не является глобально уникальным именем, в данном случае оно абсолютно указывает на уникальную фиксацию и не изменяется в течение долгого времени (если вы явно не измените его).

Ссылки и символьные ссылки

Ссылка - это SHA1-хэш, который относится к объекту в хранилище объектов Git. Хотя ссылка может ссылаться на любой объект Git, она обычно ссылается на объект фиксации. Символическая ссылка - это имя, косвенно указывающее на объект Git. По сути, это тоже ссылка.

Локальные имена ветки, удаленные имена ветки и имена тегов - все это ссылки.

У каждой символической ссылки есть явное, полное имя, которое начинается с refs/ и хранится в каталоге .git/refs/ репозитария. Обычно представлено три пространства имен в refs/: refs/heads/refs - локальные ответвления, refs/remotes/ref ваши удаленные ответвления и refs/tags/ref - ваши теги (ответвления подробно описаны в главах 7 и 12).

Например, локальное ответвление с именем dev является короткой формой refs/heads/dev. Удаленные ветки находятся в пространстве имен refs/remotes/, поэтому имя origin/master на самом деле соответствует имени refs/remotes/origin/master. А тег v2.6.23 является коротким именем для refs/tags/v2.6.23.

Вы можете использовать, как полное, так и короткое имена, но если у вас есть и ветвление и тег с одинаковыми именами, учтите, что Git будет искать совпадение с указанным именем в следующем порядке (см. страницу man для git rev-parse):

```
.git/ref
.git/refs/ref
.git/refs/tags/ref
.git/refs/heads/ref
.git/refs/remotes/ref
.git/refs/remotes/ref/HEAD
```

Первое правило обычно только для нескольких ссылок, описанных позже: HEAD, ORIG_HEAD, FETCH_HEAD, CHERRY_PICK_HEAD и MERGE_HEAD.

> **Примечание. Технически, имя каталога Git (.git) можно изменить. Поэтому документация Git использует переменную окружения $GIT_DIR вместо имени каталога .git.**

Рассмотрим несколько специальных символьных ссылок, которые Git использует для разных целей.

HEAD

Ссылка HEAD всегда указывает на самую последнюю фиксацию в текущей ветке. Когда вы изменяете ветки, HEAD обновляется так, чтобы она снова указывала на самую последнюю фиксацию, но уже в новой ветке.

ORIG_HEAD

Некоторые операции, например, слияние и сброс, записывают предыдущую версию HEAD в ORIG_HEAD. Вы можете использовать ORIG_HEAD для восстановления предыдущего состояния или сравнения.

FETCH_HEAD

При использовании удаленных репозитариев команда git fetch записывает «головы» всех веток, выбранных в файле .git/FETCH_HEAD. FETCH_HEAD - это короткое имя последней выбранной ветки и оно допустимо сразу после операции выборки. Подробно об этой ссылке мы поговорим в главе 12.

MERGE_HEAD

Указывает на верхушку ветки, с которой только что сделано слияние. Понятно, что данная ссылка имеет смысл только после процесса слияния.

Все эти символические ссылки управляются командой **git symbolic-ref**.

> **Примечание. Хотя вы можете создать собственную ветку с этими специальными именами ссылок (например, HEAD), это не очень хорошая идея.**

Существуют и другие специальные имена ссылок. Два таких имени, каретка (^) и тильда (~), используются чаще всего и будут описаны в следующем разделе.

Относительные имена фиксаций

Git также предоставляет механизмы для идентификации фиксации относительно другой ссылки.

Вы же видели некоторые из этих имен, например, master^, где master^ всегда относится к предпоследней фиксации на ветке master. Есть и другие формы: вы можете использовать master^^, master~2 и даже более сложную форму вроде master~10^2~2^2.

За исключением первой корневой фиксации, каждая фиксация получена, по крайней мере, из одной более ранней фиксации и, возможно, многих других предков, которые называются *родительскими фиксациями*. Если у фиксации есть несколько родительских фиксаций, значит, она - результат работы слияния. Ведь в результате будет одна родительская фиксация для каждой ветки.

Символ каретки используется для выбора родительской фиксации. Представим, что у нас есть фиксация C, тогда C^1 - это первая родительская фиксация, C^2 - вторая, C^3 - третья и т.д., см. рис. 6.1.

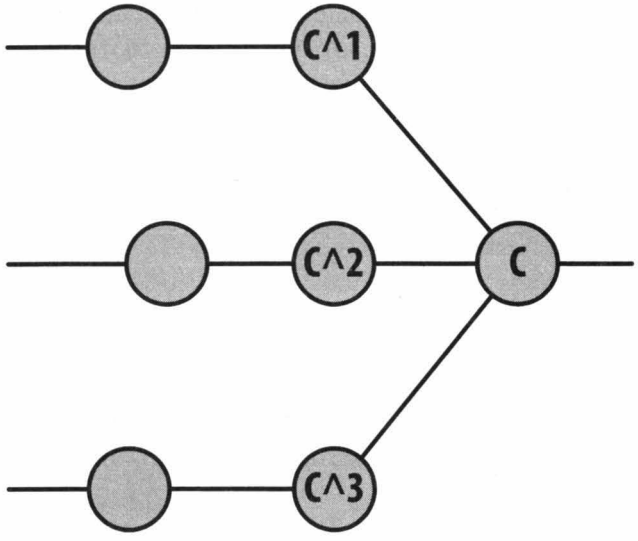


Рис. 6.1. Несколько родительских имен

Тильда используется для возврата назад и выбора предшествующего поколения: C~1 - первый родитель, C~2 - первый прародитель, C~2 - первый прапрародитель. Вы можете заметить, что для ссылки на первого родителя используется, как ссылка C^1, так и C~1. Обе ссылки верны, что и показано на рис. 6.2.

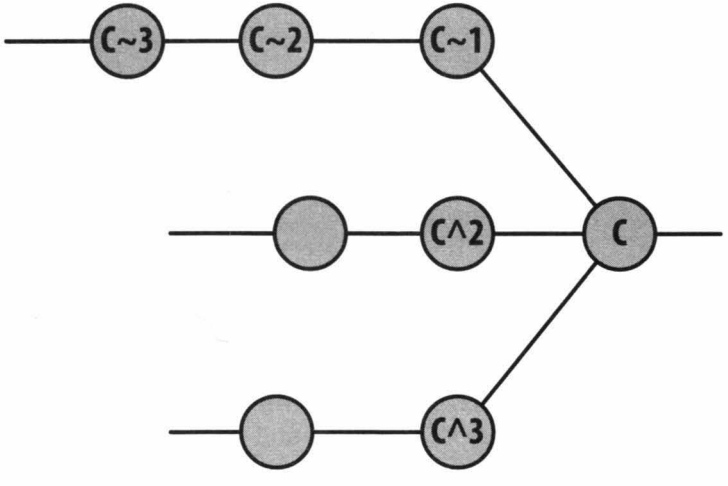


Рис. 6.2. Несколько родительских имен

Git позволяет использовать сокращения и комбинации имен ссылок. Например, C^ и C~ - это то же самое, что и C^1 и C~1 соответственно. C^^ - это то же самое, что и C^1^1, что означает «первый родитель первого родителя фиксации C» и то же самое, что и C~2.

Используя команды:

```
git show-branch --more=35
```

и ограничив вывод 10 последними строками, вы можете изучить историю графа и исследовать сложную структуру слияния ветвления:

```
$ git rev-parse master
1fbb58b4153e90eda08c2b022ee32d90729582e6

$ git show-branch --more=35 | tail -10
-- [master~15] Merge branch 'maint'
-- [master~3^2^] Merge branch 'maint-1.5.4' into maint
+* [master~3^2^2^] wt-status.h: declare global variables as
extern
-- [master~3^2~2] Merge branch 'maint-1.5.4' into maint
-- [master~16] Merge branch 'lt/core-optim'
+* [master~16^2] Optimize symlink/directory detection
+* [master~17] rev-parse --verify: do not output anything
on error
+* [master~18] rev-parse: fix using "--default" with
"--verify"
+* [master~19] rev-parse: add test script for "--verify"
+* [master~20] Add svn-compatible "blame" output format to
git-svn

$ git rev-parse master~3^2^2^
32efcd91c6505ae28f87c0e9a3e2b3c0115017d8
```

Между master~15 и master~16 имело место слияние, которое представляло несколько других слияний, а также простую фиксацию с именем master~3^2^2^. Это, оказывается, фиксация 32efcd91c6505ae28f87c0e9a3e2 b3c0115017d8.

Команда git rev-parse - окончательное решение при переводе любой формы фиксации - тега, относительной, сокращенной или абсолютной - в актуальный, абсолютный идентификатор в пределах базы данных объектов.

6.3. История фиксаций

Просмотр старых фиксаций

Основной командой для просмотра истории фиксаций является git log. У нее довольно много опций, параметров, средств форматирования, селекторов и прочих возможностей. Но не беспокойтесь! Вам совсем не обязательно знать все ее параметры.

Если команда git log вызвана без параметров, она работает как git log HEAD, то есть выводит сообщение журнала, связанное с каждой фиксацией в вашей истории, которая доступна из HEAD. Показанные изменения начинаются с фиксации HEAD и будут выведены в обратном хронологическом порядке. Но вспомните, что при перемещении по истории Git придерживается графика фиксации, а не времени.

Если вы введете команду git log <фиксация>, вывод начнется с указанной фиксации. Следующая форма команды полезна для просмотра истории ветки:

```
$ git log master

commit 1fbb58b4153e90eda08c2b022ee32d90729582e6
Merge: 58949bb... 76bb40c...
Author: Junio C Hamano <gitster@pobox.com>
Date: Thu May 15 01:31:15 2020 -0700

Merge git://repo.or.cz/git-gui

* git://repo.or.cz/git-gui:
  git-gui: Delete branches with 'git branch -D' to clear
config
  git-gui: Setup branch.remote,merge for shorthand git-pull
  git-gui: Update German translation
  git-gui: Don't use '$$cr master' with aspell earlier than
0.60
  git-gui: Report less precise object estimates for
database compression
```

```
commit 58949bb18a1610d109e64e997c41696e0dfe97c3
Author: Chris Frey <cdfrey@foursquare.net>
Date: Wed May 14 19:22:18 2020 -0400

Documentation/git-prune.txt: document unpacked logic

Clarifies the git-prune manpage, documenting that it only
prunes unpacked objects.

Signed-off-by: Chris Frey <cdfrey@foursquare.net>
Signed-off-by: Junio C Hamano <gitster@pobox.com>

commit c7ea453618e41e05a06f05e3ab63d555d0ddd7d9
...
```

Журналы авторитетные, но откат назад через всю историю фиксации вашего репозитария, вероятно, не очень практичен. Как правило, ограниченная история более информативна. Один из методов - ограничить историю определенным диапазоном фиксаций, используя диапазон *от...до*. Пример:

```
$ git log --pretty=short --abbrev-commit master~12..
master~10

commit 6d9878c...
Author: Jeff King <peff@peff.net>

clone: bsd shell portability fix

commit 30684df...
Author: Jeff King <peff@peff.net>

t5000: tar portability fix
```

Здесь git low показывает фиксации между master~12 и master~10 или между 10-ой и 11-ой фиксациями в ветке master. Подробно диапазоны фиксаций описаны в разделе «Диапазоны фиксаций» далее в этой главе.

Предыдущий пример также демонстрирует две опции форматирования: --pretty=short и --abbrev-commit. Первая выводит разную информацию о каждой фиксации, значение этой опции позволяет корректировать объем этой информации: short (короткая) и full (полная). Вторая просто запрашивает ID-хэш и сокращает его.

Используйте опцию **-p** для вывода патча или изменений, представленных фиксацией:

```
$ git log -1 -p 4fe86488
```

```
commit 4fe86488e1a550aa058c081c7e67644dd0f7c98e
Author: Федя Колобков <kolobok@lobok.com>
Date: Wed Apr 23 16:14:30 2020 -0500

Add otherwise missing --strict option to unpack-objects
summary.

Signed-off-by: Петя Перетятько <pere@mail.ru>
Signed-off-by: Хулио Альмадовар <hulio@mail.ru>

diff --git a/Documentation/git-unpack-objects.txt b/
Documentation/git-unpack-objects.txt
index 3697896..50947c5 100644
--- a/Documentation/git-unpack-objects.txt
+++ b/Documentation/git-unpack-objects.txt
@@ -8,7 +8,7 @@ git-unpack-objects - Unpack objects from a
packed archive

SYNOPSIS
--------
-'git-unpack-objects' [-n] [-q] [-r] <pack-file
+'git-unpack-objects' [-n] [-q] [-r] [--strict] <pack-file
```

Обратите внимание на опцию -1: она ограничивает вывод одной фиксацией. Вы также можете использовать опцию **-n**, которая ограничить вывод **n** фиксациями.

Опция *--stat* выводит файлы, измененные в фиксации, и сообщает, сколько строк было изменено в каждом файле.

```
$ git log --pretty=short --stat master~12..master~10
```

```
commit 6d9878cc60ba97fc99aa92f40535644938cad907
Author: Jeff King <peff@peff.net>

clone: bsd shell portability fix
```

```
 git-clone.sh | 3 +--
 1 files changed, 1 insertions(+), 2 deletions(-)

commit 30684dfaf8cf96e5afc01668acc01acc0ade59db
Author: Jeff King <peff@peff.net>

t5000: tar portability fix

 t/t5000-tar-tree.sh | 8 ++++----
 1 files changed, 4 insertions(+), 4 deletions(-)
```

Примечание. Сравните вывод git log --stat с выводом git diff --stat. Есть принципиальное различие в их выводе. Первая команда выводит сводку для каждой отдельной фиксации, а вторая - выводит единственную общую сводку.

Еще одна команда для отображения объектов из хранилища объектов - **git show**. Вы можете использовать ее для просмотра фиксации:

```
$ git show HEAD~2
```

или для просмотра определенного блоб-объекта:

```
$ git show origin/master:Makefile
```

Будет показан блоб Makefile из ветки с именем origin/master.

Графы фиксации

В главе 4 было представлено несколько рисунков, визуализирующих разметку и связи объектов в модели данных Git. Такие эскизы помогают разобраться, что к чему, особенно, если вы мало знакомы с Git. Однако даже маленький репозитарий, в котором есть ряд фиксаций, слияний и патчей, становится очень громоздким, чтобы представить его детально. Например, рис. 6.3 показывает более полный, но все еще несколько упрощенный граф фиксации. Вообразите, каким был бы граф, если бы на нем были отображены все фиксации и все структуры данных.

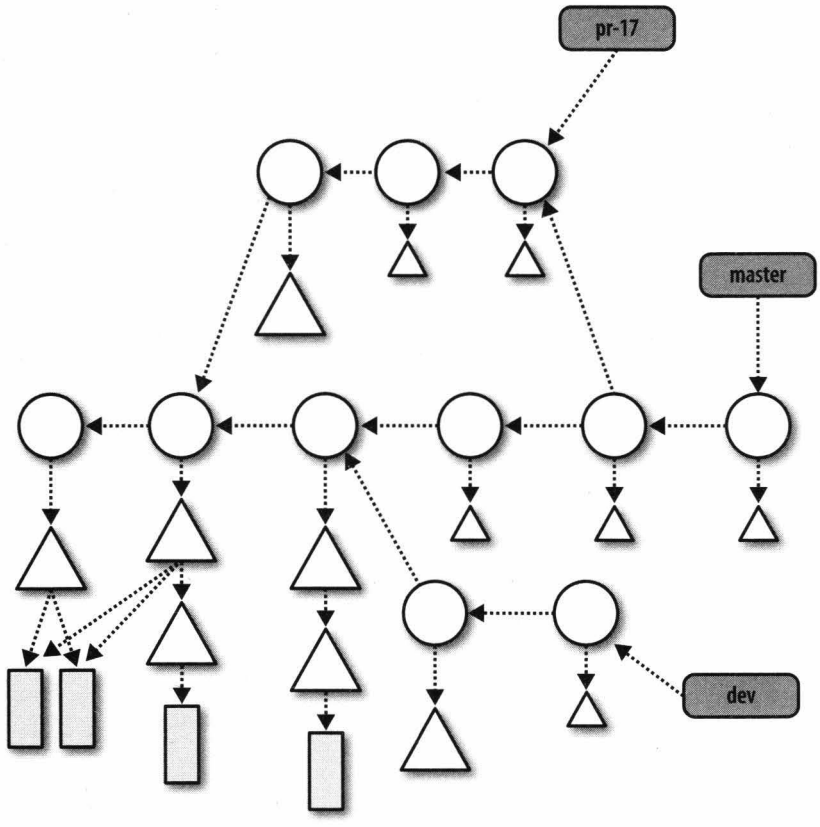


Рис. 6.3. Полный граф фиксации

Каждая фиксация представляет древовидный объект, который представляет весь репозитарий. Поэтому фиксация может быть изображена как просто имя.

Рисунок 6.4 показывает тот же самый граф фиксации, что и рисунок 6.3, но не изображая блоб-объекты и объекты дерева. Обычно с целью обсуждения или ссылки имена ветки также показывают в графах фиксации.

В области информатики граф - это набор вершин и ряда ребер (связей) между вершинами. Есть несколько типов графов с различными свойствами. Git использует специальный граф, названный направленным ациклическим графом (DAG, directed acyclic graph). У такого графа есть два важных свойства. Во-первых, ребра в пределах графа направлены от одной вершины к другой. Во-вторых, если начать с любой вершины графа, нет никакого пути, который бы позволил вернуться в исходную вершину.

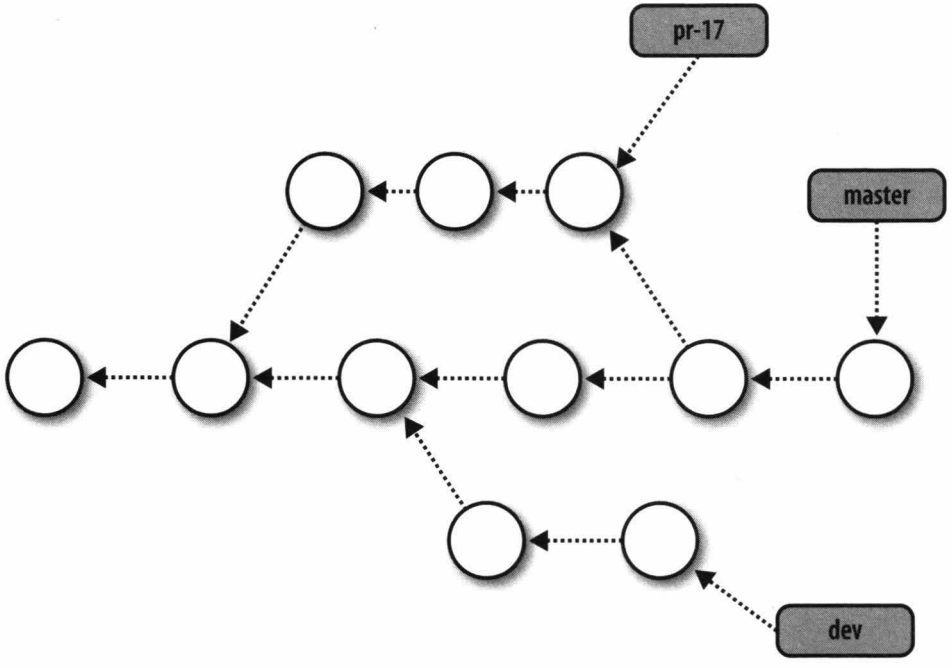


Рис. 6.4. Упрощенный граф фиксации

История фиксаций в пределах репозитария Git реализована как DAG. В графе фиксации каждая вершина - это одна фиксация и все ребра направлены от узла потомка к родительской вершине. Оба графа, которые вы видели на рис. 6.3 и 6.4 являются DAG-графами. Говоря об истории фиксаций и обсуждая отношения между фиксациями в графе, отдельные узлы фиксации часто отмечаются так, как показано на рис. 6.5.

На этих диаграммах время идет слева направо. A - это начальная фиксация, поскольку у нее нет родителя, фиксация B произошла после A. Обе фиксации, E и C, произошли после B, но нет никакого требования об относительной синхронизации между C и E. Фактически Git не беспокоится о времени и синхронизации (абсолютной и относительной) фиксаций. Помните, что реальное время фиксации может отличаться от реального, поскольку часы компьютера могут быть выставлены некорректно. В среде распределенной разработки эта проблема очень важна. Меткам времени нельзя верить. Для решения этой проблемы нужно, чтобы все участники распределенной разработки автоматически синхронизировали время с одним и тем же сервером времени.

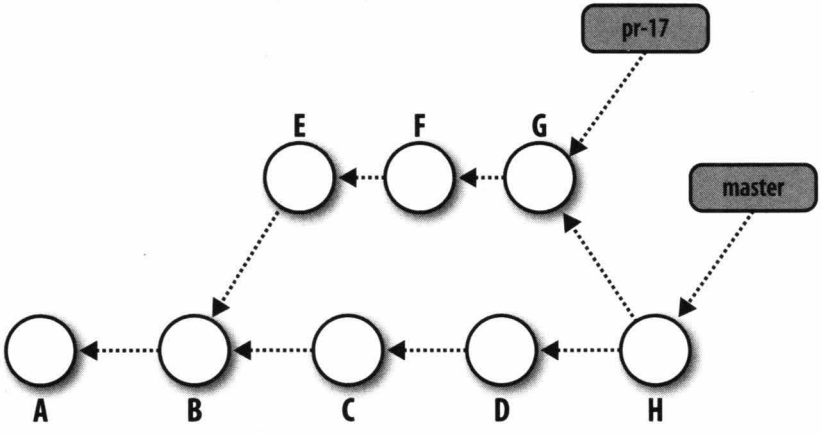


Рис. 6.5. Маркированный граф фиксации

У фиксаций E и C один и тот же родитель - B. Поэтому B считается источником ветки. Главная (master) ветка начинается фиксациями A, B, C и D. Тем временем последовательность фиксаций A, B, E, F и G формируют ветку с именем pr-17. Ветка pr-17 указывает на фиксацию G (подробно о ветках мы поговорим в главе 7).

Фиксация H - это фиксация слияния, где ветка pr-17 соединяется с веткой master. Поскольку это слияние, у H есть больше одного родителя - в этом случае два - D и G. После этой фиксации ветка master будет обновлена так,

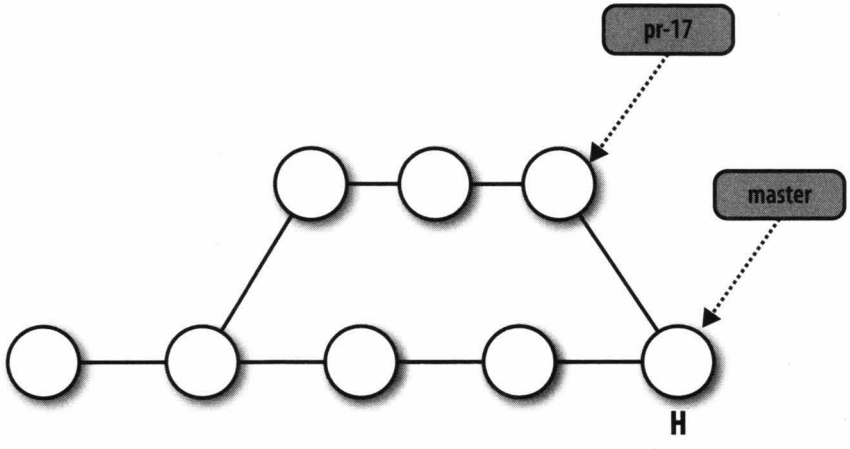


Рис. 6.6. Граф фиксации без стрелок

чтобы ссылаться на новую фиксацию H, но ветка pr-17 будет продолжать ссылаться на G (слияние более подробно описано в главе 9).

Практически, тонкости прошедших фиксаций считаются незначительными. Ссылка фиксации на своего родителя часто игнорируется, как показано на рис. 6.6.

Время все еще идет слева направо, показаны две ветки, но граф не направленный.

Использование gitk для просмотра графа фиксации

Назначение графа - помочь вам визуализировать сложную структуру и отношения. Команда gitk поможет вам нарисовать картину репозитария. Давайте посмотрим на наш веб-сайт:

```
$ cd public_html
$ gitk
```

Программа gitk много чего умеет, но давайте сфокусируемся сейчас на DAG. Граф будет примерно таким, как показано на рис. 6.7.

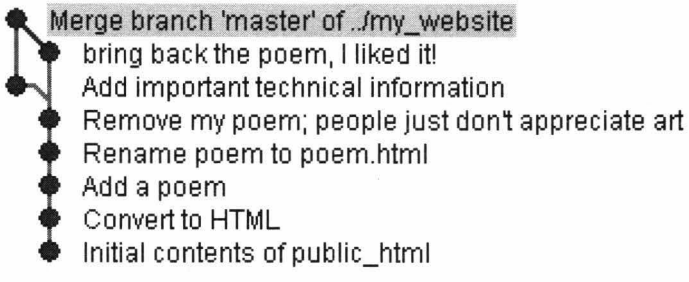


Рис. 6.7. Слияние, визуализированное программой gitk

Теперь о том, что вы должны знать, чтобы понять DAG фиксации. Прежде всего, у каждой фиксации есть 0 или больше *родителей*:

- У обычных фиксаций есть только один родитель - это предыдущая фиксация в истории. Когда вы делаете изменение, ваше изменение - это разница между вашей новой фиксацией и ее родителем.

- У исходной фиксации вообще нет родителей, она появляется внизу графа.
- У фиксации слияния (вроде той, которая сейчас на вершине графа) более одного родителя.

Фиксация, у которой есть более одного потомка - это место, где история формирует ветку. На рис. 6.7 фиксация Remove my poem - это точка ветки.

Диапазоны фиксаций

Множество команд Git позволяют вам указывать диапазон фиксаций. В самой простой форме диапазон фиксаций - это сокращение для серии фиксаций. Более сложные формы позволяют вам включать и исключать фиксации.

Диапазон указывается с помощью двоеточия, например, start..end, где start и end должны быть указаны, как описано в разделе «Идентификация фиксаций» ранее в этой главе. Обычно диапазон указывается для изучения ветки или части ветки.

В разделе «Просмотр старых фиксаций» вы увидели, как использовать диапазон фиксации для команды **git log**. В том примере использовался диапазон master~12..master~10 для указания 10-ой и 11-ой фиксации ветки master. Для визуализации этого диапазона давайте рассмотрим граф фиксации на рис. 6.8. Пусть у нас будет ветка M с несколькими фиксациями. Для простоты пусть история фиксации будет линейной.

Помните, что время идет слева направо, поэтому M~14 - это самая старая фиксация, а M~9 - самая последняя фиксация.

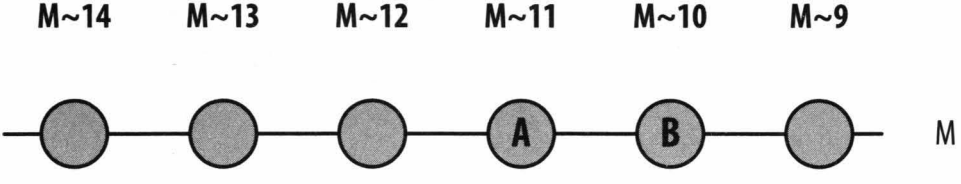


Рис. 6.8. Линейная история фиксации

Диапазон M~12..M~10 представлен двумя фиксациями - 11-ой и 10-ой, которые отмечены на рис. 6.8 как A и B. Диапазон не включает M~12. Почему? Все дело в определении. Диапазон фиксации, start..end, определен как набор фиксаций, достижима с конца (end), и недостижима с позиции start. Другими словами «фиксация end включена», в то время как «фиксация start не включена».

6.4. Достижимость в графах

В теории графов узел X является достижимым из узла A, если вы можете начать с узла A, «пойти» по дугам графам по определенным правилам и «дойти» в узел X. Набор достижимых узлов для узла A - это коллекция всех узлов, достижимых из A.

В графе фиксаций Git набор достижимых фиксаций - это те, которые вы можете достичь из заданной фиксации, обойдя по направленным родительским ссылкам.

Когда вы указываете фиксацию Y команде git log, вы запрашиваете Git показать журнал для всех фиксаций, которые достижимы из Y. Вы можете исключить определенную фиксацию X и все фиксации, достижимые из X с помощью выражения ^X.

Комбинирования двух форм, например, git log ^XY, аналогично команде git log X..Y и означает: «предоставь мне все фиксации, которые достижимы из Y и меня не интересуют любые фиксации, достижимые из X, включая саму фиксацию X».

Диапазон фиксации X..Y математически эквивалентен ^XY. Вы можете также думать о нем, как о вычитании: все до Y минус все до X, включая сам X.

Вернемся к серии фиксаций из примера выше. Здесь M~12..M~10 обозначает две фиксации - A и B. Начинаем со всего, что предшествует M~10, как показано в первой строке рис. 6.9. Найдите все, что предшествует и включает M~12, как показано во второй строчке рисунка. Осталось выполнить вычитание M~12 из M~10, чтобы получить фиксации, показанные в третьей строчке рисунка.

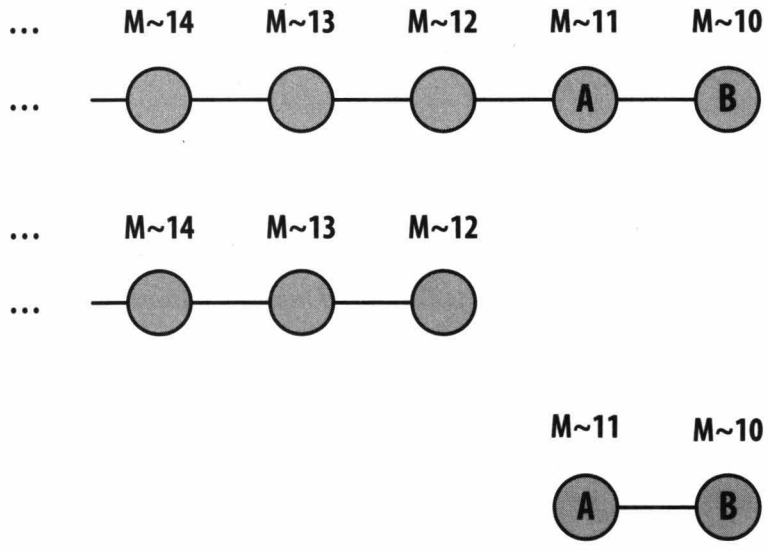


Рис. 6.9. Интерпретация диапазонов как вычитания

Когда история вашего репозитария - просто линейная серия фиксаций, предельно просто понять, как работают диапазоны. Но когда ветки или слияния появляются в графе, становится очень сложно понять, что к чему.

Давайте рассмотрим еще несколько примеров. На рис. 6.10 изображена ветка master с линейной историей, наборы B..E, ^BE и набор C,D и E аналогичны.

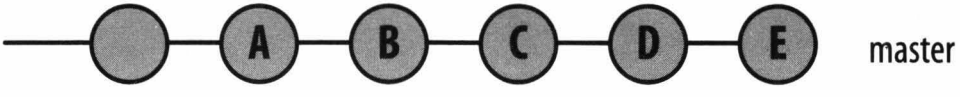


Рис. 6.10. Простая линейная история

Теперь посмотрим на рис. 6.11. На нем ветка **master** на фиксации V была объединена с веткой topic (фиксация B).

Диапазон topic..master представляет фиксации в master, но не в ветке topic. Поскольку слияния производится в фиксации V, то в слияния будут включены все фиксации, предшествующие V, в том числе сама V (то есть набор {..., T, U, V}), остальные фиксации будут исключены (W, X, Y, Z).

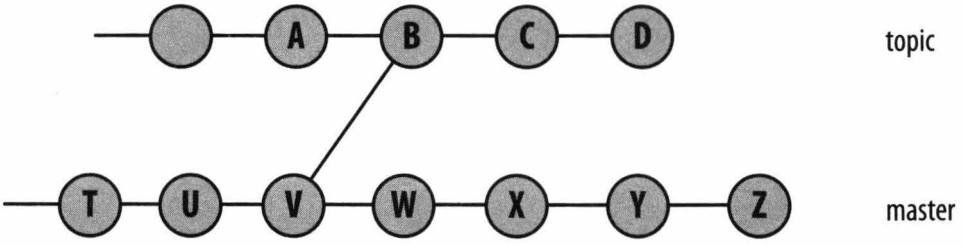


Рис. 6.11. Слияние master с topic

Обратный пример, то есть слияние ветки topic с веткой master, приведен на рис. 6.12.

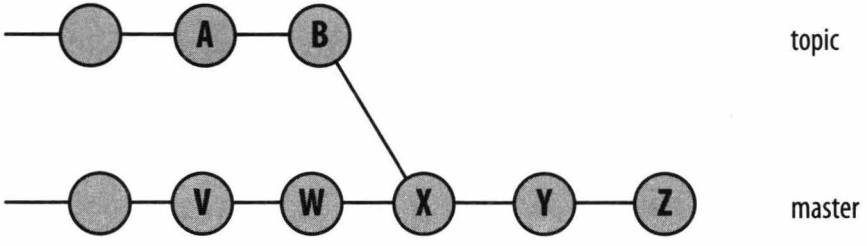


Рис. 6.12. Слияние topic с master

Однако, мы должны быть немного осторожными и рассмотреть полную историю ветки topic. Рассмотрим случай, где она первоначально отошла от ветки master и затем снова объединена с ней, как показано на рис. 6.13.

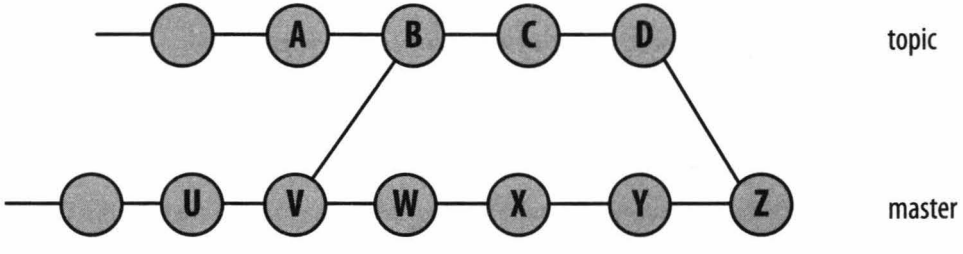


Рис. 6.13. Ветка и слияние

В этом случае topic..master содержит только фиксации W, X, Y и Z. Помните, что диапазон исключит все фиксации, которые достижимы из topic (то есть фиксации D, C, B, A и более ранние), как и фиксации V, U и ранние. Результат - только фиксации с W по Z.

Есть две других перестановки диапазона. Если вы опускаете одно из значений, start или end, вместо него подразумевается HEAD. Таким образом, ..end эквивалентно HEAD..end, а start.. эквивалентно start..HEAD.

Ранее уже было отмечено, что диапазон statr..end представляет операцию вычитания, а запись A...B (троеточие) представляет симметрическую разницу между A и B или набор фиксаций, которые доступны из A или B, но не доступны из обоих. Поскольку функция симметрична, первая и вторая фиксации не рассматриваются как начальная (start) и конечная (end). В этом смысле A и B эквивалентны.

Более формально, набор версий в симметрической разнице между A и B (A...B) можно получить командой:

```
$ git rev-list A B --not $(git merge-base --all A B)
```

Давайте посмотрим на пример, изображенный на рис. 6.14.

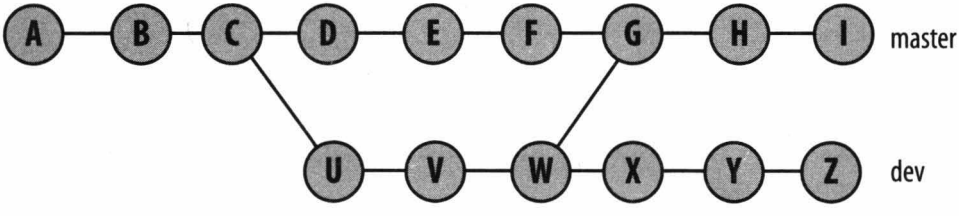


Рис. 6.14. Симметрическая разница

Мы можем вычислить каждую часть симметрической разницы так:

```
master...dev = (master OR dev) AND NOT (merge-base --all
master dev)
```

Фиксациями, которые будут внесены в master, являются (I, H, . . . , B, A, W, V, U). Фиксации, которые будут внесены в dev - (Z, Y, . . . , U, C, B, A).

Объединение этих двух множеств - (A, . . . , I, U, . . . , Z). База слияния между master и dev - это фиксация W. В более сложных случаях может быть не-

сколько баз слияния, но у нас есть только одна. Фиксации, которые будут внесены к W: (W, V, U, C, B и A); следующие фиксации являются общими для master и dev, поэтому они должны быть исключены для формирования симметрической разницы: (I, H, Z, Y, X, G, F, E, D).

Теперь вы знаете, что такое диапазоны фиксаций, как их записать и как они работают, что очень важно, если вы хотите эффективно использовать Git. Команды Git, кроме диапазонов, могут принимать произвольные последовательности фиксаций, например вы можете использовать:

```
$ git log ^dev ^topic ^bugfix master
```

для выбора фиксаций в ветке master, но не в dev, topic или bugfix.

Все эти примеры немного абстрактны, но они демонстрируют всю мощь диапазонов Git. К диапазонам фиксаций мы еще вернемся в главе 12, в которой будет описана ситуация, когда одна из ваших веток представляет фиксации из другого репозитария.

6.5. Поиск фиксаций

Часть хорошей RCS - это функция «археологии» или исследования репозитария. Git предоставляет несколько механизмов, помогающих вам найти фиксации, соответствующие определенным критериям в пределах вашего репозитария.

Использование git bisect

Команда **git bisect** - мощная утилита для изоляции определенной, дефектной фиксации на основе чрезвычайно произвольных критериев поиска. Инструмент подходит для ситуаций, когда вы обнаружили, что что-то «неправильное» или «плохое» влияет на ваш репозитарий и вы знаете, что без этого ваш код будет прекрасен. Скажем, вы работаете над ядром Linux и обнаружили сбой в начальной загрузке, но вы знаете, что начальная загрузка нормально работала ранее, скажем, на прошлой неделе. В этом случае ут-

верждают, что ваш репозитарий перешел из «хорошего» состояния к «плохому».

Но когда это произошло? Как найти те фиксации, которые повредили наш код? Git как раз и разработан для того, чтобы дать ответ на эти вопросы.

Команда **git bisect** часто используется для изоляции определенной фиксации, которая внесла ошибку в репозитарий. Например, если вы работаете с ядром Linux, git bisect может помочь вам найти проблемы и ошибки, такие как сбои компиляции, проблемы с загрузкой и другие проблемы. В этих случаях git bisect поможет вычислить фиксацию, которая вызвала проблему.

Прежде всего, нужно идентифицировать «хорошую» и «плохую» фиксации. На практике «плохая» версия - это часто ваша текущая HEAD, поскольку, работая с ней, вы обнаружили что что-то не то.

Найти хорошую версию немного сложнее, поскольку она находится где-то в дебрях вашей истории. Вероятно, вы помните, что некоторая версия в истории репозитария работала корректно. Возможно, вы помните ее тег, например, v2.6.25 или же вы помните, что данная фиксация была 100 фиксаций назад в ветке master, то есть master~100. Идеально, чтобы «хорошая» фиксация была ближе к «плохой», то есть master~25 лучше, чем master~100. Старайтесь не выбирать слишком «древние» фиксации. В любом случае вы или должны явно знать (например, когда вы знаете тег фиксации) или должны быть в состоянии проверить, что вы нашли именно «хорошую» фиксацию.

Важно запустить **git bisect** из чистого рабочего каталога. Процесс обязательно скорректирует ваш рабочий каталог, чтобы он содержал разные версии вашего репозитария. Запуск с «грязного» каталога чреват неприятностями; очень легко потерять ваш рабочий каталог.

Мы будем использовать клон ядра Linux и скажем Git начать поиск:

```
$ cd linux-2.6
$ git bisect start
```

После начала поиска Git перейдет в режим bisect, настраивая для себя некоторую информацию о состоянии репозитария. Git использует отсоединенную HEAD для управления текущей проверенной версией репозитария. Данная отсоединенная фиксация HEAD - по существу - отдельная ветка, которая может указывать на разные версии в случае необходимости.

Сразу скажите Git, какая фиксация «плохая». Повторюсь, на практике часто оказывается, что текущая версия является «плохой» (HEAD).

```
# Говорим git, что версия HEAD является плохой
$ git bisect bad
```

Аналогично, указываем «хорошую» (рабочую) версию:

```
$ git bisect good v2.6.27
Bisecting: 3857 revisions left to test after this
[cf2fa66055d718ae13e62451bb546505f63906a2] Merge branch 'for_
linus'
of git://git.kernel.org/pub/scm/linux/kernel/git/mchehab/
linux-2.6
```

Теперь ваше дело отвечать на вопрос: «Хорошая эта версия или плохая?». Каждый раз, когда вы отвечаете на этот вопрос, Git сужает пространство поиска на половину, идентифицирует новую версию, проверяет ее и повторяет вопрос.

Представим, что версия является хорошей:

```
$ git bisect good
Bisecting: 1939 revisions left to test after this
[2be508d847392e431759e370d21cea9412848758] Merge git://git.
infradead.org/mtd-2.6
```

Обратите внимание: что 3857 ревизий были сокращены до 1939. Двигаемся далее:

```
$ git bisect good
Bisecting: 939 revisions left to test after this
[b80de369aa5c7c8ce7ff7a691e86e1dcc89accc6] 8250: Add more
OxSemi devices

$ git bisect bad
Bisecting: 508 revisions left to test after this
[9301975ec251bab1ad7cfcb84a688b26187e4e4a] Merge branch
'genirq-v28-for-linus'
of git://git.kernel.org/pub/scm/linux/kernel/git/tip/linux-
2.6-tip
```

В идеальном случае данная процедура занимает log2 исходного числа версий шагов, чтобы сузиться всего до одной фиксации. После еще одного «хорошего» и «плохого» ответа:

```
$ git bisect good
Bisecting: 220 revisions left to test after this
[7cf5244ce4a0ab3f043f2e9593e07516b0df5715] mfd: check for
platform_get_irq() return value in sm501

$ git bisect bad
Bisecting: 104 revisions left to test after this
[e4c2ce82ca2710e17cb4df8eb2b249fa2eb5af30] ring_buffer:
allocate
buffer page pointer
```

Во время этого процесса Git ведет журнал ваших ответов и ID фиксаций:

```
$ git bisect log
git bisect start
# bad: [49fdf6785fd660e18a1eb4588928f47e9fa29a9a] Merge branch
'for-linus' of git://git.kernel.dk/linux-2.6-block
git bisect bad 49fdf6785fd660e18a1eb4588928f47e9fa29a9a
# good: [3fa8749e584b55f1180411ab1b51117190bac1e5] Linux
2.6.27
git bisect good 3fa8749e584b55f1180411ab1b51117190bac1e5
# good: [cf2fa66055d718ae13e62451bb546505f63906a2] Merge
branch 'for_linus'
of git://git.kernel.org/pub/scm/linux/kernel/git/mchehab/
linux-2.6
git bisect good cf2fa66055d718ae13e62451bb546505f63906a2
# good: [2be508d847392e431759e370d21cea9412848758] Merge
git://git.infradead.org/mtd-2.6
git bisect good 2be508d847392e431759e370d21cea9412848758
# bad: [b80de369aa5c7c8ce7ff7a691e86e1dcc89accc6] 8250: Add
more
OxSemi devices
git bisect bad b80de369aa5c7c8ce7ff7a691e86e1dcc89accc6
# good: [9301975ec251bab1ad7cfcb84a688b26187e4e4a] Merge
branch
'genirq-v28-for-linus' of
git://git.kernel.org/pub/scm/linux/kernel/git/tip/linux-2.6-
tip
git bisect good 9301975ec251bab1ad7cfcb84a688b26187e4e4a
# bad: [7cf5244ce4a0ab3f043f2e9593e07516b0df5715] mfd: check
for
platform_get_irq() return value in sm501
git bisect bad 7cf5244ce4a0ab3f043f2e9593e07516b0df5715
```

Если вы заблудились во время процесса, введите команду **git bisect replay** - Git начнет все сначала, используя журнал ваших ответов в качестве ввода.

Давайте сужать дефект пятью (или более) «плохими» ответами:

```
$ git bisect bad
Bisecting: 51 revisions left to test after this
[d3ee6d992821f471193a7ee7a00af9ebb4bf5d01] ftrace: make it
depend on DEBUG_KERNEL

$ git bisect bad
Bisecting: 25 revisions left to test after this
[3f5a54e371ca20b119b73704f6c01b71295c1714] ftrace: dump out
ftrace buffers to console on panic

$ git bisect bad
Bisecting: 12 revisions left to test after this
[8da3821ba5634497da63d58a69e24a97697c4a2b] ftrace: create
_mcount_loc section

$ git bisect bad
Bisecting: 6 revisions left to test after this
[fa340d9c050e78fb21a142b617304214ae5e0c2d] tracing: disable
tracepoints by default

$ git bisect bad
Bisecting: 2 revisions left to test after this
[4a0897526bbc5c6ac0df80b16b8c60339e717ae2] tracing:
tracepoints, samples
```

Для визуализации процесса вы можете использовать команду **git bisect visualize**. Git использует графическую утилиту gitk, если переменная окружения DISPLAY установлена. Если же нет, Git использует команду git log. В этом случае будет полезной опция --pretty=oneline.

```
$ git bisect visualize --pretty=oneline
fa340d9c050e78fb21a142b617304214ae5e0c2d tracing: disable
tracepoints
by default
b07c3f193a8074aa4afe43cfa8ae38ec4c7ccfa9 ftrace: port to
tracepoints
0a16b6075843325dc402edf80c1662838b929aff tracing, sched: LTTng
```

```
instrumentation - scheduler
4a0897526bbc5c6ac0df80b16b8c60339e717ae2 tracing: tracepoints,
samples
24b8d831d56aac7907752d22d2aba5d8127db6f6 tracing: tracepoints,
documentation
97e1c18e8d17bd87e1e383b2e9d9fc740332c8e2 tracing: Kernel
Tracepoints
```

Текущая версия на рассмотрении находится примерно в середине диапазона.

```
$ git bisect good
Bisecting: 1 revisions left to test after this
[b07c3f193a8074aa4afe43cfa8ae38ec4c7ccfa9] ftrace: port to
tracepoints
```

Когда вы окончательно протестируете последнюю ревизию и Git изолируют одну ревизию, вызывающую проблему, будет отображено:

```
$ git bisect good
fa340d9c050e78fb21a142b617304214ae5e0c2d is first bad commit
commit fa340d9c050e78fb21a142b617304214ae5e0c2d
Author: Ingo Molnar <mingo@elte.hu>
Date: Wed Jul 23 13:38:00 2020 +0200

tracing: disable tracepoints by default

while it's arguably low overhead, we dont enable new features
by default.

Signed-off-by: Ingo Molnar <mingo@elte.hu>

:040000 040000 4bf5c05869a67e184670315c181d76605c973931
fd15e1c4adbd37b819299a9f0d4a6ff589721f6c M init
```

По завершению сего процесса жизненно важно сказать Git, что вы закончили. Как уже было отмечено, весь процесс деления осуществлялся на отсоединенной HEAD:

```
$ git branch
* (no branch)
master
```

```
$ git bisect reset
Switched to branch "master"

$ git branch
* master
```

Команда **git bisect reset** возвращает обратно исходную ветку.

Использование git blame

Есть еще одна утилита, позволяющая идентифицировать конкретную фиксацию - **git blame**. Данная команда говорит вам, кто в последний раз модифицировал каждую строку файла и какая фиксация была выполнена.

```
$ git blame -L 35, init/version.c

4865ecf1 (Serge E. Hallyn 2006-10-02 02:18:14 -0700 35) },
^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700 36) };
4865ecf1 (Serge E. Hallyn 2006-10-02 02:18:14 -0700 37) EXPORT_
SYMBOL_GPL(init_uts_ns);
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 38)
c71551ad (Linus Torvalds 2007-01-11 18:18:04 -0800 39) /* FIXED
STRINGS!
                                                      Don't touch!
                                                   */
c71551ad (Linus Torvalds 2007-01-11 18:18:04 -0800 40) const char
linux_banner[] =
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 41) "Linux
version "
                                                      UTS_RELEASE "
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 42) (" LINUX_
COMPILE_BY "@"
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 43) LINUX_
COMPILE_HOST ")
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 44) (" LINUX_
COMPILER ")
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 45) " UTS_VERSION
"\n";
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 46)
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 47) const char
linux_proc_banner[] =
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 48) "%s version
%s"
```

```
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 49) " (" LINUX_
COMPILE_BY
                                                      "@"
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 50) LINUX_
COMPILE_HOST ")"
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 51) " (" LINUX_
COMPILER ")
                                                      %s\n»;
```

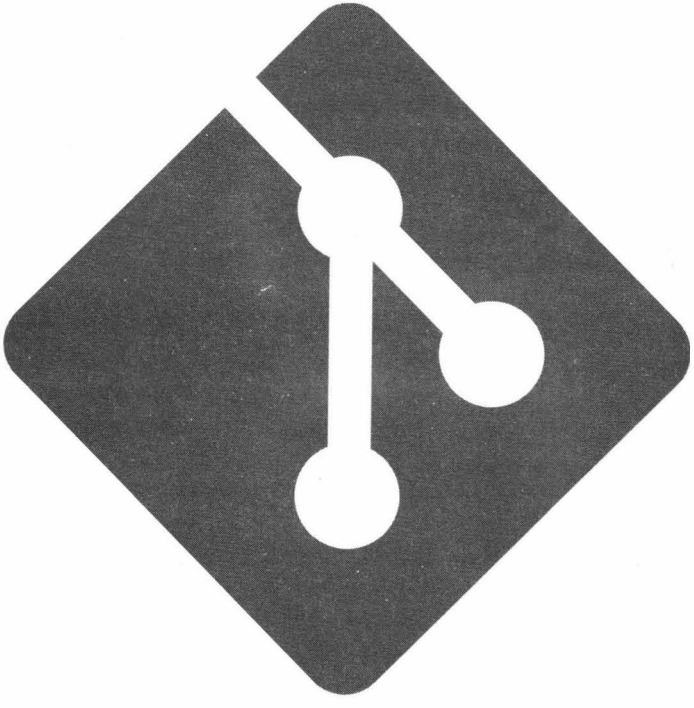
Использование Pickaxe

Команда **git blame** показывает вам текущее состояние файла, а git log -Sстрока производит поиск в истории дельт файла заданной строки. По умолчанию производится поиск актуальных дельт между ревизиями, команда может найти фиксацию, которая произвела изменение (как добавление, так и удаление).

```
$ git log -Sinclude --pretty=oneline --abbrev-commit init/
version.c
cd354f1... [PATCH] remove many unneeded #includes of sched.h
4865ecf... [PATCH] namespaces: utsname: implement utsname
namespaces
63104ee... kbuild: introduce utsrelease.h
1da177e... Linux-2.6.12-rc2
```

Каждая из фиксаций, перечисленных слева (cd354f1, и т.д.) или добавляет или удаляет строки, которые содержат заданное слово. Все же будьте осторожны. Если фиксация и добавляет и удаляет одинаковое число экземпляров строки с вашей ключевой фразой, она не будет показана. У фиксации должно быть *изменение* в числе добавление и удалений.

Опция -S команды **git log** называется pickaxe. Это археология грубой силы, доступная вам.



Глава 7.

Ветки в Git



Ветка - это фундаментальное понятие, означающее запуск отдельной линии разработки в проекте программного обеспечения. Ветка позволяет вести одновременную разработку в нескольких направлениях, которые, потенциально, могут привести к разным версиям проекта. Часто выполняется слияние одних веток с другими для воссоединения несоизмеримых усилий.

Git позволяет использовать много веток, то есть много разных линий разработки, в одном репозитарии. Система ветвления Git легка и проста. К тому же у Git есть замечательная система слияния.

В этой главе вы узнаете, как выбрать, создать, просмотреть и удалить ветки. Также в ней будут даны советы относительно эффективного использования веток.

7.1. Причины использования веток

Отдельные ветки могут быть созданы по целому ряду технических, философских, маркетинговых и даже социальных соображений. Рассмотрим наиболее вероятные причины создания разных веток:

- Ветки часто представляют отдельные версии программы. Если вы хотите начать версию 1.1 вашего проекта, но вы знаете, что некоторые ваши

клиенты желают остаться на версии 1.0, сохраните старую версию в виде отдельной ветки.

- Ветка может представлять этап разработки, например, prototype (прототип), beta (бета-версия), stable (стабильная). Вы можете думать о версии 1.1 как об отдельном этапе.
- Ветка может изолировать разработку одной функции или помочь в исследовании сложной ошибки. Конечно, чтобы исправить одну ошибку не очень целесообразно создавать новую ветку, но система ветвления Git поддерживает и малогабаритное использование.
- Отдельная ветка может представлять работу отдельного программиста.

Сам же Git называет ветки или веткой темы (topic branch) или веткой разработки (development branch). Слово «тема» просто подчеркивает, что у каждой ветки в репозитарии свое назначение.

В документации Git можно столкнуться с еще одним термином – ветка отслеживания (tracking branch) или ветка, использующаяся для синхронизации клонов репозитария.

7.2. Ветка или тег?

Ответвление и тег кажутся подобными, возможно даже взаимозаменяемыми. Когда же вы должны использовать имя тега, а когда – имя ветки?

Тег и ветка служат различным целям. **Тег** – это статическое имя, которое не изменяется и не перемещается в течение долгого времени. После того, как вы его применили, вы должны его оставить в покое. **Ветка** – это динамический объект, который перемещается с каждой фиксацией. Имя ветки предназначено, чтобы следовать за вашей продолжительной разработкой.

Любопытно, у вас могут быть тег и ветка с одинаковыми именами (только не запутайтесь!). Если вы назвали ветку и тег одинаковыми именами, вы должны использовать полное имя ссылки, чтобы различить их, например, refs/tags/v1.0 и refs/heads/v1.0. Во время разработки вы можете использовать, какое-то имя ветки, например, v2.0, а по ее завершению преобразовать это имя в имя тега.

Название веток и тегов, в конечном счете, ваше личное дело и дело вашей политики проекта. Однако рекомендуется использовать следующую дифференциацию: статическое и неизменное имя должно быть тегом, динамическое – веткой.

Наконец, у вас есть неопровержимый довод избегать одинаковых имен для веток и тегов!

Имена веток

Имя, которое вы присваиваете ветке, абсолютно произвольно, хотя есть некоторые ограничения. Ветка по умолчанию в репозитарии называется **master** и большинство разработчиков хранит в этой ветке самую устойчивую и надежную линию разработки. В ветке master нет ничего волшебного, за исключением того, что Git создает ее при инициализации репозитария. При желании вы можете удалить или переименовать эту ветку, но, как показывает практика, лучшее решение – просто оставить ее в покое.

Для поддержки масштабируемости вы можете создать иерархическое имя ветки, которое напоминает путь Unix. Предположим, что вы - часть группы разработчиков, которая занимается исправлением ошибок. Вы можете создать ветку bug, а в ее подветки поместить разработку каждого восстановления, например, bug/pr-1023 и bug/pr-17.

Примечание. Одна из причин использования иерархических имен – то, что Git, подобно оболочке Unix, поддерживает маски. Например, если у вас есть ветки с именами bug/pr-1023 и bug/pr-17, то выбрать все ветки внутри ветки bug можно так: git show-branch 'bug/*'

Правила именования веток

Имена веток должны соответствовать нескольким простым правилам:

- Вы можете использовать прямой слеш (/) для создания иерархической схемы имен. Однако, имя не может заканчиваться слешем.
- Имя не может начинаться со знака минус (-)

- Если имя не содержит слеш, тогда оно может начинаться с точки (.). Имя вроде feature/.new некорректное.
- Имя не может содержать две последовательные (..) точки
- Также имя не может содержать следующее:
  - » Пробел или пробельный символ
  - » Символы, имеющие специальное назначение в Git: тильда (~), каретка (^), двоеточие (:), вопросительный знак (?), звездочка (*), открытая квадратная скобка ([).
  - » Управляющий символ ASCII, то есть любой символ с кодом меньше \40 (восьм.) или символ DEL (\177 восьм.)

Проверить, соответствуют ли ваши имена данным правилам можно командой **git check-ref-format**. Данная команда проверит, являются ли имена веток легкими для ввода и подходящими для использования в качестве имен файлов внутри каталога .git и в сценариях.

7.3. Использование веток

В репозитарии может быть много разных веток, но есть только одна активная или текущая ветка. Активная ветка определяет, какие файлы будут проверены в рабочем каталоге. Кроме того, текущая ветка – это неявный операнд в командах Git, например, в командах слияния. По умолчанию активной веткой является ветка master, но вы можете сделать любую ветку репозитария.

> **Примечание. В главе 6 мы представили графы фиксации, содержащие несколько веток. Помните эту структуру графа, когда работаете с ветками.**

Ветка позволяет контенту репозитария происходить во многих направлениях, по одному для ветки. Репозитарий работает как минимум с одной веткой, каждая фиксация применяется к той или иной ветки, обычно к активной ветке.

У каждой ветки в определенном репозитарии должно быть уникальное имя. Имя всегда ссылается на более последнюю ревизию, зафиксированную в этой ветке. Наиболее последняя фиксация в ветке называется HEAD.

Git не хранит информацию о том, где произошло ответвление (где началась ветка). Вместо этого новые фиксации создаются уже на новой ветке. Доступ к старым фиксациям можно получить по их хэшу (SHA1 ID) или через относительное имя, например, dev~5. Если вы хотите отслеживать определенную фиксацию, поскольку она представляет стабильную точку в вашем проекте или, скажем, является версией, которую вы хотите протестировать, вы можете присвоить ей легко запоминаемое имя тега.

Поскольку исходная фиксация, из которой произошла ветка, не может быть явно идентифицирована, ее можно найти алгоритмически, используя имя ветки, из которой произошла новая ветка:

```
$ git merge-base original-branch new-branch
```

Слияние – это дополнение ветки. При слиянии контент одной или нескольких веток соединяется с неявной целевой веткой. Однако слияние не устраняет исходные ветки и не изменяет их имена. В главе 9 мы сфокусируемся на сложном процессе слияния веток.

Вы можете думать об имени ветки как об указателе на определенную фиксацию. Ветка содержит фиксации, достаточные, чтобы восстановить всю историю проекта вдоль ветки, из которой она произошла, полностью до самого начала проекта.

На рис. 7.1 ветка **dev** указывает на голову фиксации – Z. Если нужно собрать репозитарий, начиная с состояния Z до исходной фиксации, фиксации A, вы можете это сделать. Достижимая порция графа выделена жирными линиями и она охватывает почти каждую фиксацию, кроме (S, G, H, J, K, L).

Каждое из ваших имен веток, как и содержимое фиксации на каждой ветке, локально для вашего репозитария. Однако, когда вы предоставляете доступ к репозитарию другим пользователям, вы можете *опубликовать* или выбрать одну или любое другое число веток и связанных с ними фиксаций. Публикация ветки должна быть произведена явно. Кроме того, при клонировании вашего репозитария имена веток и разработка в тех ветках станут частью нового клонированного репозитария.

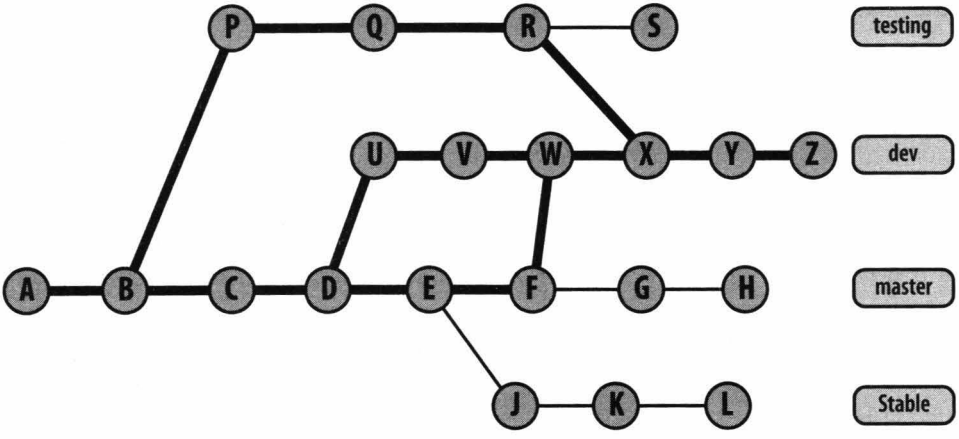


Рис. 7.1. Фиксации, доступные из ветки dev

7.4. Создание веток

Новая ветка основывается на существующей фиксации в репозитарии. Какая фиксация станет началом новой ветки – полностью ваше дело. Git поддерживает произвольно сложную структуру ветвления, включая разветвление нескольких веток из одной фиксации.

Время жизни ветки – опять ваше решение. Ветка может жить недолго, а может быть вечной. Имя ветки может быть многократно добавлено и удалено на протяжении всей жизни репозитария. Например, для тестирования вы можете создать ветку test, а по завершению тестирования – удалить ее, а когда вам опять нужно будет протестировать проект – вы опять можете создать ветку с именем test.

Как только вы определили фиксацию, из которой должна начаться новая ветка, используйте команду **git branch**. Чтобы создать новую ветку, начиная с фиксации HEAD текущей ветки из соображений исправления проблемы 1138 (Problem Report #1138), введите команду:

```
$ git branch prs/pr-1138
```

Базовая форма команды выглядит так:

```
git branch ветка [начальная фиксация]
```

Если начальная фиксация не указана, по умолчанию используется самая последняя фиксация текущей ветки (HEAD). Другими словами, по умолчанию новая ветка начинается в точке, в которой вы сейчас находитесь.

Обратите внимание на то, что команда **git branch** просто вводит имя ветки в репозитарий. Она не изменяет ваш рабочий каталог для использования новой ветки. Она не производит никаких изменений ни в файлах рабочего каталога, ни в контексте ветки, она не создает новую фиксацию. Команда просто создает имя ветки в данной фиксации. Вы не можете начать работу над веткой, пока не переключитесь на нее, но об этом мы поговорим позже.

Иногда нужно указать определенную фиксацию, как начало ветки. Например, предположим, что ваш проект создает новую ветку для каждой ошибки, о которой сообщили разработчикам. В этом случае удобно использовать параметр <начальная фиксация> команды **git branch**.

Обычно, ваш проект устанавливает соглашения, которые позволяют вам с уверенностью определять начальную фиксацию. Например, чтобы исправить ошибку в версии 2.3 вашего проекта, вы можете создать ветку, указав rel-2.3 в качестве начальной фиксации:

```
$ git branch prs/pr-1138 rel-2.3
```

Только одно имя фиксации гарантирует уникальность – это ID-хэш. Если вы знаете хэш, вы можете указать его непосредственно:

```
$ git branch prs/pr-1138 db7de5feebef8bcd18c5356cb47c337236
b50c13
```

7.5. Вывод имен веток

Команда **git branch** (если она задана без дополнительных параметров) выводит имена веток, найденных в репозитарии:

```
$ git branch
bug/pr-1
dev
* master
```

В этом примере у нас есть три ветки. Текущая (активная) в данный момент ветка выделена звездочкой. В этом примере кроме активной ветки есть еще две другие ветки – bug/pr-1 и dev.

7.6. Просмотр веток

Команда **git show-branch** предоставляет более детальный вывод, чем команда git branch, выводя фиксации, которые входят в одну или несколько веток в обратном хронологическом порядке. Команда git branch без параметров выводит список веток, с опцией **–r** выводит удаленные ветки, а **–a** – выводит все ветки.

Давайте посмотрим пример:

```
$ git show-branch
! [bug/pr-1] Fix Problem Report 1
 * [dev] Improve the new development
  ! [master] Added Bob's fixes.
---
* [dev] Improve the new development
* [dev^] Start some new development.
+ [bug/pr-1] Fix Problem Report 1
+*+ [master] Added Bob's fixes.
```

Вывод команды **git show-branch** разделен на две секции, разделенные строкой из тире. Секция выше разделителя перечисляет имена веток, заключенных в квадратные скобки, одна за одной. Каждое имя ветки помечается восклицательным знаком, если только это не текущая ветка, которая отмечается звездочкой. В данном примере текущей веткой является ветка dev.

Нижняя секция вывода – матрица, утверждающая, какие фиксации присутствуют в каждом ответвлении. Фиксация есть в ветке, если в колонке ветки есть знак плюс (+), звездочка (*), или минус (-). Знак «плюс» указывает, что фиксация находится в ветке, «звездочка» просто выделяет фиксацию как присутствующую в активной ветке. Знак «минус» означает фиксацию слияния.

Например, обе из следующих фиксаций отмечены звездочкой, следовательно, они есть в ветке **dev**:

```
* [dev] Improve the new development
* [dev^] Start some new development.
```

Эти две фиксации не представлены в любой другой ветке. Они выведены в обратном хронологическом порядке: наиболее последняя фиксация выводится на вершине списка, старые фиксации находятся внизу списка.

В квадратных скобках в каждой строке фиксации Git выводит имя фиксации. Как уже было сказано, Git присваивает имя ветки каждой последней фиксации. У предшествующих фиксаций будет имя с символом каретки ^. В главе 6, если помните, имя master использовалось для самой последней фиксации, а имя master^ для предпоследней фиксации. Аналогично, в этой главе используются имена dev и dev^ - это две последние фиксации на ветке dev.

Несмотря на то, что фиксации в ветке упорядочены, сами ветки перечислены в произвольном порядке. Это вызвано тем, что у всех веток одинаковый статус, нет никакого правила, которое бы утверждало, что одна ветка – важнее, чем другая.

Если одна и та же фиксация присутствует в нескольких ветках, у нее будет знак + или * для каждой ветки. Следующая фиксация есть во всех трех ветках:

```
+*+ [master] Added Bob's fixes.
```

Первый знак + означает, что фиксация есть в ветке bug/pr-1, звездочка означает, что фиксация есть в активной ветке (то есть в dev), а второй знак плюс указывает, что фиксация есть в ветке master.

Команда **git show-branch** обходит все фиксации на всех ветках, пока она не достигнет фиксации, общей для всех ветках. В нашем примере было выведено 4 фиксации, прежде, чем была найденная общая для всех веток фиксация (Added Bob's fixes). В этой точке команда остановилась.

Остановка на первой общей фиксации – поведение по умолчанию. Предполагается, что достижение такой общей точки приводит к достаточному контексту, чтобы понять, как ветки касаются друг друга. Если по некоторым причинам вы хотите вывести больше фиксаций, используйте опцию –more=num, где num – это число дополнительных фиксаций.

Команда git show-branch принимает набор имен веток в виде параметров, позволяя вам ограничить историю только указанными ветками. Например,

если ветка с именем bug/pr-2 была добавлена, начиная с фиксации master, она была бы похожа на это:

```
$ git branch bug/pr-2 master
$ git show-branch
! [bug/pr-1] Fix Problem Report 1
 ! [bug/pr-2] Added Bob's fixes.
  * [dev] Improve the new development
   ! [master] Added Bob's fixes.
----
* [dev] Improve the new development
* [dev^] Start some new development.
+ [bug/pr-1] Fix Problem Report 1
++*+ [bug/pr-2] Added Bob's fixes.
```

Если вы хотите увидеть историю фиксаций только для веток bug/pr-1 и bug/pr-2, используйте команду:

```
$ git show-branch bug/pr-1 bug/pr-2
```

Такой способ подходит, чтобы указать несколько веток. Если же нужно указать больше веток, то он может быть очень неудобным. Вместо этого Git предлагает использовать маски, например, bug/*:

```
$ git show-branch bug/pr-1 bug/pr-2
! [bug/pr-1] Fix Problem Report 1
 ! [bug/pr-2] Added Bob's fixes.
--
+ [bug/pr-1] Fix Problem Report 1
++ [bug/pr-2] Added Bob's fixes.

$ git show-branch bug/*
! [bug/pr-1] Fix Problem Report 1
 ! [bug/pr-2] Added Bob's fixes.
--
+ [bug/pr-1] Fix Problem Report 1
++ [bug/pr-2] Added Bob's fixes.
```

7.7. Переключение веток и выгрузка файлов

Как уже было отмечено в этой главе, ваш рабочий каталог может отображать только одну ветку за раз. Чтобы начать работу с другой веткой, используйте команду **git checkout**. Данной команде нужно передать имя ветки, которая станет новой текущей веткой. Данная команда изменяет ваше рабочее дерево файлов и структуру каталогов, чтобы все это соответствовало состоянию текущей ветки. Однако, как будет показано далее, Git делает все возможное, чтобы предотвратить потерю еще незафиксированных данных.

Кроме того, команда **git checkout** позволяет извлекать отдельные файлы из истории фиксаций. Но об этом мы поговорим позже.

Базовое переключение между ветками

Представим, что вам нужно переключиться из ветки dev (которая была активной в предыдущем примере) на ветку bug/pr-1. Давайте посмотрим на состояние рабочего каталога до и после выполнения команды **git checkout**:

```
$ git branch
bug/pr-1
bug/pr-2
* dev
master

$ git checkout bug/pr-1
Switched to branch «bug/pr-1»

$ git branch
* bug/pr-1
bug/pr-2
dev
master
```

Структура файлов и каталогов рабочего дерева будет обновлена, чтобы отразить состояние и содержимое новой ветки – bug/pr-1. Чтобы убедиться в этом, вы можете использовать команду оболочки Unix – ls.

Выбор новой текущей ветки оказывает существенное влияние на рабочее дерево файлов и каталогов. Это влияние выражается в следующем:

- Файлы и каталоги, представленные ветке, на которую производится переключение, но отсутствующие в текущей ветке, будут помещены в ваше рабочее дерево.
- Файлы и каталоги, существующие в вашей текущей ветке, но отсутствующие в загружаемой ветке, будут удалены из рабочего дерева.
- Файлы, имеющиеся в обоих ветках, будут изменены, чтобы отразить содержимое, представленное в загружаемой ветке.

Переключение веток при имеющихся незафиксированных изменениях

Git предотвращает случайное удаление или модификацию данных в вашем локальном рабочем дереве без вашего явного запроса. Неотслеживаемые файлы и каталоги в вашем текущем каталоге никогда не будут затронуты при переключении веток. Git не будет удалять или изменять их. Однако, если у вас есть локальные изменения в файле, которые отличаются от изменений, представленных в новой ветке, Git выведет сообщение об ошибке и откажется переключать ветку:

```
$ git branch
bug/pr-1
bug/pr-2
dev
* master

$ git checkout dev
error: Your local changes to the following files would be
overwritten by checkout:
NewStuff
Please, commit your changes or stash them before you can
switch branches.
Aborting
```

В этом случае сообщение предупреждает, что что-то заставило Git отказаться от переключения ветки. Но что? Если вы исследуете содержимое файла

NewStuff, находящегося в вашем рабочем каталоге и в ветке **dev**, на которую производится переключение, вы поймете, в чем причина:

```
# Посмотрим, что в файле NewStuff в рабочем каталоге
$ cat NewStuff
Something
Something else

# В локальной версии файла есть дополнительная строка, которая
# незафиксирована в текущей ветке (master)
$ git diff NewStuff
diff --git a/NewStuff b/NewStuff
index 0f2416e..5e79566 100644
--- a/NewStuff
+++ b/NewStuff
@@ -1 +1,2 @@
Something
+Something else

# Посмотрим, как выглядит файл в ветке
$ git show dev:NewStuff
Something
A Change
```

Если бы Git таки переключил ветку, то ваши локальные изменения, внесенные в файл NewStuff в рабочем каталоге, были бы переписаны версией файла из ветки dev. По умолчанию Git обнаруживает эти возможные потери и предотвращает их.

> **Примечание. Если вы не боитесь потерь данных в вашем рабочем каталоге, вы можете принудительно сменить ветку, указав опцию –f.**

Чтобы избавиться от сообщения об ошибке нужно фиксировать изменения в текущей ветке (команда git commit). Одной только команды **git add** недостаточно, вы опять получите сообщение об ошибке:

```
$ git add NewStuff
$ git checkout dev
error: Your local changes to the following files would be
overwritten by checkout:
NewStuff
```

```
Please, commit your changes or stash them before you can
switch branches.
Aborting
```

Используйте команду **git commit** для фиксации изменений в текущей ветке (master), после чего вы можете попытаться изменить ветку на dev. Вы хотите поместить изменения сразу в ветку dev? У вас ничего не получится, поскольку нужно сначала переключить ветку, а потом уже фиксировать изменения, а Git не переключит ветку до тех пор, пока есть даже незначительные изменения. Если же изменений много и они предназначены для ветки dev, на которую вы забыли заранее переключиться, тогда вам нужно обязательно нужно прочитать следующий раздел, в котором говорится об объединении изменений в другую ветку.

Объединение изменений в другую ветку

В предыдущем разделе текущее состояние вашего рабочего каталога конфликтовало с той веткой, на которую вы хотели переключиться. Здесь необходимо объединение (слияние): изменения в вашем рабочем каталоги должны быть объединены с веткой, на которую происходит переключение.

Если возможно или если явно потребовано опцией *-m*, Git пытается перенести ваше локальное изменение в новый рабочий каталог, осуществив операцию слияния между вашими локальными изменениями и целевой веткой.

```
$ git checkout -m dev
M NewStuff
Switched to branch «dev»
```

Здесь видно, что Git модифицировал файл NewStuff и успешно переключился на ветку dev.

Эта операция по слиянию происходит полностью в вашем текущем каталоге. Она несколько похожа на команду **cvs update**, при которой ваши локальные изменения сливаются с целевой веткой и остаются в вашем текущем каталоге.

Как бы там ни было, будьте осторожны. Все может быть похоже на то, что слияние прошло чисто и все хорошо, но Git может просто изменить файл и оставить индикаторы конфликта слияния в нем. Вам нужно все еще разрешать любые присутствующие конфликты:

```
$ cat NewStuff
Something
<<<<<<< dev:NewStuff
A Change
=======
Something else
>>>>>>> local:NewStuff
```

В главе 9 будет подробно рассмотрено слияние и полезные техники разрешения конфликтов слияния.

Если Git может выгрузить ветку, изменить ее и объединить ваши локальные изменения без каких-либо конфликтов слияния, это значит, что переключение ветки прошло успешно.

Представим, что вы используете ветку **master** в вашем репозитарии разработки, и вы внесли некоторые изменения в файл NewStuff. Но изменения, которые вы внесли, на самом деле должны были быть внесены в другой ветке, возможно, потому что они исправляют Problem Report #1 (отчет о проблеме 1) и должны быть переданы ветке bug/pr-1.

Начните с ветки **master**, внесите необходимые изменения в некоторые файлы, у нас они выражены добавлением текста Some bug fix в файлNewStuff.

```
$ git show-branch
! [bug/pr-1] Fix Problem Report 1
 ! [bug/pr-2] Added Bob's fixes.
  ! [dev] Started developing NewStuff
   * [master] Added Bob's fixes.
----
   +  [dev] Started developing NewStuff
   +  [dev^] Improve the new development
   +  [dev~2] Start some new development.
+     [bug/pr-1] Fix Problem Report 1
+++*  [bug/pr-2] Added Bob's fixes.
```

```
$ echo "Some bug fix" >> NewStuff
```

```
$ cat NewStuff
Something
Some bug fix
```

В данной точке внесенные вами изменения должны быть зафиксированы в ветке bug/pr-1, но не в ветке master. До этого в ветке bug/pr-1 файл NewStuff выглядел так:

```
$ git show bug/pr-1:NewStuff
Something
```

Чтобы внести ваши изменения в желаемую ветку просто попытаемся переключиться на нее:

```
$ git checkout bug/pr-1
M NewStuff
Switched to branch «bug/pr-1»

$ cat NewStuff
Something
Some bug fix
```

Здесь у Git получилось корректно слить изменения из вашего рабочего каталога в целевую ветку и оставить их в новой структуре рабочего каталога. С помощью команды **git diff** вы должны проверить, что слияние произошло в соответствии с вашими ожиданиями:

```
$ git diff
diff --git a/NewStuff b/NewStuff
index 0f2416e..b4d8596 100644
--- a/NewStuff
+++ b/NewStuff
@@ -1 +1,2 @@
Something
+Some bug fix
```

Это означает, что была добавлена одна строка.

Создание новой ветки и переключение на нее

Другой частый сценарий – это когда вам нужно создать новую ветку и одновременно переключиться на нее. Для этого Git предоставляет опцию –b новая_ветка.

Давайте рассмотрим уже знакомый пример за тем лишь исключением, что мы сразу переключимся на созданную ветку вместо внесения изменений в существующую ветку. Другими словами, вы находитесь в ветке master, редактируете файлы, а затем внезапно понимаете, что внесенные изменения нужно поместить в новую ветку с именем bug/pr-3. Последовательность команд будет следующей:

```
$ git branch
bug/pr-1
bug/pr-2
dev
* master

$ git checkout -b bug/pr-3
M NewStuff
Switched to a new branch «bug/pr-3»

$ git show-branch
! [bug/pr-1] Fix Problem Report 1
 ! [bug/pr-2] Added Bob's fixes.
  * [bug/pr-3] Added Bob's fixes.
   ! [dev] Started developing NewStuff
    ! [master] Added Bob's fixes.
-----
   +  [dev] Started developing NewStuff
   +  [dev^] Improve the new development
   +  [dev~2] Start some new development.
+     [bug/pr-1] Fix Problem Report 1
++*++ [bug/pr-2] Added Bob's fixes.
```

Если никакая проблема не препятствует переключению на другую ветку, следующая команда:

```
$ git checkout -b новая_ветка начальная_точка
```

аналогична последовательности команд:

```
$ git branch новая_ветка начальная_точка
$ git checkout новая_ветка
```

Отсоединение головы ветки

Обычно достаточно проверить только вершину ветки, указав название самой ветки. По умолчанию Git контролирует только вершину желаемого отделения.

Однако вы можете перейти к любой фиксации. В этом случае Git создает своего рода анонимную ветку, которая называется отсоединенной головой (detached HEAD). Git создает отсоединенную голову когда вы:

- Выгружаете фиксацию, которая не является головой ветки
- Выгружаете отслеживаему ветку. Вы должны сделать это для исследования недавно внесенных изменений в ваш локальный репозитарий из удаленного репозитария
- Выгружаете фиксацию, на которую ссылаетесь по ссылке
- Начинаете операцию **git bisect**, описанную в главе 6
- Используете команду **git submodule update**

В этих случаях Git предупреждает вас, что он переместил вас в отсоединенную HEAD:

```
# У меня есть копия исходниковGit
$ cd git.git

$ git checkout v1.6.0
Note: moving to «v1.6.0» which isn't a local branch
If you want to create a new branch from this checkout, you may
do so
(now or later) by using -b with the checkout command again.
Example:
git checkout -b <new_branch_name>
HEAD is now at ea02eef... GIT 1.6.0
```

Если, находясь в отсоединенной HEAD, вы позже решите, что вам нужно сделать новые фиксации в этой точке и сохранить их, вам нужно сначала создать новую ветку:

```
$ git checkout -b new_branch
```

В результате вы получите новую ветку, основанную на фиксации из отсоединенной головы. Затем вы можете продолжить нормальную разработку. По существу, вы назвали ветку, которая ранее была анонимной.

Чтобы определить, находитесь ли вы в отсоединенной голове или нет, просто введите команду:

```
$ git branch
* (no branch)
Master
```

С другой стороны, если вы завершили работу с отсоединенной головой и хотите просто отказаться от этого состояния, вы можете конвертировать его в именованную ветку командой **git checkout** ветка.

```
$ git checkout master
Previous HEAD position was ea02eef... GIT 1.6.0
Checking out files: 100% (608/608), done.
Switched to branch «master»

$ git branch
* master
```

7.8. Удаление веток

Для удаления ветки из репозитария используется команда **git branch –d**. Git не позволит вам удалить текущую ветку:

```
$ git branch -d bug/pr-3
error: Cannot delete the branch 'bug/pr-3' which you are
currently on.
```

Удаление текущей ветки оставило бы Git неспособным определить, на что должно быть похоже результирующее дерево каталогов.

Но есть другая тонкая проблема. Git не позволит удалить вам ветку, которая содержит фиксации, которые не представлены в текущей ветке. Таким образом, Git предотвращает вас от случайного удаления разработки в фиксациях, которые будут потеряны, если ветка будет удалена.

```
$ git checkout master
Switched to branch «master»

$ git branch -d bug/pr-3
error: The branch 'bug/pr-3' is not an ancestor of your
current HEAD.
If you are sure you want to delete it, run 'git branch -D bug/
pr-3'.
```

В этом выводе **git show-branch** говорится, что фиксация «Added a bug fix for pr-3» найдена только в ветке «bug/pr-3». Если эта ветка будет удалена, вы больше не сможете получить доступ к этой фиксации.

Заявляя, что ветка bug/pr-3 не является предком вашей текущей головы (HEAD), Git говорит вам, что линия разработки, представленная веткой bug/pr-3 не добавлена в разработку текущей ветки - master.

Git не заставляет вас объединить все ветки с веткой master перед удалением. Помните, что ветка – это просто имя или указатель на фиксацию, у которой есть фактическое содержимое. Вместо этого Git предотвращает случайную потерю содержимого удаляемой ветки, которое не было передано в текущую ветку.

Если содержимое из удаляемой ветки уже присутствует в другой ветке, такую ветку можно смело удалить. То есть сначала нужно «слить» содержимое из ветки, которую вы хотите удалить, в другую ветку (не обязательно текущую), после чего вы можете безопасно приступить к удалению ветки. Рассмотрим небольшой пример:

```
$ git merge bug/pr-3
Updating 7933438..401b78d
Fast forward
 NewStuff | 1 +
 1 files changed, 1 insertions(+), 0 deletions(-)

$ git show-branch
! [bug/pr-1] Fix Problem Report 1
 ! [bug/pr-2] Added Bob's fixes.
  ! [bug/pr-3] Added a bug fix for pr-3.
   ! [dev] Started developing NewStuff
* [master] Added a bug fix for pr-3.
-----
   + * [bug/pr-3] Added a bug fix for pr-3.
```

```
    +   [dev] Started developing NewStuff
    +   [dev^] Improve the new development
    +   [dev~2] Start some new development.
+       [bug/pr-1] Fix Problem Report 1
++++*   [bug/pr-2] Added Bob's fixes.
```

```
$ git branch -d bug/pr-3
Deleted branch bug/pr-3.
```

```
$ git show-branch
! [bug/pr-1] Fix Problem Report 1
 ! [bug/pr-2] Added Bob's fixes.
  ! [dev] Started developing NewStuff
   * [master] Added a bug fix for pr-3.
----
    *   [master] Added a bug fix for pr-3.
   +    [dev] Started developing NewStuff
   +    [dev^] Improve the new development
   +    [dev~2] Start some new development.
+       [bug/pr-1] Fix Problem Report 1
+++*    [bug/pr-2] Added Bob's fixes.
```

Вы можете переопределить проверку безопасности, указав опцию *–D* вместо *–d*, как вам советует сам Git (см. выше). Однако в этом случае вы потеряете все наработки в удаляемой ветке.

Git не ведет какую-либо историю создания, удаления, изменения, слияния и удаления веток. Как только ветка удалена, ее не стало.

История фиксаций ветки – отдельный вопрос. Git, в конечно счете, удалит фиксации на которые больше не ссылаются ни ветки, ни теги. Если вы хотите сохранить эти фиксации, вам нужно «слить» их в отдельную ветку: сделать для них отдельную ветку или поместить в уже существующую. Иначе, без ссылки на них, фиксации и блобы будут недостижимыми и будут удалены сборщиком мусора (утилитой git gc).

> **Примечание. После случайного удаления ветки или другой ссылки вы можете восстановить ее, используя команду git reflog. Другие команды, такие как git fsck, а также опции конфигурации вроде gc.reflogExpire и gc.prundeExpire могут также помочь восстановить потерянные фиксации и головы веток.**

Глава 8.

Различия в Git



Для начала нужно сказать, что такое diff (сокращение от differences) – это компактная сводка различий между двумя элементами. Например, дано два файла, и команда diff (она по умолчанию есть в Unix или Linux), сравнивающая файлы построчно и выводящая имеющиеся различия, как показано в примере 8.1. В этом примере **initial** – первая версия файла, **rewrite** – вторая версия. Опция –u обеспечивает вывод в формате, широко используемом для представления изменений.

Пример 8.1. Использование Unix-команды diff

```
$ cat initial           $ cat rewrite
Now is the time     Today is the time
For all good men    For all good men
To come to the aid  And women
Of their country.   To come to the aid
Of their country.

$ diff -u initial rewrite
--- initial 1867-01-02 11:22:33.000000000 -0500
+++ rewrite 2000-01-02 11:23:45.000000000 -0500
@@ -1,4 +1,5 @@
-Now is the time
+Today is the time
For all good men
+And women
To come to the aid
Of their country.
```

Давайте подробнее взглянем на разницу. В заголовке исходный файл (initial) обозначен как ---, новый файл – как +++. Строка @@ предоставляет число строк контекста для обоих версий файла. Строка со знаком минус (-) обозначает строку, которую нужно удалить из исходного файла, чтобы получить новый файл. Строка со знаком плюс (+) обозначает строку, которую нужно добавить в исходный файл, чтобы получить новый файл. Никак не обозначенные строки одинаковы для обоих файлов.

Понятно, что diff не сообщает причины изменений, она просто выводит начальное и конечное состояние. Однако diff предлагает намного больше, чем просто набор того, чем отличаются файлы. Она предоставляет формальное описание того, как преобразовать один файл в другой (при желании вы можете их использовать при возвращении файла в исходное состояние). Кроме того, diff может использоваться для показа разницы среди множества файлов и всех иерархий каталогов.

Unix-команда diff может вычислить разницу всех пар файлов, найденных в двух иерархиях каталогов. Для рекурсивного обхода дерева каталогов используйте команду diff –r, команда обойдет два дерева каталогов и выполнит сравнение файлов-близнецов по имени, например, original/src/main.c и new/src/main.c и выведет разницу между каждой парой файлов. Команда diff –r –u будет использовать унифицированный формат вывода, как и команда diff –u.

В Git есть собственная команда diff и она называется **git diff**. Обратите внимание, что команда git diff сравнивает два файла, подобно команде Unix diff. Как и diff –r средство сравнения Git может также обойти два два дерева объектов и сгенерировать представление различий. Но у git diff есть свои нюансы и мощные функции, адаптированные в соответствии с определенными потребностями пользователей Git.

В этой главе мы рассмотрим азы команды **git diff** и некоторые ее специальные возможности.

8.1. Формы команды git diff

Если вы выбираете для сравнения два разных объекта корневого уровня, **git diff** выведет все отклонения между двумя состояниями проекта. Это очень мощно. Вы можете использовать эту особенность для синхронизации ре-

позитариев. Например, если вы и ваш коллега одновременно работаете над одним и тем же проектом, разница корневого уровня может эффективно синхронизировать репозитарии в любое время.

Есть три основных источника для дерева, которые вы можете использовать в **git diff**:

- Любой объект дерева в пределах всего графа фиксаций
- Ваш рабочий каталог
- Индекс

Деревья, сравниваемые в команде git diff, указываются через фиксации, имена веток или тегов (как было указано в разделе «Идентификация фиксаций»). Кроме того, иерархия файлов и каталогов, подготовленных в индексе, также может быть обработана как деревья.

Команда **git diff** может выполнить четыре фундаментальных сравнения, используя различные комбинации тех трех источников.

git diff

Команда показывает разницу между вашим рабочим каталогом и индексом. Она показывает, что в вашем рабочем каталоге является кандидатом для следующей фиксации. Эта команда не показывает различия между тем, что находится в вашем индексе и тем, что сохранено в репозитарии.

git diff фиксация

Эта форма выводит различия между вашим рабочим каталогом и указанной фиксацией. По умолчанию, если фиксация не задана, используется HEAD.

git diff --cached фиксация

Данная команда показывает разницу между подготовленными в индексе изменениями и заданной фиксацией. Если фиксация не задана, используется HEAD. Если у вас эта команда не работает, попробуйте вместо нее выпол-

нить команду git diff --staged фиксация. Данная команда является синонимом для git diff --cached.

git diff фиксация1 фиксация2

Показывает разницу между двумя произвольными фиксациями. Эта команда игнорирует индекс и рабочий каталог. Она работает только с фиксациями, которые уже находятся в хранилище объектов.

Число параметров в командной строке определяет, какая форма используется и что именно сравнивается. Вы можете сравнить две фиксации или два дерева. У сравниваемых объектов не должно быть прямых или косвенных родительских отношений.

Давайте исследуем, как эти три разные формы применяются к объектной модели Git.

В дополнение к четырем каноническим формам разности Git обеспечивает бесчисленные опции. Вот самые полезные из них:

--M

Опция --M обнаруживает переименования и файлов.

-w или --ignore-all-space

Обе эти опции сравнивает строки, не считая пробелы и пробельные символы значимыми, то есть попросту игнорируют их.

--stat

Опция --stat добавляет статистику относительно разностью между двумя состояниями дерева. Она сообщает, сколько строк изменено, сколько добавлено и сколько удалено.

--color

Опция --color "раскрашивает" вывод. Каждому типу разницы назначается уникальный цвет. Опция улучшает восприятие вывода программы, особенно, когда различий между файлами много.

Опция -a для команды git diff ничего не делает, в отличие от опции -a команды git commit. Чтобы получить помещенные и не помещенные в индекс изменения, используйте команду git diff HEAD. Нехватка симметрии в данном случае непонятна и парадоксальна.

8.2. Простой пример git diff

Здесь мы попробуем реконструировать сценарий, представленный на рис. 8.1, «пройтись» по этому сценарию и посмотреть на различные формы команды git diff в действии. Первым делом давайте создадим репозитарий для нашего примера с двумя файлами в нем.

```
$ mkdir /tmp/diff_example

$ cd /tmp/diff_example

$ git init
Initialized empty Git repository in /tmp/diff_example/.git/

$ echo «foo» > file1
$ echo «bar» > file2

$ git add file1 file2

$ git commit -m "Add file1 and file2"
[master (root-commit)]: created fec5ba5: «Add file1 and file2»
2 files changed, 2 insertions(+), 0 deletions(-)
create mode 100644 file1
create mode 100644 file2
Next, let's edit file1 by replacing the word "foo" with "quux."

$ echo «quux» > file1
```

Файл file1 был модифицирован в рабочем каталоге, но не был помещен в индекс. Это еще не ситуация, изображенная на рис. 8.1, но вы можете уже произвести сравнение. Вы должны ожидать вывод, если вы сравниваете рабочий каталог с индексом или с существующими HEAD-версиями. Однако между индексом и HEAD не должно быть никаких различий, поскольку ничего еще не помещено в индекс.

```
# рабочий каталог vs индекс
$ git diff
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux

# рабочий каталог vs HEAD
$ git diff HEAD
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644

--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux

# индекс vs HEAD
$ git diff --cached
$
```

Теперь нужно подготовить файл file1 (поместить его в индекс). Давайте сделаем это:

```
$ git add file1
$ git status
# On branch master
# Changes to be committed:
# (use «git reset HEAD <filed>...» to unstage)
#
# modified: file1
```

Здесь дублируется ситуация, описанная рисунком 8.1. Поскольку file1 теперь добавлен в индекс, рабочий каталоги и индекс синхронизированы и между ними не будет разницы. Однако теперь есть разница между HEAD-версией (то есть репозитарием) и версией, которая находится в рабочем каталоге и индексе.

```
# рабочий каталог vs индекс
$ git diff

# рабочий каталог vs HEAD
$ git diff HEAD
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux

# индекс vs HEAD
$ git diff --cached
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux
```

Если вы теперь запустите git commit, новая фиксация поместит изменения, показанные последней командой (git diff --cached), в репозитарий. Команда git diff --cached является синонимом команды git diff --staged.

Теперь давайте посмотрим, что произойдет, если вы отредактировали файл file1 перед фиксацией? Давайте посмотрим!

```
$ echo «baz» > file1

# рабочий каталог vs индекс
$ git diff
diff --git a/file1 b/file1
index d90bda0..7601807 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-quux
+baz

# рабочий каталог vs HEAD
$ git diff HEAD
diff --git a/file1 b/file1
```

```
index 257cc56..7601807 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+baz

# индекс vs HEAD
$ git diff --cached
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux
```

Все три diff-операции показывают теперь какую-то из форм различия. Так какая же версия будет фиксирована? Запомните, **git commit** работает с содержимым индекса, то есть будет зафиксирована версия файла со словом «quux».

```
$ git commit -m "quux uber alles"
[master]: created f8ae1ec: «quux uber alles»
1 files changed, 1 insertions(+), 1 deletions(-)
```

Теперь в хранилище объектов есть две фиксации. Давайте испробуем общую форму команды **git diff** и сравним обе эти две фиксации:

```
# предыдущая HEAD-версия против текущей
$ git diff HEAD^ HEAD
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@

-foo
+quux
```

Эта разница подтверждает, что предыдущая фиксация изменила file1: строка «foo» была заменена строкой «quux».

Неужели сейчас все синхронизировано? Нет! Ведь копия файла в рабочем каталоге содержит строку «baz».

```
$ git diff
diff --git a/file1 b/file1
index d90bda0..7601807 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-quux
+baz
```

8.3. Команда git diff и диапазоны фиксаций

Существуют еще две дополнительных формы команды git diff и они требуют отдельного объяснения, особенно на фоне команды git log.

Команда git diff поддерживает синтаксис двух точек для представления разницы между двумя фиксациями. Следующие две команды являются эквивалентными:

```
$ git diff master bug/pr-1
$ git diff master..bug/pr-1
```

К сожалению, синтаксис двух точек в git diff означает нечто другое, чем тот же синтаксис в команде git log, с которой вы познакомились в главе 6.

Стоит сравнить **git diff** и **git log**, поскольку эти команды по-разному обрабатывают диапазоны фиксаций. Вот что нужно иметь в виду:

- Git diff не беспокоится об истории файлов, которые она сравнивает, также она ничего не хочет знать о ветках.
- Git log «чувствует», как один файл изменился, чтобы стать другим, например, когда две ветки разделились и что произошло в каждой из веток.

Команды **log** и **diff** осуществляют две фундаментально разные операции. Команда log оперирует с набором фиксаций, а diff сравнивает две разные конечные точки. Представьте следующую последовательность событий:

- Кто-то создает новую ветку на основании ветки master и называет ее bug/pr-1.
- Тот же разработчик добавляет строку «Fix Problem report 1» в один из файлов в ветке bug/pr-1
- Тем временем другой разработчик исправлял баг pr-3 в ветке master и добавил строку «Fix Problem report 3» в тот же файл, но в ветке master

Если быть предельно кратким, то был изменен один и тот же файл, но в разных ветках. Если посмотреть на изменения в ветках на высоком уровне, то можно увидеть, когда была создана ветка bug/pr-1 и когда было внесено каждое изменение:

```
$ git show-branch master bug/pr-1
* [master] Added a bug fix for pr-3.
 ! [bug/pr-1] Fix Problem Report 1
--
* [master] Added a bug fix for pr-3.
 + [bug/pr-1] Fix Problem Report 1
*+ [master^] Added Bob's fixes.
```

Если вы введете команду git log -p master..bug/pr-1, вы увидите одну фиксацию, поскольку синтаксис master..bug/pr-1 означает все фиксации, которые есть в bug/pr-1, но которых нет в master. Эта команда «перемотает» события назад в точку, в которой ветка bug/pr-1 произошла от ветки master, она не покажет, что произошло в ветке master с тех пор:

```
$ git log -p master..bug/pr-1
commit 8f4cf5757a3a83b0b3dbecd26244593c5fc820ea
Author: Федя Колобков <kolobok@lobok.com>
Date: Wed May 14 17:53:54 2020 -0500
Fix Problem Report 1
diff --git a/ready b/ready
index f3b6f0e..abbf9c5 100644
--- a/ready
```

```
+++ b/ready
@@ -1,3 +1,4 @@
stupid
znill
frot-less
+Fix Problem report 1
```

Команда git diff master..bug/pr-1, в отличие от git log, покажет полный набор различий между двумя деревьями, представленными головами веток master и bug/pr-1. История не имеет значения, учитывается только текущее состояние файлов.

```
$ git diff master..bug/pr-1
diff --git a/ready b/ready
index f3b6f0e..abbf9c5 100644
--- a/ready
+++ b/ready
@@ -1,4 +1,4 @@
 stupid
 znill
 frot-less
-Fix Problem report 3
+Fix Problem report 1
```

Перефразировать вывод git diff можно так: вы можете преобразовать файл в ветке master к версии bug/pr-1, удалив строку «Fix Problem report 3», а затем добавив строку «Fix Problem report 1» в файл.

Как видите, эта разница содержит фиксации из обоих веток. Посмотрите на рис. 8.1, чтобы было понятнее.

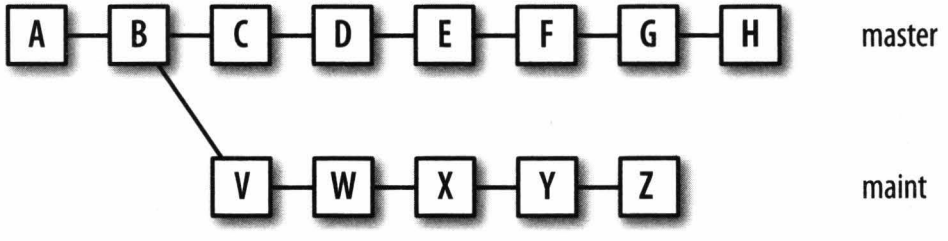


Рис. 8.2. История git diff

В этом случае у нас есть две ветки - master и maint. Команда git log master..maint представляет 5 отдельных фиксаций - V, W, ..., Z. С другой стороны, команда git diff master..maint представляет разницу в деревьях между H и Z и в эту разницу войдут фиксации C, D, ... H и V...Z (всего 11 фиксаций).

Команды git log и git diff принимают форму `фиксация1..фиксация2`, чтобы показать симметричную разницу. Однако у `git log фиксация1..фиксация2` и у `git diff фиксация1..фиксация2` будут разные результаты.

Как было показано в главе 6, команда `git log фиксация1..фиксация2` покажет фиксации, достижимые из каждой фиксации, но не из обоих. А это означает, что git log master..main из предыдущего примера покажет фиксации C, D, ..., H и V, ..., Z.

Симметричная разница в git diff покажет разницу между фиксацией, которая является общим предком (или базой слияния) фиксации1 и фиксации2. Если посмотреть на рис. 8.1, то `git diff master..maint` комбинирует изменения в фиксациях V, W, ..., Z.

8.4. Команда git diff с ограничением пути

По умолчанию **git diff** оперирует со всей структурой каталога, представленного заданным объектом каталога. Однако, вы можете использовать технику ограничения пути, которую вы уже применяли при работе с командой **git log**, для ограничения вывода команды **git diff** определенным подмножеством репозитария. Например, рассмотрим следующую команду:

```
$ git diff --stat master~5 master
Documentation/git-add.txt                  | 2 +-
Documentation/git-cherry.txt               | 6 +++++
Documentation/git-commit-tree.txt          | 2 +-
Documentation/git-format-patch.txt         | 2 +-
Documentation/git-gc.txt                   | 2 +-
Documentation/git-gui.txt                  | 4 +-
Documentation/git-ls-files.txt             | 2 +-
Documentation/git-pack-objects.txt         | 2 +-
Documentation/git-pack-redundant.txt       | 2 +-
```

```
Documentation/git-prune-packed.txt          | 2 +-
Documentation/git-prune.txt                 | 2 +-
Documentation/git-read-tree.txt             | 2 +-
Documentation/git-remote.txt                | 2 +-
Documentation/git-repack.txt                | 2 +-
Documentation/git-rm.txt                    | 2 +-
Documentation/git-status.txt                | 2 +-
Documentation/git-update-index.txt          | 2 +-
Documentation/git-var.txt                   | 2 +-
Documentation/gitk.txt                      | 2 +-
builtin-checkout.c                          | 7 ++++-
builtin-fetch.c                             | 6 ++--
git-bisect.sh                               | 29
++++++++++++-------------
t/t5518-fetch-exit-status.sh                | 37 +++++++++++++
++++++++++++++++++
23 files changed, 83 insertions(+), 40 deletions(-)
```

Чтобы ограничить вывод только изменениями в документации (Documentation), используйте следующую команду:

```
$ git diff --stat master~5 master Documentation
Documentation/git-add.txt                   | 2 +-
Documentation/git-cherry.txt                | 6 ++++++
Documentation/git-commit-tree.txt           | 2 +-
Documentation/git-format-patch.txt          | 2 +-
Documentation/git-gc.txt                    | 2 +-
Documentation/git-gui.txt                   | 4 ++--
Documentation/git-ls-files.txt              | 2 +-
Documentation/git-pack-objects.txt          | 2 +-
Documentation/git-pack-redundant.txt        | 2 +-
Documentation/git-prune-packed.txt          | 2 +-
Documentation/git-prune.txt                 | 2 +-
Documentation/git-read-tree.txt             | 2 +-
Documentation/git-remote.txt                | 2 +-
Documentation/git-repack.txt                | 2 +-
Documentation/git-rm.txt                    | 2 +-
Documentation/git-status.txt                | 2 +-
Documentation/git-update-index.txt          | 2 +-
Documentation/git-var.txt                   | 2 +-
Documentation/gitk.txt                      | 2 +-
19 files changed, 25 insertions(+), 19 deletions(-)
```

Конечно, вы можете просмотреть разницы и для единственного файла:

```
$ git diff master~5 master Documentation/git-add.txt
diff --git a/Documentation/git-add.txt b/Documentation/git-add.
txt
index bb4abe2..1afd0c6 100644
--- a/Documentation/git-add.txt
+++ b/Documentation/git-add.txt
@@ -246,7 +246,7 @@ characters that need C-quoting. `core.
quotepath` configuration can be
used to work this limitation around to some degree, but
backslash,
double-quote and control characters will still have problems.
-See Also
+SEE ALSO
 --------
 linkgit:git-status[1]
 linkgit:git-rm[1]
```

В следующем примере, также взятом из собственного репозитария Git, опция -S»строка» производит поиск последних 50 фиксаций в ветке master для изменений, содержащий указанную строку.

```
$ git diff -S»octopus» master~50
diff --git a/Documentation/RelNotes-1.5.5.3.txt b/
Documentation/RelNotes-1.5.5.3.txt
new file mode 100644
index 0000000..f22f98b
--- /dev/null
+++ b/Documentation/RelNotes-1.5.5.3.txt
@@ -0,0 +1,12 @@
+GIT v1.5.5.3 Release Notes
+==========================
+
+Fixes since v1.5.5.2
+--------------------
+
+ * «git send-email --compose» did not notice that non-ascii
contents
+ needed some MIME magic.
+
+ * «git fast-export» did not export octopus merges correctly.
+
+Also comes with various documentation updates.
```

Использование опции -S уже было рассмотрено в главе 6, поэтому вы должны быть уже знакомы с ней.

Сравнение как Subversion (SVN) и Git получают разницы

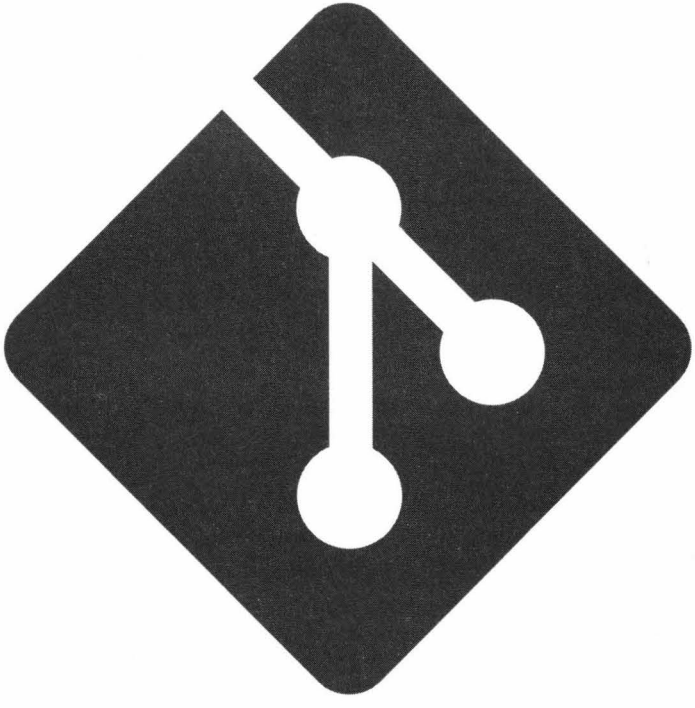
Большинство систем, таких как CVS или SVN, отслеживает серию версий и хранит просто изменения между каждой парой файлов. Этот метод позволяет сохранить дисковое пространство и сократить издержки. Внутренне такие системы проводят много времени, думая о вещах как "серии изменений между A и B". Когда вы обновляете свои файлы центрального репозитария, например, SVN помнит, что при предыдущим обновлении файла репозитарий был в версии r1095, а сейчас репозитарий находится в версии r1123. Таким образом, сервер должен отправить вам разницу между r1095 и r1123. Как только ваш SVN-клиент получит эти разницы, он может включить их в вашу рабочую копию и воспроизвести r1123 (с помощью этого трюка SVN избегает отправки вам всего содержания всех файлов при каждом отправлении).

Чтобы сохранить дисковое пространство, SVN также хранит свой собственный репозитарий как серию изменений на сервере. Когда вы просите показать разницу между r1095 и r1123, он ищет отдельные различия для каждой версии между теми двумя версиями, объединяет их в одну большую разность и отправляет вам результат. Но Git работает иначе.

Как вы видели, в Git каждая фиксация содержит дерево, которое является списком всех файлов и каталогов, затронутых этой фиксацией. Каждое дерево независимо от всех других деревьев. Пользователи Git все еще говорят о различиях и патчах, поскольку они все еще очень полезны. Все же, в Git разница и патч - производные, а не фундаментальные данные. Если вы посмотрите в каталог .git, то не увидите ни одной разницы, а если заглянуть в репозитарий SVN, то он состоит преимущественно из различий.

Git, как и SVN, способен получить полный набор различий между двумя произвольными состояниями. Но SVN нужно смотреть на каждую версию между версиями r1095 и r1123, в то время как Git не заботится о промежуточных шагах.

У каждой ревизии есть свое собственное дерево, но Git не требует его для создания разницы, поскольку он может оперировать со снимками полного состояния каждой из этих двух версий. Эта простая разница в системах хранения - одна из наиболее важных причин того, что Git быстрее, чем другие RCS.



Глава 9.

Слияния



Git - это распределенная система управления версиями (Distributed VCS). Она позволяет разным разработчикам, например, когда один находится в Японии, а другой в - Нью-Джерси, независимо вносить изменения в проект. В любое время эти два разработчика могут комбинировать свои изменения без наличия центрального репозитария. В этой главе вы узнаете, как объединить две или больше линий разработки.

Слияние объединяет две или более веток истории фиксаций. Чаще всего слияние объединяет всего две ветки, хотя Git поддерживает объединение большего числа веток (3, 4 и даже больше).

В Git слияние должно происходит в пределах одного репозитария - все ветки, которые нужно объединить, должны находиться в одном и том же репозитарии. Как эти ветки попали в репозитарий - не важно (как вы увидите в главе 12, в Git есть механизм, позволяющий ссылаться на удаленные репозитарии и помещать удаленные ветки в ваш текущий рабочий репозитарий).

Когда изменения в одной ветке не конфликтуют с изменениями, найденными во второй ветке, Git вычисляет результат слияния и создает новую фиксацию, которая представляет новое, унифицированное состояние. Но когда ветки конфликтуют, Git отмечает такие конкурирующие изменения

как «необъединенные» (unmerged), а разработчик уже сам принимает решение, какое из изменений нужно принять.

9.1. Примеры слияния

Чтобы объединить ветку branch с веткой other_branch, вам нужно сначала переключиться на ветку branch, а затем выполнить слияние новой ветки в текущую:

```
$ git checkout branch
$ git merge other_branch
```

Теперь давайте рассмотрим пример слияний, один без конфликтов и один с существенными перекрытиями. Для упрощения примеров в этой главе мы будем использовать методы, представленные в главе 7.

Подготовка к слиянию

Прежде, чем начать слияние, нужно произвести генеральную уборку в вашем рабочем каталоге. Во время обычного слияния Git создает новые версии файлов и помещает их в ваш рабочий каталог, когда слияние будет завершено. Git также использует индекс для хранения временных и промежуточных версий файлов во время операции слияния.

Если вы изменили файлы в вашем рабочем каталоге или вы модифицировали индекс посредством **git add** или **git rm**, тогда у вашего репозитария «грязный» рабочий каталог или индекс. Если вы начнете объединение в таком «грязном» состоянии, Git не сможет правильно скомбинировать изменения из других веток.

> **Примечание. Не всегда нужно начинать с "чистого" каталога. Git выполнить слияние, например, если измененные («грязные») файлы в рабочем каталоге и затрагиваемые слиянием файлы не пересекаются. Однако все же рекомендуется взять за правило начинать каждое слияние с чистого рабочего каталога и индекса, что в будущем позволит избежать многих проблем.**

Объединение двух веток

Давайте рассмотрим простейший сценарий. Мы настроим репозитарий, содержащий один файл, затем создадим две ветки, а затем - выполним объединение этих двух веток.

```
$ git init
Initialized empty Git repository in /tmp/conflict/.git/
$ git config user.email «kolobok@lobok.com»
$ git config user.name «Федя Колобков»
$ cat > file
Line 1 stuff
Line 2 stuff
Line 3 stuff
^D
$ git add file
$ git commit -m «Initial 3 line file»
Created initial commit 8f4d2d5: Initial 3 line file
1 files changed, 3 insertions(+), 0 deletions(-)
create mode 100644 file
```

Давайте создадим другую фиксацию на ветке master:

```
$ cat > other_file
Here is stuff on another file!
^D
$ git add other_file
$ git commit -m «Another file»
Created commit 761d917: Another file
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 other_file
```

Итак, у нас есть репозитарий, в котором создана одна ветка с двумя фиксациями, каждая фиксация представляет новый файл. Далее, давайте переключимся на новую ветку и модифицируем первый файл:

```
$ git checkout -b alternate master^
Switched to a new branch «alternate»
$ git show-branch
* [alternate] Initial 3 line file
 ! [master] Another file
```

```
--
+ [master] Another file
*+ [alternate] Initial 3 line file
```

Здесь видно, что ветка alternate изначально создана из фиксации master^, то есть из предпоследней фиксации в ветке.

Внесите простые изменения в файл, чтобы у вас было что объединять, и затем фиксируйте эти изменения. Помните, что лучше начать слияние с «чистого» рабочего каталога.

```
$ cat >> file
Line 4 alternate stuff
^D
$ git commit -a -m «Add alternate's line 4»
Created commit b384721: Add alternate's line 4
1 files changed, 1 insertions(+), 0 deletions(-)
```

Теперь у нас есть две ветки и каждая из них содержит определенную работу разработчика. Второй файл был сначала добавлен в ветку master, а затем он был модифицирован в ветке alternate. Поскольку изменения не относятся к одной и той же части общего файла, слияние должно пройти гладко и без происшествий.

Операция **git merge** чувствительна к контексту. Ваша текущая ветка всегда является целевой веткой, а другая ветка (или ветки) будет объединена с текущей веткой. В этом случае ветка alternate должна быть объединена с веткой master, поэтому сначала нужно переключиться на ветку master:

```
$ git checkout master
Switched to branch «master»

$ git status
# Мы на ветке master
nothing to commit (working directory clean)

# Теперь мы готовы к слиянию!
$ git merge alternate
Merge made by recursive.
file | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

Чтобы просмотреть, что же произошло, вы можете использовать утилиту просмотра графа фиксации, которая является частью git log:

```
$ git log --graph --pretty=oneline --abbrev-commit
* 1d51b93... Merge branch 'alternate'
|\
| * b384721... Add alternate's line 4
* | 761d917... Another file
|/
* 8f4d2d5... Initial 3 line file
```

Концепция графов фиксации была описана ранее в главе 6, поэтому в общих чертах выведенный граф вам должен быть понятен. В отличие от графов, представленных в главе 6, на этом графе наиболее последние фиксации находятся сверху, а не справа. ID начальной фиксации - 8f4d2d5. Каждая ветка содержит одну фиксации (761d917 и b384721), вторая ветка начинается на фиксации 8f4d2d5, затем обе ветки объединяются в одно целое снова - на фиксации 1d51b93.

Примечание. Использование утилиты git log --graph - отличная альтернатива графическим утилитам вроде gitk. Визуализация, предоставляемая git log --graph, отлично подходит для медленных терминалов.

Технически Git производит каждое слияние симметрично для создания одной комбинированной фиксации, которая добавляется в вашу текущую ветку. Другая ветка никак не изменяется слиянием. Поскольку фиксация слияния добавляется только в вашу текущую ветку, вы можете сказать «Я объединил несколько веток *в эту* ветку».

Слияние с конфликтом

Слияние, по сути, проблематично, поскольку оно обязательно соединяет потенциально конфликтные изменения от разных линий разработки. Изменения в одной ветке могут быть или подобными или радикально отличаться от других веток. Изменения могут затрагивать или те же самые файлы, что и в первой ветке, или же непересекающийся набор файлов. Git может обработать все эти ситуации, но он часто требует вмешательства разработчика для разрешения конфликтов.

Давайте рассмотрим сценарий, в котором слияние приводит к конфликту. Мы начнем с результатов слияния из предыдущего раздела и внесем независимые и конфликтующие изменения в ветки master и alternate. Затем мы объединим ветку alternate с веткой master, разрешим конфликт и получим окончательный результат.

В ветке **master** давайте создадим новую версию файла file, добавив в него несколько дополнительных строк, а затем фиксируем изменения:

```
$ git checkout master
$ cat >> file
Line 5 stuff

Line 6 stuff
^D
$ git commit -a -m «Add line 5 and 6»
Created commit 4d8b599: Add line 5 and 6
1 files changed, 2 insertions(+), 0 deletions(-)
```

Теперь на ветке **alternate** модифицируем тот же файл, но немного иначе:

```
$ git checkout alternate
Switched branch «alternate»

$ git show-branch
* [alternate] Add alternate's line 4
 ! [master] Add line 5 and 6
--
+ [master] Add line 5 and 6
*+ [alternate] Add alternate's line 4

# Редактируем файл
$ cat >> file
Line 5 alternate stuff
Line 6 alternate stuff
^D
$ cat file
Line 1 stuff
Line 2 stuff
Line 3 stuff
Line 4 alternate stuff
Line 5 alternate stuff
Line 6 alternate stuff
```

```
$ git diff
diff --git a/file b/file
index a29c52b..802acf8 100644
--- a/file
+++ b/file
@@ -2,3 +2,5 @@ Line 1 stuff
Line 2 stuff
Line 3 stuff
Line 4 alternate stuff
+Line 5 alternate stuff
+Line 6 alternate stuff
```

```
$ git commit -a -m "Add alternate line 5 and 6"
Created commit e306e1d: Add alternate line 5 and 6
1 files changed, 2 insertions(+), 0 deletions(-)
```

Теперь давайте просмотрим сценарий. Текущая ветка истории выглядит так:

```
$ git show-branch
* [alternate] Add alternate line 5 and 6
 ! [master] Add line 5 and 6
--
* [alternate] Add alternate line 5 and 6
+ [master] Add line 5 and 6
*+ [alternate^] Add alternate's line 4
```

Чтобы продолжить, переключитесь на ветку master и попытайтесь осуществить слияние:

```
$ git checkout master
Switched to branch «master»
```

```
$ git merge alternate
Auto-merged file
CONFLICT (content): Merge conflict in file
Automatic merge failed; fix conflicts and then commit the
result.
```

Когда произошел конфликт слияния, вы должны внимательно исследовать причину конфликта командой **git diff**. В нашем случае причина конфликта кроется в содержимом файла file:

```
$ git diff
diff --cc file
index 4d77dd1,802acf8..0000000
--- a/file
+++ b/file
@@@ -2,5 -2,5 +2,10 @@@ Line 1 stuf
Line 2 stuff
Line 3 stuff
Line 4 alternate stuff
++<<<<<<< HEAD:file
+Line 5 stuff
+Line 6 stuff
++=======
+ Line 5 alternate stuff
+ Line 6 alternate stuff
++>>>>>>> alternate:file
```

Команда git diff показывает разницы между файлом file в вашем рабочем каталоге и индексе. В стиле обычной команды diff измененный контент представлен между <<<<<< и ======, альтернатива - между ===== и >>>>>>>.

Дополнительные знаки плюс и минус используются в комбинированном формате diff для обозначения изменений из разных источников относительно финальной версии.

Предыдущий вывод показывает, что конфликт покрывает строки 5 и 6, именно эти строки отличаются в версиях файла из двух разных веток. Теперь давайте попытаемся разрешить конфликт. Когда вы разрешаете конфликт слияния, вы вольны выбрать любое решение на свое усмотрение. Вы можете выбрать один из вариантов файла, вы можете смешать оба варианта или даже создать что-то полностью новое и отличающееся от представленных вариантов. Хотя последняя возможность довольно странная, но она тоже доступна.

В этом случае я выбираю строку из каждой ветки в качестве моей разрешенной версии. Отредактированная версия файла теперь выглядит так:

```
$ cat file
Line 1 stuff
Line 2 stuff
Line 3 stuff
Line 4 alternate stuff
```

```
Line 5 stuff
Line 6 alternate stuff
```

Если вы довольны разрешением конфликта, введите команду **git add** для добавления файла в индекс и подготовить его для фиксации слияния:

```
$ git add file
```

После разрешения конфликтов и подготовки окончательной версии каждого файла в индексе с использованием **git add**, вам нужно зафиксировать слияние с использованием команды **git commit**. Git откроет ваш любимый редактор с шаблоном сообщения, который будет выглядеть примерно так:

```
Merge branch 'alternate'

Conflicts:

file
#
# It looks like you may be committing a MERGE.
# If this is not correct, please remove the file
# .git/MERGE_HEAD
# and try again.
#

# Please enter the commit message for your changes.
# (Comment lines starting with '#' will not be included)
# On branch master
# Changes to be committed:
# (use «git reset HEAD <file>...» to unstage)
#
# modified: file
#
```

Как обычно, строки, которые начинаются решеткой (#), являются комментариями и предназначены исключительно для вашей информации.

Когда вы выйдете из редактора, Git сообщит об успешном создании новой фиксации слияния:

```
$ git commit

# Редактирование сообщения фиксации слияния
```

```
Created commit 7015896: Merge branch 'alternate'

$ git show-branch
! [alternate] Add alternate line 5 and 6
 * [master] Merge branch 'alternate'
--
- [master] Merge branch 'alternate'
+* [alternate] Add alternate line 5 and 6
```

Просмотреть результат фиксации слияния можно так:

```
$ git log
```

9.2. Работа с конфликтами слияния

Как показано в предыдущем примере, при наличии конфликтов слияние не может быть выполнено автоматически.

Давайте рассмотрим другой сценарий с конфликтом для изучения утилит, предоставляемых Git для разрешения конфликтов. Пусть у нас будет файл hello со строкой «hello», потом мы создадим две разных ветки с двумя разными вариантами этого файла.

```
$ git init
Initialized empty Git repository in /tmp/conflict/.git/

$ echo hello > hello
$ git add hello
$ git commit -m «Initial hello file»
Created initial commit b8725ac: Initial hello file
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 hello

$ git checkout -b alt
Switched to a new branch «alt»

$ echo world >> hello
$ echo 'Yay!' >> hello
```

```
$ git commit -a -m «One world»
Created commit d03e77f: One world
1 files changed, 2 insertions(+), 0 deletions(-)

$ git checkout master
$ echo worlds >> hello
$ echo 'Yay!' >> hello
$ git commit -a -m «All worlds»
Created commit eddcb7d: All worlds
1 files changed, 2 insertions(+), 0 deletions(-)
```

Одна ветка содержит «world», а другая - «worlds». Как и в предыдущем примере, если вы переключитесь на ветку master и попытаетесь выполнить слияние в нее ветки **alt**, вы столкнетесь с конфликтом:

```
$ git merge alt
Auto-merged hello
CONFLICT (content): Merge conflict in hello
Automatic merge failed; fix conflicts and then commit the
result.
```

Как мы и ожидали, Git предупредил о наличии конфликта, найденного в файле hello.

Обнаружение конфликтных файлов

Когда у нас есть только один файл, то все понятно. Но что если в каталоге есть множество файлов с конфликтами? Как их обнаружить? Git заботливо помечает проблемные файлы как conflidted или unmerged.

Вы можете использовать команду git status или git ls-files -u для получения списка файлов, слияние которых выполнить не получилось:

```
$ git status
hello: needs merge
# On branch master
# Changed but not updated:
# (use «git add <file>...» to update what will be committed)
#
# unmerged: hello
#
```

```
no changes added to commit (use «git add» and/or «git commit
-a»)
```

```
$ git ls-files -u
100644 ce013625030ba8dba906f756967f9e9ca394464a 1 hello
100644 e63164d9518b1e6caf28f455ac86c8246f78ab70 2 hello
100644 562080a4c6518e1bf67a9f58a32a67bff72d4f00 3 hello
```

После этого вы можете использовать git diff, чтобы понять, почему не произошло слияние.

Исследование конфликтов

Когда имеются конфликты, рабочая копия каждого проблемного файла добавляется маркерами слияния. Теперь результирующий конфликтный файл похож на это:

```
$ cat hello
hello
<<<<<<< HEAD:hello
worlds
=======
world
>>>>>>> 6ab5ed10d942878015e38e4bab333daff614b46e:hello
Yay!
```

Маркеры слияния показывают две возможные версии конфликтного участка файла. В первой версии участок содержит слово «worlds», в другой версии этот участок содержит слово «world». Вам нужно просто выбрать одну из фраз, удалить маркеры конфликта и выполнить команды **git add** и **git commit**. Но давайте рассмотрим некоторые другие функции Git, помогающие нам в решении конфликтов.

> **Примечание. Маркеры слияния (<<<<<<<<, ======== и >>>>>>>>) генерируются автоматически. Замените эти маркеры собственным текстом для разрешения конфликта.**

git diff с конфликтами

У Git есть специальный, ориентированный для слияния вариант команды git diff, предназначенный для отображения изменений против обоих родителей одновременно. Пример:

```
$ git diff
diff --cc hello
index e63164d,562080a..0000000
--- a/hello
+++ b/hello
@@@ -1,3 -1,3 +1,7 @@@
hello
++<<<<<<< HEAD:hello
+worlds
++=======
+ world
++>>>>>>> alt:hello
Yay!
```

Что это все означает? Это просто комбинация двух разниц: одна по сравнению с родителем (HEAD), одна по сравнению со вторым родителем (alt). Чтобы сделать вещи проще, Git предоставляет второму родителю специальное имя - MERGE_HEAD.

Вы можете сравнить HEAD и MERGE_HEAD с версией файла в рабочем каталоге:

```
$ git diff HEAD
diff --git a/hello b/hello
index e63164d..4e4bc4e 100644
--- a/hello
+++ b/hello
@@ -1,3 +1,7 @@
hello
+<<<<<<< HEAD:hello
worlds
+=======
+world
+>>>>>>> alt:hello
Yay!
```

А теперь сравним с MERGE_HEAD:

```
$ git diff MERGE_HEAD
diff --git a/hello b/hello
index 562080a..4e4bc4e 100644
--- a/hello
+++ b/hello

@@ -1,3 +1,7 @@
hello
+<<<<<<< HEAD:hello
+worlds
+=======
world
+>>>>>>> alt:hello
Yay!
```

Примечание. В новой версии Git, команда git diff --ours является синонимом git diff HEAD, поскольку она показывает разницу между «нашей» версией и объединенной версией. Аналогично, git diff MERGE_HEAD может быть написано как git diff --theirs. Вы можете использовать git diff --base, чтобы увидеть комбинированный набор изменений, начиная с базы слияния, что иначе может быть записано как:

```
$ git diff $(git merge-base HEAD MERGE_HEAD)
```

Если выстроить в одну линию две разницы рядом, весь текст, кроме столбцов + будет одинаковым. Git выводит главный текст только один раз и выводит столбцы + рядом друг с другом.

У конфликта, найденного командой git diff, есть два столбца с информацией, предшествующие каждой строке вывода. Знак «+» в столбце указывает на добавление строки, а знак «-» - удаление строки, пробел - это строка без изменения. Первый столбец показывает то, что изменилось по сравнению с вашей версией, а второй столбец - то, что изменилось по сравнению с другой версией. Строки маркеров конфликта новые в обоих версиях, поэтому они отмечаются знаками ++.

Строки world и worlds новые только в какой-то одной версии, поэтому они отмечаются одним знаком + в соответствующей колонке.

Предположим, что вы отредактировали файл так:

```
$ cat hello
hello
worldly ones
Yay!
```

Затем просмотрим на новый вывод git diff:

```
$ git diff
diff --cc hello
index e63164d,562080a..0000000
--- a/hello
+++ b/hello
@@@ -1,3 -1,3 +1,3 @@@
hello
- worlds
-world
++worldly ones
Yay!
```

Альтернативно, вы можете выбрать одну из оригинальных версий:

```
$ cat hello
hello
world
Yay!
```

Вывод git diff будет таким:

```
$ git diff
diff --cc hello
index e63164d,562080a..0000000
--- a/hello
+++ b/hello
```

Стоп! Нечто странное происходит здесь. Куда делась строка world, которая была добавлена к базовой версии? Почему не отображается строка worlds, удаленная из HEAD-версии? Поскольку вы разрешили конфликт в пользу версии MERGE_HEAD, Git сознательно опускает разность, потому что она думает, что она вам больше не нужна.

Запуск git diff на проблемном файле покажет только те разделы, которые действительно содержат конфликт. В большом файле с многочисленными изменениями, рассеянными повсюду, далеко не во всех изменениях есть конфликт. Поэтому когда вы пытаетесь разрешить конфликт, вы редко заботитесь о тех секциях, поэтому git diff сокращает вывод, не показывая «неинтересные» изменения.

У этой оптимизации есть немного запутывающий побочный эффект: как только вы разрешаете что-то, что раньше было конфликтом, просто выбирая один вариант или другой, данная (ранее конфликтная) секция уже больше не показывается. После этого Git смотрит на эту секцию так, как будто бы в ней вообще не было конфликтов.

На самом деле - это больше побочный эффект реализации, чем намеренная функция, но вы должны помнить: git diff показывает только те секции файла, которые еще конфликтуют, вы можете использовать эту команду для отслеживания конфликтов, которые еще не исправлены.

git log с конфликтами

Некоторые специальные опции команды **git log** могут помочь вам обнаружить, какие изменения откуда пришли и почему. Попробуйте ввести эту команду:

```
$ git log --merge --left-right -p

commit <eddcb7dfe63258ae4695eb38d2bc22e726791227
Author: Федя Колобков <kolobok@lobok.com>
Date: Wed Oct 22 21:29:08 2020 -0500

All worlds

diff --git a/hello b/hello
index ce01362..e63164d 100644

--- a/hello
+++ b/hello
@@ -1 +1,3 @@
hello
+worlds
+Yay!
```

```
commit >d03e77f7183cde5659bbaeef4cb51281a9ecfc79
Author: Федя Колобков <kolobok@lobok.com>
Date: Wed Oct 22 21:27:38 2020 -0500

One world

diff --git a/hello b/hello
index ce01362..562080a 100644
--- a/hello
+++ b/hello
@@ -1 +1,3 @@
hello
+world
+Yay!
```

Если вы задумывались, когда, почему, как и кем была добавлена строка worlds в файл, вы должны использовать команду git log.

Команда **git log** предоставляет следующие опции:

- **--merge** показывает только те фиксации, которые относятся к конфликтующим файлам
- **--left-right** показывает <, если фиксация пришла «слева» («наша» версия) или >, если фиксация пришла «справа» («их» версия, то есть ветка, которую вы объединяете с текущей)
- **-p** показывает сообщение фиксации и патч, связанный с каждой фиксацией

Если ваш репозитарий более сложный и есть конфликты в нескольких файлах, можно указать интересующие вас файлы (или просто файл) в командной строке:

```
$ git log --merge --left-right -p hello
```

В этой главе из демонстрационных соображений были приведены совсем небольшие примеры. Конечно, на практике ваши файлы будут значительно больше и сложнее. Один из способов уменьшить головную боль при больших слияниях с огромными конфликтами заключается в использовании нескольких небольших фиксаций с четко определенными эффектами. Небольшие фиксации и более частые циклы слияния позволяют немного упростить разрешение конфликтов.

Как Git отслеживает конфликты

Как Git отслеживает всю информацию о конфликтном слиянии? Есть несколько частей:

- .git/MERGE_HEAD - содержит SHA1-хэш фиксации слияния. Не нужно использовать этот SHA1 непосредственно. Git сам заглянет в этот файл, когда речь зайдет о MERGE_HEAD.
- .git/MERGE_MSG - содержит сообщение слияния по умолчанию, введенное вами при использовании команды git commit после разрешения конфликтов.
- index содержит три копии каждого конфликтного файла: база слияния, «наша» версия и «их версия». Этим трем копиям назначены, соответственно, номера 1, 2 и 3.
- Конфликтная версия (с маркерами слияния) не сохраняется в индексе. Она сохранена в файле в вашем рабочем каталоге. Когда вы запустите git diff без параметров, сравнение всегда будет между тем, что в индексе и тем, что находится в рабочем каталоге.

Чтобы увидеть, как хранятся записи индекса, вы можете использовать команду git ls-files:

```
$ git ls-files -s
100644 ce013625030ba8dba906f756967f9e9ca394464a 1 hello
100644 e63164d9518b1e6caf28f455ac86c8246f78ab70 2 hello
100644 562080a4c6518e1bf67a9f58a32a67bff72d4f00 3 hello
```

Опция -s показывает все файлы со всеми этапами. Если нужно просмотреть только конфликтные файлы, используйте опцию -u.

Другими словами, файл hello был сохранен три раза, у каждой версии есть уникальный хэш. Вы можете просмотреть определенную версию файла, используя команду git cat-file:

```
$ git cat-file -p e63164d951
hello
worlds
Yay!
```

Вы также можете использовать специальный синтаксис для команды git diff для сравнения разных версий файла. Например, если вы хотите просмотреть, что изменилось между базой слияния и версией из ветки, которую вы объединяете с текущей, используйте команду:

```
$ git diff :1:hello :3:hello
diff --git a/:1:hello b/:3:hello
index ce01362..562080a 100644
--- a/:1:hello
+++ b/:3:hello
@@ -1 +1,3 @@
hello
+world
+Yay!
```

Примечание. Начиная с версии 1.6.1, команда git checkout принимает опции --ours (наша ветка) или --theirs («их» ветка) для упрощения работы с ветками. Эти же две опции вы можете использовать во время разрешения конфликтов.

Номера этапов используются, чтобы назвать версию, отличную от git diff --theirs, которая показывает различия между их версией и результирующей (объединенной) версией в вашем рабочем каталоге. Объединенная версия еще не находится в индексе, поэтому у нее даже нет номера.

Поскольку вы полностью отредактировали и разрешили рабочую версию в пользу их версии, теперь не должно быть никакого различия:

```
$ cat hello
hello
world
Yay!

$ git diff --theirs
* Unmerged path hello
```

Все, что осталось - это необъединенное напоминание о пути, нужно добавить его в индекс.

Завершение разрешения конфликта

Давайте сделаем одно изменение в файл hello, прежде, чем объявить его объединенным:

```
$ cat hello
hello
everyone
Yay!
```

Теперь, когда файл полностью объединен и разрешен, команда git add уменьшит индекс до одной версии файла hello:

```
$ git add hello
$ git ls-files -s
100644 ebc56522386c504db37db907882c9dbd0d05a0f0 0 hello
```

Тот одинокий 0 между SHA1 и путем говорит вам, что число этапа для непроблемного файла равно 0.

Вы должны обработать все конфликтные файлы в индексе. Вы не можете выполнить фиксацию, если есть хотя бы один неразрешенный конфликт.

> **Примечание. Будьте осторожны, чтобы не добавить (командой git add) файлы с маркерами конфликта. Да, это очистит конфликт в индексе и разрешит вам фиксацию, но ваш файл не будет корректен.**

Итак, введите **git commit** для фиксации результата и **git show**, чтобы просмотреть фиксацию слияния:

```
$ cat .git/MERGE_MSG
Merge branch 'alt'
Conflicts:
hello

$ git commit

$ git show
commit a274b3003fc705ad22445308bdfb172ff583f8ad
```

```
Merge: eddcb7d... d03e77f...
Author: Федя Колобков <kolobok@lobok.com>
Date: Wed Oct 22 23:04:18 2020 -0500

      Merge branch 'alt'

      Conflicts:
            hello

diff --cc hello
index e63164d,562080a..ebc5652
--- a/hello
+++ b/hello
@@@ -1,3 -1,3 +1,3 @@@
hello
- worlds
-world
++everyone
Yay!
```

Вы должны заметить три интересных факта, когда смотрите на фиксацию слияния:

- Есть новая, вторая, строка в заголовке, которая говорит: «Обычно нет никакой потребности показывать родителя фиксации в git log или git show, так как есть только один родитель и это обычно тот, который прибывает сразу после него в журнале. Но фиксации обычно имеют два (иногда больше) родителя и важно это понимать. Следовательно, git log и git show всегда выведут SHA1 каждого предка.
- Автоматически сгенерированное сообщение журнала фиксации услужливо отмечает список конфликтующих файлов. Это может быть полезно позже, если окажется, что определенная проблема была вызвана вашим слиянием. Обычно проблемы, вызванные слиянием, заключаются в файлах, которые вы объединяли вручную.
- Разница фиксации слияния - не нормальная разница. Она всегда находится в формате «конфликтующего» слияния. Успешное слияние вообще не является изменением, это просто комбинация других изменений, которые уже появились в истории. Таким образом, показ содержимого фиксации слияния показывает только те части, которые отличаются от одной из объединенных веток, а не весь набор изменений.

Отмена или перезапуск слияния

Если вы запустите операцию слияния, но потом по некоторым причинам решите отказаться от нее, Git предоставляет легкий способ отмены операции. Перед выполнением последней операции git commit используйте команду:

```
$ git reset --hard HEAD
```

Эта команда восстанавливает рабочий каталог и индекс в состояние, предшествующее команде **git merge**.

Если же нужно отметить слияние после того, как оно уже завершено (после того, как оно представлено новой фиксацией слияния), используйте команду:

```
$ git reset --hard ORIG_HEAD
```

Перед началом операции слияния Git сохраняет оригинальную ветку HEAD в ссылке ORIG_HEAD, чтобы можно было выполнить откат слияния.

Вы должны быть предельно осторожными. Если вы не запустили слияния с чистого рабочего каталога и индекса, вы можете потерять любое незафиксированное изменение, имеющееся в вашем рабочем каталоге.

Вы можете инициировать запрос git merge с грязным рабочим каталогом, но, если вы выполните git reset –hard, ваше грязное состояние не будет полностью восстановлено и вы можете потерять данные. Ведь вы же запрашиваете «жесткий» (--hard) сброс к состоянию HEAD, значит, вы его и получите (подробнее о сбросе Git мы поговорим в следующей главе).

Начиная с версии Git 1.6.1, у вас есть другая возможность. Если вы испортили разрешение конфликта и хотите вернуться к исходному состоянию конфликта, чтобы попытаться разрешить его снова, используйте команду **git checkout -m**.

9.3. Стратегии слияния

До сих пор наши примеры были достаточно простыми, поскольку у нас было только две ветки. Так давайте же их немного усложним.

Предположите, что вместо всего одного человека, работающего в вашем репозитарии, есть трое программистов. Пусть их зовут Белка, Бобик и Козлик. Каждый из них может внести изменения как фиксации в три отдельных одноименных (Allice, Bob, Cal) ветки в вашем репозитарии.

В конечном счете, программисты разработают репозитарий с историей фиксации, приведенной на рис. 9.1.

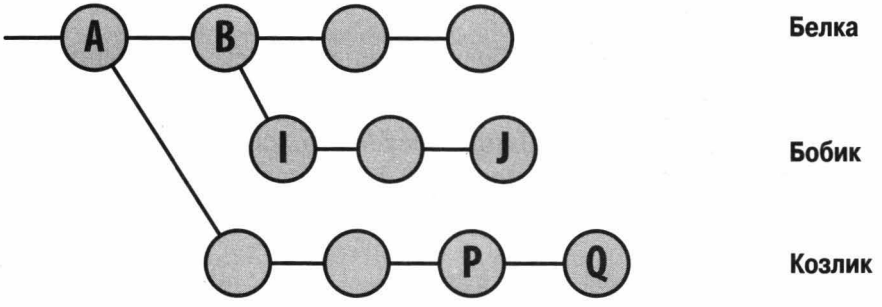


Рис. 9.1. Перекрестная установка слияния

Представьте, что Козлик начал проект и Белка присоединилась к нему. Белка работала над проектом некоторое время, после чего к ним присоединился Бобик. Тем временем Козлик продолжал работать над его собственной версией.

В конечном счете, ветка Белки была объединена с веткой Бобика, а Бобик продолжал работать без слияния изменений Белка обратно в свое дерево. Теперь должно быть три разных истории ветвления.

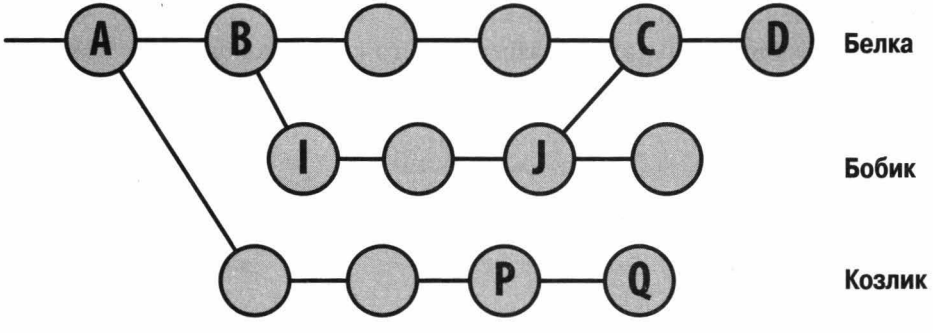


Рис. 9.2. После того, как ветка Белка объединена в ветку Бобика

Теперь давайте представим, что Бобик хочет получить последние изменения Козлика. Диаграмма выглядит предельно сложной, но эта часть относительно проста. Пройдитесь по дереву от ветки Бобика, через ветку Белки, пока не достигнете точки, где она впервые отделилась от Козлика. Это будет точка A, это база слияния между ветками Бобика и Козлика. Чтобы получить изменения Козлика, Бобику нужно взять набор изменений между базой (основой) слияния (A) и последней версией Козлика (Q), что приводит к фиксации K. Результат - история, показанная на рис. 9.3.

Примечание. Основу слияния между двумя и более ветками можно всегда определить командой git merge-base.

Пока неплохо.

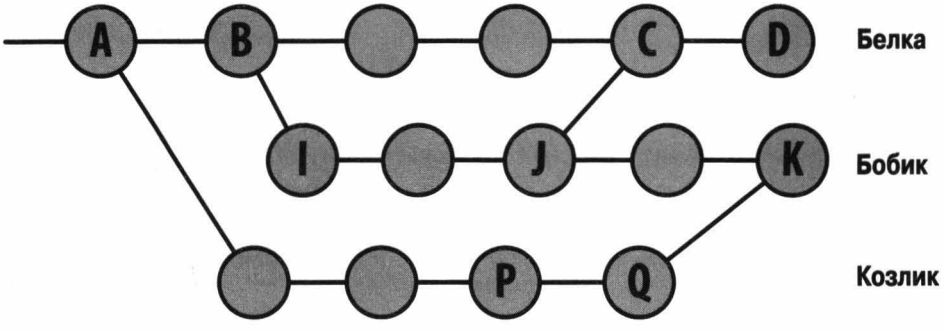


Рис. 9.3. После того, как ветка Бобика слилась в ветку Козлика

Белка теперь решает, что она тоже хочет получить последние изменения Козлика, но она не понимает, что Бобик уже объединил дерево Козлика со своим деревом. Поэтому она просто объединяет дерево Козлика в свое. Получилась история, изображенная на рис. 9.4.

Затем Белка понимает, что Бобик сделал некоторую работу (L) и хочет объединиться еще раз? Что же будет базой слияния на этот раз (между L и E)?

К сожалению, ответ неоднозначен. Если вы отслеживаете дерево, вы можете подумать, что исходная версия от Козлика - хороший выбор. Но это не целесообразно: ведь у Белки и Бобика уже есть самые последние версии Козлика. Если вы запросите различия от исходной версии Козлика до последней версии Бобика, то она будет также содержать более новые изменения

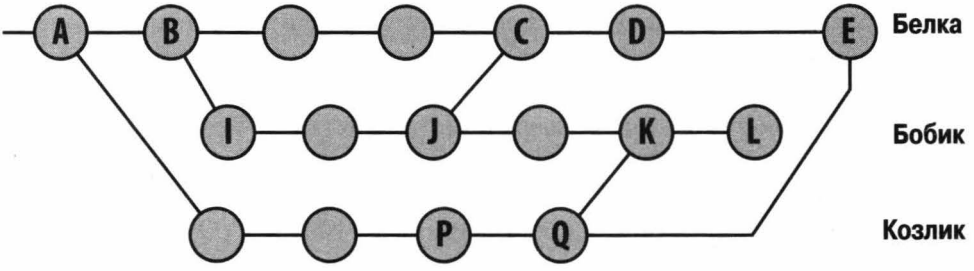


Рис. 9.4. После того, как ветка Белки слилась в ветку Козлика

Козлика, которые уже есть у Белки, что, вероятно, приведет к конфликту слияния.

Что, если использовать последнюю версию Козлика в качестве основы? Это лучше, но все еще не совершенно. Если вы возьмете разницу между последними версиями Козлика и Бобика, вы получите *все* изменения Бобика. Но у Белки уже есть *некоторые* изменения Бобика, таким образом, вы также получите конфликт слияния.

А что, если использовать версию, которую Белка получила в последний раз от Бобика (J)? Создание разницы отсюда до последней версии Бобика включает только последние изменения от Бобика, которые и являются тем, что вам нужно. Но это также включает изменения от Козлика, которые уже есть у Белки!

Что делать?

Этот тип ситуации называется перекрестным слиянием, потому что были объединены и вперед между ответвлениями. Если бы изменения перемещались только в одном направлении (например, от Козлика к Белке, а затем - к Бобику, но никогда от Бобика к Белке или от Белки - к Козлику), то слияние было бы простым. К сожалению, жизнь не всегда настолько проста.

Разработчики Git первоначально написали механизм объединения двух веток с фиксацией слияния, но сценарии, подобные этому, заставили их разработать более умный подход. Для обработки таких сложных сценариев используются настраиваемые *стратегии слияния*.

Давайте рассмотрим различные стратегии слияния и разберемся, как их применять.

Вырожденные слияния

Есть два общих вырожденных сценария, которые приводят к слияниям: актуальная ветка и перемотка. Поскольку ни один из этих сценариев фактически не представляет новую фиксацию слияния после выполнения git merge, некоторые эксперты не считают, что они являются настоящими стратегиями слияния.

- *Актуальная ветка (Already up-to-date).* Когда все фиксации из одной ветки (HEAD) уже присутствуют в вашей целевой ветке, говорят, что целевая ветка актуальна. Как результат, в вашу ветку не будут добавлены новые фиксации. Например, если вы осуществляете слияние и немедленно пытаетесь отправить тот же запрос слияния, вам сообщат, что ваша ветка уже актуальна.

```
# Ветка alternate уже добавлена в master
$ git show-branch
! [alternate] Add alternate line 5 and 6
 * [master] Merge branch 'alternate'
--
  - [master] Merge branch 'alternate'
+* [alternate] Add alternate line 5 and 6

# Пытаемся выполнить слияние alternate в master снова
$ git merge alternate
Already up-to-date.
```

- *Перемотка вперед (Fast-forward).* Слияние перемотки вперед (далее просто перемотки) происходит, когда HEAD вашей ветки уже полностью присутствует и представлена в другой ветке. Это - инверсия предыдущего случая. Поскольку ваша HEAD уже присутствует в другой ветке, Git просто прикрепляет к вашей HEAD новые фиксации из другой ветки. Git затем перемещает HEAD так, чтобы она указывала на заключительную, новую фиксацию. Естественно, индекс и ваш рабочий каталог также будут скорректированы надлежащим образом, чтобы отразить новое, заключительное состояние фиксации. В главе 12 мы более подробно рассмотрим перемотку, которая часто используется на отслеживаемых ветках.

Для Git важно обработать эти случаи, не представляя фактические фиксации. Представьте, что бы произошло в случае перемотки, если бы Git создал фиксацию? Слияние ветки A в B сначала произвело бы рисунок 9.5. Тогда слияние B в A произвело бы рисунок 9.6, а еще одно обратное слияние приведет к рисунку 9.7.

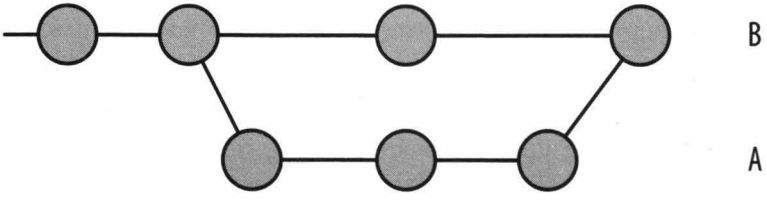


Рис. 9.5. Первое несходящееся слияние

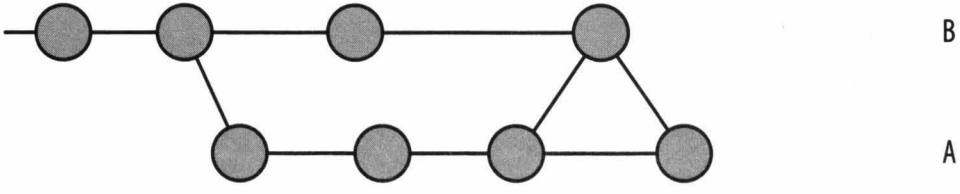


Рис. 9.6. Второе несходяшееся слияние

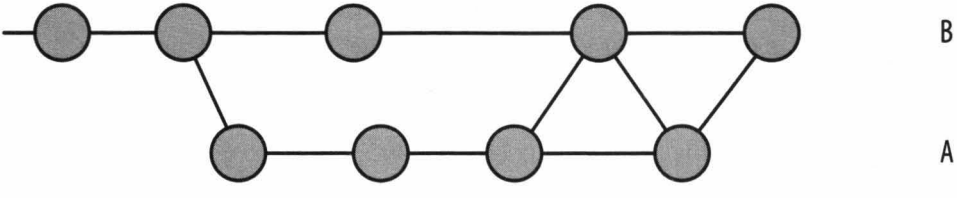


Рис. 9.7. Третье несходяшееся слияние

Каждое новое слияние - новая фиксация, таким образом, последовательность никогда не будет сходиться на устойчивом состоянии и показывать, что две ветки идентичны.

Обычные слияния

Обычные стратегии слияния производят заключительную фиксацию, которая добавляется к вашей текущей ветке, которая представляет объединенное состояние слияния.

- *Стратегия решения (resolve).* Воздействует только на две ветки, определяя местоположение общего предка как основания слияния и выполняя прямое трехстороннее слияние, применяя изменения от основы слияния до HEAD другой ветки на текущей ветке. Эта стратегия имеет интуитивный смысл.
- *Рекурсивная стратегия (recursive).* Подобна стратегии решения, в которой она может присоединиться к двум веткам сразу. Однако она разработана для обработки сценария, в котором есть больше чем одна основа слияния между двумя ветками. В таких случаях Git формирует временное слияние всех общих основ слияния и затем использует его в качестве основы, из которой можно получить слияние двух данных ответвлений через нормальный алгоритм трехстороннего слияния. Временная база слияния будет отброшена, а заключительное состояние слияния будет зафиксировано в вашей целевой ветке.
- *Стратегия осьминога (octopus).* Специально предназначена, чтобы объединить вместе больше чем две ветки одновременно. Концептуально, это довольно просто; Она многократно вызывает рекурсивную стратегию, один раз для каждой объединяемой ветки. Однако, эта стратегия не может обработать слияние, требующее любой формы разрешения конфликтов, что потребовало бы взаимодействия с пользователем. В таком случае вы вынуждены будете сделать серию нормальных слияний, разрешая конфликты шаг за шагом.

Рекурсивные слияния

Простое перекрестное слияние приведено на рис. 9.8.

Узлы **a** и **b** являются основами слияния для слияния между ветками A и B. Любой узел может использоваться в качестве базы слияния, и это бы привело к разумным результатам. В этом случае рекурсивная стратегия объединила бы **a** и **b** во временную базу слияния, используя ее в качестве базы слияния для A и B.

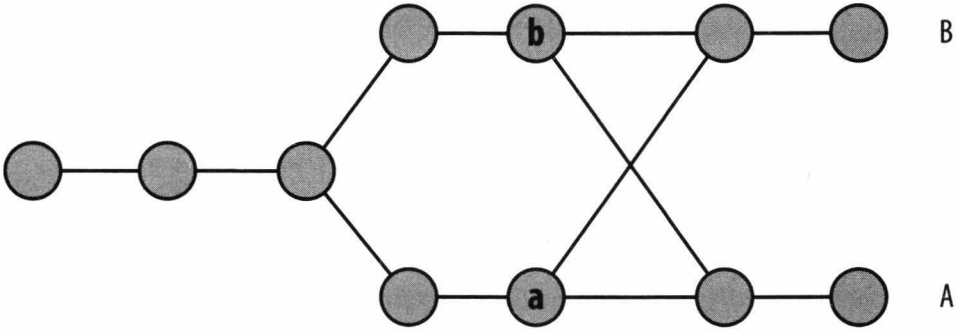


Рис. 9.8. Простое перекрестное слияние

Поскольку у **a** и **b** могла быть та же проблема, объединение их могло потребовать другого слияния еще более старых фиксаций. Именно поэтому этот алгоритм называют рекурсивным.

Слияния осьминога

Обобщенность и элегантность дизайна - главные причины, по которым Git поддерживает объединение нескольких веток в одну. В Git фиксация может быть без родителя (начальная фиксация), с одним родителем (обычная фиксация) и более чем с одним родителем (фиксация слияния). Как только у вас есть больше, чем один родитель, нет никакой видимой причины ограничивать это число родителей только двумя, таким образом, структуры данных Git поддерживают множество родителей. Стратегия осьминога - естественное следствие общего проектного решения, позволяющего использование нескольких родителей фиксации.

Слияния осьминога отлично выглядят в диаграммах, поэтому пользователи Git стараются использовать их максимально часто. Только представьте себе порыв эндорфинов, которые разработчик получает при слиянии шести веток программы в одну! Помнимо симпатичного графа слияния осьминога, фактически, ничего не делают дополнительно. С тем же успехом вы можете использовать многократные фиксации слияния, по одной за ветку и получить тот же результат.

Специальные слияния

Есть две специальных стратегии слияния, о которых вы должны знать, потому что они могут иногда помочь вам решить странные проблемы. Если вы не сталкивались с какими-либо странными проблемами, вы можете пропустить этот раздел, но вы вернетесь к нему позже. Данные две стратегии называются **ours** (наши) и **subtree** (поддерево).

Эти стратегии слияния производят заключительную фиксацию, добавленную в вашу текущую ветку, которая представляет объединенное состояние слияния.

- **Ours**. Стратегия Ours объединяет любое число веток, но фактически отбрасывает изменения от тех веток и использует только файлы текущей ветки. Результат этой стратегии - слияние, идентичное текущей HEAD, но любые другие ветки будут зарегистрированы как родители фиксации.
- **Subtree**. Стратегия subtree выполняет слияние в другую ветку, но все в той ветке объединяется в определенное поддерево текущего дерева. Вы не определяете само поддерево, Git определяет его автоматически.

9.4. Применение стратегий слияния

Как Git знает или определяет, какую стратегию использовать? Или, если вам не нравится выбор Git, как определить иной вариант?

Git пытается выбрать наименее простой и дешевый (в плане системных ресурсов) алгоритм. Поэтому он сначала пытается использовать методы актуальной ветки или перемотки, чтобы устранить тривиальные, простые сценарии, если это возможно.

Если вы указываете более, чем одну ветку, которая будет слита в текущую ветку, у Git нет выбора, он попытается использовать стратегию осьминога, поскольку это единственная стратегия, приспособленная к слиянию нескольких веток в одну.

Если все эти варианты не увенчались успехом, Git должен использовать стратегию по умолчанию, которая будет работать во всех других сценариях. По умолчанию используется *стратегия решения*.

При перекрестном слиянии, где есть больше, чем одно возможное основание слияния, Git использует стратегию слияния так: выбирает одну из возможных основ слияния (или последнее слияние от ветки Бобика или последнее слияние от ветки Козлика) и надеется на лучшее. Это не настолько плохо, как звучит. Часто оказывается, что Белка, Бобик и Козлик работали над разными частями кода. В этом случае Git обнаружит, что он повторно объединяет некоторые изменения и просто пропустит их, что позволит ему избежать конфликта. Если же есть небольшие изменения, вызывающие конфликт, их будет достаточно просто обработать разработчику.

Поскольку стратегия решения больше не является стратегией по умолчанию, Белка хотела бы использовать ее явно, что можно сделать с помощью следующего запроса:

```
$ git merge -s resolve Bob
```

В 2005 Фредрик Куивинен внес новое понятие - понятие рекурсивной стратегии слияния, которая с тех пор стала стратегией по умолчанию. Она более общая, чем стратегия решения и именно она привела к меньшему числу конфликтов на ядре Linux. Она также хорошо обрабатывает слияния с переименованиями.

В следующей статье можно подробно ознакомиться с рекурсивной стратегией:

http://codicesoftware.blogspot.com/2011/09/merge-recursive-strategy.html.

Общий алгоритм этой стратегии выглядит так:

- Составляется список всех общих предков, начиная с самого свежего
- Берется текущая фиксация самого первого предка
- Выполняется слияние текущей фиксации со следующим предком. В результате получим виртуальную фиксацию, которую принимаем за текущую
- Выполняем предыдущую операцию до тех пор, пока не закончится список общих предков. Именно поэтому данная стратегия и называется рекурсивной

Этот метод вызывают «рекурсивный», потому что могут быть дополнительные итерации, в зависимости от того, сколько уровней перекрещивания и оснований слияния встретит Git. И это работает. Мало того, что рекурсивный метод имеет интуитивный смысл, он приводит к меньшему количеству конфликтов, чем более простая стратегия решения. Именно поэтому рекурсивная стратегия стала стратегией слияния по умолчанию. Конечно, независимо от того, какую стратегию решит использовать Белка, заключительная история будет такой же (см. Рис. 9.9).

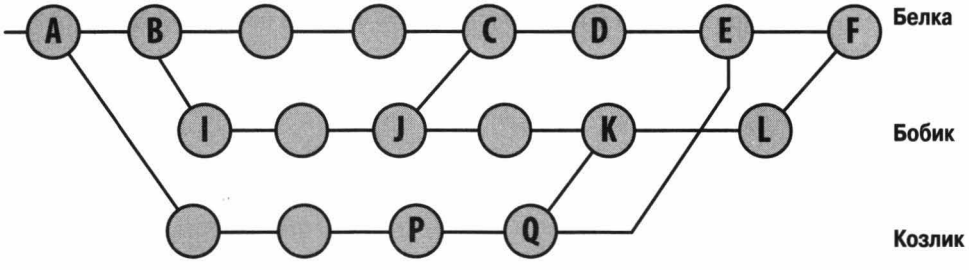


Рис. 9.9. Окончательная история слияния

Использование стратегий ours и subtree

Вы можете использовать эти две стратегии слияния вместе. Например, когда-то давно программа gitweb (которая теперь часть git) была разработана за пределами главного репозитария git.git. Но в версии 0a8f4f вся ее история слилась с репозитарием git.git и стала поддеревом gitweb. Если вы хотите сделать нечто подобное, вам нужно выполнить следующие действия:

- Скопируйте текущие файлы проекта gitweb.git в подкаталог вашего проекта
- Зафиксируйте их как обычно
- Заберите изменения из проекта gitweb.git, используя стратегию ours:

```
$ git pull -s ours gitweb.git master
```

Мы используем ours, поскольку знаем, что у нас есть последняя версия файлов и вы поместили их точно туда, куда хотели (а не туда, куда бы их поместила рекурсивная стратегия).

- В будущем вы можете продолжить забирать последние изменения из проекта gitweb.git, используя стратегию subtree:

```
$ git pull -s subtree gitweb.git master
```

Поскольку файлы уже существуют в вашем репозитарии, Git автоматически знает, какое поддерево нужно использовать и произведет обновления без конфликтов.

Драйверы слияния

Каждая из стратегий слияния, описанных в этой главе, использует базовый драйвер слияния для решения и объединения каждого отдельного файла. Драйвер слияния принимает имена трех временных файлов, которые представляют общего предка, целевую версию ветки и другую версию ветки файла. Драйвер изменяет целевую версию ветки, чтобы получить объединенный результат.

Драйвер text оставляет уже привычные трехсторонние маркеры слияния (<<<<<<<, ======= и >>>>>>>).

Драйвер слияния binary просто сохраняет целевую версию файла целевой ветки и помещает файл как конфликтный в индексе. Это вынуждает Вас обработать двоичные файлы вручную.

Заключительный драйвер слияния (union) просто оставляет все строки из обоих версий в объединенном файле.

Используя механизм атрибута, Git может связать определенные файлы или образцы файла с определенными драйверами слияния. Большинство текстовых файлов будут обработаны драйвером text, а большинство двоичных файлов будут обработаны драйвером binary. Если вам мало этих драйверов, вы можете создать и определить свой собственный драйвер слияния и связать его со своими определенными файлами.

Как Git думает о слияниях

Сначала, автоматическая поддержка слияния Git кажется не чем иным, как волшебством, особенно по сравнению с более сложными и подверженными ошибкам шагами слияния, необходимыми в другой VCS.

Давайте посмотрим на то, что делает все это волшебство возможным.

Слияния и объектная модель Git

В большинстве VCS у одной фиксации есть только один родитель. В такой системе, когда вы объединяете some_branch в my_branch, вы создаете новую фиксацию на my_branch с изменениями от some_branch. С другой стороны, если вы объединяете my_branch в some_branch, тогда будет создана фиксация на на some_branch, содержащая изменения от my_branch. То есть слияние ветки A в ветку B и ветки B в ветку A - это две разные операции.

Однако разработчики Git заметили, что каждая из этих двух операций приводит к одному и тому же набору файлов, когда все закончится. Естественный способ выразить эту операцию звучит так: «Объедините все изменения от some_branch и another_branch в единственную ветку».

В Git слияние приводит к новому объекту дерева с объединенными файлами, но он также представляет новый объект фиксации только на целевой ветке. После выполнения следующих команд объектная модель будет выглядеть, как показано на рис. 9.10:

```
$ git checkout my_branch
$ git merge some_branch
```

На рис. 9.10 каждый узел Cx - это объект фиксации, а каждый Tx представляет соответствующий объект дерева. Есть одна общая объединенная фиксация (CZC), у которой есть родители фиксации CC и CZ. Получившийся в результате слияния набор файлов - это дерево TZC. Объединенный объект дерева симметрично представляет две исходных ветки. Но потому что my_branch была активной веткой, в которую произошло слияние, была обновлена только она, именно в ней была представлена новая фиксация, а some_branch останется как есть.

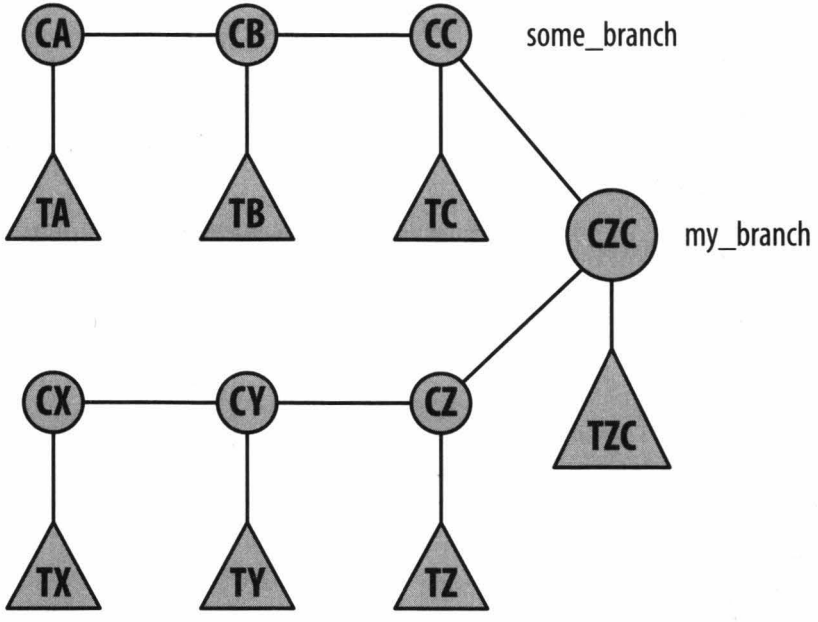


Рис. 9.10. Объектная модель после слияния

Слияние squash

Предположим, что ветка **some_branch** содержала не всего одну новую фиксацию, а 5, 10 или даже сотни фиксаций. В большинстве систем объединение some_branch в my_branch включило бы создание единственной разницы, которая бы была применена в виде одного патча на ветке my_branch, и создание одного нового элемента в истории. Это называется фиксацией squash, поскольку она помещает все отдельные фиксации в одно большое изменение. В результате история some_branch будет потеряна.

В Git вся история фиксации сохраняется с обеих стороны. Рис. 9.10 демонстрирует, как пользователю Git расплачиваются за эту сложность. Если бы Git использовать squash-фиксацию, вы бы не видели всей этой схемы, а история my_bransh, возможно, была бы просто прямой линией.

Примечание. При желании Git может выполнить squash-фиксацию. Для этого используйте опцию --squash в командах git merge или git pull. Однако будьте осторожны - у нас появятся сложности при будущих слияниях после выполнения squash-фиксации.

Добавленная сложность может покататься неудачной, но это того стоит. Например, эта функция означает, что команды **git blame** и **git bisect**, описанные в главе 6, станут намного мощнее, чем эквивалентные в других системах. Рекурсивная стратегия слияния тоже доступна благодаря этой сложности.

Примечание. Хотя операции слияния считают оба родителя равными, вы можете выбрать первого родителя, указав специальную опцию --first-parent, которую поддерживают некоторые команды вроде git log и gitk.

Почему не выполнять слияние каждого изменения одно за другим?

Вы можете спросить, можно ли использовать простую, линейную историю с каждой отдельной фиксацией. Git может взять все фиксации от ветки some_branch и применить их, одну за другой, к ветке my_branch. Но это несколько другая вещь.

Важное наблюдение за историей фиксации Git заключается в том, что каждая версия в истории реальна (подробнее об этом мы поговорим в другой главе).

Если вы примените последовательность чьих-либо патчей поверх вашей версии, то вы создадите серию полностью новых версий с объединением их изменений и ваших. По-видимому, в конце концов, вы получите окончательную версию, но что относительно всех новых промежуточных версий? В реальности эти версии никогда не существовали: никто и никогда не производил эти фиксации таким образом и никто не может сказать наверняка, работали ли они когда-либо.

Git хранит историю так, чтобы вы могли позже посмотреть, как выглядели ваши файлы в определенный момент в прошлом. Если некоторые объединенные фиксации отражают версии файла, которые никогда не существовали, то вы потеряли причину наличия подробной истории.

Именно поэтому слияния в Git работают иначе. Если вы спросите «Что было за 5 минут до фиксации?», ответ будет неоднозначен. Вместо этого нужно уточнить, о какой ветке (my_branch или some_branch) идет речь, и тогда Git сможет дать истинный ответ.

Даже если вы хотите применить стандартное поведение слияния истории, Git может также применить последовательность патчей (см. гл. 14), как описано здесь. Этот процесс описан в главе 10.

Глава 10.

Изменение коммитов (фиксаций)



Фиксация записывает историю вашей работы и сохраняет ваши изменения священными, но сама фиксация не создана из камня. Git предоставляет несколько инструментов и команд, специально предназначенных, чтобы помочь вам изменять и улучшать историю фиксации, каталогизируемую в вашем репозитарии.

Есть много причин, почему вам нужно изменить или переделать фиксацию или последовательность фиксации в общем:

- Вы можете решить проблему прежде, чем это станет наследством
- Вы можете разбить большое изменение на несколько маленьких, тематических фиксаций. С другой стороны, вы можете объединить отдельные изменения в одну большую фиксацию
- Вы можете просматривать обратную связь и предложения
- Вы можете переупорядочить фиксации в последовательность, которая соответствует требованиям сборки
- Вы можете упорядочить фиксации в более логическую последовательность
- Вы можете удалить код отладки, который вы случайно зафиксировали.

Как будет показано в главе 12, которая объясняет, как предоставить общий доступ к репозитарию, существует много разных причин изменений фиксаций, прежде, чем вы опубликуете ваш репозитарий.

10.1. Зачем производить изменения истории

Когда дело доходит до управления историей разработки есть несколько философских концепций.

Одну философию можно было бы назвать реалистической историей: сохраняется каждая фиксация и ничего не изменяется.

Один из вариантов - мелкомодульная реалистическая история, где вы фиксируете каждое изменение, гарантируя, что каждый шаг сохранен для потомства. Другой вариант - дидактическая реалистическая история, где вы не торопитесь и фиксируете свою лучшую работу только в удобные и подходящие моменты.

Учитывая возможность корректировки историю, можно очистить плохое промежуточное решение или произвести реконструкцию фиксаций в более логический поток. Вы можете создать «идеальную» историю.

Как разработчику, вам может пригодиться полная, мелкомодульная история, поскольку она может предоставить «археологические» детали о том, как была разработана та или иная идея. Полная история может обеспечить понимание появления ошибки или объяснить исправление ошибки. Анализ истории может привести к пониманию, как работает разработчик или целая команда и как можно улучшить процесс разработки.

Многие из этих деталей могут быть потеряны, если пересмотренная история удалит промежуточные шаги. Разработчик сразу пришел к такому хорошему решению? Или потребовалось несколько итераций? Какова причина ошибки? Если удалить промежуточные фиксации, ответы на эти вопросы будут потеряны.

С другой стороны, ведение только чистой истории, показывающей четко определенные шаги, доставит удовольствие при чтении и изменении. Нет потребности волноваться о каком-то неоптимальном шаге в истории репо-

зитария. Кроме того, другие разработчики, читающие историю, смогут изучить лучший метод и стиль разработки.

Так что лучше? Подробная история без потери информации? Или все же лучше чистая история? Git предоставляет вам возможность очистить фактическую историю и превратить ее в идеальную и чистую историю прежде, чем репозитарий будет опубликован публично. Хранить полную историю или идеальную историю - это дело сугубо вашей политики проекта.

Предостережение об изменении истории

Вы не должны стесняться изменять и улучшать историю фиксации вашего репозитария, пока никакой другой разработчик не получил ваш репозитарий. Чтобы быть предельно точным, вы можете изменять определенную ветку своего репозитария, пока ни у кого нет копии этой ветки.

Например, скажем, вы работали над своей основной веткой и сделали фиксации A - D доступными другому разработчику, как показано на рис. 10.1. Как только вы сделаете свою историю разработки доступной для другого разработчика, она станет называться «опубликованной историей».

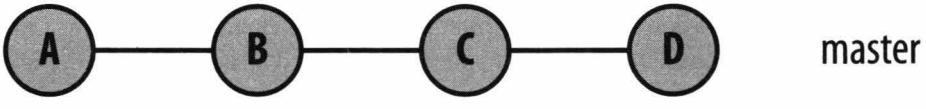


Рис. 10.1. Опубликованная история

Теперь представим, что вы продолжили разработку и создали новые фиксации W - Z, но это уже неопубликованная история в той же ветке. Эта ситуация изображена на рис. 10.2.

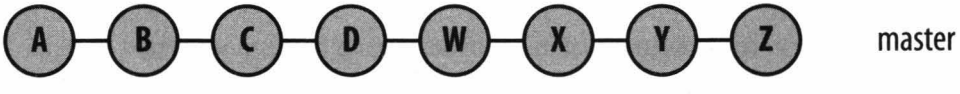


Рис. 10.2. Ваша неопубликованная история

В этой ситуации вы должны быть очень осторожными и оставить все фиксации, предшествующие фиксации W, в покое. Однако, пока вы не переопубликуете свою основную ветку, нет никакой причины не изменять фиксации W - Z.

Вы могли бы закончить с новой и улучшенной историей фиксации, показанной на рис. 10.3. В этом примере примере фиксации X и Y были объединены в одну новую фиксацию; фиксация W была немного изменена, что привело к новой фиксации W'; фиксация Z была перемещена назад; P - это новая фиксация.

В этой главе рассмотрены техники, с помощью которых вы можете изменить и улучшить историю фиксации.

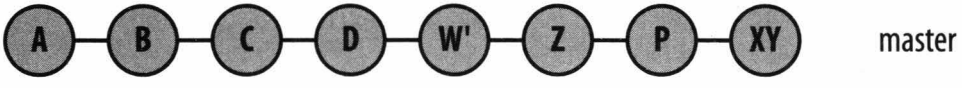


Рис. 10.3. Ваша новая история

10.2. Использование команды git reset

Команда **git reset** переводит ваш репозитарий и рабочий каталог в известное состояние. В частности, git reset корректирует ссылку HEAD заданной фиксации и обновляет индекс, чтобы он соответствовал той фиксации. При желании git reset может также изменить ваш рабочий каталог, чтобы отразить версию вашего проекта, представленную данной фиксацией.

Вы можете толковать команду git reset как «разрушительную», потому что она может перезаписать и уничтожить изменения в вашем рабочем каталоге. Действительно, данные могут быть потеряны. Даже если у вас есть резервные копии ваших файлов, вы не можете восстановить свою работу. Однако цель этой команды - восстановить известное состояние HEAD, индекса и рабочего каталога.

У команды git reset есть три основных опции: --soft, --mixed и --hard.

```
git reset --soft фиксация
```

Параметр --soft изменяет ссылку HEAD на точку, заданную фиксацией. Содержимое вашего индекса и рабочего каталога останется без изменений. Эта версия команды обладает «наименьшим» эффектом, изменяя состояние только символической ссылки.

```
git reset --mixed фиксация
```

Параметр --mixed изменяет ссылку HEAD так, что она будет указывать на заданную фиксацию. Содержимое вашего индекса также будет модифицировано, но файлы в рабочем каталоге останутся без изменений. Эта версия команды позволяет сохранить изменения, которые вы произвели в рабочем каталоге, но еще не поместили в индекс. Параметр --mixed используется по умолчанию для команды git reset.

```
git reset --hard фиксация
```

Этот вариант устанавливает ссылку HEAD на заданную фиксацию, содержимое индекса и рабочего каталога будет изменено так, чтобы соответствовать указанной фиксации. Это есть жесткий сброс, при котором возможна потеря данных, как в индексе, так и в рабочем каталоге.

Эффекты этих параметров подытожены в таблице 10.1.

Таблица 10.1. Эффекты опций команды git reset

Опция	HEAD	Индекс	Рабочий каталог
--soft	Да	Нет	Нет
--mixed	Да	Да	Нет
--hard	Да	Да	Да

Команда **git reset** также сохраняет исходное значение HEAD в ссылке ORIG_HEAD. Это полезно, например, если вы хотите использовать оригинальную HEAD как базу для некоторой последующей фиксации.

В терминах объектной модели команда **git reset** перемещает текущую ветку HEAD в графе фиксаций в определенную фиксацию. Если вы укажите опцию --hard, ваш рабочий каталог тоже будет изменен.

Давайте посмотрим на некоторые примеры того, как работает git reset.

В следующем примере файл foo.c был случайно помещен в индекс. Использование команду git status подтверждает, что файл будет фиксироваться:

```
$ git add foo.c
# Упс! Я не хотел добавлять foo.c!

$ git status
# On branch master
# Changes to be committed:
# (use «git reset HEAD <file>...» to unstage)
#
# new file: foo.c
#
```

Чтобы избежать фиксации файла, используйте команду git reset:

```
$ git ls-files
foo.c
main.c

$ git reset HEAD foo.c

$ git ls-files
main.c
```

В фиксации HEAD не было пути foo.c. Поэтому выполнение git reset на ветке HEAD для файла foo.c должно звучать так: «Удалите foo.c из индекса».

Другой общий пример использования git reset - простая отмена последней фиксации в ветке. Давайте создадим ветку с двумя фиксациями в ней.

```
$ git init
Initialized empty Git repository in /tmp/reset/.git/
$ echo foo >> master_file
$ git add master_file
$ git commit
```

```
Created initial commit e719b1f: Add master_file to master
branch.
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 master_file

$ echo "more foo" >> master_file
$ git commit master_file
Created commit 0f61a54: Add more foo.
1 files changed, 1 insertions(+), 0 deletions(-)

$ git show-branch --more=5
[master] Add more foo.
[master^] Add master_file to master branch.
```

Предположим теперь, что вторая фиксация неправильная, и вы хотите вернуться обратно. Это классическое применение команды git reset --mixed HEAD^. Вспомните, в главе 6 мы говорили о том, что ссылка HEAD^ указывает на родителя текущей фиксации HEAD ветки master и представляет состояние, предшествующее неправильной фиксации.

```
# По умолчанию используется параметр --mixed
$ git reset HEAD^
master_file: locally modified

$ git show-branch --more=5
[master] Add master_file to master branch.

$ cat master_file
foo
more foo
```

После git reset HEAD^ Git восстановил содержимое master_file и всего рабочего каталога в состояние, которое было до фиксации «Add more foo».

Поскольку опция --mixed сбрасывает индекс, вам нужно заново подготовить любые изменения, которые вы хотите видеть в новой фиксации. Возможно, вам нужно повторно отредактировать master_file и другие файлы, а также выполнить какие-либо другие изменения перед созданием новой фиксации.

```
$ echo «even more foo» >> master_file
$ git commit master_file
Created commit 04289da: Updated foo.
```

```
1 files changed, 2 insertions(+), 0 deletions(-)

$ git show-branch --more=5
[master] Updated foo.
[master^] Add master_file to master branch.
```

Теперь на ветке master создано всего две фиксации, а не три. Если вам не нужно изменять индекс, тогда вы можете использовать опцию --soft:

```
$ git reset --soft HEAD^
$ git commit
```

Предположим, однако, что вы хотите полностью устранить вторую фиксацию и при этом вам все равно, что будет с ее содержимым. В этом случае используйте опцию --hard:

```
$ git reset --hard HEAD^
HEAD is now at e719b1f Add master_file to master branch.

$ git show-branch --more=5
[master] Add master_file to master branch.
```

Данная команда немедленно переводит ветку master в предыдущее состояние, при этом изменяются как индекс, так и рабочий каталог в состояние фиксации HEAD^. Состояние файла master_file в вашем рабочем каталоге будет снова модифицировано, и вы получите исходный файл:

```
$ cat master_file
Foo
```

Хотя все примеры в той или иной форме используют HEAD, вы можете применить **git apply** к любой фиксации в репозитарии. Например, чтобы устранить несколько фиксаций на вашей текущей ветке, вы можете использовать команду git reset --hard HEAD~3 или даже git reset --hard master~3.

Будьте осторожны. Хотя вы можете указать имена ветки, это не то же самое, что и переключение ветки. В случае с git reset вы должны всегда оставаться на одной и той же ветке. Вы можете изменить свой рабочий каталог, чтобы ваш рабочий каталог был похож на HEAD другой ветки, но вы все еще остаетесь на исходной ветке.

Чтобы проиллюстрировать использование **git reset** на других ветках, давайте добавим вторую ветку с названием dev и добавим в нее новый файл.

```
# Мы должны быть на master, но лучше убедиться в этом точно
$ git checkout master
Already on «master»

$ git checkout -b dev
$ echo bar >> dev_file
$ git add dev_file
$ git commit
Created commit 7ecdc78: Add dev_file to dev branch
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 dev_file
```

Вернемся на ветку master, здесь только один файл:

```
$ git checkout master
Switched to branch «master»

$ git rev-parse HEAD
e719b1fe81035c0bb5e1daaa6cd81c7350b73976

$ git rev-parse master
e719b1fe81035c0bb5e1daaa6cd81c7350b73976

$ ls
master_file
```

Использования опции --soft изменит только ссылку HEAD:

```
# Изменяем HEAD, чтобы она указывала на фиксацию dev
$ git reset --soft dev
$ git rev-parse HEAD
7ecdc781c3eb9fbb9969b2fd18a7bd2324d08c2f

$ ls
master_file

$ git show-branch
! [dev] Add dev_file to dev branch
 * [master] Add dev_file to dev branch
--
```

```
+* [dev] Add dev_file to dev branch
```

Конечно, кажется, как будто ветка **master** и ветка **dev** находятся в одной и той же фиксации. И вы все еще находитесь на ветке master, что хорошо - но выполнение этой операции оставляет вещи в специфическом состоянии. Что бы произошло, если бы вы теперь сделали фиксацию? HEAD указывает на фиксацию, в которой есть файл dev_file, но этот файл не находится в ветке master.

```
$ echo «Funny» >> new
$ git add new
$ git commit -m «Which commit parent?»
Created commit f48bb36: Which commit parent?
2 files changed, 1 insertions(+), 1 deletions(-)
delete mode 100644 dev_file
create mode 100644 new

$ git show-branch
! [dev] Add dev_file to dev branch
 * [master] Which commit parent?
--
* [master] Which commit parent?
+* [dev] Add dev_file to dev branch
```

Git корректно добавил новый файл **new** и, очевидно, решил, что dev_file отсутствует в этой фиксации. Но почему Git удалил dev_file? Git прав, dev_file не является частью этой фиксации, но ошибочно говорить, что он был удален, поскольку его никогда там и не было. Итак, почему Git выбрал удалить этот файл? Git использует фиксацию, на которую указывает HEAD тогда, когда была сделана новая фиксация. Давайте посмотрим, как это было:

```
$ git cat-file -p HEAD
tree 948ed823483a0504756c2da81d2e6d8d3cd95059
parent 7ecdc781c3eb9fbb9969b2fd18a7bd2324d08c2f
author Федя Колобков <kolobok@lobok.com> 1229631494 -0600
committer Федя Колобков <kolobok@lobok.com> 1229631494 -0600
```

Какой из родителей совершает ошибку?

Родитель этой фиксации - 7ecdc7, который является вершиной ветки **dev**, а не ветки **master**. Но эта фиксация была сделана на ветке master! Путаница не должна стать неожиданностью, потому что основная HEAD была из-

менена, чтобы указать на голову ветки dev. В этой точке вы можете прийти к заключению, что последняя фиксация полностью поддельная и должна быть удалена. И это хорошо. Такому запутанному состоянию не место в репозитарии.

Как показал ранний пример, вся эта ситуация подходит команды git reset --hard HEAD^. Но перед выполнением этой команды нужно добраться до предыдущей версии HEAD ветки master:

```
# Сначала нужно убедиться, что мы находимся на ветке master
$ git checkout master

# ПЛОХОЙ ПРИМЕР!
# Сброс обратно на предшествующее состояние master
$ git reset --hard HEAD^
```

Так в чем же проблема? Ведь вы просто увидели, что родитель HEAD указывает на dev, а не на предшествующую фиксацию на исходной ветки master.

```
# Да, HEAD^ указывает на HEAD ветки dev.
$ git rev-parse HEAD^
7ecdc781c3eb9fbb9969b2fd18a7bd2324d08c2f
```

Есть несколько способов определения фиксации, к которой будет сброшена ветка master.

```
$ git log
commit f48bb36016e9709ccdd54488a0aae1487863b937
Author: Федя Колобков <kolobok@lobok.com>
Date: Thu Dec 18 14:18:14 2020 -0600

Which commit parent?

commit 7ecdc781c3eb9fbb9969b2fd18a7bd2324d08c2f
Author: Федя Колобков <kolobok@lobok.com>
Date: Thu Dec 18 13:05:08 2020 -0600

Add dev_file to dev branch

commit e719b1fe81035c0bb5e1daaa6cd81c7350b73976
Author: Федя Колобков <kolobok@lobok.com>
```

```
Date: Thu Dec 18 11:44:45 2020 -0600

Add master_file to master branch.
```

Последняя фиксация (e719b1f) является корректной.

Другой метод использует команду **git reflog**, которая выводит историю изменений ссылок в вашем репозитарии:

```
$ git reflog
f48bb36... HEAD@{0}: commit: Which commit parent?
7ecdc78... HEAD@{1}: dev: updating HEAD
e719b1f... HEAD@{2}: checkout: moving from dev to master
7ecdc78... HEAD@{3}: commit: Add dev_file to dev branch
e719b1f... HEAD@{4}: checkout: moving from master to dev
e719b1f... HEAD@{5}: checkout: moving from master to master
e719b1f... HEAD@{6}: HEAD^: updating HEAD
04289da... HEAD@{7}: commit: Updated foo.
e719b1f... HEAD@{8}: HEAD^: updating HEAD
72c001c... HEAD@{9}: commit: Add more foo.
e719b1f... HEAD@{10}: HEAD^: updating HEAD
0f61a54... HEAD@{11}: commit: Add more foo.
```

Просмотрите этот список. Третья строка свидетельствует о переключении с ветки **dev** на ветку **master**. В то время основной HEAD была фиксация с ID e719b1f. Если еще вам понадобится использовать e719b1f, вы можете или указать ее идентификатор или же использовать имя HEAD@{2}.

```
$ git rev-parse HEAD@{2}
e719b1fe81035c0bb5e1daaa6cd81c7350b73976

$ git reset --hard HEAD@{2}
HEAD is now at e719b1f Add master_file to master branch.

$ git show-branch
! [dev] Add dev_file to dev branch
 * [master] Add master_file to master branch.
--
+ [dev] Add dev_file to dev branch
+* [master] Add master_file to master branch.
```

Как только что было показано, reflog можно часто использоваться для нахождения предыдущей информации состояния для ссылок, таких как имена веток.

Аналогично, неправильно пытаться изменять ветки, используя команду git reset --hard:

```
$ git reset --hard dev
HEAD is now at 7ecdc78 Add dev_file to dev branch
$ ls
dev_file master_file
```

Кажется, что это сработало. В этом случае, рабочий каталог даже был заполнен правильными файлами из ветки dev. Но на самом деле это не работает! Ветка **master** все еще осталась текущей:

```
$ git branch
  dev
* master
```

Так же, как в предыдущем примере, фиксация в этой точке перепутала бы граф. Поэтому самое правильное действие в данном случае - определить корректное состояние и выполнить сброс к нему:

```
$ git reset --hard e719b1f
```

Или, возможно, даже:

```
$ git reset --soft e719b1f
```

При использовании опции --soft рабочий каталог не будет изменен, что означает, что ваш рабочий каталог теперь представляет содержимое (файлы и каталоги), присутствующие на вершине ветки dev. Кроме того, поскольку HEAD теперь правильно указывает на голову ветки master, фиксация в этой точке приведет к допустимому графу с новым основным состоянием, идентичным вершине ветки dev.

Может, это не совсем то, что вы хотели, но такой вариант тоже возможен.

10.3. Использование команды git cherry-pick

Команда `git cherry-pick фиксация` применяет изменения, представленные *фиксацией* на текущей ветке. Она представит новую, отдельную фиксацию. Строго говоря, использование команды **git cherry-pick** не изменяет существующую историю в репозитарии; вместо этого, она добавляет фиксацию в историю.

Как с другими операциями Git, которые представляют изменения через процесс применения разности, вы, возможно, должны будете разрешить конфликты, чтобы полностью применить изменения от данной фиксации.

Команда git cherry-pick обычно используется, чтобы представить определенные фиксации из одной ветки в репозитарии в другую ветку.

На рис. 10.4 у ветки **dev** есть нормальная разработка, тогда как rel_2.3 содержит фиксации для обслуживания выпуска 2.3.

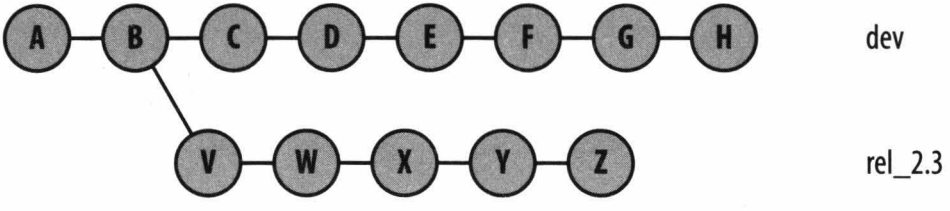


Рис. 10.4. До команды git cherry-pick

В течение нормальной разработки была исправлена ошибка фиксацией F. Если та же ошибка была также и в релизе 2.3, то с помощью git cherry-pick можно перенести фиксацию F в ветку rel_2.3:

```
$ git checkout rel_2.3
$ git cherry-pick dev~2 # фиксация F
```

На рис. 10.5 фиксация F' подобна фиксации F, но это полностью новая фиксация и, возможно, она должна быть скорректирована для разрешения кон-

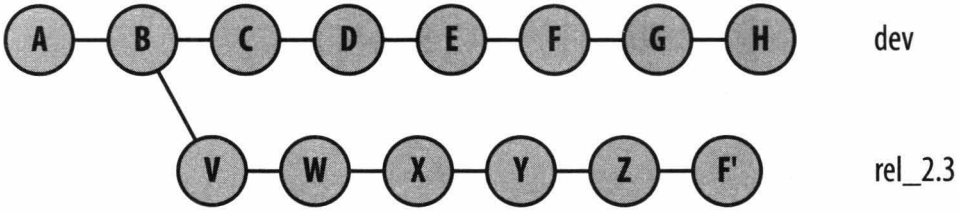


Рис. 10.5. После команды git cherry-pick

фликтов. Обратите внимание: фиксация F' применена после фиксации Z - последней фиксации ветки rel_2.3.

Другое применение **git cherry-pick** - восстановление серии фиксаций путем выбора пакета от одной ветки и перенос его в новую ветку.

Предположим, что у нас есть серия фиксаций в вашей ветке my_dev, как показано на рис. 10.6, и вы хотите перенести их в ветку master, но немного в другом порядке.

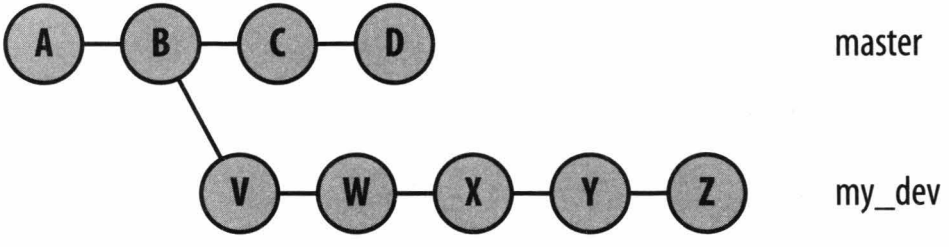


Рис. 10.6. Перед переносом фиксаций командой git cherry-pick

Чтобы применить их на ветке master в порядке Y, W, X, Z, используйте следующие команды:

```
$ git checkout master
$ git cherry-pick my_dev^       # Y
$ git cherry-pick my_dev~3      # W
$ git cherry-pick my_dev~2      # X
$ git cherry-pick my_dev # Z
```

После этого ваша история фиксаций будет похожа на рис. 10.7.

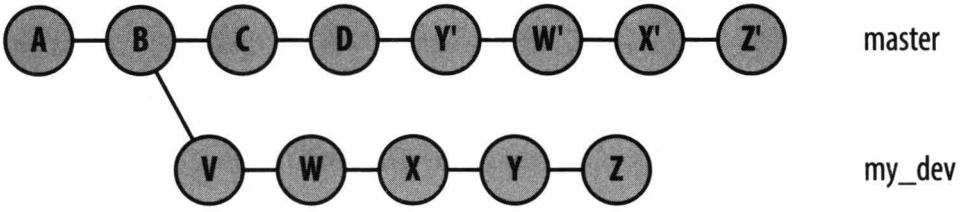


Рис. 10.7. После переноса фиксаций командой git cherry-pick

В подобных ситуациях, где порядок фиксаций претерпевает довольно большие изменения, скорее всего, вам придется разрешать конфликты. Наличие конфликтов полностью зависит от отношений между фиксациями. Если фиксации будут тесно связаны, у вас будут конфликты, которые придется разрешать. Если они будут не очень зависимым, конфликтов будет меньше.

Изначально, команда git cherry-pick могла выбирать только одну фиксацию за один раз. Однако, в более поздних версиях Git данная команда может выбирать диапазон фиксаций. Например, следующая команда:

```
# на ветке master
$ git cherry-pick X..Z
```

применит новые фиксации X', Y', Z' на ветке master.

10.4. Использование команды git revert

Команда `git revert фиксация` подобна команде `git cherry-pick фиксация` с одним важным отличием: она применяет инверсию заданной фиксации. Другими словами, данная команда отменяет фиксацию, используя хэш этой фиксации.

Подобно команде git cherry-pick, команда git revert не изменяет существующую историю в репозитарии. Вместо этого она добавляет в историю новую фиксацию.

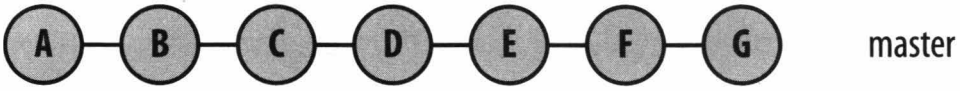


Рис. 10.8. Ветка до команды git revert

Обычное применение для команды git revert - это «отмена» эффекта заданной фиксации. На рис. 10.8 показана история ветки до применения команды git revert. По некоторым причинам, фиксацию D нужно отметить.

Для отмены фиксации D просто запустите команду **git revert**:

```
$ git revert master~3 # фиксация D
```

Результат показан на рис. 10.9, где фиксация D' - это отмена фиксации D.

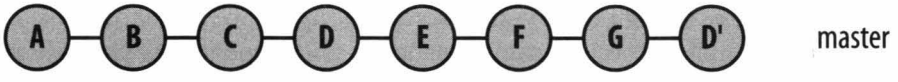


Рис. 10.9. Ветка после команды git revert

10.5. Команды reset, revert и checkout

Эти три команды, а именно reset, revert и checkout, могут быть немного непонятными, поскольку все они могут выполнять подобные операции.

Однако есть правила и рекомендации, когда каждая из этих команд должна использоваться, а когда - нет.

Если вы хотите переключиться на другую ветку, используйте команду **git checkout**. Ваша текущая ветка и ссылка HEAD будут изменены, чтобы соответствовать вершине заданной ветки.

Команда git reset не изменяет вашу ветку. Однако, если вы предоставите имя ветки, она изменит состояние вашего текущего рабочего каталога так,

что он будет выглядеть, как будто бы загружена указанная ветка. Другими словами, git reset предназначен для сброса ссылки HEAD текущей ветки.

Поскольку git reset --hard разработан для восстановления к известному состоянию, чего не делает команда git checkout, которая просто переключает ветку. Таким образом, если у вас есть незаконченная фиксация слияния, и вы попытались восстановиться при помощи команды git checkout вместо git reset --hard, ваша следующая фиксация будет ошибочной.

Путаница с git checkout происходит из-за его дополнительной возможности извлечения файла их хранилища объектов и его помещения в ваш рабочий каталог с возможной заменой версии, находящейся в вашем рабочем каталоге. Иногда версия того файла соответствует текущей версии HEAD, иногда - это более ранняя версия.

```
# Выгружаем файл file.c из индекса
$ git checkout -- path/to/file.c
# Выгружаем файл file.c из версии v2.3
$ git checkout v2.3 -- some/file.c
```

Git называет это «проверкой пути».

В первом случае при получении текущей версии из хранилища объектов, кажется, что это форма операции «сброса» («reset»), то есть изменения в вашем локальном рабочем каталоге будут потеряны, потому что файл будет сброшен к его текущей, HEAD-версии.

Во втором случае более ранняя версия файла извлекается из хранилища объектов и помещается в ваш рабочий каталог. Это похоже на команду «отмены» («revert»). Такая двойственность запутывает пользователей.

В обоих случаях неправильно думать о данной операции как о git reset или revert. В обоих случаях файл «извлекается» из определенной фиксации: HEAD или v2.3 соответственно. Команда git revert работает с фиксациями, а не с файлами!

Если другой разработчик клонировал ваш репозитарий или получил некоторые из ваших фиксаций, есть импликации для изменения истории фиксации. В этом случае вы не должны использовать команды, которые изменяют историю в вашем репозитарии. Лучше использовать команду git revert, не используйте команды git reset или git commit --amend (будет описана в следующем разделе).

10.6. Изменение последней фиксации

Самый простой способ изменить последнюю фиксацию вашей текущей ветки - это команда git commit --amend. Как правило, исправление (amend) подразумевает, что у фиксации есть существенно то же содержимое, но некоторый аспект требует корректировки или удаления. Фактический объект фиксации, который введен в хранилище объектов, будет, конечно, отличаться.

Типичное использование git commit --amend - исправление опечаток сразу после фиксации. Это не единственное применение этой команды, эта команда может исправить любой файл (файлы) и, действительно, может добавить или удалить файл как часть новой фиксации. Как и в случае с обычной командой git command, команда git command --amend запустит сеанс редактора, в котором вы можете изменить сообщение о фиксации.

Например, предположим, что вы работаете над речью и сделали следующую последнюю фиксацию:

```
$ git show
commit 0ba161a94e03ab1e2b27c2e65e4cbef476d04f5d
Author: Федя Колобков <kolobok@lobok.com>
Date: Thu Jun 26 15:14:03 2020 -0500

Initial speech

diff --git a/speech.txt b/speech.txt
new file mode 100644
index 0000000..310bcf9
--- /dev/null
+++ b/speech.txt
@@ -0,0 +1,5 @@
+Three score and seven years ago
+our fathers brought forth on this continent,
+a new nation, conceived in Liberty,
+and dedicated to the proposition
+that all men are created equal.
```

В данный момент фиксация сохранена в хранилище объектов Git, но в тексте есть небольшие ошибки. Чтобы исправить их, вам можно просто отредактировать файл снова и сделать вторую фиксацию. История будет примерно такой:

```
$ git show-branch --more=5
[master] Fix timeline typo
[master^] Initial speech
```

Однако, если вы хотите хранить чистую историю в вашем репозитарии, вы можете непосредственно отредактировать эту фиксацию и заменить ее.

Чтобы сделать это, исправьте файл в вашем рабочем каталоге. Исправьте опечатки и добавьте/удалите файл, если необходимо. Как и с обычной фиксацией для обновления индекса используйте команды **git add** и **git rm**. После чего выполните команду git commit --amend:

```
# исправьте опечатки в speech.txt
$ git diff
diff --git a/speech.txt b/speech.txt
index 310bcf9..7328a76 100644
--- a/speech.txt
+++ b/speech.txt
@@ -1,5 +1,5 @@
-Three score and seven years ago
+Four score and seven years ago
our fathers brought forth on this continent,
a new nation, conceived in Liberty,
and dedicated to the proposition
-that all men are created equal.
+that all men and women are created equal.

$ git add speech.txt
$ git commit --amend
# При необходимости измените сообщение фиксации
# («Initial speech»)
# Мы его немного изменили
```

При использовании --amend любой сможет увидеть, что исходная фиксация была изменена:

```
$ git show-branch --more=5
```

```
[master] Initial speech that sounds familiar.

$ git show
commit 47d849c61919f05da1acf983746f205d2cdb0055
Author: Федя Колобков <kolobok@lobok.com>
Date: Thu Jun 26 15:14:03 2020 -0500

Initial speech that sounds familiar.

diff --git a/speech.txt b/speech.txt
new file mode 100644
index 0000000..7328a76
--- /dev/null
+++ b/speech.txt
@@ -0,0 +1,5 @@
+Four score and seven years ago
+our fathers brought forth on this continent,
+a new nation, conceived in Liberty,
+and dedicated to the proposition
+that all men and women are created equal.
```

Эта команда может редактировать метаинформацию фиксации. Например, указав опции --author, вы можете изменить автора фиксации:

```
$ git commit --amend --author "Bob Miller <kbob@example.com>"
# ...just close the editor...

$ git log
commit 0e2a14f933a3aaff9edd848a862e783d986f149f
Author: Bob Miller <kbob@example.com>
Date: Thu Jun 26 15:14:03 2020 -0500

Initial speech that sounds familiar.
```

Посмотрим на граф фиксаций, на рис. 10.10 изображен граф до выполнения команды git commit --amend, а на 10.11 - после.

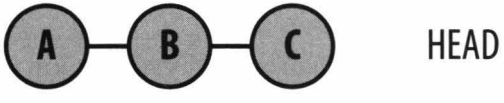


Рис. 10.10. Граф фиксации до выполнения команды git commit --amend

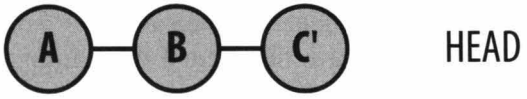


Рис. 10.11. Граф фиксации после выполнения команды git commit --amend

Здесь экземпляр фиксации C остался практически тем же самым, но был изменен, чтобы получить фиксацию C'. Ссылка HEAD была изменена и теперь она указывает не на фиксацию C, а на фиксацию C'.

10.7. Перебазирование фиксаций

Команда **git rebase** используется для изменения последовательности фиксаций. Эта команда требует, как минимум, одного имени другой ветки, в которую будут перемещены ваши фиксации. По умолчанию перебазируются фиксации из текущей ветки, которых еще нет в целевой ветке.

Типичное использование команды git rebase - сохранить серию фиксаций из указанной ветки на последнюю фиксацию ветки master.

На рис. 10.12 показано, что было разработано две ветки. Изначально ветка topic произошла от ветки master (с фиксации B). Тем временем основная ветка master тоже не стояла на месте и дошла до фиксации E.

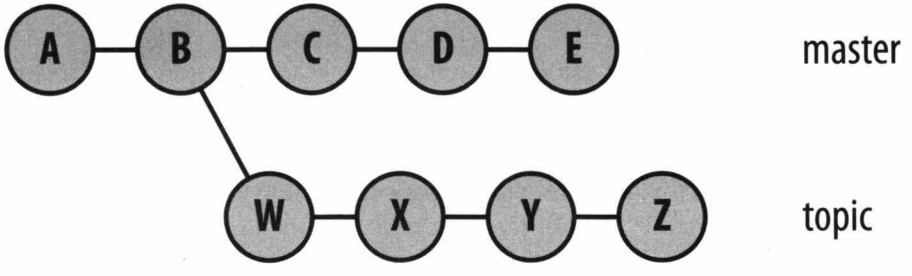


Рис. 10.12. Перед выполнением команды git rebase

Предположим, что вы хотите перенести фиксации из ветки topic в ветку master. Они будут присоединены к последней фиксации ветки master, то есть к фиксации E, а не к B. Поскольку ветка topic должна быть текущей, сначала на нее нужно переключиться:

```
$ git checkout topic
$ git rebase master
```

Или же использовать эту команду:

```
$ git rebase master topic
```

После завершения операции перебазирования новый граф будет таким, как показано на рис. 10.13.

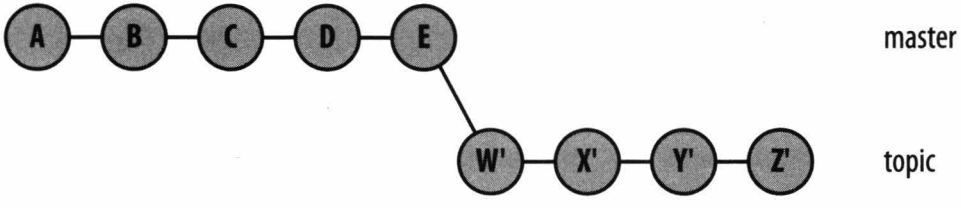


Рис. 10.13. После команды git rebase

Использование git rebase в ситуациях, подобной показанной на рис. 10.12 часто называется *портированием*. В этом примере, ветка topic была *портирована* в ветку master.

Команда git rebase может также использоваться для полной трансплантации линии разработки из одной ветки в другую, для этого используется опция --onto.

Например, предположим, что вы разработали новую функцию в ветке feature с фиксациями P и Q. Ветка feature основана на ветке maint, что показано на рис. 10.14. Чтобы трансплантировать фиксации P и Q из ветки feature в ветку master (не maint!), используйте команду:

```
$ git rebase --onto master maint^ feature
```

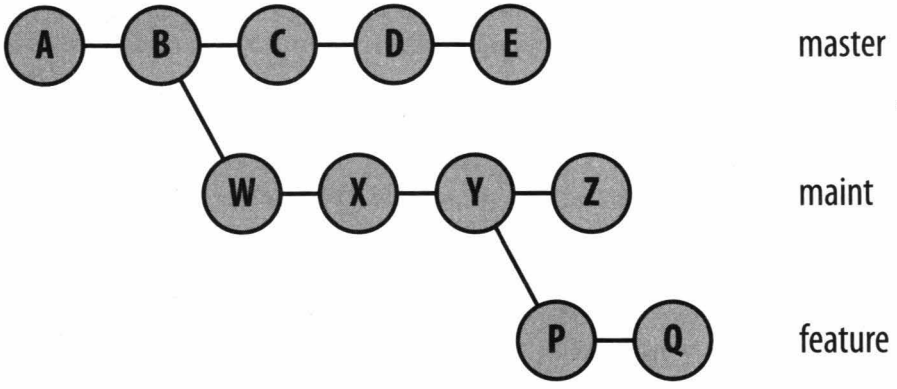


Рис. 10.14. Перед трансплантацией командой git rebase

Результат приведен на рис. 10.15.

При трансплантации могут произойти конфликты, которые придется разрешить вручную. Имейте это в виду при использовании команды git rebase.

Если конфликт будет найден, операция rebase временно прекратит обработку - пока вы не разрешите конфликт. О том, как разрешать конфликты, было сказано в главе 9.

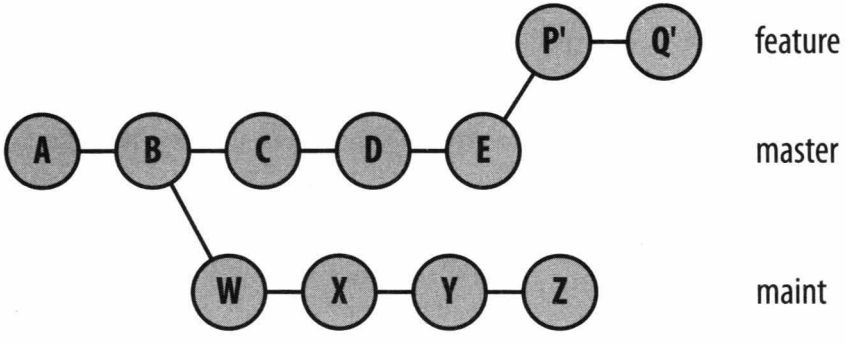


Рис. 10.15. После трансплантации командой git rebase

Когда вы разрешите все конфликты и обновите индекс, операцию rebase можно продолжить командой git rebase --continue. Данная команда продолжит выполнение перебазирования фиксаций до следующего конфликта, если таковой будет найден.

Если во время исследования конфликта вы решите, что определенная фиксация не является необходимой, используйте команду git rebase --skip для перехода к следующей фиксации. Это не очень корректно и может привести к еще большим конфликтам, поэтому лучше все-таки попытаться разрешить конфликт.

Использование команды git rebase -i

Предположим, что вы начали писать хайку и уже написали две полные строки:

```
$ git init
Initialized empty Git repository in .git/
$ git config user.email "kolobok@lobok.com"

$ cat haiku
Talk about colour
No jealous behaviour here

$ git add haiku
$ git commit -m "Start my haiku"
Created initial commit a75f74e: Start my haiku
1 files changed, 2 insertions(+), 0 deletions(-)
create mode 100644 haiku
```

Но потом вы решили использовать *Американский вариант* написания слова «color» вместо *Британского*. Вы сделали фиксацию, чтобы исправить это:

```
$ git diff
diff --git a/haiku b/haiku
index 088bea0..958aff0 100644
--- a/haiku
+++ b/haiku
@@ -1,2 +1,2 @@
-Talk about colour
+Talk about color
No jealous behaviour here

$ git commit -a -m "Use color instead of colour"
Created commit 3d0f83b: Use color instead of colour
1 files changed, 1 insertions(+), 1 deletions(-)
```

Затем вы придумали последнюю строчку и зафиксировали ее:

```
$ git diff
diff --git a/haiku b/haiku
index 958aff0..cdeddf9 100644
--- a/haiku
+++ b/haiku
@@ -1,2 +1,3 @@
Talk about color
No jealous behaviour here
+I favour red wine

$ git commit -a -m "Finish my colour haiku"
Created commit 799dba3: Finish my colour haiku
1 files changed, 1 insertions(+), 0 deletions(-)
```

Однако, вы снова написали *Британский вариант* вместо *Американского* и хотите исправить его:

```
$ git diff
diff --git a/haiku b/haiku
index cdeddf9..064c1b5 100644
--- a/haiku
+++ b/haiku
@@ -1,3 +1,3 @@
Talk about color
-No jealous behaviour here
-I favour red wine
+No jealous behavior here
+I favor red wine

$ git commit -a -m "Use American spellings"
Created commit b61b041: Use American spellings
1 files changed, 2 insertions(+), 2 deletions(-)
```

В этой точке ваша история фиксаций будет следующей:

```
$ git show-branch --more=4
[master] Use American spellings
[master^] Finish my colour haiku
[master~2] Use color instead of colour
[master~3] Start my haiku
```

После просмотра последовательности фиксаций вы решаете изменить ее так:

```
[master] Use American spellings
[master^] Use color instead of colour
[master~2] Finish my colour haiku
[master~3] Start my haiku
```

Затем в вашу голову приходит мысль объединить две похожие фиксации - те, которые исправляют написание разных слов. Поэтому вы объединяете фиксации master и master^ в одну:

```
[master] Use American spellings
[master^] Finish my colour haiku
[master~2] Start my haiku
```

Изменение порядка, редактирование, перемещение и слияние нескольких фиксаций в одну - все это можно сделать с помощью команды git rebase -i, где опция -i (или --interactive) означает «интерактивность».

Эта команда позволяет вам изменить фиксации и поместить их или в исходную, или в другую ветку. Типичное использование - изменение фиксаций в той же ветке. При запуске команды нужно указать имя фиксации, с которой вы хотите начать вносить изменения:

```
$ git rebase -i master~3
```

Затем будет открыт редактор, файл в котором будет выглядеть примерно так:

```
pick 3d0f83b Use color instead of colour
pick 799dba3 Finish my colour haiku
pick b61b041 Use American spellings
# Rebase a75f74e..b61b041 onto a75f74e
#
# Commands:
# pick = use commit
# edit = use commit, but stop for amending
# squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
```

```
# However, if you remove everything, the rebase will be
aborted.
#
```

Первые три строки перечисляют фиксации в доступном для редактирования диапазоне фиксаций, который вы указали в командной строке. Фиксации приводятся в порядке от самой старой до самой новой, а перед каждой из них есть команда pick. Если вы сейчас выйдете из редактора, каждая из этих фиксаций будет выбрана (в указанном порядке) и применена к целевой ветке. Строки, отмеченные знаком #, являются комментариями и игнорируются программой.

Сейчас вы можете переупорядочить фиксации, объединить их вместе, изменить фиксации или удалить навсегда. Представим, что вы хотите просто переупорядочить фиксации. Для этого нужно просто переупорядочить строки в вашем редакторе, а потом выйти из него:

```
pick 799dba3 Finish my colour haiku
pick 3d0f83b Use color instead of colour
pick b61b041 Use American spellings
```

Вспомните, что первая фиксация для перебазировки - это фиксация «Start my haiku». Следующей фиксацией станет «Finish my colour haiku», после которой последуют фиксации «User color...» и «Use American...».

```
$ git rebase -i master~3
# переупорядочим первые две фиксации и выйдем из редактора
Successfully rebased and updated refs/heads/master.

$ git show-branch --more=4
[master] Use American spellings
[master^] Use color instead of colour
[master~2] Finish my colour haiku
[master~3] Start my haiku
```

Итак, мы переупорядочили историю. Следующий шаг - объединить (операция squash) две фиксацию в одну. Снова выполните команду git rebase -i master~3. На этот раз конвертируйте список фиксаций из:

```
pick d83f7ed Finish my colour haiku
pick 1f7342b Use color instead of colour
```

```
pick 1915dae Use American spellings
```

в:

```
pick d83f7ed Finish my colour haiku
pick 1f7342b Use color instead of colour
squash 1915dae Use American spellings
```

Третья фиксация будет помещена непосредственно в предшествующую фиксацию, а новый шаблон сообщения журнала фиксации будет сформирован из комбинации фиксаций, объединенных вместе.

В этом примере два сообщения журнала фиксации будут объединены вместе и представлены как одно в редакторе:

```
# This is a combination of two commits.
# The first commits message is:
Use color instead of colour
# This is the 2nd commit message:
Use American spellings
```

Эти сообщения будут отредактированы так:

```
Use American spellings
```

Снова, все строки, отмеченные # игнорируются.

Теперь посмотрим результаты перебазирования последовательности:

```
$ git rebase -i master~3
# squash and rewrite the commit log message
Created commit cf27784: Use American spellings
1 files changed, 3 insertions(+), 3 deletions(-)
Successfully rebased and updated refs/heads/master.

$ git show-branch --more=4
[master] Use American spellings
[master^] Finish my colour haiku
[master~2] Start my haiku
```

Хотя мы переупорядочили и объединили фиксации за два шага, эти операции можно было бы выполнить за одно действие.

Сравнение rebase и merge

В дополнение к обычному изменению истории, у операции перебазирования есть особенности, о которых вы должны знать.

Перебазирование последовательности фиксацию на вершину ветку подобно объединению двух веток.

Вы можете задать себе вопрос: что лучше использовать - **rebase** или **merge**? В главе 12 этот вопрос станет особенно остро, особенно, когда в игру вступают множество разработчиков, репозитариев и веток.

В процессе перебазирования последовательности фиксаций Git сгенерирует полностью новые последовательности фиксаций. У них будут новые SHA1-идентификаторы фиксации, основанные на новом начальном состоянии и представляющие другие разницы, даже при том, что они включают изменения, которые достигают того же окончательного состояния.

Когда сталкиваешься с ситуацией, изображенной на рис. 10.12, перебазировать ее в ситуацию, изображенную на рис. 10.13, не составляет проблемы, поскольку никакая друга фиксация не основана на перебазируемой ветке. Однако, даже в вашем собственном репозитарии у вас могут быть дополнительные ветки, основанные на той, которую вы хотите перебазировать. Рассмотрим граф, изображенный на рис. 10.16.

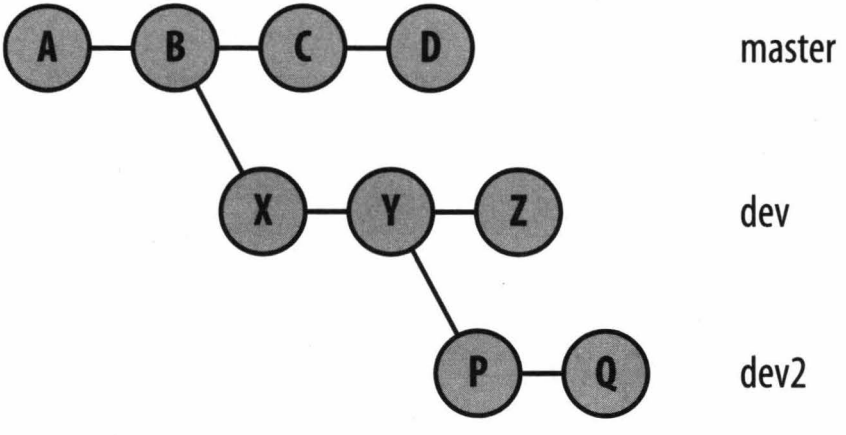


Рис. 10.16. Перед многоветковым перебазированием

Вы можете подумать, что выполнение команды:

```
$ git rebase master dev
```

Приведет к графу, изображенному на рис. 10.17. Но это не так. Первая подсказка, что этого не произошло, следует из вывода команды.

```
$ git rebase master dev
First, rewinding head to replay your work on top of it...
Applying: X
Applying: Y
Applying: Z
```

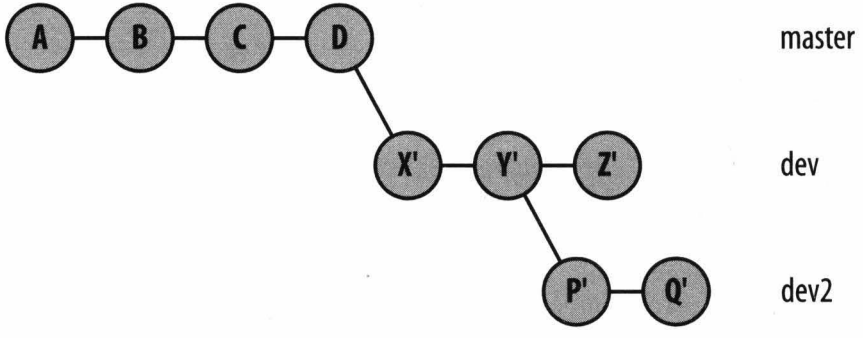


Рис. 10.17. Требуемый граф

Это говорит о том, что Git применит только фиксации X, Y и Z. Но ничего не сказано о фиксациях P и Q, вместо этого вы получите граф, изображенный на рис. 10.18.

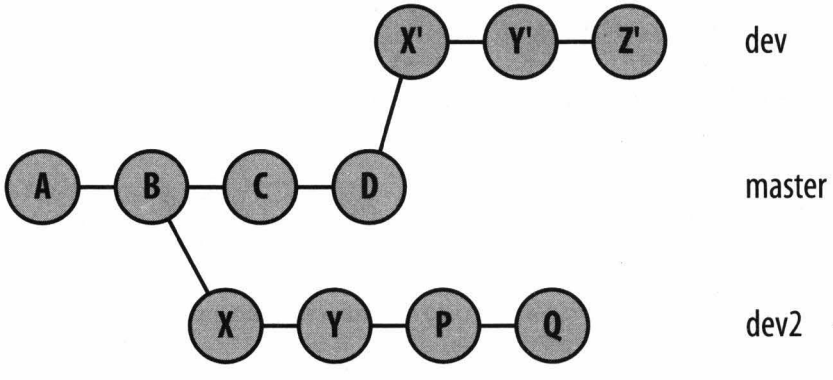


Рис. 10.18. То, что получилось

Фиксации X', Y' и Z' - новые версии старых фиксаций, произошедших от B.

Старые фиксации X и Y все еще существуют в графе, поскольку они все еще достижимы от ветки dev2. Однако, исходная фиксация Z была удалена, поскольку она больше не достижима. Имя ветки, которая указывала на нее, было перемещено в новую версию той фиксации.

Теперь, похоже, что в истории ветки есть дублирующиеся сообщения:

```
$ git show-branch
* [dev] Z
 ! [dev2] Q
  ! [master] D
---
*   [dev] Z
*   [dev^] Y
*   [dev~2] X
* + [master] D
* + [master^] C
+   [dev2] Q
+   [dev2^] P
+   [dev2~2] Y
+   [dev2~3] X
*++ [master~2] B
```

Но помните, это различные фиксации, которые делают по существу то же изменение. Если вы объедините ветку с одной из новых фиксаций в другую ветку, у которой есть одна из старых фиксаций, Git никак не узнает, что вы применяете одно и то же изменение дважды. В результате появятся двойные записи в git log, вероятнее всего, будет конфликт слияния и общий хаос. Вы должны придумать, как избавиться от этой ситуации.

Если получившийся граф то, что вы хотите сделать, то вы это сделали. Но, скорее всего, это не то, что вы хотите. Чтобы достигнуть графа, изображенного на рис. 10.17, вам нужно перебазировать ветку dev2 на новую фиксацию Y ветки dev:

```
$ git rebase dev^ dev2
First, rewinding head to replay your work on top of it...
Applying: P
Applying: Q

$ git show-branch
! [dev] Z
```

```
 * [dev2] Q
  ! [master] D
---
 *   [dev2] Q
 *   [dev2^] P
+    [dev] Z
+*   [dev2~2] Y
+*   [dev2~3] X
+*+ [master] D
```

Наконец-то мы достигли графа, показанного на рис. 10.17.

Другая запутывающая ситуация - это перебазирование ветки, у которой есть слияние на ней. Например, предположим, что у вас есть структура веток, подобная изображенной на рис. 10.19.

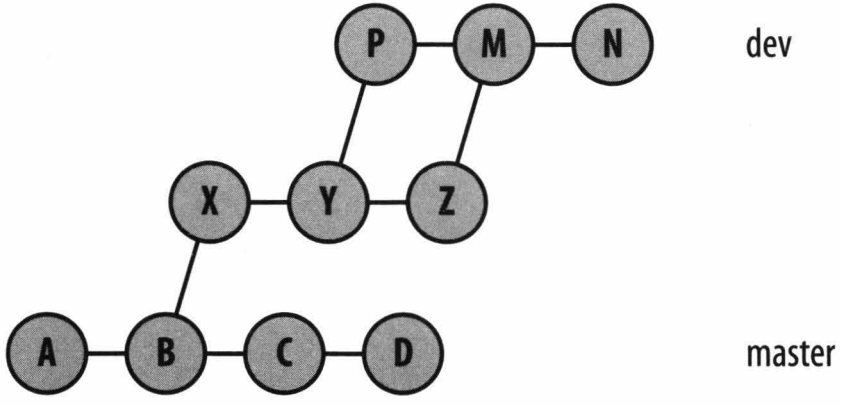


Рис. 10.19. Перед слиянием средствами git rebase

Если вы хотите переместить всю структуру ветки dev из фиксации N вниз до фиксации X и присоединить к фиксации D (не B), чтобы получить структуру, изображенную на рис. 10.20, вам нужно использовать команду git rebase master dev.

Снова, однако, эта команда приводит к некоторым удивительным результатам:

```
$ git rebase master dev
First, rewinding head to replay your work on top of it...
Applying: X
Applying: Y
```

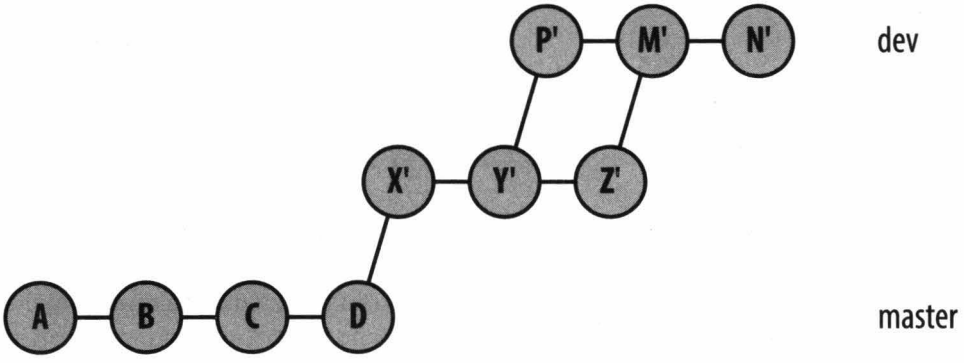


Рис. 10.20. Ожидаемая структура дерева фиксаций

```
Applying: Z
Applying: P
Applying: N
```

Кажется, все правильно. Но нужно разобраться, так это или нет.

```
$ git show-branch
* [dev] N
 ! [master] D
--
* [dev] N
* [dev^] P
* [dev~2] Z
* [dev~3] Y
* [dev~4] X
*+ [master] D
```

Все фиксации теперь выстроились в одну длинную строку. Что же произошло здесь?

Git должен переместить часть графа, достижимого от dev к базе слияния B, поэтому он найдет фиксации в диапазоне master..dev. Чтобы вывести все те фиксации, Git осуществляет топологическую сортировку той части графа для произведения линеаризированной последовательности всех фиксаций в диапазоне. Как только та последовательность была определена, Git применяет фиксации по одной, начиная с фиксации D, как показано на рис. 10.21.

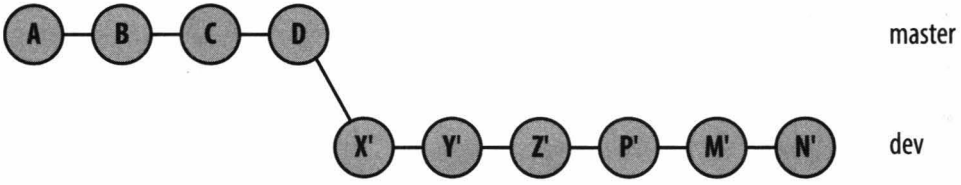


Рис. 10.21. Дерево фиксаций после линеризации

Снова, если вы хотели именно этого или вас не заботит форма графа, тогда все готово. Но обычно в таких случаях хочется получить именно требуемую структуру графа, поэтому нужно использовать опцию --preserve-merges.

```
# Эта опция появилась в версии 1.6.1
$ git rebase --preserve-merges master dev
Successfully rebased and updated refs/heads/dev.
```

Просмотреть структуру результирующего графа можно командой:

```
$ git show-graph
* 061f9fd... N
* f669404... Merge branch 'dev2' into dev
|\
| * c386cfc... Z
* | 38ab25e... P
|/
* b93ad42... Y
* 65be7f1... X
* e3b9e22... D
* f2b96c4... C
* 8619681... B
* d6fba18... A
```

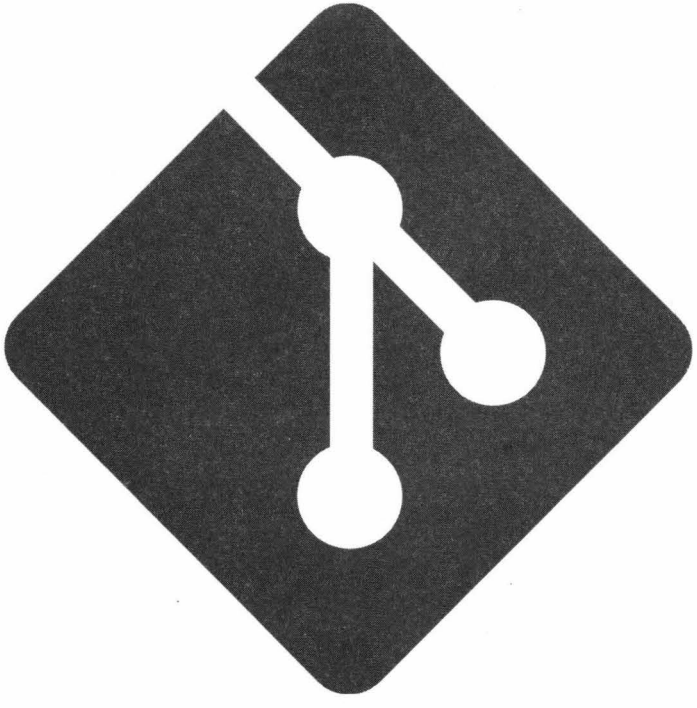
Это уже похоже на структуру, изображенную на рис. 10.20.

Ответ на вопрос, что лучше использовать - перебазирование или слияние, зависит от того, какой репозитарий вы используете - ваш собственный или распределенный. Также выбор стратегии зависит от вашего стиля разработки и вашего окончательного намерения.

Если операция перебазирования - это не ваш выбор, но вам все еще нужно получить изменения ветки, тогда вам поможет слияние.

Важно помнить следующее:

- Перебазирование перезаписывает фиксации как новые фиксации.
- Старые фиксации, которые больше не будут доступны, будут потеряны.
- Если у вас есть ветка, которая содержит пред-перебазированную фиксацию, вам нужно перебазировать ее снова.
- Если есть пользователь пред-перебазированной фиксации в другом репозитарии, у него все еще есть копия той фиксации, даже если она была перемещена в ваш репозитарий. Пользователю нужно согласовать свою историю фиксаций.



Глава 11.

Stash и reflog



11.1. Stash

Вы когда-нибудь чувствовали себя уставшим в процессе разработки, когда накапливаются постоянные прерывания, требования об исправлениях ошибок и запросы от коллег, создающие помехи реальной работы? Тогда механизм stash создан именно для вас!

Stash - это механизм, позволяющий «захватить» ваш рабочий процесс, позволяя вам сохранить ее и вернуться к ней позже, когда вам будет удобно. По сути, у вас есть механизм фиксации, но stash - это более продвинутый механизм, позволяющий захватить состояние вашего индекса и рабочего каталога одной простой командой. Он оставляет чистым и непротиворечивым ваш репозитарий, а также готовым к альтернативному направлению разработки. В результате вы можете возобновить работу именно с того места, где закончили.

Давайте посмотрим, как stash работает с каноническим вариантом использования: с так называемым «прерываемым потоком операций».

В этом сценарии вы счастливо работаете в своем репозитарии Git, за время работы вы изменили несколько файлов и даже подготовили некоторые

файлы в индексе. Затем происходит прерывание. Возможно, обнаружена критическая ошибка и вы должны сразу же ее исправить. Возможно, руководитель вашей группы внезапно приоритезировал новую функцию и настаивает, чтобы вы отбросили все и работали именно над ней. Хочется или нет, но вы должны убрать все и начать заново. И это потрясающее применение для команды **git stash**!

```
$ cd the-git-project
# обычный процесс работы
# прерывание!
# нужно сохранить все и заняться чем-то другим
$ git stash save

# занимаемся чем-то более приоритетным
$ git commit -a -m «Исправление высоко-приоритетной проблемы»
$ git stash pop
```

Мы только что сохранили рабочий процесс, а затем продолжили работу с момента сохранения.

Операция по умолчанию для git stash - save. При желании в качестве параметра этой операции вы можете добавить описание сохраняемой работы, но конечно же, лучше самому помнить, что вы делали:

```
$ git stash save «Работаю над текущим проектом»
```

Чтобы проделать то же самое, но с помощью базовых команд Git, вам нужно вручную создать новую ветку, на которой вы фиксируете все ваши изменения и выполняете много других операций по сохранению и восстановления текущего процесса работы. Давайте рассмотрим небольшой пример:

```
# ... прерван нормальный процесс разработки ...
# Создаем новую ветку, в которой мы сохраним текущее состояние
$ git checkout -b saved_state
$ git commit -a -m "Сохраненное состояние"
# Обратно к предыдущей ветке для немедленного исправления
ошибки
```

```
$ git checkout master
# исправляем срочную ошибку
$ git commit -a -m «Исправляем что-то.»
# Восстанавливаем сохраненное состояние на вершине рабочего
каталога
$ git checkout saved_state
$ git reset --soft HEAD^
# ... продолжаем работу ...
```

Как видите, все можно реализовать базовыми командами Git, но этот процесс чувствительным к деталям. При сохранении своего состояния вы должны получить все свои изменения, а сам процесс восстановления может быть разрушен, если вы забудете переместить обратно HEAD.

Команда **git stash save** сохраняет ваш текущий индекс, рабочий каталог и «очищают» их так, чтобы они соответствовали голове вашей текущей ветки. Хотя, кажется, что вы потеряете все изменения, внесенные в файлы, это не так. Содержимое вашего индекса и рабочего каталога независимо сохраняется и доступно через refs/stash.

```
$ git show-branch stash
[stash] WIP on master: 3889def Some initial files.
```

Примечание. WIP - это сокращение от Work In Progress

Как вы можете предположить, команда **pop** восстанавливает ваше состояние. Две базовых команды stash, git stash save и git stash pop реализуют стек состояний stash. Это позволяет вашему потоку операций прерываться снова и снова!

Команда git stash pop восстанавливает контекст, сохраненный предыдущей операцией save на вершину вашего текущего рабочего каталога и индекса. При восстановлении содержимое стека объединяется с текущим состоянием, а не просто перезаписывается с заменой файлов. Хорошо, да?

Команду **git stash pop** можно выполнить только в чистый рабочий каталог. Даже тогда команда может не полностью преуспеть в воссоздании предше-

ствующего состояния. Поскольку восстановление может быть выполнено поверх какой-то фиксации, может потребоваться слияние, которое, в свою очередь, может завершиться с конфликтом, которые нужно будет разрешить вручную.

После успешной операции pop, Git автоматически удалит ваше сохраненное состояние из стека сохраненных состояний. То есть после применение состояние сразу «отбрасывается». Однако, когда необходимо разрешение конфликтов, Git не будет отбрасывать состояние - на всякий случай. Как только конфликты будут разрешены, вам нужно вручную отбросить предыдущее сохраненное состояние командой git stash drop, чтобы удалить его из стека состояний.

Если вы хотите только воссоздать контекст, сохраненный в стеке состояний без его удаления из стека, используйте команду git stash apply. Команда **pop** - это успешная команда apply, после которой следует команда drop.

Какую команду, **apply** или **pop**, использовать, думаю понятно. Если вам еще понадобится это сохраненное состояние, используйте apply, если вы хотите восстановиться без сохранения состояния, используйте pop.

Команда **git stash list** выводит стек сохраненных контекстов от самого последнего до самого старого:

```
$ cd my-repo
$ ls
file1 file2

$ echo «some foo» >> file1

$ git status

# On branch master
# Changes not staged for commit:
# (use «git add <file>...» to update what will be committed)
# (use «git checkout -- <file>...» to discard changes in
working directory)
#
# modified: file1
#
```

```
no changes added to commit (use «git add» and/or «git commit
-a»)

$ git stash save "Исправление file1"
Saved working directory and index state On master: Исправление
file1
HEAD is now at 3889def Add some files

$ git commit --dry-run
# On branch master
nothing to commit (working directory clean)

$ echo «some bar» >> file2

$ git stash save "Редактирование file2"
Saved working directory and index state On master:
Редактирование file2
HEAD is now at 3889def Add some files

$ git stash list
stash@{0}: On master: Редактирование file2
stash@{1}: On master: Исправление file1
```

Git всегда присваивает самому последнему состоянию число 0. Поскольку записи становятся старше, увеличиваются и их номера. Имена различных записей stash - stash@{0}, stash@{1} - о них мы еще поговорим в разделе «Reflog».

Команда **git stash show** показывает изменения индекса и файлов, записанные для заданной записи stash, относительно к ее родительской фиксации:

```
$ git stash show
file2 | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

Эта сводка может предоставить вам ту информацию, которую вы искали. Если же она показалась вам малоинформативной, добавьте опцию -p, чтобы увидеть другую полезную информацию. Обратите внимание на то, что по

умолчанию команда git stash show показывает наиболее последнюю запись, stash@{0}.

Поскольку изменения, которые используются для создания stash-состояния, относительны определенной фиксации, показ состояния - это сравнение состояние-состояние, что больше похоже на вывод git diff, а не на последовательность фиксаций (git log). Таким образом, все опции, подходящие для git diff, можно указывать и для git stash show.

А сейчас давайте посмотрим на использование опции **-p** для получения различий в патче для данного stash-состояния:

```
$ git stash show -p stash@{1}
diff --git a/file1 b/file1
index 257cc56..f9e62e5 100644
--- a/file1
+++ b/file1

@@ -1 +1,2 @@
foo
+some foo
```

Рассмотрим другой классический случай использования **git stash**. Пока вы не знакомы с использованием удаленных репозиториев и получением изменений по запросу, для вас данная информация мало целесообразно. Но очень скоро она для вас будет очень актуальной. Допустим, вы ведете разработку в своем локальном репозитарии и сделали несколько фиксаций. У вас есть некоторые измененные файлы, которые вы еще не зафиксировали, но вы понимаете, что есть восходящие изменения, которые вам нужны. Если у вас будут конфликтные изменения, обычная команда **git pull** провалится, отказываясь перезаписать ваши локальные изменения. Один из способов решить эту проблему - использовать git stash:

```
$ git pull
# ... сбой pull из-за конфликтов слияния ...
$ git stash save
$ git pull
$ git stash pop
```

В данной точке, возможно, вам придется разрешить конфликты, созданные командой pop. В случае если у вас есть новые, незафиксированные (и значит «неотслеживаемые») файлы как часть вашей локальной разработки, возможно, что git pull также представит файл с таким же именем, что приведет к сбою, поскольку команда не захочет перезаписать вашу версию новым файлом. В этом случае добавьте опцию --include-untracked команде git stash, чтобы она также спрятала ваши новые, неотслеживаемые файлы вместе с остальной частью ваших изменений. Это гарантирует полностью чистый рабочий каталог для pull.

Опция -all получает неотслеживаемые файлы, а также явно игнорируемые файлы, указанные в .gitignore.

Наконец, для более сложных stash-операций, где вы хотите выборочно выбрать, что должно быть помещено в stash-стек, используйте опцию **-p** или **--patch**.

В другом подобном сценарии git stash может использоваться, когда вы хотите переместить измененную работу в удаленный репозитарий, когда невозможно выполнить pull --rebase. Обычно это может произойти до помещения ваших локальных фиксаций в удаленный репозитарий:

```
# ... редактируем и фиксируем ...
# ... продолжаем редактирование ...
$ git commit --dry-run
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
# Changed but not updated:
# (use «git add <file>...» to update what will be committed)
# (use «git checkout -- <file>...» to discard changes in
working directory)
#
# modified: file1.h
# modified: file1.c

#
```

```
no changes added to commit (use «git add» and/or «git commit
-a»)
```

В этой точке вы можете решить выполнить команду pull --rebase, но Git откажется ее выполнить, поскольку в вашем рабочем каталоге есть измененные, но незафиксированные файлы:

```
$ git pull --rebase
file1.h: needs update
file1.c: needs update
refusing to pull with rebase: your working tree is not up-to-
date
```

Этот сценарий не столь надуман, как может показаться на первый взгляд. Например, я часто вношу изменения в Makefile, например, для включения/выключения отладки или для изменения других параметров конфигурации. Я не хочу фиксировать эти изменения, но и не хочу терять их между обновлениями из удаленного репозитария. Я просто хочу, чтобы они задержались здесь, в моем рабочем каталоге.

Снова, здесь может помочь команда git stash:

```
$ git stash save
Saved working directory and index state WIP on master: 5955d14
Some commit log.
HEAD is now at 5955d14 Some commit log.

$ git pull --rebase
remote: Counting objects: 63, done.
remote: Compressing objects: 100% (43/43), done.
remote: Total 43 (delta 36), reused 0 (delta 0)
Unpacking objects: 100% (43/43), done.
From ssh://git/var/git/my_repo
871746b..6687d58 master -> origin/master
First, rewinding head to replay your work on top of it...
Applying: A fix for a bug.
Applying: The fix for something else.
```

После того, как вы получите фиксации из удаленного репозитария и перебазируете ваши локальные фиксации на их вершину, ваш репозитарий подходит для отправки вашей работы в удаленный репозитарий. Как решите, вы можете отправить свою работу в удаленный репозитарий командой **git push**:

```
# Отправляем вашу работу в удаленный репозитарий!
$ git push
```

Или после восстановления вашего предыдущего состояния рабочего каталога:

```
$ git stash pop
Auto-merging file1.h
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
# Changed but not updated:
# (use «git add <file>...» to update what will be committed)
# (use «git checkout -- <file>...» to discard changes in
working directory)
#
# modified: file1.h
# modified: file1.c
#
no changes added to commit (use "git add" and/or "git commit
-a")
Dropped refs/stash@{0} (7e2546f5808a95a2e6934fcffb5548651badf0
0d)
```

```
$ git push
```

Если вы решите использовать git push после извлечения вашего stash-состояния, помните, что только завершенная, зафиксированная работа будет отправлена в удаленный репозитарий. Не нужно волноваться об отправке вашей частичной, незавершенной работы.

Иногда восстановление предыдущего состояния может повлечь за собой трудноразрешимые конфликты слияния. Однако, вы все еще можете восстановить сохраненное состояние в новую ветку. Для этого используется команда **git stash branch**. Данная команда преобразовывает содержимое сохраненного состояния в новую ветку на основе фиксация, которая была текущей на момент создания stash-записи.

Давайте посмотрим, как это работает на практике:

```
$ git log --pretty=one --abbrev-commit
d5ef6c9 Some commit.
efe990c Initial commit.
```

Теперь сохраняем ваше текущее состояние:

```
$ git stash
Saved working directory and index state WIP on master: d5ef6c9
Some commit.
HEAD is now at d5ef6c9 Some commit.
```

Обратите внимание на ID фиксации d5ef6c9.

Далее вы сделали несколько фиксаций, и ветка ушла из состояния d5ef6c9.

```
$ git log --pretty=one --abbrev-commit
2c2af13 Another mod
1d1e905 Drifting file state.
d5ef6c9 Some commit.
efe990c Initial commit.

$ git show-branch -a
[master] Another mod
```

И хотя сохраненная работа доступна, она не может быть чисто применена к текущей ветке master:

```
$ git stash list
```

```
stash@{0}: WIP on master: d5ef6c9 Some commit.
```

```
$ git stash pop
Auto-merging foo
CONFLICT (content): Merge conflict in foo
Auto-merging bar
CONFLICT (content): Merge conflict in bar
```

Давайте создадим новую ветку с названием **mod**, в которую и выгрузим сохраненное ранее состояние. Только перед этим нужно сбросить текущее состояние:

```
$ git reset --hard master
HEAD is now at 2c2af13 Another mod
```

```
$ git stash branch mod
Switched to a new branch 'mod'
# On branch mod
# Changes not staged for commit:
# (use «git add <file>...» to update what will be committed)
# (use «git checkout -- <file>...» to discard changes in
working directory)
#
# modified: bar
# modified: foo
#
no changes added to commit (use «git add» and/or «git commit
-a»)
Dropped refs/stash@{0} (96e53da61f7e5031ef04d68bf60a34bd4f13bd
9f)
```

Есть несколько важных моментов, на которые следует обратить ваше внимание. Во-первых, заметьте, что созданная ветка основана на исходной фиксации d5ef6c9, а не на текущей фиксации 2c2af13.

```
$ git show-branch -a
! [master] Another mod
```

```
 * [mod] Some commit.
--
+  [master] Another mod
+  [master^] Drifting file state.
+* [mod] Some commit.
```

Во-вторых, поскольку сохранное состояние всегда воссоздается против исходной фиксации, команда всегда будет успешно выполнена, а восстановление состояние - отброшено из стека состояний.

Наконец, воссоздание сохраненного состояние автоматически не фиксирует ни одно из ваших изменений в новой ветке. Все сохраненные модификации файла (изменения индекса, если требуется) останутся в вашем рабочем каталоге на новой активной ветке.

```
$ git commit --dry-run
# On branch mod
# Changes not staged for commit:
# (use «git add <file>...» to update what will be committed)
# (use «git checkout -- <file>...» to discard changes in
working directory)
#
# modified: bar
# modified: foo
#
no changes added to commit (use «git add» and/or «git commit
-a»)
```

В этой точке, конечно, вы можете фиксировать изменения на новую ветку, если посчитаете необходимым. Нет, это не чудодейственное средство против конфликтов слияния. Однако, если при попытке восстановления сохраненного состояния непосредственно на ветку master будут конфликты, попытайтесь восстановиться в новую ветку. Однако слияние этой новой ветки с веткой master приведет к тем же конфликтам слияния.

```
$ git commit -a -m "Stuff from the stash"
[mod 42c104f] Stuff from the stash
```

```
2 files changed, 2 insertions(+), 0 deletions(-)

$ git show-branch
! [master] Another mod
 * [mod] Stuff from the stash
--
*  [mod] Stuff from the stash
+  [master] Another mod
+  [master^] Drifting file state.
+* [mod^] Some commit.

$ git checkout master
Switched to branch 'master'

$ git merge mod
Auto-merging foo
CONFLICT (content): Merge conflict in foo
Auto-merging bar
CONFLICT (content): Merge conflict in bar
Automatic merge failed; fix conflicts and then commit the
result.
```

Напоследок позвольте дать небольшой совет относительно команды git stash. Пусть приведенная аналогия будет грубой, но я все же ее приведу: вы называете своих домашних животных, но нумеруете свой скот. Аналогично, ветки называют, а stash-состояния - нумеруют. Возможность сохранять рабочее состояние - прекрасна, но не злоупотребляйте ею и не создавайте слишком много состояний. Если вы хотите долго хранить то или иное состояние, преобразуйте его в ветку.

11.2. Reflog

Хорошо, я признаюсь. Иногда Git делает что-то, что выглядит чем-то таинственным или волшебным и что заставляет задавать вопрос: «Что же произошло?». Иногда вопрос может звучать иначе: «Что же он сделал? Он

не должен быть сделать этого!». Но уже слишком поздно, вы потеряли главную фиксацию и результаты целой недели работы!

Не волнуйтесь! Git предоставляет команду **reflog**, которая, по крайней мере, поможет разобраться в том, что же произошло. У вас также будет возможность восстановить потерянные фиксации в случае, если что-то таки потерялось.

Reflog - это запись изменений в головы веток в пределах непустых репозитариев. Каждый раз, когда делается изменение ссылки, в том числе и HEAD, reflog обновляет запись, содержащую информацию об этом изменении. Думайте об этом механизме как о следе из хлебных крошек, который позволяет определить, где вы были.

Reflog записывает сведения о следующих операциях:

- Клонирование;
- Помещение данных в удаленный репозитарий;
- Создание новых фиксаций;
- Изменение или создание веток;
- Операции перебазирования;
- Операции сброса.

Обратите внимание, что некоторые сложные операции, такие как git filter-branch, в конечном счете, сводятся к простым фиксациям и тоже регистрируются. Любая операция Git, изменяющая вершину ветки, будет записана.

По умолчанию reflog включен во все непустых репозитариях и отключен в пустых репозитариях. В частности, reflog контролируется логической опцией конфигурации core.logAllRefUpdates. Он может быть включен командой git config core.logAllRefUpdates true или отключен путем передачи значения false в ту же команду. Reflog включается/отключается отдельно для каждого репозитария.

На что же похож **reflog**?

```
$ git reflog show
a44d980 HEAD@{0}: reset: moving to master
```

```
79e881c HEAD@{1}: commit: last foo change
a44d980 HEAD@{2}: checkout: moving from master to fred
a44d980 HEAD@{3}: rebase -i (finish): returning to refs/heads/
master
a44d980 HEAD@{4}: rebase -i (pick): Tinker bar
a777d4f HEAD@{5}: rebase -i (pick): Modify bar
e3c46b8 HEAD@{6}: rebase -i (squash): More foo and bar with
additional stuff.
8a04ca4 HEAD@{7}: rebase -i (squash): updating HEAD
1a4be28 HEAD@{8}: checkout: moving from master to 1a4be28
ed6e906 HEAD@{9}: commit: Tinker bar
6195b3d HEAD@{10}: commit: Squash into 'more foo and bar'
488b893 HEAD@{11}: commit: Modify bar
1a4be28 HEAD@{12}: commit: More foo and bar
8a04ca4 HEAD@{13}: commit (initial): Initial foo and bar.
```

Не смотря на то, что reflog записывает транзакции для всех ссылок, **git reflog show** показывает транзакции только для одной ссылки за один раз. Предыдущий пример показывает ссылку по умолчанию, то есть HEAD. Если вы хотите просмотреть журнал изменений для другой ветки, укажите ее в качестве параметра. В следующем примере мы выводим журнал изменений ветки fred:

```
$ git reflog fred
a44d980 fred@{0}: reset: moving to master
79e881c fred@{1}: commit: last foo change
a44d980 fred@{2}: branch: Created from HEAD
```

Каждая строка - это отдельная транзакция из истории ссылки, начиная с самого последнего изменения и вплоть до самого первого. Самый первый столбец - это ID фиксации на тот момент, когда она была сделана. Записи вроде HEAD@{7} во втором столбце предоставляют удобные имена для фиксации в каждой транзакции. Здесь HEAD@{0} - самая последняя запись, HEAD@{1} - предпоследняя и т.д. Самая старая запись в нашем примере - HEAD@{13}, это начальная запись в данном репозитарии. Оставшаяся часть каждой строки описывает, что же произошло. Также для каждой

транзакции записывается время и дата ее осуществления (но здесь не показывается).

Reflog выводит имена фиксаций вроде HEAD@{1}, которые можно использовать в качестве символьных имен фиксаций для любой другой команды Git. Например:

```
$ git show HEAD@{10}
commit 6195b3dfd30e464ffb9238d89e3d15f2c1dc35b0
Author: Федя Колобков <kolobok@lobok.com>
Date: Sat Oct 29 09:57:05 2020 -0500
Squash into 'more foo and bar'
diff --git a/foo b/foo
index 740fd05..a941931 100644
--- a/foo
+++ b/foo
@@ -1,2 +1 @@
-Foo!
-more foo
+junk
```

Это означает, что в процессе разработки вы записываете фиксации, перемещаете ветки, выполняете перебазирование и другие манипуляции, но вы всегда можете обратиться к reflog, чтобы узнать символьное имя фиксации. Имя HEAD@{1} всегда ссылается на предыдущую фиксацию ветки, имя HEAD@{2} - на основную фиксацию до этого и т.д. Имейте в виду, что не всегда имя транзакции представляет имя фиксации. Ведь reflog записывает каждое изменение ветки, в том числе, когда вы перемещаете вашу ветку на другую фиксацию. Поэтому HEAD@{3} не всегда означает третью с конца операцию git commit.

> **Примечание. Вы испортили слияние и хотите попытаться снова? Используйте команду git reset HEAD@{1}. Добавьте параметр --hard, если нужно.**

Git также поддерживает англоязычные спецификаторы, позволяющие указать, когда была сделана транзакция. Следующая команда показывает фиксацию HEAD, сделанную в прошлую субботу:

```
$ git log 'HEAD@{last saturday}'
commit 1a4be2804f7382b2dd399891eef097eb10ddc1eb
Author: Федя Колобков <kolobok@lobok.com>
Date: Sat Oct 29 09:55:52 2020 -0500

More foo and bar

commit 8a04ca4207e1cb74dd3a3e261d6be72e118ace9e
Author: Федя Колобков <kolobok@lobok.com>
Date: Sat Oct 29 09:55:07 2020 -0500

Initial foo and bar.
```

Git поддерживает много спецификаторов для ссылок. Например: yesterday (вчера), noon (полдень), midnight (полночь), названия дней недели, названия месяцев, A.M., P.M., абсолютные даты и время, относительные фразы вроде last Monday (прошлый понедельник), 1 hour ago (1 час назад), 10 minutes ago (10 минут назад) и их комбинации вроде 1 day 2 housr ago (1 день 2 часа назад).

Также вместо названия ветки, например, fix@{noon} вы можете использовать сокращение @{noon}, при условии, что ветка fix - активна. Сокращение @{noon} подразумевает использование текущей ветки.

Несмотря на то, что эти спецификаторы довольно либеральны, они не совершенны. Помните, что Git использует эвристику, чтобы интерпретировать их. Также помните, что понятие времени локально и относительно вашего локального репозитария. Используя ту же фразу в другом репозитарии, вы получите разные результаты. Если у вас нет истории reflog, покрывающей заданный период, вы получите предупреждение. Например:

```
$ git log HEAD@{last-monday}
warning: Log for 'HEAD' only goes back to Sat, 29 Oct 2020
09:55:07 -0500.
```

```
commit 8a04ca4207e1cb74dd3a3e261d6be72e118ace9e
Author: Федя Колобков <kolobok@lobok.com>
Date: Sat Oct 29 09:55:07 2020 -0500

Initial foo and bar.
```

Одно последнее предупреждение. Не позволяйте оболочке обманывать вас. Между этими двумя командами есть значительная разница:

```
# Неправильно!
$ git log dev@{2 days ago}
# Правильно для вашей оболочке
$ git log 'dev@{2 days ago}'
```

Второй вариант важен для командной оболочки, чтобы она считала название ветка со спецификаторами в ней одним параметром командной строки, в противном случае строка dev@{2 days ago} будет расценена как 3 параметра. Чтобы упростить проблему разрыва параметра, Git позволяет указывать спецификаторы так:

```
# Все три варианта аналогичны
$ git log 'dev@{2 days ago}'
$ git log dev@{2.days.ago}
$ git log dev@{2-days-ago}
```

Есть еще один повод для беспокойства. Если Git поддерживает полную историю транзакций, то как это сказывается на дисковом пространстве?

К счастью, об этом не нужно беспокоиться. Время от времени Git запускает процесс сборки «мусора». Во время этого процесса удаляются некоторые старые записи **reflog**. Недостижимые фиксации удаляются спустя 30 дней после создания, а достижимые фиксации - спустя 90 дней.

Если такое расписание для вас не подходит, используйте переменные конфигурации gc.reflogExpireUnreachable и gc.reflogExpire соответственно. Также вы можете использовать команду git reflog delete для удаления отдельных записей или использовать команду git reflog expire для непосред-

ственного удаления старых записей. Эта же команда может использоваться для принудительного истечения записей reflog:

```
$ git reflog expire --expire=now --all
$ git gc
```

Как вы могли предложить к настоящему времени, stash и reflog глубоко связаны. Фактически, stash реализован на базе reflog.

Одна важная деталь реализации: reflog хранит свои записи в каталоге .git/logs. Файл .git/logs/HEAD содержит историю значений HEAD, а каталог .git/logs/refs содержит историю всех ссылок, в том числе stash. Подкаталог .git/logs/refs/heads содержит историю голов веток.

Вся информация хранится в reflog. Удаление каталога .git/logs или отключение reflog нарушает внутреннюю структуру данных Git. Это означает, что все ссылки вида master@{4} больше не могут быть разрешены.

С другой стороны, включенный **reflog** позволяет доступ к фиксациям, которые иначе могут быть недостижимыми. Если вы пытаетесь очистить репозитарий с целью уменьшения его размера, отключение reflog приведет к удалению недостижимых (то есть неважных) фиксаций.

Глава 12.

Удаленные репозитарии



До сих пор вы работали с одним локальным репозитарием. Пришло время рассмотреть хваленные распределенные функции Git и изучить, как сотрудничать с другими разработчиками через совместно используемые репозитарии. Для работы с удаленными репозитариями вам нужно знать несколько новых терминов.

Клон - копия репозитария. Клон содержит все объекта оригинального репозитария, в результате каждый клон - это независимый и автономный репозитарий и истинный, симметричный колена оригинала. Клон позволяет каждому разработчику работать локально и независимо без централизации, загрузок и блокировок. В конечном счете, клонирование способствует масштабируемости Git и стирает географические границы.

Отдельные репозитарии особенно полезны в следующих ситуациях:

- Разработчики работают автономно

- Разработчики распределены по всему миру. Группа разработчиком в одном месте может совместно использовать локальный репозитарий для накопления локализованных изменений
- Ожидается, что будут несколько версий проекта, которые будут существенно отличаться друг от друга

В предыдущих главах мы рассмотрели ветки, которые можно создавать в пределах одного репозитария, а также механизмы их слияния. Однако при желании и для получения большей масштабируемости отдельные направления разработки можно помещать в различные репозитарии, которые могут быть снова объединены, если это уместно. Клонирование репозитария является первым шагом в совместном использовании кода. Вы также должны связать один репозитарий с другим, чтобы установить пути для обмена данными. Git устанавливает эти соединения репозитария через *удаленные*.

Удаленная - это ссылка (или дескриптор) на другой репозитарий через файловую систему или сетевой путь. Вы можете использовать удаленную как сокращенное имя для длинного и сложного Git URL. Вы можете определить любое число удаленных в репозитарии.

Как только удаленная установлена, Git может передавать данные от одного репозитария к другому с помощью модели push или pull. Установившаяся практика - передавать данные фиксации от исходного репозитария к его клону из соображений синхронизации клона. Вы можете создать удаленную для перемещения данных от клона к его оригиналу или же настроить двунаправленный режим обмена данными.

Чтобы отслеживать данные из других репозитариев Git использует *удаленные ветки отслеживания* (remote-tracking branches). Каждая удаленная ветка - это ветка, которая работает как прокси для определенной ветки в удаленном каталоге. Вы можете настроить *локальные ветки отслеживания*, которые формируют базис для интеграции ваших локальных изменений с удаленными изменения из соответствующей удаленной ветки.

Наконец, вы можете сделать свой репозитарий доступным для других. Git обычно называет это *публикацией* репозитария и предоставляет для этой задачи несколько методов. В этой главе будут рассмотрены примеры и методы для совместного использования, отслеживания и получения данных через множество репозитариев.

12.1. Понятия репозитария

Чистый репозитарий и репозитарий разработки

Репозитарий в Git может быть или *чистый* (bare) или *репозитарием разработки* (development, nonbare).

Репозитарий разработки используется для нормальной, ежедневной разработки. Он обслуживает текущую ветку и предоставляет копию текущей ветки в рабочем каталоге. Все репозитарии, с которыми вы сталкивались до этого в данной книге, были репозитариями разработки.

У чистого репозитария нет рабочего каталога, и он не может использоваться для обычной разработки. В чистом репозитарии нет понятия ветки. Думайте о нем как о просто содержимом каталога .git. Другими словами, вы не должны делать фиксации в чистом репозитарии.

Чистый репозитарий, как может показаться, мало полезен, но его роль крайне важна: он служит авторитетным фокусом для совместной разработки. Операции **clone**, **fetch** и **push** выполняются именно через чистый репозитарий. Чуть позже мы рассмотрим пример, как все это происходит.

Если вы запустите команду git clone с опцией --bare, Git создаст чистый репозитарий. В противном случае будет создан репозитарий разработки.

Примечание. Заметьте, мы не говорили, что git clone --bare создает новый или пустой репозитарий. Мы сказали, что эта команда создает чистый репозитарий. И этот только что клонированный репозитарий будет содержать копию содержимого от upstream-репозитария.

Команда git init создает новый и пустой репозитарий, но этот новый репозитарий может быть в одном из вариантов - чистый или разра-

ботки. Для большей конкретики нужно указывать опцию --bare, если нужен именно чистый репозитарий:

```
$ cd /tmp
$ git init fluff2
Initialized empty Git repository in /tmp/fluff2/.git/
$ git init --bare fluff
Initialized empty Git repository in /tmp/fluff/
```

По умолчанию Git включает **reflog** (запись изменений в ссылках) на репозитариях разработки, но reflog выключен на чистых репозитариях. Это снова означает, что именно первый тип репозитариев предназначен для разработки, но никак не последний. По тем же причинам нельзя создать удаленные в чистом репозитарии.

Если вы настраиваете репозитарий, в который разные разработчики будут помещать (с помощью команды push) изменения, он должен быть чистым.

Клоны репозитариев

Команда **git clone** создает новый репозитарий Git, основанный на оригинальном репозитарии, который вы определяете через файловую систему или сетевой адрес. Git не копирует всю информацию из оригинала в клон. Git игнорирует информацию, которая подходит только для оригинального репозитария, например, удаленные ветки отслеживания.

При нормальном использовании git clone локальные ветки разработки исходного репозитария, сохраненные в refs/heads становятся *удаленными ветками отслеживания* в новом репозитарии-клоне и помещаются в refs/remotes. Удаленные ветки отслеживания исходного репозитария не клонируются.

Теги исходного репозитария копируются в клон, как и все достижимые объекты. Однако, специфичная для репозитария информация, такая как перехватчики (hooks), конфигурационные файлы, reflog и stash исходного репозитария не воспроизводится в клоне.

В главе 3 было показано, как использовать git clone для создания клона вашего репозитария public_html:

```
$ git clone public_html my_website
```

Здесь public_html рассматривается как исходный, «удаленный» репозитария. Клон будет называться my_website.

Аналогично, вы можете использовать git clone для создания копии удаленного репозитария, например:

```
# Все в одной строке...
$ git clone \
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-
2.6.git
```

По умолчанию, каждый новый клон содержит ссылку обратно на свой родительский репозитарий через удаленную с именем *origin*. Однако сам родительский репозитарий ничего не знает об этой ссылке. Это типичные односторонние отношения.

Если вам не нравится имя «origin», вы можете изменить его, указав опцию --origin имя во время операции клонирования.

Git также настраивает удаленную origin для refspec fetch:

```
fetch = +refs/heads/*:refs/remotes/origin/*
```

Установка этого **refspec** означает, что вы хотите обновлять ваш локальный репозитарий, получая изменения от исходного репозитария. В этом случае ветки удаленного репозитария доступны в клоне с именами с префиксом origin/, например, origin/master, origin/dev или origin/maint.

Удаленные

Репозитарий, в котором вы в настоящее время работаете, вызывают *локальным* или *текущим репозитарием*, а репозитарий, с которым Вы обмениваетесь файлами, вызывают *удаленным репозитарием*. Но последний термин не очень точный, потому что этот репозитарий может физически находить-

ся, как на удаленной машине, так и на вашей локальной машине - это просто может быть другой репозитарий в локальной файловой системе.

Git использует удаленные отслеживания для упрощения соединения с другим репозитарием. Удаленная ветка предоставляет дружественное имя для репозитария и может использоваться вместо фактического URL репозитария. Для создания, удаления, манипуляции и просмотра удаленных используйте команду **git remote**. Все созданные удаленные регистрируются в файле .git/config. Также все удаленные могут управляться посредством команды **git config**.

В дополнение к git clone доступны другие команды, относящиеся к удаленным репозитариям:

git fetch
Получает объекты и связанные с ними метаданные из удаленного репозитария.

git pull
Подобна git fetch, но также объединяет изменения в соответствующую локальную ветку.

git push

Отправляет объекты и относящиеся к ним метаданные в удаленный репозитарий

git ls-remote

Показывает список ссылок, хранимых данной удаленной. Позволяет ответить на вопрос: «Доступны ли обновления?».

Ветки отслеживания

Как только вы клонируете репозитарий, вы можете синхронизироваться с оригинальным репозитарием, даже если вы делаете локальные фиксации и создаете локальные ветки.

По мере развития самого Git развивалась и терминология. Некоторая терминология уже успела стать стандартной. Чтобы разъяснить назначение разных веток были созданные разные пространства имен. Несмотря на то, что любые ветки в вашем локальном репозитарии все еще считаются локальными ветками, они могут быть разделены на разные категории.

- *Удаленные ветки отслеживания* - ассоциируются с удаленными и имеют специальное назначение следовать за изменениями каждой ветки в удаленном репозитарии.
- *Локальная ветка отслеживания* - это форма ветки интеграции, которая собирает изменения из вашей локальной разработки и изменения из удаленной ветки отслеживания.
- Любую локальную, неотслеживаемую ветку обычно называют топиком или веткой разработки.
- Наконец, удаленная ветка - это ветка, расположенная в удаленном репозитарии. Она может быть источником для удаленной ветки отслеживания.

Во время операции клонирования Git создает удаленную ветку отслеживания в клоне для каждой ветки разработки в удаленном репозитария. Набор удаленных веток отслеживания представлен в новом, отдельном пространстве имен в локальном репозитарии, который является определенным для удаленного, клонируемого. Локальный репозитарий использует свои удаленные ветки отслеживания, чтобы отслеживать изменения, внесенные в удаленном репозитарии.

Примечание. Вы можете вспомнить ссылки и символьные ссылки из главы 6 и что локальная ветка разработки, которую вы называете dev, на самом деле называется refs/heads/dev. Точно так же удаленные вет-

ки отслеживания хранятся в пространстве имен refs/remotes. Следовательно, ветка удаленного отслеживания origin/master на самом деле - это refs/remotes/origin/master.

Поскольку удаленные ветки отслеживания помещены в собственное пространство имен, есть четкое разделение между ветками, созданными в репозитарии вами (ветки разработки), и ветками, которые, фактически, основаны на удаленном репозитарии (удаленные ветки отслеживания).

Все операции, которые вы можете выполнить на обычной ветке, могут быть также выполнены на ветке отслеживания. Однако есть определенные ограничения и инструкции, которые нужно соблюдать. Поскольку удаленные ветки отслеживания используются исключительно для наблюдения за изменениями другого репозитария, вы должны обрабатывать их в режиме только чтение. Вы не должны пытаться объединять их или создавать фиксации на них. В противном случае удаленные ветки отслеживания выйдут из синхронизации с удаленным репозитарием. Надлежащее управление ветками отслеживания описано далее в этой главе.

12.2. Ссылки на другие репозитарии

Чтобы координировать ваш репозитарий с другим репозитариям, вы определяете удаленную, которая сохранена в файле конфигурации. Она состоит из двух частей. Первая часть задает имя другого репозитария в форме URL. Вторая часть называется refspec и она определяет, как ссылка (которая обычно представляет ветку) должна быть отображена от пространства имен одного репозитария в пространство имен другого репозитария.

Давайте посмотрим на каждый из этих компонентов поочередно.

Ссылки на удаленные репозитарии

Git поддерживает несколько форм URL (Uniform Resource Locators), которые вы можете использовать для задания имен удаленных репозитариев. Эти формы определяют протокол доступа и размещение или адрес данных.

Технически, Git не поддерживает ни URL, ни URI (универсальные идентификаторы ресурса), поскольку формы Git URL не соответствуют, ни RFC 1738 или RFC 2396. Поэтому формы URL, которые поддерживаются Git, для краткости мы будем называть Git URL. Кроме того, в файле .git/config также используются Git URL.

Самая простая форма Git URL ссылается на репозитарий, находящийся в локальной файловой системе, будь это физическая файловая система или виртуальная файловая система, смонтированная локально через сетевую файловую систему (Network File System, NFS). Рассмотрим две перестановки:

```
/путь/к/репо.git
file:///путь/к/репо.git
```

Несмотря на то, что эти две формы чрезвычайно похожи, между ними есть тонкое, но важное отличие. Чтобы избежать проблем, связанных с совместно используемыми репозитариями, рекомендуется использовать форму file://.

Другие формы Git URL относятся к репозитариям в удаленных системах. Когда у вас есть действительно удаленный репозитарий, данные которого нужно получить через сеть, самую эффективную форму передачи данных начато называют собственным протоколом Git (Git native protocol). Это внутренний протокол Git, используемый для передачи данных. Примеры URL с использованием этого протокола:

```
git://example.com/путь/к/репо.git
git://example.com/~пользователь/путь/к/репо.git
```

Git-daemon использует эти формы для публикации репозиториев для анонимного чтения. Эти формы URL можно использовать, как в операции clone, так и в операции fetch.

Данные формы не предусматривают аутентификации клиента, никакие пароли для доступа к репозитарию не запрашиваются. Следовательно, как может использоваться формат ~пользователь для ссылки на домашний каталог пользователя, если ~ не несет никакой смысловой нагрузки. По сути, здесь нет аутентифицированного пользователя, чей бы домашний каталог можно было использовать. Поэтому форма ~пользователь работает только, если сторона сервера разрешает ее опцией --user-path.

Для безопасных, аутентифицируемых соединения Git предлагает туннелирование посредством SSH-соединения:

```
ssh://[пользователь@]example.com[:порт]/путь/к/репо.git
ssh://[пользователь@]example.com/путь/к/репо.git
ssh://[пользователь@]example.com/~пользователь2/путь/к/репо.
git
ssh://[пользователь@]example.com/~/путь/к/репо.git
```

Третья форма позволяет два разных имени пользователя. Первое используется для аутентификации сеанса, а второе - используется для доступа к домашнему каталогу.

Git также поддерживает URL в scp-стиле. Они подобны SSH-формату, но в них не указывается порт:

```
[пользователь@]example.com:/путь/к/репо.git
[пользователь@]example.com:~пользователь/путь/к/репо.git
[пользователь@]example.com:путь/к/репо.git
```

Хотя протоколы HTTP и HTTPS поддерживались с самых ранних версий Git, в версии 1.6.6 они подверглись важным изменениям.

```
http://example.com/путь/к/репо.git
https://example.com/путь/к/репо.git
```

До версии 1.6.6 ни HTTP, ни HTTPS не были так эффективны, как собственный протокол Git. В версии 1.6.6 была существенно улучшена поддержка протоколов HTTP/HTTPS и они стали столь же эффективными как собственный протокол Git.

Учитывая улучшенную поддержку HTTP/HTTPS, теперь эти формы URL станут более популярными. Хотя бы потому, что большинство корпоративных брандмауэров оставляет порты 80 (HTTP) и 443 (HTTPS) открытыми, а родной протокол Git использует порт 9418, который обычно блокируется, что требует настройки брандмауэра. К тому же эти протоколы используются многими популярными хостинг-сайтами Git вроде GitHub.

Наконец, поддерживается протокол Rsync:

```
rsync://example.com/путь/к/репо.git
```

Протокол Rsync использовать не рекомендуется. С его помощью вы можете создать только начальный клон, но если вы продолжите использовать этот протокол для дальнейших обновлений, это может привести к потере локально созданных данных.

Refspec

В главе 6 было показано, как ссылка именует определенную фиксацию в истории репозитария. Обычно ссылка - это имя ветки. **Refspec** - это особый вид ссылки, который отображает имена веток удаленного репозитария в имена веток в вашем локальном репозитарии.

Поскольку refspec-ссылка должна одновременно ссылаться и на локальный и на удаленный репозитарии, в refspec часто требуется указывать полные имена веток. В refspec имена веток разработки начинаются префиксом refs/heads, а имена удаленных веток отслеживания начинаются префиксом refs/remotes/.

Refspec-синтаксис следующий:

```
[+]источник:назначение
```

Refspec состоит из двух частей: ссылки источника и ссылки назначения, разделенных двоеточием. В начале строки может стоять «плюс» (+), который указывает, что во время передачи не будет использоваться обычная проверка безопасности. Кроме того, звездочка (*) означает все имена веток. В некоторых случаях ссылка `источник` необязательна, но вторая часть, то есть двоеточие и `назначение` - обязательна.

Refspec-ссылки используются командами git fetch и git push. Но здесь есть особый трюк. Формат refspec-ссылки всегда одинаковый, то есть `источник:назначение`. Однако, роли источника и назначения зависит от выполняемой Git-операции. Здесь вам поможет таблица 12.1.

Таблица 12.1. Потоки данных refspec

Операция	Источник	Назначение
push	Локальная ссылка (будет отправлена)	Удаленная ссылка (будет обновлена)
fetch	Удаленная ссылка (будет получена)	Локальная ссылка (будет обновлена)

Типичная команда git fetch использует ссылку refspec вида:

```
+refs/heads/*:refs/remotes/remote/*
```

Этот refspec можно перефразировать так:

> Все ветки источника из удаленного репозитария в пространстве имен ref/heads будут отображены в ваш локальный репозитарий с использованием имени, собранного из имени remote и помещены в пространство имен refs/renotes/remote.

Вы можете использовать звездочку (*), например, refs/heads/*, что позволяет применять такую маску сразу ко многим веткам. Из-за звездочек эта refspec применяется ко всем веткам, которые будут найдены в refs/heads

удаленной. Эта спецификация заставляет ветки темы отображаться в пространство имен вашего репозитария в качестве удаленных веток отслеживания и разделяет их на подимена на основе удаленного имени.

> **Примечание. Используйте команду** `git show-ref` **для вывода списка ссылок в вашем текущем репозитарии. Команда** `git ls-remote репозитарий` **выводит ссылки в удаленном репозитарии.**

Поскольку первый шаг команды **git pull** - это выборка (fetch), refspec выборки также применяется и к git pull. Вы не должны производить фиксации или слияния на удаленной ветке отслеживания на правой стороне refspec-ссылки команд pull или fetch. Эти ссылки будут использоваться как удаленные ветки отслеживания.

Операция **git push** обычно используется для публикации изменений, которые вы сделали в ваших локальных ветках разработки. Чтобы разрешить другим пользователям найти ваши изменения в удаленном репозитарии после того, как вы загрузите их, ваши изменения должны появиться в том репозитарии как ветки разработки.

Итак, во время обычной команды git push ветки из вашего репозитария отправляются на удаленный репозитарий, при этом используется примерно такая refspec:

```
+refs/heads/*:refs/heads/*
```

Данную refspec можно перефразировать так:

> Из локального репозитария взять каждую ветку, найденную в пространстве имен refs/heads/ и поместить ее в пространство refs/heads удаленного репозитария под тем же именем.

Первая refs/heads ссылается на ваш локальный репозитарий, а вторая ссылается на удаленный репозитарий. Звездочки позволяют убедиться, что все ветки будут реплицированы.

Командам **git fetch** и **git push** можно передать несколько refspec-ссылок. Можно вообще не указывать refspec в команде git push. Но как же тогда Git узнает, куда отправить данные?

Во-первых, если явно удаленный репозитарий не указан в командной строке, Git предполагает, что вы хотите использовать origin. Без указания refspec команда git push отправит ваши фиксации в удаленный репозитарий для всех веток, которые будут общими между вашим локальным и удаленным репозитариями. Любая локальная ветка, которой нет в удаленном репозитарии, не будет отправлена. Таким образом, чтобы записать в удаленный репозитарий новую ветку, это нужно сделать явно - по имени этой ветки. После их можно будет отправлять одной командой git push. Исходя из всего сказанного, следующие две команды эквивалентны:

```
$ git push origin branch
$ git push origin branch:refs/heads/branch
```

12.3. Примеры использования удаленных репозитариев

Теперь у вас есть необходимый минимум, чтобы приступить к некоторому сложному совместному использованию Git. В этом разделе будет показано, как использовать несколько репозитариев на одной физической машине. В реальной жизни все будет практически также, за исключением того, что, возможно, эти репозитарии будут расположены на разных узлах Интернета.

Давайте рассмотрим общий сценарий использования Git. Ради иллюстрации давайте настроим репозитарий, который все разработчики будут считать авторитетным, несмотря на то, что технически он ничем не отличается от других репозитариев. Другими словами, авторитетность основана лишь в том, что все считают его авторитетном, а не в каких-либо особых мерах безопасности.

Данный репозитарий, который мы будем считать авторитетным, часто помещается в специальный каталог и называется *базой* (depot). Не нужно употреблять термины **главный** (master) и **репозитарий** (repository), чтобы не было путаницы в терминологии.

Часто есть серьезные основания для того, чтобы настроить базу. Например, вы хотите заставить своих коллег помещать свои наработки в базу, чтобы избежать катастрофических потерь. База будет удаленным источником для всех разработчиком.

Следующие разделы показывают вам, как создать базу, клонировать ее, а также произвести синхронизацию с ней.

Чтобы проиллюстрировать параллельную разработку базе, ее клонирует второй разработчик, после чего он будет работать с клоном, а затем поместит свои изменения обратно в базу, чтобы они стали доступны всем.

Создание авторитетного репозитария

Вы можете поместить свою базу в любой место вашей файловой системы. В этом примере мы будем использовать каталог /tmp/Depot. В этом каталоге или в одном из его репозитариев не должна вестись фактическая разработка. Вместо этого отдельные разработчики должны использовать его клоны.

На практике этот авторитетный репозитарий часто размещается на некотором сервере, например, на GitHub, git.kernel.org или одной из частных машин.

Данные шаги обрисовывают в общих чертах то, что нужно, чтобы преобразовать репозитарий в другой чистый клон, способный быть авторитетным репозитарием, то есть базой.

Первый шаг - заполнить /tmp/Depot начальным репозитарием. Представим, что вы хотите работать с содержимым веб-сайта, которое сейчас находится в ~/public_html. Также представим, что в ~/public_html уже есть Git-репозитарий, поэтому нам нужно сделать копию репозитария ~/public_html и поместить его в /tmp/Depot/public_html.git.

```
#  Считаем, что в ~/public_html уже создан репозитарий Git
```

```
$ cd /tmp/Depot/
$ git clone --bare ~/public_html public_html.git
Initialized empty Git repository in /tmp/Depot/public_html.
git/
```

Команда **clone** копирует удаленный репозитарий Git из ~/public_html в текущий рабочий каталог, то есть в /tmp/Depot. Последний аргумент - это имя нового репозитария, то есть public_html.git. В данном случае мы используем имя с суффиксом .git. Это не обязательно, но считается хорошим тоном.

У исходного репозитария разработки есть полный набор файлов проекта, проверенных на верхнем уровне, хранилище объектов и все конфигурационные файлы расположены в подкаталоге .git:

```
$ cd ~/public_html/
$ ls -aF
./ fuzzy.txt index.html techinfo.txt
../ .git/ poem.html
$ ls -aF .git
./ config hooks/ objects/
../ description index ORIG_HEAD
branches/ FETCH_HEAD info/ packed-refs
COMMIT_EDITMSG HEAD logs/ refs/
```

Поскольку у чистого (bare) репозитария нет рабочего каталога, у его файлов более простая разметка:

```
$ cd /tmp/Depot/
$ ls -aF public_html.git
./ branches/ description hooks/ objects/ refs/
../ config HEAD info/ packed-refs
```

Теперь вы можете обработать этот чистый репозитарий /tmp/Depot/public_html.git как авторитетную версию.

Поскольку использовалась опция --bare во время операции клонирования, Git не представил удаленную remote.

Далее приведена конфигурация нового, чистого репозитария:

```
# В /tmp/Depot/public_html.git
$ cat config
[core]
      repositoryformatversion = 0
      filemode = true
      bare = true
```

Создание вашей собственной удаленной origin

Прямо сейчас у вас есть два репозитария, которые фактически идентичны, за исключением того, что у начального репозитария есть рабочий каталог, а у клона - нет.

Кроме того, поскольку репозитарий ~/public_html в вашем домашнем каталоге создавался с использованием git init, а не через операцию клонирования, у него нет origin. Фактически, у него вообще нет удаленной конфигурации.

Однако ее достаточно просто добавить. В некотором смысле вы должны вручную преобразовать свой начальный репозитарий в полученный клон. Разработчик, который клонирует базу первым, автоматически создаст origin.

Команда для управления удаленными называется **git remote**. Эта операция представляет несколько новых настроек в файле .git/config:

```
$ cd ~/public_html
$ cat .git/config
[core]
      repositoryformatversion = 0
      filemode = true
      bare = false
      logallrefupdates = true

$ git remote add origin /tmp/Depot/public_html
```

```
$ cat .git/config
[core]
      repositoryformatversion = 0
      filemode = true
      bare = false
      logallrefupdates = true
[remote «origin»]
      url = /tmp/Depot/public_html
      fetch = +refs/heads/*:refs/remotes/origin/*
```

Команда git remote добавила новый раздел remote с названием origin. Имя origin не является каким-то магическим или специальным. Вы можете использовать любое другое имя, если в этом есть необходимость.

Удаленная устанавливает связь между вашим текущим репозитарием и удаленным репозитарием, в нашем случае с /tmp/Depot/public_html.git, как указано в параметре url. Как уже было отмечено, использовать суффикс .git совсем необязательно: и /tmp/Depot/public_html, и /tmp/Depot/public_html.git будут работать. Теперь в этом репозитарии вы можете использовать имя origin для доступа к удаленному репозитарию.

Отношения между репозитарием, который содержит удаленную ссылку и удаленным репозитарием асимметричны. Удаленная всегда указывает в одном направлении от репозитария до удаленного репозитария. Другими словами: клон всегда знает, какой репозитарий является основным, но основной репозитарий ничего не знает о клонах.

Давайте завершим процесс настройки удаленной **origin** путем установки новой удаленной ветки отслеживания в исходном репозитарии для представления веток из удаленного репозитария. Во-первых, как вы можете видеть, здесь есть только одна ветка - master:

```
# Вывод всех веток
$ git branch -a
* master
```

Теперь используйте команду **git remote update**:

```
$ git remote update
```

```
Updating origin
From /tmp/Depot/public_html
* [new branch] master -> origin/master

$ git branch -a
* master
  origin/master
```

В зависимости от вашей версии Git удаленная ссылка отслеживания может быть покащана без или с префиксом remotes/:

```
* master
remotes/origin/master

$ git branch -a
* master
  remotes/origin/master
```

Git представил новую ссылку с именем origin/master в репозитарий. Это удаленная ветка отслеживания в удаленной origin. Никто не ведет разработку в этой ветке. Вместо этого ее назначение - хранение и отслеживание изменений, сделанных в удаленной origin ветки master репозитария. Вы должны считать ее прокси локального репозитария для фиксаций, сделанных в удаленной, в конечном счете, вы можете использовать ее, чтобы перенести те фиксации в ваш репозитарий.

Фраза «Updating origin», выводимая командой **git remote update**, вовсе не означает, что удаленный репозитарий будет обновлен. Вместо этого она означает, что будет обновлена ссылка origin локального репозитария. Эта ссылка будет обновлена на основе информации, полученной из удаленного репозитария.

> **Примечание. Команда git remote update обновит каждую удаленную в пределах репозитария. Вместо того, чтобы обновлять все удаленные, вы можете указать имя определенной удаленной:**
>
> ```
> $ git remote update имя_удаленной
> ```

Также вы можете указать опцию -f, чтобы немедленно произвести выборку (fetch) из удаленного репозитария, заданного опцией -f:

```
$ git remote add -f origin репозитарий
```

Теперь вы соединили ваш репозитарий с удаленным репозитарием в вашей базе.

Разработка в вашем репозитарии

Давайте произведем некоторую работу в вашем репозитарии и добавим другую поэму - файл fuzzy.txt:

```
$ cd ~/public_html
$ git show-branch -a
[master] Merge branch 'master' of ../my_website

$ cat fuzzy.txt
Fuzzy Wuzzy was a bear
Fuzzy Wuzzy had no hair
Fuzzy Wuzzy wasn't very fuzzy,
Was he?

$ git add fuzzy.txt
$ git commit
Created commit 6f16880: Add a hairy poem.
1 files changed, 4 insertions(+), 0 deletions(-)

create mode 100644 fuzzy.txt

$ git show-branch -a
* [master] Add a hairy poem.
 ! [origin/master] Merge branch 'master' of ../my_website
--
*  [master] Add a hairy poem.
-- [origin/master] Merge branch 'master' of ../my_website
```

В этой точке у вашего репозитария на одну фиксацию больше, чем в репозитарии в /tmp/Depot. Более интересно, что у вашего репозитария есть две ветки, одна (master) с новой фиксацией, а другая (origin/master) - ветка отслеживания удаленного репозитария.

Передача ваших изменений

Любое фиксируемое вами изменение абсолютно локально для вашего репозитария и не передается автоматически в удаленный репозитарий. Чтобы передать изменения ветки master в удаленный репозитарий, используется команда **git push** с параметром master:

```
$ git push origin master
Counting objects: 4, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 400 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /tmp/Depot/public_html
  0d4ce8a..6f16880 master -> master
```

Этот вывод Git означает, что изменения ветки master отправлены удаленному репозитарию с именем origin. Git также производит дополнительное действие: он берет эти же изменения и добавляет их в ветку origin/master в вашем репозитарии. Git достаточно умен, чтобы не загружать эти же изменения с удаленного репозитария, ведь, по сути, фиксации уже находятся в вашем репозитарии.

Теперь обе локальные ветки (master и origin/master) отображают одинаковые фиксации:

```
$ git show-branch -a
* [master] Add a hairy poem.
 ! [origin/master] Add a hairy poem.
--
*+ [master] Add a hairy poem.
```

Можно зондировать удаленный репозитарий и проверить, что он также был обновлен. Если удаленный репозитарий находится в локальной файловой системе, это легко сделать, перейдя в каталог базы:

```
$ cd /tmp/Depot/public_html.git
$ git show-branch
[master] Add a hairy poem.
```

Когда удаленный репозитарий физически находится на другой машине, следующая команда может быть использована для определения информации ветки удаленного репозитария:

```
# Переходим к действительно удаленному репозитарию и
опрашиваем его
$ git ls-remote origin
6f168803f6f1b987dffd5fff77531dcadf7f4b68 HEAD
6f168803f6f1b987dffd5fff77531dcadf7f4b68 refs/heads/master
```

Затем вы можете показать, что те ID фиксации соответствуют вашим текущим, локальным веткам с использованием чего-то подобного git rev-parse HEAD или git show id-фиксации.

Добавление нового разработчика

Как только вы установили авторитетный репозитарий, достаточно просто добавить нового разработчика в проект, разрешив ему клонировать репозитария и начинать работу.

Давайте добавим в проект пользователя Bob, предоставив ему его собственный клон репозитария, в котором он будет работать:

```
$ cd /tmp/bob
$ git clone /tmp/Depot/public_html.git
Cloning into 'public_html'...
done.
```

```
$ ls
public_html
$ cd public_html

$ ls
fuzzy.txt index.html poem.html techinfo.txt

$ git branch
* master

$ git log -1
commit 6f168803f6f1b987dffd5fff77531dcadf7f4b68
Author: Федя Колобков <kolobok@lobok.com>
Date: Sun Sep 14 21:04:44 2020 -0500

  Add a hairy poem.
```

Из вывода команды *ls* видно, что у клона заполнен рабочий каталог, в него помещены все файлы, которые находятся под управлением системы контроля версиями. Клон Бобика - это репозитарий разработки, а не чистый репозитарий. В этом репозитарии пользователь Бобик будет вести некоторую разработку.

Из вывода команды **git log** видно, что в репозитарии Бобика доступна самая новая фиксация. Кроме того, репозитарий Бобика был клонирован из родительского каталога и у него есть удаленная с именем origin. Получить больше информации об этой удаленной можно, выполнив следующую команду внутри репозитария Бобика:

```
$ git remote show origin
* remote origin
  URL: /tmp/Depot/public_html.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branch
    master
```

Рассмотрим полное содержимое конфигурационного файла после клонирования, обратите внимание, как представлена удаленная origin:

```
$ cat .git/config
[core]
      repositoryformatversion = 0
      filemode = true
      bare = false
      logallrefupdates = true
[remote "origin"]
      url = /tmp/Depot/public_html.git
      fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
      remote = origin
      merge = refs/heads/master
```

В дополнение к удаленной origin в репозитарии Бобика есть несколько веток. Просмотреть список веток в своем репозитарии он может с помощью команды git branch -a:

```
$ git branch -a
* master
  origin/HEAD
  origin/master
```

Ветка master - это главная ветка разработки Бобика. Это обычная локальная ветка разработки. Есть также локальная ветка отслеживания, ассоциированная с соответствующей удаленной веткой отслеживания master. Ветка origin/master - это удаленная ветка отслеживания, отслеживающая фиксации из ветки master репозитария origin. Ссылка origin/HEAD указывает, какая удаленная рассматривается как активная ветка. Звездочка рядом с веткой master говорит о том, что эта ветка является текущей.

Представим, что Бобик сделал фиксацию, которая изменяет поэму и затем отправляет (команда push) эти изменения в базу. Сначала мы просмотрим изменения, затем сделаем фиксацию:

```
$ git diff

diff --git a/fuzzy.txt b/fuzzy.txt
index 0d601fa..608ab5b 100644
--- a/fuzzy.txt
+++ b/fuzzy.txt
@@ -1,4 +1,4 @@
 Fuzzy Wuzzy was a bear
 Fuzzy Wuzzy had no hair
 Fuzzy Wuzzy wasn't very fuzzy,
-Was he?
+Wuzzy?

$ git commit fuzzy.txt
Created commit 3958f68: Make the name pun complete!
1 files changed, 1 insertions(+), 1 deletions(-)
```

Чтобы завершить цикл разработки Бобика, нужно отправить его изменения в базу, используя команду **git push**:

```
$ git push
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 377 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /tmp/Depot/public_html.git
   6f16880..3958f68 master -> master
```

Получение обновлений репозитария

Давайте предположим, что Бобик ушел в отпуск, а тем временем вы внесли свои изменения и отправили их в базу. Давайте разберемся, что нужно сделать Бобику, чтобы получить последние изменения.

Ваша фиксация выглядит так:

```
$ cd ~/public_html
$ git diff
diff --git a/index.html b/index.html
index 40b00ff..063ac92 100644
--- a/index.html
+++ b/index.html
@@ -1,5 +1,7 @@
  <html>
  <body>
  My web site is alive!
+ <br/>
+ Read a <a href="fuzzy.txt">hairy</a> poem!
  </body>
  <html>

$ git commit -m "Add a hairy poem link." index.html
Created commit 55c15c8: Add a hairy poem link.
1 files changed, 2 insertions(+), 0 deletions(-)
```

Используя команду git push, поместите вашу фиксацию в репозитарий:

```
$ git push
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
Writing objects: 100% (3/3), 348 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To /tmp/Depot/public_html
3958f68..55c15c8 master -> master
```

Теперь, когда Бобик вернулся из отпуска, он хочет обновить свой клон репозитария. Основная команда для загрузки обновлений - git pull:

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
```

```
Unpacking objects: 100% (3/3), done.
From /tmp/Depot/public_html
   3958f68..55c15c8 master -> origin/master
Updating 3958f68..55c15c8
Fast forward
 index.html | 2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)
```

Полный синтаксис команды git pull выглядит так: git pull репозитарий refspec.

Если репозитарий не задан в командной строке или как Git URL, по умолчанию используется удаленная origin. Если вы не укажите refspec в командной строке, будет использоваться refspec удаленной. Если вы укажите репозитарий (непосредственно или с и с использованием удаленной), но не укажите refspec, Git получит ссылку HEAD удаленной.

Операция git pull состоит из двух шагов. Фактически - это команда git fetch, за которой следует команда git merge или git rebase. По умолчанию, второй шаг - это merge, но практически всегда это требуемое поведение.

Поскольку pull всегда производит на втором шаге команду merge или step, команды git push и git pull не считают противоположностями. Вместо них, противоположными считаются команды git push и git fetch. Но push и fetch являются ответственными за передачу данных между репозитариями, но в противоположных направлениях.

Иногда вы можете захотеть выполнить git fetch и git merge как две отдельные операции. Например, вы можете получить обновления в ваш репозитарий для того, чтобы исследовать их, но не обязательно сразу же выполнять слияние. Вы можете сначала исследовать полученные изменения (с помощью команд git log, git diff или даже gitk), а затем выполнить слияние, как будете готовы.

Даже если вы никогда не будете разделять операции загрузки обновлений и обновления, вы должны знать, что происходит при их вызове. Давайте посмотрим на каждую из них в деталях.

Шаг fetch: получение обновлений

На первом шаге (операция fetch) Git определяет местоположение удаленного репозитария. Если имя удаленного репозитария не задано в командной строке (как URL или как имя удаленной), подразумевается имя удаленной по умолчанию, то есть origin. Информация об этой удаленной есть в конфигурационном файле:

```
[remote "origin"]
url = /tmp/Depot/public_html.git
fetch = +refs/heads/*:refs/remotes/origin/*
```

Git теперь знает, что нужно использовать URL /tmp/Depot/public_html в качестве исходного репозитария. Поскольку командная строка не содержит refspec, Git будет использовать вместо нее строку fetch из записи remote. В данном случае будут получены все ветки refs/heads/* из удаленной.

Далее Git выполняет протокол согласования с исходным репозитарием, чтобы определить, какие новые фиксации находятся в удаленном репозитарии и отсутствуют в вашем репозитарии (при этом используется refspec или строка fetch из файла конфигурации).

Примечание. Если вам не нужно получать все ветки (как в нашем случае refs/heads/*), вы не обязаны использовать звездочку. Вместо этого вы можете указать определенную ветку (или две):

```
[remote "newdev"]
   url = /tmp/Depot/public_html.git
   fetch = +refs/heads/dev:refs/remotes/origin/dev
   fetch = +refs/heads/stable:refs/remotes/origin/
stable
```

Вывод pull, снабженный префиксом remote отражает информацию о согласовании, сжатии и протоколе передачи. Вы узнаете, каким способом фиксации придут в ваш репозитарий:

```
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
```

```
remote: Total 3 (delta 1), reused 0 (delta 0)
```

Git помещает новые фиксации в ваш репозитарий в соответствующую ветку отслеживания и затем сообщает вам, где искать новые фиксации:

```
From /tmp/Depot/public_html
   3958f68..55c15c8 master -> origin/master
```

Эти строки сообщают, что Git соединяется с удаленным репозитарием /tmp/Depot/public_html, «берет» его ветку master и помещает ее содержимое обратно в ваш репозитарий, в *вашу* ветку origin/master. Этот процесс - это сердце отслеживания. На этом шаг fetch заканчивается.

Шаг merge (rebase): слияние

На втором шаге операции pull, Git выполняет слияние - операцию merge (по умолчанию) или rebase. В этом примере Git объединяет содержимое удаленной ветки отслеживания origin/master в вашу локальную ветку отслеживания master, используя специальный тип слияния - перемотку (fast-forward).

Но как Git узнал, что нужно объединить именно эти две определенные ветки? Ответ приходит из конфигурационного файла:

```
[branch "master"]
      remote = origin
      merge = refs/heads/master
```

Этот раздел содержит параметр для ветки master. Когда ветка master является текущей, удаленная origin используется по умолчанию для получения обновлений (командами fetch или pull). Во время шага merge команды **git pull** используется refs/heads/master удаленной как ветка по умолчанию для слияния с веткой master.

Значение поля merge в разделе branch файла конфигурации (refs/heads/master) обрабатывается как удаленная часть refspec и она должна соответствовать одной из ссылок источника, выбранной во время работы команды

git pull. Это немного замысловато, но думайте о ней как о подсказке, передаваемой от шага fetch шагу merge.

Поскольку значение конфигурации merge применяется только во время git pull, оно не распространяется на команду git merge, введенную вручную. В этом случае вам нужно указать имя ветки явно:

```
# Или полное имя refs/remotes/origin/master
$ git merge origin/master
Updating 3958f68..55c15c8
Fast forward
  index.html | 2 ++
  1 files changed, 2 insertions(+), 0 deletions(-)
```

Примечание. Есть небольшие семантические различия между поведением слияния, когда задано несколько refspec в командной строке или когда они найдены в записи remote. В первом случае выполняет слияние осьминога, где все ветки объединяются одновременно в n-стороннюю операцию, что не происходит во втором случае. Внимательно прочитайте страницу руководства команды git pull!

Если вы выбрали операцию rebase вместо merge, Git выполнит портирование изменений в локальную ветку отслеживания только что полученной HEAD соответствующей удаленной ветки отслеживания. Данная операция отображена на рис. 10.12 и 10.13 в главе 10.

Команда git pull --rebase заставит Git выполнить перебазирование (вместо слияния) вашей локальной ветки отслеживания в удаленную ветку отслеживания. Чтобы выполнить обычную операцию rebase для ветки, установите переменную конфигурации branch.имя_ветки.rebase в true:

```
[branch "mydev"]
      remote = origin
      merge = refs/heads/master
      rebase = true
```

Нужно выполнять слияние или перебазирование?

Какую операцию (слияние или перебазирование) выполнить во время команды pull? Короткий ответ: «Ту, которую хотите». Но почему выбираете ту или иную операцию? Давайте рассмотрим некоторые нюансы.

При использовании слияния вы потенциально подвергнетесь дополнительной фиксации слияния при каждом получении (pull) для записи обновленных изменений одновременно существующие в каждой ветке. В некотором смысле это - истинное отражение двух путей разработки, которые до этого были независимы, а потом объединены вместе. Во время слияния должны быть разрешены конфликты. Каждая последовательность фиксаций на каждой ветке будет основываться точно на фиксации, на которой она была исходно записана. Некоторые считают эти слияния лишними и что они загромождают историю. Другие считают эти слияния более точным отображением истории разработки и хотят видеть их сохраненными.

Поскольку перебазирование существенно изменяет понятие того, когда и где была разработана последовательность фиксаций, некоторые аспекты истории разработки будут потеряны. В частности исходная фиксация, на которой первоначально была основана ваша разработка, будет изменена, чтобы быть только что извлеченной HEAD удаленной ветки отслеживания. Естественно, вы должны будете разрешить конфликты во время перебазирования (по мере необходимости). Поскольку перебазируемые изменения все еще строго локальны в вашем репозитарии и еще не были опубликованы, нет никакой причины бояться изменять историю изменений этим перебазированием.

И со слиянием, и с перебазированием вы должны полагать, что новое, заключительное содержание отличается от того, что присутствовало на любой ветки разработки. Также желательно провести некоторую форму проверки допустимости: компиляция и цикл тестов должны предшествовать помещению в удаленный репозитарий.

Я склонен любить видеть более простые, линейные истории. Если же вы хотите установить один непротиворечивый подход, рассмотрим копии конфигурации branch.autosetupmerge или branch.autosetuprebase в true, false или always, в зависимости от желаемого эффекта. Кроме этих опций есть еще опции, способные влиять на поведением между полностью локальными ветками и между локальной и удаленной ветками.

12.4. Удаленная конфигурация

Отслеживание всей информации об удаленной ссылке репозитария вручную может стать утомительным и трудным: вы должны помнить полный URL для репозитария; вы должны вводить удаленные ссылки и refspec в командной строке каждый раз, когда вы хотите получить обновления; вы должны восстанавливать отображения веток и т.д. Весь этот процесс, вероятно, будет довольно подвержен ошибкам.

Как же Git помнит URL для использования в последующих операциях fetch или push с использованием origin?

Git предоставляет три механизма для установки и поддерживания информации об удаленных: команда **git remote**, команда **git config** и редактирование файла **.git/config**. В результате всех трех способов будет изменен файл конфигурации .git/config.

Использование команды git remote

Команда git remote обладает более специализированным интерфейсом, специфичным для удаленных, который манипулирует данными конфигурационного файла и ссылками удаленных. У нее есть несколько подкоманд с довольно интуитивными именами. Никакой справки нет, но вы можете использовать «прием неизвестной подкоманды»: когда вы передаете неизвестную подкоманду, то видите информацию об использовании команды **git remote**:

```
$ git remote xyzzy
error: Unknown subcommand: xyzzy
usage: git remote
or: git remote add <name> <url>
or: git remote rm <name>
```

```
or: git remote show <name>
or: git remote prune <name>
or: git remote update [group]
-v, --verbose be verbose
```

С командами **git remote add**, **show** и **update** мы уже знакомы (они были рассмотрены ранее в этой главе). Команда git remote add origin используется для добавления новой удаленной с именем origin в недавно созданный репозитарий в базе. Команда **git remote show origin** показывает всю информацию об удаленной origin. Наконец, команда **git remote update** используется для получения всех обновлений, доступных в удаленном репозитарии в ваш локальный репозитарий.

Команда **git remote rm** удаляет заданную удаленную и все связанные с ней удаленные ветки отслеживания из вашего локального репозитария. Чтобы удалить только какую-то одну удаленную ветку отслеживания, используйте команду вроде этой:

```
$ git branch -r -d origin/dev
```

Но вы не должны делать это, если соответствующая удаленная ветка действительно не была удалена из исходного репозитария. Иначе при следующей выборке (fetch) из исходного репозитария эта ветка будет снова воссоздана.

В удаленном репозитарии могут быть ветки, удаленные из него другими разработчиками, однако копии этих веток могут до сих пор храниться в вашем репозитарии.

Команда **git remote prune** может быть использована для удаления имен устарелых (по отношению к удаленному репозитарию) удаленных веток отслеживания из вашего локального репозитария.

Для еще лучшей синхронизации с удаленным репозитарием, используйте команду git `remote update --prune удаленная` для получения обновлений из удаленного репозитария и удаления устаревших веток отслеживания за один шаг.

Чтобы переименовать удаленную и все ее ссылки, используйте команду git remote rename старое_имя новое_имя. После выполнения этой команды:

```
$ git remote rename jon jdl
```

Любая ссылка вроде jon/bugfixes будет переименована в jdl/bugfixes.

В дополнение к манипуляциям с удаленной и ее ссылками, вы можете также обновлять или изменять URL удаленной:

```
$ git remote set-url origin git://repos.example.com/stuff.git
```

Использование git config

Команда **git config** может быть использована для манипуляции записями в вашем файле конфигурации. В нем есть несколько переменных конфигурации, настраивающая удаленные.

Например, чтобы добавить новую удаленную с именем publish в refspec-ссылкой для всех веток, которые вы хотите опубликовать, нужно ввести что-то вроде:

```
$ git config remote.publish.url 'ssh://git.example.org/pub/
repo.git'
$ git config remote.publish.push '+refs/heads/*:refs/heads/*'
```

Каждая из предыдущих команд добавляет строку в файл .git/config. Если remote-раздел publish не существует, то первая команда, которая к нему относится, создаст этот раздел в вашем файле конфигурации. В результате ваш .git/config будет содержать примерно следующее определение:

```
[remote «publish»]
url = ssh://git.example.org/pub/repo.git
push = +refs/heads/*:refs/heads/*
```

Примечание. Используйте опцию -l (-L в нижнем регистре, не -i) в команде git config -l для вывода содержимого конфигурационного файла с полными именами переменных.

```
# Из клона репозитария git.git
$ git config -l
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.url=git://git.kernel.org/pub/scm/git/
git.git
remote.origin.fetch=+refs/heads/*:refs/remotes/
origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
```

Редактирование файла конфигурации вручную

Вместо использования команд **git remote** или **git config** вы можете непосредственно редактировать файл конфигурации в вашем любимом текстовом редакторе. В некоторых случаях это может быть проще и быстрее. Однако будьте осторожны, чтобы не внести в файл конфигурации ошибки. Если вы не знакомы с поведением Git и его конфигурационным файлом, лучше не редактировать его вручную.

Несколько удаленных репозитариев

Операции вроде git remote add URL_удаленного_репозитария можно выполнять много раз, чтобы добавить несколько удаленных в ваш репозитарий. С несколькими удаленными репозитариями вы можете одновременно получать фиксации из нескольких источников и комбинировать их в своем репозитарии. Эта функция также позволяет установить несколько push-назначений, которые могут принимать часть вашего репозитария или весь репозитарий сразу.

12.5. Работа с ветками отслеживания

Поскольку создание и управление ветками отслеживания - это жизненно важная часть методики разработки, очень важно понимать, как и почему Git создает разные ветки отслеживания и каких действий над ними Git ожидает от вас.

Создание веток отслеживания

Вы можете создать новую ветку на основании любой удаленной веткой отслеживания и использовать ее для расширения линии разработки.

Мы уже видели, что удаленные ветки предоставлены во время операции клонирования или при добавлении удаленных в репозитарий. В более поздних версиях Git (начиная с 1.6.6) создать пару локальная-удаленная ветка очень просто: нужно использовать непротиворечивое имя для них. Простой запрос переключения ветки с использованием имени удаленной ветки создает новую локальную ветку отслеживания и связывает ее с соответствующей удаленной веткой. Однако, Git делает это, только если имя вашей ветки совпадает с одной из имен удаленных веток из списка всех удаленных репозитария.

Далее мы будем использовать репозитарий Git для демонстрации некоторых примеров. Путем загрузки из GitHub и git.kernel.org мы создаем репозитарий, содержащий коллекцию имен веток, состоящую из двух удаленных, некоторые из них являются копиями.

```
# Получаем репозитарий GitHub
$ git clone git://github.com/gitster/git.git
Cloning into 'git'...
...
$ git remote add korg git://git.kernel.org/pub/scm/git/git.git
```

```
$ git remote update
Fetching origin
Fetching korg
remote: Counting objects: 3541, done.
remote: Compressing objects: 100% (1655/1655), done.
remote: Total 3541 (delta 1796), reused 3451 (delta 1747)
Receiving objects: 100% (3541/3541), 1.73 MiB | 344 KiB/s,
done.
Resolving deltas: 100% (1796/1796), done.
From git://git.kernel.org/pub/scm/git/git
* [new branch] maint -> korg/maint
* [new branch] master -> korg/master
* [new branch] next -> korg/next
* [new branch] pu -> korg/pu
* [new branch] todo -> korg/todo

# Находим уникальное имя ветки и переключаемся на нее

$ git branch -a | grep split-blob
remotes/origin/jc/split-blob

$ git branch
* master

$ git checkout jc/split-blob
Branch jc/split-blob set up to track remote branch jc/split-
blob from origin.
Switched to a new branch 'jc/split-blob'

$ git branch
* jc/split-blob
master
```

Заметьте, что мы использовали полное имя ветки - js/split-blob, а не просто split-blob. В этом случае, когда имя ветки неоднозначно, вы можете установить и настроить ветку непосредственно:

```
$ git branch -a | egrep 'maint$'
remotes/korg/maint
remotes/origin/maint

$ git checkout maint
error: pathspec 'maint' did not match any file(s) known to git.

# Просто выберите одну из maint-веток
$ git checkout --track korg/maint
Branch maint set up to track remote branch maint from korg.
Switched to a new branch 'maint'
```

Вероятно, что две ветки представляют ту же фиксацию, как найдено в двух разных репозитариях и вы можете просто выбрать одну из них, на которой можно базировать вашу локальную ветку.

Если по некоторым причинам вы хотите использовать другое имя для вашей локальной ветки, используйте опцию **-b**:

```
$ git checkout -b mypu --track korg/pu
Branch mypu set up to track remote branch pu from korg.
Switched to a new branch 'mypu'
```

Git автоматически добавляет запись branch в файл .git/config, чтобы указать, что удаленная ветка должна быть объединена с вашей новой локальной ветки. Собранные изменения из предыдущего ряда команд приводят к следующему файлу:

```
$ cat .git/config
[core]
      repositoryformatversion = 0
      filemode = true
      bare = false
      logallrefupdates = true
[remote «origin»]
      fetch = +refs/heads/*:refs/remotes/origin/*
      url = git://github.com/gitster/git.git
```

```
[branch "master"]
      remote = origin
      merge = refs/heads/master
[remote «korg»]
      url = git://git.kernel.org/pub/scm/git/git.git
      fetch = +refs/heads/*:refs/remotes/korg/*
[branch «jc/split-blob»]
      remote = origin
      merge = refs/heads/jc/split-blob
[branch «maint»]
      remote = korg
      merge = refs/heads/maint
[branch «mypu»]
      remote = korg
      merge = refs/heads/pu
```

Как обычно, вы можете использовать git config или текстовый редактор для изменения вашего конфигурационного файла.

> **Примечание. Если вы заблудитесь в своих ветках, используйте команду git remote show удаленная, чтобы отсортировать ваши удаленные и ветки.**

В этой точке должно быть точно понятно, что по умолчанию в клоне будет локальная ветка отслеживания master для удаленной ветки origin/master.

Чтобы укрепить идею, что создание фиксаций непосредственно на удаленной ветки не является хорошей мыслью, переключение удаленной ветки с использованием ранних версий Git (до версии 1.6.6) приводит к отсоединению HEAD. Как было упомянуто в главе 7, отсоединенная HEAD - по сути, является анонимной веткой. Создание фиксаций на отсоединенной HEAD возможно, но вы не должны потом обновлять вашу HEAD удаленной ветки никакими локальными фиксациями, чтобы не перенести проблемы позже при получении новых обновлений из той удаленной. Если вам нужно сохранить любые фиксации на отсоединенной HEAD, используйте команду git checkout -b my_branch для создания новой, локальной ветки, на которой вы продолжите свою разработку. А вообще это не очень хороший подход.

Если вы не хотите переключаться на локальную ветку после ее создания, вы можете использовать команду git branch --track локальная-ветка удаленная-ветка для создания локальной ветки и записи ассоциации локальная-удаленная ветка в файл .git/config, например:

```
$ git branch --track dev origin/dev
Branch dev set up to track remote branch dev from origin.
```

Если у вас уже есть ветка и вы решили связать его с удаленной веткой удаленного репозитария, вы можете установить это отношение, используя опцией --set-upstream. Обычно это делается после добавления новой удаленной:

```
$ git remote add upstreamrepo git://git.example.org/
upstreamrepo.git
# Ветка mydev уже существует.
# Оставьте ее в покое, но свяжите с upstreamrepo/dev.
$ git branch --set-upstream mydev upstreamrepo/dev
```

Впереди и сзади

После установки пары локальная-удаленная ветка, между этими ветками могут быть произведены относительные сравнения. В дополнение к обычному diff, log и другим контентно-зависимым сравнениям, Git предоставляет быструю сводку числа фиксаций на каждой из веток и состояний веток, способных определить, где находится ветка - «позади» или «сзади» относительно другой ветки.

Если ваша локальная разработка представляет новые фиксации на локальной ветки, считается, что она *«впереди»* соответствующей удаленной ветки. Аналогично, если вы получили новые фиксации в удаленную ветку и они отсутствуют в вашей локальной ветке, Git рассматривает вашу локальную ветку как находящуюся за (*«сзади»*) соответствующей удаленной веткой.

Команда **git status** обычно показывает этот статус:

```
$ git fetch
```

```
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 6 (delta 4), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From example.com:SomeRepo
b1a68a8..b722324 ver2 -> origin/ver2
```

```
$ git status
# On branch ver2
# Your branch is behind 'origin/ver2' by 2 commits, and can be
fast-forwarded.
```

Чтобы увидеть, какие фиксации у вас есть в ветке master, но отсутствуют в ветке origin/master, выполните команду:

```
$ git log origin/master..master
Yes, it is possible to be both ahead and behind
simultaneously!
# Make one local commit on top of previous example
$ git commit -m «Something» main.c
...
$ git status
# On branch ver2
# Your branch and 'origin/ver2' have diverged,
# and have 1 and 2 different commit(s) each, respectively.
```

И в этом случае вы, возможно, пожелаете использовать симметрическую разницу для просмотра изменений:

```
$ git log origin/master...master
```

12.6. Добавление и удаление удаленных веток

Любая новая разработка, для которой вы создаете новую ветку в вашем локальном клоне, не видима в родительском репозитарии до тех пор, пока вы явно не сообщите об этом. Точно так же удаление ветки в вашем репозитарии остается локальным изменением и эта ветка не удаляется из родительского репозитария, пока вы явно не запросите это.

В главе 7 вы узнали, как добавить новые ветки и удалить существующие с помощью команды **git branch**. Но команда git branch работает только с локальным репозитарием.

Чтобы произвести аналогичные операции над ветками удаленного репозитария, вам нужно указать различные формы refspec в команде **git push**. Вспомним синтаксис refspec:

```
[+]источник:назначение
```

Отправка ваших изменений (push) использует refspec, в котором указывается только ссылка источник (ссылка назначения не нужна), создает новую ветку в удаленном репозитарии:

```
$ cd ~/public_html

$ git checkout -b foo
Switched to a new branch «foo»

$ git push origin foo
Total 0 (delta 0), reused 0 (delta 0)
To /tmp/Depot/public_html
* [new branch] foo -> foo
```

Отправка (push) данных, которая содержит только ссылку `источник`, является просто сокращением для того, чтобы использовать то же имя и для

источника и для назначения. Отправка, содержащая и источник, и назначение, имена которых отличаются, может использоваться для создания нового назначения с именем *ветка* или расширения существующей целевой ветки контентом из локальной ветки.

Команда git push origin mystuff:dev отправляет локальную ветку mystuff в удаленный репозитарий и создает или расширяет ветку с именем dev. Из-за чрезвычайной гибкости Git, у всех следующих команд будет одинаковый эффект:

```
$ git push upstream new_dev
$ git push upstream new_dev:new_dev
$ git push upstream new_dev:refs/heads/new_dev
```

Обычно upstream - это ссылка на надлежащий удаленный репозитарий и обычно принимает значение origin.

Отправка (push), использующая только ссылку `назначение` (то есть когда ссылка `источник` не задана) приведет к удалению ссылки `назначение` из удаленного репозитария. Чтобы указать только `назначение`, но не указать источник, вы должны задать перед именем назначения двоеточие:

```
$ git push origin :foo
To /tmp/Depot/public_html
- [deleted] foo
```

Если форма `:ветка` вам не нравится, вы можете использовать эквивалентную форму:

```
$ git push origin --delete foo
```

А что насчет переименования удаленной ветки? К сожалению, здесь нет простого решения. Проще всего создать новую ветку в удаленном репозитарии с новым именем, а затем удалить старую:

```
# Создаем новое имя (new) на существующей фиксации (old)
$ git branch new origin/old
```

```
$ git push origin new
# Удаляем имя old
$ git push origin :old
```

Но это - просто и очевидно. А что насчет распределенного использования? Вы узнаете, у кого есть клон удаленного репозитария, который только что изменен? Если да, они должны выполнить операции **fetch** и **remote prune** для обновления своих репозитариев. Если нет, они столкнутся с проблемами. К сожалению, нет никакого реального способа для распределенного переименования ветки.

Могу только порекомендовать быть предельно осторожными, как и с перезаписью вашей истории.

12.7. Чистые (bare) репозитарии и команда git push

В следствие одноранговой семантики репозитариев Git у всех репозитариев одинаковая важность. Вы можете отправить изменения (и получить изменения, соответственно) в чистый репозитарий точно так же, как и в обычный репозитарий, поскольку между ними нет никакого фундаментального различия.

Такой симметричный дизайн критически важен для Git, но приводит к некоторому неожиданному поведению, если вы пытаетесь считать чистые репозитарии и репозитарии разработки равными.

Вспомните, что команда **git push** не проверяет файлы в получающем репозитарии. Она просто передает объекты из исходного репозитария в целевой репозитария и затем обновляет соответствующие ссылки на целевом репозитарии.

В чистом репозитарии это поведение - все, что можно ожидать, поскольку здесь нет никакого рабочего каталога, который мог бы быть обновлен загруженными файлами. Это хорошо. Однако в репозитарии разработки, ко-

торый является получателем операции push, такое поведение push может вызвать беспорядок.

Операция push может обновить состояние репозитария, включая фиксацию HEAD. Т.е. даже при том, что разработчик на удаленной стороне ничего не сделал, ссылки ветки и HEAD могли измениться и выйти из синхронизации с загруженными файлами и индексом.

Разработчик, активно работающий в репозитарии, который происходит асинхронная операция push, не видит push. Но последующая фиксация этого разработчика произойдет на неожиданной HEAD, создавая странную историю. Принудительная операция push потеряет добавленные другим разработчиком изменения. Разработчик в том репозитарии может оказаться неспособным согласовать его историю с удаленным репозитарием или загруженным клоном, поскольку больше нет перемотки вперед, которая должна быть. И он не узнает почему: репозитарий без всякого предупреждения изменил ее. Кошки и собаки будут жить вместе. Это плохо.

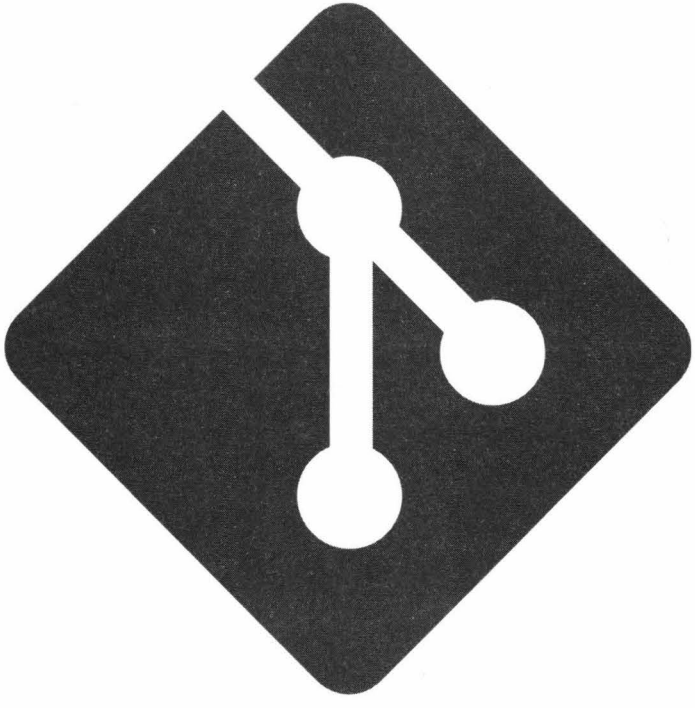
В результате вы будете стараться отправить свои изменения в чистый репозитарий. Это не обязательное правило, но хороший совет среднестатистическому разработчику и является отличной практикой. Есть несколько сценариев, в которых нужно отправить изменения в репозитария разработки, но вы должны точно отдавать себе отчет в том, что вы делаете. Если нужно таки отправить изменения в репозитарий разработки, вы можете использовать один из двух основных подходов.

В первом сценарии вы действительно хотите иметь рабочий каталог с активной веткой в целевом репозитарии. Например, вы знаете, что ни один другой разработчик не ведет в нем активную разработку и нет никого другого, кто может вслепую добавить изменения в репозитарий.

В этом случае вам нужно включить обработчик в целевом репозитарии, выполняющий проверку некоторой ветки. Чтобы проверить, что целевой репозитарий находится в нормальном состоянии, обработчик должен убедиться, что рабочий каталог репозитария разработки не содержит измененных файлов и что его индекс не содержит неподготовленных файлов. Когда эти два условия не соблюдены, вы рискуете потерять те изменения, поскольку они будут перезаписаны.

Есть и другой сценарий, при котором push в репозитарий разработки целесообразен. По соглашению каждый разработчик, который отправляет изме-

нения, должен отправить их в неактивную ветку, которая считаются просто веткой получения. Разработчик никогда не должен отправлять изменения в активную ветку. Также должен быть определенный разработчик, который должен переключать ветки. Возможно, этот человек будет ответственный за обработку веток получения и их слияние в основную ветку.



Издательство "Наука и Техника" рекомендует:



Пономарев С. Н.
Angular на примерах. Создаем web-приложения с нуля. — СПб.: "Наука и Техника" — 288 с., ил.

Angular – это передовой фреймворк по разработке одностраничных веб-приложений (англ. single page application, SPA). Одностраничность приложения позволяет изменять содержимое в окне браузера не путем открытия новых страниц с новым содержимым (как это реализуется на обычных сайтах (не SPA)), а путем обновления/замены содержимого одной и той же, первоначальной страницы. Короче говоря, одностраничное приложение — это веб-приложение или веб-сайт, использующий единственный HTML-документ как оболочку для всех веб-страниц и организующий взаимодействие с пользователем через динамически подгружаемые HTML, CSS, JavaScript.

С помощью практических примеров этой книги Вы сможете быстро освоить базовые навыки работы с Angular. По ходу прочтения вы создадите несколько готовых веб-приложений разной степени сложности.

Издательство "Наука и Техника" рекомендует:



Дьяков М. Ю.
Landing. Создание и продвижение продающих веб-страниц своими руками — СПб.: "Наука и Техника" — 256 с., ил.

Эта книга о том, что такое посадочная страница Landing Page и как знание правил создания посадочных страниц помогает продвигать товары и услуги на онлайн-рынке. В книге разобран простой пример создания посадочной страницы с нуля: объяснены основы до непосредственного внесения информации на страницу, алгоритм внесения информации на страницу и последующее продвижение, совершенствование и улучшение механизмов продажи через лендинг-страницы.

В книге содержатся все инструкции, необходимые для создания Вами минимально жизнеспособного лендинга/посадочной страницы, с настроенными процессами, как привлечения клиентов, удержания клиентов на лендинге/посадочной странице, так и обеспечения улучшения всех настроенных процессов

Издательство «Наука и Техника»

КНИГИ ПО КОМПЬЮТЕРНЫМ ТЕХНОЛОГИЯМ, МЕДИЦИНЕ, РАДИОЭЛЕКТРОНИКЕ

Уважаемые читатели!

Книги издательства «Наука и Техника» вы можете:

➢ **заказать в нашем интернет-магазине БЕЗ ПРЕДОПЛАТЫ по ОПТОВЫМ ценам**

www.nit.com.ru

- более 3000 пунктов выдачи на территории РФ, доставка 3—5 дней
- более 300 пунктов выдачи в Санкт-Петербурге и Москве, доставка — на следующий день

Справки и заказ:

- на сайте **www.nit.com.ru**
- по тел. (812) 412-70-26
- по эл. почте nitmail@nit.com.ru

➢ **приобрести в магазине издательства по адресу:**

Санкт-Петербург, пр. Обуховской обороны, д.107
М. Елизаровская, 200 м за ДК им. Крупской
Ежедневно с 10.00 до 18.30
Справки и заказ: тел. (812) 412-70-26

➢ **приобрести в Москве:**

«Новый книжный» Сеть магазинов ТД «БИБЛИО-ГЛОБУС»	тел. (495) 937-85-81, (499) 177-22-11 ул. Мясницкая, д. 6/3, стр. 1, ст. М «Лубянка» тел. (495) 781-19-00, 624-46-80
Московский Дом Книги, «ДК на Новом Арбате»	ул. Новый Арбат, 8, ст. М «Арбатская», тел. (495) 789-35-91
Московский Дом Книги, «Дом технической книги»	Ленинский пр., д.40, ст. М «Ленинский пр.», тел. (499) 137-60-19
Московский Дом Книги, «Дом медицинской книги»	Комсомольский пр., д. 25, ст. М «Фрунзенская», тел. (499) 245-39-27
Дом книги «Молодая гвардия»	ул. Б. Полянка, д. 28, стр. 1, ст. М «Полянка» тел. (499) 238-50-01

➢ **приобрести в Санкт-Петербурге:**

Санкт-Петербургский Дом Книги	Невский пр. 28, тел. (812) 448-23-57
Буквоед. Сеть магазинов	тел. (812) 601-0-601

➢ **приобрести в регионах России:**

г. Воронеж, «Амиталь» Сеть магазинов	тел. (473) 224-24-90
г. Екатеринбург, «Дом книги» Сеть магазинов	тел. (343) 289-40-45
г. Нижний Новгород, «Дом книги» Сеть магазинов	тел. (831) 246-22-92
г. Владивосток, «Дом книги» Сеть магазинов	тел. (423) 263-10-54
г. Иркутск, «Продалить» Сеть магазинов	тел. (395) 298-88-82
г. Омск, «Техническая книга» ул. Пушкина, д.101	тел. (381) 230-13-64

Мы рады сотрудничеству с Вами!

Фишерман Л.В.

Git

Практическое руководство

Управление и контроль версий в разработке программного обеспечения

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *М. В. Финков*

Редактор: *Е. В. Финков*

Корректор: *А. В. Громова*

12+

ООО «Наука и Техника»
Лицензия №000350 от 23 декабря 1999 года.
192029, г. Санкт-Петербург, пр.Обуховской обороны, д. 107.
Подписано в печать 23.10.2020. Формат 70x100 1/16.
Бумага офсетная. Печать офсетная. Объем 19 п. л.
Тираж 1350. Заказ 12311.

Отпечатано ООО «Принт-М»
142300, Московская область,
г. Чехов, ул. Полиграфистов, дом 1

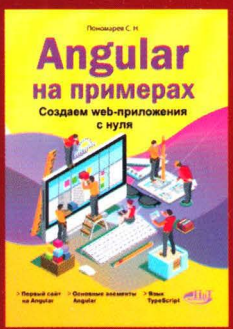
Фишерман Л. В.

Git ПРАКТИЧЕСКОЕ РУКОВОДСТВО

Управление и контроль версий в разработке программного обеспеченния

Git в настоящее время нужен практически всем программистам, которые занимаются разработкой программного обеспечения. Git — это система управления версиями, с помощью которой вы сможете вести и контролировать разработку нескольких версий одного приложения, осуществлять совместную разработку одного приложения несколькими разработчиками (учитывать изменения, которые делаются на том или ином шаге разработки тем или иным разработчиком). С помощью системы Git у вас будет полная иерархия всех версий программного кода разрабатываемого приложения.

Данная книга представляет собой подробное практическое руководство по Git, в котором описывается Git и приводится разбор конкретных ситуаций и применений, например, как изменения из одной ветки разработки включить в другую ветку, но не все. Изложение начинается с самых азов, никакой предварительной подготовки не требуется: по ходу изложения даются все необходимые определения и пояснения. Лучший выбор, чтобы освоить Git и максимально быстро начать его применять на практике.







Издательство "Наука и Техника" рекомендует

ISBN 978-5-94387-547-2

9 78- 5- 94387- 547- 2

Издательство "Наука и Техника"
г. Санкт-Петербург

Для заказа книг:
(812) 412-70-26
e-mail: nitmail@nit.com.ru
www.nit.com.ru

НиТ издательство