

THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

*EVERYTHING YOU NEED TO
KNOW ABOUT GIT*

Scott Chacon and Ben Straub

Apress®

Это произведение распространяется по свободной лицензии Creative Commons Attribution-NonCommercial-ShareAlike 3.0. Ознакомиться с текстом лицензии вы можете на сайте <http://creativecommons.org/licenses/by-nc-sa/3.0/deed.ru> или по почте, отправив письмо в организацию Creative Commons по адресу: PO Box 1866, Mountain View, CA 94042, USA.

Предисловие

Добро пожаловать во второе издание Pro Git. Первое издание было опубликовано более четырех лет назад. С тех пор многое изменилось, но многие важные вещи остались неизменны. Хотя большинство ключевых команд и концепций по-прежнему работают, так как команда, разрабатывающая ядро Git, фантастическим образом оставляет всё обратно совместимым, произошло несколько существенных дополнений и изменений в сообществе вокруг Git. Второе издание призвано обозначить эти изменения и обновить книгу для помощи новичкам.

Когда я писал первое издание, Git ещё был относительно сложным в использовании и подходил лишь для настоящих хакеров. И хотя в некоторых сообществах он уже начинал набирать обороты, ему было далеко до сегодняшней распространённости. С тех пор его приняло практически всё сообщество свободного программного обеспечения. Git достиг невероятного прогресса в Windows, взрывными темпами получил графический интерфейс для всех платформ, поддержку сред разработки и стал использоваться в бизнесе. Pro Git четырехлетней давности ничего подобного не подозревал. Одна из главных целей издания — затронуть в Git сообществе эти рубежи.

Сообщество свободного программного обеспечения тоже испытalo взрывной рост. Когда я лет пять назад впервые сел писать книгу (первая версия потребовала времени), я как раз начал работать в крохотной компании, разрабатывающей сайт для Git хостинга под названием Гитхаб. На момент публикации у сайта было лишь несколько тысяч пользователей и четверо разработчиков. Когда же я пишу это предисловие, Гитхаб объявляет о десяти миллионах размещенных проектов, около пяти миллионах аккаунтах разработчиков и более 230 сотрудниках. Его можно любить или ненавидеть, в любом случае Гитхаб сильнейшим образом изменил сообщество свободного программного обеспечения, что было едва мыслимо, когда я только сел писать первое издание.

Небольшую часть исходной версии Pro Git я посвятил Гитхабу в качестве примера хостинга, с которым мне никогда не особо удобно

работать. Мне это не сильно нравится. То, что я писал, было, по сути, ресурсом сообщества и говорило о моей компании в нём. Я по-прежнему не люблю этот конфликт интересов, но важность Гитхаба в Git сообществе бесспорна. Вместо образца Git хостинга, я решил посвятить этот раздел книги детальному описанию сути Гитхаба и его эффективному использованию. Если вы собираетесь узнать, как пользоваться Git, знание о том, как пользоваться Гитхабом даст вам возможность поучаствовать в гигантском сообществе, ценном вне зависимости от выбранного вами Git хостинга.

Другой большой переменой с момента первой публикации стала разработка и развитие HTTP протокола для сетевых Git транзакций. Большинство примеров из книги были переделаны из SSH на HTTP, так гораздо проще.

Было изумительно смотреть, как за несколько прошедших лет Git вырос из весьма невзрачной системы контроля версий до безусловно лидирующей в коммерческой и некоммерческой сферах. Я счастлив, что Pro Git так хорошо выполнил свою работу, оказавшись одним из немногих представителей успешной и при этом полностью открытой технической литературы.

Я надеюсь, вам понравится это новое издание Pro Git.

Contributors

Since this is an Open Source book, we have gotten several errata and content changes donated over the years. Here are all the people who have contributed to the English version of Pro Git as an open source project. Thank you everyone for helping make this a better book for everyone.

- 2 Aaron Schumacher
- 4 Aggelos Orfanakos
- 4 Alec Clews
- 1 Alex Moundalexis
- 2 Alexander Harkness
- 1 Alexander Kahn
- 1 Andrew McCarthy
- 1 AntonioK
- 1 Benjamin Bergman
- 1 Brennon Bortz
- 2 Brian P O'Rourke
- 1 Bryan Goines
- 1 Cameron Wright
- 1 Chris Down
- 1 Christian Kluge
- 1 Christoph Korn
- 2 Ciro Santilli
- 2 Cor
- 1 Dan Croak
- 1 Dan Johnson
- 1 Daniel Kay
- 2 Daniel Rosen
- 1 DanielWeber
- 1 Dave Dash
- 10 Davide Fiorentino lo Regio
- 2 Dilip M
- 1 Dimitar Bonev
- 1 Emmanuel Trillaud
- 1 Eric-Paul Lecluse
- 1 Eugene Serkin
- 1 Fernando Dobladez
- 2 Gordon McCreight
- 1 Helmut K. C. Tessarek

31 Igor Murzov
1 Ilya Kuznetsov
1 Jason St. John
1 Jay Taggart
1 Jean Jordaan
51 Jean-Noël Avila
1 Jean-Noël Rouvignac
1 Jed Hartman
1 Jeffrey Forman
1 John DeStefano
1 Junior
1 Kieran Spear
1 Larry Shatzer, Jr
1 Linquize
1 Markus
7 Matt Deacalion Stevens
1 Matthew McCullough
1 Matthieu Moy
1 Max F. Albrecht
1 Michael Schneider
8 Mike D. Smith
1 Mike Limansky
1 Olivier Trichet
1 Ondrej Novy
6 Ori Avtalion
1 Paul Baumgart
1 Peter Vojtek
1 Philipp Kempgen
2 Philippe Lhoste
1 PowerKiKi
1 Radek Simko
1 Rasmus Abrahamsen
1 Reinhard Holler
1 Ross Light
1 Ryuichi Okumura
1 Sebastian Wiesinger
1 Severyn Kozak
1 Shane
2 Shannen
8 Sitaram Chamarty
5 Soon Van
4 Sven Axelsson
2 Tim Court
1 Tuomas Suutari
1 Vlad Gorodetsky
3 W. Trevor King
1 Wyatt Carss
1 Włodzimierz Gajda
1 Xue Fuqiao
1 Yue Lin Ho

2 adelcambre
1 anaran
1 bdukes
1 burningTyger
1 cor
1 iosias
7 nicesw123
1 onovy
2 pcasaretto
1 sampablokuper

Вступление

Вы собираетесь потратить несколько часов своей жизни, читая о Git. Давайте уделим минуту на объяснение, что же вы получите. Здесь представлено краткое описание десяти глав и трех приложений данной книги.

В **Главе 1** мы охватим Системы Контроля Версий (VCS) и азы Git. Никаких технических штучек, только то, что, собственно, такое Git, почему он пришел на землю уже полную систем контроля версий, что его отличает и почему так много людей им пользуются. Затем мы объясним как впервые скачать и настроить Git, если в вашей системе его ещё нет.

В **Главе 2** мы перейдём к основам использования Git — как использовать Git в 80% случаев с которыми вы столкнётесь. После прочтения этой главы вы сможете клонировать репозитории, смотреть изменения в истории проекта, изменять файлы и публиковать эти изменения. Если на этом моменте книга самопроизвольно воспламенится, вы уже достаточно оцените время, потраченное на знакомство с Git, чтобы сходить за ещё одной копией.

Глава 3 про модель ветвления в Git, часто описываемую как киллер-фичу Git. Отсюда вы узнаете, что на самом деле отличает Git от обычного пакета. Когда вы дочитаете, возможно, вам понадобится ещё немного времени на размышления о том, как же вы существовали до того как Git ветвление вошло в вашу жизнь.

Глава 4 опишет Git на сервере. Эта глава для тех из вас, кто хочет настроить Git внутри компании или на собственном сервере для совместной работы. Так же мы разберём различные настройки хостинга, если вы предпочитаете держать сервер у кого-нибудь другого.

В **Главе 5** мы детально рассмотрим всевозможные распределенные рабочие процессы и то, как совмещать их с Git. После этой главы вы будете мастерски справляться с множеством удаленных репозиториев, работать с Git через почту, ловко жонглировать несколькими удаленными ветвями и новыми патчами.

Глава 6 посвящена хостингу Гитхаба и его инструментам. Мы разберём регистрацию, управление учетной записью, создание и использование Git репозиториев, как вносить вклад в чужие проекты и как принимать чужой вклад в собственный проект, а так же программный интерфейс Гитхаба и ещё множество мелочей, который облегчат вам жизнь.

Глава 7 про дополнительные Git команды. Здесь раскроются темы освоения пугающей команды *reset*, использования бинарного поиска для нахождения багов, правки истории, инспекции кода и многие другие. На этой главе вы уже станете настоящим мастером Git.

Глава 8 о настройке собственного Git окружения, включая и перехватывающие скрипты, применяющие или поощряющие заданную политику, и использование специфических настроек окружения, чтобы вы могли работать так, как вам хочется. К тому же мы поговорим о собственных наборах скриптов, реализующих заданную вами политику в отношении коммитов.

Глава 9 разберется с Git и другими системами контроля версий, в том числе использование Git в мире системы контроля версий Subversion (SVN) и конвертацию проектов в Git из прочих систем. Многие организации всё ещё используют SVN и не собираются ничего менять, но к этому моменту вы познаете всю мощь Git и эта глава научит вас, что делать если вам по прежнему приходится пользоваться сервером SVN. Так же мы расскажем как импортировать проекты из нескольких прочих систем, если вы убедите всех приступить к решительным действиям.

Глава 10 углубляется в мрачные и прекрасные глубины внутренностей Git. Теперь, когда вы знаете всё о Git и грациозно с ним управляетесь, можно двигаться дальше и разобраться, как Git хранит свои объекты, что такое объектная модель, из чего состоят файлы пакетов, каковы серверные протоколы и многое другое. На протяжении всей книги мы будем давать отсылки к этой главе, на случай, если вам захочется углубиться в детали. Если же вам, как и нам, интереснее всего техническая реализация, то, возможно, вам захочется начать именно с десятой главы. Оставим это на ваше усмотрение.

В **Приложении А** мы рассмотрим примеры использования Git в различных окружениях, разберём варианты с разными средами разработки и интерфейсами, в которых вам может захотеться попробовать Git и в которых это вообще возможно. Загляните сюда, если вы заинтересованы в использовании Git в командной строке, Visual Studio или Eclipse.

В **Приложении В** мы изучим скрипты и расширения для Git с помощью libgit2 и JGit. Если вы заинтересованы в написании сложных и быстрых инструментов и вам нужен низкоуровневый доступ к Git, здесь описано как это выглядит.

Наконец, в **Приложении С** мы заново пройдемся через все основные команды Git и вспомним, где и для чего в книге мы их применяли. Если вы хотите узнать, где в книге используется конкретная Git команда, можете посмотреть здесь.

Начнём же.

Table of Contents

Предисловие	iii
Contributors	v
Вступление	ix
CHAPTER 1: Введение	25
О системе контроля версий	25
Локальные системы контроля версий	26
Централизованные системы контроля версий	27
Децентрализованные системы контроля версий	28
Краткая история Git	30
Основы Git	30
Снимки, а не различия	31
Почти все операции выполняются локально	32
Целостность Git	33
Git только добавляет данные	33
Три состояния	34
Командная строка	36
Установка Git	36
Установка в Linux	37
Установка на Mac	37
Установка в Windows	38

Установка из исходников	39
Первоначальная настройка Git	40
Имя пользователя	40
Выбор редактора	41
Проверка настроек	41
Как получить помощь?	42
Заключение	43
CHAPTER 2: Основы Git	45
Создание Git-репозитория	45
Создание репозитория в существующей директории	45
Клонирование существующего репозитория	46
Recording Changes to the Repository	47
Checking the Status of Your Files	48
Tracking New Files	49
Staging Modified Files	50
Short Status	51
Ignoring Files	52
Viewing Your Staged and Unstaged Changes	53
Committing Your Changes	56
Skipping the Staging Area	57
Removing Files	58
Moving Files	60
Просмотр истории коммитов	60
Ограничение вывода	66
Операции отмены	68
Отмена подготовки файла	69
Отмена изменения измененного файла	70
Working with Remotes	72
Showing Your Remotes	72
Adding Remote Repositories	73

Fetching and Pulling from Your Remotes	74
Pushing to Your Remotes	75
Inspecting a Remote	75
Removing and Renaming Remotes	76
Работа с метками	77
Просмотр меток	77
Создание меток	78
Аннотированные метки	78
Легковесные метки	79
Выставление меток позже	80
Обмен метками	81
Переход на метку	81
Псевдонимы в Git	82
Заключение	84
CHAPTER 3: Ветвление в Git	85
О ветвлении в двух словах	85
Создание новой ветки	88
Переключение веток	89
Основы ветвления и слияния	93
Основы ветвления	93
Основы слияния	98
Основные конфликты слияния	100
Управление ветками	103
Branching Workflows	105
Long-Running Branches	105
Topic Branches	106
Удалённые ветки	108
Отправка изменений	114
Отслеживание веток	116
Получение изменений	118

Удаление веток на удалённом сервере	119
Rebasing	119
The Basic Rebase	119
More Interesting Rebases	122
The Perils of Rebasing	125
Rebase When You Rebase	127
Rebase vs. Merge	129
Итоги	129
CHAPTER 4: Git на сервере	131
Протоколы	132
Локальный протокол	132
Протоколы HTTP	134
Протокол SSH	136
Git-протокол	137
Установка Git на сервер	138
Размещение голого репозитория на сервере	139
Малые установки	140
Генерация открытого SSH ключа	142
Настраиваем сервер	143
Git-демон	146
Умный HTTP	148
GitWeb	150
GitLab	152
Установка	152
Администрирование	153
Базовое использование	156
Совместная работа	156
Git-хостинг	157

Заключение	158
CHAPTER 5: Распределенный Git	159
Distributed Workflows	159
Centralized Workflow	159
Integration-Manager Workflow	160
Dictator and Lieutenants Workflow	161
Workflows Summary	162
Contributing to a Project	163
Commit Guidelines	164
Private Small Team	166
Private Managed Team	173
Forked Public Project	179
Public Project over E-Mail	183
Summary	186
Maintaining a Project	186
Working in Topic Branches	187
Applying Patches from E-mail	187
Checking Out Remote Branches	191
Determining What Is Introduced	192
Integrating Contributed Work	193
Tagging Your Releases	200
Generating a Build Number	201
Preparing a Release	202
The Shortlog	202
Заключение	203
CHAPTER 6: GitHub	205
Настройка и конфигурация учетной записи	205
Доступ по SSH	206
Ваш аватар	208

Ваши почтовые адреса	209
Двухфакторная аутентификация	210
Внесение собственного вклада в проекты	211
Создание ответвлений (fork)	212
The GitHub Flow	213
Advanced Pull Requests	221
Markdown	226
Maintaining a Project	231
Creating a New Repository	231
Adding Collaborators	233
Managing Pull Requests	235
Mentions and Notifications	240
Special Files	244
README	244
CONTRIBUTING	245
Project Administration	245
Managing an organization	247
Organization Basics	247
Teams	248
Audit Log	250
Scripting GitHub	251
Hooks	252
The GitHub API	256
Basic Usage	257
Commenting on an Issue	258
Changing the Status of a Pull Request	259
Octokit	261
Заключение	262
CHAPTER 7: Инструменты Git	263
Выбор ревизии	263

Одиночные ревизии	263
Сокращенный SHA-1	263
Ссылки на ветки	265
RefLog-сокращения	266
Ссылки на предков	268
Диапазоны фиксаций	270
Интерактивное индексирование	273
Добавление и удаление файлов из индекса	274
Индексирование по частям	276
Прибережение и очистка	278
Прибережение ваших наработок	278
Продуктивное прибережение	281
Создание ветки из спрятанных изменений	282
Очистка вашей рабочей директории	283
Подпись результатов вашей работы	285
Введение в GPG	285
Подпись тегов	286
Проверка тегов	287
Подпись коммитов	287
Каждый должен подписываться	289
Поиск	289
Git Grep	290
Поиск в журнале изменений Git	292
Исправление истории	293
Изменение последней фиксации	294
Изменение сообщений нескольких фиксаций	295
Переупорядочивание фиксаций	297
Объединение фиксаций	298
Разбиение фиксации	299
Продвинутый инструмент: filter-branch	300
Раскрытие тайн reset	303

Три дерева	303
Технологический процесс	305
Назначение reset	311
Reset с указанием пути	316
Слияние коммитов	319
Сравнение с checkout	322
Заключение	324
Продвинутое слияние	325
Конфликты слияния	326
Отмена слияний	338
Другие типы слияний	342
Rerere	347
Обнаружение ошибок с помощью Git	355
Аннотация файла	355
Бинарный поиск	357
Подмодули	359
Начало работы с подмодулями	360
Клонирование проекта с подмодулями	362
Работа над проектом с подмодулями	364
Полезные советы для работы с подмодулями	376
Проблемы с подмодулями	378
Создание пакетов	381
Замена	386
Хранилище учетных данных	394
Под капотом	396
Собственное хранилище учетных данных	399
Заключение	401
CHAPTER 8: Настройка Git	403
Git Configuration	403
Basic Client Configuration	404

Colors in Git	407
External Merge and Diff Tools	408
Formatting and Whitespace	412
Server Configuration	414
Git Attributes	415
Binary Files	416
Keyword Expansion	419
Exporting Your Repository	422
Merge Strategies	423
Git Hooks	424
Installing a Hook	424
Client-Side Hooks	424
Server-Side Hooks	426
An Example Git-Enforced Policy	427
Server-Side Hook	428
Client-Side Hooks	434
Заключение	437
CHAPTER 9: Git и другие системы контроля версий	439
Git как клиент	439
Git и Subversion	439
Git и Mercurial	453
Git и Perforce	462
Git и TFS	479
Миграция на Git	489
Subversion	490
Mercurial	492
Perforce	495
TFS	497
A Custom Importer	498

Заключение	506
CHAPTER 10: Git изнутри	507
Сантехника и Фарфор	508
Объекты Git	509
Деревья	511
Commit Objects	515
Хранение объектов	518
Ссылки в Git	520
HEAD	522
Метки	523
Ссылки на удалённые ветки	524
Pack-файлы	525
Спецификации ссылок	529
Спецификации ссылок для отправки данных на сервер	531
Удаление ссылок	531
Протоколы передачи данных	532
Глупый протокол	532
Умный протокол	535
Заключение	538
Уход за репозиторием и восстановление данных	538
Уход за репозиторием	539
Восстановление данных	540
Removing Objects	543
Переменные среды	547
Глобальное поведение	547
Расположение репозитория	548
Пути к файлам	549
Фиксация изменений	549
Работа с сетью	550
Сравнение файлов и слияния	550

Отладка	551
Разное	553
Заключение	554
Git в других окружениях	555
Встраивание Git'a в ваши приложения	571
Команды Git	583
Index	603

Введение

Эта глава о том, как начать работу с Git. Вначале изучим основы инструментария системы контроля версий, затем перейдём к тому, как запустить Git на вашей ОС и окончательно настроить для работы. В конце главы вы уже будете знать, что такое Git и почему им следует пользоваться, а также получите окончательно настроенную для работы систему.

О системе контроля версий

Что такое “система контроля версий”, и почему это важно? Система контроля версий — это система, записывающая изменения в файл или набор файлов в течение большого периода времени, так что вы сможете позже вернуться к определенной версии. Для контроля версий файлов в этой книге, в качестве примера, будет использоваться исходный код программного обеспечения, хотя на самом деле вы можете использовать контроль версий практически для любых типов файлов.

Если вы графический или web дизайнер и хотите сохранить каждую версию изображения или макета (скорее всего, захотите), система контроля версий (далее СКВ) как раз то, что нужно. Она позволяет вернуть файлы к состоянию, в котором они были до изменений, вернуть проект к исходному состоянию, увидеть изменения, увидеть, кто последний менял что-то и спровоцировал проблему, кто поставил задачу и когда, и многое другое. Использование VCS также значит в целом, что, если вы сломали что-то или потеряли файлы, вы спокойно можете всё исправить. В дополнение ко всему вы получите всё это без каких-либо накладок.

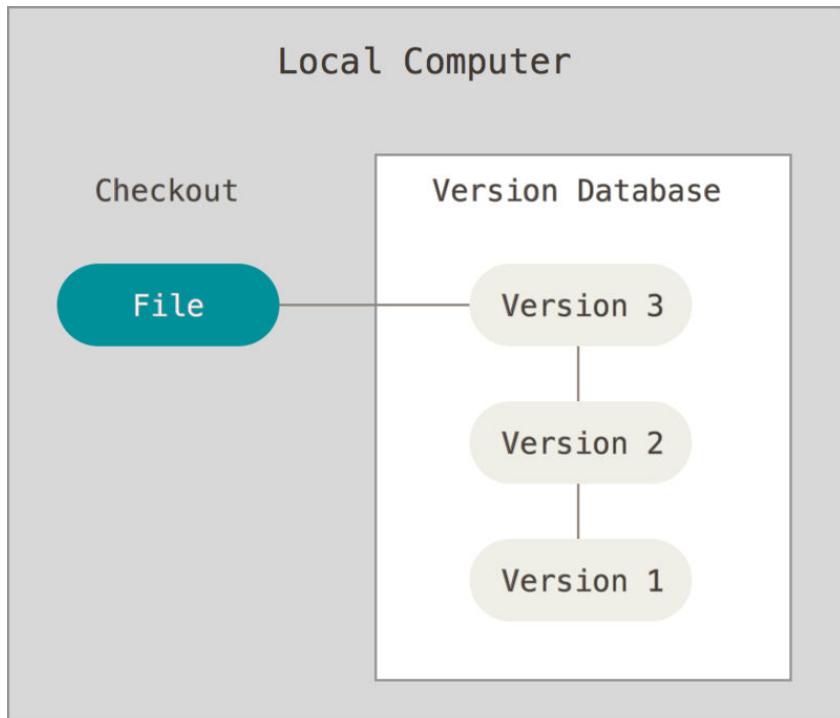
Локальные системы контроля версий

Многие люди в качестве метода контроля версий применяют копирование файлов в отдельную директорию (возможно даже директорию с отметкой по времени, если они достаточно умны). Данный подход очень распространён из-за его простоты, однако он, невероятным образом, подвержен появлению ошибок. Можно легко забыть в какой директории вы находитесь и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели.

Для того, чтобы решить эту проблему, программисты давным-давно разработали локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий.

FIGURE 1-1

Локальный
контроль версий.

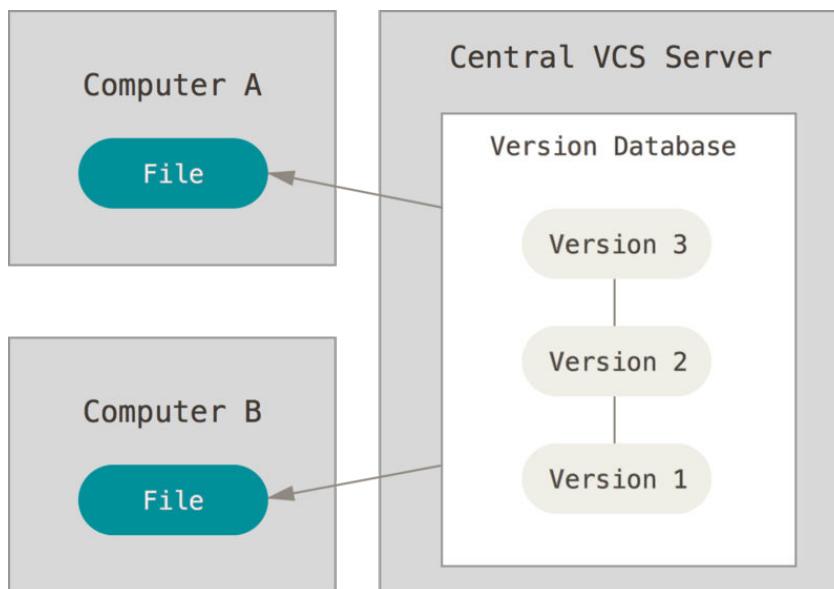


Одной из популярных СКВ была система RCS, которая и сегодня распространяется со многими компьютерами. Даже популярная операционная система Mac OS X предоставляет команду `rcs`, после установки Developer Tools. RCS хранит на диске наборы патчей

(различий между файлами) в специальном формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени.

Централизованные системы контроля версий

Следующая серьёзная проблема, с которой сталкиваются люди - это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как: CVS, Subversion и Perforce, имеют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Применение ЦСКВ являлось стандартом на протяжении многих лет.

**FIGURE 1-2**

Централизованный
контроль версий.

Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта, в определённой степени, знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще, администрировать ЦСКВ, чем оперировать локальными базами данных на каждом клиенте.

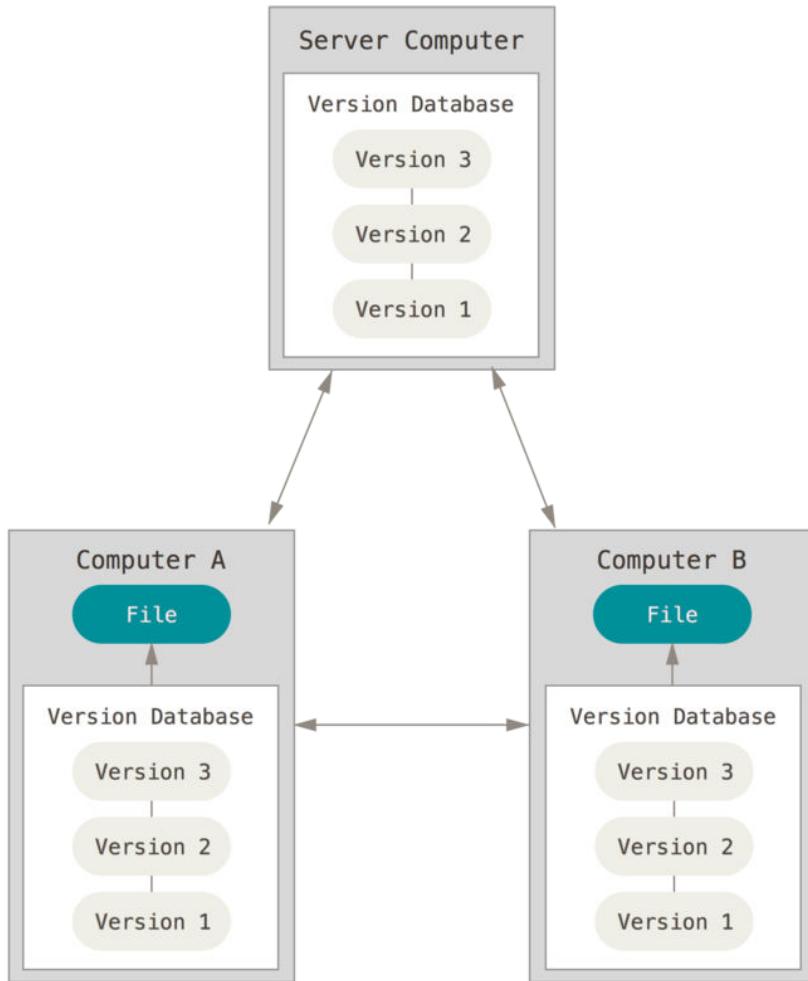
Несмотря на это, данный подход тоже имеет серьёзные минусы. Самый очевидный минус — это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений над которыми он работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы — когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

Децентрализованные системы контроля версий

Здесь в игру вступают децентрализованные системы контроля версий (ДСКВ). В ДСКВ (таких как Git, Mercurial, Bazaar или Darcs), клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени): они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.

FIGURE 1-3

*Децентрализованный
контроль версий.*



Более того, многие ДСКВ могут одновременно взаимодействовать с несколькими удалёнными репозиториями, благодаря этому вы можете работать с различными группами людей, применяя различные подходы единовременно, в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

Краткая история Git

Как и многие вещи в жизни, Git начинался с капелькой творческого хаоса и бурных споров.

Ядро Linux — это достаточно большой проект с открытым исходным кодом. Большую часть времени разработки ядра Linux (1991-2002 гг.), изменения передавались между разработчиками в виде патчей и архивов. В 2002 году проект ядра Linux начал использовать проприетарную децентрализованную СКВ BitKeeper.

В 2005 году отношения между сообществом разработчиков ядра Linux и коммерческой компанией, которая разрабатывала BitKeeper, прекратились, и бесплатное использование утилиты стало невозможным. Это сподвигло сообщество разработчиков ядра Linux (а в частности Линуса Торвальдса — создателя Linux) разработать свою собственную утилиту, учитывая уроки, полученные при работе с BitKeeper. Некоторыми целями, которые преследовала новая система, были:

- Скорость
- Простая архитектура
- Хорошая поддержка нелинейной разработки (тысячи параллельных веток)
- Полная децентрализация
- Возможность эффективного управления большими проектами, такими как ядро Linux (скорость работы и разумное использование дискового пространства)

С момента своего появления в 2005 году, Git развился в простую в использовании систему, сохранив при этом свои изначальные качества. Он удивительно быстр, эффективен в работе с большими проектами и имеет великолепную систему веток для нелинейной разработки (См. Chapter 3).

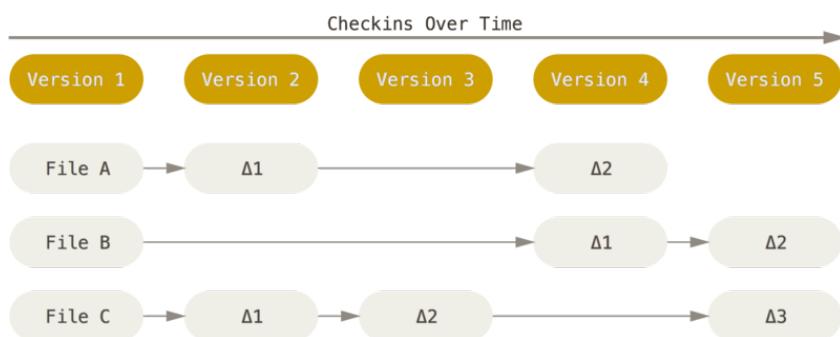
Основы Git

Что же такое Git, если говорить коротко? Очень важно понять эту часть материала, потому что если вы поймёте что такое Git и основы того, как он работает, тогда, возможно, вам будет гораздо проще его использовать. Пока вы изучаете Git, попробуйте забыть всё что вы знаете о других СКВ, таких как Subversion и Perforce; это позволит вам избежать определенных проблем при использовании утилиты. Git

хранит и использует информацию совсем иначе по сравнению с другими системами, даже несмотря на то, что интерфейс пользователя достаточно похож, и понимание этих различий поможет вам избежать путаницы во время использования.

Снимки, а не различия

Основное отличие Git'a от любой другой СКВ (Subversion и друзья включительно), это подход Git'a к работе со своими данными. Концептуально, большинство других систем хранят информацию в виде списка изменений в файлах. Эти системы (CVS, Subversion, Perforce, Bazaar и т.д.) представляют информацию в виде набора файлов и изменений, сделанных в каждом файле, по времени.

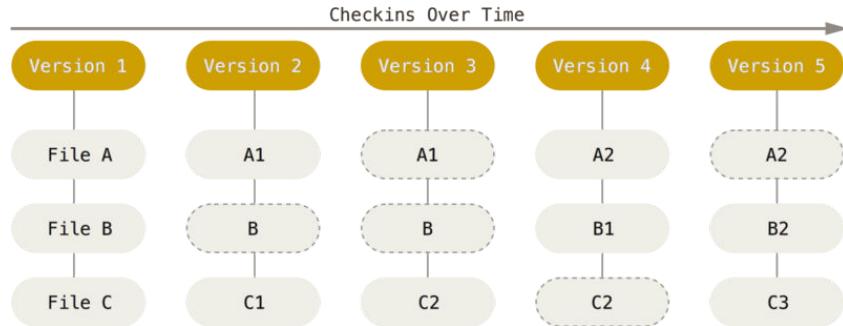
**FIGURE 1-4**

Хранение данных, как набора изменений относительно первоначальной версии каждого из файлов.

Git не хранит и не обрабатывает данные таким способом. Вместо этого, подход Git'a к хранению данных больше похож на набор снимков миниатюрной файловой системы. Каждый раз, когда вы делаете коммит, то есть сохраняете состояние своего проекта в Git'e, система запоминает как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок. Для увеличения эффективности, если файлы не были изменены, Git не запоминает эти файлы вновь, а только создаёт ссылку на предыдущую версию идентичного файла, который уже сохранён. Git представляет свои данные как, скажем, **поток снимков**.

FIGURE 1-5

Хранение данных, как снимков проекта во времени.



Это очень важное отличие между Git и почти любой другой СКВ. Git переосмысливает практически все аспекты контроля версий, которые были скопированы из предыдущего поколения большинством других систем. Это делает Git больше похожим на миниатюрную файловую систему с удивительно мощными утилитами, надстроенными над ней, нежели просто на СКВ. Когда мы будем рассматривать управление ветками в [Chapter 3](#), мы увидим какие преимущества вносит такой подход к работе с данными в Git.

Почти все операции выполняются локально

Для работы большинства операций в Git достаточно локальных файлов и ресурсов — в основном, системе не нужна никакая информация с других компьютеров в вашей сети. Если вы привыкли к ЦСКВ, где большинство операций имеют задержку из-за работы с сетью, то этот аспект Git'a заставит вас думать, что боги скорости наделили Git несказанной мощью. Так как вся история проекта хранится прямо на вашем локальном диске, большинство операций кажутся чуть ли не мгновенными.

Для примера, чтобы посмотреть историю проекта, Git'у не нужно соединяться с сервером, для её получения и отображения — система просто считывает данные напрямую из локальной базы данных. Это означает, что вы увидите историю проекта практически моментально. Если вам необходимо посмотреть изменения, сделанные между текущей версией файла и версией, созданной месяц назад, Git может найти файл месячной давности и локально вычислить изменения, вместо того, чтобы запрашивать удалённый сервер выполнить эту операцию, либо вместо получения старой версии файла с сервера и выполнения операции локально.

Это также означает, что есть лишь небольшое количество действий, которые вы не сможете выполнить если вы находитесь офлайн или не имеете доступа к ВПН в данный момент. Если вы в самолёте или в поезде и хотите немного поработать, вы сможете создавать коммиты без каких-либо проблем, когда будет возможность подключиться к сети, все изменения можно будет синхронизировать. Если вы ушли домой и не можете подключиться через виртуальную частную сеть (ВЧС), вы всё равно сможете работать. Добиться такого же поведения во многих других системах либо очень сложно, либо вовсе невозможно. В Perforce, для примера, если вы не подключены к серверу, вам не удастся сделать многое; в Subversion и CVS вы можете редактировать файлы, но вы не сможете сохранить изменения в базу данных (потому что вы не подключены к БД). Всё это может показаться не таким уж и значимым, но вы удивитесь, какое большое значение это может иметь.

Целостность Git

В Git'е для всего вычисляется хеш-сумма, и только потом происходит сохранение, в дальнейшем, обращение к сохранённым объектам происходит по этой хеш-сумме. Это значит, что невозможно изменить содержимое файла или директории так, чтобы Git не узнал об этом. Данная функциональность встроена в Git на низком уровне и является неотъемлемой частью его философии. Вы не потеряете информацию во время её передачи и не получите повреждённый файл без ведома Git.

Механизм, которым пользуется Git при вычислении хеш-сумм называется SHA-1 хеш. Это строка длинной в 40 шестнадцатеричных символов (0-9 и a-f), она вычисляется на основе содержимого файла или структуры каталога. SHA-1 хеш выглядит примерно так:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Вы будете постоянно встречать хеши в Git'е, потому что он использует их повсеместно. На самом деле, Git сохраняет все объекты, в свою базу данных, не по имени, а по хеш-сумме содержимого объекта.

Git только добавляет данные

Когда вы производите какие-либо действия в Git, практически все из них только добавляют новые данные в базу Git. Очень сложно

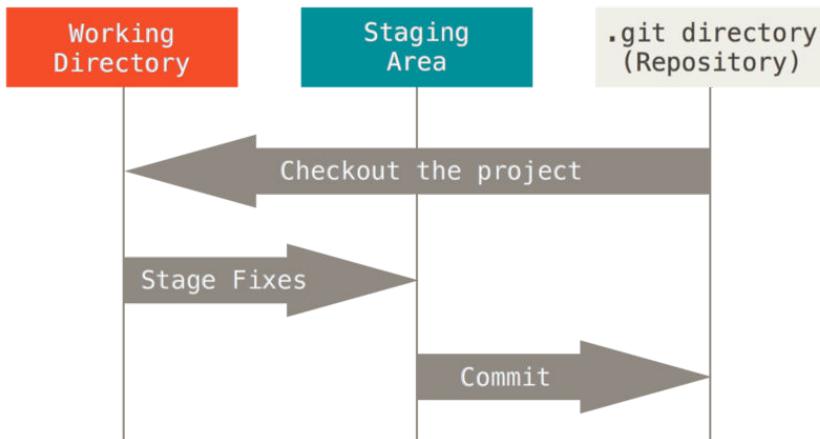
заставить систему удалить данные, либо сделать что-то, что нельзя впоследствии отменить. Как и в любой другой СКВ, вы можете потерять или испортить свои изменения, пока они не закоммичены, но после того, как вы закоммитите снимок в Git, будет очень сложно что-либо потерять, особенно, если вы регулярно синхронизируете свою базу с другим репозиторием.

Всё это превращает использование Git в одно удовольствие, потому что мы знаем, что можем экспериментировать, не боясь серьёзных проблем. Для более глубокого понимания того, как Git хранит свои данные и как вы можете восстановить данные, которые кажутся утерянными, см. “[Операции отмены](#)”.

Три состояния

Теперь слушайте внимательно. Это самая важная вещь, которую нужно запомнить о Git, если вы хотите, чтобы остаток процесса обучения прошёл гладко. Git имеет три основных состояния, в которых могут находиться ваши файлы: зафиксированном (committed), изменённом (modified) и подготовленном (staged). “Зафиксированный” значит, что файл уже сохранён в вашей локальной базе. К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит.

Мы подошли к трём основным секциям проекта Git: Git директория (Git directory), рабочая директория (working directory) область подготовленных файлов (staging area).

**FIGURE 1-6**

Рабочая директория, stage область и директория Git.

Git директория — это то место где Git хранит метаданные и базу объектов вашего проекта. Это самая важная часть Git, и это та часть, которая копируется при клонировании репозитория с другого компьютера.

Рабочая директория является снимком версии проекта. Файлы распаковываются из сжатой базы данных в Git директории и располагаются на диске, для того, чтобы их можно было изменять и использовать.

Область подготовленных файлов — это файл, располагающийся в вашей Git директории, в нём содержится информация о том, какие изменения попадут в следующий коммит. Эту область ещё называют “индекс”, однако называть её stage область также общепринято.

Базовый подход в работе с Git выглядит так:

1. Вы изменяете файлы в вашей рабочей директории.
2. Вы добавляете файлы в индекс, добавляя тем самым их снимки в область подготовленных файлов.
3. Когда вы делаете коммит, используются файлы из индекса, как есть и этот снимок сохраняется в вашу Git директорию.

Если определённая версия файла есть в Git директории, эта версия закоммичена. Если файл изменен и добавлен в область измененных файлов, значит он будет добавлен в следующий коммит. И если файл был изменён с момента последнего распаковывания из репозитория, но не был добавлен в индекс, он считается изменённым. В главе Chapter 2, вы узнаете больше об этих состояниях и какую пользу вы

можете извлечь из них, либо как полностью пропустить часть с индексом.

Командная строка

Есть много различных способов использования Git. Помимо оригинального клиента, имеющего интерфейс командной строки, существует множество клиентов с графическим пользовательским интерфейсом в той или иной степени реализующих функциональность Git. В рамках данной книги мы будем использовать Git в командной строке. С одной стороны, команда строка — это единственное место, где вы можете запустить **все** команды Git, так как большинство клиентов с графическим интерфейсом реализуют для простоты только некоторую часть функционала Git. Если вы знаете, как выполнить какое-либо действие в командной строке, вы, вероятно, сможете выяснить, как то же самое сделать и в GUI-версии, а вот обратное не всегда верно. Кроме того, в то время, как выбор графического клиента — это дело личного вкуса, инструменты командной строки доступны *всем* пользователям сразу после установки Git'a.

Поэтому мы предполагаем, что вы знаете, как открыть терминал в Mac или командную строку, или Powershell в Windows. Если вам не понятно, о чём мы здесь говорим, то вам, возможно, придется ненадолго прерваться и изучить эти вопросы, чтобы вы могли понимать примеры и пояснения из этой книги.

Установка Git

Прежде чем использовать Git, вы должны установить его на своем компьютере. Даже если он уже установлен, наверное, это хороший повод, чтобы обновиться до последней версии. Вы можете установить Git из собранного пакета или другого установщика, либо скачать исходный код и скомпилировать его самостоятельно.

В этой книге используется Git версии 2.0.0. Хотя большинство команд, рассматриваемых в книге, должны корректно работать и в более ранних версиях Git, некоторые из них могут действовать несколько по иному при использовании старых версий. Поскольку Git довольно хорош в вопросе сохранения обратной совместимости, примеры книги должны корректно работать в любой версии старше 2.0.

Установка в Linux

Если вы хотите установить Git под Linux как бинарный пакет, это можно сделать, используя обычный менеджер пакетов вашего дистрибутива. Если у вас Fedora, можно воспользоваться yum'ом:

```
$ yum install git
```

Если же у вас дистрибутив, основанный на Debian, например, Ubuntu, попробуйте apt-get:

```
$ apt-get install git
```

Чтобы воспользоваться дополнительными возможностями, посмотрите инструкцию по установке для нескольких различных разновидностей Unix на сайте Git'a <http://git-scm.com/download/linux>.

Установка на Mac

Существует несколько способов установки Git'a на Mac. Самый простой — установить Xcode Command Line Tools. В версии Mavericks (10.9) и выше вы можете добиться этого просто первый раз выполнив *git* в терминале. Если Git не установлен, вам будет предложено его установить.

Если Вы хотите получить более актуальную версию, то можете воспользоваться бинарным установщиком. Установщик Git для OS X доступен для скачивания с сайта Git'a <http://git-scm.com/download/mac>.

FIGURE 1-7

OS X инсталлятор
Git'a.



Вы также можете установить Git, при установке GitHub для Mac. Их графический интерфейс Git также имеет возможность установить и утилиты командной строки. Скачать клиент GitHub для Mac вы можете с сайта, в <http://mac.github.com> [] .

Установка в Windows

Для установки Git в Windows также имеется несколько способов. Официальная сборка доступна для скачивания на официальном сайте Git'a. Просто перейдите на страницу <http://git-scm.com/download/win>, и загрузка запустится автоматически. Обратите внимание, что это проект, называемый Git для Windows (другое название msysGit), который отделен от самого Git; для получения дополнительной информации о нем, перейдите на <http://msysgit.github.io/> [] .

Другой простой способ установки Git — установить GitHub для Windows. Его установщик включает в себя утилиты командной строки и GUI Git'a. Он также корректно работает с Powershell, обеспечивает четкое сохранение учетных данных и правильные настройки CRLF. Вы познакомитесь с этими вещами подробнее несколько позже, здесь же отметим, что они будут вам необходимы. Вы можете загрузить GitHub для Windows с сайта <http://windows.github.com>.

Установка из исходников

Многие предпочитают устанавливать Git из исходников, поскольку такой способ позволяет получить самую свежую версию. Обновление бинарных инсталляторов как правило немного отстает, хотя в последнее время разница не столь существенна.

Если вы действительно хотите установить Git из исходников, у вас должны быть установлены следующие библиотеки, от которых он зависит: curl, zlib, openssl, expat, and libiconv. Например, если в вашей системе используется yum (например, Fedora) или apt-get (например, системы, базирующиеся на Debian), вы можете использовать одну из следующих команд для установки всех зависимостей, используемых для сборки и установки бинарных файлов Git:

```
$ yum install curl-devel expat-devel gettext-devel \
openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
libbz-dev libssl-dev
```

Для того, чтобы собрать документацию в различных форматах (doc, html, info), понадобятся следующие зависимости:

```
$ yum install asciidoc xmlto docbook2x

$ apt-get install asciidoc xmlto docbook2x
```

Если у вас есть все необходимые зависимости, вы можете двинуться дальше и скачать самый свежий архив с исходниками из следующих мест. С сайта Kernel.org <https://www.kernel.org/pub/software/scm/git>, или зеркала на сайте GitHub <https://github.com/git/git/releases>. Конечно, немного проще, скачать последнюю версию с сайта GitHub, но на странице kernel.org релизы имеют подписи, если вы хотите проверить что скачиваете.

Затем, скомпилируйте и установите:

```
$ tar -zxf git-2.0.0.tar.gz
$ cd git-2.0.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

После этого, вы можете получить Git с помощью службу обновлений Git:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Первоначальная настройка Git

Теперь, когда Git установлен в вашей системе, самое время настроить среду для работы с Git'ом под себя. Это нужно сделать только один раз — при обновлении версии Git'a настройки сохраняются. Но, при необходимости, вы можете поменять их в любой момент, выполнив те же команды снова.

В состав Git'a входит утилита `git config`, которая позволяет просматривать и настраивать параметры, контролирующие все аспекты работы Git'a, а также его внешний вид. Эти параметры могут быть сохранены в трёх местах:

1. Файл `/etc/gitconfig` содержит значения, общие для всех пользователей системы и для всех их репозиториев. Если при запуске `git config` указать параметр `--system`, то параметры будут читаться и сохраняться именно в этот файл.
2. Файл `~/.gitconfig` или `~/.config/git/config` хранит настройки конкретного пользователя. Этот файл используется при указании параметра `--global`.
3. Файл `config` в каталоге Git'a (т.е. `.git/config`) в том репозитории, который вы используете в данный момент, хранит настройки конкретного репозитория.

Настройки на каждом следующем уровне подменяют настройки из предыдущих уровней, то есть значения в `.git/config` перекрывают соответствующие значения в `/etc/gitconfig`.

В системах семейства Windows Git ищет файл `.gitconfig` в каталоге `$HOME` (`C:\Users\$USER` для большинства пользователей). Кроме того Git ищет файл `/etc/gitconfig`, но уже относительно корневого каталога MSys, который находится там, куда вы решили установить Git, когда запускали инсталлятор.

Имя пользователя

Первое, что вам следует сделать после установки Git'a, — указать ваше имя и адрес электронной почты. Это важно, потому что каждый коммит в Git'e содержит эту информацию, и она включена в коммиты, передаваемые вами, и не может быть далее изменена:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Опять же, если указана опция `--global`, то эти настройки достаточно сделать только один раз, поскольку в этом случае Git будет использовать эти данные для всего, что вы делаете в этой системе. Если для каких-то отдельных проектов вы хотите указать другое имя или электронную почту, можно выполнить эту же команду без параметра `--global` в каталоге с нужным проектом.

Многие GUI-инструменты предлагают сделать это при первом запуске.

Выбор редактора

Теперь, когда вы указали своё имя, самое время выбрать текстовый редактор, который будет использоваться, если будет нужно набрать сообщение в Git'e. По умолчанию Git использует стандартный редактор вашей системы, которым обычно является Vim. Если вы хотите использовать другой текстовый редактор, например, Emacs, можно проделать следующее:

```
$ git config --global core.editor emacs
```

Vim и Emacs — популярные текстовые редакторы часто используется разработчиками в Unix-подобных системах, таких как Linux и Mac. Если Вы не знакомы с каким-либо из этих редакторов или работаете на Windows системе, вам вероятно потребуется инструкция по настройке используемого вами редактора для работы с Git. В случае, если вы не установили свой редактор, и не знакомы с Vim или Emacs, то можете попасть в затруднительное положение, когда они будут запущены.

Проверка настроек

Если вы хотите проверить используемую конфигурацию, можете использовать команду `git config --list`, чтобы показать все настройки, которые Git найдёт:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
```

```
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Некоторые ключи (названия) настроек могут появиться несколько раз, потому что Git читает один и тот же ключ из разных файлов (например из `/etc/gitconfig` и `~/.gitconfig`). В этом случае Git использует последнее значение для каждого ключа.

Также вы можете проверить значение конкретного ключа, выполнив `git config <key>`:

```
$ git config user.name
John Doe
```

Как получить помощь?

Если вам нужна помощь при использовании Git, есть три способа открыть страницу руководства по любой команде Git:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

Например, так можно открыть руководство по команде `config`

```
$ git help config
```

Эти команды хороши тем, что ими можно пользоваться всегда, даже без подключения к сети. Если руководства в этой книге недостаточно и вам нужна персональная помощь, вы можете попытаться поискать её на каналах `#git` и `#github` сервера Freenode IRC (`irc.freenode.net`). Обычно там сотни людей, отлично знающих Git, которые могут помочь.

Заключение

Вы получили базовые знания о том, что такое Git и чем он отличается от централизованных систем контроля версий, которыми вы, возможно, пользовались. Также вы теперь получили рабочую версию Git в вашей ОС, настроенную и персонализированную. Самое время изучить основы Git.

Основы Git 2

Если вы хотите начать работать с Git'ом, прочитав всего одну главу, то эта глава — то, что вам нужно. Здесь рассмотрены все базовые команды, необходимые вам для решения подавляющего большинства задач, возникающих при работе с Git'ом. После прочтения этой главы вы научитесь настраивать и инициализировать репозиторий, начинать и прекращать контроль версий файлов, а также подготавливать и фиксировать изменения. Мы также продемонстрируем вам, как настроить в Git'e игнорирование отдельных файлов или их групп, как быстро и просто отменить ошибочные изменения, как просмотреть историю вашего проекта и изменения между отдельными коммитами (commit), а также как отправлять (push) и получать (pull) изменения в/из удалённого (remote) репозитория.

Создание Git-репозитория

Для создания Git-репозитория вы можете использовать два основных подхода. Во-первых, импорт в Git уже существующего проекта или директории. Во-вторых, клонирование существующего репозитория с другого сервера.

Создание репозитория в существующей директории

Если вы собираетесь начать использовать Git для существующего проекта, то вам необходимо перейти в директорию проекта и в командной строке ввести

```
$ git init
```

Эта команда создаёт в текущей директории новую поддиректорию с именем `.git`, содержащую все необходимые файлы репозитория — основу Git-репозитория. На этом этапе ваш проект ещё не находится под версионным контролем. (Подробное описание файлов содержащихся в только что созданной вами директории `.git` приведено в главе [Chapter 10](#))

Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит проиндексировать эти файлы и осуществить первый коммит изменений. Добиться этого вы сможете запустив команду `git add` несколько раз, указав индексируемые файлы, а затем выполнив `git commit`:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

Мы разберем, что делают эти команды чуть позже. Теперь у вас есть Git-репозиторий с отслеживаемыми файлами и начальным коммитом.

Клонирование существующего репозитория

Для получения копии существующего Git-репозитория, например, проекта, в который вы хотите внести свой вклад, необходимо использовать команду `git clone`. Если вы знакомы с другими системами контроля версий, такими как Subversion, то заметите, что команда называется “clone”, а не “checkout”. Это важное различие — вместо того, чтобы просто получить рабочую копию, Git получает копию практически всех данных, которые есть на сервере. При выполнении `git clone` с сервера зибирается (pulled) каждая версия каждого файла из истории проекта. Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов, для того, чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования (вы можете потерять часть серверных перехватчиков (server-side hooks) и т.п., но все данные, помещённые под версионный контроль, будут сохранены, подробнее об этом смотрите в главе “Установка Git на сервер”).

Клонирование репозитория осуществляется командой `git clone` [url]. Например, если вы хотите клонировать библиотеку libgit2, вы можете сделать это следующим образом:

```
$ git clone https://github.com/libgit2/libgit2
```

Эта команда создаёт директорию “libgit2”, инициализирует в ней поддиректорию .git, скачивает все данные для этого репозитория и создаёт (checks out) рабочую копию последней версии. Если вы зайдёте в новую директорию libgit2, то увидите в ней файлы проекта, готовые для работы или использования. Для того, чтобы клонировать репозиторий в директорию с именем, отличающимся от “libgit2”, необходимо указать желаемое имя, как параметр командной строки:

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван mylibgit.

В Git'e реализовано несколько транспортных протоколов, которые вы можете использовать. В предыдущем примере использовался протокол https://, вы также можете встретить git:// или user@server:path/to/repo.git, использующий протокол передачи SSH. В главе “Установка Git на сервер” мы познакомимся со всеми доступными вариантами конфигурации сервера для обеспечения доступа к вашему Git-репозиторию, а также рассмотрим их достоинства и недостатки.

Recording Changes to the Repository

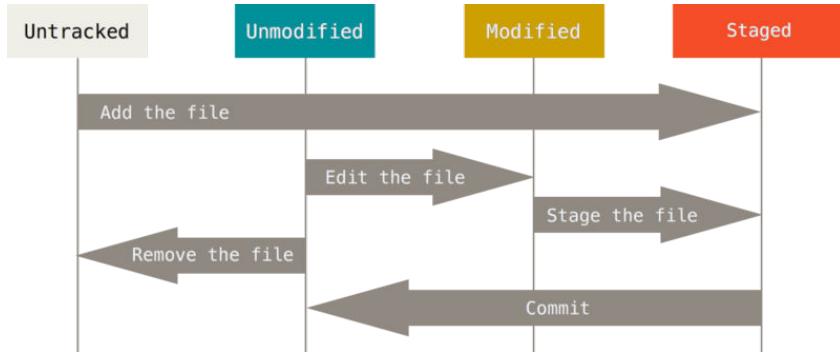
You have a bona fide Git repository and a checkout or working copy of the files for that project. You need to make some changes and commit snapshots of those changes into your repository each time the project reaches a state you want to record.

Remember that each file in your working directory can be in one of two states: tracked or untracked. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. Untracked files are everything else – any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because you just checked them out and haven't edited anything.

As you edit files, Git sees them as modified, because you've changed them since your last commit. You stage these modified files and then commit all your staged changes, and the cycle repeats.

FIGURE 2-1

The lifecycle of the status of your files.



Checking the Status of Your Files

The main tool you use to determine which files are in which state is the `git status` command. If you run this command directly after a clone, you should see something like this:

```
$ git status
On branch master
nothing to commit, working directory clean
```

This means you have a clean working directory – in other words, there are no tracked and modified files. Git also doesn't see any untracked files, or they would be listed here. Finally, the command tells you which branch you're on and informs you that it has not diverged from the same branch on the server. For now, that branch is always “master”, which is the default; you won't worry about it here. [Chapter 3](#) will go over branches and references in detail.

Let's say you add a new file to your project, a simple README file. If the file didn't exist before, and you run `git status`, you see your untracked file like so:

```
$ echo 'My Project' > README
$ git status
On branch master
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)

README

nothing added to commit but untracked files present (use "git add" to track)
```

You can see that your new README file is untracked, because it's under the "Untracked files" heading in your status output. Untracked basically means that Git sees a file you didn't have in the previous snapshot (commit); Git won't start including it in your commit snapshots until you explicitly tell it to do so. It does this so you don't accidentally begin including generated binary files or other files that you did not mean to include. You do want to start including README, so let's start tracking the file.

Tracking New Files

In order to begin tracking a new file, you use the command `git add`. To begin tracking the README file, you can run this:

```
$ git add README
```

If you run your status command again, you can see that your README file is now tracked and staged to be committed:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

You can tell that it's staged because it's under the "Changes to be committed" heading. If you commit at this point, the version of the file at the time you ran `git add` is what will be in the historical snapshot. You may recall that when you ran `git init` earlier, you then ran `git add (files)` – that was to begin tracking files in your directory. The `git add` command takes a path name for either a file or a directory; if it's a directory, the command adds all the files in that directory recursively.

Staging Modified Files

Let's change a file that was already tracked. If you change a previously tracked file called "CONTRIBUTING.md" and then run your `git status` command again, you get something that looks like this:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file: README

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: CONTRIBUTING.md
```

The "CONTRIBUTING.md" file appears under a section named "Changed but not staged for commit" – which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, you run the `git add` command. `git add` is a multipurpose command – you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as "add this content to the next commit" rather than "add this file to the project". Let's run `git add` now to stage the "CONTRIBUTING.md" file, and then run `git status` again:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file: README
    modified: CONTRIBUTING.md
```

Both files are staged and will go into your next commit. At this point, suppose you remember one little change that you want to make in `CONTRIBUTING.md` before you commit it. You open it again and make that change, and you're ready to commit. However, let's run `git status` one more time:

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

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:   CONTRIBUTING.md
```

What the heck? Now `CONTRIBUTING.md` is listed as both staged *and* unstaged. How is that possible? It turns out that Git stages a file exactly as it is when you run the `git add` command. If you commit now, the version of `CONTRIBUTING.md` as it was when you last ran the `git add` command is how it will go into the commit, not the version of the file as it looks in your working directory when you run `git commit`. If you modify a file after you run `git add`, you have to run `git add` again to stage the latest version of the file:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Short Status

While the `git status` output is pretty comprehensive, it's also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run `git status -s` or `git status --short` you get a far more simplified output from the command.

```
$ git status -s
M README
MM Rakefile
A lib/git.rb
```

```
M lib/simplegit.rb
?? LICENSE.txt
```

New files that aren't tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M and so on. There are two columns to the output - the left hand column indicates that the file is staged and the right hand column indicates that it's modified. So for example in that output, the README file is modified in the working directory but not yet staged, while the lib/simplegit.rb file is modified and staged. The Rakefile was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

Ignoring Files

Often, you'll have a class of files that you don't want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named `.gitignore`. Here is an example `.gitignore` file:

```
$ cat .gitignore
*.[oa]
*~
```

The first line tells Git to ignore any files ending in ".o" or ".a" – object and archive files that may be the product of building your code. The second line tells Git to ignore all files that end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files. You may also include a log, tmp, or pid directory; automatically generated documentation; and so on. Setting up a `.gitignore` file before you get going is generally a good idea so you don't accidentally commit files that you really don't want in your Git repository.

The rules for the patterns you can put in the `.gitignore` file are as follows:

- Blank lines or lines starting with # are ignored.
- Standard glob patterns work.
- You can end patterns with a forward slash (/) to specify a directory.
- You can negate a pattern by starting it with an exclamation point (!).

Glob patterns are like simplified regular expressions that shells use. An asterisk (*) matches zero or more characters; [abc] matches any character inside

the brackets (in this case a, b, or c); a question mark (?) matches a single character; and brackets enclosing characters separated by a hyphen([0-9]) matches any character between them (in this case 0 through 9). You can also use two asterisks to match nested directories; a/**/z would match a/z, a/b/z, a/b/c/z, and so on.

Here is another example .gitignore file:

```
# no .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the root TODO file, not subdir/TODO
/TODO

# ignore all files in the build/ directory
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .txt files in the doc/ directory
doc/**/.txt
```

GitHub maintains a fairly comprehensive list of good .gitignore file examples for dozens of projects and languages at <https://github.com/github/gitignore> if you want a starting point for your project.

Viewing Your Staged and Unstaged Changes

If the git status command is too vague for you – you want to know exactly what you changed, not just which files were changed – you can use the git diff command. We'll cover git diff in more detail later, but you'll probably use it most often to answer these two questions: What have you changed but not yet staged? And what have you staged that you are about to commit? Although git status answers those questions very generally by listing the file names, git diff shows you the exact lines added and removed – the patch, as it were.

Let's say you edit and stage the README file again and then edit the CONTRIBUTING.md file without staging it. If you run your git status command, you once again see something like this:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file: README

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: CONTRIBUTING.md
```

To see what you've changed but not yet staged, type `git diff` with no other arguments:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
Please include a nice description of your changes when you submit your PR;
if we have to read the whole diff to figure out why you're contributing
in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.
```

If you are starting to work on a particular area, feel free to submit a PR that highlights your work in progress (and note in the PR title that it's

That command compares what is in your working directory with what is in your staging area. The result tells you the changes you've made that you haven't yet staged.

If you want to see what you've staged that will go into your next commit, you can use `git diff --staged`. This command compares your staged changes to your last commit:

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
```

```
+++ b/README
@@ -0,0 +1 @@
+My Project
```

It's important to note that `git diff` by itself doesn't show all changes made since your last commit – only changes that are still unstaged. This can be confusing, because if you've staged all of your changes, `git diff` will give you no output.

For another example, if you stage the `CONTRIBUTING.md` file and then edit it, you can use `git diff` to see the changes in the file that are staged and the changes that are unstaged. If our environment looks like this:

```
$ git add CONTRIBUTING.md
$ echo 'test line' >> CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md

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:   CONTRIBUTING.md
```

Now you can use `git diff` to see what is still unstaged

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects
```

```
See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+# test line
```

and `git diff --cached` to see what you've staged so far (`--staged` and `--cached` are synonyms):

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
Please include a nice description of your changes when you submit your PR;
if we have to read the whole diff to figure out why you're contributing
in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.
```

If you are starting to work on a particular area, feel free to submit a PR that highlights your work in progress (and note in the PR title that it's

GIT DIFF IN AN EXTERNAL TOOL

We will continue to use the `git diff` command in various ways throughout the rest of the book. There is another way to look at these diffs if you prefer a graphical or external diff viewing program instead. If you run `git difftool` instead of `git diff`, you can view any of these diffs in software like Araxis, emerge, vimdiff and more. Run `git difftool --tool-help` to see what is available on your system.

Committing Your Changes

Now that your staging area is set up the way you want it, you can commit your changes. Remember that anything that is still unstaged – any files you have created or modified that you haven't run `git add` on since you edited them – won't go into this commit. They will stay as modified files on your disk. In this case, let's say that the last time you ran `git status`, you saw that everything was staged, so you're ready to commit your changes. The simplest way to commit is to type `git commit`:

```
$ git commit
```

Doing so launches your editor of choice. (This is set by your shell's `$EDITOR` environment variable – usually vim or emacs, although you can configure it with whatever you want using the `git config --global core.editor` command as you saw in [Chapter 1](#)).

The editor displays the following text (this example is a Vim screen):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       new file: README
#       modified: CONTRIBUTING.md
#
#
#
#
#.git/COMMIT_EDITMSG" 9L, 283C
```

You can see that the default commit message contains the latest output of the `git status` command commented out and one empty line on top. You can remove these comments and type your commit message, or you can leave them there to help you remember what you’re committing. (For an even more explicit reminder of what you’ve modified, you can pass the `-v` option to `git commit`. Doing so also puts the diff of your change in the editor so you can see exactly what changes you’re committing.) When you exit the editor, Git creates your commit with that commit message (with the comments and diff stripped out).

Alternatively, you can type your commit message inline with the `commit` command by specifying it after a `-m` flag, like this:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

Now you’ve created your first commit! You can see that the commit has given you some output about itself: which branch you committed to (`master`), what SHA-1 checksum the commit has (`463dc4f`), how many files were changed, and statistics about lines added and removed in the commit.

Remember that the commit records the snapshot you set up in your staging area. Anything you didn’t stage is still sitting there modified; you can do another commit to add it to your history. Every time you perform a commit, you’re recording a snapshot of your project that you can revert to or compare to later.

Skip the Staging Area

Although it can be amazingly useful for crafting commits exactly how you want them, the staging area is sometimes a bit more complex than you need in your workflow. If you want to skip the staging area, Git provides a simple shortcut.

Adding the `-a` option to the `git commit` command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the `git add` part:

```
$ 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:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
  1 file changed, 5 insertions(+), 0 deletions(-)
```

Notice how you don't have to run `git add` on the "CONTRIBUTING.md" file in this case before you commit.

Removing Files

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The `git rm` command does that, and also removes the file from your working directory so you don't see it as an untracked file the next time around.

If you simply remove the file from your working directory, it shows up under the "Changed but not updated" (that is, *unstaged*) area of your `git status` output:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    deleted:   PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Then, if you run `git rm`, it stages the file's removal:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:   PROJECTS.md
```

The next time you commit, the file will be gone and no longer tracked. If you modified the file and added it to the index already, you must force the removal with the `-f` option. This is a safety feature to prevent accidental removal of data that hasn't yet been recorded in a snapshot and that can't be recovered from Git.

Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your `.gitignore` file and accidentally staged it, like a large log file or a bunch of `.a` compiled files. To do this, use the `--cached` option:

```
$ git rm --cached README
```

You can pass files, directories, and file-glob patterns to the `git rm` command. That means you can do things such as

```
$ git rm log/\*.log
```

Note the backslash (\) in front of the *. This is necessary because Git does its own filename expansion in addition to your shell's filename expansion. This command removes all files that have the `.log` extension in the `log/` directory. Or, you can do something like this:

```
$ git rm \*~
```

This command removes all files that end with ~.

Moving Files

Unlike many other VCS systems, Git doesn't explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file. However, Git is pretty smart about figuring that out after the fact – we'll deal with detecting file movement a bit later.

Thus it's a bit confusing that Git has a `mv` command. If you want to rename a file in Git, you can run something like

```
$ git mv file_from file_to
```

and it works fine. In fact, if you run something like this and look at the status, you'll see that Git considers it a renamed file:

```
$ git mv README.md README
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

However, this is equivalent to running something like this:

```
$ mv README.md README
$ git rm README.md
$ git add README
```

Git figures out that it's a rename implicitly, so it doesn't matter if you rename a file that way or with the `mv` command. The only real difference is that `mv` is one command instead of three – it's a convenience function. More important, you can use any tool you like to rename a file, and address the add/rm later, before you commit.

Просмотр истории коммитов

После того, как вы создали несколько коммитов или же склонировали репозиторий с уже существующей историей коммитов, вероятно вам понадобится возможность посмотреть что было сделано – историю

коммитов. Одним из основных и наиболее мощных инструментов для этого является команда `git log`.

Следующие несколько примеров используют очень простой проект “simplegit”. Что бы склонировать проект, используйте команду:

```
git clone https://github.com/schacon/simplegit-progit
```

Если вы запустите команду `git log` в папке склонированного проекта, вы увидите следующий вывод:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

По умолчанию (без аргументов) `git log` перечисляет коммиты, сделанные в репозитории в обратном к хронологическому порядке – последние коммиты находятся вверху. С примера можно увидеть, что данная команда перечисляет коммиты с их SHA-1 чек-суммами, именем и электронной почтой автора, датой создания и сообщением коммита.

Команда `git log` имеет очень большое количество опций для поиска коммитов по разным критериям. Посмотрим на наиболее популярные из них.

Одним из наиболее полезных аргументов является `-p`, который показывает разницу, внесенную в каждый коммит. Так же вы можете использовать аргумент `-2`, который позволяет установить лимит на вывод количества коммитов. В данном случае их будет только два:

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
spec = Gem::Specification.new do |s|
  s.platform  = Gem::Platform::RUBY
  s.name      = "simplegit"
-  s.version   = "0.1.0"
+  s.version   = "0.1.1"
  s.author    = "Scott Chacon"
  s.email     = "schacon@gee-mail.com"
  s.summary   = "A simple gem for using Git in Ruby code."
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
end

end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file
```

Эта опция отображает аналогичную информацию но содержит разницу для каждой записи. Очень удобно использовать данную опцию для код рев'ю или для быстрого просмотра серии изменений. Так же есть возможность использовать серию опций для обобщения.

Например, если вы хотите увидеть сокращенную статистику для каждого коммита, вы можете использовать опцию `--stat`:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

        changed the version number

Rakefile | 2 ++
1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

        removed unnecessary test

lib/simplegit.rb | 5 -----
1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

        first commit

 README          |  6 ++++++
 Rakefile        | 23 ++++++++++++++++++++++
 lib/simplegit.rb | 25 ++++++++++++++++++++++
 3 files changed, 54 insertions(+)
```

Как вы видите, опция `--stat` печатает под каждым из коммитов список и количество измененных файлов, а также сколько строк в каждом из файлов было добавлено и удалено. В конце можно увидеть суммарную таблицу изменений.

Следующей действительно полезной опцией является `--pretty`. Эта опция меняет формат вывода. Существует несколько встроенных вариантов отображения. Например, опция `oneline` печатает каждый коммит в одну строку, что может быть очень удобным если вы просматриваете большое количество коммитов. К тому же, опции `short`, `full` и `fuller` делают вывод приблизительно в том же формате, но с меньшим или большим количеством информации соответственно:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

Наиболее интересной опцией является `format`. Она позволяет создать свой формат для вывода информации. Особенно это может быть полезным когда вы хотите сгенерировать вывод для автоматического анализа – так как вы указываете формат явно, он не будет изменен даже после обновления Git:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

Table 2-1 отображает наиболее полезные опции для изменения формата.

TABLE 2-1. Полезные опции для `git log --pretty=format`

Опция	Описания вывода
%H	Хеш коммита
%h	Сокращенный хеш коммита
%T	Хеш дерева
%t	Сокращенный хеш дерева
%P	Хеш родителей
%p	Сокращенный хеш родителей
%an	Имя автора
%ae	Электронная почта автора
%ad	Дата автора (формат даты можно задать опцией <code>--date=options</code>)
%ag	Относительная дата автора
%cp	Имя коммитера
%ce	Электронная почта коммитера

Опция	Описания вывода
%cd	Дата коммитера
%сг	Относительная дата коммитера
%s	Содержание

Вам наверное интересно, какая же разница между *автором* и *коммитером*. Автор – это человек, изначально сделавший работу, а коммитер – это человек, который последним применил эту работу. Другими словами, если вы создадите патч для какого-то проекта, а один из основных членов команды этого проекта применит этот патч, вы оба получите статус участника – вы как автор и основной член команды как коммитер.

Опции `oneline` и `format` являются особенно полезными с опцией `--graph` команды `log`. С этой опцией вы сможете увидеть небольшой граф в формате ASCII, который показывает текущую ветку и историю слияний:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
  \
  | * 420eac9 Added a method for getting the current branch.
  * | 30e367c timeout code and tests
  * | 5a09431 add timeout protection to grit
  * | e1193f8 support for heads with slashes in them
  /
* d6016bc require time for xmlschema
* 11d191e Merge branch 'defunkt' into local
```

Данный вывод будет нам очень интересен в следующей главе, где мы рассмотрим ветвления и слияния.

Мы рассмотрели только несколько простых опций для форматирования вывода с помощью команды `git log`. На самом деле их немного больше. **Table 2-2** содержит описание как уже рассмотренных, так и нескольких новых опций, которые могут быть полезными в зависимости от нужного формата вывода.

TABLE 2-2. Наиболее распространенные опции для команды `git log`

Опция	Описание
<code>-p</code>	Показывает патч для каждого коммита.
<code>--stat</code>	Показывает статистику измененных файлов для каждого коммита.
<code>--shortstat</code>	Отображает только строку с количеством изменений/вставок/удалений для команды <code>--stat</code> .
<code>--name-only</code>	Показывает список измененных файлов после информации о коммите.
<code>--name-status</code>	Показывает список файлов, которые добавлены/изменены/удалены.
<code>--abbrev-commit</code>	Показывает только несколько символом SHA-1 чек-суммы вместо всех 40.
<code>--relative-date</code>	Отображает дату в относительном формате (например, “2 weeks ago”) вместо стандартного формата даты.
<code>--graph</code>	Отображает ASCII график с ветвлением и историей слияний.
<code>--pretty</code>	Показывает коммиты в альтернативном формате. Возможные варианты опций: <code>oneline</code> , <code>short</code> , <code>full</code> , <code>fuller</code> и <code>format</code> (с помощью последней опции вы можете указать свой формат).

Ограничение вывода

В дополнение к опциям форматирования вывода, команда `git log` принимает несколько опций для ограничения вывода – опций, с помощью которых можно увидеть определенное подмножество коммитов. Вы уже видели одну из таких опций – это опция `-2`, которая показывает только последние два коммита. В действительности вы можете использовать `-<n>`, где `n` – это любое натуральное число, что и представляет собой `n` последних коммитов. На самом деле, вы не будете часто использовать эту опцию, потому что Git по умолчанию использует систему отображения страниц и вы можете видеть только одну страницу вывода в определенный момент времени.

Однако, опции для ограничения вывода по времени, такие как `--since` и `--until`, являются очень удобными. Например, следующая команда покажет список коммитов, сделанных за последние две недели:

```
$ git log --since=2.weeks
```

Это команда работает с большим количеством форматов – вы можете указать определенную дату вида "2008-01-15" или же относительную дату, например "2 years 1 day 3 minutes ago".

Также вы можете фильтровать список коммитов, которые удовлетворяют каким-то критериям. Опция `--author` дает возможность фильтровать по автору коммита, а опция `--grep` искать по ключевым словам в сообщении коммита. (Имейте ввиду, что если вы хотите фильтровать коммиты по автору и ключевым словам одновременно, вам нужно также добавить `--all-match`. В противном случае, команда отфильтрует вывод по одному из двух критериев.)

Следующим действительно полезным фильтром является опция `-S`, которая, как аргумент, принимает строку и показывает только те коммиты, в которых изменение в коде повлекло за собой добавление или удаление этой строки. Например, если вы хотите найти последний коммит, который добавил или удалил вызов определенной функции, вы можете запустить команду:

```
$ git log -Sfunction_name
```

Последней полезной опцией, которую принимает команда `git log` как фильтр, является путь. Если вы укажете директорию или имя файла, вы ограничите вывод только теми коммитами, в которых были изменения этих файлов. Эта опция всегда указывается последней после двойного тире (`--`), что отделяет указываемый путь от опций.

В таблице **Table 2-3** вы можете увидеть эти и другие распространенные опции.

TABLE 2-3. Опции для ограничения вывода команды `git log`

Опция	Описание
<code>-n</code>	Показывает только последние n коммитов.
<code>--since, --after</code>	Показывает только те коммиты, которые были сделаны после указанной даты.
<code>--until, --before</code>	Показывает только те коммиты, которые были сделаны до указанной даты.
<code>--author</code>	Показывает только те коммиты, в которых запись author совпадает с указанной строкой.

Опция	Описание
--committer	Показывает только те коммиты, в которых запись committer совпадает с указанной строкой.
--grep	Показывает только коммиты, сообщение которых содержит указанную строку.
-S	Показывает только коммиты, в которых изменение в коде повлекло за собой добавление или удаление указанной строки.

Например, если вы хотите увидеть, в каких коммитах произошли изменения в тестовых файлах в истории исходного кода Git, автором которых был Junio Hamano и которые не были слияниями в октябре 2008 года, вы можете запустить следующую команду:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
--before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attributes are in use
acd3b9e - Enhance hold_lock_file_for_{update,append}() API
f563754 - demonstrate breakage of detached checkout with symbolic link HEAD
d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
b0ad11e - pull: allow "git pull origin $something:$current_branch" into an unborn
```

Из почти 40,000 коммитов в истории исходного кода Git, эта команда показывает только 6, которые соответствуют этим критериям.

Операции отмены

В любой момент вам может потребоваться что-либо отменить. Здесь мы рассмотрим несколько основных способов отмены сделанных изменений. Будьте осторожны, не все операции отмены в свою очередь можно отменить! Это одна из редких областей Git'a, где неверными действиями можно необратимо удалить результаты своей работы.

Отмена может потребоваться, если вы сделали коммит слишком рано, например, забыв добавить какие-то файлы или комментарий к коммиту. Если вы хотите переделать коммит, можно запустить commit с параметром `--amend` (дополнить):

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

Эта команда использует для дополнения коммита вашу область подготовки (индекс). Если вы ничего не меняли с момента последнего коммита (например, команда запущена сразу после предыдущего коммита), то снимок состояния останется в точности таким же, а изменится лишь комментарий к коммиту.

Запустится тот же редактор комментария к коммиту, но уже с комментарием к предыдущему коммиту. Комментарий можно отредактировать точно так же, как обычно, просто он заменит собой предыдущий.

Например, если вы фиксируете изменения, и понимаете, что забыли проиндексировать изменения в файле, который хотели включить в коммит, можно сделать примерно так:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

В итоге получится единый коммит — второй коммит заменит результаты первого.

Отмена подготовки файла

В следующих двух разделах показано, как разбираться с изменениями вашей области подготовки (staging area) и рабочего каталога. Радует, что команда, которой вы определяете состояние этих областей, также напоминает вам, как отменять их изменения. Например, скажем, вы изменили два файла, и хотите закоммитить их двумя раздельными изменениями, но случайно набрали `git add *`, и добавили оба в индекс. Как отменить добавление одного из них? Команда `git status` напомнит вам:

```
$ git add .
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
renamed: README.md -> README
modified: CONTRIBUTING.md
```

Прямо под текстом “Changes to be committed” говорится: `git reset HEAD <file>...` для отмены добавления в индекс. Давайте последуем этому совету, и отменим индексирование файла `CONTRIBUTING.md`:

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M      CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:   README.md -> README

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:  CONTRIBUTING.md
```

Команда выглядит несколько странно, но — работает! Файл `CONTRIBUTING.md` изменен, но снова не добавлен в область подготовки к коммиту.

Хотя `git reset` может стать опасной командой, если ее вызвать с `--hard`, в приведенном примере файл в вашем рабочем каталоге не затрагивается. Вызов `git reset` без параметра не опасен — он затрагивает только область подготовки.

Пока этот волшебный вызов — всё, что вам нужно знать о команде `git reset`. Мы гораздо глубже погрузимся в подробности действия `reset` и научимся с ее помощью делать действительно интересные вещи в “Раскрытие тайн `reset`”.

Отмена изменения измененного файла

Что делать, если вы поняли, что не хотите сохранять свои изменения файла `CONTRIBUTING.md`? Как можно просто «разызменить» его — вернуть к тому виду, который был в последнем коммите (или к

изначально склонированому, или еще как-то полученному в рабочий каталог)? Нам повезло, что `git status` рассказывает и это тоже. В последнем примере рассказ о неподготовленных изменениях выглядит примерно так:

```
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:   CONTRIBUTING.md
```

Здесь довольно ясно указано, как отбросить сделанные изменения. Давайте так и сделаем:

```
$ git checkout -- CONTRIBUTING.md  
$ git status  
On branch master  
Changes to be committed:  
  (use "git reset HEAD <file>..." to unstage)  
  
  renamed:   README.md -> README
```

Как видите, откат изменений выполнен.

Важно понимать, что `git checkout -- [file]` — опасная команда. Любые изменения соответствующего файла пропадают — вы просто копируете поверх него другой файл. Ни в коем случае не используйте эту команду, если вы не убеждены, что файл вам не нужен.

Если вы хотите сохранить изменения файла, но пока отложить их в сторону, давайте пройдемся по тому, как прятать (`stash`) и создавать ветки (`branch`) в [Chapter 3](#); эти способы обычно лучше.

Помните, все, что [зарегистрировано коммитом](#) в Git, почти всегда можно восстановить. Можно восстановить даже коммиты, сделанные в удаленных ветках, или коммиты, замещенные параметром `-amend` (см. “[Восстановление данных](#)”). Но все, что вы потеряете, не сделав коммит, скорее всего, вам больше не увидеть.

Working with Remotes

To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work. Managing remote repositories includes knowing how to add remote repositories, remove remotes that are no longer valid, manage various remote branches and define them as being tracked or not, and more. In this section, we'll cover some of these remote-management skills.

Showing Your Remotes

To see which remote servers you have configured, you can run the `git remote` command. It lists the shortnames of each remote handle you've specified. If you've cloned your repository, you should at least see `origin` – that is the default name Git gives to the server you cloned from:

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

You can also specify `-v`, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

If you have more than one remote, the command lists them all. For example, a repository with multiple remotes for working with several collaborators might look something like this.

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
origin    git@github.com:mojombo/grit.git (push)
```

This means we can pull contributions from any of these users pretty easily. We may additionally have permission to push to one or more of these, though we can't tell that here.

Notice that these remotes use a variety of protocols; we'll cover more about this in “[Установка Git на сервер](#)”.

Adding Remote Repositories

We've mentioned and given some demonstrations of adding remote repositories in previous sections, but here is how to do it explicitly. To add a new remote Git repository as a shortname you can reference easily, run `git remote add [shortname] [url]`:

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb     https://github.com/paulboone/ticgit (fetch)
pb     https://github.com/paulboone/ticgit (push)
```

Now you can use the string `pb` on the command line in lieu of the whole URL. For example, if you want to fetch all the information that Paul has but that you don't yet have in your repository, you can run `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
```

```
Unpacking objects: 100% (43/43), done.  
From https://github.com/paulboone/ticgit  
 * [new branch]      master      -> pb/master  
 * [new branch]      ticgit      -> pb/ticgit
```

Paul’s master branch is now accessible locally as `pb/master` – you can merge it into one of your branches, or you can check out a local branch at that point if you want to inspect it. (We’ll go over what branches are and how to use them in much more detail in [Chapter 3](#).)

Fetching and Pulling from Your Remotes

As you just saw, to get data from your remote projects, you can run:

```
$ git fetch [remote-name]
```

The command goes out to that remote project and pulls down all the data from that remote project that you don’t have yet. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time.

If you clone a repository, the command automatically adds that remote repository under the name “origin”. So, `git fetch origin` fetches any new work that has been pushed to that server since you cloned (or last fetched from) it. It’s important to note that the `git fetch` command pulls the data to your local repository – it doesn’t automatically merge it with any of your work or modify what you’re currently working on. You have to merge it manually into your work when you’re ready.

If you have a branch set up to track a remote branch (see the next section and [Chapter 3](#) for more information), you can use the `git pull` command to automatically fetch and then merge a remote branch into your current branch. This may be an easier or more comfortable workflow for you; and by default, the `git clone` command automatically sets up your local master branch to track the remote master branch (or whatever the default branch is called) on the server you cloned from. Running `git pull` generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.

Pushing to Your Remotes

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: `git push [remote-name] [branch-name]`. If you want to push your master branch to your `origin` server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you've done back up to the server:

```
$ git push origin master
```

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to pull down their work first and incorporate it into yours before you'll be allowed to push. See [Chapter 3](#) for more detailed information on how to push to remote servers.

Inspecting a Remote

If you want to see more information about a particular remote, you can use the `git remote show [remote-name]` command. If you run this command with a particular shortname, such as `origin`, you get something like this:

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                  tracked
    dev-branch              tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

It lists the URL for the remote repository as well as the tracking branch information. The command helpfully tells you that if you're on the `master` branch and you run `git pull`, it will automatically merge in the `master` branch on the remote after it fetches all the remote references. It also lists all the remote references it has pulled down.

That is a simple example you're likely to encounter. When you're using Git more heavily, however, you may see much more information from `git remote show`:

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
    master                  tracked
    dev-branch              tracked
    markdown-strip          tracked
    issue-43                new (next fetch will store in remotes/origin)
    issue-45                new (next fetch will store in remotes/origin)
    refs/remotes/origin/issue-11   stale (use 'git remote prune' to remove)
  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
    master     merges with remote master
  Local refs configured for 'git push':
    dev-branch      pushes to dev-branch
    markdown-strip  pushes to markdown-strip
    master         pushes to master
(up to date)
(up to date)
(up to date)
```

This command shows which branch is automatically pushed to when you run `git push` while on certain branches. It also shows you which remote branches on the server you don't yet have, which remote branches you have that have been removed from the server, and multiple branches that are automatically merged when you run `git pull`.

Removing and Renaming Remotes

If you want to rename a reference you can run `git remote rename` to change a remote's shortname. For instance, if you want to rename `pb` to `paul`, you can do so with `git remote rename`:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

It's worth mentioning that this changes your remote branch names, too. What used to be referenced at `pb/master` is now at `paul/master`.

If you want to remove a remote for some reason – you've moved the server or are no longer using a particular mirror, or perhaps a contributor isn't contributing anymore – you can use `git remote rm`:

```
$ git remote rm paul
$ git remote
origin
```

Работа с метками

Как и большинство СКВ, Git имеет возможность помечать (tag) определённые моменты в истории как важные. Как правило, этот функционал используется для отметки моментов выпуска версий (v1.0, и т.п.). В этом разделе вы узнаете, как посмотреть имеющиеся метки (tag), как создать новые. А также вы узнаете, что из себя представляют разные типы меток.

Просмотр меток

Просмотр имеющихся меток (tag) в Git'е делается просто. Достаточно набрать `git tag`:

```
$ git tag
v0.1
v1.3
```

Данная команда перечисляет метки в алфавитном порядке; порядок их появления не имеет значения.

Для меток вы также можете осуществлять поиск по шаблону. Например, репозиторий Git'a содержит более 500 меток. Если вас интересует просмотр только выпусков 1.8.5, вы можете выполнить следующее:

```
$ git tag -l 'v1.8.5*'
v1.8.5
v1.8.5-gc0
v1.8.5-gc1
v1.8.5-gc2
```

```
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Создание меток

Git использует два основных типа меток: легковесные и аннотированные.

Легковесная метка — это что-то весьма похожее на ветку, которая не меняется — это просто указатель на определённый коммит.

А вот аннотированные метки хранятся в базе данных Git'a как полноценные объекты. Они имеют контрольную сумму, содержат имя поставившего метку, e-mail и дату, имеют комментарий и могут быть подписаны и проверены с помощью GNU Privacy Guard (GPG). Обычно рекомендуется создавать аннотированные метки, чтобы иметь всю перечисленную информацию; но если вы хотите сделать временную метку или по какой-то причине не хотите сохранять остальную информацию, то для этого годятся и легковесные метки.

Аннотированные метки

Создание аннотированной метки в Git'e выполняется легко. Самый простой способ это указать -а при выполнении команды `tag`:

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

Опция `-m` задаёт сообщение метки, которое будет храниться вместе с меткой. Если не указать сообщение для аннотированной метки, Git запустит редактор, чтобы вы смогли его ввести.

Вы можете посмотреть данные метки вместе с коммитом, который был помечен, с помощью команды `git show`:

```
$ git show v1.4
tag v1.4
```

```

Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

changed the version number

```

Она показывает информацию о выставившем метку, дату отметки коммита и аннотирующее сообщение перед информацией о коммите.

Легковесные метки

Легковесная метка — это ещё один способ отметки коммитов. В сущности, это контрольная сумма коммита, сохранённая в файл — больше никакой информации не хранится. Для создания легковесной метки не передавайте опций `-a`, `-s` и `-m`:

```

$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5

```

На этот раз при выполнении `git show` на этой метке вы не увидите дополнительной информации. Команда просто покажет помеченный коммит:

```

$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

changed the version number

```

Выставление меток позже

Также возможно помечать уже пройденные коммиты. Предположим, что история коммитов выглядит следующим образом:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbe added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fce02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Теперь предположим, что вы забыли отметить версию проекта v1.2, которая была там, где находится коммит “updated rakefile”. Вы можете добавить метку и позже. Для отметки коммита укажите его контрольную сумму (или её часть) в конце команды:

```
$ git tag -a v1.2 9fce02d
```

Можете проверить, что коммит теперь отмечен:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fce02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700
```

```
updated rakefile
...
```

Обмен метками

По умолчанию, команда `git push` не отправляет метки на удалённые серверы. Необходимо явно отправить (`push`) метки на общий сервер после того, как вы их создали. Это делается так же, как и добавление в репозиторий для совместного использования удалённых веток — нужно выполнить `git push origin [имя метки]`.

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]           v1.5 -> v1.5
```

Если у вас есть много меток, которые хотелось бы отправить все за один раз, можно использовать опцию `--tags` для команды `git push`. В таком случае все ваши метки отправятся на удалённый сервер (если только их уже там нет).

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]           v1.4 -> v1.4
 * [new tag]           v1.4-lw -> v1.4-lw
```

Теперь, если кто-то склонирует (`clone`) или выполнит `git pull` из вашего репозитория, то он получит вдобавок к остальному и ваши метки.

Переход на метку

В действительности вы не можете переходить на метки в Git, поскольку они не могут быть перемещены. Если вы хотите установить

версию вашего репозитория в рабочую директорию, которая выглядит, как определенная метка, вы можете создать новую ветку с определенной меткой:

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

Конечно, если вы так делаете и коммите, ваша ветка `version2` будет немного отличаться от вашей метки `v2.0.0`, поскольку она будет двигаться вперед с новыми изменениями, так что будьте осторожны.

Псевдонимы в Git

Прежде, чем закончить эту главу по основам Git, рассмотрим ещё одну маленькую хитрость, которая поможет сделать использование Git'a проще, легче, и более привычным: псевдонимы (aliases). Мы не будем ссылаться на них дальше или предполагать, что вы будете пользоваться ими по ходу чтения книги, но вам лучше было бы знать, как их использовать.

Git не будет пытаться сделать вывод о том, какую команду вы хотели ввести, если вы ввели её неполностью. Если вы не хотите печатать каждую команду для Git'a целиком, вы легко можете настроить псевдонимы (alias) для любой команды с помощью `git config`. Вот несколько примеров псевдонимов, которые вы, возможно, захотите задать:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

Это означает, что, например, вместо ввода `git commit`, вам достаточно набрать только `git ci`. По мере освоения Git'a вам, вероятно, придётся часто пользоваться и другими командами. В этом случае без колебаний создавайте новые псевдонимы.

Такой способ может также быть полезен для создания команд, которые, как вы думаете, должны существовать. Например, чтобы исправить неудобство, с которым мы столкнулись при исключении

файла из индекса, можно добавить в Git свой собственный псевдоним `unstage`:

```
$ git config --global alias.unstage 'reset HEAD --'
```

Это делает эквивалентными следующие две команды:

```
$ git unstage fileA
$ git reset HEAD fileA
```

Такой вариант кажется немного более понятным. Также, обычно, добавляют команду `last` следующим образом:

```
$ git config --global alias.last 'log -1 HEAD'
```

Таким образом, можно легко просмотреть последний коммит:

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    test for current head

Signed-off-by: Scott Chacon <schacon@example.com>
```

Можно сказать, что Git просто заменяет эти команды на созданные вами псевдонимы (`alias`). Однако, возможно, вы захотите выполнить внешнюю команду, а не подкоманду Git'a. В этом случае, следует начать команду с символа `!`. Это полезно, если вы пишете свои утилиты для работы с Git-репозиторием. Продемонстрируем этот случай на примере создания псевдонима `git visual` для запуска `gitk`:

```
$ git config --global alias.visual "!gitk"
```

Заключение

Теперь вы умеете выполнять все базовые локальные операции с Git'ом: создавать или клонировать репозиторий, вносить изменения, индексировать и фиксировать эти изменения, а также просматривать историю всех изменений в репозитории. Дальше мы рассмотрим киллер-фичу Git'a — его модель ветвлений.

3

Ветвление в Git

Почти каждая система контроля версий (СКВ) в какой-то форме поддерживает ветвление. Используя ветвление, Вы отклоняетесь от основной линии разработки и продолжаете работу независимо от нее, не вмешиваясь в основную линию. Во многих СКВ создание веток очень затратный процесс, часто требующий создания новой копии директории, что может занять много времени для большого проекта.

Некоторые люди, говоря о модели ветвления Git, называют ее “киллер-фича”, что выгодно выделяет Git на фоне остальных СКВ. Что в ней такого особенного? Ветвление Git очень легковесно. Операция создания ветки выполняется почти мгновенно, переключение между ветками туда-сюда, обычно, также быстро. В отличии от многих других СКВ, Git поощряет процесс работы, при котором ветвление и слияние выполняется часто, даже по несколько раз в день. Понимание и владение этой функциональностью дает Вам уникальный и мощный инструмент, который может полностью изменить привычный Вам процесс разработки.

О ветвлении в двух словах

Для четкого понимания механизма ветвлений, необходимо вернуться назад и изучить то, как Git хранит данные.

Как вы можете помнить из [Chapter 1](#), Git не хранит данные в виде последовательности изменений, он использует набор снимков ([snapshot](#)).

Когда вы делаете коммит, Git сохраняет его в виде объекта, который содержит указатель на снимок ([snapshot](#)) подготовленных данных. Этот объект так же содержит имя автора и email, сообщение и указатель на коммит или коммиты непосредственно предшествующие данному (его родителей): отсутствие родителя для первоначального

коммита, один родитель для обычного коммита, и несколько родителей для результатов слияния веток.

Представьте себе каталог, который содержит дерево файлов, и вы подготавливаете их все вместе, а затем сохраняете в виде одного коммита. В процессе подготовки вычисляется контрольная сумма каждого файла (SHA-1 как мы узнали из [Chapter 1](#)), хранящая версию файла в репозитории Git (Git ссылается на них), затем эти контрольные суммы добавляются в область подготовленных файлов:

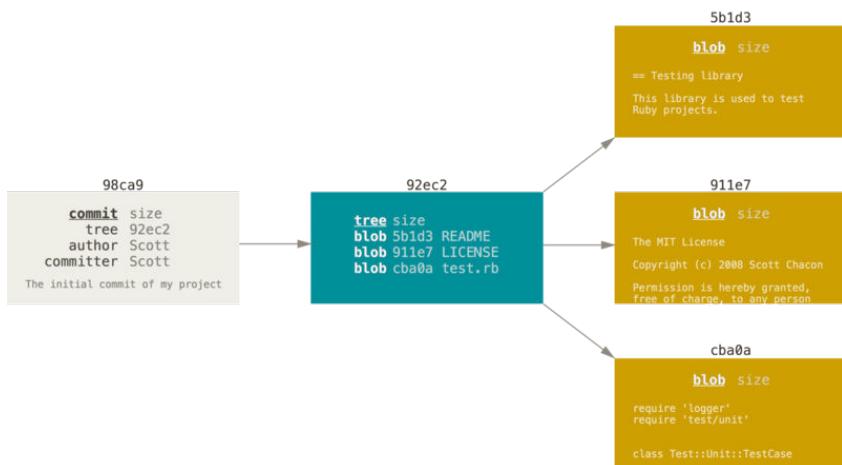
```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

Когда вы создаете коммит командой `git commit`, Git вычисляет контрольные суммы каждого подкаталога (в нашем случае, только основной каталог проекта) и сохраняет эти объекты дерева в репозитории. Затем Git создает объект коммита с метаданными и указателем на основное дерево проекта для возможности воссоздать этот снимок (snapshot) в случае необходимости.

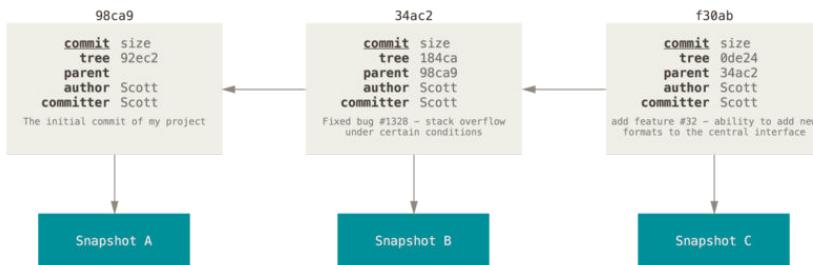
Ваш репозиторий Git теперь хранит пять объектов: блоб (blob) для содержимого каждого файла, содержимое каталога в виде дерева с указателями на блобы сохраненных файлов, сам коммит с указателем на основное дерево, метаданные коммита.

FIGURE 3-1

Коммит и его дерево



Если вы сделаете изменения и еще один коммит, тогда следующий коммит сохранит указатель на коммит, предшествующий ему.

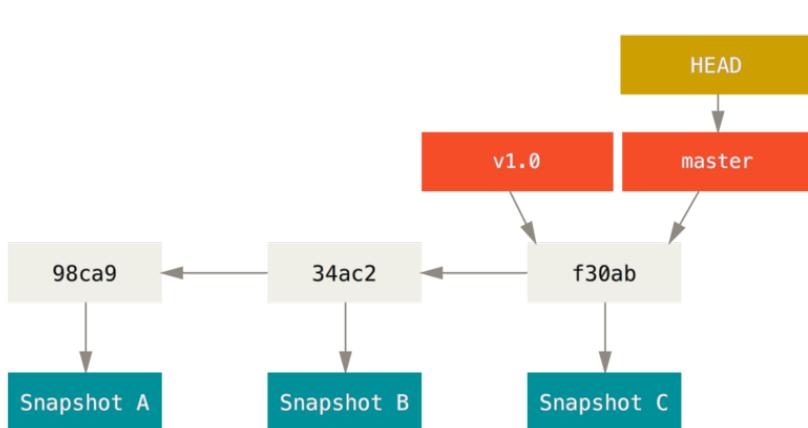
**FIGURE 3-2**

Коммит и его родители

Ветка (branch) в Git — это легко перемещаемый указатель на один из этих коммитов. Имя основной ветки по умолчанию в Git — master.

Когда вы делаете коммиты, то получаете основную ветку, указывающую на ваш последний коммит. Каждый коммит автоматическидвигает этот указатель вперед.

Ветка “master” в Git — это не специальная ветка. Она точно такая же, как и все остальные ветки. Она существует почти во всех репозиториях только лишь потому, что ее создает команда `git init`, а большинство людей не меняют ее название.

**FIGURE 3-3**

Ветка и история коммитов

Создание новой ветки

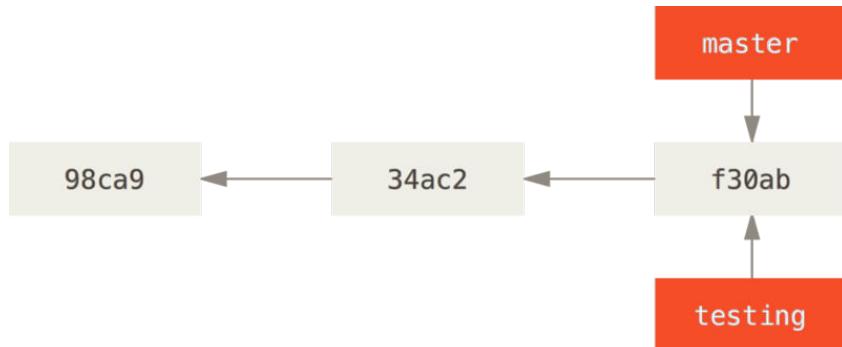
Что же на самом деле происходит, когда вы создаете ветку? Всего лишь создается новый указатель для дальнейшего перемещения. Допустим вы хотите создать новую ветку с именем “testing”. Вы можете это сделать командой `git branch`:

```
$ git branch testing
```

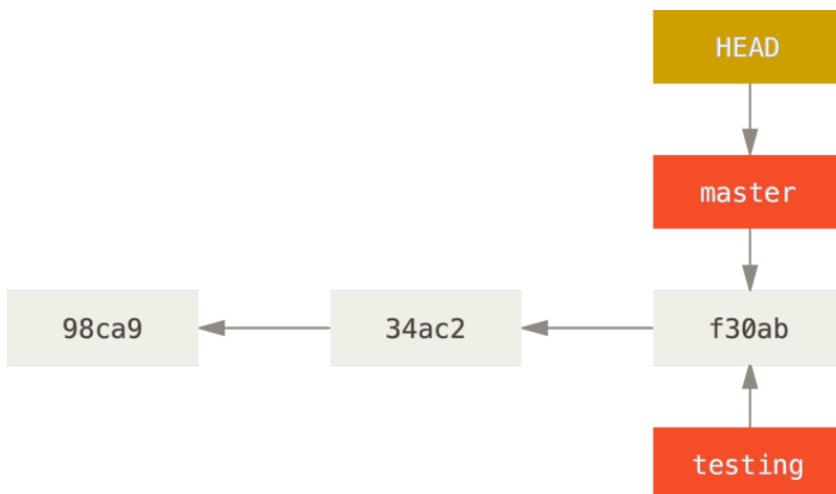
В результате создается новый указатель на тот же самый коммит, в котором вы находитесь.

FIGURE 3-4

Две ветки
указывают на одну
и ту же
последовательность
коммитов



Как Git определяет, в какой ветке вы находитесь? Он хранит специальный указатель HEAD. Имейте ввиду, что в Git концепция HEAD значительно отличается от других систем контроля версий, которые вы могли использовать раньше (Subversion или CVS). В Git это указатель на локальную ветку, в которой вы находитесь. В нашем случае мы все еще находимся в ветке “master”. Команда `git branch` только *создает* новую ветку. Переключения не происходит.

**FIGURE 3-5**

HEAD указывает на ветку

Вы можете легко это увидеть при помощи простой команды `git log`. Она покажет вам, куда указывают указатели веток. Эта опция называется `--decorate`.

```
$ git log --oneline --decorate
f30ab (HEAD, master, testing) add feature #32 - ability to add new
34ac2 fixed bug #1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

Видны ветки “master” и “testing”, которые указывают на коммит f30ab.

Переключение веток

Чтобы переключиться на существующую ветку, выполните команду `git checkout`. Давайте переключимся на ветку “testing”:

```
$ git checkout testing
```

В результате указатель HEAD переместится на ветку testing.

FIGURE 3-6

HEAD указывает на текущую ветку

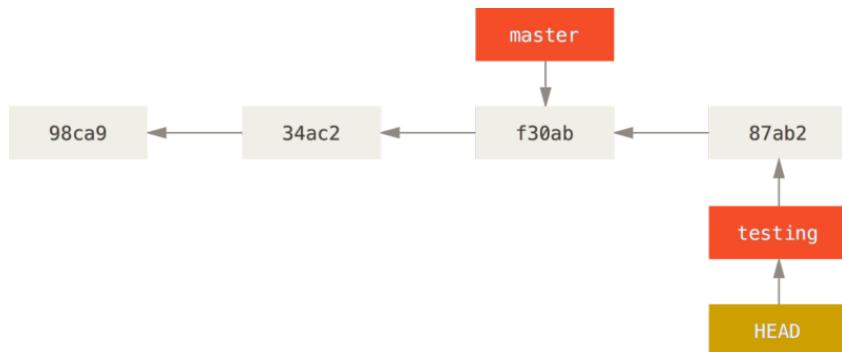


Какой в этом смысл? Давайте сделаем еще один коммит:

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

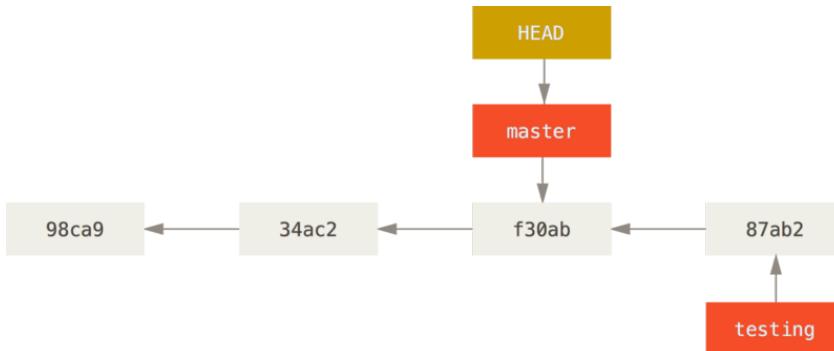
FIGURE 3-7

Указатель на ветку HEAD переместился вперед после коммита



Это интересно, потому что указатель на вашу ветку “testing” переместился вперед, а “master” все еще указывает на тот коммит, где вы были в момент выполнения команды `git checkout` для переключения веток. Давайте переключимся назад на ветку “master”:

```
$ git checkout master
```

**FIGURE 3-8**

HEAD
перемещается
когда вы делаете
checkout

Эта команда сделала две вещи. Она переместила указатель HEAD назад на ветку “master” и вернула файлы в рабочем каталоге в то состояние, которое было сохранено в снимке (snapshot), на который указывает ветка. Это также означает, что все изменения, вносимые с этого момента, будут отнесены к старой версии проекта. Другими словами, откатилась вся работа, выполненная в ветке “testing”, а вы можете продолжать в другом направлении.

ПЕРЕКЛЮЧЕНИЕ ВЕТОК МЕНЯЕТ ФАЙЛЫ В РАБОЧЕМ КАТАЛОГЕ

Важно запомнить, что когда вы переключаете ветки в Git, файлы в рабочем каталоге меняются. Если вы переключаетесь на старую ветку, то рабочий каталог будет выглядеть так же, как выглядел на момент последнего коммита в ту ветку. Если Git по каким-то причинам не может этого сделать — он не позволит вам переключиться.

Давайте сделаем еще несколько изменений и очередной коммит:

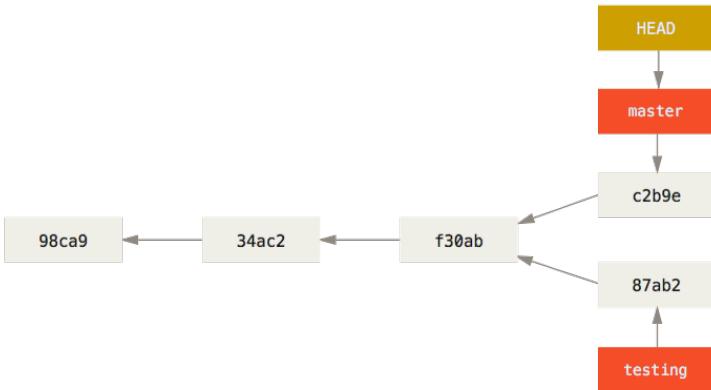
```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

Теперь история вашего проекта разделилась (см **Figure 3-9**). Вы создали ветку, переключились в нее, поработали, а затем вернулись в основную ветку и поработали в ней. Эти изменения изолированы друг от друга: вы можете свободно переключаться туда и обратно, а когда

будете готовы — слить их вместе. И все это делается простыми командами: `branch`, `checkout` и `commit`.

FIGURE 3-9

Разветвленная история



Все это вы можете увидеть при помощи команды `git log`. Команда `git log --oneline --decorate --graph --all` выдаст историю ваших коммитов и покажет, где находятся указатели ваших веток, и как ветвилась история проекта.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Создание и удаление веток совершенно не затратно, так как ветка в Git — это всего лишь файл, содержащий 40 символов контрольной суммы SHA-1 того коммита, на который он указывает. Создание новой ветки совершенно быстро и просто — это всего лишь запись 41 байта в файл (40 знаков и перевод строки).

Это совершенно отличает Git от ветвления в большинстве более старых систем контроля версий, где все файлы проекта копируются в

другой подкаталог. Там ветвление для проектов разного размера может занять от секунд до минут. В Git ветвление всегда мгновенное. Также, поскольку при коммите мы сохраняем указатель на родительский коммит, найти подходящую базу для слияния в основном очень просто, и это делается для нас автоматически. Эти возможности побуждают разработчиков чаще создавать и использовать ветки.

Давайте посмотрим, почему и вам имеет смысл делать так же.

Основы ветвления и слияния

Давайте рассмотрим простой пример рабочего процесса, который может быть полезен в вашем проекте. Ваша работа построена так:

1. Вы работаете над сайтом.
2. Вы создаете ветку для новой статьи, которую вы пишете.
3. Вы работаете в этой ветке.

В этот момент вы получаете сообщение, что обнаружена критическая ошибка, требующая скорейшего исправления. Ваши действия:

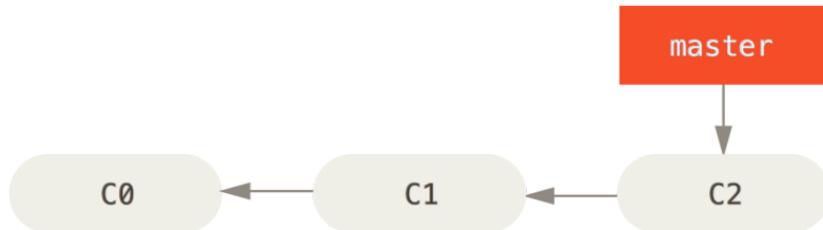
1. Переключиться на основную ветку.
2. Создать ветку для добавления исправления.
3. После тестирования слить ветку содержащую исправление с основной веткой.
4. Переключиться назад в ту ветку где вы пишите статью и продолжить работать.

Основы ветвления

Предположим, вы работаете над проектом и уже имеете несколько коммитов.

FIGURE 3-10

*Простая история
коммитов*



Вы решаете, что теперь вы будете заниматься проблемой #53 из вашей системы отслеживания ошибок. Чтобы создать ветку и сразу переключиться на нее, можно выполнить команду `git checkout` с параметром `-b`:

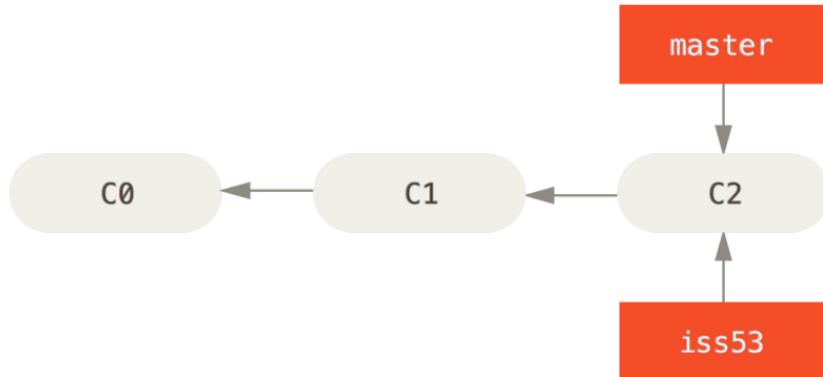
```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Это тоже самое что и:

```
$ git branch iss53
$ git checkout iss53
```

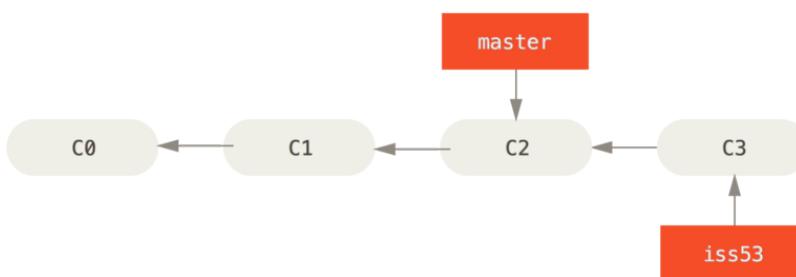
FIGURE 3-11

*Создание нового
указателя ветки*



Вы работаете над своим сайтом и делаете коммиты. Это приводит к тому, что ветка `iss53` движется вперед, так как вы переключились на нее ранее (HEAD указывает на нее).

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```

**FIGURE 3-12**

Ветка `iss53`
двигается вперед

Тут вы получаете сообщение об обнаружении уязвимости на вашем сайте, которую нужно немедленно устраниТЬ. Благодаря Git, не требуется размещать это исправление вместе с тем, что вы сделали в `iss53`. Вам даже не придется прилагать усилий, чтобы откатить все эти изменения для начала работы над исправлением. Все, что вам нужно — переключиться на ветку `master`.

Но перед тем как сделать это — имейте в виду, что если ваш рабочий каталог либо область подготовленных файлов содержат изменения, не попавшие в коммит и конфликтующие с веткой, на которую вы хотите переключиться, то Git не позволит вам переключить ветки. Лучше всего переключаться из чистого рабочего состояния проекта. Есть способы обойти это (спрятать (`stash`) или исправить (`amend`) коммиты), но об этом мы поговорим позже в главе “Прибережение и очистка”. Теперь предположим, что вы зафиксировали все свои изменения и можете переключиться на основную ветку:

```
$ git checkout master
Switched to branch 'master'
```

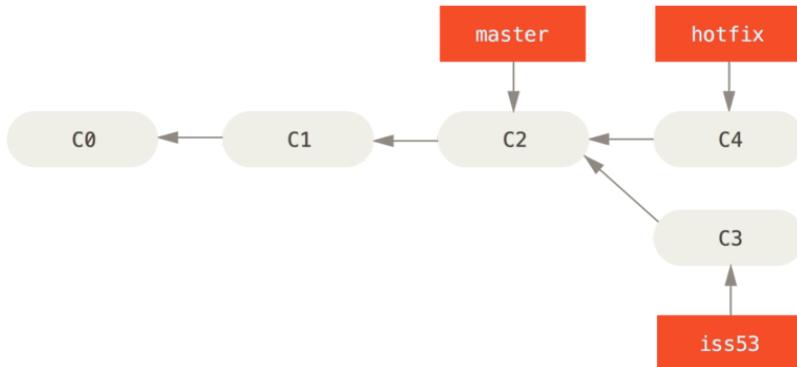
С этого момента ваш рабочий каталог имеет точно такой же вид, какой был перед началом работы над проблемой #53. Теперь вы можете сосредоточиться на работе над исправлением. Важно запомнить: когда вы переключаете ветки, Git возвращает состояние рабочего каталога к тому виду, какой он имел в момент последнего коммита в эту ветку. Он добавляет, удаляет и изменяет файлы автоматически, чтобы состояние рабочего каталога соответствовало тому, когда был сделан последний коммит.

Теперь вы можете перейти к написанию исправления. Давайте создадим новую ветку для исправления, в которой будем работать, пока не закончим исправление.

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
 1 file changed, 2 insertions(+)
```

FIGURE 3-13

Ветка `hotfix` основана на ветке `master`



Вы можете прогнать тесты, чтобы убедиться, что ваше исправление делает именно то, что нужно. И если это так — выполнить слияние (merge) с основной веткой для включения в продукт. Это делается командой `git merge`:

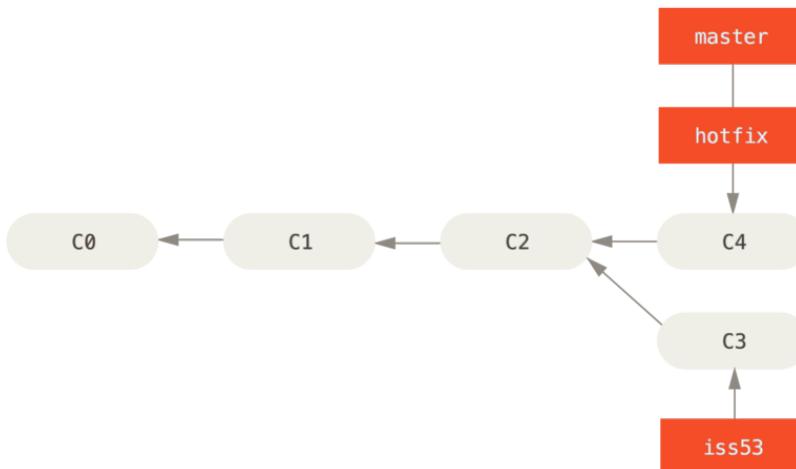
```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
```

```
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Заметили фразу “fast-forward” в этом слиянии? Из-за того, что коммит, на который указывала ветка, которую вы слили, был прямым потомком того коммита, на котором вы находились, Git просто переместил указатель ветки вперед. Другими словами, если коммит сливается с тем, до которого можно добраться, двигаясь по истории прямо, Git упрощает слияние, просто перенося указатель метки вперед (так как нет разветвления в работе). Это называется ``fast-forward`` (перемотка). Теперь ваши изменения — в снимке (snapshot) коммита, на который указывает ветка `master`, и исправления продукта можно внедрять.

FIGURE 3-14

master перемотан
до *hotfix*



После внедрения вашего архиважного исправления вы готовы вернуться к работе над тем, что были вынуждены отложить. Как бы то ни было, сначала нужно удалить ветку `hotfix`, потому что она больше не нужна — ветка `master` указывает на то же самое место. Для удаления ветки выполните команду `git branch` с параметром `-d`:

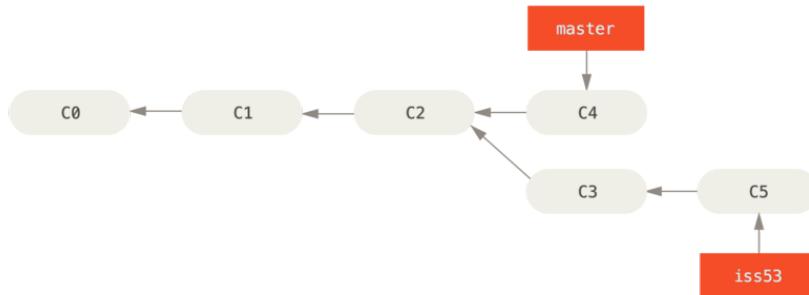
```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Теперь вы можете переключить ветку и вернуться к работе над своей проблемой #53:

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

FIGURE 3-15

Продолжение работы над iss53



Стоит обратить внимание на то, что все изменения из ветки hotfix не включены в вашу ветку iss53. Если их нужно включить, вы можете влить ветку master в вашу ветку iss53 командой `git merge master`, или же вы можете отложить слияние этих изменений до завершения работы, и затем влить ветку iss53 в master.

Основы слияния

Предположим, вы решили, что работа по проблеме #53 закончена, и ее можно влить в ветку master. Для этого нужно выполнить слияние ветки iss53 точно так же, как вы делали это с веткой hotfix ранее. Все что нужно сделать — переключиться на ветку, в которую вы хотите включить изменения, и выполнить команду `git merge`:

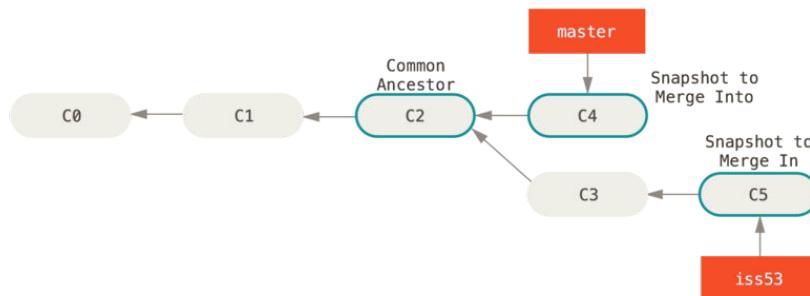
```
$ git checkout master
Switched to branch 'master'
```

```
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
 1 file changed, 1 insertion(+)
```

Результат этой операции отличается от результата слияния ветки `hotfix`. В данном случае, у нас история работы разветвилась. Так как коммит, на котором мы находимся, не является прямым потомком ветки, с которой мы выполняем слияние, Git придется немного потрудиться. В этом случае Git выполняет простое трехстороннее слияние двух снимков (snapshot) сливаемых веток и общего для двух веток родительского снимка.

FIGURE 3-16

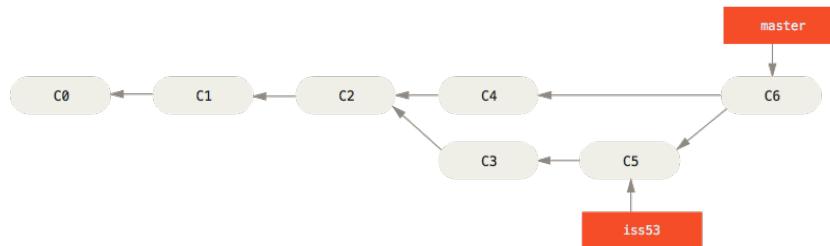
Использование трех снимков при слиянии



Вместо того, чтобы просто передвинуть указатель ветки вперед, Git создает новый снимок-результат трехстороннего слияния, а затем автоматически делает коммит. Этот особый коммит называют коммитом слияния, так как у него более одного предка.

FIGURE 3-17

Коммит слияния



Стоит отметить, что Git сам определяет наилучшего общего предка, подходящего как база для слияния; это отличает его от более старых инструментов, таких как CVS или Subversion (до версии 1.5), где разработчикам, выполнявшим слияние, приходилось самими находить лучшую базу. Это безумно упрощает слияние в Git по сравнению с указанными системами.

Теперь, когда работа влита, ветка `iss53` больше не нужна. Вы можете закрыть вопрос в системе отслеживания ошибок и удалить ветку:

```
$ git branch -d iss53
```

Основные конфликты слияния

Иногда процесс не проходит гладко. Если вы изменили одну и ту же часть одного и того же файла по-разному в двух объединяемых ветках, Git не сможет их чисто объединить. Если ваше исправление ошибки #53 потребовало изменить ту же часть файла, что и `hotfix`, вы получите примерно такое сообщение о конфликте слияния:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git не создал коммит слияния автоматически. Он остановил процесс до тех пор, пока вы не разрешите конфликт. Чтобы в любой

момент после появления конфликта увидеть, какие файлы не объединены, вы можете запустить `git status`:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:    index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Все, где есть неразрешенные конфликты слияния, перечисляется как неслитое. Git добавляет в конфликтующие файлы стандартные пометки разрешения конфликтов, чтобы вы могли вручную открыть их и разрешить конфликты. В вашем файле появился раздел, выглядящий примерно так:

```
<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>> iss53:index.html
```

Это означает, что версия из HEAD (вашей ветки `master`, поскольку именно ее вы выгрузили, запустив команду слияния) — это верхняя часть блока (все, что над `=====`), а версия из вашей ветки `iss53` представлена в нижней части. Чтобы разрешить конфликт, придется выбрать одну из сторон, либо объединить содержимое по-своему. Например, вы можете разрешить конфликт, заменив весь блок этим:

```
<div id="footer">
  please contact us at email.support@github.com
</div>
```

В этом разрешении есть немного от каждой части, а строки `<<<<<`, `=====` и `>>>>>` совсем убраны. Разрешив каждый конфликт во всех файлах, запустите `git add` для каждого файла,

чтобы отметить конфликт как решенный. Подготовка (staging) файла помечает его для Git как разрешенный конфликт.

Если вы хотите использовать графический инструмент для разрешения конфликтов, можно запустить `git mergetool`, что откроет соответствующее визуальное средство, которое проведет вас по всем конфликтам:

```
$ git mergetool
This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge
Merging:
index.html

Normal merge conflict for 'index.html':
{local}: modified file
{remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

Если вы хотите использовать средство слияния не по умолчанию (в данном случае Git выбрал opendiff, поскольку команда запускалась на Mac), список всех поддерживаемых инструментов представлен вверху после фразы “one of the following tools.” Просто введите название инструмента, который нужно использовать.

Описание расширенных средств разрешения сложных конфликтов слияния мы приводим в разделе “Продвинутое слияние”.

После выхода из средства слияния Git спрашивает, успешно ли слияние. Если вы утвердительно ответите скрипту, он подготовит (stage) файл, чтобы отметить его как разрешенный. Теперь можно снова запустить `git status`, чтобы убедиться, что все конфликты разрешены:

```
$ git status
On branch master
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

Changes to be committed:
```

```
modified: index.html
```

Если это вас устраивает, и вы убедились, что все, где были конфликты, подготовлено (staged), можете ввести `git commit`, чтобы завершить коммит слияния. Комментарий к коммиту по умолчанию выглядит примерно так:

```
Merge branch 'iss53'

Conflicts:
  index.html
#
# 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. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#       modified: index.html
#
```

Вы можете дополнить это сообщение подробностями того, как были разрешены конфликты, если считаете, что это поможет другим в будущем разобраться в данном слиянии, если это не очевидно: что вы сделали и почему.

Управление ветками

Теперь, когда вы уже попробовали создавать, объединять и удалять ветки, пора познакомиться с некоторыми инструментами для управления ветками, которые вам пригодятся, когда вы начнёте использовать ветки постоянно.

Команда `git branch` делает несколько больше, чем просто создаёт и удаляет ветки. При запуске без параметров, вы получите простой список имеющихся у вас веток:

```
$ git branch
  iss53
* master
  testing
```

Обратите внимание на символ *, стоящий перед веткой `master`: он указывает на ветку, на которой вы находитесь в настоящий момент (т.е. ветку, на которую указывает `HEAD`). Это означает, что если вы сейчас выполните коммит, ветка `master` переместится вперёд в соответствии с вашими последними изменениями. Чтобы посмотреть последний коммит на каждой из веток, выполните команду `git branch -v`:

```
$ git branch -v
  iss53  93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

Ещё одна полезная возможность для выяснения состояния веток состоит в том, чтобы оставить в этом списке только те ветки, которые вы слили (или не слили) в ветку, на которой сейчас находитесь. Для этих целей в Git'е есть опции `--merged` и `--no-merged`. Чтобы посмотреть те ветки, которые вы уже слили с текущей, можете выполнить команду `git branch --merged`:

```
$ git branch --merged
  iss53
* master
```

Из-за того что вы ранее слили `iss53`, она присутствует в этом списке. Те ветки из этого списка, перед которыми нет символа *, можно смело удалять командой `git branch -d`; наработки из этих веток уже включены в другую ветку, так что ничего не потеряется.

Чтобы увидеть все ветки, содержащие наработки, которые вы пока ещё не слили в текущую ветку, выполните команду `git branch --no-merged`:

```
$ git branch --no-merged
  testing
```

Вы увидите оставшуюся ветку. Так как она содержит ещё не слитые наработки, попытка удалить её командой `git branch -d` приведет к ошибке:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Если вы действительно хотите удалить ветку и потерять наработки, вы можете сделать это при помощи опции `-D`, как указано в подсказке.

Branching Workflows

Now that you have the basics of branching and merging down, what can or should you do with them? In this section, we'll cover some common workflows that this lightweight branching makes possible, so you can decide if you would like to incorporate it into your own development cycle.

Long-Running Branches

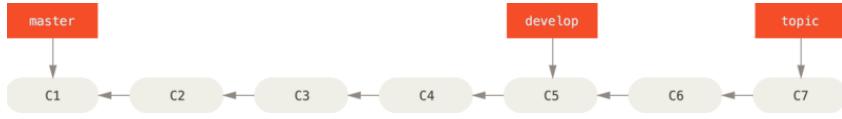
Because Git uses a simple three-way merge, merging from one branch into another multiple times over a long period is generally easy to do. This means you can have several branches that are always open and that you use for different stages of your development cycle; you can merge regularly from some of them into others.

Many Git developers have a workflow that embraces this approach, such as having only code that is entirely stable in their `master` branch – possibly only code that has been or will be released. They have another parallel branch named `develop` or `next` that they work from or use to test stability – it isn't necessarily always stable, but whenever it gets to a stable state, it can be merged into `master`. It's used to pull in topic branches (short-lived branches, like your earlier `iss53` branch) when they're ready, to make sure they pass all the tests and don't introduce bugs.

In reality, we're talking about pointers moving up the line of commits you're making. The stable branches are farther down the line in your commit history, and the bleeding-edge branches are farther up the history.

FIGURE 3-18

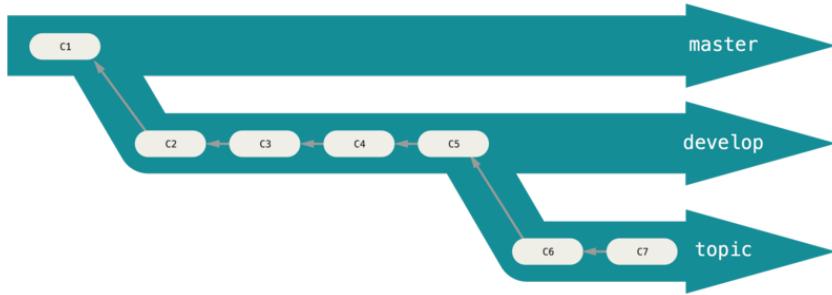
A linear view of progressive-stability branching



It's generally easier to think about them as work silos, where sets of commits graduate to a more stable silo when they're fully tested.

FIGURE 3-19

A “silo” view of progressive-stability branching



You can keep doing this for several levels of stability. Some larger projects also have a `proposed` or `pu` (proposed updates) branch that has integrated branches that may not be ready to go into the `next` or `master` branch. The idea is that your branches are at various levels of stability; when they reach a more stable level, they're merged into the branch above them. Again, having multiple long-running branches isn't necessary, but it's often helpful, especially when you're dealing with very large or complex projects.

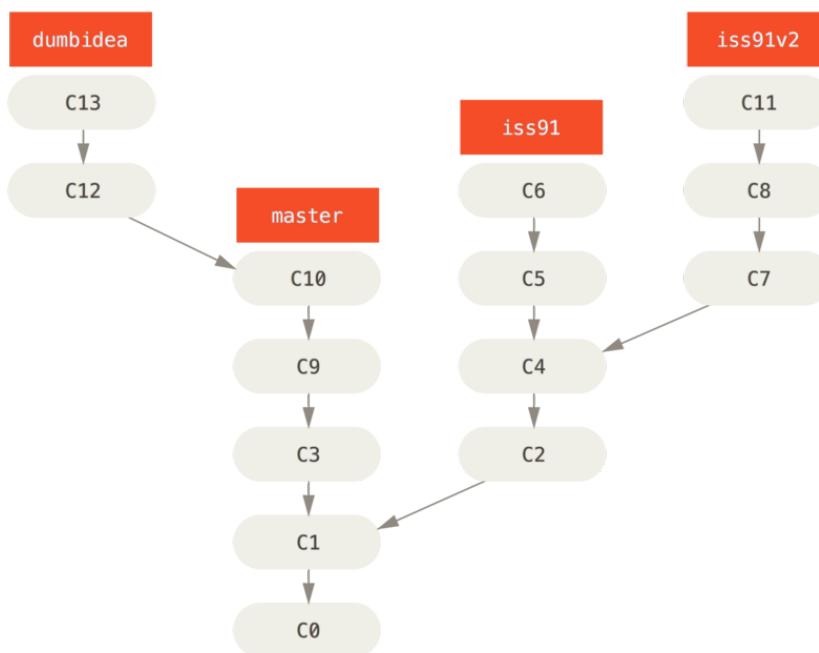
Topic Branches

Topic branches, however, are useful in projects of any size. A topic branch is a short-lived branch that you create and use for a single particular feature or related work. This is something you've likely never done with a VCS before because it's generally too expensive to create and merge branches. But in Git it's common to create, work on, merge, and delete branches several times a day.

You saw this in the last section with the `iss53` and `hotfix` branches you created. You did a few commits on them and deleted them directly after merging them into your main branch. This technique allows you to context-switch quickly and completely – because your work is separated into silos where all

the changes in that branch have to do with that topic, it's easier to see what has happened during code review and such. You can keep the changes there for minutes, days, or months, and merge them in when they're ready, regardless of the order in which they were created or worked on.

Consider an example of doing some work (on `master`), branching off for an issue (`iss91`), working on it for a bit, branching off the second branch to try another way of handling the same thing (`iss91v2`), going back to your `master` branch and working there for a while, and then branching off there to do some work that you're not sure is a good idea (`dumbidea` branch). Your commit history will look something like this:

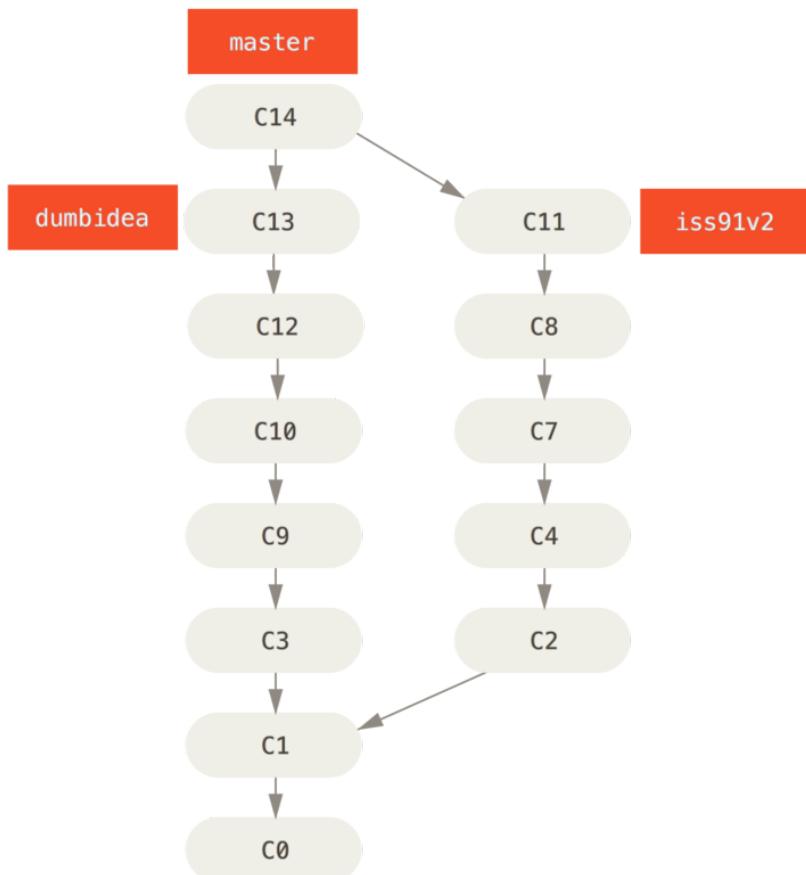
**FIGURE 3-20**

Multiple topic branches

Now, let's say you decide you like the second solution to your issue best (`iss91v2`); and you showed the `dumbidea` branch to your coworkers, and it turns out to be genius. You can throw away the original `iss91` branch (losing commits `C5` and `C6`) and merge in the other two. Your history then looks like this:

FIGURE 3-21

*History after
merging dumbidea
and iss91v2*



We will go into more detail about the various possible workflows for your Git project in **Chapter 5**, so before you decide which branching scheme your next project will use, be sure to read that chapter.

It's important to remember when you're doing all this that these branches are completely local. When you're branching and merging, everything is being done only in your Git repository – no server communication is happening.

Удалённые ветки

Удалённые ветки — это ссылки (pointers) на состояние веток в ваших удалённых репозиториях. Это локальные ветки, которые нельзя перемещать; они двигаются автоматически всякий раз, когда вы

осуществляете связь по сети. Удалённые ветки действуют как закладки для напоминания о том, где ветки в удалённых репозиториях находились во время последнего подключения к ним.

Они выглядят как (имя удал. репоз.)/(ветка). Например, если вы хотите посмотреть, как выглядела ветка `master` на сервере `origin` во время последнего соединения с ним, проверьте ветку `origin/master`. Если вы с партнёром работали над одной проблемой, и он выложил ветку `iss53`, у вас может быть своя локальная ветка `iss53`; но та ветка на сервере будет указывать на коммит в `origin/iss53`.

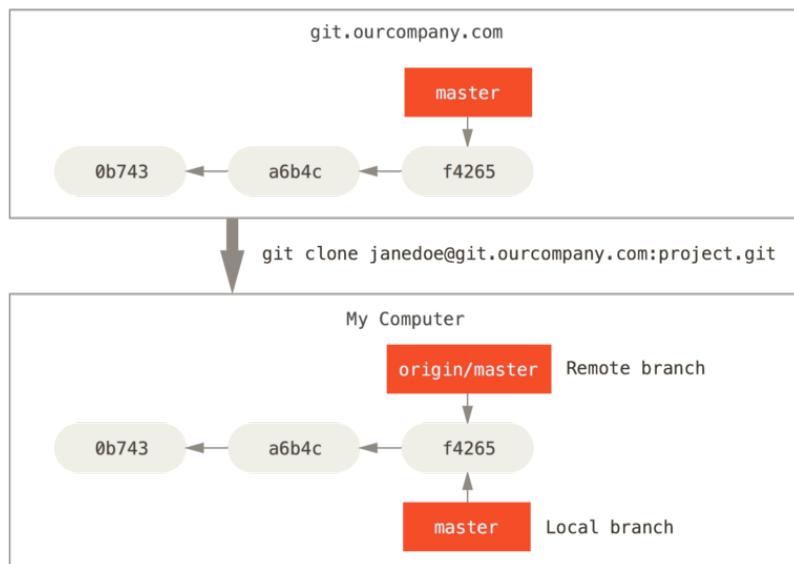
Всё это, возможно, сбивает с толку, поэтому давайте рассмотрим пример. Скажем, у вас в сети есть свой Git-сервер на `git.ourcompany.com`. Если вы с него что-то склонируете, Git-команда `clone` автоматически назовёт его `origin`, заберёт оттуда все данные, создаст указатель на то, на что там указывает ветка `master`, и назовёт его локально `origin/master`. Git также сделает вам вашу собственную локальную ветку `master`, которая будет начинаться там же, где и ветка `master` в `origin`, так что вам будет с чем работать.

“ORIGIN” — ЭТО НЕ СПЕЦИАЛЬНОЕ НАЗВАНИЕ

Подобное название ветки “`master`” не имеет какого-либо назначения в Git, также и название “`origin`”. В то время как “`master`” — это название по умолчанию для исходной ветки, когда вы запускаете `git init`, по единственной причине, что широко используется, “`origin`” — это название по умолчанию для удалённой ветки, когда вы запускаете `git clone`. Если вы запустите `git clone -o booyah`, так вы будете иметь `booyah/master` как вашу удалённую ветку по умолчанию.

FIGURE 3-22

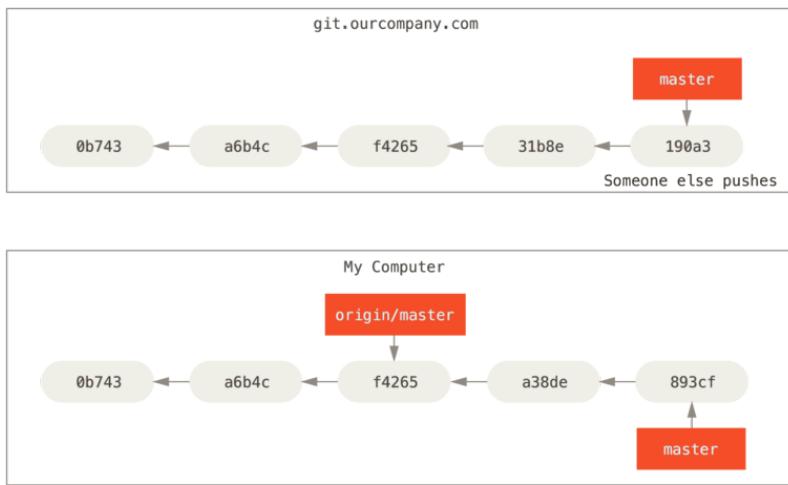
Серверный и локальный репозитории после клонирования



Если вы сделаете что-то в своей локальной ветке `master`, а тем временем кто-то ещё отправит (push) изменения на `git.ourcompany.com` и обновит там ветку `master`, то ваши истории продолжатся по-разному. Ещё, до тех пор, пока вы не свяжетесь с сервером `origin`, ваш указатель `origin/master` не будет сдвигаться.

FIGURE 3-23

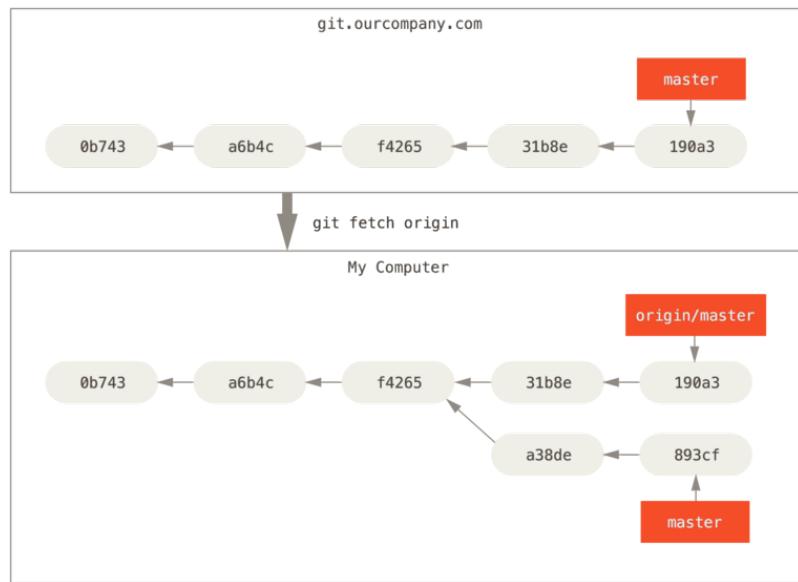
Локальная и удаленная работа может расходиться



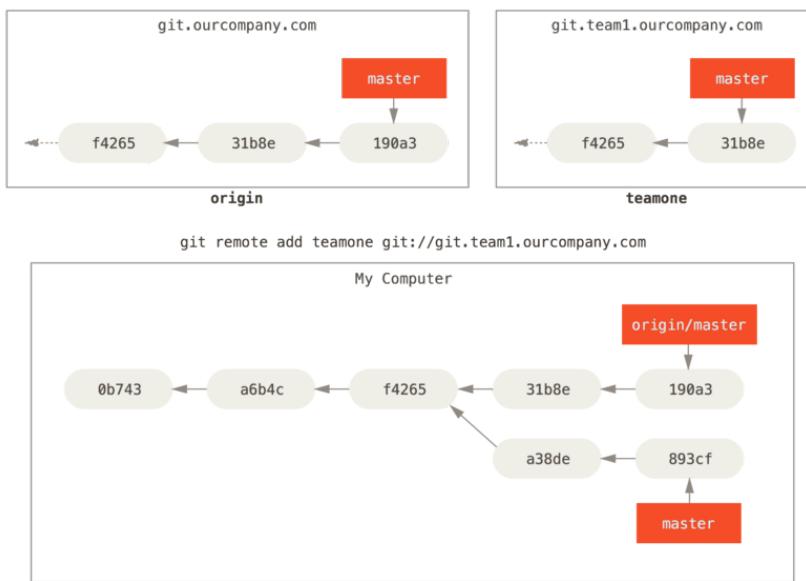
Для синхронизации вашей работы выполняется команда `git fetch origin`. Эта команда ищет, какому серверу соответствует “origin” (в нашем случае это `git.ourcompany.com`); извлекает оттуда все данные, которых у вас ещё нет, и обновляет ваше локальное хранилище данных; сдвигает указатель `origin/master` на новую позицию.

FIGURE 3-24

*git fetch
обновляет ваши
удаленные ссылки*



Чтобы продемонстрировать то, как будут выглядеть удалённые ветки в ситуации с несколькими удалёнными серверами, предположим, что у вас есть ещё один внутренний Git-сервер, который используется для разработки только одной из ваших команд разработчиков. Этот сервер находится на `git.team1.ouгcompany.com`. Вы можете добавить его в качестве новой удалённой ссылки на проект, над которым вы сейчас работаете с помощью команды `git remote add` так же, как было описано в [Chapter 2](#). Дайте этому удалённому серверу имя `teamone`, которое будет сокращением для полного URL.

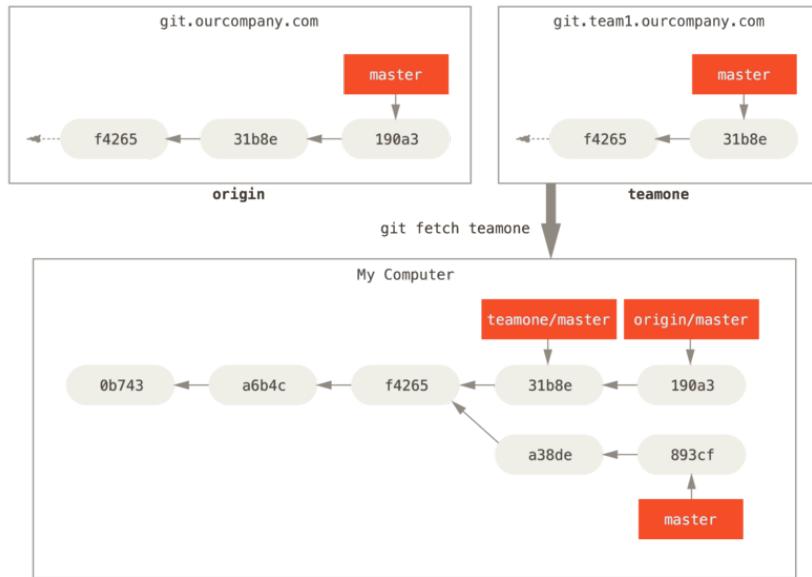
**FIGURE 3-25**

Добавление еще одного сервера в качестве удаленной ветки

Теперь можете выполнить `git fetch teamone`, чтобы извлечь всё, что есть на сервере и нет у вас. Так как в данный момент на этом сервере есть только часть данных, которые есть на сервере `origin`, Git не получает никаких данных, но выставляет удалённую ветку с именем `teamone/master`, которая указывает на тот же коммит, что и ветка `master` на сервере `teamone`.

FIGURE 3-26

*Удаленное
отслеживание ветки
teamone/master*



Отправка изменений

Когда вы хотите поделиться веткой с окружающими, вам необходимо отправить (push) её на удалённый сервер, на котором у вас есть права на запись. Ваши локальные ветки автоматически не синхронизируются с удалёнными серверами — вам нужно явно отправить те ветки, которыми вы хотите поделиться. Таким образом, вы можете использовать свои личные ветки для работы, которую вы не хотите показывать, и отправлять только те тематические ветки, над которыми вы хотите работать с кем-то совместно.

Если у вас есть ветка `serverfix`, над которой вы хотите работать с кем-то ещё, вы можете отправить её точно так же, как вы отправляли вашу первую ветку. Выполните `git push` (удал. сервер) (ветка):

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

Это в некотором роде сокращение. Git автоматически разворачивает имя ветки `serverfix` до `refs/heads/serverfix:refs/heads/serverfix`, что означает “возьми мою локальную ветку `serverfix` и обнови ей удалённую ветку `serverfix`.“ Мы подробно обсудим часть с `refs/heads/` в [Chapter 10](#), но обычно её можно опустить. Вы также можете выполнить `git push origin serverfix:serverfix` — произойдёт то же самое — здесь говорится “возьми мой `serverfix` и сделай его удалённым `serverfix`.“ Можно использовать этот формат для отправки локальной ветки в удалённую ветку с другим именем. Если вы не хотите, чтобы ветка называлась `serverfix` на удалённом сервере, то вместо предыдущей команды выполните `git push origin serverfix:awesomebranch`. Так ваша локальная ветка `serverfix` отправится в ветку `awesomebranch` удалённого проекта.

НЕ ВВОДИТЕ КАЖДЫЙ РАЗ СВОЙ ПАРОЛЬ

Если вы используете HTTPS URL для отправки изменений, Git-сервер спросит имя пользователя и пароль для аутентификации. По умолчанию вам будет предложено ввести в терминале эту информацию, чтобы сервер мог сказать, что вам разрешена отправка изменений.

Если вы не хотите каждый раз вводить ваши данные, когда вы отправляете изменения, вы можете установить “кэш учетных данных”. Проще всего просто держать их в памяти несколько минут, вы можете легко настроить с помощью команды `git config --global credential.helper cache`.

Для получения более подробной информации о различных вариантах кэша учетных данных, можно посмотреть на “Хранилище учетных данных”.

В следующий раз, когда один из ваших соавторов будет получать обновления с сервера, он получит ссылку на то, на что указывает `serverfix` на сервере, как удалённую ветку `origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

Важно отметить, что когда при получении данных у вас появляются новые удалённые ветки, вы не получаете автоматически для них локальных редактируемых копий. Другими словами, в нашем случае

вы не получите новую ветку `serverfix` — только указатель `origin/serverfix`, который вы не можете менять.

Чтобы слить эти наработки в свою текущую рабочую ветку, выполните `git merge origin/serverfix`. Если вам нужна своя собственная ветка `serverfix`, над которой вы сможете работать, то вы можете создать её на основе удалённой ветки:

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Это даст вам локальную ветку, на которой можно работать. Она будет начинаться там, где и `origin/serverfix`.

Отслеживание веток

Получение локальной ветки из удалённой ветки автоматически создаёт то, что называется “отслеживаемой веткой” (или иногда “upstream branch”). Отслеживаемые ветки — это локальные ветки, которые напрямую связаны с удалённой веткой. Если, находясь на отслеживаемой ветке, вы наберёте `git pull`, Git уже будет знать, с какого сервера получить все удалённые ссылки и сделает слияние с соответствующей удалённой веткой. Аналогично выполнение `git pull` на одной из таких веток, сначала получает все удалённые ссылки, а затем автоматически делает слияние с соответствующей удалённой веткой.

При клонировании репозитория, как правило, автоматически создаётся ветка `master`, которая отслеживает `origin/master`. Однако, вы можете настроить отслеживание и других веток, допустим если вы хотите, чтобы одни ветки отслеживались с другого удаленного репозитория или не хотите отслеживать ветку `master`. Простой пример, как это сделать, вы увидели только что — `git checkout -b [ветка] [удал. сервер]/[ветка]`. Существует общепринятая операция, которую `git` предоставляет, `--track`:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Чтобы настроить локальную ветку с именем, отличным от имени удалённой ветки, вы можете легко использовать первую версию с другим именем локальной ветки:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
```

Теперь ваша локальная ветка `sf` будет автоматически получать (`pull`) изменения из `origin/serverfix`.

Если у вас уже есть локальная ветка и вы хотите настроить ее на удаленную ветку, которую вы только получили, или хотите изменить `upstream`-ветку, которую вы отслеживаете, вы можете воспользоваться ключами `-u` или `--set-upstream-to` с командой `git branch`, чтобы явно установить ее в любое время.

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

СОКРАЩЕНИЕ UPSTREAM

Если у вас есть установленная отслеживаемая ветка, вы можете ссылаться на нее с помощью `@{upstream}` или `@{u}` сокращенно. Итак, если вы находитесь на `master`-ветке, а отслеживается `origin/master`, вы можете вызвать что-то вроде `git merge @{u}` вместо `git merge origin/master` если хотите.

Если вы хотите посмотреть какие отслеживаемые ветки у вас установлены, вы можете воспользоваться опцией `-vv` в команде `git branch`. Отобразиться список ваших локальных веток с дополнительной информацией, включая то, какая из веток отслеживается, и если локальная ветка опережает, отстает или равняется относительно основной ветки.

```
$ git branch -vv
iss53      7e424c3 [origin/iss53: ahead 2] forgot the brackets
master     1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
  testing   5ea463a trying something new
```

Итак, здесь мы можем видеть, что наша `iss53`-ветка отслеживает `origin/iss53` и она опережает на два изменения, означающее, что мы имеем два локальных коммита, которые не отправлены на сервер. Мы можем также увидеть, что наша `master`-ветка отслеживает `origin/master` и она в актуальном состоянии. Далее мы можем видеть, что наша `serverfix`-ветка отслеживает `server-fix-good`-ветку на нашем `teamone`-сервере и опережает на три коммита и отстает на один, означающее, что есть один коммит на сервере, который мы еще не слили и три локальных коммита, которые вы еще не отправили. В конце мы видим, что наша `testing`-ветка не отслеживает удаленную ветку.

Важно отметить, что эти цифры — только с каждого сервера, которые последний раз были извлечены. Эта команда не обращается к серверам, она говорит вам о том, что в кэше есть локальная информация с серверов. Если вы хотите полностью быть в курсе опережающих и отстающих коммитов, вам необходимо извлечь данные из всех ваших удаленных серверов перед запуском этой команды. Вы можете сделать нечто подобное: `$ git fetch --all; git branch -vv`

Получение изменений

Команда `git fetch` загрузит с сервера все изменения, которых у вас еще нет, но пока не будет изменять вашу рабочую директорию. Эта команда просто получает данные для вас и позволяет вам самостоятельно сделать слияние. Тем не менее, существует команда под названием `git pull`, которая является по существу командой `git fetch`, непосредственно за которой следует команда `git merge`, в большинстве случаев. Если у вас есть отслеживаемая ветка как показано в предыдущем разделе, либо она явно установлена или она содержится вследствие создания вами командами `clone` или `checkout`, `git pull` увидит, что сервер и ветка вашей текущей ветки отслеживается, извлечет с сервера и затем попытается объединить в удаленную ветку.

Обычно лучше просто явно использовать команды `fetch` и `merge`, поскольку магия `git pull` может часто сбивать с толку.

Удаление веток на удалённом сервере

Скажем, вы и ваши соавторы закончили с нововведением и слили его в ветку `master` на удалённом сервере (или в какую-то другую ветку, где хранится стабильный код). Вы можете удалить ветку на удалённом сервере, используя опцию `-delete` для `git push`. Если вы хотите удалить ветку `serverfix` на сервере, выполните следующее:

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]          serverfix
```

В основном всё, что делает эта строка, — удаляет указатель на сервере. Как правило, Git-сервер оставит данные на некоторое время, пока не запуститься сборщик мусора, итак, если ветка случайно была удалена, чаще всего ее легко восстановить.

Rebasing

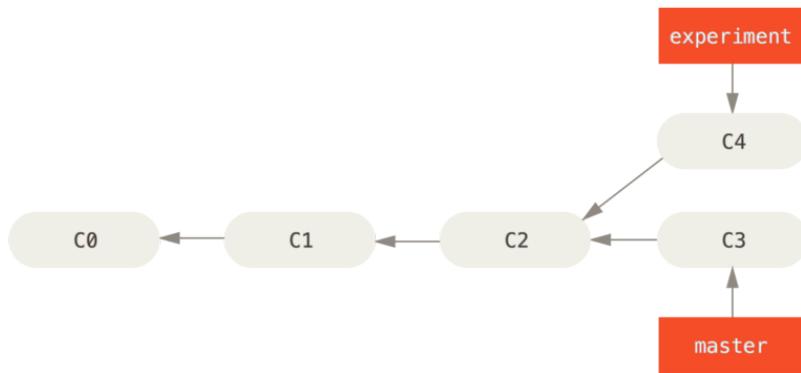
In Git, there are two main ways to integrate changes from one branch into another: the `merge` and the `rebase`. In this section you'll learn what rebasing is, how to do it, why it's a pretty amazing tool, and in what cases you won't want to use it.

The Basic Rebase

If you go back to an earlier example from “Основы слияния”, you can see that you diverged your work and made commits on two different branches.

FIGURE 3-27

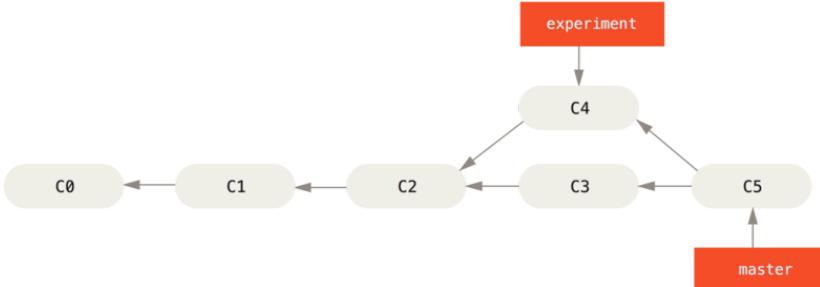
Simple divergent history



The easiest way to integrate the branches, as we've already covered, is the `merge` command. It performs a three-way merge between the two latest branch snapshots (C3 and C4) and the most recent common ancestor of the two (C2), creating a new snapshot (and commit).

FIGURE 3-28

Merging to integrate diverged work history



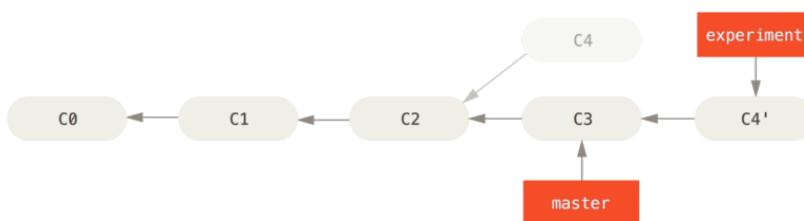
However, there is another way: you can take the patch of the change that was introduced in C4 and reapply it on top of C3. In Git, this is called *rebasing*. With the `rebase` command, you can take all the changes that were committed on one branch and replay them on another one.

In this example, you'd run the following:

```
$ git checkout experiment
$ git rebase master
```

First, rewinding head to replay your work on top of it...
Applying: added staged command

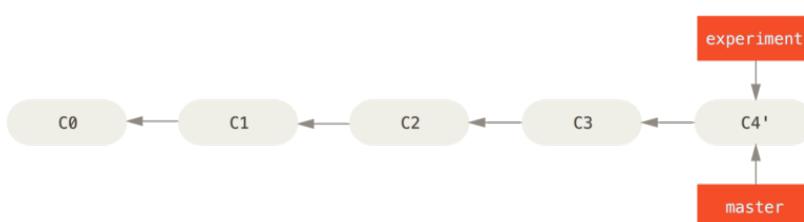
It works by going to the common ancestor of the two branches (the one you're on and the one you're rebasing onto), getting the diff introduced by each commit of the branch you're on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

**FIGURE 3-29**

Rebasing the change introduced in C4 onto C3

At this point, you can go back to the master branch and do a fast-forward merge.

```
$ git checkout master
$ git merge experiment
```

**FIGURE 3-30**

Fast-forwarding the master branch

Now, the snapshot pointed to by C4' is exactly the same as the one that was pointed to by C5 in the merge example. There is no difference in the end product of the integration, but rebasing makes for a cleaner history. If you examine the log of a rebased branch, it looks like a linear history: it appears that all the work happened in series, even when it originally happened in parallel.

Often, you'll do this to make sure your commits apply cleanly on a remote branch – perhaps in a project to which you're trying to contribute but that you don't maintain. In this case, you'd do your work in a branch and then rebase your work onto `origin/master` when you were ready to submit your patches to the main project. That way, the maintainer doesn't have to do any integration work – just a fast-forward or a clean apply.

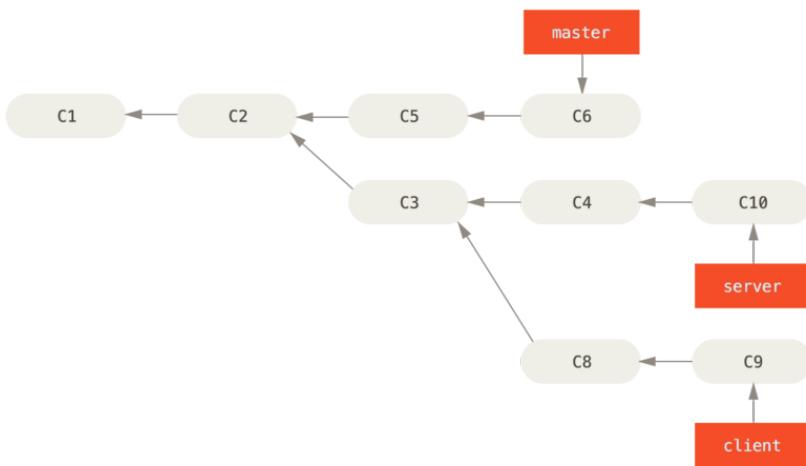
Note that the snapshot pointed to by the final commit you end up with, whether it's the last of the rebased commits for a rebase or the final merge commit after a merge, is the same snapshot – it's only the history that is different. Rebasing replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together.

More Interesting Rebases

You can also have your rebase replay on something other than the rebase target branch. Take a history like [Figure 3-31](#), for example. You branched a topic branch (`server`) to add some server-side functionality to your project, and made a commit. Then, you branched off that to make the client-side changes (`client`) and committed a few times. Finally, you went back to your server branch and did a few more commits.

FIGURE 3-31

A history with a topic branch off another topic branch

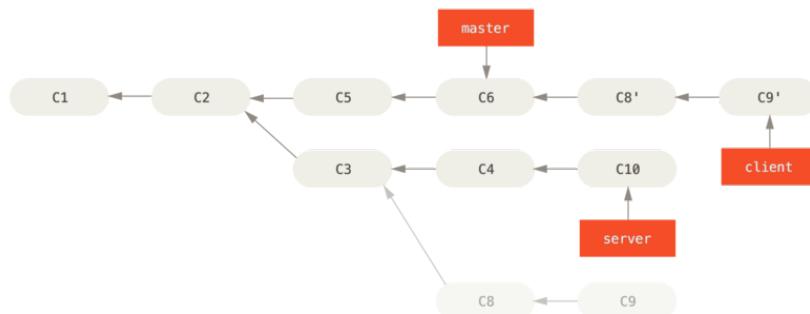


Suppose you decide that you want to merge your client-side changes into your mainline for a release, but you want to hold off on the server-side changes

until it's tested further. You can take the changes on client that aren't on server (C8 and C9) and replay them on your master branch by using the `--onto` option of `git rebase`:

```
$ git rebase --onto master server client
```

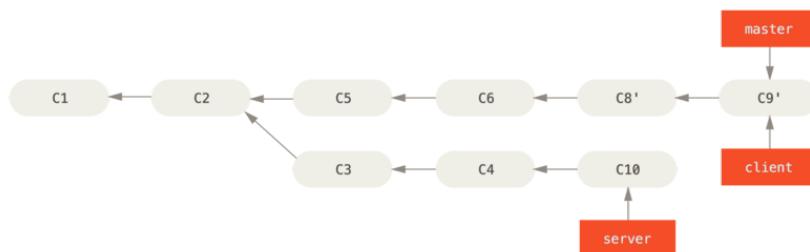
This basically says, "Check out the client branch, figure out the patches from the common ancestor of the `client` and `server` branches, and then replay them onto `master`." It's a bit complex, but the result is pretty cool.

**FIGURE 3-32**

Rebasing a topic branch off another topic branch

Now you can fast-forward your master branch (see **Figure 3-33**):

```
$ git checkout master
$ git merge client
```

**FIGURE 3-33**

Fast-forwarding your master branch to include the client branch changes

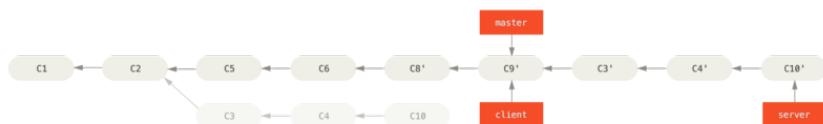
Let's say you decide to pull in your server branch as well. You can rebase the server branch onto the master branch without having to check it out first by running `git rebase [basebranch] [topicbranch]` – which checks out the topic branch (in this case, `server`) for you and replays it onto the base branch (`master`):

```
$ git rebase master server
```

This replays your `server` work on top of your `master` work, as shown in **Figure 3-34**.

FIGURE 3-34

Rebasing your server branch on top of your master branch



Then, you can fast-forward the base branch (`master`):

```
$ git checkout master
$ git merge server
```

You can remove the `client` and `server` branches because all the work is integrated and you don't need them anymore, leaving your history for this entire process looking like **Figure 3-35**:

```
$ git branch -d client
$ git branch -d server
```

FIGURE 3-35

Final commit history



The Perils of Rebasing

Ahh, but the bliss of rebasing isn't without its drawbacks, which can be summed up in a single line:

Do not rebase commits that exist outside your repository.

If you follow that guideline, you'll be fine. If you don't, people will hate you, and you'll be scorned by friends and family.

When you rebase stuff, you're abandoning existing commits and creating new ones that are similar but different. If you push commits somewhere and others pull them down and base work on them, and then you rewrite those commits with `git rebase` and push them up again, your collaborators will have to re-merge their work and things will get messy when you try to pull their work back into yours.

Let's look at an example of how rebasing work that you've made public can cause problems. Suppose you clone from a central server and then do some work off that. Your commit history looks like this:

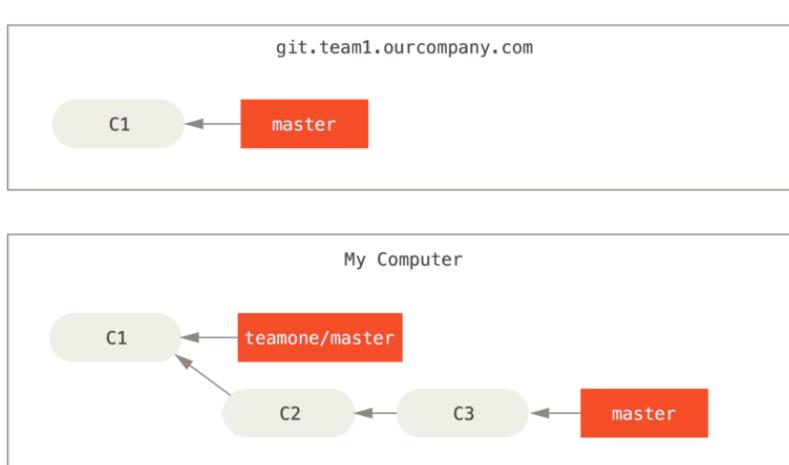


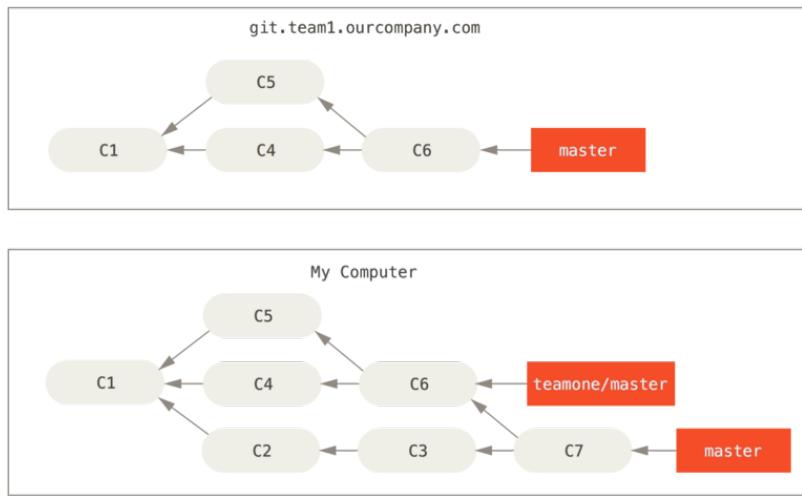
FIGURE 3-36

Clone a repository, and base some work on it

Now, someone else does more work that includes a merge, and pushes that work to the central server. You fetch them and merge the new remote branch into your work, making your history look something like this:

FIGURE 3-37

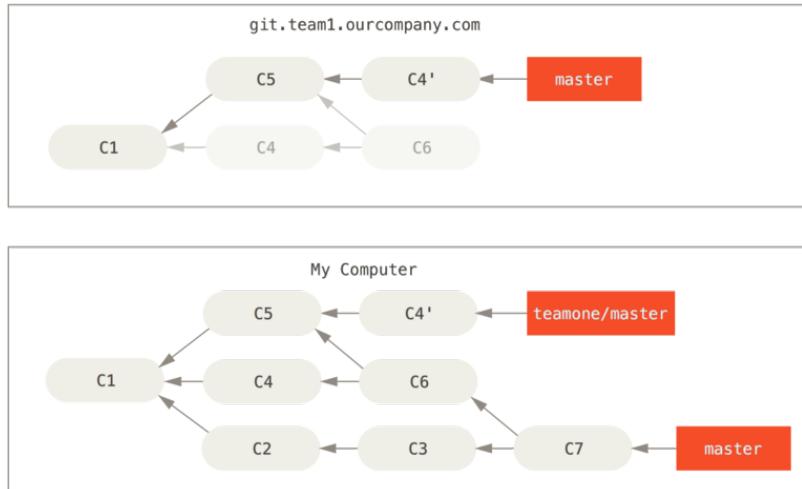
*Fetch more commits,
and merge them
into your work*



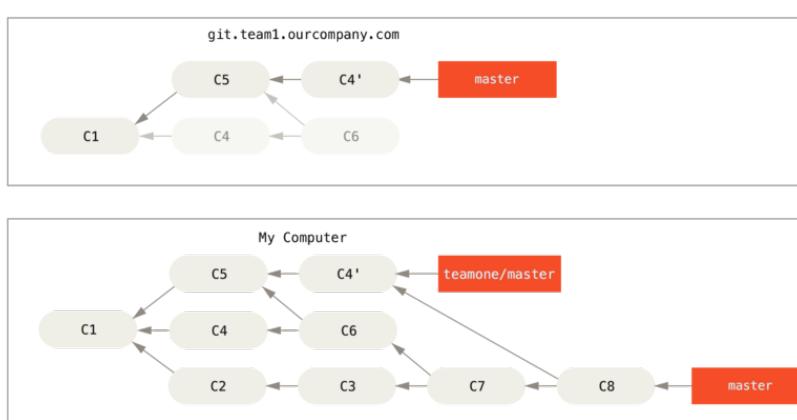
Next, the person who pushed the merged work decides to go back and rebase their work instead; they do a `git push --force` to overwrite the history on the server. You then fetch from that server, bringing down the new commits.

FIGURE 3-38

*Someone pushes
rebased commits,
abandoning
commits you've
based your work on*



Now you're both in a pickle. If you do a `git pull`, you'll create a merge commit which includes both lines of history, and your repository will look like this:

**FIGURE 3-39**

You merge in the same work again into a new merge commit

If you run a `git log` when your history looks like this, you'll see two commits that have the same author, date, and message, which will be confusing. Furthermore, if you push this history back up to the server, you'll reintroduce all those rebased commits to the central server, which can further confuse people. It's pretty safe to assume that the other developer doesn't want C4 and C6 to be in the history; that's why she rebased in the first place.

Rebase When You Rebase

If you **do** find yourself in a situation like this, Git has some further magic that might help you out. If someone on your team force pushes changes that overwrite work that you've based work on, your challenge is to figure out what is yours and what they've rewritten.

It turns out that in addition to the commit SHA-1 checksum, Git also calculates a checksum that is based just on the patch introduced with the commit. This is called a “patch-id”.

If you pull down work that was rewritten and rebase it on top of the new commits from your partner, Git can often successfully figure out what is uniquely yours and apply them back on top of the new branch.

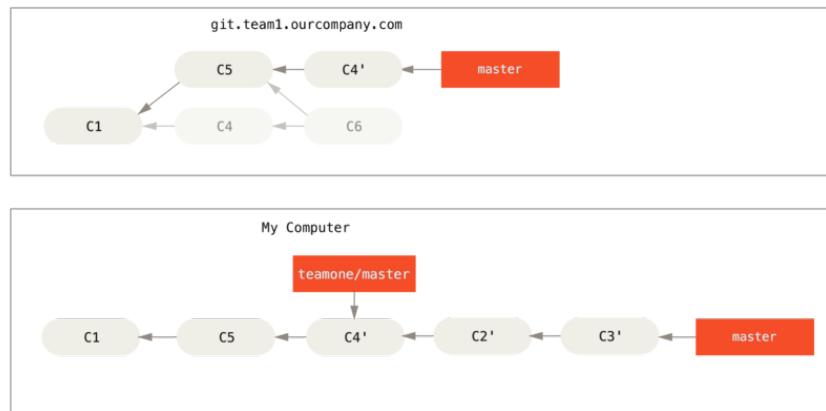
For instance, in the previous scenario, if instead of doing a merge when we're at [Figure 3-38](#) we run `git rebase teamone/master`, Git will:

- Determine what work is unique to our branch (C2, C3, C4, C6, C7)
- Determine which are not merge commits (C2, C3, C4)
- Determine which have not been rewritten into the target branch (just C2 and C3, since C4 is the same patch as C4')
- Apply those commits to the top of teamone/master

So instead of the result we see in [Figure 3-39](#), we would end up with something more like [Figure 3-40](#).

FIGURE 3-40

Rebase on top of force-pushed rebase work.



This only works if C4 and C4' that your partner made are almost exactly the same patch. Otherwise the rebase won't be able to tell that it's a duplicate and will add another C4-like patch (which will probably fail to apply cleanly, since the changes would already be at least somewhat there).

You can also simplify this by running `git pull --rebase` instead of a normal `git pull`. Or you could do it manually with a `git fetch` followed by a `git rebase teamone/master` in this case.

If you are using `git pull` and want to make `--rebase` the default, you can set the `pull.rebase` config value with something like `git config --global pull.rebase true`.

If you treat rebasing as a way to clean up and work with commits before you push them, and if you only rebase commits that have never been available publicly, then you'll be fine. If you rebase commits that have already been pushed publicly, and people may have based work on those commits, then you may be in for some frustrating trouble, and the scorn of your teammates.

If you or a partner does find it necessary at some point, make sure everyone knows to run `git pull --rebase` to try to make the pain after it happens a little bit simpler.

Rebase vs. Merge

Now that you've seen rebasing and merging in action, you may be wondering which one is better. Before we can answer this, let's step back a bit and talk about what history means.

One point of view on this is that your repository's commit history is a **record of what actually happened**. It's a historical document, valuable in its own right, and shouldn't be tampered with. From this angle, changing the commit history is almost blasphemous; you're *lying* about what actually transpired. So what if there was a messy series of merge commits? That's how it happened, and the repository should preserve that for posterity.

The opposing point of view is that the commit history is the **story of how your project was made**. You wouldn't publish the first draft of a book, and the manual for how to maintain your software deserves careful editing. This is the camp that uses tools like rebase and filter-branch to tell the story in the way that's best for future readers.

Now, to the question of whether merging or rebasing is better: hopefully you'll see that it's not that simple. Git is a powerful tool, and allows you to do many things to and with your history, but every team and every project is different. Now that you know how both of these things work, it's up to you to decide which one is best for your particular situation.

In general the way to get the best of both worlds is to rebase local changes you've made but haven't shared yet before you push them in order to clean up your story, but never rebase anything you've pushed somewhere.

Итоги

Мы рассмотрели базовые функции ветвления и слияния в Git. Вы должны быть способны свободно создавать и переключаться на новую ветку, переключаться между ветками и сливать локальные ветки вместе. Также Вы должны уметь выкладывать Ваши ветки на общий сервер, работать с другими людьми над общими ветками и интегрировать Ваши ветки до того, как они будут доступны другим разработчикам. Далее мы поговорим о том, что Вам нужно, чтобы запустить Ваш собственный сервер с хостингом для Git-репозитория.

Git на сервере

4

К этому моменту вы уже должны уметь делать большую часть повседневных задач, для которых вы будете использовать Git. Однако, для совместной работы в Git, вам необходим удаленный репозиторий. Несмотря на то, что технически вы можете отправлять и забирать изменения непосредственно из личных репозиториев, делать это не рекомендуется. Вы легко можете испортить то, над чем работают другие, если не будете аккуратны. К тому же, вам бы наверняка хотелось, чтобы остальные имели доступ к репозиторию даже если ваш компьютер выключен, поэтому наличие более надежного репозитория обычно весьма полезно. Предпочтительный метод взаимодействия с кем-либо – это создание промежуточного репозитория, к которому вы оба будете иметь доступ, и отправка и получение изменений через него.

Запустить Git-сервер достаточно просто. Для начала следует выбрать протокол, который вы будете использовать для связи с сервером. Доступные протоколы с их достоинствами и недостатками описываются в первой части этой главы. Следующие части освещают базовые конфигурации с использованием этих протоколов, а также настройку вашего сервера для работы с ними. Далее мы рассмотрим несколько вариантов готового хостинга, которые можно использовать если, вы не против разместить ваш код на чужом сервере и не хотите мучиться с настройками и поддержкой вашего собственного сервера.

Если вас не интересует настройка собственного сервера, вы можете перейти сразу к последней части этой главы для настройки аккаунта на Git-хостинге, и затем перейти к следующей главе, где мы обсудим различные аспекты работы с распределенной системой контроля версий.

Удаленный репозиторий — это обычно *голый* (*чистый*, *bare*) *репозиторий* – репозиторий Git, не имеющий рабочего каталога. Поскольку этот репозиторий используется только для обмена, то нет

причин создавать рабочую копию файлов на диске, и он содержит только данные Git.

Проще говоря, голый репозиторий содержит только каталог `.git` вашего проекта и ничего больше.

Протоколы

Git умеет работать с четырьмя сетевыми протоколами для передачи данных: локальный, HTTP, Secure Shell (SSH) и Git. В этой части мы обсудим каждый из них, и в каких случаях стоит (или не стоит) его использовать.

Локальный протокол

Базовым протоколом является *Локальный протокол*, при использовании которого удаленный репозиторий – другой каталог на диске. Наиболее часто он используется, если все члены команды имеют доступ к общей файловой системе, например к NFS, или, что менее вероятно, когда все работают на одном компьютере. Последний вариант не столь хорош, поскольку все копии вашего репозитория находятся на одном компьютере, делая возможность потерять всё более вероятной.

Если у вас смонтирована общая файловая система, вы можете клонировать, отправлять и получать изменения из локального репозитория. Чтобы клонировать такой репозиторий или добавить его в качестве удаленного в существующий проект, используйте путь к репозиторию в качестве URL. Например, для клонирования локального репозитория вы можете выполнить что-то вроде этого:

```
$ git clone /opt/git/project.git
```

Или этого:

```
$ git clone file:///opt/git/project.git
```

Git работает немного по-другому, если вы явно укажете префикс `file://` в начале вашего URL. Когда вы просто указываете путь, Git пытается использовать жесткие ссылки и копировать файлы, когда это нужно. Если вы указываете `file://`, Git работает с данными так

же, как при использовании сетевых протоколов, что в целом — менее эффективный способ передачи данных. Причиной для использования `file://` может быть необходимость создания чистой копии репозитория без лишних внешних ссылок и объектов, обычно после импорта из другой системы управления версиями или чего-то похожего (см. [Chapter 10](#) о задачах поддержки). Мы будем использовать обычные пути, поскольку это практически всегда быстрее.

Чтобы добавить локальный репозиторий в существующий проект, вы можете воспользоваться командой:

```
$ git remote add local_proj /opt/git/project.git
```

Теперь вы можете отправлять и получать изменения из этого репозитория так, как вы это делали по сети.

ДОСТОИНСТВА

Преимущества основанных на файлах хранилищ в том, что они просты и используют существующие разграничения прав на файлы и сетевой доступ. Если у вас уже есть общая файловая система, доступ к которой имеет вся команда, настройка репозитория очень проста. Вы помещаете голый репозиторий туда, куда все имеют доступ, и выставляете права на чтение и запись, как вы бы это сделали для любого другого общего каталога. Мы обсудим, как экспортовать голую копию репозитория для этой цели, в следующем разделе: “[Установка Git на сервер](#)”.

Также это хорошая возможность быстро получить наработки из чьего-то рабочего репозитория. Если вы и ваш коллега работаете над одним и тем же проектом, и он хочет, чтобы вы что-то проверили, то запуск команды вроде `git pull /home/john/project` зачастую проще, чем отправлять и забирать с удалённого сервера.

НЕДОСТАТКИ

Недостаток этого метода в том, что общий доступ обычно сложнее настроить и получить из разных мест, чем простой сетевой доступ. Если вы хотите отправлять со своего ноутбука, когда вы дома, вы должны смонтировать удалённый диск, что может оказаться сложнее и медленнее, чем сетевой доступ.

Также важно упомянуть, что не всегда использование общей точки монтирования является быстрейшим вариантом. Локальный репозиторий быстр, только если вы имеете быстрый доступ к данным. Репозиторий на NFS часто медленнее, чем репозиторий через SSH на том же сервере, позволяющий Git использовать на полную локальные диски на каждой системе.

Протоколы HTTP

Git может работать через HTTP в двух различных режимах. До версии Git 1.6.6 был только один режим, очень простой и предназначенный только для чтения. В версии 1.6.6 появился новый, более умный режим, позволяющий Git более интеллектуально определять необходимость передачи данных, наподобие того, как это происходит при использовании SSH. В последние годы новый протокол стал очень популярен, так как он проще для пользователя и более эффективен. Новая версия часто называется “Умным” (“Smart”) HTTP, а старая “Тупым” (“Dumb”) HTTP. Мы рассмотрим сначала “умный” протокол.

УМНЫЙ HTTP

“умный” протокол HTTP работает схожим с SSH или Git-протоколами образом, но поверх стандартных HTTP/S портов и может использовать различные механизмы аутентификации HTTP, это часто проще для пользователя, чем что-то вроде SSH, так как можно использовать вещи вроде базовой парольной идентификации вместо установки SSH-ключей.

Он стал наверное наиболее популярным способом использования Git, так как может использоваться и для анонимного доступа, как протокол `git://`, и для отправки изменений с аутентификацией и шифрованием, как протокол SSH. Вместо использования разных адресов URL для этих вещей, можно использовать один для всего. Если вы пытаетесь отослать изменения в репозиторий требует аутентификации (обычно так и есть), сервер может спросить логин и пароль. То же касается и доступа на чтение.

На самом деле для сервисов вроде GitHub, адрес URL который вы используете для просмотра репозитория в браузере (например, [“`https://github.com/schacon/simplegit`”](https://github.com/schacon/simplegit)) — тот же, который вы можете использовать для клонирования или, если у вас есть доступ, для отправки изменений.

ТУПОЙ HTTP

Если сервер не отвечает на умный запрос Git по HTTP, клиент Git попытается откатиться на более простой “тупой” HTTP-протокол. Тупой протокол ожидает, что голый репозиторий Git будет обслуживаться веб-сервером как набор файлов. Прелесть тупого протокола HTTP — в простоте настройки. По сути, всё, что необходимо сделать — поместить голый репозиторий внутрь каталога с HTTP-документами, установить обработчик `post-update` и всё (см. “[Git Hooks](#)”). Теперь каждый, имеющий доступ к веб-серверу, на котором был размещен репозиторий, может его клонировать. Таким образом, чтобы открыть доступ к вашему репозиторию на чтение через HTTP, нужно сделать что-то наподобие этого:

```
$ cd /var/www/htdocs/  
$ git clone --bare /path/to/git_project gitproject.git  
$ cd gitproject.git  
$ mv hooks/post-update.sample hooks/post-update  
$ chmod a+x hooks/post-update
```

Вот и всё. Обработчик `post-update`, входящий в состав Git по умолчанию, выполняет необходимую команду (`git update-server-info`), чтобы извлечение (`fetch`) и клонирование (`clone`) по HTTP работали правильно. Эта команда выполняется, когда вы отправляете изменения в репозиторий (возможно посредством SSH); Затем остальные могут клонировать его командой

```
$ git clone https://example.com/gitproject.git
```

В рассмотренном примере мы использовали каталог `/var/www/htdocs`, обычно используемый сервером Apache, но вы можете использовать любой веб-сервер, отдающий статические данные, расположив голый репозиторий в нужном каталоге. Данные Git представляют собой обычные файлы (в [Chapter 10](#) предоставление данных рассматривается более подробно).

Чаще всего вы будете использовать умный HTTP для чтения/записи, либо тупой только для чтения. Случай совместного их использования встречаются редко.

ДОСТОИНСТВА

Мы сосредоточимся на преимуществах умной версии протокола HTTP.

Простота использования одного адреса URL для всех типов доступа и аутентификация только при необходимости делает работу очень простой для конечного пользователя. Возможность аутентификации посредством логина и пароля также даёт преимущество перед SSH, так как пользователям перед использованием не нужно создавать SSH-ключи и загружать публичную часть на сервер. Для неопытных пользователей или пользователей систем, где SSH мало распространён, это большой плюс. Это также очень быстрый и эффективный протокол, сравнимый с SSH.

Вы также можете раздавать доступ к своим репозиториям только для чтения по HTTPS, шифруя содержимое передачи; или вы можете зайти так далеко, что клиенты будут использовать персональные подписанные SSL-сертификаты.

Другой плюс в том, что HTTP/S очень распространённые протоколы и корпоративные брандмауэры часто настроены для разрешения их работы.

НЕДОСТАТКИ

На некоторых серверах Git поверх HTTP/S может быть немного сложнее в настройке по сравнению с SSH. Кроме этого, преимущества других протоколов доступа к Git перед “Умным” HTTP незначительны.

Если вы используете HTTP для отправки изменений, удостоверение ваших полномочий зачастую сложнее, чем при использовании SSH-ключей. Но есть несколько инструментов для кеширования полномочий, включая Keychain access на OSX и Credential Manager на Windows, которые вы можете использовать для упрощения процесса. В “Хранилище учетных данных” кеширование паролей HTTP рассматривается подробней.

Протокол SSH

Часто используемый транспортный протокол для хостинга Git — это SSH. Причина этого в том, что доступ по SSH уже есть на многих серверах, а если его нет, то его очень легко настроить. К тому же, SSH — протокол с аутентификацией, и его благодаря распространенности обычно легко настроить и использовать.

Чтобы клонировать Git-репозиторий по SSH, вы можете указать префикс ssh:// в URL, например:

```
$ git clone ssh://user@server/project.git
```

Или можно использовать для протокола SSH краткий синтаксис наподобие scp:

```
$ git clone user@server:project.git
```

Также вы можете не указывать имя пользователя, Git будет использовать то, под которым вы вошли в систему.

ДОСТОИНСТВА

SSH имеет множество достоинств. Во-первых, SSH достаточно легко настроить – демоны SSH распространены, многие системные администраторы имеют опыт работы с ними, и во многих дистрибутивах они уже настроены или есть утилиты для управления ими. Далее, доступ по SSH безопасен – данные передаются зашифрованными по авторизованным каналам. Наконец, так же как и протоколы HTTP/S, Git и локальный протокол, SSH эффективен, максимально сжимая данные перед передачей.

НЕДОСТАТКИ

Недостаток SSH в том, что, используя его, вы не можете обеспечить анонимный доступ к репозиторию. Клиенты должны иметь доступ к машине по SSH, даже для работы в режиме только на чтение, что делает SSH неподходящим для проектов с открытым исходным кодом. Если вы используете Git только внутри корпоративной сети, то, возможно, SSH – единственный протокол, с которым вам придется иметь дело. Если же вам нужен анонимный доступ на чтение для своих проектов, придется настроить SSH для себя, чтобы выкладывать изменения, и что-нибудь другое для других, для скачивания.

Git-протокол

Следующий протокол – Git-протокол. Вместе с Git поставляется специальный демон, который слушает отдельный порт (9418) и предоставляет сервис, схожий с протоколом SSH, но абсолютно без аутентификации. Чтобы использовать Git-протокол для репозитория, вы должны создать файл `git-export-daemon-ok`, иначе демон не

будет работать с этим репозиторием, но следует помнить, что в протоколе отсутствуют средства безопасности. Соответственно, любой репозиторий в Git может быть либо доступен для клонирования всем, либо нет. Как следствие, обычно отправлять изменения по этому протоколу нельзя. Вы можете открыть доступ на запись, но из-за отсутствия аутентификации в этом случае кто угодно, зная URL вашего проекта, сможет его изменить. В общем, это редко используемая возможность.

ДОСТОИНСТВА

Git-протокол – часто самый быстрый из доступных протоколов. Если у вас проект с публичным доступом и большой трафик, или у вас очень большой проект, для которого не требуется аутентификация пользователей для чтения, вам стоит настроить демон Git для вашего проекта. Он использует тот же механизм передачи данных, что и протокол SSH, но без дополнительных затрат на шифрование и аутентификацию.

НЕДОСТАТКИ

Недостатком Git-протокола является отсутствие аутентификации. Поэтому обычно не следует использовать этот протокол как единственный способ доступа к вашему проекту. Обычно он используется в паре с SSH или HTTPS для нескольких разработчиков, имеющих доступ на запись, тогда как все остальные используют `git://` с доступом только на чтение. Кроме того, это, вероятно, самый сложный для настройки протокол. Вы должны запустить собственно демон, для чего необходим сервис `xinetd` или ему подобный, что не всегда легко сделать. Также необходимо, чтобы сетевой экран позволял доступ на порт 9418, который не относится к стандартным портам, всегда разрешённым в корпоративных брандмауэрах. За сетевыми экранами крупных корпораций этот неизвестный порт часто заблокирован.

Установка Git на сервер

Рассмотрим теперь установку сервиса Git с поддержкой этих протоколов на сервер.

Здесь мы будем приводить команды и шаги, необходимые для базовой, упрощённой установки на Linux-сервер, но эти сервисы можно запустить и на Mac или Windows-сервере. На самом деле, установка боевого сервера в вашей инфраструктуре неминуемо будет иметь отличия в настройках безопасности или инструментах операционной системы, но мы надеемся дать вам общее понимание происходящего.

Для того чтобы приступить к установке любого сервера Git, вы должны экспортировать существующий репозиторий в новый голый репозиторий, т.е. репозиторий без рабочего каталога. Обычно это несложно сделать. Чтобы клонировать ваш репозиторий и создать новый голый репозиторий, выполните команду `clone` с параметром `--bare`. По существующему соглашению, каталоги с голыми репозиториями заканчиваются на `.git`, например:

```
$ git clone --bare my_project my_project.git
Cloning into bare repository 'my_project.git'...
done.
```

Теперь у вас должна быть копия данных из каталога Git в каталоге `my_project.git`.

Грубо говоря, это что-то наподобие

```
$ cp -Rf my_project/.git my_project.git
```

Тут есть пара небольших различий в файле конфигурации; но в вашем случае эту разницу можно считать несущественной. Можно считать, что в этом случае берётся собственно репозиторий Git без рабочего каталога и создаётся каталог только для него.

Размещение голого репозитория на сервере

Теперь, когда у вас есть голая копия вашего репозитория, всё, что вам нужно сделать, это поместить ее на сервер и настроить протоколы. Условимся, что вы уже настроили сервер `git.example.com`, имеете к нему доступ по SSH и хотите размещать все ваши репозитории Git в каталоге `/opt/git`. Считая, что `/opt/git` уже есть на сервере, вы можете добавить ваш новый репозиторий копированием голого репозитория:

```
$ scp -r my_project.git user@git.example.com:/opt/git
```

Теперь другие пользователи, имеющие доступ к серверу по SSH и право на чтение к каталогу `/opt/git`, могут клонировать ваш репозиторий, выполнив

```
$ git clone user@git.example.com:/opt/git/my_project.git
```

Если у пользователя сервера есть право на запись в каталог `/opt/git/my_project.git`, он автоматически получает возможность отправки изменений в репозиторий.

Git автоматически добавит право на запись в репозиторий для группы, если вы запустите команду `git init` с параметром `--shared`.

```
$ ssh user@git.example.com
$ cd /opt/git/my_project.git
$ git init --bare --shared
```

Видите, как это просто, взять репозиторий Git, создать голую версию и поместить ее на сервер, к которому вы и ваши коллеги имеете доступ по SSH. Теперь вы готовы работать вместе над одним проектом.

Важно отметить, что это практически всё, что вам нужно сделать, чтобы получить рабочий Git-сервер, к которому имеют доступ несколько человек – просто добавьте учетные записи SSH на сервер, и положите голый репозиторий в то место, к которому эти пользователи имеют доступ на чтение и запись. И всё.

Из нескольких последующих разделов вы узнаете, как получить более сложные конфигурации. В том числе как не создавать учетные записи для каждого пользователя, как сделать публичный доступ на чтение репозитория, как установить веб-интерфейс и др. Однако, помните, что для совместной работы пары человек на закрытом проекте, всё, что вам *нужно* – это SSH-сервер и голый репозиторий.

Малые установки

Если вы небольшая фирма или вы только пробуете Git в вашей организации и у вас мало разработчиков, то всё достаточно просто. Один из наиболее сложных аспектов настройки сервера Git –

управление пользователями. Если вы хотите, чтобы некоторые репозитории были доступны некоторым пользователям только на чтение, а другие и на чтение, и на запись, вам может быть не очень просто привести права доступа в порядок.

SSH ДОСТУП

Если у вас уже есть сервер, к которому все ваши разработчики имеют доступ по SSH, проще всего разместить ваш первый репозиторий там, поскольку вам не нужно практически ничего делать (как мы уже обсудили в предыдущем разделе). Если вы хотите более сложного управления правами доступа к вашим репозиториям, вы можете сделать это обычными правами файловой системы, предоставляемыми операционной системой вашего сервера.

Если вы хотите разместить ваши репозитории на сервер, на котором нет учетных записей для каждого в вашей команде, кому нужен доступ на запись, вы должны настроить доступ по SSH для них. Будем считать, что если у вас есть сервер, на котором вы хотите это сделать, то SSH-сервер на нем уже установлен, и через него вы имеете доступ к серверу.

Есть несколько способов дать доступ каждому в вашей команде. Первый — настроить учетные записи для каждого, это просто, но может быть весьма обременительно. Вероятно, вы не захотите для каждого пользователя выполнять `adduser` и задавать временные пароли.

Второй способ — создать на машине одного пользователя `git`, попросить каждого пользователя, кому нужен доступ на запись, прислать вам открытый ключ SSH, и добавить эти ключи в файл `~/ssh/authorized_keys` вашего нового пользователя `git`. Теперь все будут иметь доступ к этой машине через пользователя `git`. Это никак не повлияет на данные коммита — пользователь, под которым вы соединяетесь с сервером по SSH, не затрагивает сделанные вами фиксации.

Другой способ сделать это — использовать SSH-сервер, аутентифицирующий по LDAP-серверу или любому другому централизованному источнику, который у вас может быть уже настроен. Любой способ аутентификации по SSH, какой вы только сможете придумать, должен работать, если пользователь может получить доступ к консоли.

Генерация открытого SSH ключа

Как отмечалось ранее, многие Git-серверы используют аутентификацию по открытым SSH-ключам. Для того чтобы предоставить открытый ключ, пользователь должен его сгенерировать, если только этого уже не было сделано ранее. Этот процесс аналогичен во всех операционных системах. Сначала вам стоит убедиться, что у вас ещё нет ключа. Пользовательские SSH-ключи по умолчанию хранятся в каталоге `~/.ssh` этого пользователя. Вы можете легко проверить, есть ли у вас ключ, зайдя в этот каталог и посмотрев его содержимое:

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa      known_hosts
config           id_dsa.pub
```

Ищите здесь файл `id_dsa` или `id_rsa` и соответствующий ему файл с расширением `.pub`. Файл с расширением `.pub` — это ваш открытый ключ, а второй файл — ваш закрытый (секретный) ключ. Если указанные файлы у вас отсутствуют (или даже нет каталога `.ssh`), вы можете создать их, запустив программу `ssh-keygen`, которая входит в состав пакета SSH в системах Linux/Mac, а для Windows поставляется в составе MSysGit:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):
Created directory '/home/schacon/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local
```

Сначала необходимо указать расположение файла для сохранения ключа (`.ssh/id_rsa`), затем дважды ввести пароль, который, впрочем, можно оставить пустым, если вы не хотите его вводить каждый раз, когда используете ключ.

Теперь каждый пользователь должен послать свой открытый ключ вам или тому, кто администрирует Git-сервер (предположим, что ваш SSH-сервер уже настроен на работу с открытыми ключами). Для этого им нужно скопировать всё содержимое файла с расширением .pub и отправить его по электронной почте. Открытый ключ выглядит примерно так:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABlOUpkDHrfHY17SbrmTIpNLTK9Tjom/BWDSU
GPl+nafzLHDTYW7hdI4yZ5ew18JH4JW9jbhUFrvlQzM7xleLEVf4h9lFX5QVkbPppSwg0cda3
Pbv7k0dJ/MTyB1WXFRC+Hao3FXRitBqxjX1nKhXpHAZsMcilQ8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrx88XypNDvjYNby6vw/Pb0gwert/En
mZ+AW40ZPnPPI89ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2so1d01QraTLMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

Более подробное руководство по созданию SSH-ключей на различных системах вы можете найти в руководстве GitHub по SSH-ключам на <https://help.github.com/articles/generating-ssh-keys>.

Настраиваем сервер

Давайте рассмотрим настройку доступа по SSH на стороне сервера. В этом примере мы будем использовать метод authorized_keys для аутентификации пользователей. Мы подразумеваем, что вы используете стандартный дистрибутив Linux типа Ubuntu. Для начала создадим пользователя git и каталог .ssh для этого пользователя:

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh && chmod 700 .ssh
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

Затем вам нужно добавить открытый SSH-ключ некоторого разработчика в файл authorized_keys пользователя git. Предположим, вы уже получили несколько ключей и сохранили их во временные файлы. Напомню, открытый ключ выглядит как-то так:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
```

```
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdGw1GYEigS9Ez
Sdf8AccIicTDWbqlAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
07TCUsBdLQlgMV0Fq1I2uPWQ0kOWQAHukE0mfjy2jctxSDBQ220ymjaNsHT4kgTZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Вы просто добавляете их в файл `authorized_keys` пользователя `git` в его каталоге `.ssh`:

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

Теперь вы можете создать пустой репозиторий для них, запустив `git init` с параметром `--bare`, что инициализирует репозиторий без рабочего каталога:

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git init --bare
Initialized empty Git repository in /opt/git/project.git/
```

Затем Джон, Джози или Джессика могут отправить первую версию своего проекта в этот репозиторий, добавив его как удаленный и отправив ветку. Заметьте, что кто-то должен заходить на сервер и создавать голый репозиторий каждый раз, когда вы хотите добавить проект. Пусть `gitserver` – имя хоста сервера, на котором вы создали пользователя `git` и репозиторий. Если он находится в вашей внутренней сети, вы можете настроить запись DNS для `gitserver`, ссылающуюся на этот сервер, и использовать эти команды(считая что `myproject` есть существующий проект с файлами):

```
# На компьютере Джона
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/git/project.git
$ git push origin master
```

Теперь остальные могут клонировать его и отправлять (push) туда изменения так же легко:

```
$ git clone git@gitserver:/opt/git/project.git
$ cd project
$ vim README
$ git commit -am 'fix for the README file'
$ git push origin master
```

Этим способом вы можете быстро получить Git-сервер с доступом на чтение/запись для небольшой группы разработчиков.

Заметьте, что теперь все эти пользователи могут заходить на сервер как пользователь `git`. Чтобы это предотвратить, нужно изменить ему оболочку на что-то другое в файле `passwd`.

Вы можете легко ограничить пользователя `git` только действиями, связанными с Git, с помощью ограниченной оболочки `git-shell`, поставляемой вместе с Git. Если вы выставите её в качестве командного интерпретатора пользователя `git`, то этот пользователь не сможет получить доступ к обычной командной оболочке на вашем сервере. Чтобы её использовать, укажите `git-shell` вместо `bash` или `csh` в качестве командной оболочки пользователя. Для этого вы должны сначала добавить `git-shell` в `/etc/shells` если её там ещё нет:

```
$ cat /etc/shells  # посмотрим, присутствует ли `git-shell`. Если нет...
$ which git-shell  # проверим, что git-shell установлена.
$ sudo vim /etc/shells # и добавим путь к git-shell из предыдущей команды
```

Теперь можно изменить оболочку для пользователя используя `chsh <username>`:

```
$ sudo chsh git  # и вводим путь к git-shell, обычно /usr/bin/git-shell
```

Теперь пользователь `git` может использовать SSH соединение только для работы с репозиториями Git и не может зайти на машину. Вы можете попробовать и увидите, что вход в систему отклонен:

```
$ ssh git@gitserver
fatal: Interactive git shell is not enabled.
```

```
hint: ~/git-shell-commands should exist and have read and execute access.
Connection to gitserver closed.
```

Теперь сетевые команды Git будут работать, но пользователи не смогут заходить на сервер. Как указывает вывод, вы также можете изменить домашний каталог пользователя `git`, чтобы немного изменить поведение `git-shell`. Например, вы можете ограничить команды Git, которые сервер будет принимать или сообщение которое увидят пользователи если попробуют зайти по SSH. Запустите `git help shell` для получения дополнительной информации.

Git-демон

Далее мы установим демон, обслуживающий репозитории по протоколу “Git”. Это широко распространённый вариант для быстрого доступа без аутентификации. Помните, что раз сервис — без аутентификации, всё, что обслуживается по этому протоколу — публично доступно в сети.

Если вы запускаете демон на сервере не за сетевым экраном, он должен использоваться только для проектов, которые публично видны внешнему миру. Если сервер находится за вашим сетевым экраном, вы можете использовать его для проектов, к которым большое число людей или компьютеров (серверов непрерывной интеграции или сборки) должно иметь доступ только на чтение, и если вы не хотите для каждого из них заводить SSH-ключ.

В любом случае, протокол Git относительно просто настроить. Упрощённо, вам нужно запустить следующую команду в демонизированной форме:

```
git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

`--reuseaddr` позволит серверу перезапуститься без ожидания истечения старых подключений, `--base-path` позволит людям не указывать полный путь, чтобы склонировать проект, а путь на конце говорит демону Git, где искать экспортруемые репозитории. Если у вас запущен сетевой экран, вы должны проколоть в нём дырочку, открыв порт 9418 на машине, где всё это запущено.

Вы можете демонизировать этот процесс несколькими путями, в зависимости от операционной системы. На машине с Ubuntu используйте Upstart-сценарий. Итак, в этом файле

```
/etc/event.d/local-git-daemon
```

поместите такой сценарий:

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
--user=git --group=git \
--reuseaddr \
--base-path=/opt/git/ \
/opt/git/
respawn
```

По соображениям безопасности крайне приветствуется, если вы будете запускать этого демона от имени пользователя с правами только на чтение репозиториев — вы легко можете сделать это, создав пользователя *git-ro* и запустив этого демона из-под него. Для простоты мы запустим его от того же пользователя *git*, от которого запущен *git-shell*.

Когда вы перезапустите машину, Git-демон запустится автоматически, и перезапустится, если вдруг завершится. Чтобы запустить его без перезагрузки машины, выполните следующее:

```
initctl start local-git-daemon
```

На других системах вы можете использовать *xinetd*, сценарий вашей системы *sysvinit*, или что-то другое — главное, чтобы вы могли эту команду как-либо демонизировать и присматривать за ней.

Затем нужно указать Git, к каким репозиториям предоставить неаутентифицированный доступ через Git-сервер. Вы можете сделать это для каждого репозитория, создав файл с именем *git-daemon-export-ok*.

```
$ cd /path/to/project.git
$ touch git-daemon-export-ok
```

Наличие этого файла скажет Git'у, что можно обслуживать этот проект без аутентификации.

Умный HTTP

Теперь у нас есть доступ с аутентификацией через SSH и неаутентифицированный доступ через `git://`, но есть ещё протокол, который может делать и то и другое. Настройка умного HTTP — это просто установка CGI-скрипта `git-http-backend`, поставляемого с Git, на сервер . Этот CGI-скрипт будет читать путь и заголовки, посылаемые `git fetch` или `git push` в URL и определять, может ли клиент работать через HTTP (это так для любого клиента, начиная с версии 1.6.6). Если CGI-скрипт видит, что клиент умный, то и общаться с ним будет по-умному, иначе откатится на простое поведение (так что он обратно совместим по чтению со старыми клиентами).

Давайте пройдёмся по самой базовой установке. Мы настроим Apache как сервер CGI. Если у вас не установлен Apache, вы можете сделать это на Linux-машине примерно так:

```
$ sudo apt-get install apache2 apache2-utils
$ a2enmod cgi alias env
```

Это также включит модули `mod_cgi`, `mod_alias` и `mod_env`, необходимые для корректной работы.

Далее мы добавим некоторые вещи в конфигурационный файл Apache, чтобы запускать `git-http-backend` как обработчик для всего по пути `/git` на веб-сервере.

```
SetEnv GIT_PROJECT_ROOT /opt/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/libexec/git-core/git-http-backend/
```

Если пропустить переменную `GIT_HTTP_EXPORT_ALL`, тогда Git будет отдавать только неаутентифицированным клиентам репозитории с файлом `git-daemon-export-ok` внутри, также как делает Git-демон.

Далее нужно сказать Apache разрешить запросы к этому пути примерно так:

```
<Directory "/usr/lib/git-core*">
  Options ExecCGI Indexes
  Order allow,deny
  Allow from all
```

```
Require all granted
</Directory>
```

Наконец, нужно как-то аутентифицировать запись, например с помощью такого блока Auth:

```
<LocationMatch "^/git/.*/git-receive-pack$">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /opt/git/.htpasswd
    Require valid-user
</LocationMatch>
```

Это потребует создания файла `.htaccess` с паролями всех пользователей. Вот пример добавления пользователя “`schacon`” в этот файл:

```
$ htdigest -c /opt/git/.htpasswd "Git Access" schacon
```

Есть множество путей аутентифицировать пользователей Apache, придётся выбрать и реализовать один из них. Это просто простейший пример, который можно привести. Вы также почти наверняка захотите настроить SSL, чтобы все данные были зашифрованы.

Мы не хотим погружаться слишком глубоко в бездну настроек Apache, так как у вас может быть другой сервер или другие требования к аутентификации. Идея в том, что Git идёт с CGI-скриптом `git-http-backend`, который берет на себя согласование передачи и приёма данных по HTTP. Он не реализует аутентификации сам по себе, но это легко настраивается на уровне веб-сервера, который его запускает. Вы можете сделать это практически на любом веб-сервере, поддерживающем CGI, так что используйте тот, который знаете лучше всего.

За дополнительной информацией о настройке аутентификации в Apache, обратитесь к документации: <http://httpd.apache.org/docs/current/howto/auth.html>

GitWeb

Теперь, когда у вас есть основной доступ на чтение и запись и доступ только на чтение к вашему проекту, вероятно, вы захотите настроить простой веб-визуализатор. Git поставляется в комплекте с CGI-сценарием, называющимся GitWeb, который используется для этого.

FIGURE 4-1

Веб-интерфейс
GitWeb.

The screenshot shows the GitWeb interface for a repository. At the top, there's a navigation bar with links for 'summary', 'shortlog', 'log', 'commit', 'commitdiff', and 'tree'. Below the navigation is a search bar and a 'commit' button. The main area is divided into sections:

- summary**: Displays basic repository information: 'description: Unnamed repository; edit this file `description` to name the repository.', 'owner: Ben Straub', and 'last change: Wed, 11 Jun 2014 12:20:23 -0700 (21:20 +0200)'.
- shortlog**: A list of commit messages from June 2014, showing merges and bug fixes related to libgit2 and its dependencies. One commit is highlighted with a yellow background and labeled 'v0.21.0-rc1'.
- tags**: A list of tags ordered by creation date, ranging from '3 weeks ago v0.21.0-rc1' to '3 years ago v0.11.0'.

Если вы хотите проверить, как GitWeb будет выглядеть для вашего проекта, Git поставляется с командой для быстрой установки временного экземпляра, если в вашей системе есть легковесный веб-сервер, такой как lighttpd или webgrick. На машинах с Linux lighttpd часто установлен, поэтому возможно вы сможете его запустить, выполнив `git instaweb` в каталоге с вашим проектом. Если вы используете Mac, Leopard поставляется с предустановленным Ruby, поэтому webgrick может быть лучшим выбором. Чтобы запустить `instaweb` не с lighttpd, вы можете выполнить команду с параметром `--httpd`.

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

Это запустит сервер HTTPD на порту 1234, и затем запустит веб-браузер, открытый на этой странице. Это очень просто. Когда вы закончили и хотите остановить сервер, вы можете запустить ту же команду с параметром `--stop`:

```
$ git instaweb --httpd=webrick --stop
```

Если вы хотите иметь постоянно работающий веб-интерфейс на сервере для вашей команды или для проекта с открытым кодом на хостинге, вам необходимо установить CGI-сценарий на вашем обычном веб-сервере. В некоторых дистрибутивах Linux есть пакет `gitweb`, который вы можете установить, используя `apt` или `yum`, так что вы можете попробовать сначала этот способ. Мы рассмотрим установку GitWeb вручную очень вкратце. Для начала вам нужно получить исходный код Git, с которым поставляется GitWeb, и сгенерировать CGI-сценарий под свою систему:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" prefix=/usr gitweb
      SUBDIR gitweb
      SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
      GEN gitweb.cgi
      GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

Помните, что вы должны указать команде, где расположены ваши репозитории Git, с помощью переменной `GITWEB_PROJECTROOT`. Теперь вы должны настроить Apache на использование этого CGI-сценария, для чего вы можете добавить виртуальный хост:

```
<VirtualHost *:80>
  ServerName gitserver
  DocumentRoot /var/www/gitweb
  <Directory /var/www/gitweb>
    Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
```

```
AllowOverride All
order allow,deny
Allow from all
AddHandler cgi-script cgi
DirectoryIndex gitweb.cgi
</Directory>
</VirtualHost>
```

Повторюсь, GitWeb может быть установлен на любой веб-сервер, совместимый с CGI или Perl; если вы предпочитаете использовать что-то другое, настройка не должна стать для вас проблемой. К этому моменту вы должны иметь возможность зайти на <http://gitserver/> для просмотра ваших репозиториев онлайн.

GitLab

GitWeb довольно-таки прост. Если вам нужен более современный, полнофункциональный Git-сервер, есть несколько решений с открытым исходным кодом, которые можно использовать. Так как GitLab это один из самых популярных, мы рассмотрим его установку и использование в качестве примера. Это немного сложнее, чем GitWeb, и скорее всего потребует больше обслуживания, но и функционал гораздо богаче.

Установка

GitLab — это веб-приложение на основе базы данных, так что его установка немного сложней, чем у некоторых других серверов git. К счастью, этот процесс хорошо задокументирован и поддерживается.

Есть несколько подходов к установке GitLab. Для экономии времени, вы можете загрузить образ виртуальной машины или инсталлятор из <https://bitnami.com/stack/gitlab>, и поправить конфигурацию под своё окружение. Приятная возможность, включенная Bitnami, это экран входа (доступен по нажатию alt-→); он указывает IP-адрес и логин с паролем по умолчанию для установленного GitLab.



```
*** Welcome to the BitNami Gitlab Stack ***
*** Built using Ubuntu 12.04 - Kernel 3.2.0-53-virtual (tty2). ***

*** You can access the application at http://10.0.1.17 ***
*** The default username and password is 'user@example.com' and 'bitnami1'. ***
*** Please refer to http://wiki.bitnami.com/Virtual_Machines for details. ***

linux login: _
```

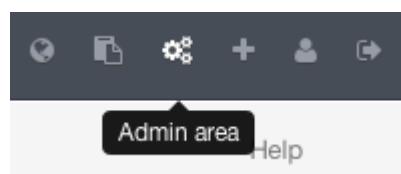
FIGURE 4-2

Экран входа виртуальной машины Bitnami GitLab.

Для всего остального, следуйте инструкциям GitLab Community Edition, которые можно найти на <https://gitlab.com/gitlab-org/gitlab-ce/tree/master>. Там вы найдёте помощь по установке GitLab посредством рецептов Chef, на виртуальную машину Digital Ocean, или из RPM или DEB пакетов (которые на момент написания всё находились в бета-версии). Есть также “неофициальная” инструкция по запуску GitLab на нестандартных операционных системах и базах данных, полностью ручной сценарий установки и множество других тем.

Администрирование

Административный интерфейс GitLab доступен через веб. Просто направьте ваш браузер на имя или IP-адрес хоста, где установлен GitLab, и войдите как администратор. Имя пользователя по умолчанию `admin@local.host`, пароль по умолчанию `5iveL!fe` (вас попросят изменить их при входе). Войдя, нажмите иконку “Административная зона” в меню справа и сверху.

**FIGURE 4-3**

Пункт “Административная зона” в меню GitLab.

ПОЛЬЗОВАТЕЛИ

Пользователи в GitLab — это учётные записи, соответствующие людям. Пользовательские учётные записи не очень сложны; в основном это набор персональной информации, прикреплённый к имени. У каждого пользователя есть **пространство имён**, логически группирующее проекты данного пользователя. Если у пользователя *jane* есть проект *project*, адрес этого проекта будет <http://server/jane/project>.

FIGURE 4-4

Экран управления пользователями GitLab.

Users (3)		
Administrator (Admin) It's you!	user@example.com	<button>Edit</button>
Rachel Myers (Admin)	foo@example.com	<button>Edit</button> <button>Block</button> <button>Destroy</button>
Russell Beiffer	bar@example.com	<button>Edit</button> <button>Block</button> <button>Destroy</button>

Удаление пользователя может быть выполнено двумя способами. “Блокирование” (“Blocking”) пользователя запрещает ему вход в GitLab, но все данные в его пространстве имен сохраняются, и коммиты, подписанные этим пользователем, будут указывать на его профиль.

“Разрушение” (“Destroying”) пользователя, с другой стороны, полностью удаляет его из базы данных и файловой системы. Все проекты и данные в его пространстве имен удаляются, как и все принадлежащие ему группы. Конечно, этим более постоянным и разрушительным действием пользуются реже.

ГРУППЫ

Группы GitLab — это коллекция проектов с указанием того, как пользователи получают к ним доступ. Каждая группа имеет пространство имён проектов (так же как и пользователи), так что если в группе *training* есть проект *materials*, его адрес будет <http://server/training/materials>.

FIGURE 4-5

Экран управления группами GitLab.

Каждая группа связана с пользователями, каждый из которых имеет уровень доступа к проектам группы и к самой группе. Он разнится от “Гостя” (“Guest”, только проблемы и чат) до “Владельца” (“Owner”, полный контроль над группой, её членами и проектами). Типы разрешений слишком обширны, чтобы перечислять их здесь, но на экране управления GitLab есть полезная ссылка с описанием.

ПРОЕКТЫ

Проект GitLab примерно соответствует одному git-репозиторию. Каждый проект принадлежит одному пространству имён, групповому или пользовательскому. Если проект принадлежит пользователю, владелец контролирует, кто имеет доступ к проекту; если проект принадлежит группе, действуют групповые уровни доступа для пользователей.

Каждый проект также имеет уровень видимости, который контролирует, кто имеет доступ на чтение страниц проекта или репозитория. Если проект *Приватный* (*Private*), владелец должен явно дать доступ на чтение отдельным пользователям. *Внутренний* (*Internal*) проект виден любому вошедшему пользователю GitLab, а *Публичный* (*Public*) проект видим всем. Это относится как к доступу git “fetch”, так и к доступу к проекту через веб-интерфейс.

ХУКИ

GitLab включает поддержку хуков (перехватчиков, hooks) на уровне проектов и всей системы. В обоих случаях, когда происходит

некоторое событие, сервер GitLab выполняет запрос HTTP POST с осмысленным JSON-содержанием. Это отличный способ соединить ваши git-репозитории и инсталляцию GitLab с автоматикой инфраструктуры разработки, такой как сервера непрерывной интеграции, комнаты чатов или инструменты деплоя.

Базовое использование

Первое, чего вы захотите от GitLab, это создать новый проект. Это достигается нажатием иконки “+” на панели инструментов. Будут запрошены имя проекта, пространство имён, которому он должен принадлежать, и уровень видимости. Большинство из этих настроек можно потом изменить через интерфейс настроек. Нажмите “Создать проект” (“Create Project”), чтобы закончить.

Когда проект создан, вы, наверное, захотите соединить его с локальным git-репозиторием. Каждый проект может быть доступен через HTTPS или SSH, каждый из которых может быть использован для указания удалённого репозитория. Адреса (URL) видимы наверху домашней страницы проекта. Для существующего локального репозитория, следующая команда создаст удалённый репозиторий с именем `gitlab` и размещением на сервере:

```
$ git remote add gitlab https://server/namespace/project.git
```

Если у вас нет локального репозитория, можно просто сделать его:

```
$ git clone https://server/namespace/project.git
```

Веб-интерфейс даёт доступ к некоторым полезным видам самого репозитория. Домашняя страница каждого проекта показывает недавнюю активность, а ссылки наверху ведут на список файлов проекта и журнала коммитов.

Совместная работа

Самый простой метод совместной работы над проектом GitLab – это выдача другому пользователю прямого доступа на запись (push) в git-репозитории. Вы можете добавить пользователя в проект в разделе “Участники” (“Members”) настроек проекта, указав уровень доступа (уровни доступа кратко обсуждались в “Группы”). Получая уровень

доступа “Разработчик” (“Developer”) или выше, пользователь может невозбранно отсылать свои коммиты и ветки непосредственно в репозиторий.

Другой, более разобщённый способ совместной работы — использование запросов на слияние (merge requests). Эта возможность позволяет любому пользователю, который видит проект, вносить свой вклад подконтрольным способом. Пользователи с прямым доступом могут просто создать ветку, отослать в неё коммиты и открыть запрос на слияние из их ветки обратно в master или любую другую ветку. Пользователи без доступа на запись могут “форкнуть” репозиторий (“fork”, создать собственную копию), отправить коммиты в эту копию и открыть запрос на слияние из их форка обратно в основной проект. Эта модель позволяет владельцу полностью контролировать, что попадает в репозиторий и когда, принимая помощь от недоверенных пользователей.

Запросы на слияние и проблемы (issues) это основные единицы долгоживущих дискуссий в GitLab. Каждый запрос на слияние допускает построчное обсуждение предлагаемого изменения (поддерживая облегчённое рецензирование кода), равно как и общее обсуждение. И те и другие могут присваиваться пользователям или организовываться в вехи (milestones).

Мы в основном сосредоточились на частях GitLab, связанных с git, но это — довольно зрелая система, и она предоставляет много других возможностей, помогающих вашей команде работать совместно, например вики-страницы для проектов и инструменты поддержки системы. Одно из преимуществ GitLab в том, что, однажды запустив и настроив сервер, вам редко придётся изменять конфигурацию или заходить на него по SSH; большинство административных и пользовательских действий можно выполнять через веб-браузер.

Git-хостинг

Если вы не хотите связываться со всей работой по установке собственного Git-сервера, у вас есть несколько вариантов размещения ваших Git-проектов на внешних специальных хостинг сайтах. Это предоставляет множество преимуществ: на хостинг сайте обычно можно быстро настроить и запустить проект, и не требуется никакого мониторинга или поддержки сервера. Даже если вы установили и запустили свой собственный внутренний сервер, вы можете захотеть использовать публичный хостинг сайт для вашего

открытого кода — обычно сообществу открытого кода так будет проще вас найти и помочь.

В наши дни у вас есть огромное количество вариантов хостинга на выбор, все со своими преимуществами и недостатками. Чтобы увидеть актуальный список, проверьте страницу GitHosting в главной вики Git: <https://git.kernel.org/index.php/GitHosting>

Мы в деталях рассмотрим GitHub в [Chapter 6](#), так как это крупнейший Git-хостинг и вам может понадобиться взаимодействовать с проектами, хостящимися на нём, но есть и множество других, если вы не хотите устанавливать свой Git-сервер.

Заключение

Существует несколько вариантов получения удалённого Git-репозитория, чтобы принять участие в совместном проекте или поделиться своими наработками.

Запуск своего сервера даёт полный контроль и позволяет запускать его за вашим сетевым экраном, но такой сервер обычно требует значительного времени на настройку и поддержку. В случае размещения данных на хостинге, его просто настроить и поддерживать; однако вам необходимо иметь возможность хранить код на чужом сервере, а некоторые организации этого не позволяют.

Выбор решения или сочетания решений, которое подойдет вам и вашей организации, не должен вызвать затруднений.

5

Распределенный Git

Теперь, когда вы обзавелись настроенным удалённым Git-репозиторием, т.е. местом, где разработчики могут обмениваться своим кодом, а также познакомились с основными командами Git'а для локальной работы, мы рассмотрим, как задействовать некоторые распределённые рабочие процессы, предлагаемые Git'ом

В этой главе мы рассмотрим работу с Git'ом в распределённой среде как в роли рядового разработчика, так и в роли системного интегратора. То есть вы научитесь успешно вносить свой код в проект, делая это как можно более просто и для вас, и для владельца проекта, а также научитесь тому, как сопровождать проекты, в работе в которых участвует множество разработчиков.

Distributed Workflows

Unlike Centralized Version Control Systems (CVCSs), the distributed nature of Git allows you to be far more flexible in how developers collaborate on projects. In centralized systems, every developer is a node working more or less equally on a central hub. In Git, however, every developer is potentially both a node and a hub – that is, every developer can both contribute code to other repositories and maintain a public repository on which others can base their work and which they can contribute to. This opens a vast range of workflow possibilities for your project and/or your team, so we'll cover a few common paradigms that take advantage of this flexibility. We'll go over the strengths and possible weaknesses of each design; you can choose a single one to use, or you can mix and match features from each.

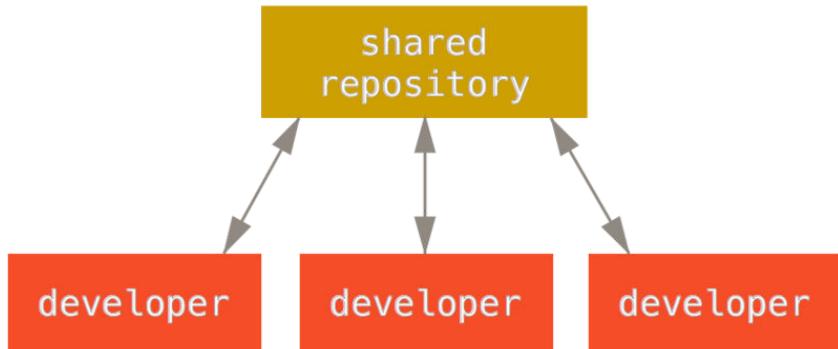
Centralized Workflow

In centralized systems, there is generally a single collaboration model—the centralized workflow. One central hub, or repository, can accept code, and every-

one synchronizes their work to it. A number of developers are nodes – consumers of that hub – and synchronize to that one place.

FIGURE 5-1

Centralized workflow.



This means that if two developers clone from the hub and both make changes, the first developer to push their changes back up can do so with no problems. The second developer must merge in the first one's work before pushing changes up, so as not to overwrite the first developer's changes. This concept is as true in Git as it is in Subversion (or any CVCS), and this model works perfectly well in Git.

If you are already comfortable with a centralized workflow in your company or team, you can easily continue using that workflow with Git. Simply set up a single repository, and give everyone on your team push access; Git won't let users overwrite each other. Say John and Jessica both start working at the same time. John finishes his change and pushes it to the server. Then Jessica tries to push her changes, but the server rejects them. She is told that she's trying to push non-fast-forward changes and that she won't be able to do so until she fetches and merges. This workflow is attractive to a lot of people because it's a paradigm that many are familiar and comfortable with.

This is also not limited to small teams. With Git's branching model, it's possible for hundreds of developers to successfully work on a single project through dozens of branches simultaneously.

Integration-Manager Workflow

Because Git allows you to have multiple remote repositories, it's possible to have a workflow where each developer has write access to their own public repository and read access to everyone else's. This scenario often includes a canonical repository that represents the "official" project. To contribute to that

project, you create your own public clone of the project and push your changes to it. Then, you can send a request to the maintainer of the main project to pull in your changes. The maintainer can then add your repository as a remote, test your changes locally, merge them into their branch, and push back to their repository. The process works as follows (see **Figure 5-2**):

1. The project maintainer pushes to their public repository.
2. A contributor clones that repository and makes changes.
3. The contributor pushes to their own public copy.
4. The contributor sends the maintainer an e-mail asking them to pull changes.
5. The maintainer adds the contributor's repo as a remote and merges locally.
6. The maintainer pushes merged changes to the main repository.

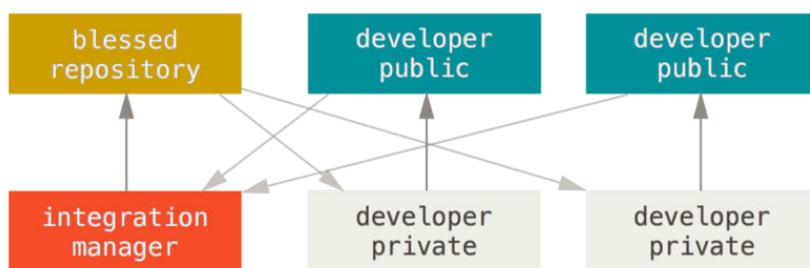


FIGURE 5-2
Integration-manager workflow.

This is a very common workflow with hub-based tools like GitHub or GitLab, where it's easy to fork a project and push your changes into your fork for everyone to see. One of the main advantages of this approach is that you can continue to work, and the maintainer of the main repository can pull in your changes at any time. Contributors don't have to wait for the project to incorporate their changes – each party can work at their own pace.

Dictator and Lieutenants Workflow

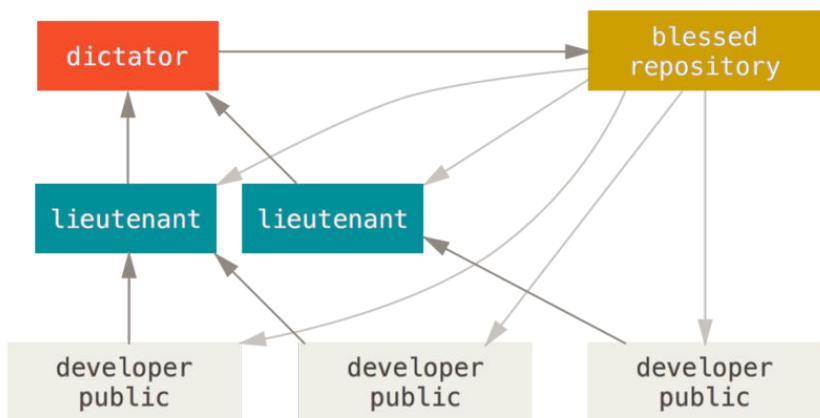
This is a variant of a multiple-repository workflow. It's generally used by huge projects with hundreds of collaborators; one famous example is the Linux kernel. Various integration managers are in charge of certain parts of the repository; they're called lieutenants. All the lieutenants have one integration manager known as the benevolent dictator. The benevolent dictator's repository serves

as the reference repository from which all the collaborators need to pull. The process works like this (see **Figure 5-3**):

1. Regular developers work on their topic branch and rebase their work on top of `master`. The `master` branch is that of the dictator.
2. Lieutenants merge the developers' topic branches into their `master` branch.
3. The dictator merges the lieutenants' `master` branches into the dictator's `master` branch.
4. The dictator pushes their `master` to the reference repository so the other developers can rebase on it.

FIGURE 5-3

Benevolent dictator workflow.



This kind of workflow isn't common, but can be useful in very big projects, or in highly hierarchical environments. It allows the project leader (the dictator) to delegate much of the work and collect large subsets of code at multiple points before integrating them.

Workflows Summary

These are some commonly used workflows that are possible with a distributed system like Git, but you can see that many variations are possible to suit your particular real-world workflow. Now that you can (hopefully) determine which workflow combination may work for you, we'll cover some more specific examples of how to accomplish the main roles that make up the different flows. In

the next section, you'll learn about a few common patterns for contributing to a project.

Contributing to a Project

The main difficulty with describing how to contribute to a project is that there are a huge number of variations on how it's done. Because Git is very flexible, people can and do work together in many ways, and it's problematic to describe how you should contribute – every project is a bit different. Some of the variables involved are active contributor count, chosen workflow, your commit access, and possibly the external contribution method.

The first variable is active contributor count – how many users are actively contributing code to this project, and how often? In many instances, you'll have two or three developers with a few commits a day, or possibly less for somewhat dormant projects. For larger companies or projects, the number of developers could be in the thousands, with hundreds or thousands of commits coming in each day. This is important because with more and more developers, you run into more issues with making sure your code applies cleanly or can be easily merged. Changes you submit may be rendered obsolete or severely broken by work that is merged in while you were working or while your changes were waiting to be approved or applied. How can you keep your code consistently up to date and your commits valid?

The next variable is the workflow in use for the project. Is it centralized, with each developer having equal write access to the main codeline? Does the project have a maintainer or integration manager who checks all the patches? Are all the patches peer-reviewed and approved? Are you involved in that process? Is a lieutenant system in place, and do you have to submit your work to them first?

The next issue is your commit access. The workflow required in order to contribute to a project is much different if you have write access to the project than if you don't. If you don't have write access, how does the project prefer to accept contributed work? Does it even have a policy? How much work are you contributing at a time? How often do you contribute?

All these questions can affect how you contribute effectively to a project and what workflows are preferred or available to you. We'll cover aspects of each of these in a series of use cases, moving from simple to more complex; you should be able to construct the specific workflows you need in practice from these examples.

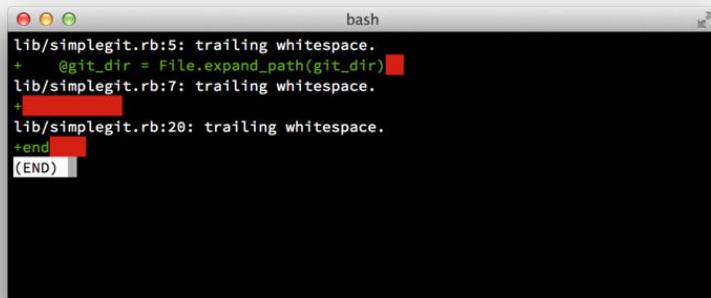
Commit Guidelines

Before we start looking at the specific use cases, here's a quick note about commit messages. Having a good guideline for creating commits and sticking to it makes working with Git and collaborating with others a lot easier. The Git project provides a document that lays out a number of good tips for creating commits from which to submit patches – you can read it in the Git source code in the [Documentation/SubmittingPatches](#) file.

First, you don't want to submit any whitespace errors. Git provides an easy way to check for this – before you commit, run `git diff --check`, which identifies possible whitespace errors and lists them for you.

FIGURE 5-4

Output of `git diff --check`.



```
bash
lib/simplegit.rb:5: trailing whitespace.
+ @git_dir = File.expand_path(git_dir)
lib/simplegit.rb:7: trailing whitespace.
+
lib/simplegit.rb:20: trailing whitespace.
+end
(END)
```

If you run that command before committing, you can tell if you're about to commit whitespace issues that may annoy other developers.

Next, try to make each commit a logically separate changeset. If you can, try to make your changes digestible – don't code for a whole weekend on five different issues and then submit them all as one massive commit on Monday. Even if you don't commit during the weekend, use the staging area on Monday to split your work into at least one commit per issue, with a useful message per commit. If some of the changes modify the same file, try to use `git add --patch` to partially stage files (covered in detail in “[Интерактивное индексирование](#)”). The project snapshot at the tip of the branch is identical whether you do one commit or five, as long as all the changes are added at some point, so try to make things easier on your fellow developers when they have to review your changes. This approach also makes it easier to pull out or revert one of the changesets if you need to later. “[Исправление истории](#)”

describes a number of useful Git tricks for rewriting history and interactively staging files – use these tools to help craft a clean and understandable history before sending the work to someone else.

The last thing to keep in mind is the commit message. Getting in the habit of creating quality commit messages makes using and collaborating with Git a lot easier. As a general rule, your messages should start with a single line that's no more than about 50 characters and that describes the changeset concisely, followed by a blank line, followed by a more detailed explanation. The Git project requires that the more detailed explanation include your motivation for the change and contrast its implementation with previous behavior – this is a good guideline to follow. It's also a good idea to use the imperative present tense in these messages. In other words, use commands. Instead of “I added tests for” or “Adding tests for,” use “Add tests for.” Here is a template originally written by Tim Pope:

Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); tools like rebase can get confused if you run the two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here

If all your commit messages look like this, things will be a lot easier for you and the developers you work with. The Git project has well-formatted commit messages – try running `git log --no-merges` there to see what a nicely formatted project-commit history looks like.

In the following examples, and throughout most of this book, for the sake of brevity this book doesn't have nicely-formatted messages like this; instead, we use the `-m` option to `git commit`. Do as we say, not as we do.

Private Small Team

The simplest setup you're likely to encounter is a private project with one or two other developers. "Private," in this context, means closed-source – not accessible to the outside world. You and the other developers all have push access to the repository.

In this environment, you can follow a workflow similar to what you might do when using Subversion or another centralized system. You still get the advantages of things like offline committing and vastly simpler branching and merging, but the workflow can be very similar; the main difference is that merges happen client-side rather than on the server at commit time. Let's see what it might look like when two developers start to work together with a shared repository. The first developer, John, clones the repository, makes a change, and commits locally. (The protocol messages have been replaced with ... in these examples to shorten them somewhat.)

```
# John's Machine
$ git clone john@githost:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

The second developer, Jessica, does the same thing – clones the repository and commits a change:

```
# Jessica's Machine
$ git clone jessica@githost:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Now, Jessica pushes her work up to the server:

```
# Jessica's Machine
$ git push origin master
```

```
...
To jessica@githost:simplegit.git
  1edee6b..fbff5bc  master -> master
```

John tries to push his change up, too:

```
# John's Machine
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]      master -> master (non-fast forward)
error: failed to push some refs to 'john@githost:simplegit.git'
```

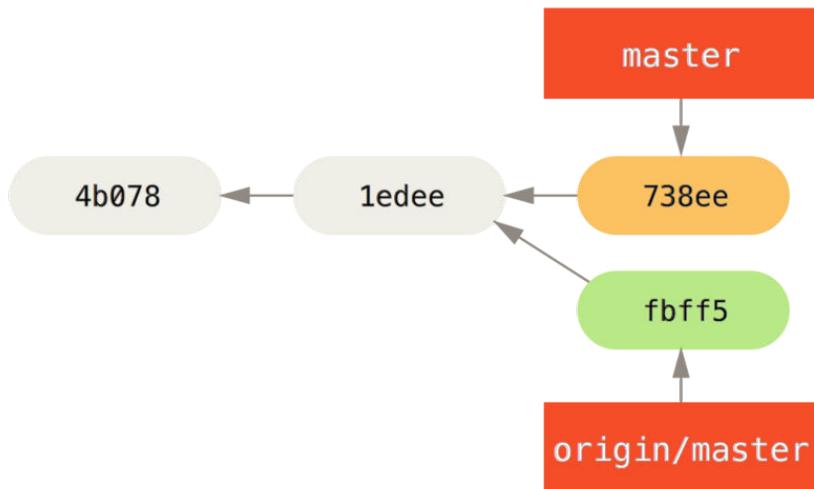
John isn't allowed to push because Jessica has pushed in the meantime. This is especially important to understand if you're used to Subversion, because you'll notice that the two developers didn't edit the same file. Although Subversion automatically does such a merge on the server if different files are edited, in Git you must merge the commits locally. John has to fetch Jessica's changes and merge them in before he will be allowed to push:

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master      -> origin/master
```

At this point, John's local repository looks something like this:

FIGURE 5-5

John's divergent history.



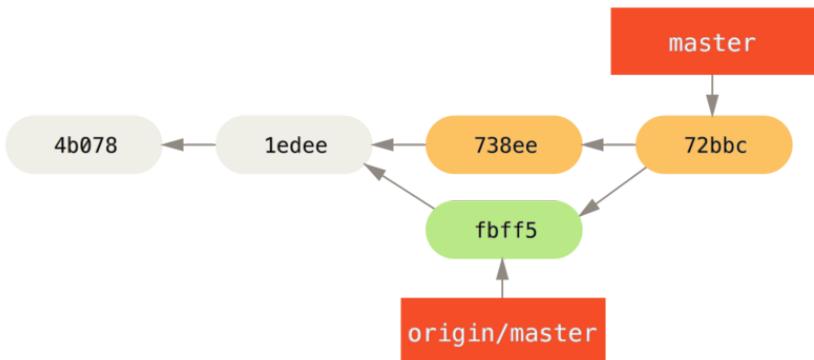
John has a reference to the changes Jessica pushed up, but he has to merge them into his own work before he is allowed to push:

```
$ git merge origin/master
Merge made by recursive.
 TODO | 1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

The merge goes smoothly – John's commit history now looks like this:

FIGURE 5-6

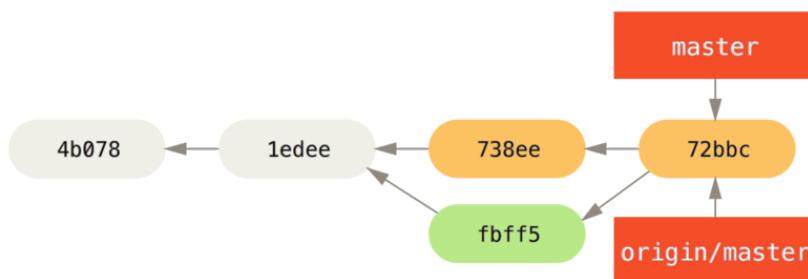
John's repository after merging origin/master.



Now, John can test his code to make sure it still works properly, and then he can push his new merged work up to the server:

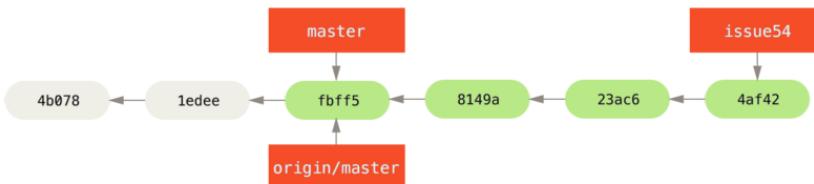
```
$ git push origin master
...
To john@githost:simplegit.git
  fbff5bc..72bbc59  master -> master
```

Finally, John's commit history looks like this:

**FIGURE 5-7**

John's history after pushing to the origin server.

In the meantime, Jessica has been working on a topic branch. She's created a topic branch called `issue54` and done three commits on that branch. She hasn't fetched John's changes yet, so her commit history looks like this:

**FIGURE 5-8**

Jessica's topic branch.

Jessica wants to sync up with John, so she fetches:

```
# Jessica's Machine
$ git fetch origin
...

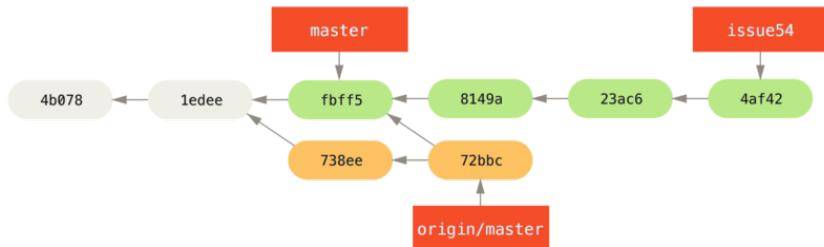
```

```
From jessica@githost:simplegit
fbff5bc..72bbc59  master      -> origin/master
```

That pulls down the work John has pushed up in the meantime. Jessica's history now looks like this:

FIGURE 5-9

Jessica's history after fetching John's changes.



Jessica thinks her topic branch is ready, but she wants to know what she has to merge into her work so that she can push. She runs `git log` to find out:

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

        removed invalid default value
```

The `issue54..origin/master` syntax is a log filter that asks Git to only show the list of commits that are on the latter branch (in this case `origin/master`) that are not on the first branch (in this case `issue54`). We'll go over this syntax in detail in “[Диапазоны фиксаций](#)”.

For now, we can see from the output that there is a single commit that John has made that Jessica has not merged in. If she merges `origin/master`, that is the single commit that will modify her local work.

Now, Jessica can merge her topic work into her `master` branch, merge John's work (`origin/master`) into her `master` branch, and then push back to the server again. First, she switches back to her `master` branch to integrate all this work:

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

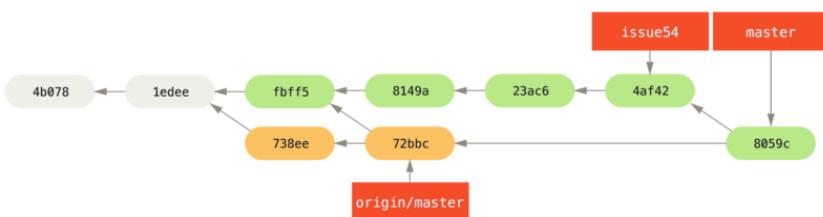
She can merge either `origin/master` or `issue54` first – they're both upstream, so the order doesn't matter. The end snapshot should be identical no matter which order she chooses; only the history will be slightly different. She chooses to merge in `issue54` first:

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

No problems occur; as you can see it was a simple fast-forward. Now Jessica merges in John's work (`origin/master`):

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 ++
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Everything merges cleanly, and Jessica's history looks like this:

**FIGURE 5-10**

Jessica's history after merging John's changes.

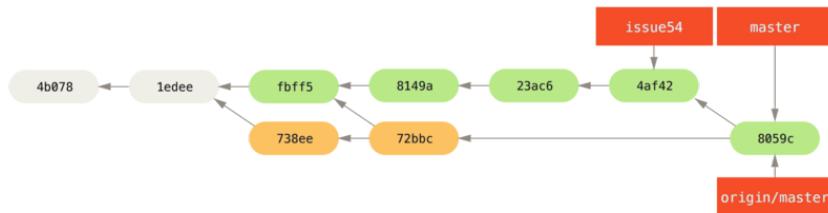
Now `origin/master` is reachable from Jessica's `master` branch, so she should be able to successfully push (assuming John hasn't pushed again in the meantime):

```
$ git push origin master
...
To jessica@githost:simplegit.git
  72bbc59..8059c15  master -> master
```

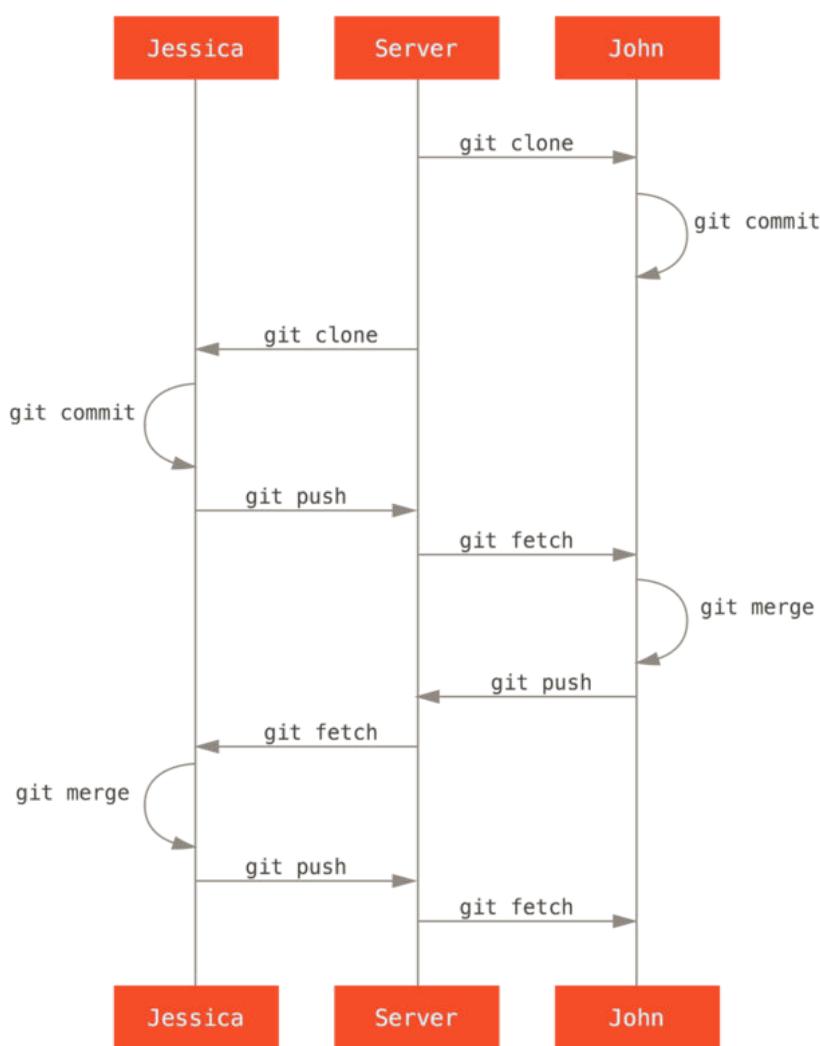
Each developer has committed a few times and merged each other's work successfully.

FIGURE 5-11

Jessica's history after pushing all changes back to the server.



That is one of the simplest workflows. You work for a while, generally in a topic branch, and merge into your master branch when it's ready to be integrated. When you want to share that work, you merge it into your own master branch, then fetch and merge `origin/master` if it has changed, and finally push to the `master` branch on the server. The general sequence is something like this:

**FIGURE 5-12**

General sequence of events for a simple multiple-developer Git workflow.

Private Managed Team

In this next scenario, you'll look at contributor roles in a larger private group. You'll learn how to work in an environment where small groups collaborate on features and then those team-based contributions are integrated by another party.

Let's say that John and Jessica are working together on one feature, while Jessica and Josie are working on a second. In this case, the company is using a

type of integration-manager workflow where the work of the individual groups is integrated only by certain engineers, and the `master` branch of the main repo can be updated only by those engineers. In this scenario, all work is done in team-based branches and pulled together by the integrators later.

Let's follow Jessica's workflow as she works on her two features, collaborating in parallel with two different developers in this environment. Assuming she already has her repository cloned, she decides to work on `featureA` first. She creates a new branch for the feature and does some work on it there:

```
# Jessica's Machine
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ vim lib/simplegit.rb
$ git commit -am 'add limit to log function'
[featureA 3300904] add limit to log function
 1 files changed, 1 insertions(+), 1 deletions(-)
```

At this point, she needs to share her work with John, so she pushes her `featureA` branch commits up to the server. Jessica doesn't have push access to the `master` branch – only the integrators do – so she has to push to another branch in order to collaborate with John:

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica e-mails John to tell him that she's pushed some work into a branch named `featureA` and he can look at it now. While she waits for feedback from John, Jessica decides to start working on `featureB` with Josie. To begin, she starts a new feature branch, basing it off the server's `master` branch:

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

Now, Jessica makes a couple of commits on the `featureB` branch:

```
$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
```

```
[featureB e5b0fdc] made the ls-tree function recursive
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Jessica's repository looks like this:

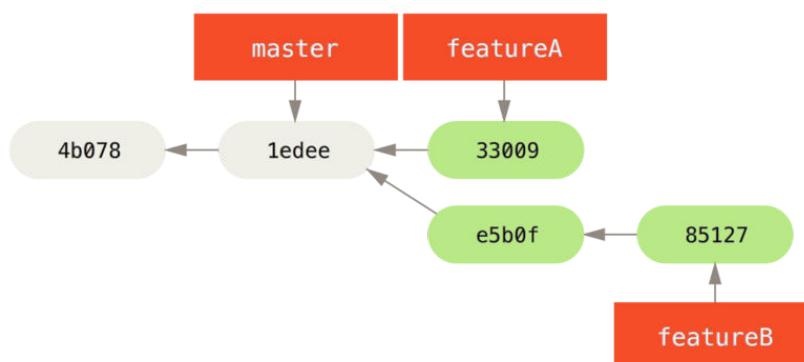


FIGURE 5-13

Jessica's initial commit history.

She's ready to push up her work, but gets an e-mail from Josie that a branch with some initial work on it was already pushed to the server as `featureBee`. Jessica first needs to merge those changes in with her own before she can push to the server. She can then fetch Josie's changes down with `git fetch`:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Jessica can now merge this into the work she did with `git merge`:

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
lib/simplegit.rb |    4 +---
1 files changed, 4 insertions(+), 0 deletions(-)
```

There is a bit of a problem – she needs to push the merged work in her `featureB` branch to the `featureBee` branch on the server. She can do so by specifying the local branch followed by a colon (`:`) followed by the remote branch to the `git push` command:

```
$ git push -u origin featureB:featureBee
...
To jessica@githost:simplegit.git
  fba9af8..cd685d1  featureB -> featureBee
```

This is called a *refspec*. See “[Спецификации ссылок](#)” for a more detailed discussion of Git refspecs and different things you can do with them. Also notice the `-u` flag; this is short for `--set-upstream`, which configures the branches for easier pushing and pulling later.

Next, John e-mails Jessica to say he’s pushed some changes to the `featureA` branch and asks her to verify them. She runs a `git fetch` to pull down those changes:

```
$ git fetch origin
...
From jessica@githost:simplegit
  3300904..aad881d  featureA -> origin/featureA
```

Then, she can see what has been changed with `git log`:

```
$ git log featureA..origin/featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

changed log output to 30 from 25
```

Finally, she merges John’s work into her own `featureA` branch:

```
$ git checkout featureA
Switched to branch 'featureA'
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
  lib/simplegit.rb |    10 ++++++----
  1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica wants to tweak something, so she commits again and then pushes this back up to the server:

```
$ git commit -am 'small tweak'
[featureA 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
 3300904..774b3ed  featureA -> featureA
```

Jessica's commit history now looks something like this:

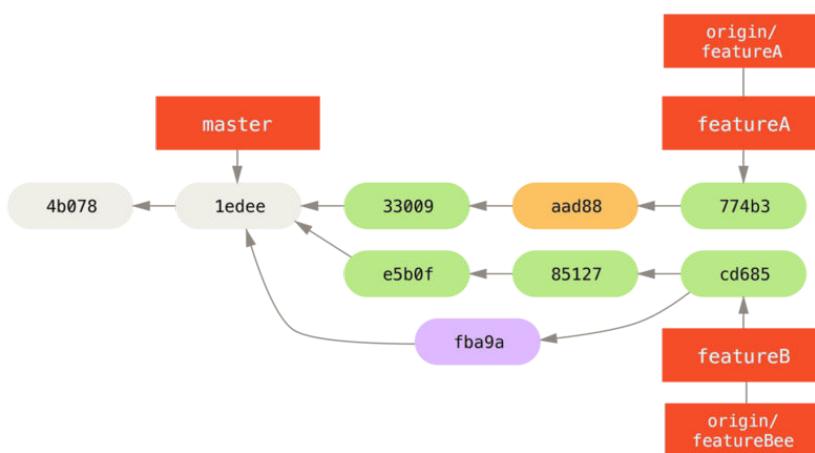
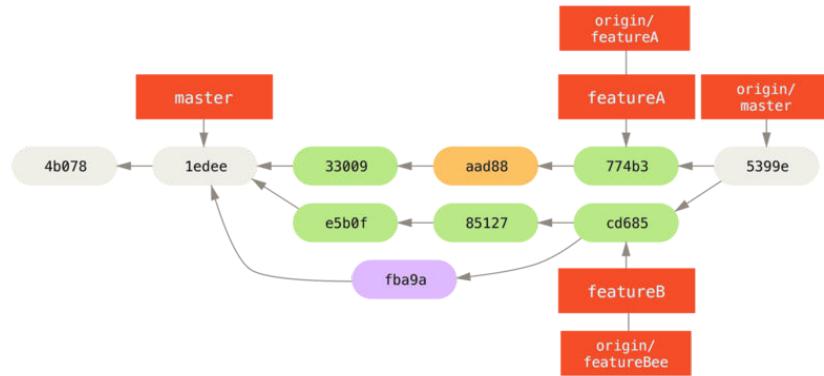


FIGURE 5-14
Jessica's history
after committing on
a feature branch.

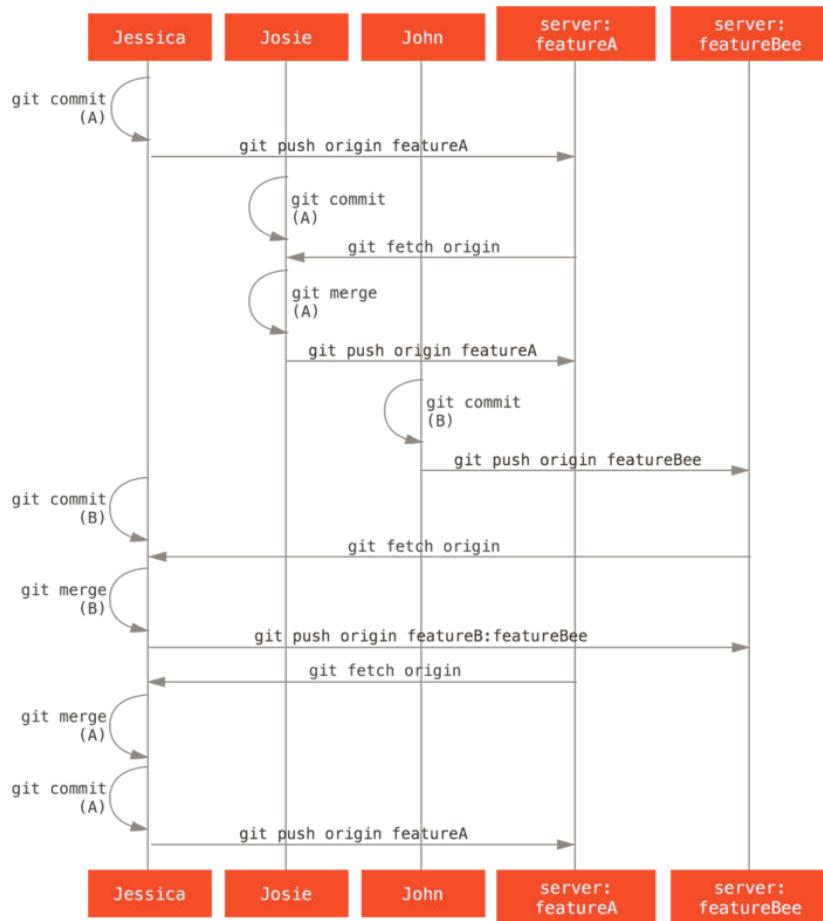
Jessica, Josie, and John inform the integrators that the `featureA` and `featureBee` branches on the server are ready for integration into the mainline. After the integrators merge these branches into the mainline, a fetch will bring down the new merge commit, making the history look like this:

FIGURE 5-15

*Jessica's history
after merging both
her topic branches.*



Many groups switch to Git because of this ability to have multiple teams working in parallel, merging the different lines of work late in the process. The ability of smaller subgroups of a team to collaborate via remote branches without necessarily having to involve or impede the entire team is a huge benefit of Git. The sequence for the workflow you saw here is something like this:

**FIGURE 5-16**

Basic sequence of this managed-team workflow.

Forked Public Project

Contributing to public projects is a bit different. Because you don't have the permissions to directly update branches on the project, you have to get the work to the maintainers some other way. This first example describes contributing via forking on Git hosts that support easy forking. Many hosting sites support this (including GitHub, BitBucket, Google Code, repo.or.cz, and others), and many project maintainers expect this style of contribution. The next section deals with projects that prefer to accept contributed patches via e-mail.

First, you'll probably want to clone the main repository, create a topic branch for the patch or patch series you're planning to contribute, and do your work there. The sequence looks basically like this:

```
$ git clone (url)
$ cd project
$ git checkout -b featureA
# (work)
$ git commit
# (work)
$ git commit
```

You may want to use `rebase -i` to squash your work down to a single commit, or rearrange the work in the commits to make the patch easier for the maintainer to review – see “[Исправление истории](#)” for more information about interactive rebasing.

When your branch work is finished and you’re ready to contribute it back to the maintainers, go to the original project page and click the “Fork” button, creating your own writable fork of the project. You then need to add in this new repository URL as a second remote, in this case named `myfork`:

```
$ git remote add myfork (url)
```

Then you need to push your work up to it. It’s easiest to push the topic branch you’re working on up to your repository, rather than merging into your master branch and pushing that up. The reason is that if the work isn’t accepted or is cherry picked, you don’t have to rewind your master branch. If the maintainers merge, rebase, or cherry-pick your work, you’ll eventually get it back via pulling from their repository anyhow:

```
$ git push -u myfork featureA
```

When your work has been pushed up to your fork, you need to notify the maintainer. This is often called a pull request, and you can either generate it via the website – GitHub has its own Pull Request mechanism that we’ll go over in [Chapter 6](#) – or you can run the `git request-pull` command and e-mail the output to the project maintainer manually.

The `request-pull` command takes the base branch into which you want your topic branch pulled and the Git repository URL you want them to pull from, and outputs a summary of all the changes you’re asking to be pulled in. For instance, if Jessica wants to send John a pull request, and she’s done two commits on the topic branch she just pushed up, she can run this:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
    added a new function

are available in the git repository at:

  git://githost/simplegit.git featureA

  Jessica Smith (2):
    add limit to log function
    change log output to 30 from 25

  lib/simplegit.rb |   10 ++++++++-+
  1 files changed, 9 insertions(+), 1 deletions(-)
```

The output can be sent to the maintainer—it tells them where the work was branched from, summarizes the commits, and tells where to pull this work from.

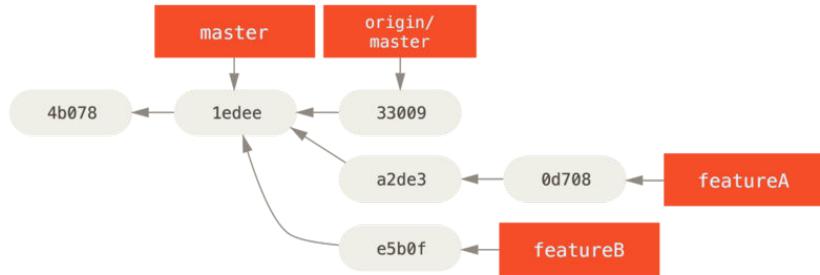
On a project for which you’re not the maintainer, it’s generally easier to have a branch like `master` always track `origin/master` and to do your work in topic branches that you can easily discard if they’re rejected. Having work themes isolated into topic branches also makes it easier for you to rebase your work if the tip of the main repository has moved in the meantime and your commits no longer apply cleanly. For example, if you want to submit a second topic of work to the project, don’t continue working on the topic branch you just pushed up – start over from the main repository’s `master` branch:

```
$ git checkout -b featureB origin/master
# (work)
$ git commit
$ git push myfork featureB
# (email maintainer)
$ git fetch origin
```

Now, each of your topics is contained within a silo – similar to a patch queue – that you can rewrite, rebase, and modify without the topics interfering or interdepending on each other, like so:

FIGURE 5-17

Initial commit history with featureB work.



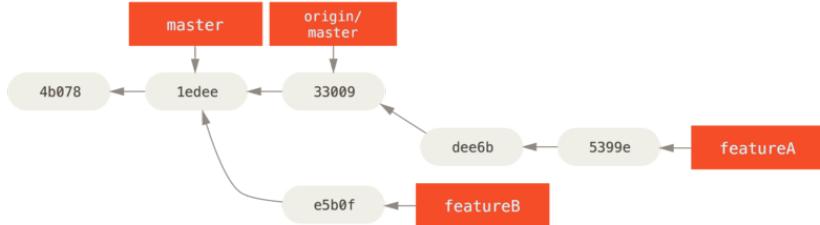
Let's say the project maintainer has pulled in a bunch of other patches and tried your first branch, but it no longer cleanly merges. In this case, you can try to rebase that branch on top of `origin/master`, resolve the conflicts for the maintainer, and then resubmit your changes:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

This rewrites your history to now look like **Figure 5-18**.

FIGURE 5-18

Commit history after featureA work.



Because you rebased the branch, you have to specify the `-f` to your push command in order to be able to replace the `featureA` branch on the server with a commit that isn't a descendant of it. An alternative would be to push this new work to a different branch on the server (perhaps called `featureAv2`).

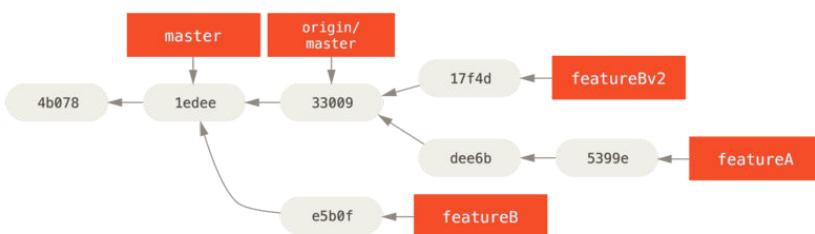
Let's look at one more possible scenario: the maintainer has looked at work in your second branch and likes the concept but would like you to change an implementation detail. You'll also take this opportunity to move the work to be

based off the project's current `master` branch. You start a new branch based off the current `origin/master` branch, squash the `featureB` changes there, resolve any conflicts, make the implementation change, and then push that up as a new branch:

```
$ git checkout -b featureBv2 origin/master
$ git merge --no-commit --squash featureB
# (change implementation)
$ git commit
$ git push myfork featureBv2
```

The `--squash` option takes all the work on the merged branch and squashes it into one non-merge commit on top of the branch you're on. The `--no-commit` option tells Git not to automatically record a commit. This allows you to introduce all the changes from another branch and then make more changes before recording the new commit.

Now you can send the maintainer a message that you've made the requested changes and they can find those changes in your `featureBv2` branch.

**FIGURE 5-19**

Commit history after featureBv2 work.

Public Project over E-Mail

Many projects have established procedures for accepting patches – you'll need to check the specific rules for each project, because they will differ. Since there are several older, larger projects which accept patches via a developer mailing list, we'll go over an example of that now.

The workflow is similar to the previous use case – you create topic branches for each patch series you work on. The difference is how you submit them to the project. Instead of forking the project and pushing to your own writable version, you generate e-mail versions of each commit series and e-mail them to the developer mailing list:

```
$ git checkout -b topicA
# (work)
$ git commit
# (work)
$ git commit
```

Now you have two commits that you want to send to the mailing list. You use `git format-patch` to generate the mbox-formatted files that you can e-mail to the list – it turns each commit into an e-mail message with the first line of the commit message as the subject and the rest of the message plus the patch that the commit introduces as the body. The nice thing about this is that applying a patch from an e-mail generated with `format-patch` preserves all the commit information properly.

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
```

The `format-patch` command prints out the names of the patch files it creates. The `-M` switch tells Git to look for renames. The files end up looking like this:

```
$ cat 0001-add-limit-to-log-function.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20

---
lib/simplegit.rb |    2 ++
1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
end

def log(treeish = 'master')
-   command("git log #{treeish}")
+   command("git log -n 20 #{treeish}")
```

```
end

def ls_tree(treeish = 'master')
-- 
2.1.0
```

You can also edit these patch files to add more information for the e-mail list that you don't want to show up in the commit message. If you add text between the `--` line and the beginning of the patch (the `diff --git` line), then developers can read it; but applying the patch excludes it.

To e-mail this to a mailing list, you can either paste the file into your e-mail program or send it via a command-line program. Pasting the text often causes formatting issues, especially with “smarter” clients that don't preserve newlines and other whitespace appropriately. Luckily, Git provides a tool to help you send properly formatted patches via IMAP, which may be easier for you. We'll demonstrate how to send a patch via Gmail, which happens to be the e-mail agent we know best; you can read detailed instructions for a number of mail programs at the end of the aforementioned Documentation/SubmittingPatches file in the Git source code.

First, you need to set up the `imap` section in your `~/.gitconfig` file. You can set each value separately with a series of `git config` commands, or you can add them manually, but in the end your config file should look something like this:

```
[imap]
folder = "[Gmail]/Drafts"
host = imaps://imap.gmail.com
user = user@gmail.com
pass = p4ssw0rd
port = 993
sslverify = false
```

If your IMAP server doesn't use SSL, the last two lines probably aren't necessary, and the host value will be `imap://` instead of `imaps://`. When that is set up, you can use `git send-email` to place the patch series in the Drafts folder of the specified IMAP server:

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
```

```
Who should the emails be sent to? jessica@example.com  
Message-ID to be used as In-Reply-To for the first email? y
```

Then, Git spits out a bunch of log information looking something like this for each patch you're sending:

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from  
      \line 'From: Jessica Smith <jessica@example.com>'  
OK. Log says:  
Sendmail: /usr/sbin/sendmail -i jessica@example.com  
From: Jessica Smith <jessica@example.com>  
To: jessica@example.com  
Subject: [PATCH 1/2] added limit to log function  
Date: Sat, 30 May 2009 13:29:15 -0700  
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>  
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty  
In-Reply-To: <y>  
References: <y>
```

Result: OK

At this point, you should be able to go to your Drafts folder, change the To field to the mailing list you're sending the patch to, possibly CC the maintainer or person responsible for that section, and send it off.

Summary

This section has covered a number of common workflows for dealing with several very different types of Git projects you're likely to encounter, and introduced a couple of new tools to help you manage this process. Next, you'll see how to work the other side of the coin: maintaining a Git project. You'll learn how to be a benevolent dictator or integration manager.

Maintaining a Project

In addition to knowing how to effectively contribute to a project, you'll likely need to know how to maintain one. This can consist of accepting and applying patches generated via `format-patch` and e-mailed to you, or integrating changes in remote branches for repositories you've added as remotes to your project. Whether you maintain a canonical repository or want to help by verifying or approving patches, you need to know how to accept work in a way that is clearest for other contributors and sustainable by you over the long run.

Working in Topic Branches

When you’re thinking of integrating new work, it’s generally a good idea to try it out in a topic branch – a temporary branch specifically made to try out that new work. This way, it’s easy to tweak a patch individually and leave it if it’s not working until you have time to come back to it. If you create a simple branch name based on the theme of the work you’re going to try, such as `ruby_client` or something similarly descriptive, you can easily remember it if you have to abandon it for a while and come back later. The maintainer of the Git project tends to namespace these branches as well – such as `sc/ruby_client`, where `sc` is short for the person who contributed the work. As you’ll remember, you can create the branch based off your master branch like this:

```
$ git branch sc/ruby_client master
```

Or, if you want to also switch to it immediately, you can use the `checkout -b` option:

```
$ git checkout -b sc/ruby_client master
```

Now you’re ready to add your contributed work into this topic branch and determine if you want to merge it into your longer-term branches.

Applying Patches from E-mail

If you receive a patch over e-mail that you need to integrate into your project, you need to apply the patch in your topic branch to evaluate it. There are two ways to apply an e-mailed patch: with `git apply` or with `git am`.

APPLYING A PATCH WITH APPLY

If you received the patch from someone who generated it with the `git diff` or a Unix `diff` command (which is not recommended; see the next section), you can apply it with the `git apply` command. Assuming you saved the patch at `/tmp/patch-ruby-client.patch`, you can apply the patch like this:

```
$ git apply /tmp/patch-ruby-client.patch
```

This modifies the files in your working directory. It's almost identical to running a `patch -p1` command to apply the patch, although it's more paranoid and accepts fewer fuzzy matches than `patch`. It also handles file adds, deletes, and renames if they're described in the `git diff` format, which `patch` won't do. Finally, `git apply` is an "apply all or abort all" model where either everything is applied or nothing is, whereas `patch` can partially apply patchfiles, leaving your working directory in a weird state. `git apply` is overall much more conservative than `patch`. It won't create a commit for you – after running it, you must stage and commit the changes introduced manually.

You can also use `git apply` to see if a patch applies cleanly before you try actually applying it – you can run `git apply --check` with the patch:

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

If there is no output, then the patch should apply cleanly. This command also exits with a non-zero status if the check fails, so you can use it in scripts if you want.

APPLYING A PATCH WITH AM

If the contributor is a Git user and was good enough to use the `format-patch` command to generate their patch, then your job is easier because the patch contains author information and a commit message for you. If you can, encourage your contributors to use `format-patch` instead of `diff` to generate patches for you. You should only have to use `git apply` for legacy patches and things like that.

To apply a patch generated by `format-patch`, you use `git am`. Technically, `git am` is built to read an `mbox` file, which is a simple, plain-text format for storing one or more e-mail messages in one text file. It looks something like this:

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20
```

This is the beginning of the output of the `format-patch` command that you saw in the previous section. This is also a valid mbox e-mail format. If someone has e-mailed you the patch properly using `git send-email`, and you download that into an mbox format, then you can point `git am` to that mbox file, and it will start applying all the patches it sees. If you run a mail client that can save several e-mails out in mbox format, you can save entire patch series into a file and then use `git am` to apply them one at a time.

However, if someone uploaded a patch file generated via `format-patch` to a ticketing system or something similar, you can save the file locally and then pass that file saved on your disk to `git am` to apply it:

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

You can see that it applied cleanly and automatically created the new commit for you. The author information is taken from the e-mail's `From` and `Date` headers, and the message of the commit is taken from the `Subject` and body (before the patch) of the e-mail. For example, if this patch was applied from the mbox example above, the commit generated would look something like this:

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author: Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit: Scott Chacon <scchacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

add limit to log function

Limit log functionality to the first 20
```

The `Commit` information indicates the person who applied the patch and the time it was applied. The `Author` information is the individual who originally created the patch and when it was originally created.

But it's possible that the patch won't apply cleanly. Perhaps your main branch has diverged too far from the branch the patch was built from, or the patch depends on another patch you haven't applied yet. In that case, the `git am` process will fail and ask you what you want to do:

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

```
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

This command puts conflict markers in any files it has issues with, much like a conflicted merge or rebase operation. You solve this issue much the same way – edit the file to resolve the conflict, stage the new file, and then run `git am --resolved` to continue to the next patch:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

If you want Git to try a bit more intelligently to resolve the conflict, you can pass a `-3` option to it, which makes Git attempt a three-way merge. This option isn't on by default because it doesn't work if the commit the patch says it was based on isn't in your repository. If you do have that commit – if the patch was based on a public commit – then the `-3` option is generally much smarter about applying a conflicting patch:

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

In this case, this patch had already been applied. Without the `-3` option, it looks like a conflict.

If you're applying a number of patches from an mbox, you can also run the `am` command in interactive mode, which stops at each patch it finds and asks if you want to apply it:

```
$ git am -3 -i mbox
Commit Body is:
-----
seeing if this helps the gem
```

```
-----  
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

This is nice if you have a number of patches saved, because you can view the patch first if you don't remember what it is, or not apply the patch if you've already done so.

When all the patches for your topic are applied and committed into your branch, you can choose whether and how to integrate them into a longer-running branch.

Checking Out Remote Branches

If your contribution came from a Git user who set up their own repository, pushed a number of changes into it, and then sent you the URL to the repository and the name of the remote branch the changes are in, you can add them as a remote and do merges locally.

For instance, if Jessica sends you an e-mail saying that she has a great new feature in the `ruby-client` branch of her repository, you can test it by adding the remote and checking out that branch locally:

```
$ git remote add jessica git://github.com/jessica/myproject.git  
$ git fetch jessica  
$ git checkout -b rubyclient jessica/ruby-client
```

If she e-mails you again later with another branch containing another great feature, you can fetch and check out because you already have the remote set-up.

This is most useful if you're working with a person consistently. If someone only has a single patch to contribute once in a while, then accepting it over e-mail may be less time consuming than requiring everyone to run their own server and having to continually add and remove remotes to get a few patches. You're also unlikely to want to have hundreds of remotes, each for someone who contributes only a patch or two. However, scripts and hosted services may make this easier – it depends largely on how you develop and how your contributors develop.

The other advantage of this approach is that you get the history of the commits as well. Although you may have legitimate merge issues, you know where in your history their work is based; a proper three-way merge is the default rather than having to supply a `-3` and hope the patch was generated off a public commit to which you have access.

If you aren't working with a person consistently but still want to pull from them in this way, you can provide the URL of the remote repository to the `git pull` command. This does a one-time pull and doesn't save the URL as a remote reference:

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
 * branch            HEAD      -> FETCH_HEAD
Merge made by recursive.
```

Determining What Is Introduced

Now you have a topic branch that contains contributed work. At this point, you can determine what you'd like to do with it. This section revisits a couple of commands so you can see how you can use them to review exactly what you'll be introducing if you merge this into your main branch.

It's often helpful to get a review of all the commits that are in this branch but that aren't in your master branch. You can exclude commits in the master branch by adding the `--not` option before the branch name. This does the same thing as the `master..contrib` format that we used earlier. For example, if your contributor sends you two patches and you create a branch called `contrib` and applied those patches there, you can run this:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

        seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

        updated the gemspec to hopefully work better
```

To see what changes each commit introduces, remember that you can pass the `-p` option to `git log` and it will append the diff introduced to each commit.

To see a full diff of what would happen if you were to merge this topic branch with another branch, you may have to use a weird trick to get the correct results. You may think to run this:

```
$ git diff master
```

This command gives you a diff, but it may be misleading. If your `master` branch has moved forward since you created the topic branch from it, then you'll get seemingly strange results. This happens because Git directly compares the snapshots of the last commit of the topic branch you're on and the snapshot of the last commit on the `master` branch. For example, if you've added a line in a file on the `master` branch, a direct comparison of the snapshots will look like the topic branch is going to remove that line.

If `master` is a direct ancestor of your topic branch, this isn't a problem; but if the two histories have diverged, the diff will look like you're adding all the new stuff in your topic branch and removing everything unique to the `master` branch.

What you really want to see are the changes added to the topic branch – the work you'll introduce if you merge this branch with `master`. You do that by having Git compare the last commit on your topic branch with the first common ancestor it has with the `master` branch.

Technically, you can do that by explicitly figuring out the common ancestor and then running your diff on it:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

However, that isn't convenient, so Git provides another shorthand for doing the same thing: the triple-dot syntax. In the context of the `diff` command, you can put three periods after another branch to do a `diff` between the last commit of the branch you're on and its common ancestor with another branch:

```
$ git diff master...contrib
```

This command shows you only the work your current topic branch has introduced since its common ancestor with `master`. That is a very useful syntax to remember.

Integrating Contributed Work

When all the work in your topic branch is ready to be integrated into a more mainline branch, the question is how to do it. Furthermore, what overall work-

flow do you want to use to maintain your project? You have a number of choices, so we'll cover a few of them.

MERGING WORKFLOWS

One simple workflow merges your work into your `master` branch. In this scenario, you have a `master` branch that contains basically stable code. When you have work in a topic branch that you've done or that someone has contributed and you've verified, you merge it into your `master` branch, delete the topic branch, and then continue the process. If we have a repository with work in two branches named `ruby_client` and `php_client` that looks like **Figure 5-20** and merge `ruby_client` first and then `php_client` next, then your history will end up looking like **Figure 5-21**.

FIGURE 5-20

History with several topic branches.

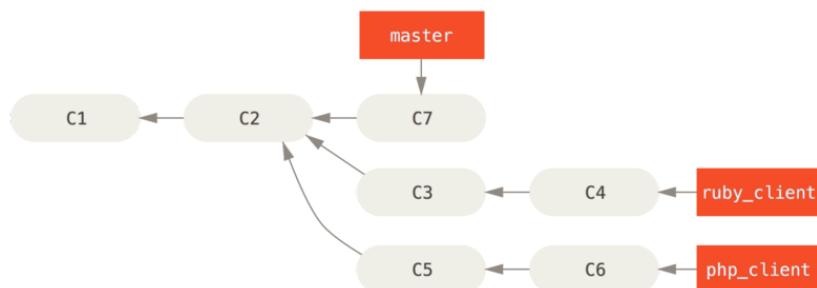
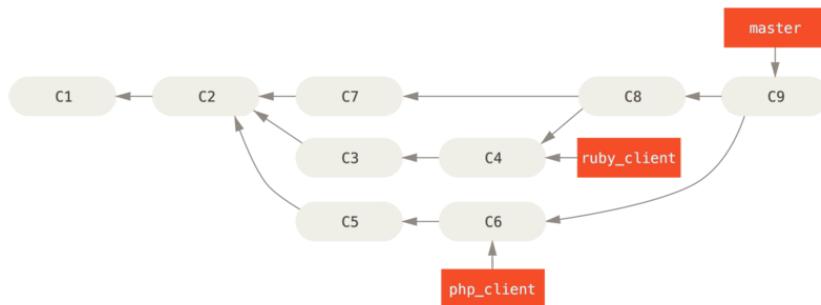


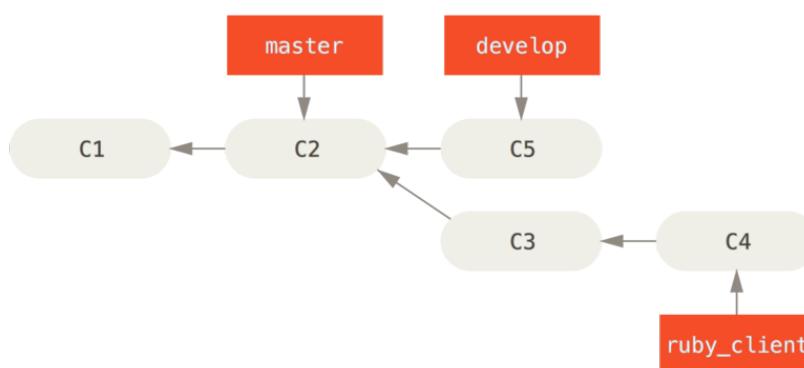
FIGURE 5-21

After a topic branch merge.

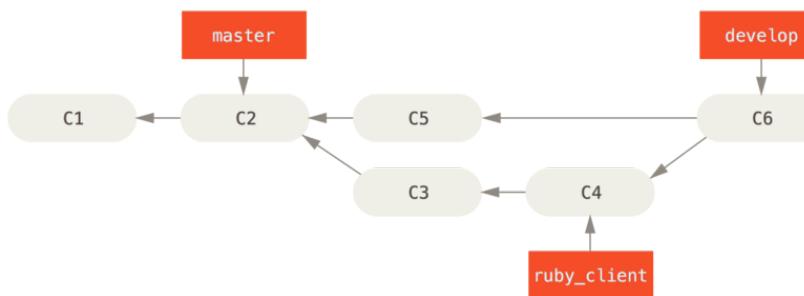


That is probably the simplest workflow, but it can possibly be problematic if you're dealing with larger or more stable projects where you want to be really careful about what you introduce.

If you have a more important project, you might want to use a two-phase merge cycle. In this scenario, you have two long-running branches, `master` and `develop`, in which you determine that `master` is updated only when a very stable release is cut and all new code is integrated into the `develop` branch. You regularly push both of these branches to the public repository. Each time you have a new topic branch to merge in ([Figure 5-22](#)), you merge it into `develop` ([Figure 5-23](#)); then, when you tag a release, you fast-forward `master` to wherever the now-stable `develop` branch is ([Figure 5-24](#)).

**FIGURE 5-22**

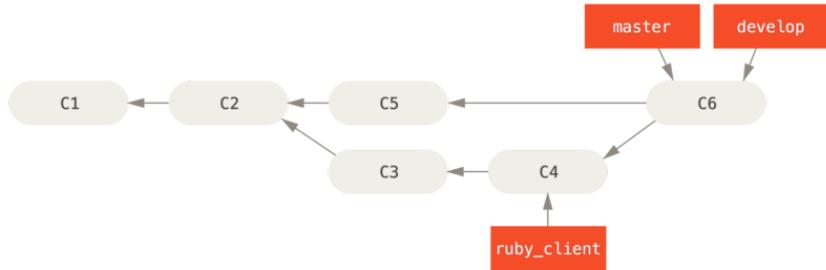
Before a topic branch merge.

**FIGURE 5-23**

After a topic branch merge.

FIGURE 5-24

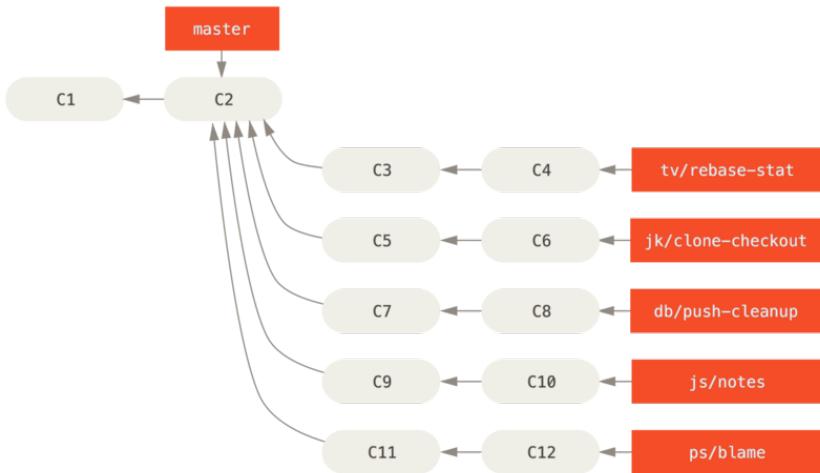
After a project release.



This way, when people clone your project’s repository, they can either check out master to build the latest stable version and keep up to date on that easily, or they can check out develop, which is the more cutting-edge stuff. You can also continue this concept, having an integrate branch where all the work is merged together. Then, when the codebase on that branch is stable and passes tests, you merge it into a develop branch; and when that has proven itself stable for a while, you fast-forward your master branch.

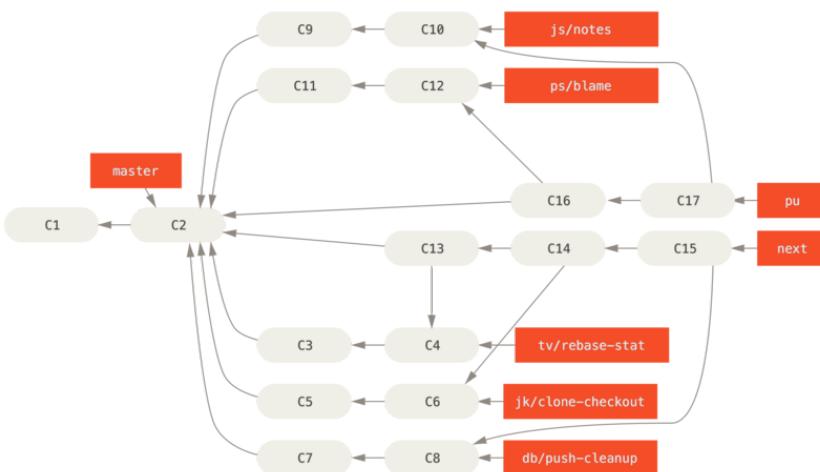
LARGE-MERGING WORKFLOWS

The Git project has four long-running branches: `master`, `next`, and `pu` (proposed updates) for new work, and `maint` for maintenance backports. When new work is introduced by contributors, it’s collected into topic branches in the maintainer’s repository in a manner similar to what we’ve described (see **Figure 5-25**). At this point, the topics are evaluated to determine whether they’re safe and ready for consumption or whether they need more work. If they’re safe, they’re merged into `next`, and that branch is pushed up so everyone can try the topics integrated together.

**FIGURE 5-25**

Managing a complex series of parallel contributed topic branches.

If the topics still need work, they're merged into pu instead. When it's determined that they're totally stable, the topics are re-merged into master and are then rebuilt from the topics that were in next but didn't yet graduate to master. This means master almost always moves forward, next is rebased occasionally, and pu is rebased even more often:

**FIGURE 5-26**

Merging contributed topic branches into long-term integration branches.

When a topic branch has finally been merged into `master`, it's removed from the repository. The Git project also has a `maint` branch that is forked off from the last release to provide backported patches in case a maintenance release is required. Thus, when you clone the Git repository, you have four branches that you can check out to evaluate the project in different stages of development, depending on how cutting edge you want to be or how you want to contribute; and the maintainer has a structured workflow to help them vet new contributions.

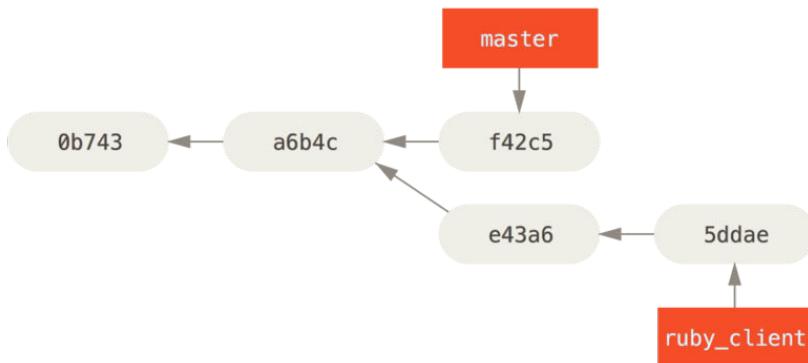
REBASING AND CHERRY PICKING WORKFLOWS

Other maintainers prefer to rebase or cherry-pick contributed work on top of their master branch, rather than merging it in, to keep a mostly linear history. When you have work in a topic branch and have determined that you want to integrate it, you move to that branch and run the `rebase` command to rebuild the changes on top of your current `master` (or `develop`, and so on) branch. If that works well, you can fast-forward your `master` branch, and you'll end up with a linear project history.

The other way to move introduced work from one branch to another is to cherry-pick it. A cherry-pick in Git is like a rebase for a single commit. It takes the patch that was introduced in a commit and tries to reapply it on the branch you're currently on. This is useful if you have a number of commits on a topic branch and you want to integrate only one of them, or if you only have one commit on a topic branch and you'd prefer to cherry-pick it rather than run rebase. For example, suppose you have a project that looks like this:

FIGURE 5-27

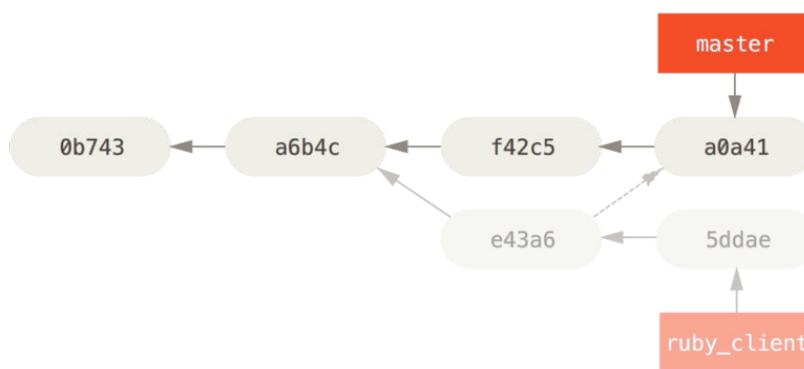
Example history before a cherry-pick.



If you want to pull commit `e43a6` into your `master` branch, you can run

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
  3 files changed, 17 insertions(+), 3 deletions(-)
```

This pulls the same change introduced in e43a6, but you get a new commit SHA-1 value, because the date applied is different. Now your history looks like this:

**FIGURE 5-28**

History after cherry-picking a commit on a topic branch.

Now you can remove your topic branch and drop the commits you didn't want to pull in.

RERERE

If you're doing lots of merging and rebasing, or you're maintaining a long-lived topic branch, Git has a feature called “rerere” that can help.

Rerere stands for “reuse recorded resolution” – it's a way of shortcircuiting manual conflict resolution. When rerere is enabled, Git will keep a set of pre- and post-images from successful merges, and if it notices that there's a conflict that looks exactly like one you've already fixed, it'll just use the fix from last time, without bothering you with it.

This feature comes in two parts: a configuration setting and a command. The configuration setting is `rerere.enabled`, and it's handy enough to put in your global config:

```
$ git config --global rerere.enabled true
```

Now, whenever you do a merge that resolves conflicts, the resolution will be recorded in the cache in case you need it in the future.

If you need to, you can interact with the rerere cache using the `git rerere` command. When it's invoked alone, Git checks its database of resolutions and tries to find a match with any current merge conflicts and resolve them (although this is done automatically if `rerere.enabled` is set to `true`). There are also subcommands to see what will be recorded, to erase specific resolution from the cache, and to clear the entire cache. We will cover rerere in more detail in “[Rerere](#)”.

Tagging Your Releases

When you've decided to cut a release, you'll probably want to drop a tag so you can re-create that release at any point going forward. You can create a new tag as discussed in [Chapter 2](#). If you decide to sign the tag as the maintainer, the tagging may look something like this:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'  
You need a passphrase to unlock the secret key for  
user: "Scott Chacon <schacon@gmail.com>"  
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

If you do sign your tags, you may have the problem of distributing the public PGP key used to sign your tags. The maintainer of the Git project has solved this issue by including their public key as a blob in the repository and then adding a tag that points directly to that content. To do this, you can figure out which key you want by running `gpg --list-keys`:

```
$ gpg --list-keys  
/Users/schacon/.gnupg/pubring.gpg  
-----  
pub 1024D/F721C45A 2009-02-09 [expires: 2010-02-09]  
uid Scott Chacon <schacon@gmail.com>  
sub 2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Then, you can directly import the key into the Git database by exporting it and piping that through `git hash-object`, which writes a new blob with those contents into Git and gives you back the SHA-1 of the blob:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Now that you have the contents of your key in Git, you can create a tag that points directly to it by specifying the new SHA-1 value that the `hash-object` command gave you:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

If you run `git push --tags`, the `maintainer-pgp-pub` tag will be shared with everyone. If anyone wants to verify a tag, they can directly import your PGP key by pulling the blob directly out of the database and importing it into GPG:

```
$ git show maintainer-pgp-pub | gpg --import
```

They can use that key to verify all your signed tags. Also, if you include instructions in the tag message, running `git show <tag>` will let you give the end user more specific instructions about tag verification.

Generating a Build Number

Because Git doesn't have monotonically increasing numbers like `v123` or the equivalent to go with each commit, if you want to have a human-readable name to go with a commit, you can run `git describe` on that commit. Git gives you the name of the nearest tag with the number of commits on top of that tag and a partial SHA-1 value of the commit you're describing:

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

This way, you can export a snapshot or build and name it something understandable to people. In fact, if you build Git from source code cloned from the Git repository, `git --version` gives you something that looks like this. If you're describing a commit that you have directly tagged, it gives you the tag name.

The `git describe` command favors annotated tags (tags created with the `-a` or `-s` flag), so release tags should be created this way if you're using `git`

`describe`, to ensure the commit is named properly when described. You can also use this string as the target of a `checkout` or `show` command, although it relies on the abbreviated SHA-1 value at the end, so it may not be valid forever. For instance, the Linux kernel recently jumped from 8 to 10 characters to ensure SHA-1 object uniqueness, so older `git describe` output names were invalidated.

Preparing a Release

Now you want to release a build. One of the things you'll want to do is create an archive of the latest snapshot of your code for those poor souls who don't use Git. The command to do this is `git archive`:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

If someone opens that tarball, they get the latest snapshot of your project under a `project` directory. You can also create a zip archive in much the same way, but by passing the `--format=zip` option to `git archive`:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

You now have a nice tarball and a zip archive of your project release that you can upload to your website or e-mail to people.

The Shortlog

It's time to e-mail your mailing list of people who want to know what's happening in your project. A nice way of quickly getting a sort of changelog of what has been added to your project since your last release or e-mail is to use the `git shortlog` command. It summarizes all the commits in the range you give it; for example, the following gives you a summary of all the commits since your last release, if your last release was named `v1.0.1`:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
    Add support for annotated tags to Grit::Tag
    Add packed-refs annotated tag support.
    Add Grit::Commit#to_patch
```

```
Update version and History.txt
Remove stray `puts`
Make ls_tree ignore nils

Tom Preston-Werner (4):
    fix dates in history
    dynamic version method
    Version bump to 1.0.2
    Regenerated gemspec for version 1.0.2
```

You get a clean summary of all the commits since v1.0.1, grouped by author, that you can e-mail to your list.

Заключение

Вы должны чувствовать себя достаточно свободно, внося свой вклад в проект под управлением Git, а также занимаясь поддержкой своего собственного проекта или интегрированием наработок других пользователей. Поздравляем вас, опытный Git-разработчик! В следующей главе вы узнаете о том, как использовать самый большой и самый популярный Git хостинг, GitHub.

GitHub 6

Гитхаб это крупнейшее хранилище Git репозиториев, а так же центр сотрудничества для миллионов разработчиков и проектов. Огромный процент репозиториев хранится на Гитхабе. Многие проекты с открытым исходным кодом используют его ради Git хостинга, багтрекера, рецензирования кода и других вещей. Так что, пока всё это не часть открытого Git проекта, наверняка вы захотите, или вам придётся взаимодействовать с Гитхабом при профессиональном использовании Git.

Эта глава про эффективное использование Гитхаба. Мы разберём регистрацию, управление учетной записью, создание и использование Git репозиториев, как вносить вклад в чужие проекты и как принимать чужой вклад в собственный проект, а так же программный интерфейс Гитхаба и ещё множество мелочей, который облегчат вам жизнь.

Если вас не интересует использование Гитхаба для размещения собственных проектов или сотрудничества с другими проектами, размещенными на нём, вы можете смело перейти к [Chapter 7](#).

ИЗМЕНЕНИЯ В ИНТЕРФЕЙСЕ

Важно отметить, что, как и на многих активных веб-сайтах, элементы интерфейса со скриншотов обязательно со временем изменятся. Надеемся, общее представление о том, что мы пытаемся сделать останется, но, если вы хотите более актуальных скриншотов, возможно вы найдёте их в онлайн версии этой книги.

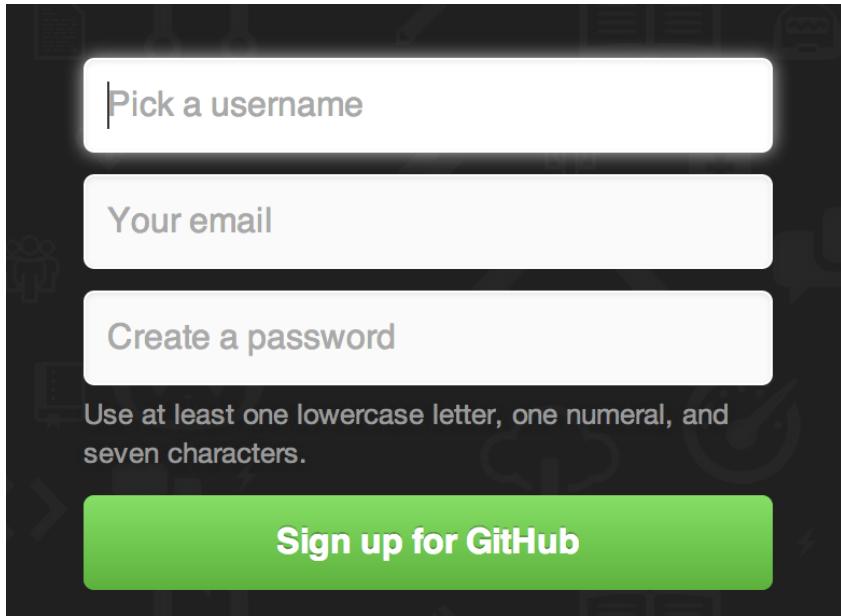
Настройка и конфигурация учетной записи

Первым делом нужно создать бесплатную учетную запись. Просто зайдите на <https://github.com>, выберите имя которое ещё не занято,

укажите адрес электронной почты и пароль, а затем нажмите большую зеленую кнопку “Sign up for GitHub”.

FIGURE 6-1

Форма регистрации на Github.



Далее вы попадете на страницу с тарифными планами, её пока можно проигнорировать. Github вышлет письмо для проверки вашего электронного адреса. Сделайте этот шаг, он достаточно важный (как мы увидим далее).

Github предоставляет весь функционал для бесплатных учетных записей, за исключением того, что все ваши репозитории полностью публичны (у любого есть доступ на чтение). Тарифные планы Github'a включают определенное количество частных проектов, но мы не будем их рассматривать в этой книге.

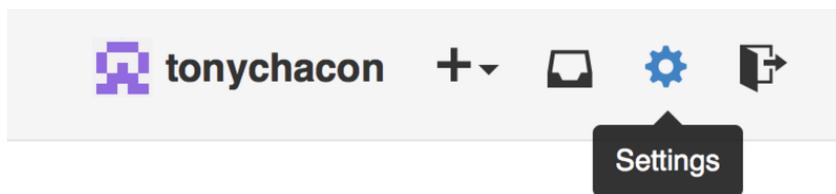
Кликнув по иконке октокота, вы попадете на страницу панели управления. Теперь Вы готовы пользоваться Github'ом.

Доступ по SSH

На данный момент вы можете подключаться к репозиториям Git используя протокол <https://> авторизуясь при помощи только что

созданного логина и пароля. Однако для того чтобы просто клонировать публично доступный проект, вам необязательно авторизироваться на сайте, но тем не менее, только что созданный аккаунт понадобится в то время, когда вы захотите загрузить (push) сделанные вами изменения.

Если же вы хотите использовать SSH доступ, в таком случае вам понадобится добавить публичный SSH ключ. (Если же у вас нет публичного SSH ключа, вы можете его сгенерировать “Генерация открытого SSH ключа”) Откройте настройки вашей учетной при помощи ссылки, расположенной в верхнем правом углу окна:

**FIGURE 6-2**

Ссылка “Настройка учетной записи” (“Account settings”).

Выберите секцию слева под названием “Ключи SSH”(“SSH keys”).

A screenshot of the GitHub "SSH keys" section. On the left, a sidebar menu shows "Profile", "Account settings", "Emails", "Notification center", "Billing", "SSH keys" (which is selected and highlighted in orange), "Security", "Applications", "Repositories", and "Organizations". The main content area has a heading "Need help? Check out our guide to generating SSH keys or troubleshoot common SSH Problems". Below this is a "SSH Keys" section with a sub-section "Add an SSH Key". It contains fields for "Title" (with a text input box) and "Key" (with a large text area). At the bottom is a green "Add key" button. Above the "Add key" button, there's a note: "There are no SSH keys with access to your account."

FIGURE 6-3

Ссылка (“SSH keys”).

Затем нажмите на кнопку "Добавить ключ SSH"("Add an SSH key"), задайте имя ключа а так же скопируйте и вставьте сам публичный ключ из `~/.ssh/id_rsa.pub` (ну или как бы у вас не назывался этот файл) в текстовое поле, затем нажмите "Добавить ключ"("Add key").

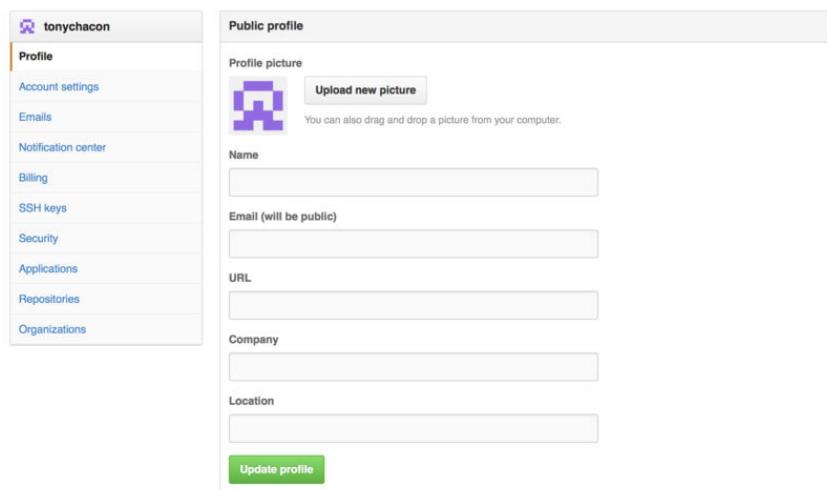
Задавайте такое имя SSH ключа, которое вы в состоянии запомнить.
Называйте каждый из добавляемых ключей по-разному (к примеру "Мой Ноутбук" или "Рабочая учётная запись"), для того чтобы в дальнейшем, при аннулировании ключа быть уверенным в правильности своего выбора.

Ваш аватар

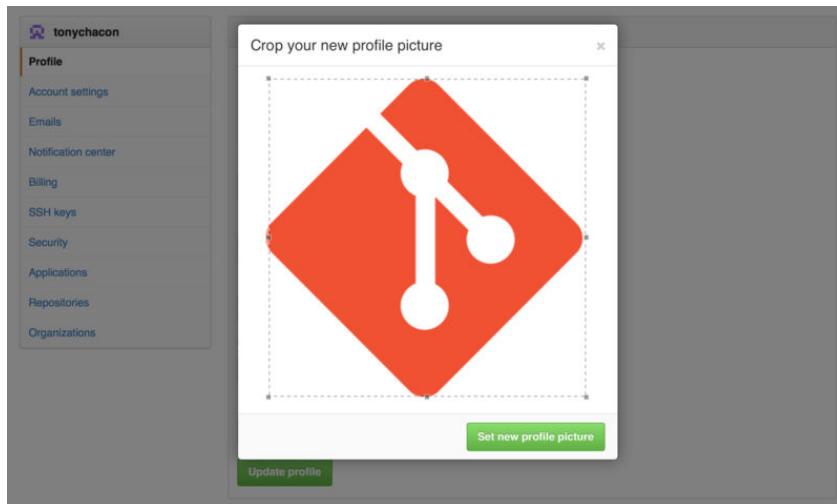
Следующий шаг, если хотите – замена аватара, который был сгенерирован для вас, на вами выбранный аватар. Пожалуйста зайдите во вкладку "Профиль"("Profile"), она расположена над вкладкой "Ключи SSH" и нажмите "Загрузить новую картинку"("Upload new picture").

FIGURE 6-4

Ссылка "Профиль" ("Profile").



Выберем логотип Git-а с жёсткого диска и отредактируем картинку под желаемый размер.

**FIGURE 6-5**

Редактирование аватара

После загрузки каждый сможет увидеть ваш аватар рядом с вашим именем пользователя.

Если вы используете такой популярный сервис как Gravatar (часто используется для учетных записей Wordpress), тот же самый аватар будет использован “по умолчанию”.

Ваши почтовые адреса

GitHub использует ваш почтовый адрес для привязки ваших Git коммитов к вашей учётной записи. Если вы используете несколько почтовых адресов в своих фиксациях (коммитах) и хотите, чтобы GitHub работал с ними корректно, то вам нужно будет добавить все используемые почтовые адреса в секцию под названием “Почтовые адреса” (“Emails”) расположенную на вкладке “Администрирование” (“Admin”).

FIGURE 6-6

Почтовые адреса

The screenshot shows the 'Email' section of the GitHub account settings for user 'tonychacon'. The left sidebar lists various account management options like Profile, Account settings, Emails, Notification center, Billing, SSH keys, Security, Applications, Repositories, and Organizations. The 'Emails' option is selected. The main content area is titled 'Email' and contains the following information:

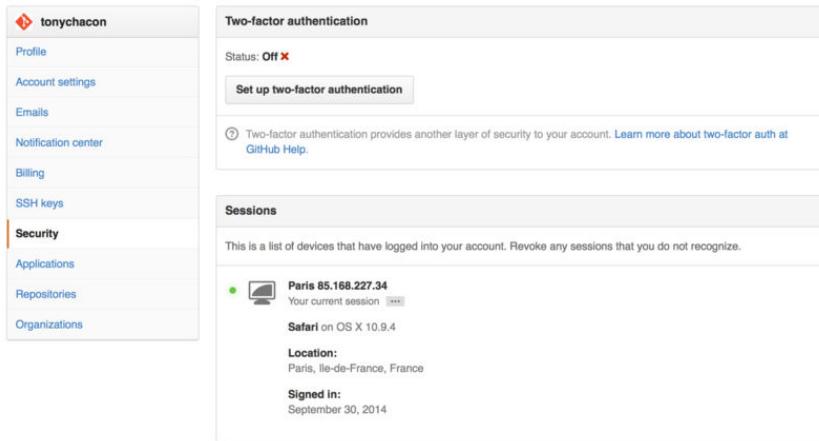
- Your primary GitHub email address will be used for account-related notifications (e.g. account changes and billing receipts) as well as any web-based GitHub operations (e.g. edits and merges).
- Three email addresses are listed:
 - tonychacon@example.com (Primary, Public)
 - tchacon@example.com (Unverified)
 - tony.chacon@example.com (Unverified)
- Buttons: 'Set as primary', 'Send verification email', and 'Add'.
- A note at the bottom: 'Keep my email address private' (unchecked) and 'We will use tonychacon@users.noreply.github.com when performing Git operations and sending email on your behalf.'

Как можно видеть в данном разделе Figure 6-6, у почтовых адресов имеются несколько состояний. Верхний почтовый адрес является основным и используется для верификации пользователя, это именно тот самый адрес, куда будут направляться оповещения, а также остальные уведомления. Второй адрес тоже верифицируемый, и также может быть использован в качестве основного, так же есть возможность поменять его местами с первым адресом. Последний адрес не является проверяемым, это значит, что вы не можете использовать его в качестве основного и получать на него уведомления. При фиксации (коммите) в любой из репозиториев, GitHub будет распознавать один из указанных почтовых адресов и автоматически привяжет этот самую фиксацию (коммит) к вашей учетной записи.

Двухфакторная аутентификация

В качестве дополнительной меры безопасности, вы можете настроить “Двухфакторную аутентификацию” (“Two-factor Authentication” или “2FA”). Двухфакторная аутентификация является механизмом, который становится все более и более популярным в качестве метода по снижению рисков компрометации ваших учетных записей в ситуациях когда пароль от вашей учетной записи, по тем или иным причинам, стал известен злоумышленникам. Активация этого механизма заставит GitHub запрашивать у вас оба метода при авторизации, то есть в ситуациях, когда пароль был скомпрометирован, злоумышленник все равно не сможет получить доступ к вашей учетной записи.

Вы сможете найти настройку “Двухфакторной аутентификации” (“Two-factor Authentication”) в секции “Безопасность” (“Security”) вкладки “Настройка учетной записи” (“Account settings”).

**FIGURE 6-7**

“Двухфакторная аутентификация” (“Two-factor Authentication”)

При нажатии на кнопку “Настроить двухфакторную аутентификацию” (“Set up two-factor authentication”) вы будете перенаправлены на страницу, где вам нужно будет настроить использование мобильного приложения для генерации вторичного кода проверки (так называемый “одноразовый пароль основанный на времени”), так же можно настроить GitHub таким образом, чтобы он отправлял вам СМС с кодом в момент, когда вам нужно авторизоваться на сайте.

После того, как вы выберете предпочтаемый вами метод и последуете предлагаемым инструкциям, ваша учетная запись будет в большей безопасности, и вам будет предоставленся дополнительный код во время вашей авторизации на сайте.

Внесение собственного вклада в проекты

Теперь наша учетная запись создана и настроена, давайте же пройдемся по деталям, которые будут полезны при внесении вклада в уже существующие проекты.

Создание ответвлений (fork)

Если вы хотите вносить свой вклад в уже существующие проекты, в которых у нас нет прав на внесения изменений путем отправки (push) изменений, вы можете создать свое собственное ответвление (“fork”) проекта. Это означает, что GitHub создаст вашу собственную копию проекта, данная копия будет находиться в вашем пространстве имен и вы сможете легко делать изменения путем отправки (push) изменений.

Исторически так сложилось, что англоязычный термин “fork” (создание ветвления проекта) имел негативный контекстный смысл, данный термин означал, что кто-то повел или ведет проект с открытым исходным кодом в другом, отличном от оригинала, направлении, иногда данный термин так же означал создание конкурирующего проекта с раздельными авторами. В контексте GitHub, “fork” (создание ветвления проекта) просто означает создание ветвления проекта в собственном пространстве имен, что позволяет вносить публичные изменения и делать свой собственный вклад в более открытом виде.

Таким образом, проекты не беспокоят пользователи, которые хотели бы выступать в роли соавторов, имели право на внесение изменений путем их отправки (push). Люди просто могут создавать свои собственные ветвления (fork), вносить туда изменения, а затем отправлять свои внесенные изменения в оригинальный репозиторий проекта путем создания запроса на принятие изменений (Pull Request), сами же запросы на принятие изменений (Pull Request) будут описаны далее. Запрос на принятие изменений (Pull Request) откроет новую ветвь с обсуждением отправляемого кода, и автор оригинального проекта, а также другие его участники, могут принимать участие в обсуждении предлагаемых изменений до тех пор, пока автор проекта не будет ими доволен, после чего автор проекта может добавить предлагаемые изменения в проект.

Для того, чтобы создать ответвление проекта (fork), зайдите на страницу проекта и нажмите кнопку “Создать ответвление” (“Fork”), которая расположена в правом верхнем углу.

FIGURE 6-8

Кнопка “Создать ответвление” (“Fork”).



Через несколько секунд вы будете перенаправлены на собственную новую проектную страницу, содержащую вашу копию, в которой у вас есть права на запись.

The GitHub Flow

GitHub is designed around a particular collaboration workflow, centered on Pull Requests. This flow works whether you're collaborating with a tightly-knit team in a single shared repository, or a globally-distributed company or network of strangers contributing to a project through dozens of forks. It is centered on the “**Topic Branches**” workflow covered in **Chapter 3**.

Here's how it generally works:

1. Create a topic branch from master.
2. Make some commits to improve the project.
3. Push this branch to your GitHub project.
4. Open a Pull Request on GitHub.
5. Discuss, and optionally continue committing.
6. The project owner merges or closes the Pull Request.

This is basically the Integration Manager workflow covered in “**Integration-Manager Workflow**”, but instead of using email to communicate and review changes, teams use GitHub's web based tools.

Let's walk through an example of proposing a change to an open source project hosted on GitHub using this flow.

CREATING A PULL REQUEST

Tony is looking for code to run on his Arduino programmable microcontroller and has found a great program file on GitHub at <https://github.com/schacon/blink>.

FIGURE 6-9

The project we want to contribute to.

```

schacon / blink
branch: master
commit: 12 my arduino blinking code (from arduino.cc)
1 contributor
25 lines (20 sloc) 0.71 kb
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.

  // Pin 13 has an LED connected on most Arduino boards.
  // give it a name:
  int led = 13;

  // the setup routine runs once when you press reset:
  void setup() {
    // initialize the digital pin as an output:
    pinMode(led, OUTPUT);
  }

  // the loop routine runs over and over again forever:
  void loop() {
    digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
    delay(1000);           // wait for a second
    digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
    delay(1000);           // wait for a second
  }
}

```

The only problem is that the blinking rate is too fast, we think it's much nicer to wait 3 seconds instead of 1 in between each state change. So let's improve the program and submit it back to the project as a proposed change.

First, we click the *Fork* button as mentioned earlier to get our own copy of the project. Our user name here is “tonychacon” so our copy of this project is at <https://github.com/tonychacon/blink> and that's where we can edit it. We will clone it locally, create a topic branch, make the code change and finally push that change back up to GitHub.

```

$ git clone https://github.com/tonychacon/blink ①
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink ②
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino ③

$ git diff --word-diff ④
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {

```

```
// the loop routine runs over and over again forever:
void loop() {
    digitalWrite(led, HIGH);      // turn the LED on (HIGH is the voltage level)
    [-delay(1000);-]{+delay(3000);+}          // wait for a second
    digitalWrite(led, LOW);       // turn the LED off by making the voltage LOW
    [-delay(1000);-]{+delay(3000);+}          // wait for a second
}

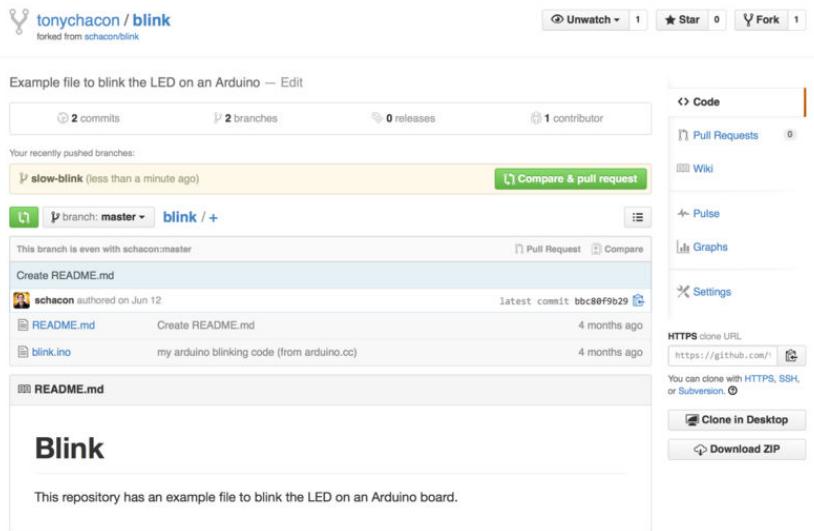
$ git commit -a -m 'three seconds is better' ⑤
[slow-blink 5ca509d] three seconds is better
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink ⑥
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink
```

- ➊ Clone our fork of the project locally
- ➋ Create a descriptive topic branch
- ➌ Make our change to the code
- ➍ Check that the change is good
- ➎ Commit our change to the topic branch
- ➏ Push our new topic branch back up to our GitHub fork

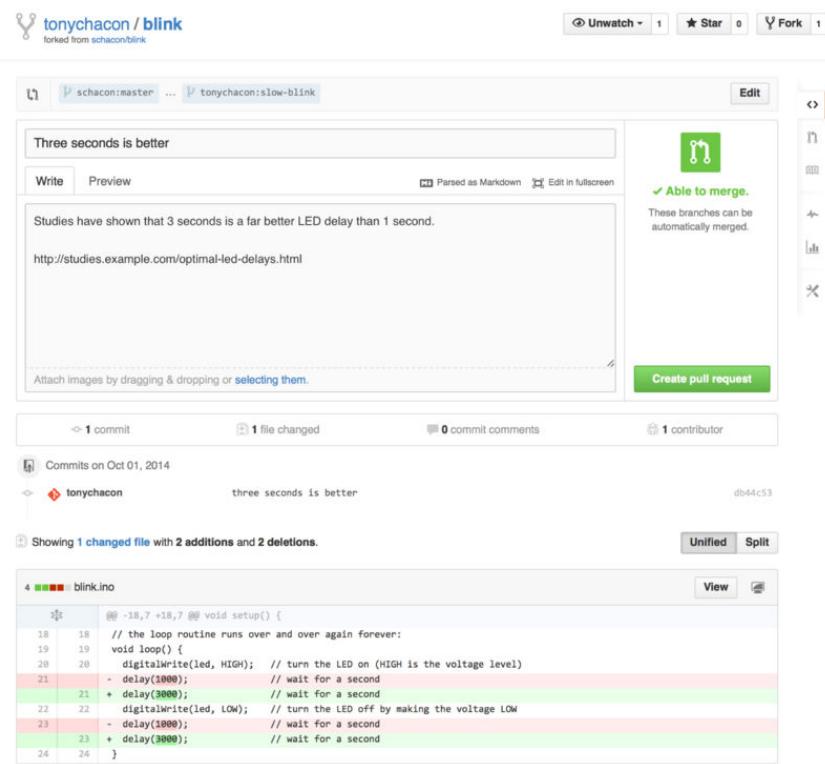
Now if we go back to our fork on GitHub, we can see that GitHub noticed that we pushed a new topic branch up and present us with a big green button to check out our changes and open a Pull Request to the original project.

You can alternatively go to the “Branches” page at <https://github.com/<user>/<project>/branches> to locate your branch and open a new Pull Request from there.

FIGURE 6-10**Pull Request button**

If we click that green button, we'll see a screen that allows us to create a title and description for the change we would like to request so the project owner has a good reason to consider it. It is generally a good idea to spend some effort making this description as useful as possible so the author knows why this is being suggested and why it would be a valuable change for them to accept.

We also see a list of the commits in our topic branch that are “ahead” of the `master` branch (in this case, just the one) and a unified diff of all the changes that will be made should this branch get merged by the project owner.

**FIGURE 6-11**

Pull Request creation page

When you hit the *Create pull request* button on this screen, the owner of the project you forked will get a notification that someone is suggesting a change and will link to a page that has all of this information on it.

Though Pull Requests are used commonly for public projects like this when the contributor has a complete change ready to be made, it's also often used in internal projects at the *beginning* of the development cycle. Since you can keep pushing to the topic branch even **after** the Pull Request is opened, it's often opened early and used as a way to iterate on work as a team within a context, rather than opened at the very end of the process.

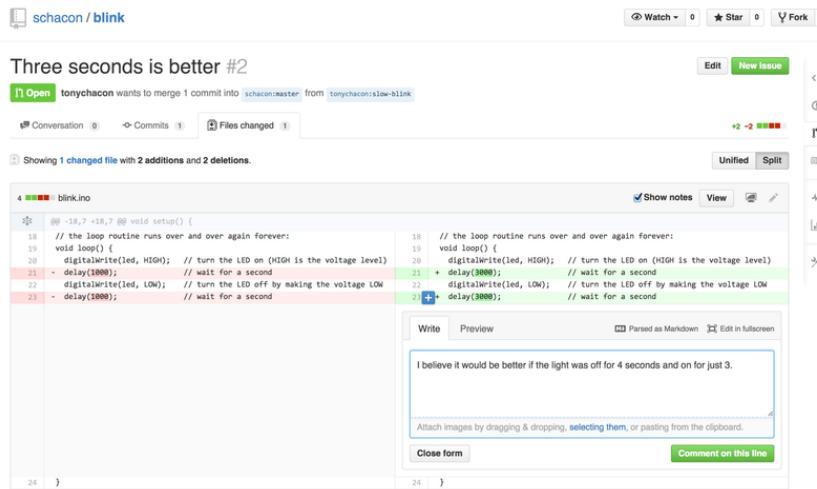
ITERATING ON A PULL REQUEST

At this point, the project owner can look at the suggested change and merge it, reject it or comment on it. Let's say that he likes the idea, but would prefer a slightly longer time for the light to be off than on.

Where this conversation may take place over email in the workflows presented in [Chapter 5](#), on GitHub this happens online. The project owner can review the unified diff and leave a comment by clicking on any of the lines.

FIGURE 6-12

Comment on a specific line of code in a Pull Request



Once the maintainer makes this comment, the person who opened the Pull Request (and indeed, anyone else watching the repository) will get a notification. We'll go over customizing this later, but if he had email notifications turned on, Tony would get an email like this:

FIGURE 6-13

Comments sent as email notifications



Anyone can also leave general comments on the Pull Request. In [Figure 6-14](#) we can see an example of the project owner both commenting on a line of code

and then leaving a general comment in the discussion section. You can see that the code comments are brought into the conversation as well.

The screenshot shows a GitHub pull request page for a file named 'blink.ino'. A comment from user 'tonychacon' is highlighted, suggesting a delay of 3 seconds instead of 1 second, with a link to a study. Another comment from 'schacon' notes that it would be better if the light was off for 4 seconds and on for just 3. The GitHub interface includes a sidebar with labels (None yet), milestones (No milestone), and assignees (No one—assign yourself). Notifications are also visible.

FIGURE 6-14

Pull Request discussion page

Now the contributor can see what they need to do in order to get their change accepted. Luckily this is also a very simple thing to do. Where over email you may have to re-roll your series and resubmit it to the mailing list, with GitHub you simply commit to the topic branch again and push.

If the contributor does that then the project owner will get notified again and when they visit the page they will see that it's been addressed. In fact, since a line of code changed that had a comment on it, GitHub notices that and collapses the outdated diff.

FIGURE 6-15

Pull Request final

Three seconds is better #2

tonychacon commented 11 minutes ago

Studies have shown that 3 seconds is a far better LED delay than 1 second.

<http://studies.example.com/optimal-led-delays.html>

three seconds is better db44c53

schacon commented on an outdated diff 5 minutes ago Show outdated diff

schacon commented 5 minutes ago

If you make that change, I'll be happy to merge this.

tonychacon added some commits 2 minutes ago

- longer off time 0c1f66f
- remove trailing whitespace ef4725c

tonychacon commented 10 seconds ago

I changed it to 4 seconds and also removed some trailing whitespace that I found. Anything else you would like me to do?

This pull request can be automatically merged. You can also merge branches on the command line. Merge pull request

An interesting thing to notice is that if you click on the “Files Changed” tab on this Pull Request, you’ll get the “unified” diff—that is, the total aggregate difference that would be introduced to your main branch if this topic branch was merged in. In git diff terms, it basically automatically shows you `git diff master...<branch>` for the branch this Pull Request is based on. See “[Determining What Is Introduced](#)” for more about this type of diff.

The other thing you’ll notice is that GitHub checks to see if the Pull Request merges cleanly and provides a button to do the merge for you on the server. This button only shows up if you have write access to the repository and a trivial merge is possible. If you click it GitHub will perform a “non-fast-forward” merge, meaning that even if the merge **could** be a fast-forward, it will still create a merge commit.

If you would prefer, you can simply pull the branch down and merge it locally. If you merge this branch into the `master` branch and push it to GitHub, the Pull Request will automatically be closed.

This is the basic workflow that most GitHub projects use. Topic branches are created, Pull Requests are opened on them, a discussion ensues, possibly more work is done on the branch and eventually the request is either closed or merged.

NOT ONLY FORKS

It's important to note that you can also open a Pull Request between two branches in the same repository. If you're working on a feature with someone and you both have write access to the project, you can push a topic branch to the repository and open a Pull Request on it to the `master` branch of that same project to initiate the code review and discussion process. No forking necessary.

Advanced Pull Requests

Now that we've covered the basics of contributing to a project on GitHub, let's cover a few interesting tips and tricks about Pull Requests so you can be more effective in using them.

PULL REQUESTS AS PATCHES

It's important to understand that many projects don't really think of Pull Requests as queues of perfect patches that should apply cleanly in order, as most mailing list-based projects think of patch series contributions. Most GitHub projects think about Pull Request branches as iterative conversations around a proposed change, culminating in a unified diff that is applied by merging.

This is an important distinction, because generally the change is suggested before the code is thought to be perfect, which is far more rare with mailing list based patch series contributions. This enables an earlier conversation with the maintainers so that arriving at the proper solution is more of a community effort. When code is proposed with a Pull Request and the maintainers or community suggest a change, the patch series is generally not re-rolled, but instead the difference is pushed as a new commit to the branch, moving the conversation forward with the context of the previous work intact.

For instance, if you go back and look again at [Figure 6-15](#), you'll notice that the contributor did not rebase his commit and send another Pull Request. Instead they added new commits and pushed them to the existing branch. This way if you go back and look at this Pull Request in the future, you can easily find all of the context of why decisions were made. Pushing the "Merge" button on

the site purposefully creates a merge commit that references the Pull Request so that it's easy to go back and research the original conversation if necessary.

KEEPING UP WITH UPSTREAM

If your Pull Request becomes out of date or otherwise doesn't merge cleanly, you will want to fix it so the maintainer can easily merge it. GitHub will test this for you and let you know at the bottom of every Pull Request if the merge is trivial or not.

FIGURE 6-16

Pull Request does not merge cleanly



If you see something like **Figure 6-16**, you'll want to fix your branch so that it turns green and the maintainer doesn't have to do extra work.

You have two main options in order to do this. You can either rebase your branch on top of whatever the target branch is (normally the master branch of the repository you forked), or you can merge the target branch into your branch.

Most developers on GitHub will choose to do the latter, for the same reasons we just went over in the previous section. What matters is the history and the final merge, so rebasing isn't getting you much other than a slightly cleaner history and in return is **far** more difficult and error prone.

If you want to merge in the target branch to make your Pull Request mergeable, you would add the original repository as a new remote, fetch from it, merge the main branch of that repository into your topic branch, fix any issues and finally push it back up to the same branch you opened the Pull Request on.

For example, let's say that in the "tonychacon" example we were using before, the original author made a change that would create a conflict in the Pull Request. Let's go through those steps.

```
$ git remote add upstream https://github.com/schacon/blink ①
```

```
$ git fetch upstream ②
```

```
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
```

```
* [new branch]      master      -> upstream/master

$ git merge upstream/master ③
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino ④
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
    into slower-blink

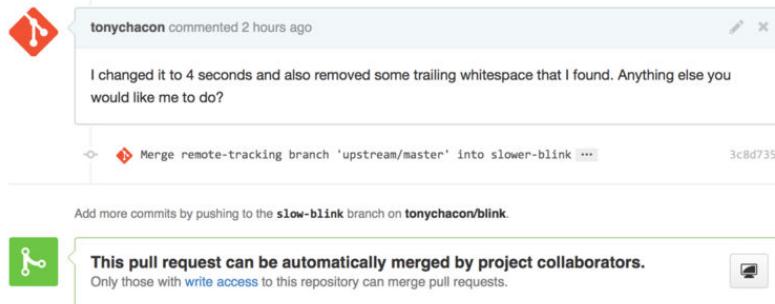
$ git push origin slow-blink ⑤
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
  ef4725c..3c8d735  slower-blink -> slow-blink
```

- ① Add the original repository as a remote named “upstream”
- ② Fetch the newest work from that remote
- ③ Merge the main branch into your topic branch
- ④ Fix the conflict that occurred
- ⑤ Push back up to the same topic branch

Once you do that, the Pull Request will be automatically updated and re-checked to see if it merges cleanly.

FIGURE 6-17

Pull Request now merges cleanly



One of the great things about Git is that you can do that continuously. If you have a very long-running project, you can easily merge from the target branch over and over again and only have to deal with conflicts that have arisen since the last time that you merged, making the process very manageable.

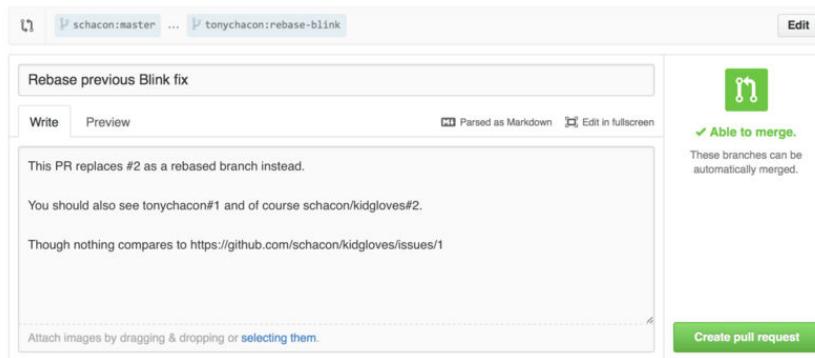
If you absolutely wish to rebase the branch to clean it up, you can certainly do so, but it is highly encouraged to not force push over the branch that the Pull Request is already opened on. If other people have pulled it down and done more work on it, you run into all of the issues outlined in “[The Perils of Rebasing](#)”. Instead, push the rebased branch to a new branch on GitHub and open a brand new Pull Request referencing the old one, then close the original.

REFERENCES

Your next question may be “How do I reference the old Pull Request?”. It turns out there are many, many ways to reference other things almost anywhere you can write in GitHub.

Let’s start with how to cross-reference another Pull Request or an Issue. All Pull Requests and Issues are assigned numbers and they are unique within the project. For example, you can’t have Pull Request #3 and Issue #3. If you want to reference any Pull Request or Issue from any other one, you can simply put #<num> in any comment or description. You can also be more specific if the Issue or Pull request lives somewhere else; write `username#<num>` if you’re referring to an Issue or Pull Request in a fork of the repository you’re in, or `username/repo#<num>` to reference something in another repository.

Let’s look at an example. Say we rebased the branch in the previous example, created a new pull request for it, and now we want to reference the old pull request from the new one. We also want to reference an issue in the fork of the repository and an issue in a completely different project. We can fill out the description just like [Figure 6-18](#).

**FIGURE 6-18**

Cross references in a Pull Request.

When we submit this pull request, we'll see all of that rendered like Figure 6-19.

The screenshot shows the GitHub pull request timeline for the same PR. It includes a header with 'Rebase previous Blink fix #4' and a 'Conversation' section with 0 comments, 2 commits, and 1 file changed. Below this, there is a comment from 'tonychacon' with the text: 'This PR replaces #2 as a rebased branch instead. You should also see tonychacon#1 and of course schacon/kidgloves#2. Though nothing compares to schacon/kidgloves#1'. At the bottom, there is a commit from 'tonychacon' with the message: 'added some commits 4 hours ago' and two commits listed: 'three seconds is better' (commit afe904a) and 'remove trailing whitespace' (commit a5a7751).

FIGURE 6-19

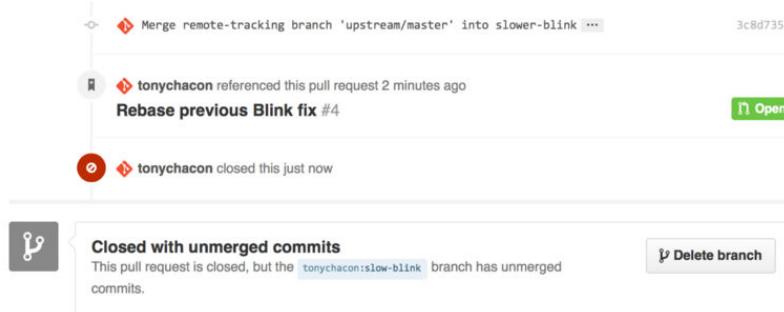
Cross references rendered in a Pull Request.

Notice that the full GitHub URL we put in there was shortened to just the information needed.

Now if Tony goes back and closes out the original Pull Request, we can see that by mentioning it in the new one, GitHub has automatically created a track-back event in the Pull Request timeline. This means that anyone who visits this Pull Request and sees that it is closed can easily link back to the one that superceded it. The link will look something like Figure 6-20.

FIGURE 6-20

Cross references rendered in a Pull Request.



In addition to issue numbers, you can also reference a specific commit by SHA-1. You have to specify a full 40 character SHA-1, but if GitHub sees that in a comment, it will link directly to the commit. Again, you can reference commits in forks or other repositories in the same way you did with issues.

Markdown

Linking to other Issues is just the beginning of interesting things you can do with almost any text box on GitHub. In Issue and Pull Request descriptions, comments, code comments and more, you can use what is called “GitHub Flavored Markdown”. Markdown is like writing in plain text but which is rendered richly.

See **Figure 6-21** for an example of how comments or text can be written and then rendered using Markdown.

FIGURE 6-21

An example of Markdown as written and as rendered.

The screenshot shows two side-by-side views of GitHub's Markdown preview. On the left, a "Markdown Example" box contains raw Markdown code. It includes a section about a problem with blink code, a list of bullet points, and a quote from Kanye West. On the right, the rendered version of this Markdown is shown as a comment from "tonychacon". The rendered text includes links, bolded text for "What is the problem?", and an image of the GitHub logo.

GITHUB FLAVORED MARKDOWN

The GitHub flavor of Markdown adds more things you can do beyond the basic Markdown syntax. These can all be really useful when creating useful Pull Request or Issue comments or descriptions.

Task Lists

The first really useful GitHub specific Markdown feature, especially for use in Pull Requests, is the Task List. A task list is a list of checkboxes of things you want to get done. Putting them into an Issue or Pull Request normally indicates things that you want to get done before you consider the item complete.

You can create a task list like this:

- [X] Write the code
- [] Write all the tests
- [] Document the code

If we include this in the description of our Pull Request or Issue, we'll see it rendered like **Figure 6-22**



FIGURE 6-22

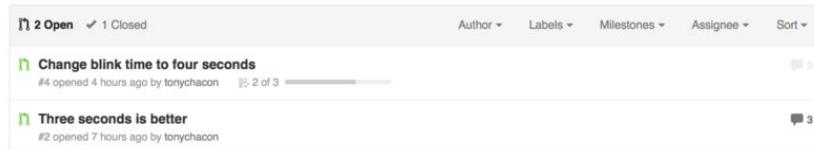
Task lists rendered in a Markdown comment.

This is often used in Pull Requests to indicate what all you would like to get done on the branch before the Pull Request will be ready to merge. The really cool part is that you can simply click the checkboxes to update the comment — you don't have to edit the Markdown directly to check tasks off.

What's more, GitHub will look for task lists in your Issues and Pull Requests and show them as metadata on the pages that list them out. For example, if you have a Pull Request with tasks and you look at the overview page of all Pull Requests, you can see how far done it is. This helps people break down Pull Requests into subtasks and helps other people track the progress of the branch. You can see an example of this in **Figure 6-23**.

FIGURE 6-23

Task list summary in the Pull Request list.



These are incredibly useful when you open a Pull Request early and use it to track your progress through the implementation of the feature.

Code Snippets

You can also add code snippets to comments. This is especially useful if you want to present something that you *could* try to do before actually implementing it as a commit on your branch. This is also often used to add example code of what is not working or what this Pull Request could implement.

To add a snippet of code you have to “fence” it in backticks.

```
```java
for(int i=0 ; i < 5 ; i++)
{
 System.out.println("i is : " + i);
}
...```

```

If you add a language name like we did there with *java*, GitHub will also try to syntax highlight the snippet. In the case of the above example, it would end up rendering like **Figure 6-24**.

**FIGURE 6-24**

Rendered fenced code example.



### Quoting

If you’re responding to a small part of a long comment, you can selectively quote out of the other comment by preceding the lines with the > character. In fact, this is so common and so useful that there is a keyboard shortcut for it. If

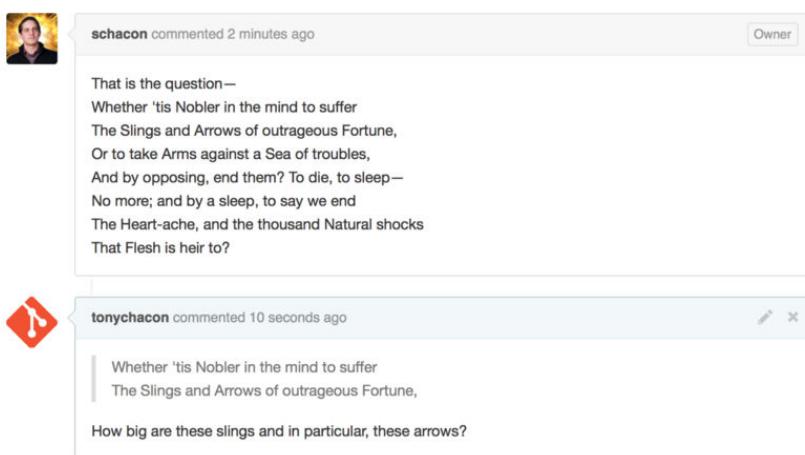
you highlight text in a comment that you want to directly reply to and hit the **r** key, it will quote that text in the comment box for you.

The quotes look something like this:

- > Whether 'tis Nobler in the mind to suffer
- > The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?

Once rendered, the comment will look like **Figure 6-25**.



**FIGURE 6-25**

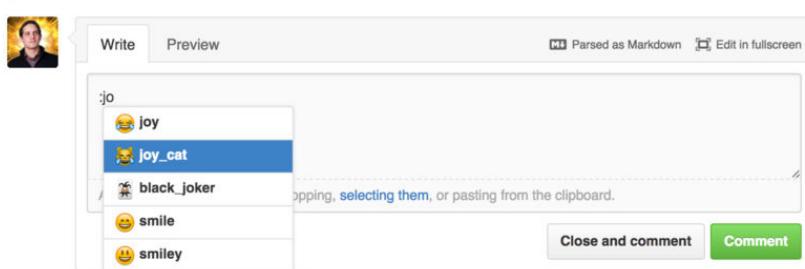
*Rendered quoting example.*

## Emoji

Finally, you can also use emoji in your comments. This is actually used quite extensively in comments you see on many GitHub Issues and Pull Requests. There is even an emoji helper in GitHub. If you are typing a comment and you start with a : character, an autocomplete will help you find what you're looking for.

**FIGURE 6-26**

*Emoji autocomplete in action.*



Emojis take the form of :<name>: anywhere in the comment. For instance, you could write something like this:

```
I :eyes: that :bug: and I :cold_sweat:.

:trophy: for :microscope: it.

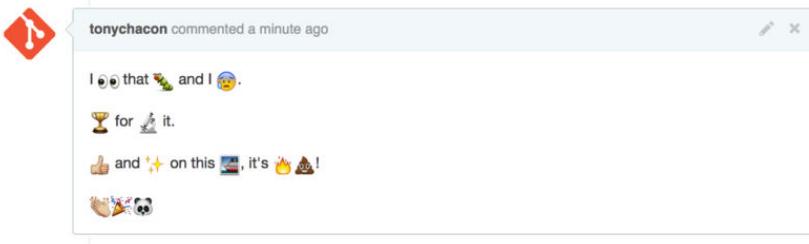
:+1: and :sparkles: on this :ship:, it's :fire::poop:!

:clap::tada::panda_face:
```

When rendered, it would look something like **Figure 6-27**.

**FIGURE 6-27**

*Heavy emoji commenting.*



Not that this is incredibly useful, but it does add an element of fun and emotion to a medium that is otherwise hard to convey emotion in.

---

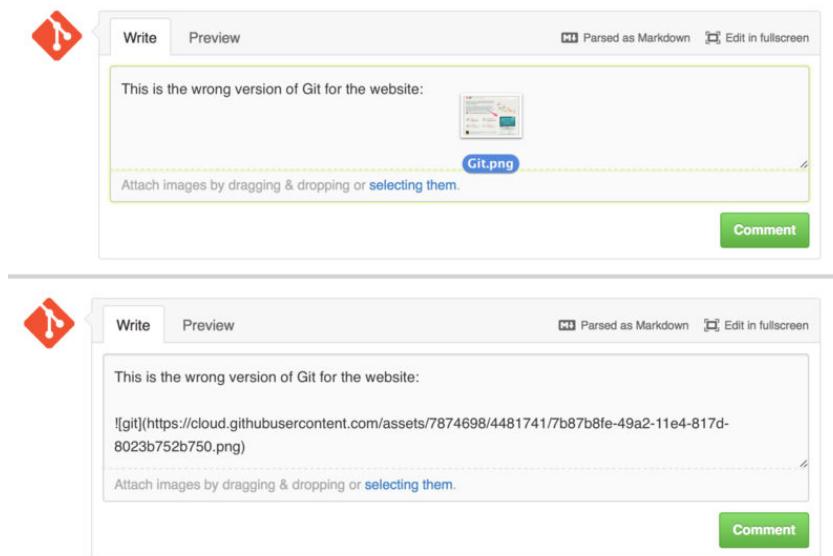
There are actually quite a number of web services that make use of emoji characters these days. A great cheat sheet to reference to find emoji that expresses what you want to say can be found at:

<http://www.emoji-cheat-sheet.com>

---

## Images

This isn't technically GitHub Flavored Markdown, but it is incredibly useful. In addition to adding Markdown image links to comments, which can be difficult to find and embed URLs for, GitHub allows you to drag and drop images into text areas to embed them.



**FIGURE 6-28**

Drag and drop images to upload them and auto-embed them.

If you look back at **Figure 6-18**, you can see a small “Parsed as Markdown” hint above the text area. Clicking on that will give you a full cheat sheet of everything you can do with Markdown on GitHub.

## Maintaining a Project

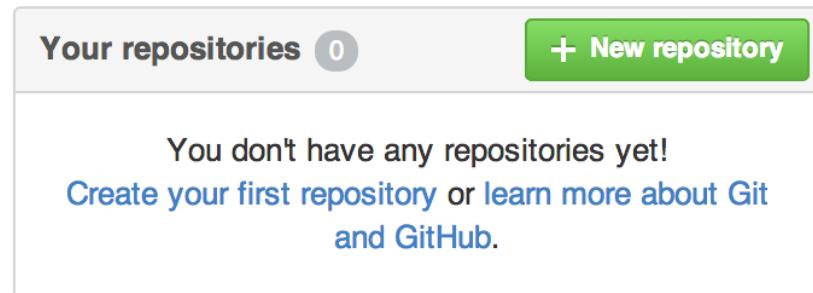
Now that we're comfortable contributing to a project, let's look at the other side: creating, maintaining and administering your own project.

### Creating a New Repository

Let's create a new repository to share our project code with. Start by clicking the “New repository” button on the right-hand side of the dashboard, or from the + button in the top toolbar next to your username as seen in **Figure 6-30**.

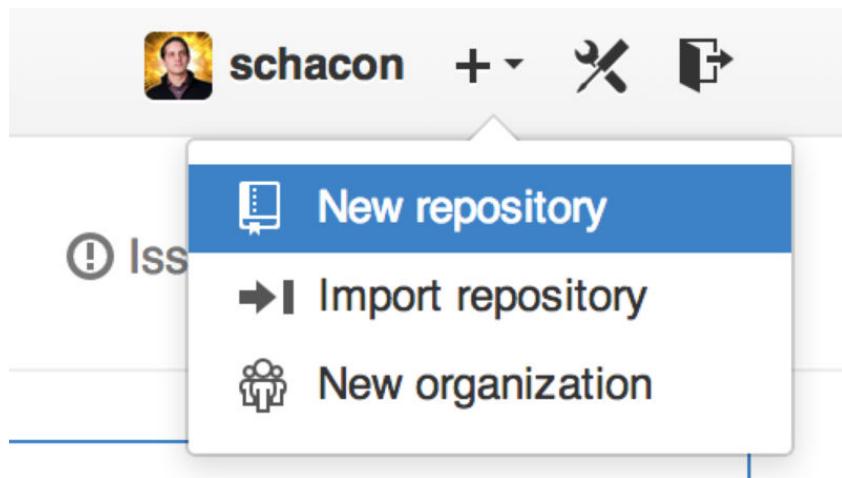
**FIGURE 6-29**

The “Your repositories” area.



**FIGURE 6-30**

The “New repository” dropdown.



This takes you to the “new repository” form:

The screenshot shows the GitHub interface for creating a new repository. At the top, it says "Owner" with a dropdown showing "ben" and "Repository name" with "iOSApp" entered. Below that is a note: "Great repository names are short and memorable. Need inspiration? How about [drunken-dubstep](#)". The "Description (optional)" field contains "iOS project for our mobile group". Under "Visibility", "Public" is selected, with the note "Anyone can see this repository. You choose who can commit.". There's also an option for "Private". A checkbox for "Initialize this repository with a README" is checked, with the note "This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally." Below this are buttons for "Add .gitignore: None" and "Add a license: None". At the bottom is a large green "Create repository" button.

**FIGURE 6-31**

*The “new repository” form.*

All you really have to do here is provide a project name; the rest of the fields are completely optional. For now, just click the “Create Repository” button, and boom – you have a new repository on GitHub, named `<user>/<project_name>`.

Since you have no code there yet, GitHub will show you instructions for how create a brand-new Git repository, or connect an existing Git project. We won’t belabor this here; if you need a refresher, check out [Chapter 2](#).

Now that your project is hosted on GitHub, you can give the URL to anyone you want to share your project with. Every project on GitHub is accessible over HTTP as [https://github.com/<user>/<project\\_name>](https://github.com/<user>/<project_name>), and over SSH as `git@github.com:<user>/<project_name>`. Git can fetch from and push to both of these URLs, but they are access-controlled based on the credentials of the user connecting to them.

---

It is often preferable to share the HTTP based URL for a public project, since the user does not have to have a GitHub account to access it for cloning. Users will have to have an account and an uploaded SSH key to access your project if you give them the SSH URL. The HTTP one is also exactly the same URL they would paste into a browser to view the project there.

---

## Adding Collaborators

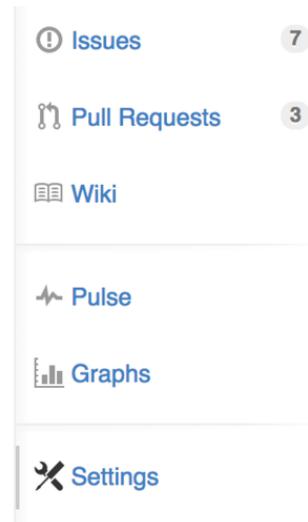
If you’re working with other people who you want to give commit access to, you need to add them as “collaborators”. If Ben, Jeff, and Louise all sign up for ac-

counts on GitHub, and you want to give them push access to your repository, you can add them to your project. Doing so will give them “push” access, which means they have both read and write access to the project and Git repository.

Click the “Settings” link at the bottom of the right-hand sidebar.

**FIGURE 6-32**

*The repository settings link.*



Then select “Collaborators” from the menu on the left-hand side. Then, just type a username into the box, and click “Add collaborator.” You can repeat this as many times as you like to grant access to everyone you like. If you need to revoke access, just click the “X” on the right-hand side of their row.

**FIGURE 6-33**

*Repository collaborators.*

 A screenshot of the GitHub 'Collaborators' page. On the left, there's a sidebar with 'Options' (highlighted), 'Collaborators' (selected), 'Webhooks & Services', and 'Deploy keys'. The main area shows a table of collaborators:
 

Collaborators		Full access to the repository
	<b>Ben Straub</b> ben	X
	<b>Jeff King</b> peff	X
	<b>Louise Corrigan</b> LouiseCorrigan	X

 Below the table is a search bar with 'Type a username' and a button 'Add collaborator'.

## Managing Pull Requests

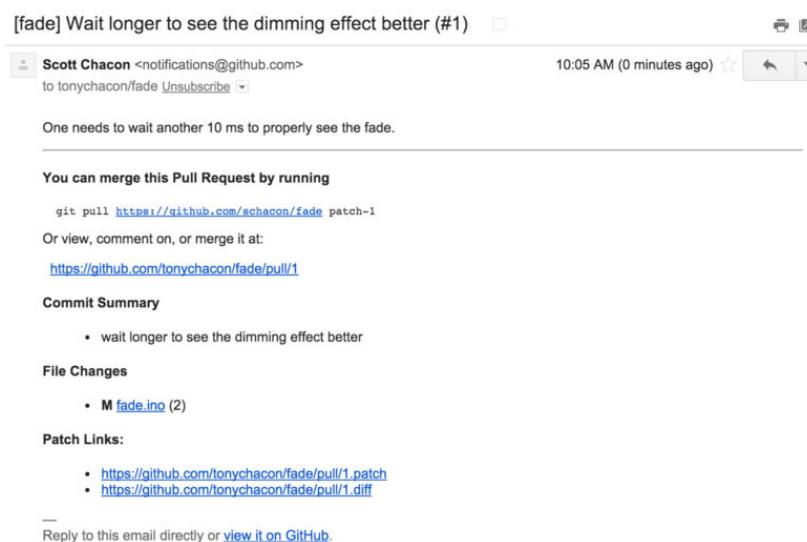
Now that you have a project with some code in it and maybe even a few collaborators who also have push access, let's go over what to do when you get a Pull Request yourself.

Pull Requests can either come from a branch in a fork of your repository or they can come from another branch in the same repository. The only difference is that the ones in a fork are often from people where you can't push to their branch and they can't push to yours, whereas with internal Pull Requests generally both parties can access the branch.

For these examples, let's assume you are "tonychacon" and you've created a new Arduino code project named "fade".

### EMAIL NOTIFICATIONS

Someone comes along and makes a change to your code and sends you a Pull Request. You should get an email notifying you about the new Pull Request and it should look something like **Figure 6-34**.



**FIGURE 6-34**

Email notification of a new Pull Request.

There are a few things to notice about this email. It will give you a small diffstat — a list of files that have changed in the Pull Request and by how much. It

gives you a link to the Pull Request on GitHub. It also gives you a few URLs that you can use from the command line.

If you notice the line that says `git pull <url> patch-1`, this is a simple way to merge in a remote branch without having to add a remote. We went over this quickly in “**Checking Out Remote Branches**”. If you wish, you can create and switch to a topic branch and then run this command to merge in the Pull Request changes.

The other interesting URLs are the `.diff` and `.patch` URLs, which as you may guess, provide unified diff and patch versions of the Pull Request. You could technically merge in the Pull Request work with something like this:

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

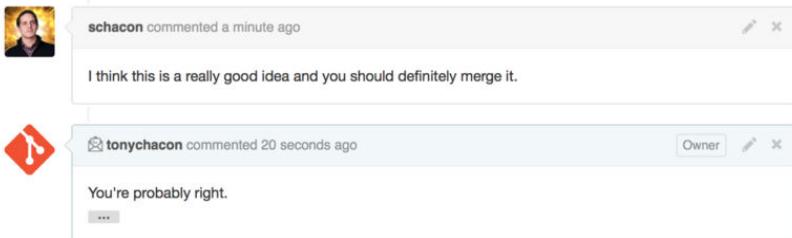
## COLLABORATING ON THE PULL REQUEST

As we covered in “**The GitHub Flow**”, you can now have a conversation with the person who opened the Pull Request. You can comment on specific lines of code, comment on whole commits or comment on the entire Pull Request itself, using GitHub Flavored Markdown everywhere.

Every time someone else comments on the Pull Request you will continue to get email notifications so you know there is activity happening. They will each have a link to the Pull Request where the activity is happening and you can also directly respond to the email to comment on the Pull Request thread.

**FIGURE 6-35**

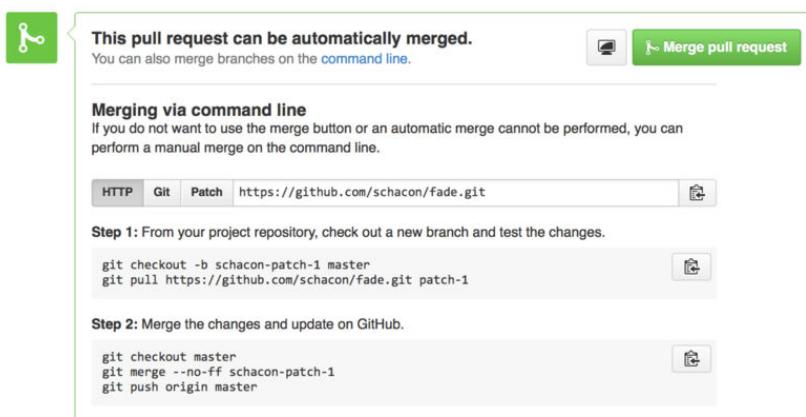
*Responses to emails are included in the thread.*



Once the code is in a place you like and want to merge it in, you can either pull the code down and merge it locally, either with the `git pull <url> <branch>` syntax we saw earlier, or by adding the fork as a remote and fetching and merging.

If the merge is trivial, you can also just hit the “Merge” button on the GitHub site. This will do a “non-fast-forward” merge, creating a merge commit even if a

fast-forward merge was possible. This means that no matter what, every time you hit the merge button, a merge commit is created. As you can see in **Figure 6-36**, GitHub gives you all of this information if you click the hint link.

**FIGURE 6-36**

*Merge button and instructions for merging a Pull Request manually.*

If you decide you don't want to merge it, you can also just close the Pull Request and the person who opened it will be notified.

## PULL REQUEST REFS

If you're dealing with a **lot** of Pull Requests and don't want to add a bunch of remotes or do one time pulls every time, there is a neat trick that GitHub allows you to do. This is a bit of an advanced trick and we'll go over the details of this a bit more in “Спецификации ссылок”, but it can be pretty useful.

GitHub actually advertises the Pull Request branches for a repository as sort of pseudo-branches on the server. By default you don't get them when you clone, but they are there in an obscured way and you can access them pretty easily.

To demonstrate this, we're going to use a low-level command (often referred to as a “plumbing” command, which we'll read about more in “Сантехника и Фарфор”) called `ls-remote`. This command is generally not used in day-to-day Git operations but it's useful to show us what references are present on the server.

If we run this command against the “blink” repository we were using earlier, we will get a list of all the branches and tags and other references in the repository.

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d HEAD
10d539600d86723087810ec636870a504f4fee4d refs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6e refs/pull/1/head
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3 refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfb2665adec1 refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a refs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166c refs/pull/4/merge
```

Of course, if you’re in your repository and you run `git ls-remote origin` or whatever remote you want to check, it will show you something similar to this.

If the repository is on GitHub and you have any Pull Requests that have been opened, you’ll get these references that are prefixed with `refs/pull/`. These are basically branches, but since they’re not under `refs/heads/` you don’t get them normally when you clone or fetch from the server — the process of fetching ignores them normally.

There are two references per Pull Request - the one that ends in `/head` points to exactly the same commit as the last commit in the Pull Request branch. So if someone opens a Pull Request in our repository and their branch is named `bug-fix` and it points to commit `a5a775`, then in **our** repository we will not have a `bug-fix` branch (since that’s in their fork), but we *will* have `pull/<pr#>/head` that points to `a5a775`. This means that we can pretty easily pull down every Pull Request branch in one go without having to add a bunch of remotes.

Now, you could do something like fetching the reference directly.

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch refs/pull/958/head -> FETCH_HEAD
```

This tells Git, “Connect to the `origin` remote, and download the ref named `refs/pull/958/head`.” Git happily obeys, and downloads everything you need to construct that ref, and puts a pointer to the commit you want under `.git/FETCH_HEAD`. You can follow that up with `git merge FETCH_HEAD` into a branch you want to test it in, but that merge commit message looks a bit weird. Also, if you’re reviewing a **lot** of pull requests, this gets tedious.

There’s also a way to fetch *all* of the pull requests, and keep them up to date whenever you connect to the remote. Open up `.git/config` in your favorite editor, and look for the `origin` remote. It should look a bit like this:

```
[remote "origin"]
 url = https://github.com/libgit2/libgit2
 fetch = +refs/heads/*:refs/remotes/origin/*
```

That line that begins with `fetch =` is a “refspec.” It’s a way of mapping names on the remote with names in your local `.git` directory. This particular one tells Git, “the things on the remote that are under `refs/heads` should go in my local repository under `refs/remotes/origin`.” You can modify this section to add another refspec:

```
[remote "origin"]
 url = https://github.com/libgit2/libgit2.git
 fetch = +refs/heads/*:refs/remotes/origin/*
 fetch = +refs/pull/*:refs/remotes/origin/pr/*
```

That last line tells Git, “All the refs that look like `refs/pull/123/head` should be stored locally like `refs/remotes/origin/pr/123`.” Now, if you save that file, and do a `git fetch`:

```
$ git fetch
...
* [new ref] refs/pull/1/head -> origin/pr/1
* [new ref] refs/pull/2/head -> origin/pr/2
* [new ref] refs/pull/4/head -> origin/pr/4
...
```

Now all of the remote pull requests are represented locally with refs that act much like tracking branches; they’re read-only, and they update when you do a fetch. This makes it super easy to try the code from a pull request locally:

```
$ git checkout pr/2
Checking out files: 100% (3769/3769), done.
Branch pr/2 set up to track remote branch pr/2 from origin.
Switched to a new branch 'pr/2'
```

The eagle-eyed among you would note the `head` on the end of the remote portion of the refspec. There’s also a `refs/pull/#/merge` ref on the GitHub side, which represents the commit that would result if you push the “merge” button on the site. This can allow you to test the merge before even hitting the button.

## PULL REQUESTS ON PULL REQUESTS

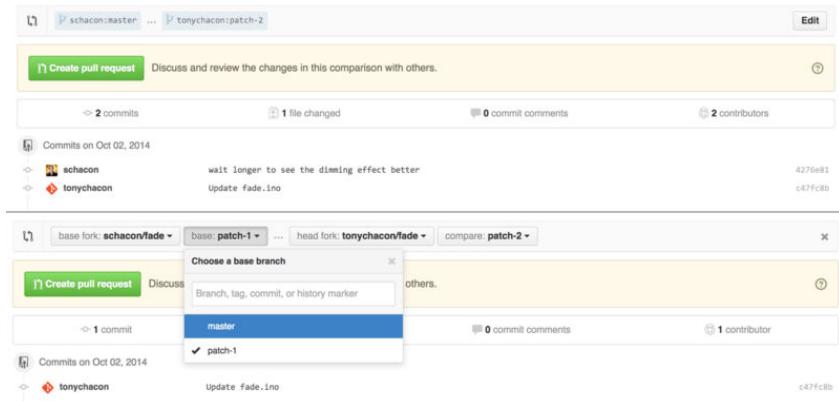
Not only can you open Pull Requests that target the main or master branch, you can actually open a Pull Request targeting any branch in the network. In fact, you can even target another Pull Request.

If you see a Pull Request that is moving in the right direction and you have an idea for a change that depends on it or you're not sure if it's a good idea, or you just don't have push access to the target branch, you can open a Pull Request directly to it.

When you go to open a Pull Request, there is a box at the top of the page that specifies which branch you're requesting to pull to and which you're requesting to pull from. If you hit the "Edit" button at the right of that box you can change not only the branches but also which fork.

**FIGURE 6-37**

*Manually change the Pull Request target fork and branch.*

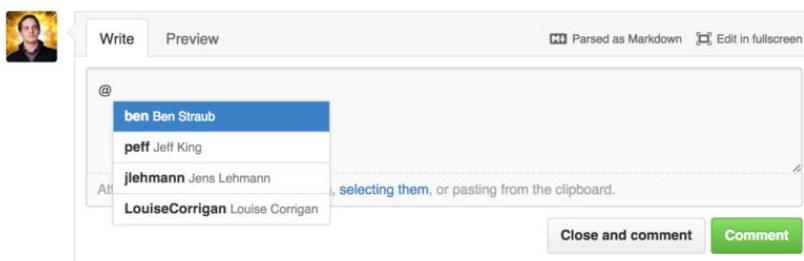


Here you can fairly easily specify to merge your new branch into another Pull Request or another fork of the project.

## Mentions and Notifications

GitHub also has a pretty nice notifications system built in that can come in handy when you have questions or need feedback from specific individuals or teams.

In any comment you can start typing a @ character and it will begin to autocomplete with the names and usernames of people who are collaborators or contributors in the project.

**FIGURE 6-38**

Start typing @ to mention someone.

You can also mention a user who is not in that dropdown, but often the autocomplete can make it faster.

Once you post a comment with a user mention, that user will be notified. This means that this can be a really effective way of pulling people into conversations rather than making them poll. Very often in Pull Requests on GitHub people will pull in other people on their teams or in their company to review an Issue or Pull Request.

If someone gets mentioned on a Pull Request or Issue, they will be “subscribed” to it and will continue getting notifications any time some activity occurs on it. You will also be subscribed to something if you opened it, if you’re watching the repository or if you comment on something. If you no longer wish to receive notifications, there is an “Unsubscribe” button on the page you can click to stop receiving updates on it.

**FIGURE 6-39**

*Unsubscribe from an Issue or Pull Request.*

# Notifications

## ►×

## Unsubscribe

You're receiving notifications because you commented.

### THE NOTIFICATIONS PAGE

When we mention “notifications” here with respect to GitHub, we mean a specific way that GitHub tries to get in touch with you when events happen and there are a few different ways you can configure them. If you go to the “Notification center” tab from the settings page, you can see some of the options you have.

**FIGURE 6-40**

*Notification center options.*

The screenshot shows the GitHub settings interface for the user 'tonychacon'. On the left, a sidebar lists various settings categories: Profile, Account settings, Emails, **Notification center** (which is currently selected), Billing, SSH keys, Security, Applications, Repositories, and Organizations. The main content area is titled 'How you receive notifications'.

**Participating:** Describes notifications for conversations and @mentions. It includes two checkboxes:  Email and  Web.

**Watching:** Describes notifications for repositories and threads. It includes two checkboxes:  Email and  Web.

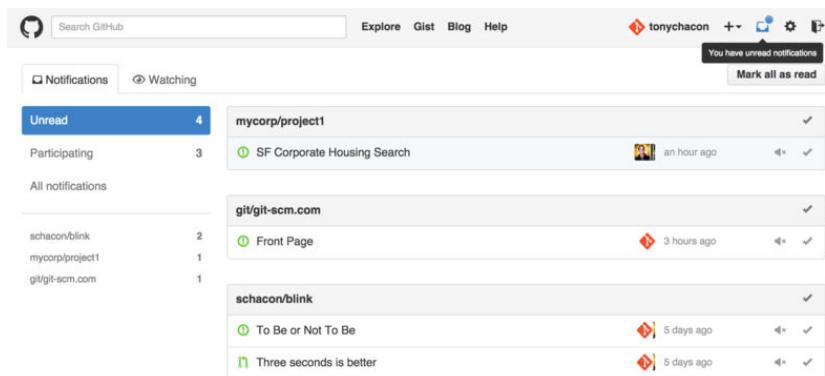
**Notification email:** A section for managing primary email addresses. It shows 'Primary email address' set to 'tchacon@example.com' and a 'Save' button.

**Custom routing:** A note stating that notifications can be sent to different verified email addresses depending on the repository owner.

The two choices are to get notifications over “Email” and over “Web” and you can choose either, neither or both for when you actively participate in things and for activity on repositories you are watching.

### Web Notifications

Web notifications only exist on GitHub and you can only check them on GitHub. If you have this option selected in your preferences and a notification is triggered for you, you will see a small blue dot over your notifications icon at the top of your screen as seen in **Figure 6-41**.



**FIGURE 6-41**

Notification center.

If you click on that, you will see a list of all the items you have been notified about, grouped by project. You can filter to the notifications of a specific project by clicking on its name in the left hand sidebar. You can also acknowledge the notification by clicking the checkmark icon next to any notification, or acknowledge *all* of the notifications in a project by clicking the checkmark at the top of the group. There is also a mute button next to each checkmark that you can click to not receive any further notifications on that item.

All of these tools are very useful for handling large numbers of notifications. Many GitHub power users will simply turn off email notifications entirely and manage all of their notifications through this screen.

### Email Notifications

Email notifications are the other way you can handle notifications through GitHub. If you have this turned on you will get emails for each notification. We saw examples of this in **Figure 6-13** and **Figure 6-34**. The emails will also be threaded properly, which is nice if you’re using a threading email client.

There is also a fair amount of metadata embedded in the headers of the emails that GitHub sends you, which can be really helpful for setting up custom filters and rules.

For instance, if we look at the actual email headers sent to Tony in the email shown in [Figure 6-34](#), we will see the following among the information sent:

```
To: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Wait longer to see the dimming effect better (#1)
X-GitHub-Recipient: tonychacon
List-ID: tonychacon/fade <fade.tonychacon.github.com>
List-Archive: https://github.com/tonychacon/fade
List-Post: <mailto:reply+i-4XXX@reply.github.com>
List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>, ...
X-GitHub-Recipient-Address: tchacon@example.com
```

There are a couple of interesting things here. If you want to highlight or re-route emails to this particular project or even Pull Request, the information in `Message-ID` gives you all the data in `<user>/<project>/<type>/<id>` format. If this were an issue, for example, the `<type>` field would have been “issues” rather than “pull”.

The `List-Post` and `List-Unsubscribe` fields mean that if you have a mail client that understands those, you can easily post to the list or “Unsubscribe” from the thread. That would be essentially the same as clicking the “mute” button on the web version of the notification or “Unsubscribe” on the Issue or Pull Request page itself.

It’s also worth noting that if you have both email and web notifications enabled and you read the email version of the notification, the web version will be marked as read as well if you have images allowed in your mail client.

## Special Files

There are a couple of special files that GitHub will notice if they are present in your repository.

### README

The first is the `README` file, which can be of nearly any format that GitHub recognizes as prose. For example, it could be `README`, `README.md`, `README.asciidoc`, etc. If GitHub sees a `README` file in your source, it will render it on the landing page of the project.

Many teams use this file to hold all the relevant project information for someone who might be new to the repository or project. This generally includes things like:

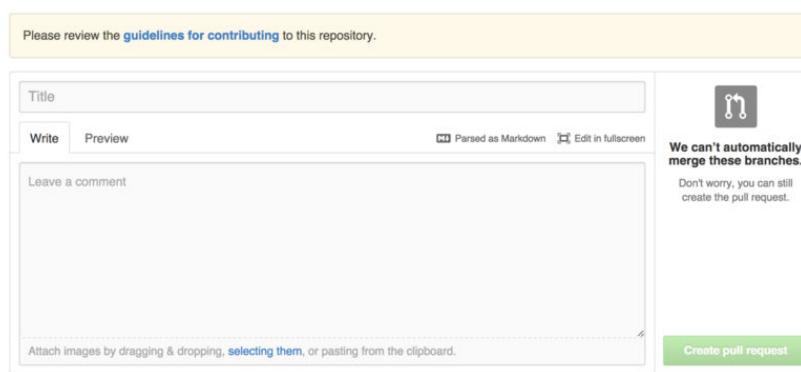
- What the project is for

- How to configure and install it
- An example of how to use it or get it running
- The license that the project is offered under
- How to contribute to it

Since GitHub will render this file, you can embed images or links in it for added ease of understanding.

## CONTRIBUTING

The other special file that GitHub recognizes is the CONTRIBUTING file. If you have a file named CONTRIBUTING with any file extension, GitHub will show **Figure 6-42** when anyone starts opening a Pull Request.



**FIGURE 6-42**

*Opening a Pull Request when a CONTRIBUTING file exists.*

The idea here is that you can specify specific things you want or don't want in a Pull Request sent to your project. This way people may actually read the guidelines before opening the Pull Request.

## Project Administration

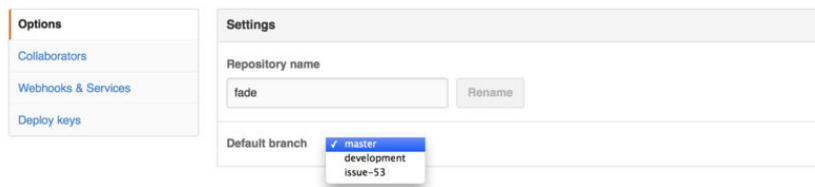
Generally there are not a lot of administrative things you can do with a single project, but there are a couple of items that might be of interest.

## CHANGING THE DEFAULT BRANCH

If you are using a branch other than “master” as your default branch that you want people to open Pull Requests on or see by default, you can change that in your repository’s settings page under the “Options” tab.

**FIGURE 6-43**

*Change the default branch for a project.*



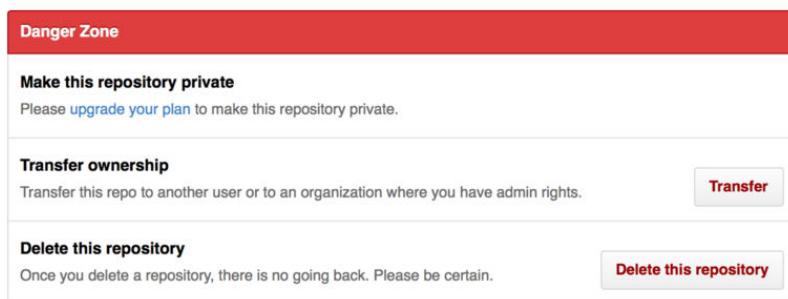
Simply change the default branch in the dropdown and that will be the default for all major operations from then on, including which branch is checked out by default when someone clones the repository.

## TRANSFERRING A PROJECT

If you would like to transfer a project to another user or an organization in GitHub, there is a “Transfer ownership” option at the bottom of the same “Options” tab of your repository settings page that allows you to do this.

**FIGURE 6-44**

*Transfer a project to another GitHub user or Organization.*



This is helpful if you are abandoning a project and someone wants to take it over, or if your project is getting bigger and want to move it into an organization.

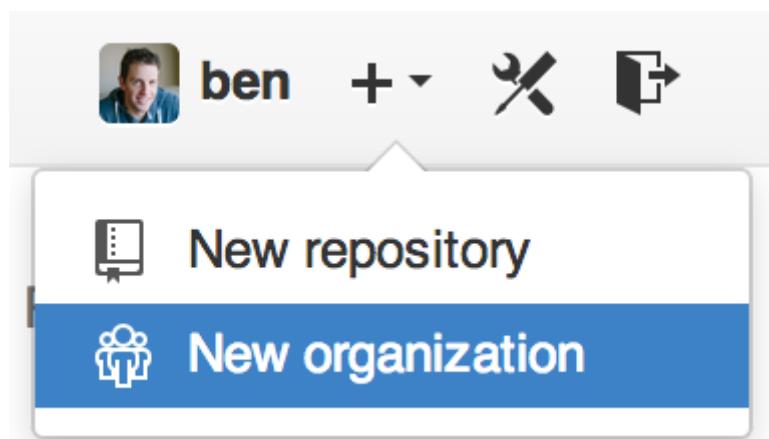
Not only does this move the repository along with all its watchers and stars to another place, it also sets up a redirect from your URL to the new place. It will also redirect clones and fetches from Git, not just web requests.

## Managing an organization

In addition to single-user accounts, GitHub has what are called Organizations. Like personal accounts, Organizational accounts have a namespace where all their projects exist, but many other things are different. These accounts represent a group of people with shared ownership of projects, and there are many tools to manage subgroups of those people. Normally these accounts are used for Open Source groups (such as “perl” or “rails”) or companies (such as “google” or “twitter”).

### Organization Basics

An organization is pretty easy to create; just click on the “+” icon at the top-right of any GitHub page, and select “New organization” from the menu.



**FIGURE 6-45**

*The “New organization” menu item.*

First you’ll need to name your organization and provide an email address for a main point of contact for the group. Then you can invite other users to be co-owners of the account if you want to.

Follow these steps and you'll soon be the owner of a brand-new organization. Like personal accounts, organizations are free if everything you plan to store there will be open source.

As an owner in an organization, when you fork a repository, you'll have the choice of forking it to your organization's namespace. When you create new repositories you can create them either under your personal account or under any of the organizations that you are an owner in. You also automatically "watch" any new repository created under these organizations.

Just like in "**Ваш аватар**", you can upload an avatar for your organization to personalize it a bit. Also just like personal accounts, you have a landing page for the organization that lists all of your repositories and can be viewed by other people.

Now let's cover some of the things that are a bit different with an organizational account.

## Teams

Organizations are associated with individual people by way of teams, which are simply a grouping of individual user accounts and repositories within the organization and what kind of access those people have in those repositories.

For example, say your company has three repositories: `frontend`, `backend`, and `deployscripts`. You'd want your HTML/CSS/Javascript developers to have access to `frontend` and maybe `backend`, and your Operations people to have access to `backend` and `deployscripts`. Teams make this easy, without having to manage the collaborators for every individual repository.

The Organization page shows you a simple dashboard of all the repositories, users and teams that are under this organization.

The screenshot shows the GitHub organization page for 'chaconcorp'. On the left, there's a list of repositories: 'deployscripts' (scripts for deployment, updated 16 hours ago), 'backend' (Backend Code, updated 16 hours ago), and 'frontend' (Frontend Code, updated 16 hours ago). On the right, there are sections for 'People' (listing 'dragonchacon' and 'schacon') and 'Teams' (listing 'Frontend Developers' with 2 members and 2 repositories, and 'Ops' with 3 members and 1 repository). A sidebar on the far right has a 'Create new team' button.

**FIGURE 6-46**  
*The Organization page.*

To manage your Teams, you can click on the Teams sidebar on the right hand side of the page in **Figure 6-46**. This will bring you to a page you can use to add members to the team, add repositories to the team or manage the settings and access control levels for the team. Each team can have read only, read/write or administrative access to the repositories. You can change that level by clicking the “Settings” button in **Figure 6-47**.

The screenshot shows the GitHub Team page for 'Frontend Developers'. It displays 2 members and 2 repositories. The team grants 'Admin' access. Members listed are 'tonychacon' (Tony Chacon) and 'schacon' (Scott Chacon), each with a 'Remove' button. A 'Leave' button and a 'Settings' button are also visible. A note at the bottom states: "This team grants Admin access: members can read from, push to, and add collaborators to the team's repositories."

**FIGURE 6-47**  
*The Team page.*

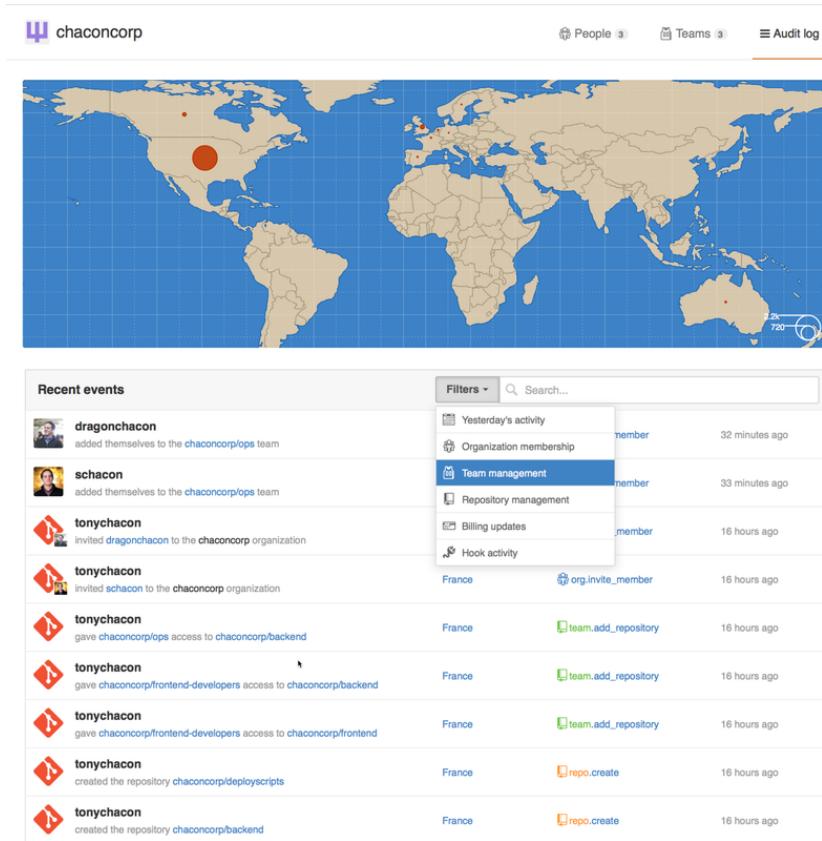
When you invite someone to a team, they will get an email letting them know they've been invited.

Additionally, team @mentions (such as @acmecorp/frontend) work much the same as they do with individual users, except that **all** members of the team are then subscribed to the thread. This is useful if you want the attention from someone on a team, but you don't know exactly who to ask.

A user can belong to any number of teams, so don't limit yourself to only access-control teams. Special-interest teams like ux, css, or refactoring are useful for certain kinds of questions, and others like legal and colorblind for an entirely different kind.

## **Audit Log**

Organizations also give owners access to all the information about what went on under the organization. You can go to the *Audit Log* tab and see what events have happened at an organization level, who did them and where in the world they were done.

**FIGURE 6-48***The Audit log.*

You can also filter down to specific types of events, specific places or specific people.

## Scripting GitHub

So now we've covered all of the major features and workflows of GitHub, but any large group or project will have customizations they may want to make or external services they may want to integrate.

Luckily for us, GitHub is really quite hackable in many ways. In this section we'll cover how to use the GitHub hooks system and its API to make GitHub work how we want it to.

## Hooks

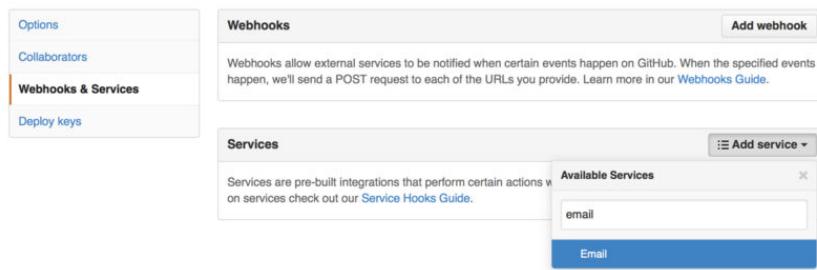
The Hooks and Services section of GitHub repository administration is the easiest way to have GitHub interact with external systems.

### SERVICES

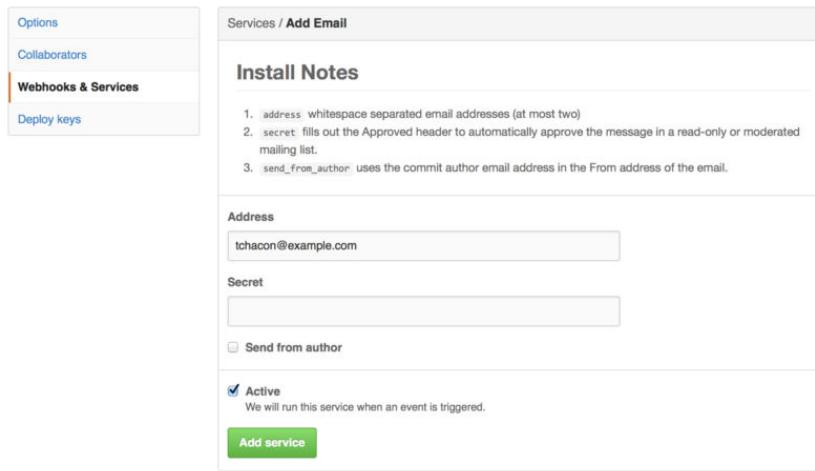
First we'll take a look at Services. Both the Hooks and Services integrations can be found in the Settings section of your repository, where we previously looked at adding Collaborators and changing the default branch of your project. Under the “Webhooks and Services” tab you will see something like **Figure 6-49**.

**FIGURE 6-49**

*Services and Hooks configuration section.*



There are dozens of services you can choose from, most of them integrations into other commercial and open source systems. Most of them are for Continuous Integration services, bug and issue trackers, chat room systems and documentation systems. We'll walk through setting up a very simple one, the Email hook. If you choose “email” from the “Add Service” dropdown, you’ll get a configuration screen like **Figure 6-50**.

**FIGURE 6-50**

*Email service configuration.*

In this case, if we hit the “Add service” button, the email address we specified will get an email every time someone pushes to the repository. Services can listen for lots of different types of events, but most only listen for push events and then do something with that data.

If there is a system you are using that you would like to integrate with GitHub, you should check here to see if there is an existing service integration available. For example, if you’re using Jenkins to run tests on your codebase, you can enable the Jenkins builtin service integration to kick off a test run every time someone pushes to your repository.

## HOOKS

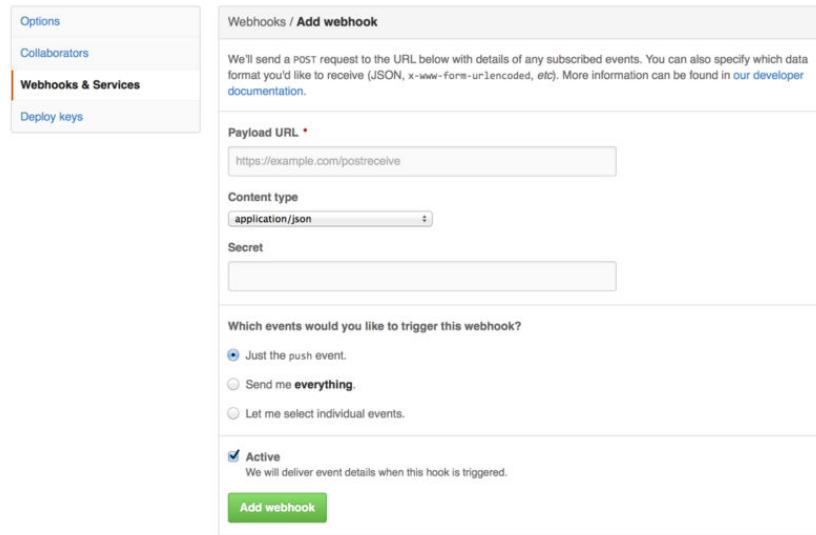
If you need something more specific or you want to integrate with a service or site that is not included in this list, you can instead use the more generic hooks system. GitHub repository hooks are pretty simple. You specify a URL and GitHub will post an HTTP payload to that URL on any event you want.

Generally the way this works is you can setup a small web service to listen for a GitHub hook payload and then do something with the data when it is received.

To enable a hook, you click the “Add webhook” button in **Figure 6-49**. This will bring you to a page that looks like **Figure 6-51**.

**FIGURE 6-51**

*Web hook configuration.*



The configuration for a web hook is pretty simple. In most cases you simply enter a URL and a secret key and hit “Add webhook”. There are a few options for which events you want GitHub to send you a payload for — the default is to only get a payload for the push event, when someone pushes new code to any branch of your repository.

Let’s see a small example of a web service you may set up to handle a web hook. We’ll use the Ruby web framework Sinatra since it’s fairly concise and you should be able to easily see what we’re doing.

Let’s say we want to get an email if a specific person pushes to a specific branch of our project modifying a specific file. We could fairly easily do that with code like this:

```

require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
 push = JSON.parse(request.body.read) # parse the JSON

 # gather the data we're looking for
 pusher = push["pusher"]["name"]
 branch = push["ref"]

 # get a list of all the files touched
 files = push["commits"].map do |commit|
 commit["files"]
 end

 # filter the files to just the ones that changed
 files = files.select { |file| file["status"] == "changed" }

 # if there were any changes, send an email
 if !files.empty?
 Mailer.deliver_push_email(pusher, branch, files)
 end
end

```

```
 commit['added'] + commit['modified'] + commit['removed']
end
files = files.flatten.uniq

check for our criteria
if pusher == 'schacon' &&
 branch == 'ref/heads/special-branch' &&
 files.include?('special-file.txt')

 Mail.deliver do
 from 'tchacon@example.com'
 to 'tchacon@example.com'
 subject 'Scott Changed the File'
 body "ALARM"
 end
end
end
```

Here we're taking the JSON payload that GitHub delivers us and looking up who pushed it, what branch they pushed to and what files were touched in all the commits that were pushed. Then we check that against our criteria and send an email if it matches.

In order to develop and test something like this, you have a nice developer console in the same screen where you set the hook up. You can see the last few deliveries that GitHub has tried to make for that webhook. For each hook you can dig down into when it was delivered, if it was successful and the body and headers for both the request and the response. This makes it incredibly easy to test and debug your hooks.

**FIGURE 6-52**

*Web hook debugging information.*

The screenshot shows the GitHub Webhook debugger interface. At the top, there's a header 'Recent Deliveries' with three entries:

- A red warning icon next to a blue file icon, followed by the ID '4aeae280-4e38-11e4-9bac-c130e992644b'. To its right is the timestamp '2014-10-07 17:40:41' and a '...' button.
- A green checkmark icon next to a blue file icon, followed by the ID 'aff20880-4e37-11e4-9089-35319435e08b'. To its right is the timestamp '2014-10-07 17:36:21' and a '...' button.
- A green checkmark icon next to a blue file icon, followed by the ID '90f37680-4e37-11e4-9508-227d13b2ccfc'. To its right is the timestamp '2014-10-07 17:35:29' and a '...' button.

Below this is a navigation bar with tabs: 'Request' (selected), 'Response' (green), and 'Completed in 0.61 seconds.' To the right is a 'Redeliver' button.

The main area is divided into two sections: 'Headers' and 'Payload'.

**Headers:**

```

Request URL: https://hooks.example.com/payload
Request method: POST
content-type: application/json
Expect:
User-Agent: GitHub-Hookshot/64a1910
X-Github-Delivery: 90f37680-4e37-11e4-9508-227d13b2ccfc
X-Github-Event: push

```

**Payload:**

```

{
 "ref": "refs/heads/remove-whitespace",
 "before": "99d4fe5bffaf827f8a9e7cde00ccb0ab06a35e48",
 "after": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
 "created": false,
 "deleted": false,
 "forced": false,
 "base_ref": null,
 "compare": "https://github.com/tonychacon/fade/compare/99d4fe5bffaf...9370a6c33493",
 "commits": [
 {
 "id": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
 "distinct": true,
 "message": "remove whitespace",
 "timestamp": "2014-10-07T17:35:22+02:00",
 "url": "https://github.com/tonychacon/fade/commit/9370a6c3349331bac7e4c3c78c10bc8460"
 }
]
}

```

The other great feature of this is that you can redeliver any of the payloads to test your service easily.

For more information on how to write webhooks and all the different event types you can listen for, go to the GitHub Developer documentation at: <https://developer.github.com/webhooks/>

## The GitHub API

Services and hooks give you a way to receive push notifications about events that happen on your repositories, but what if you need more information about these events? What if you need to automate something like adding collaborators or labeling issues?

This is where the GitHub API comes in handy. GitHub has tons of API endpoints for doing nearly anything you can do on the website in an automated fashion. In this section we'll learn how to authenticate and connect to the API, how to comment on an issue and how to change the status of a Pull Request through the API.

## Basic Usage

The most basic thing you can do is a simple GET request on an endpoint that doesn't require authentication. This could be a user or read-only information on an open source project. For example, if we want to know more about a user named "schacon", we can run something like this:

```
$ curl https://api.github.com/users/schacon
{
 "login": "schacon",
 "id": 70,
 "avatar_url": "https://avatars.githubusercontent.com/u/70",
...
 "name": "Scott Chacon",
 "company": "GitHub",
 "following": 19,
 "created_at": "2008-01-27T17:19:28Z",
 "updated_at": "2014-06-10T02:37:23Z"
}
```

There are tons of endpoints like this to get information about organizations, projects, issues, commits — just about anything you can publicly see on GitHub. You can even use the API to render arbitrary Markdown or find a `.gitignore` template.

```
$ curl https://api.github.com/gitignore/templates/Java
{
 "name": "Java",
 "source": "*.class

Mobile Tools for Java (J2ME)
.mtj.tmp/

Package Files
*.jar
*.war
*.ear

virtual machine crash logs, see http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
```

"

}

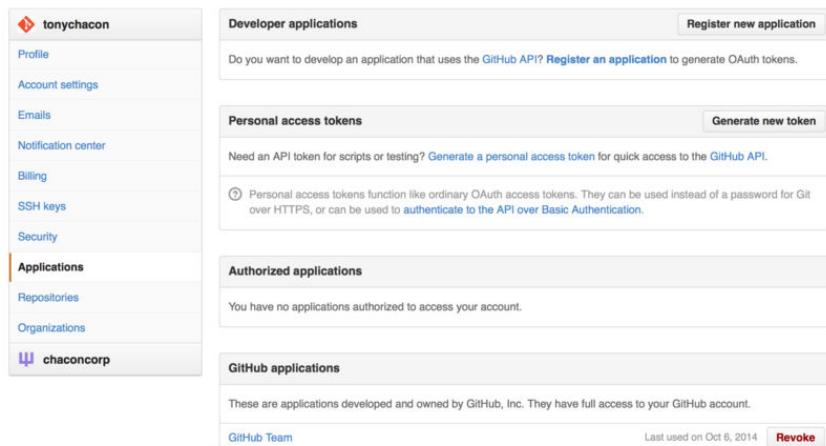
## Commenting on an Issue

However, if you want to do an action on the website such as comment on an Issue or Pull Request or if you want to view or interact with private content, you'll need to authenticate.

There are several ways to authenticate. You can use basic authentication with just your username and password, but generally it's a better idea to use a personal access token. You can generate this from the "Applications" tab of your settings page.

**FIGURE 6-53**

Generate your access token from the "Applications" tab of your settings page.



It will ask you which scopes you want for this token and a description. Make sure to use a good description so you feel comfortable removing the token when your script or application is no longer used.

GitHub will only show you the token once, so be sure to copy it. You can now use this to authenticate in your script instead of using a username and password. This is nice because you can limit the scope of what you want to do and the token is revokable.

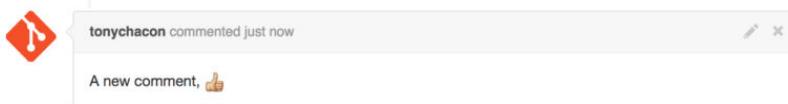
This also has the added advantage of increasing your rate limit. Without authenticating, you will be limited to 60 requests per hour. If you authenticate you can make up to 5,000 requests per hour.

So let's use it to make a comment on one of our issues. Let's say we want to leave a comment on a specific issue, Issue #6. To do so we have to do an HTTP

POST request to `repos/<user>/<repo>/issues/<num>/comments` with the token we just generated as an Authorization header.

```
$ curl -H "Content-Type: application/json" \
 -H "Authorization: token TOKEN" \
 --data '{"body": "A new comment, :+1:"}' \
 https://api.github.com/repos/schacon/blink/issues/6/comments
{
 "id": 58322100,
 "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-58322100",
 ...
 "user": {
 "login": "tonychacon",
 "id": 7874698,
 "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
 "type": "User",
 },
 "created_at": "2014-10-08T07:48:19Z",
 "updated_at": "2014-10-08T07:48:19Z",
 "body": "A new comment, :+1:"
}
```

Now if you go to that issue, you can see the comment that we just successfully posted as in **Figure 6-54**.

**FIGURE 6-54**

*A comment posted from the GitHub API.*

You can use the API to do just about anything you can do on the website — creating and setting milestones, assigning people to Issues and Pull Requests, creating and changing labels, accessing commit data, creating new commits and branches, opening, closing or merging Pull Requests, creating and editing teams, commenting on lines of code in a Pull Request, searching the site and on and on.

## Changing the Status of a Pull Request

One final example we'll look at since it's really useful if you're working with Pull Requests. Each commit can have one or more statuses associated with it and there is an API to add and query that status.

Most of the Continuous Integration and testing services make use of this API to react to pushes by testing the code that was pushed, and then report back if that commit has passed all the tests. You could also use this to check if the commit message is properly formatted, if the submitter followed all your contribution guidelines, if the commit was validly signed — any number of things.

Let's say you set up a webhook on your repository that hits a small web service that checks for a `Signed-off-by` string in the commit message.

```

require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
 push = JSON.parse(request.body.read) # parse the JSON
 repo_name = push['repository']['full_name']

 # look through each commit message
 push["commits"].each do |commit|

 # look for a Signed-off-by string
 if /Signed-off-by/.match commit['message']
 state = 'success'
 description = 'Successfully signed off!'
 else
 state = 'failure'
 description = 'No signoff found.'
 end

 # post status to GitHub
 sha = commit["id"]
 status_url = "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

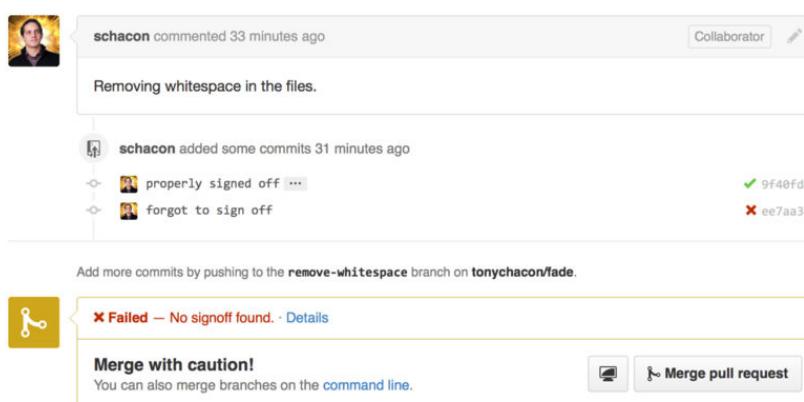
 status = {
 "state" => state,
 "description" => description,
 "target_url" => "http://example.com/how-to-signoff",
 "context" => "validate/signoff"
 }
 HTTParty.post(status_url,
 :body => status.to_json,
 :headers => {
 'Content-Type' => 'application/json',
 'User-Agent' => 'tonychacon/signoff',
 'Authorization' => "token #{ENV['TOKEN']}" })
 end
end

```

Hopefully this is fairly simple to follow. In this web hook handler we look through each commit that was just pushed, we look for the string *Signed-off-by* in the commit message and finally we POST via HTTP to the /repos/<user>/<repo>/statuses/<commit\_sha> API endpoint with the status.

In this case you can send a state (*success*, *failure*, *error*), a description of what happened, a target URL the user can go to for more information and a “context” in case there are multiple statuses for a single commit. For example, a testing service may provide a status and a validation service like this may also provide a status — the “context” field is how they’re differentiated.

If someone opens a new Pull Request on GitHub and this hook is set up, you may see something like **Figure 6-55**.



**FIGURE 6-55**

*Commit status via the API.*

You can now see a little green check mark next to the commit that has a “Signed-off-by” string in the message and a red cross through the one where the author forgot to sign off. You can also see that the Pull Request takes the status of the last commit on the branch and warns you if it is a failure. This is really useful if you’re using this API for test results so you don’t accidentally merge something where the last commit is failing tests.

## Octokit

Though we’ve been doing nearly everything through `curl` and simple HTTP requests in these examples, several open-source libraries exist that make this API available in a more idiomatic way. At the time of this writing, the supported languages include Go, Objective-C, Ruby, and .NET. Check out <http://github.com/>

*octokit* for more information on these, as they handle much of the HTTP for you.

Hopefully these tools can help you customize and modify GitHub to work better for your specific workflows. For complete documentation on the entire API as well as guides for common tasks, check out <https://developer.github.com>.

## Заключение

Теперь вы полноценный пользователь Гитхаба. Вы знаете как создать аккаунт, управлять организацией, создавать и обновлять репозитории, помогать другим проектам и принимать чужой вклад в свой проект. В следующей главе вы узнаете про ещё более мощные инструменты и получите советы для решения сложных ситуаций, которые сделают вас настоящим мастером в Git.

# Инструменты Git

К этому моменту вы уже изучили большинство повседневных команд и способов организации рабочего процесса, которые необходимы для управления Git репозиторием, используемого для управления вашим исходным кодом. Вы выполнили основные задания по отслеживанию и сохранению файлов в Git, вооружились мощью области подготовленных изменений, легковесного ветвления и слияния.

Теперь настало время познакомиться с некоторыми очень мощными возможностями Git, которые при повседневной работе вам, наверное, не потребуются, но в какой-то момент могут оказаться полезными.

## Выбор ревизии

Git позволяет различными способами указать фиксации или их диапазоны. Эти способы не всегда очевидны, но их полезно знать.

### Одиночные ревизии

Конечно, вы можете ссылаться на фиксацию по ее SHA-1 хешу, но существуют более удобные для человека способы. В данном разделе описываются различные способы обращения к одной фиксации.

#### Сокращенный SHA-1

Git достаточно умен, чтобы понять какая фиксация имеется ввиду по нескольким первым символам ее хеша, если указанная часть SHA-1 имеет в длину по крайней мере четыре символа и однозначна – то есть в текущем репозитории существует только один объект с таким частичным SHA-1.

Например, предположим, чтобы найти некоторую фиксацию, вы выполнили команду `git log` и нашли фиксацию, в которой добавили определенную функциональность:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date: Fri Jan 2 18:32:33 2009 -0800

 fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 15:08:43 2008 -0800

 Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 14:58:32 2008 -0800

 added some blame and merge stuff
```

Предположим, что в нашем примере это фиксация `1c002dd`.... Если вы хотите выполнить для нее `git show`, то следующие команды эквивалентны (предполагается, что сокращения однозначны):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git может вычислить уникальные сокращения для ваших значений SHA-1. Если вы передадите опцию `--abbrev-commit` команде `git log`, в выводе будут использоваться сокращенные значения, сохраняющие уникальность; по умолчанию используется семь символов, но для сохранения уникальности SHA-1 могут использоваться более длинные значения.

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

Обычно от восьми до десяти символов более чем достаточно для сохранения уникальности значений в проекте.

Например, в ядре Linux, который является довольно большим проектом с более чем 450 тыс. фиксаций и 3.6 млн. объектов, отсутствуют объекты, чьи SHA-1 совпадают более чем в 11 первых символах.

### НЕБОЛЬШОЕ ЗАМЕЧАНИЕ О **SHA-1**

Большинство людей в этом месте начинают беспокоиться о том, что будет, если у них в репозитории случайно появятся два объекта с одинаковыми значениями SHA-1. Что тогда?

Если вы вдруг зафиксируете объект, который имеет такое же значение SHA-1, как и предыдущий объект в вашем репозитории, Git увидит этот предыдущий объект в своей базе и посчитает, что он уже был записан. Если вы позже попытаетесь переключиться на этот объект, то вы всегда будете получать данные первого объекта.

Однако, вы должны осознавать, насколько маловероятен такой сценарий. Длина SHA-1 составляет 20 байт или 160 бит. Количество случайно хешированных объектов, необходимых для достижения 50% вероятности возникновения коллизии, равно примерно  $2^{80}$ . (формула для определения вероятности возникновения коллизии  $p = (n(n-1)/2) * (1/2^{160})$ ).  $2^{80}$  — это  $1.2 \times 10^{24}$ , или 1 миллион миллиардов миллиардов, что в 1200 раз больше количества песчинок на земле.

Приведем пример, чтобы дать вам представление, чего будет стоить получение коллизии SHA-1. Если бы все 6.5 миллиардов человек на Земле были программистами, и ежесекундно каждый из них производил количество кода, эквивалентное всей истории ядра Linux (3.6 миллиона Git-объектов), и отправлял его в один огромный Git репозитории, то потребовалось бы около 2 лет, пока этот репозиторий накопил бы количество объектов, достаточное для 50% вероятности возникновения SHA-1 коллизии. Более вероятно, что каждый член вашей команды в одну и туже ночь будет атакован и убит волками в несвязанных друг с другом происшествиях.

## Ссылки на ветки

Для наиболее простого способа указать фиксацию требуется существование ветки, указывающей на эту фиксацию. Тогда вы можете использовать имя ветки в любой команде Git, которая ожидает фиксацию или значение SHA-1. Например, если вы хотите просмотреть последнюю фиксацию в ветке, то следующие команды эквивалентны (предполагается, что ветка `topic1` указывает на фиксацию `ca82a6d`):

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

Если вы хотите узнать SHA-1 объекта, на который указывает ветка, или увидеть к чему сводятся все примеры в терминах SHA-1, то вы можете воспользоваться служебной командой Git, называемой `rev-parse`. Вы можете прочитать [Chapter 10](#) для получения дополнительной информации о служебных командах; в общем, команда `rev-parse` существует для низкоуровневых операций и не предназначена для ежедневного использования. Однако она может быть полезна, когда вам нужно увидеть, что в действительности происходит. Теперь вы можете выполнить `rev-parse` для вашей ветки.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

## RefLog-сокращения

Одна из вещей, которую Git выполняет в фоновом режиме, пока вы работаете – это ведение “журнала ссылок” – журнала, в котором за последние несколько месяцев сохраняется то, куда указывали HEAD и ветки.

Вы можете просмотреть свой журнал ссылок, используя команду `git reflog`:

```
$ git reflog
734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd HEAD@{2}: commit: added some blame and merge stuff
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
95df984 HEAD@{4}: commit: # This is a combination of two commits.
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

Каждый раз когда по каким-то причинам изменяется вершина вашей ветки, Git сохраняет информацию об этом в эту временную историю. И вы можете указывать старые фиксации, используя эти данные. Если вы хотите увидеть какой была HEAD вашего репозитория

пять шагов назад, то вы можете использовать ссылку `@{n}`, которую вы видели в выводе `reflog`:

```
$ git show HEAD@{5}
```

Вы можете также использовать такой синтаксис, чтобы увидеть где была ветка некоторое время назад. Например, чтобы увидеть где была ветка `master` вчера, вы можете использовать следующую команду:

```
$ git show master@{yesterday}
```

Она покажет вам, где была вчера верхушка ветки. Такой способ работает только для данных, которые всё еще содержатся в вашем журнале ссылок, поэтому вы не можете использовать ее для фиксаций, которые старше нескольких месяцев.

Для просмотра журнала ссылок в формате, похожем на вывод `git log`, вы можете выполнить `git log -g`:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date: Fri Jan 2 18:32:33 2009 -0800

 fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 15:08:43 2008 -0800

 Merge commit 'phedders/rdocs'
```

Важно отметить, что информация в журнале ссылок строго локальная – это лог того, что вы делали в вашем репозитории. Ссылки не будут такими же в других копиях репозитория; а сразу после первоначального клонирования репозитория, у вас будет пустой журнал ссылок, так как никаких действий в вашем репозитории

пока не производилось. Команда `git show HEAD@{2.months.ago}` будет работать только если вы клонировали проект по крайней мере два месяца назад – если вы клонировали его пять минут назад, то не получите никаких результатов.

## Ссылки на предков

Еще один популярный способ указать фиксацию – это использовать ее родословную. Если вы поместите `^` в конце ссылки, Git поймет, что нужно использовать родителя этой фиксации. Предположим, история вашего проекта выглядит следующим образом:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
* d921970 Merge commit 'phedders/rdocs'
 \\
 | * 35cfb2b Some rdoc changes
 * | 1c002dd added some blame and merge stuff
 |
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

Тогда вы можете просмотреть предыдущую фиксацию, указав `HEAD^`, что означает “родитель `HEAD`”:

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 15:08:43 2008 -0800

Merge commit 'phedders/rdocs'
```

Также вы можете указать число после `^` – например, `d921970^2` означает “второй родитель фиксации `d921970`. Такой синтаксис полезен только для фиксаций слияния, которые имеют больше одного родителя. Первым родителем является ветка, в которую вы выполняли слияние, а вторым – фиксация в ветке, которую высливали:

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
```

```
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 14:58:32 2008 -0800
```

added some blame and merge stuff

```
$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date: Wed Dec 10 22:22:03 2008 +0000
```

Some rdoc changes

Второе важное обозначение для указания предков это ~. Оно также ссылается на первого родителя, поэтому HEAD~ и HEAD^ эквивалентны. Различия становятся заметными, когда вы указываете число. HEAD~2 означает “первый родитель первого родителя” или “прадедушка” – переход к первому родителю осуществляется столько раз, сколько вы указали. Например, для показанной ранее истории, фиксацией HEAD~3 будет

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500
```

ignore \*.gem

Тоже самое можно записать как HEAD^^^, что также является первым родителем первого родителя первого родителя:

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500
```

ignore \*.gem

Вы также можете совмещать эти обозначения – можно получить второго родителя предыдущей ссылки (предполагается, что это фиксация слияния) используя запись HEAD~3^2, и так далее.

## Диапазоны фиксаций

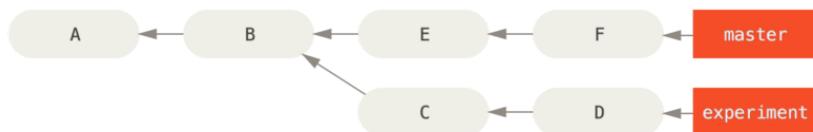
Теперь вы умеете указывать отдельные фиксации, давайте посмотрим как указывать диапазоны фиксаций. Это в частности полезно для управления вашими ветками – если у вас есть множество веток, вы можете использовать указание диапазонов фиксаций для ответа на вопрос “Что было сделано в этой ветке, что я еще не слил в основную ветку?”

### ДВЕ ТОЧКИ

Наиболее часто для указания диапазона фиксаций используется синтаксис с двумя точками. Таким образом, вы, по сути, просите Git включить в диапазон фиксаций только те, которые достижимы из одной, но не достижимы из другой. Для примера предположим, что ваша история выглядит, как представлено на **Figure 7-1**.

**FIGURE 7-1**

Пример истории  
для выбора  
диапазонов  
фиксаций.



Вы хотите посмотреть что находится в вашей экспериментальной ветке, которая еще не была слита в основную. Вы можете попросить Git отобразить в логе только такие фиксации, используя запись `master..experiment` – она означает “все фиксации, которые доступны из ветки `experiment`, но не доступны из ветки `master`”. Для краткости и наглядности в этих примерах вместо настоящего вывода лога мы будем использовать для фиксаций их буквенные обозначения из диаграммы, располагая их в нужном порядке:

```
$ git log master..experiment
D
C
```

С другой стороны, если вы хотите наоборот увидеть все фиксации ветки `master`, которых нет в ветке `experiment`, вы можете поменять имена веток в команде. При использовании записи `experiment..mas-`

ter будут отображены все фиксации ветки `master`, недоступные из ветки `experiment`:

```
$ git log experiment..master
F
E
```

Это полезно если вы хотите сохранить ветку `experiment` в актуальном состоянии и просмотреть, какие изменения нужно в нее слить. Другое частое использование такого синтаксиса – просмотр того, что будет отправлено в удаленный репозиторий.

```
$ git log origin/master..HEAD
```

Такая команда покажет вам все фиксации вашей текущей ветки, которые отсутствуют в ветке `master` удаленного репозитория `origin`. Если вы выполните `git push`, находясь на ветке отслеживающей `origin/master`, то фиксации, отображенные командой `git log origin/master..HEAD`, будут теми фиксациями, которые отправятся на сервер. Вы также можете опустить одну из частей в такой записи, Git будет считать ее равной `HEAD`. Например, вы можете получить такой же результат как в предыдущем примере, выполнив `git log origin/master..` – Git подставит `HEAD`, если одна часть отсутствует.

## МНОЖЕСТВО ТОЧЕК

Запись с двумя точками полезна как сокращение, но, возможно, вы захотите использовать более двух веток для указания нужной ревизии, например, для того, чтобы узнать какие фиксации присутствуют в любой из нескольких веток, но отсутствуют в ветке, в которой вы сейчас находитесь. Git позволяет сделать это, используя символ `^` или опцию `--not`, перед любой ссылкой, доступные фиксации из которой вы не хотите видеть. Таким образом, следующие три команды эквивалентны:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

Этот синтаксис удобен, так как позволяет указывать в запросе более двух ссылок, чего не позволяет сделать синтаксис с двумя точками. Например, если вы хотите увидеть все фиксации, доступные из `refA` и `refB`, но не доступные из `refC`, вы можете использовать одну из следующих команд:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

Это делает систему запросов ревизий более мощной и должно помочь вам лучше понять, что содержится в вашей ветке.

### ТРИ ТОЧКИ

Последний основной способ выбора ревизий – это синтаксис с тремя точками, который обозначает все фиксации, доступные хотя бы из одной ссылки, но не из обеих сразу. Вспомните пример истории фиксаций в [Figure 7-1](#). Если вы хотите узнать какие фиксации есть либо в ветке `master`, либо в `experiment`, но не в обеих сразу, вы можете выполнить:

```
$ git log master...experiment
F
E
D
C
```

Эта команда снова выводит обычный журнал фиксаций, но в нем содержится информация только об этих четырёх фиксациях, традиционно отсортированная по дате фиксаций.

В таких случаях с командой `log` часто используют опцию `--left-right`, которая отображает сторону диапазона, с которой была сделана каждая из фиксаций. Это делает данную информацию более полезной:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

С помощью этих инструментов, вам будет намного проще указать Git какую фиксацию или фиксации вы хотите изучить.

## Интерактивное индексирование

Git поставляется вместе со скриптами, которые упрощают выполнение некоторых задач из командной строки. В этом разделе мы рассмотрим несколько интерактивных команд, которые могут упростить создание коммитов, позволяя включать в них только определенный набор файлов и их частей. Эти инструменты очень полезны, если вы изменили множество файлов, а затем решили, что хотите чтобы эти изменения были в нескольких маленьких понятных коммитах, а не в одном большом и запутанном. Таким способом вы сможете гарантировать, что ваши коммиты представляют логически разделенные изменения и могут быть легко прорецензированы вашими коллегами. Если вы выполните `git add` с опцией `-i` или `--interactive`, Git перейдет в интерактивный консольный режим, отобразив что-то подобное:

```
$ git add -i
 staged unstaged path
1: unchanged +0/-1 TODO
2: unchanged +1/-1 index.html
3: unchanged +5/-1 lib/simplegit.rb

*** Commands ***
1: status 2: update 3: revert 4: add untracked
5: patch 6: diff 7: quit 8: help
What now>
```

Вы можете видеть, что эта команда показывает вашу область подготовленных изменений в уникальном представлении – вообще говоря, ту же информацию вы получите с помощью команды `git status`, но несколько более сжато и информативно. Эта команда показывает проиндексированные изменения слева, а непроиндексированные – справа.

Затем следует раздел со списком команд. С их помощью вы можете выполнить множество вещей – добавить или исключить файлы из индекса, добавить в индекс части файлов, добавить в индекс неотслеживаемые файлы и просмотреть проиндексированные изменения.

## Добавление и удаление файлов из индекса

Если вы введете 2 или и в поле ввода `What now>`, скрипт спросит у вас какие файлы вы хотите добавить в индекс:

```
What now> 2
 staged unstaged path
1: unchanged +0/-1 TODO
2: unchanged +1/-1 index.html
3: unchanged +5/-1 lib/simplegit.rb
Update>>
```

Для добавления в индекс файлов TODO и index.html, вы можете ввести их номера:

```
Update>> 1,2
 staged unstaged path
* 1: unchanged +0/-1 TODO
* 2: unchanged +1/-1 index.html
3: unchanged +5/-1 lib/simplegit.rb
Update>>
```

Символ \* у каждого из этих файлов означает, что файл выбран для индексирования. Если вы нажмете Enter, не вводя ничего в поле ввода `Update>>`, Git добавит в индекс всё, чтобы было выбрано ранее:

```
Update>>
updated 2 paths

*** Commands ***
1: status 2: update 3: revert 4: add untracked
5: patch 6: diff 7: quit 8: help
What now> 1
 staged unstaged path
1: +0/-1 nothing TODO
2: +1/-1 nothing index.html
3: unchanged +5/-1 lib/simplegit.rb
```

Как вы можете заметить, сейчас файлы TODO и index.html добавлены в индекс, а файл simplegit.rb всё еще нет. Если вы в этот момент хотите исключить файл TODO из индекса, вы можете использовать опции 3 или г (для выполнения revert):

```
*** Commands ***
1: status 2: update 3: revert 4: add untracked
5: patch 6: diff 7: quit 8: help
What now> 3
 staged unstaged path
1: +0/-1 nothing TODO
2: +1/-1 nothing index.html
3: unchanged +5/-1 lib/simplegit.rb
Revert>> 1
 staged unstaged path
* 1: +0/-1 nothing TODO
2: +1/-1 nothing index.html
3: unchanged +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

Посмотрев снова на состояние вашей рабочей директории Git, вы увидите, что файл TODO исключен из индекса:

```
*** Commands ***
1: status 2: update 3: revert 4: add untracked
5: patch 6: diff 7: quit 8: help
What now> 1
 staged unstaged path
1: unchanged +0/-1 TODO
2: +1/-1 nothing index.html
3: unchanged +5/-1 lib/simplegit.rb
```

Для того, чтобы посмотреть изменения, которые вы добавили в индекс, вы можете использовать команду б и д (для выполнения diff). Она покажет вам список добавленных в индекс файлов, и вы можете выбрать один из них, для которого вы хотите увидеть добавленные в индекс изменения. Это очень похоже на вывод команды `git diff --cached`:

```
*** Commands ***
1: status 2: update 3: revert 4: add untracked
5: patch 6: diff 7: quit 8: help
What now> 6
 staged unstaged path
1: +1/-1 nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
```

```

--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

<p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

<script type="text/javascript">

```

Использование этих основных команд в интерактивном режиме команды add, может упростить вам работу с вашей областью подготовленных изменений.

## Индексирование по частям

В Git существует возможность индексировать не только файлы целиком, но и некоторые их части. Например, если вы сделали в файле simplegit.rb два изменения и хотите добавить в индекс только одно из них, добиться этого в Git очень легко. В поле ввода в режиме интерактивного индексирования введите 5 или p (для выполнения patch). Git спросит у вас какие файлы вы хотите добавить в индекс частично; а затем для каждой части выбранных файлов он будет показывать изменения в ней и спрашивать хотите ли вы добавить в индекс эту часть:

```

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
 end

 def log(treeish = 'master')
- command("git log -n 25 #{treeish}")
+ command("git log -n 30 #{treeish}")
 end

 def blame(path)
Stage this hunk [y,n,a,d,/,,j,J,g,e,?]?

```

В это точке у вас есть множество вариантов дальнейших действий. Если вы введете ?, Git отобразит, что именно вы можете сделать:

```
Добавить в индекс эту часть [у,п,а,д,/ ,ј,Ј,г,е,?] ? ?
у - добавить в индекс эту часть
п - не добавлять в индекс эту часть
а - добавить в индекс эту и все оставшиеся в этом файле части
д - не добавлять в индекс эту и все оставшиеся в этом файле части
г - перейти к некоторой части файла (г - показывает список частей и затем выполняет переход,
/ - найти часть, соответствующую регулярному выражению
ј - отложить принятие решения по этой части, перейти к следующей части, решение по которой н
Ј - отложить принятие решения по этой части, перейти к следующей части
к - отложить принятие решения по этой части, перейти к предыдущей части, решение по которой
К - отложить принятие решения по этой части, перейти к предыдущей части
с - разбить текущую часть на части меньшего размера
е - вручную отредактировать текущую часть
? - отобразить помощь
```

Обычно вы будете вводить у или п, если вы хотите индексировать каждую часть по отдельности, но индексация всех частей в некоторых файлах или откладывание решения по индексации части также может быть полезным. Если вы добавили в индекс одну часть файла, но не добавили другую, состояние вашей рабочей директории будет подобно приведенному далее:

```
What now> 1
 staged unstaged path
1: unchanged +0/-1 TODO
2: +1/-1 nothing index.html
3: +1/-1 +4/-0 lib/simplegit.rb
```

Обратите внимание на состояние файла simplegit.rb. Оно говорит вам, что часть строк файла добавлены в индекс, а часть нет. Таким образом, вы частично проиндексировали этот файл. В данный момент вы можете выйти из интерактивного режима команды `git add` и выполнить `git commit`, чтобы зафиксировать частично проиндексированные файлы.

Также вам не обязательно находиться в интерактивном режиме индексирования файлов для выполнения частичной индексации файлов – вы также можете запустить ее, используя команды `git add -r` или `git add --patch`.

Более того, вы можете использовать работу с отдельными частями файлов для частичного восстановления файлов с помощью команды `reset --patch`, для переключения частей файлов с помощью команды `checkout --patch` и для прибережения частей файлов с

помощью `stash save --patch`. Мы рассмотрим каждую из этих команд более подробно, когда будем изучать более продвинутые варианты их использования.

## Прибережение и очистка

Часто пока вы работаете над одной частью вашего проекта и всё находится в беспорядке, у вас возникает желание сменить ветку и поработать над чем-то еще. Сложность при этом заключается в том, что вы не хотите фиксировать наполовину сделанную работу только для того, чтобы иметь возможность вернуться к ней позже. Справиться с ней помогает команда `git stash`.

Операция `stash` берет измененное состояние вашей рабочей директории, то есть измененные отслеживаемые файлы и проиндексированные изменения, и сохраняет их в хранилище незавершенных изменений, которые вы можете в любое время применить обратно.

## Прибережение ваших наработок

Для примера, предположим, что вы перешли в свой проект, начали работать над несколькими файлами и, возможно, добавили в индекс изменения одного из них. Если вы выполните `git status`, то увидите ваше измененное состояние:

```
$ git status
Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

 modified: index.html

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: lib/simplegit.rb
```

Теперь вы хотите сменить ветку, но пока не хотите фиксировать ваши текущие наработки; поэтому вы спрячете эти изменения. Для того, чтобы спрятать изменение в выделенное для этого специальное хранилище, выполните `git stash` или `git stash save`:

```
$ git stash
Saved working directory and index state \
 "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

Теперь ваша рабочая директория не содержит изменений:

```
$ git status
On branch master
nothing to commit, working directory clean
```

В данный момент вы можете легко переключать ветки и работать в любой; ваши изменения сохранены. Чтобы посмотреть список спрятанных изменений, вы можете использовать `git stash list`:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

В данном примере, предварительно были припасены два изменения, поэтому теперь вам доступны три различных отложенных наработки. Вы можете применить только что спрятанные изменения, используя команду, указанную в выводе исходной команды: `git stash apply`. Если вы хотите применить одно из предыдущих спрятанных изменений, вы можете сделать это, используя его имя, вот так: `git stash apply stash@{2}`. Если вы не укажете имя, то Git попытается восстановить самое последнее спрятанное изменение:

```
$ git stash apply
On branch master
Changed but not updated:
(use "git add <file>..." to update what will be committed)
#
modified: index.html
modified: lib/simplegit.rb
#
```

Как видите, Git восстановил в файлах изменения, которые вы отменили ранее, когда прятали свои наработки. В данном случае при

применении отложенных наработок ваша рабочая директория была без изменений, а вы пытались применить их в той же ветке, в которой вы их и сохранили; но отсутствие изменений в рабочей директории и применение их в той же ветке не являются необходимыми условиями для успешного восстановления спрятанных наработок. Вы можете спрятать изменения, находят в одной ветке, а затем переключиться на другую и попробовать восстановить эти изменения. Также при восстановлении спрятанных наработок в вашей рабочей директории могут присутствовать измененные и незафиксированные файлы – Git выдаст конфликты слияния, если не сможет восстановить какие-то наработки.

Спрятанные изменения будут применены к вашим файлам, но файлы, которые вы ранее добавляли в индекс, не будут добавлены туда снова. Для того, чтобы это было сделано, вы должны запустить `git stash apply` с опцией `--index`, при которой команда попытается восстановить изменения в индексе. Если вы выполните команду таким образом, то полностью восстановите ваше исходное состояние:

```
$ git stash apply --index
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
#
modified: index.html
#
Changed but not updated:
(use "git add <file>..." to update what will be committed)
#
modified: lib/simplegit.rb
#
```

Команда `apply` только пытается восстановить спрятанные наработки – при этом они продолжат оставаться в хранилище. Для того, чтобы удалить их, вы можете выполнить `git stash drop`, указав имя удаляемых изменений:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Вы также можете выполнить `git stash pop`, чтобы применить спрятанные изменения и тут же удалить их из хранилища.

## Продуктивное прибережение

У откладываемых изменений есть несколько дополнительных вариантов использования, которые также могут быть полезны. Первый – это использование довольно популярной опции `--keep-index` с командой `stash save`. Она просит Git не прятать то, что вы уже добавили в индекс командой `git add`.

Это может быть полезно в случае, когда вы сделали какие-то изменения, но хотите зафиксировать только часть из них, а к оставшимся вернуться через некоторое время.

```
$ git status -s
M index.html
M lib/simplegit.rb

$ git stash --keep-index
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
M index.html
```

Другой распространенный вариант, который вы, возможно, захотите использовать – это спрятать помимо отслеживаемых файлов также и неотслеживаемые. По умолчанию `git stash` будет сохранять только файлы, которые уже добавлены в индекс. Если вы укажете `--include-untracked` или `-u`, Git также спрячет все неотслеживаемые файлы, которые вы создали.

```
$ git status -s
M index.html
M lib/simplegit.rb
?? new-file.txt

$ git stash -u
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file
```

```
$ git status -s
$
```

И наконец, если вы укажете флаг `--patch`, Git не будет ничего прятать, а вместо этого в интерактивном режим спросит вас о том, какие из изменений вы хотите спрятать, а какие оставить в вашей рабочей директории.

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
 return `#{git_cmd} 2>&1`.chomp
 end
 end

+
+ def show(treeish = 'master')
+ command("git show #{treeish}")
+ end

end
test
Stash this hunk [y,n,q,a,d,/,e,?]? y

Saved working directory and index state WIP on master: 1b65b17 added the index fil
```

## Создание ветки из спрятанных изменений

Если вы спрятали некоторые изменения, оставили их на время, а сами продолжили работать в той же ветке, у вас могут возникнуть проблемы с восстановлением наработок. Если восстановление будет затрагивать файл, который уже был изменен с момента сохранения наработок, то вы получите конфликт слияния и должны будете попытаться разрешить его. Если вам нужен более простой способ снова протестировать спрятанные изменения, вы можете выполнить команду `git stash branch`, которая создаст для вас новую ветку, перейдёт на фиксацию, на котором вы были, когда прятали свои наработки, применит на нем эти наработки и затем, если они применились успешно, удалит эти спрятанные изменения:

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
On branch testchanges
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
#
modified: index.html
#
Changed but not updated:
(use "git add <file>..." to update what will be committed)
#
modified: lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

Это удобное сокращение для того, чтобы легко восстановить спрятанные изменения и поработать над ними в новой ветке.

## Очистка вашей рабочей директории

Наконец, у вас может возникнуть желание не прятать некоторые из изменений или файлов в вашей рабочей директории, а просто избавиться от них. Команда `git clean` сделает это для вас.

Одной из распространенных причин для этого может быть удаление мусора, который был сгенерирован при слиянии или внешними утилитами, или удаление артефактов сборки в процессе ее очистки.

Вам нужно быть очень аккуратными с этой командой, так как она предназначена для удаления неотслеживаемых файлов из вашей рабочей директории. Даже если вы передумаете, очень часто нельзя восстановить содержимое таких файлов. Более безопасным вариантом явление использование команды `git stash --all` для удаления всего, но с сохранением этого в виде спрятанных изменений.

Предположим, вы хотите удалить мусор и очистить вашу рабочую директорию; вы можете сделать это с помощью `git clean`. Для удаления всех неотслеживаемых файлов в вашей рабочей директории, вы можете выполнить команду `git clean -f -d`, которая удалит все файлы и также все директории, которые в результате станут пустыми. Опция `-f` значит *force* или другими словами “действительно выполнить это”.

Если вы хотите только посмотреть, что будет сделано, вы можете запустить команду с опцией `-n`, которая означает “имитируй работу команды и скажи мне, что ты будешь удалять”.

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

По умолчанию команда `git clean` будет удалять только неотслеживаемые файлы, которая не добавлены в список игнорируемых. Любой файл, который соответствует шаблону в вашем `.gitignore`, или другие игнорируемые файлы не будут удалены. Если вы хотите удалить и эти файлы (например, удалить все `.o`-файлы, генерируемые в процессе сборки, и таким образом полностью очистить сборку), вы можете передать команде очистки опцию `-x`.

```
$ git status -s
M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

Если вы не знаете, что сделает при запуске команда `git clean`, всегда сначала выполняйте ее с опцией `-n`, чтобы проверить дважды, перед заменой `-n` на `-f` и выполнением настоящей очистки. Другой способ, который позволяет вам более тщательно контролировать сам процесс – это выполнение команды с опцией `-i` (в “интерактивном” режиме).

Ниже выполнена команда очистки в интерактивном режиме.

```
$ git clean -x -i
Would remove the following items:
build.TMP test.o
```

```
*** Commands ***
1: clean 2: filter by pattern 3: select by numbers 4: ask each
6: help
What now>
```

Таким образом, вы сможете пройтись отдельно по каждому или выбранным с помощью шаблона файлам для их удаления в интерактивном режиме.

## Подпись результатов вашей работы

Git является криптографически защищенной системой, но эти механизмы сложны в использовании. На случай, если вы берете у кого-то в интернете результаты его работы и хотите проверить, что коммиты действительно получены из доверенного источника, в Git есть несколько способов подписать и проверить исходники, используя GPG.

### Введение в GPG

Во-первых, если вы хотите что-либо подписать, вам необходим настроенный GPG и персональный ключ.

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg

pub 2048R/0A46826A 2014-06-04
uid Scott Chacon (Git signing key) <schacon@gmail.com>
sub 2048R/874529A9 2014-06-04
```

Если у вас нет ключа, то можете сгенерировать его командой `gpg --gen-key`.

```
gpg --gen-key
```

Если у вас есть приватный ключ для подписи, вы можете настроить Git так, чтобы этот ключ использовался для подписи, установив значение параметра `user.signingkey`.

```
git config --global user.signingkey 0A46826A
```

Теперь, если вы захотите, Git будет по умолчанию использовать этот ключ для подписи тегов и коммитов.

## Подпись тегов

Если вы настроили приватный ключ GPG, то можете использовать его для подписи новых тегов. Для этого вы должны использовать опцию `-s` вместо `-a`:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
```

```
You need a passphrase to unlock the secret key for
user: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

Если теперь для этого тега вы выполните `git show`, то увидите прикрепленную к нему свою GPG подпись:

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date: Sat May 3 20:29:41 2014 -0700

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAABAgAGBQJTZbQlAAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kC3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfvcimnSDeSvzCpWAHl7h8Wj6hhqePmLm9lAYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysqqjcpT8+iQM1PbLGfHR4XAh0qN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number
```

## Проверка тегов

Для проверки подписанного тега вы можете воспользоваться командой `git tag -v [tag-name]`. Она использует GPG для проверки подписи. Чтобы всё это правильно работало нужно, чтобы публичный ключ автора присутствовал в вашем хранилище ключей:

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg: aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4 F311 9B9A
```

Если у вас отсутствует публичный ключ автора, вы увидите что-то подобное:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

## Подпись коммитов

В новых версиях Git (начиная с v1.7.9), вы также можете подписывать отдельные коммиты. Если вам нужно подписывать непосредственно сами коммиты, а не теги, вы можете передать команде `git commit` опцию `-S`.

```
$ git commit -a -S -m 'signed commit'

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

[master 5c3386c] signed commit
 4 files changed, 4 insertions(+), 24 deletions(-)
```

```
rewrite Rakefile (100%)
create mode 100644 lib/git.rb
```

Для просмотра и проверки таких подписей у команды `git log` есть опция `--show-signature`.

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun 4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>"
Author: Scott Chacon <schacon@gmail.com>
Date: Wed Jun 4 19:49:17 2014 -0700

signed commit
```

Также вы можете, используя формат с `%G?`, настроить `git log` так, чтобы он проверял и отображал любую обнаруженную подпись.

```
$ git log --pretty=format:"%h %G? %aN %s"
5c3386c G Scott Chacon signed commit
ca82a6d N Scott Chacon changed the version number
085bb3b N Scott Chacon removed unnecessary test code
a11bef0 N Scott Chacon first commit
```

В данном примере видно, что только последний коммит корректно подписан, а все предыдущие нет.

В Git, начиная с версии 1.8.3, команды `git merge` и `git pull` с помощью опции `--verify-signatures` можно заставить проверять и отклонять слияния, если коммит не содержит доверенной GPG подписи.

Если вы воспользуетесь этой опцией при слиянии с веткой, которая содержит неподписанные или некорректно подписанные коммиты, то слияние завершится ошибкой.

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

Если слияемая ветка содержит только корректно подписанные коммиты, команда слияние сначала покажет все проверенные ей подписи, а затем выполнит само слияние.

```
$ git merge --verify-signatures signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key) <schacon@gmail.com>
Updating 5c3386c..13ad65e
Fast-forward
 README | 2 ++
 1 file changed, 2 insertions(+)
```

Также с командой `git merge` вы можете использовать опцию `-S`, в этом случае полученный в результате слияния коммит будет подписан. В следующем примере выполняется и проверка каждого коммита сливающей ветки, и подпись полученного в результате слияния коммита.

```
$ git merge --verify-signatures -S signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key) <schacon@gmail.com>

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 2 insertions(+)
```

## Каждый должен подписываться

Конечно, подписывать теги и коммиты это хорошая идея, но если вы решите воспользоваться ей в вашем обычном рабочем процессе, то должны удостовериться, что все в вашей команде понимают, как выполнять подпись. Если этого не сделать, то в итоге вам придется потратить много времени, объясняя коллегами, как перезаписать их коммиты подписанными версиями. Удостоверьтесь, что вы разбираетесь в GPG и преимуществах, которые несут подписи, перед тем как использовать их как часть вашего рабочего процесса.

## Поиск

Вне зависимости от размера кодовой базы, часто возникает необходимость поиска места вызова/определения функции или получения истории изменения метода. Git предоставляет несколько

полезных утилит, с помощью которых легко и просто осуществлять поиск по коду и коммитам. Мы обсудим некоторые из них.

## Git Grep

Git поставляется с командой `grep`, которая позволяет легко искать в истории коммитов или в рабочем каталоге по строке или регулярному выражению. В следующих примерах, мы обратимся к исходному коду самого Git.

По умолчанию эта команда ищет по файлам в рабочем каталоге. Для отображения нумерации строк, в которых присутствуют совпадения, вы можете передать команде опцию `-n`.

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8: return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct tm *result)
compat/gmtime.c:16: ret = gmtime_r(timep, result);
compat/mingw.c:606:struct tm *gmtime_r(const time_t *timep, struct tm *result)
compat/mingw.h:162:struct tm *gmtime_r(const time_t *timep, struct tm *result);
date.c:429: if (gmtime_r(&now, &now_tm))
date.c:492: if (gmtime_r(&time, tm)) {
git-compat-util.h:721:struct tm *git_gmtime_r(const time_t *, struct tm *);
git-compat-util.h:723:#define gmtime_r git_gmtime_r
```

Вы можете передать команде `grep` и другие интересные опции.

Например, при использовании опции `--count` Git, в отличие от предыдущего примера, выведет только те файлы, в которых найдены совпадения и их количество в каждом из этих файлов:

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:2
git-compat-util.h:2
```

Если вы хотите увидеть метод или функцию, в котором присутствует совпадение, вы можете указать опцию `-r`:

```
$ git grep -r gmtime_r *.c
date.c=static int match_multi_number(unsigned long num, char c, const char *date,
```

```
date.c: if (gmtime_r(&now, &now_tm))
date.c=static int match_digit(const char *date, struct tm *tm, int *offset, int *tm_gmt)
date.c: if (gmtime_r(&time, tm)) {
```

Здесь вы можете видеть, что `gmtime_r` вызывается из функций `match_multi_number` и `match_digit` в файле `date.c`.

Вы также можете искать сложные комбинации строк, используя опцию `--and`, которая гарантирует, что будут отображены только строки, имеющие сразу несколько совпадений. Например, давайте в кодовой базе Git версии 1.8.0 найдем строки, в которых присутствует определение константы, содержащее любую из строк “LINK” и “BUF\_MAX”.

В этом примере мы также воспользуемся опциями `--break` и `--heading`, которые помогают вывести результаты в более читаемом виде.

```
$ git grep --break --heading \
-n -e '#define' --and \(-e LINK -e BUF_MAX \) v1.8.0
v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK 0160000
74:#define S_ISGITLINK(m) (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATIONUSES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK (1 << 2)

v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

Команда `git grep` имеет несколько преимуществ перед поиском с помощью таких команд, как `grep` и `ack`. Во-первых, она действительно быстрая, во-вторых – `git grep` позволяет искать не только в рабочем каталоге, но и в любом другом дереве Git. Как вы видели, в прошлом примере мы искали в старой версии исходных кодов Git, а не в текущем снимке файлов.

## Поиск в журнале изменений Git

Возможно, вы ищете не **где** присутствует некоторое выражение, а **когда** оно существовало или было добавлено. Команда `git log` обладает некоторыми мощными инструментами для поиска определенных коммитов по содержимому их сообщений или содержимому сделанных в них изменений.

Например, если вы хотите найти, когда была добавлена константа `ZLIB_BUF_MAX`, то вы можете с помощью опции `-S` попросить Git показывать только те коммиты, в которых была добавлена или удалена эта строка.

```
$ git log -SZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

Если мы посмотрим на изменения, сделанные в этих коммитах, то увидим, что в `ef49a7a` константа была добавлена, а в `e01503b` – изменена.

Если вам нужно найти что-то более сложное, вы можете с помощью опции `-G` передать регулярное выражение.

## ПОИСК ПО ЖУРНАЛУ ИЗМЕННИЙ СТРОКИ

Другой, довольно продвинутый, поиск по истории, который бывает чересчур чайно полезным – поиск по истории изменений строки. Эта возможность была добавлена совсем недавно и пока не получила известности. Для того, чтобы ей воспользоваться нужно команде `git log` передать опцию `-L`, в результате вам будет показана история изменения функции или строки кода в вашей кодовой базе.

Например, если мы хотим увидеть все изменения, произошедшие с функцией `git_deflate_bound` в файле `zlib.c`, мы можем выполнить `git log -L :git_deflate_bound:zlib.c`. Эта команда постарается определить границы функции, выполнит поиск по истории и покажет все изменения, которые были сделаны с функцией, в виде набора патчей в обратном порядке до момента создания функции.

```
$ git log -L :git_deflate_bound:zlib.c
commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Author: Junio C Hamano <gitster@pobox.com>
Date: Fri Jun 10 11:52:15 2011 -0700
```

```

zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -85,5 +130,5 @@
-unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)
{
- return deflateBound(strm, size);
+ return deflateBound(&strm->z, size);
}

commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Author: Junio C Hamano <gitster@pobox.com>
Date: Fri Jun 10 11:18:17 2011 -0700

zlib: wrap deflateBound() too

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -81,0 +85,5 @@
+unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+{
+ return deflateBound(strm, size);
+}
+

```

Если для вашего языка программирования Git не умеет правильно определять функции и методы, вы можете передать ему регулярное выражение. Например, следующая команда выполнит такой же поиск как и предыдущая `git log -L '/unsigned long git_deflate_bound/','/^}/:zlib.c`. Также вы можете передать интервал строк или номер определенной строки, и в этом случае вы получите похожий результат.

## Исправление истории

Неоднократно при работе с Git, вам может потребоваться по какой-то причине исправить вашу историю фиксаций. Одно из преимуществ Git заключается в том, что он позволяет вам отложить принятие решений на самый последний момент. Область подготовленных изменений

позволяет вам решить, какие файлы попадут в фиксацию непосредственно перед ее выполнением; благодаря команде `stash` вы можете решить, что не хотите продолжать работу над какими-то изменениями; также вы можете изменить уже совершенные фиксации так, чтобы они выглядели совершенно другим образом. В частности, можно изменить порядок фиксаций, сообщения или измененные в фиксациях файлы, объединить вместе или разбить на части фиксации, полностью удалить фиксации – но только до того, как вы поделитесь своими наработками с другими.

В данном разделе вы узнаете, как выполнять эти очень полезные задачи. Таким образом, перед тем, как поделиться вашими наработками с другими, вы сможете привести вашу историю фиксаций к нужному виду.

## Изменение последней фиксации

Изменение вашей последней фиксации, наверное, наиболее частое исправление истории, которое вы будете выполнять. Наиболее часто с вашей последней фиксацией вам будет нужно сделать две основные операции: изменить сообщение фиксации или изменить только что сделанный снимок, добавив, изменив или удалив файлы.

Если вы хотите изменить только сообщение вашей последней фиксации, это очень просто:

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

Эта команда откроет в вашем текстовом редакторе сообщение вашей последней фиксации, для того, чтобы вы могли его исправить. Когда вы сохраните его и закроете редактор, будет создана новая фиксация, содержащая это сообщение, которая теперь и будет вашей последней фиксацией.

Если вы создали фиксацию и затем хотите изменить зафиксированный снимок, добавив или изменив файлы (возможно, вы забыли добавить вновь созданный файл, когда совершали изначальную фиксацию), то процесс выглядит в основном так же. Вы добавляете в индекс необходимые изменения, редактируя файл и выполняя для него `git add` или `git rm` для отслеживаемого файла, а последующая команда `git commit --amend` берет вашу текущую область подготовленных изменений и делает ее снимок для новой фиксации.

Вы должны быть осторожными, используя этот прием, так как при этом изменяется SHA-1 фиксации. Поэтому как и с операцией `rebase` – не изменяйте вашу последнюю фиксацию, если вы уже отправили ее в общий репозиторий.

## Изменение сообщений нескольких фиксаций

Для изменения фиксации, расположенной раньше в вашей истории, вам нужно обратиться к более сложным инструментам. В Git отсутствуют инструменты для изменения истории, но вы можете использовать команду `rebase`, чтобы перебазировать группу фиксаций туда же на HEAD, где они были изначально, вместо перемещения их в другое место. С помощью интерактивного режима команды `rebase`, вы можете останавливаться после каждой нужной вам фиксации и изменять сообщения, добавлять файлы или делать что-то другое, что вам нужно. Вы можете запустить `rebase` в интерактивном режиме, добавив опцию `-i` к `git rebase`. Вы должны указать, какие фиксации вы хотите изменить, передав команде фиксацию, на которую нужно выполнить перебазирование.

Например, если вы хотите изменить сообщения последних трех фиксаций, или сообщение какой-то одной фиксации этой группы, то передайте как аргумент команде `git rebase -i` родителя последней фиксации, которую вы хотите изменить – `HEAD~2^` или `HEAD~3`. Может быть, проще будет запомнить `~3`, так как вы хотите изменить последние три фиксации; но не забывайте, что вы, в действительности, указываете четвертую фиксацию с конца – родителя последней фиксации, которую вы хотите изменить:

```
$ git rebase -i HEAD~3
```

Напомним, что это команда перебазирования – каждая фиксация, входящая в диапазон `HEAD~3..HEAD`, будет изменена вне зависимости от того, изменили вы сообщение или нет. Не включайте в такой диапазон фиксацию, которая уже была отправлена на центральный сервер – сделав это, вы можете запутать других разработчиков, предоставив вторую версию одних и тех же изменений.

Выполнение этой команды отобразит в вашем текстовом редакторе список фиксаций, в нашем случае, например, следующее:

```

pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

Rebase 710f0f8..a5f4a0d onto 710f0f8
#
Commands:
p, pick = use commit
r, reword = use commit, but edit the commit message
e, edit = use commit, but stop for amending
s, squash = use commit, but meld into previous commit
f, fixup = like "squash", but discard this commit's log message
x, exec = run command (the rest of the line) using shell
#
These lines can be re-ordered; they are executed from top to bottom.
#
If you remove a line here THAT COMMIT WILL BE LOST.
#
However, if you remove everything, the rebase will be aborted.
#
Note that empty commits are commented out

```

Важно отметить, что фиксации перечислены в порядке, противоположном порядку, который вы обычно видите при использовании команды `log`. Если вы выполните `log`, то увидите следующее:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

Обратите внимание на обратный порядок. Команда `rebase` в интерактивном режиме предоставит вам скрипт, который она будет выполнять. Она начнет с фиксации, которую вы указали в командной строке (`HEAD~3`) и повторит изменения, внесенные каждой из фиксаций, сверху вниз. Наверху отображается самая старая фиксация, а не самая новая, потому что она будет повторена первой.

Вам необходимо изменить скрипт так, чтобы он остановился на фиксации, которую вы хотите изменить. Для этого измените слово ‘`pick`’ на слово ‘`edit`’ напротив каждой из фиксаций, после которых скрипт должен остановиться. Например, для изменения сообщения только третьей фиксации, измените файл следующим образом:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Когда вы сохраните сообщение и выйдете из редактора, Git переместит вас к самой ранней фиксации из списка и вернет вас в командную строку со следующим сообщением:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... changed my name a bit
You can amend the commit now, with

 git commit --amend

Once you're satisfied with your changes, run

 git rebase --continue
```

Эти инструкции говорят вам в точности то, что нужно сделать. Введите

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

Измените сообщение фиксации и выйдите из редактора. Затем выполните

```
$ git rebase --continue
```

Эта команда автоматически применит две оставшиеся фиксации и завершится. Если вы измените ‘pick’ на ‘edit’ в других строках, то можете повторить эти шаги для соответствующих фиксаций. Каждый раз Git будет останавливаться, позволяя вам исправить фиксацию, и продолжит, когда вы закончите.

## Переупорядочивание фиксаций

Вы также можете использовать интерактивное перебазирование для переупорядочивания или полного удаления фиксаций. Если вы хотите удалить фиксацию “added cat-file” и изменить порядок, в котором были

внесены две оставшиеся, то вы можете изменить скрипт перебазирования с такого:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

на такой:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

Когда вы сохраните скрипт и выйдете из редактора, Git переместит вашу ветку на родителя этих фиксаций, применит 310154e, затем f7f3f6d и после этого остановится. Вы, фактически, изменили порядок этих фиксаций и полностью удалили фиксацию “added cat-file”.

## Объединение фиксаций

С помощью интерактивного режима команды `rebase` также можно объединить последовательность фиксаций в одну. Git добавляет полезные инструкции в сообщение скрипта перебазирования:

```
#
Commands:
p, pick = use commit
r, reword = use commit, but edit the commit message
e, edit = use commit, but stop for amending
s, squash = use commit, but meld into previous commit
f, fixup = like "squash", but discard this commit's log message
x, exec = run command (the rest of the line) using shell
#
These lines can be re-ordered; they are executed from top to bottom.
#
If you remove a line here THAT COMMIT WILL BE LOST.
#
However, if you remove everything, the rebase will be aborted.
#
Note that empty commits are commented out
```

Если вместо “pick” или “edit” вы укажете “squash”, Git применит изменения из текущей и предыдущей фиксаций и предложит вам объединить вместе сообщения фиксаций. Таким образом, если вы хотите из этих трех фиксаций сделать одну, вы должны изменить скрипт следующим образом:

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

Когда вы сохраните скрипт и выйдете из редактора, Git применит изменения всех трех фиксаций и затем вернет вас обратно в редактор, чтобы вы могли объединить сообщения фиксаций:

```
This is a combination of 3 commits.
The first commit's message is:
changed my name a bit

This is the 2nd commit message:

updated README formatting and added blame

This is the 3rd commit message:

added cat-file
```

После сохранения сообщения, вы получите одну фиксацию, содержащую изменения всех трех фиксаций, существовавших ранее.

## Разбиение фиксации

Разбиение фиксации отменяет ее и позволяет затем по частям индексировать и фиксировать изменения, создавая таким образом столько фиксаций, сколько вам нужно. Например, предположим, что вы хотите разбить среднюю фиксацию на три. Вместо одной фиксации “updated README formatting and added blame” вы хотите получить две разные: первую – “updated README formatting”, и вторую – “added blame”. Вы можете добиться этого, изменив в скрипте `rebase -i` инструкцию для разбиваемой фиксации на “edit”:

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Затем, когда скрипт вернет вас в командную строку, вам нужно будет отменить индексацию изменений этой фиксации, и создать несколько фиксаций на основе этих изменений. Когда вы сохраните скрипт и выйдете из редактора, Git переместится на родителя первой фиксации в вашем списке, применит первую фиксацию (f7f3f6d), применит вторую (310154e), и вернет вас в консоль. Здесь вы можете отменить фиксацию с помощью команды `git reset HEAD^`, которая, фактически, отменит эту фиксацию и удалит из индекса измененные файлы. Теперь вы можете добавлять в индекс и фиксировать файлы, пока не создадите требуемые фиксации, а после этого выполнить команду `git rebase --continue`:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git применит последнюю фиксацию (a5f4a0d) из скрипта, и ваша история примет следующий вид:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

И снова, при этом изменились SHA-1 хеши всех фиксаций в вашем списке, поэтому убедитесь, что ни одна фиксация из этого списка ранее не была отправлена в общий репозиторий.

## Продвинутый инструмент: `filter-branch`

Существует еще один способ изменения истории, который вы можете использовать при необходимости изменить большое количество

фиксаций каким-то программируемым способом – например, изменить глобально ваш адрес электронной почты или удалить файл из всех фиксаций. Для этого существует команда `filter-branch`, и она может изменять большие периоды вашей истории, поэтому вы, возможно, не должны ее использовать кроме тех случаев, когда ваш проект еще не стал публичным и другие люди еще не имеют наработок, основанных на фиксациях, которые вы собираетесь изменить. Однако, эта команда может быть очень полезной. Далее вы ознакомитесь с несколькими обычными вариантами использованиями этой команды, таким образом, вы сможете получить представление о том, на что она способна.

## УДАЛЕНИЕ ФАЙЛА ИЗ КАЖДОЙ ФИКСАЦИИ

Такое случается довольно часто. Кто-нибудь случайно зафиксировал огромный бинарный файл, неосмотрительно выполнив `git add ..` и вы хотите отовсюду его удалить. Возможно, вы случайно зафиксировали файл, содержащий пароль, а теперь хотите сделать ваш проект общедоступным. В общем, утилиту `filter-branch` вы, вероятно, захотите использовать, чтобы привести к нужному виду всю вашу историю. Для удаления файла `passwords.txt` из всей вашей истории вы можете использовать опцию `--tree-filter` команды `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

Опция `--tree-filter` выполняет указанную команду после переключения на каждую фиксацию и затем повторно фиксирует результаты. В данном примере, вы удаляете файл `passwords.txt` из каждого снимка вне зависимости от того, существует он или нет. Если вы хотите удалить все случайно зафиксированные резервные копии файлов, созданные текстовым редактором, то вы можете выполнить нечто подобное `git filter-branch --tree-filter 'rm -f *~' HEAD`.

Вы можете посмотреть, как Git изменит деревья и фиксации, а затем уже переместить указатель ветки. Как правило, хорошим подходом будет выполнение всех этих действий в тестовой ветке и, после проверки полученных результатов, установка на нее указателя основной ветки. Для выполнения `filter-branch` на всех ваших ветках, вы можете передать команде опцию `--all`.

## УСТАНОВКА ПОДДИРЕКТОРИИ КОРНЕВОЙ ДИРЕКТОРИЕЙ ПРОЕКТА

Предположим, вы выполнили импорт из другой системы управления исходным кодом и получили в результате поддиректории, которые не имеют никакого смысла (`trunk`, `tags` и так далее). Если вы хотите сделать поддиректорию `trunk` корневой директорией для каждой фиксации, команда `filter-branch` может помочь вам в этом:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Теперь вашей новой корневой директорией проекта будет являться поддиректория `trunk`. Git также автоматически удалит фиксации, которые не затрагивали этой поддиректории.

## ГЛОБАЛЬНОЕ ИЗМЕНЕНИЕ АДРЕСА ЭЛЕКТРОННОЙ ПОЧТЫ

Еще один типичный случай возникает, когда вы забыли выполнить `git config` для настройки своего имени и адреса электронной почты перед началом работы, или, возможно, хотите открыть исходные коды вашего рабочего проекта и изменить везде адрес вашей рабочей электронной почты на персональный. В любом случае вы можете изменить адрес электронной почты сразу в нескольких фиксациях с помощью команды `filter-branch`. Вы должны быть осторожны, чтобы изменить только свои адреса электронной почты, для этого используйте опцию `--commit-filter`:

```
$ git filter-branch --commit-filter '
 if ["$GIT_AUTHOR_EMAIL" = "schacon@localhost"];
 then
 GIT_AUTHOR_NAME="Scott Chacon";
 GIT_AUTHOR_EMAIL="schacon@example.com";
 git commit-tree "$@";
 else
 git commit-tree "$@";
 fi' HEAD
```

Эта команда пройдет по всем фиксациям и установит в них ваш новый адрес. Так как фиксации содержат значения SHA-1 хешей их родителей, эта команда изменяет SHA-1 хеш каждой фиксации в

вашей истории, а не только тех, которые соответствовали адресам электронной почты.

## Раскрытие тайн reset

Перед тем, как перейти к более специализированным утилитам, давайте поговорим о `reset` и `checkout`. Эти команды кажутся самыми непонятными из всех, которые есть в Git, когда вы в первый раз сталкиваетесь с ними. Они делают так много, что попытки по-настоящему их понять и правильно использовать кажутся безнадежными. Для того, чтобы всё же достичь этого, мы советуем воспользоваться простой аналогией.

### Три дерева

Разобраться с командами `reset` и `checkout` будет проще, если считать, что Git управляет содержимым трех различных деревьев. Здесь под “деревом” мы понимаем “набор файлов”, а не специальную структуру данных. (В некоторых случаях индекс ведет себя не совсем так, как дерево, но для наших текущих целей его проще представлять именно таким.)

В своих обычных операциях Git управляет тремя деревьями:

Дерево	Назначение
HEAD	Снимок последнего коммита, родитель следующего
Индекс	Снимок следующего намеченного коммита
Рабочий Каталог	Песочница

#### HEAD

HEAD – это указатель на текущую ветку, которая, в свою очередь, является указателем на последний коммит, сделанный в этой ветке. Это значит, что HEAD будет родителем следующего созданного коммита. Как правило, самое простое считать HEAD снимком **вашего последнего коммита**.

На самом деле, довольно легко увидеть, что представляет из себя этот снимок. Ниже приведен пример получения содержимого каталога и контрольных сумм для каждого файла в HEAD:

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon 1301511835 -0700
committer Scott Chacon 1301511835 -0700

initial commit

$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152... README
100644 blob 8f94139338f9404f2... Rakefile
040000 tree 99f1a6d12cb4b6f19... lib
```

Команды `cat-file` и `ls-tree` являются “служебными” (*plumbing*) командами, которые используются внутри системы и не требуются при ежедневной работе, но они помогают нам разобраться, что же происходит на самом деле.

## ИНДЕКС

Индекс – это ваш **следующий намеченный коммит**. Мы также упоминали это понятие как “область подготовленных изменений” Git – то, что Git просматривает, когда вы выполняете `git commit`.

Git заполняет индекс списком изначального содержимого всех файлов, выгруженных в последний раз в ваш рабочий каталог. Затем вы заменяете некоторые из таких файлов их новыми версиями и команда ‘`git commit`’ преобразует изменения в дерево для нового коммита.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0 README
100644 8f94139338f9404f26296befa88755fc2598c289 0 Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0 lib/simplegit.rb
```

Повторим, здесь мы используем служебную команду `ls-files`, которая показывает вам, как выглядит сейчас ваш индекс.

Технически, индекс не является древовидной структурой, на самом деле, он реализован как сжатый список (*flattened manifest*) – но для наших целей такого представления будет достаточно.

## РАБОЧИЙ КАТАЛОГ

Наконец, у вас есть рабочий каталог. Два других дерева сохраняют свое содержимое эффективным, но неудобным способом внутри каталога .git. Рабочий Каталог распаковывает их в настоящие файлы, что упрощает для вас их редактирование. Считайте Рабочий Каталог **песочницей**, где вы можете опробовать изменения перед их коммитом в индекс (область подготовленных изменений) и затем в историю.

```
$ tree
.
├── README
├── Rakefile
└── lib
 └── simplegit.rb

1 directory, 3 files
```

## Технологический процесс

Основное предназначение Git – это сохранение снимков последовательно улучшающихся состояний вашего проекта, путем управления этими тремя деревьями.

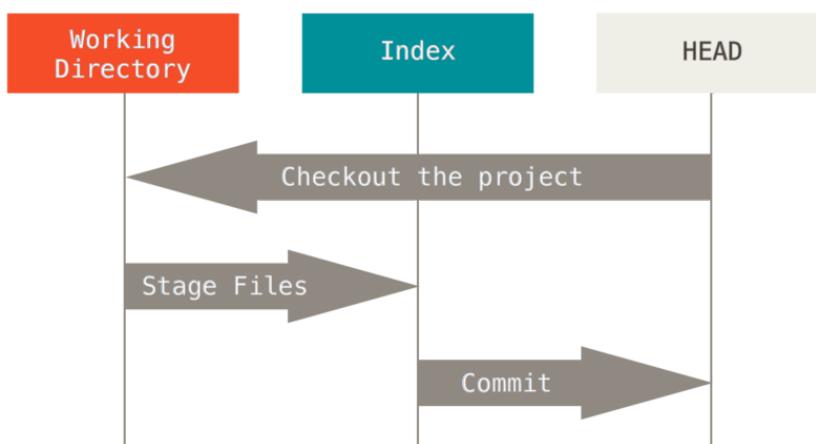
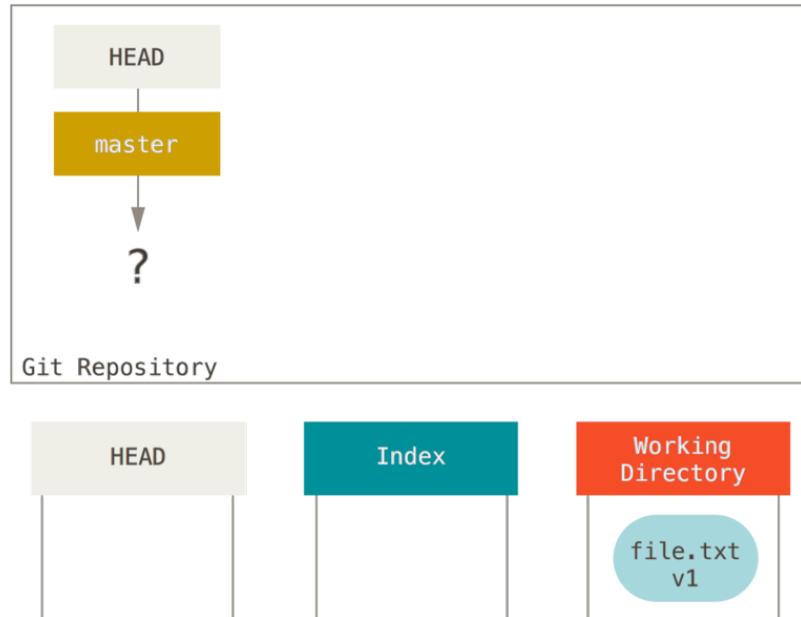


FIGURE 7-2

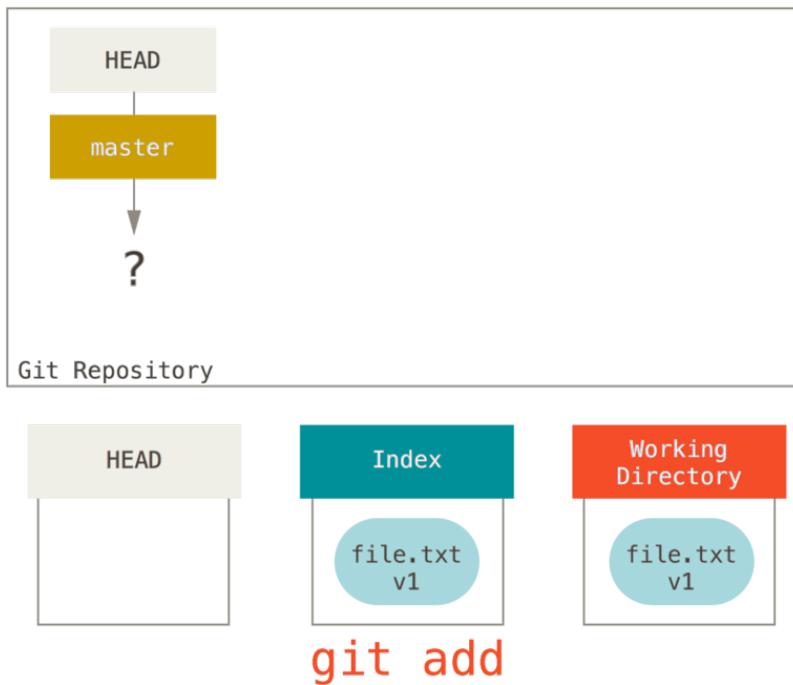
Давайте рассмотрим этот процесс: пусть вы перешли в новый каталог, содержащий один файл. Данную версию этого файла будем называть **v1** и изображать голубым цветом. Выполним команду `git init`, которая создаст Git-репозиторий, у которого ссылка `HEAD` будет указывать на еще несуществующую ветку (`master` пока не существует).

FIGURE 7-3

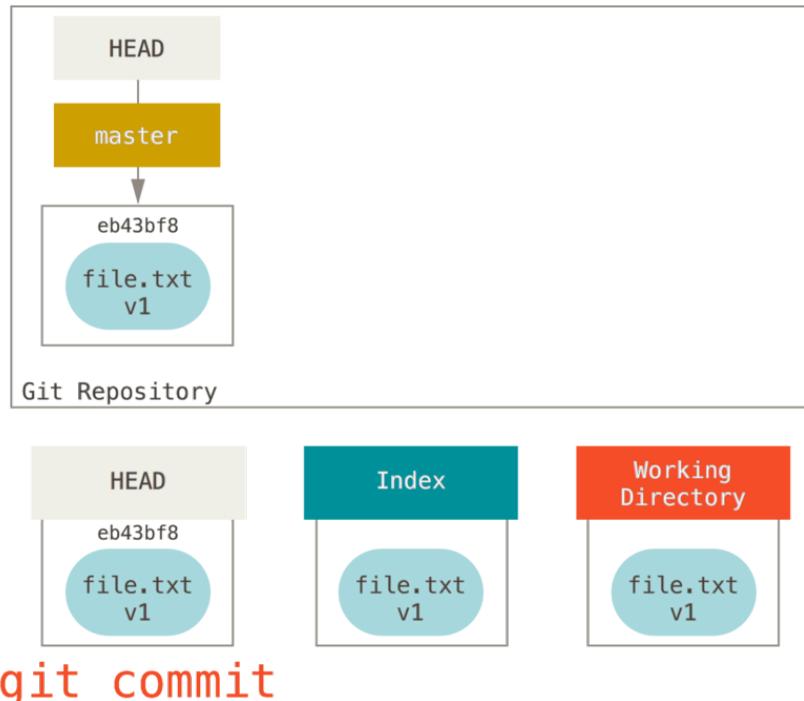


На данном этапе только дерево Рабочего Каталога содержит данные.

Теперь мы хотим закоммитить этот файл, поэтому мы используем `git add` для копирования содержимого Рабочего Каталога в Индекс.

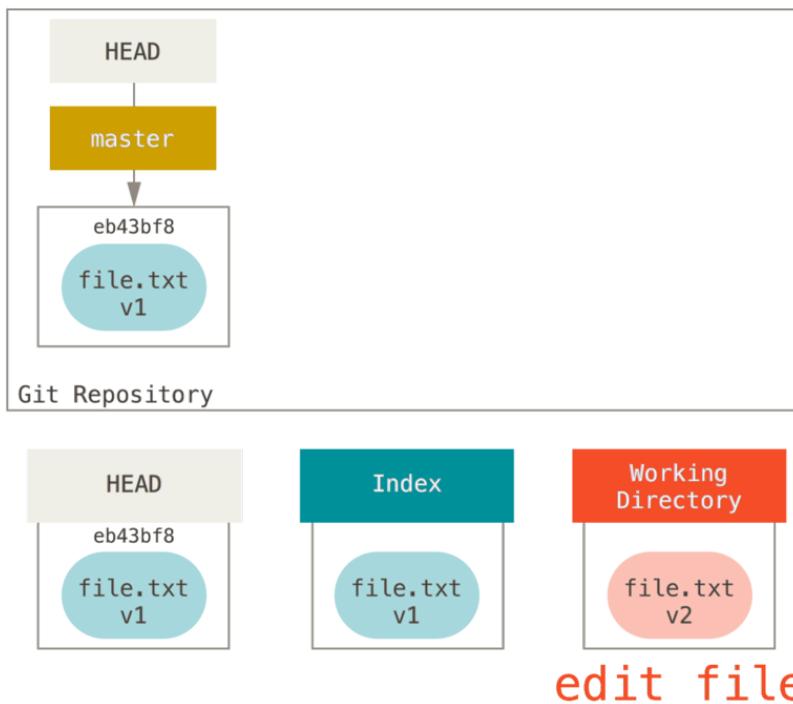
**FIGURE 7-4**

Затем, мы выполняем команду `git commit`, которая сохраняет содержимое Индекса как неизменяемый снимок, создает объект коммита, который указывает на этот снимок, и обновляет `master` так, чтобы он тоже указывал на этот коммит.

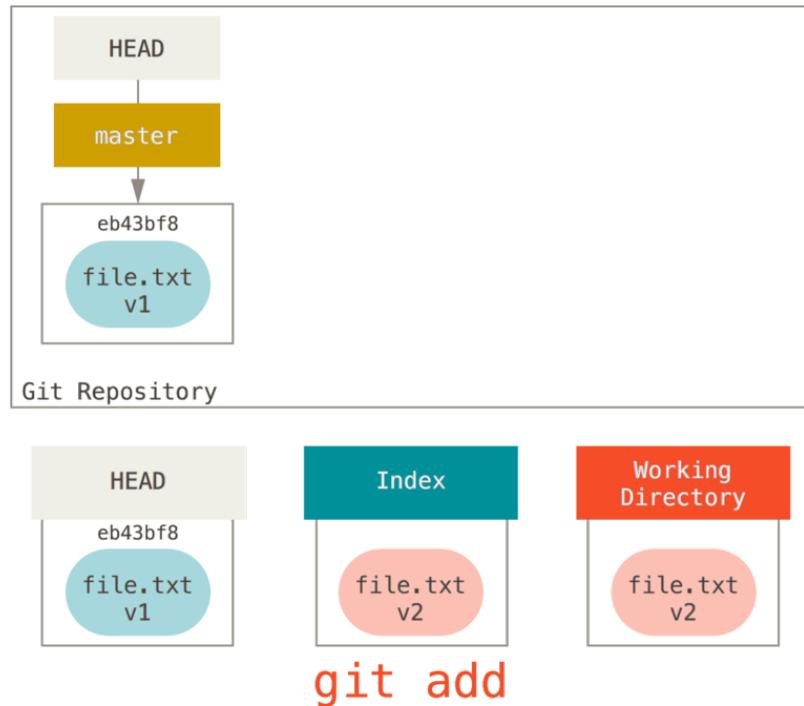
**FIGURE 7-5**

Если сейчас выполнить `git status`, то мы не увидим никаких изменений, так как все три дерева одинаковые.

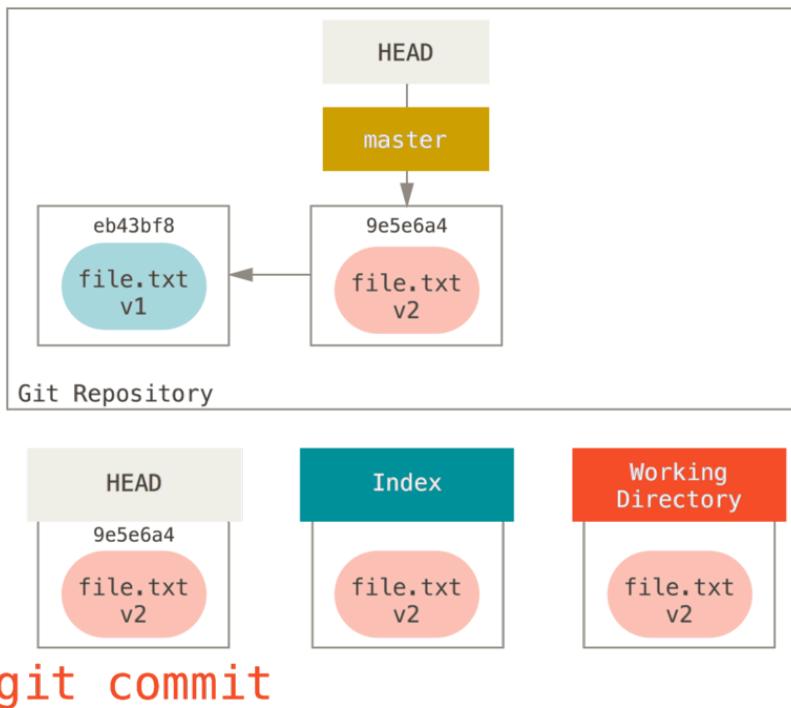
Теперь мы хотим внести изменения в файл и закомитить его. Мы пройдем через всё ту же процедуру; сначала мы отредактируем файл в нашем рабочем каталоге. Давайте называть эту версию файла **v2** и обозначать красным цветом.

**FIGURE 7-6**

Если сейчас мы выполним `git status`, то увидим, что файл выделен красным в разделе “Изменения, не подготовленные к коммиту”, так как его представления в Индексе и Рабочем Каталоге различны. Затем мы выполним `git add` для этого файла, чтобы поместить его в Индекс.

**FIGURE 7-7**

Если сейчас мы выполним `git status`, то увидим, что этот файл выделен зеленым цветом в разделе “Изменения, которые будут закоммичены”, так как Индекс и HEAD различны – то есть, наш следующий намеченный коммит сейчас отличается от нашего последнего коммита. Наконец, мы выполним `git commit`, чтобы окончательно совершить коммит.

**FIGURE 7-8**

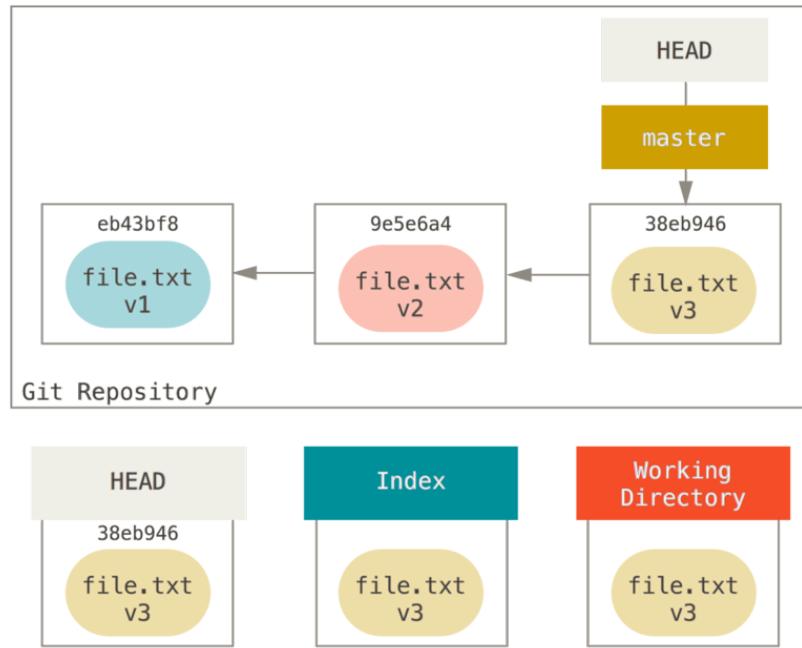
Сейчас команда `git status` не показывает ничего, так как снова все три дерева одинаковые.

Переключение веток и клонирование проходят через похожий процесс. Когда вы переключаетесь (`checkout`) на ветку, **HEAD** начинает также указывать на новую ветку, ваш **Индекс** замещается снимком коммита этой ветки, и затем содержимое **Индекса** копируется в ваш **Рабочий Каталог**.

## Назначение `reset`

Команда `reset` становится более понятной, если рассмотреть ее с учетом вышеизложенного.

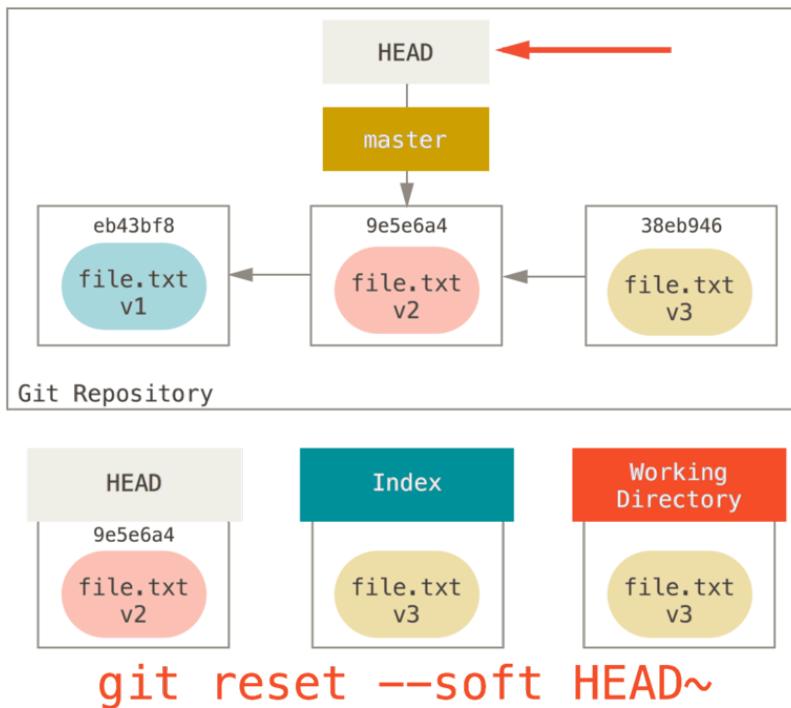
В следующих примерах предположим, что мы снова изменили файл `file.txt` и закоммитили его в третий раз. Так что наша история теперь выглядит так:

**FIGURE 7-9**

Давайте теперь внимательно проследим, что именно происходит при вызове `reset`. Эта команда простым и предсказуемым способом управляет тремя деревьями, существующими в Git. Она выполняет три основных операции.

#### ШАГ 1: ПЕРЕМЕЩЕНИЕ HEAD

Первое, что сделает `reset` – переместит то, на что указывает HEAD. Обратите внимание, изменяется не сам HEAD (что происходит при выполнении команды `checkout`); `reset` перемещает ветку, на которую указывает HEAD. Таким образом, если HEAD указывает на ветку `master` (то есть вы сейчас работаете с веткой `master`), выполнение команды `git reset 9e5e6a4` сделает так, что `master` будет указывать на `9e5e6a4`.

**FIGURE 7-10**

Не важно с какими опциями вы вызвали команду `reset` с указанием коммита (reset также можно вызывать с указанием пути), она всегда будет пытаться сперва сделать данный шаг. При вызове `reset --soft` на этом выполнение команды и остановится.

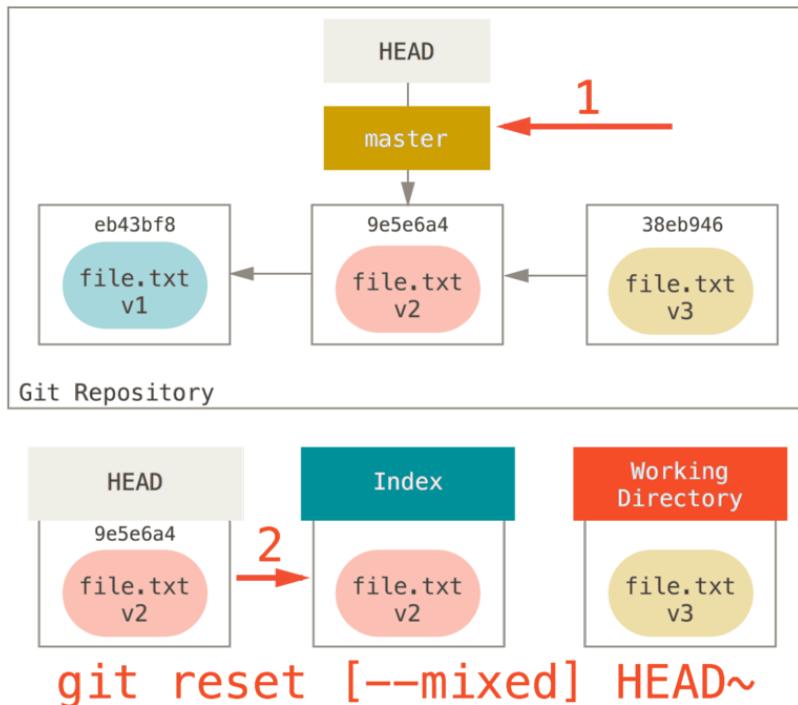
Теперь взгляните на диаграмму и постараитесь разобраться, что случилось: фактически была отменена последняя команда `git commit`. Когда вы выполняете `git commit`, Git создает новый коммит и перемещает на него ветку, на которую указывает HEAD. Если вы выполняете `reset` на `HEAD~` (родителя HEAD), то вы перемещаете ветку туда, где она была раньше, не изменяя при этом ни Индекс, ни Рабочий Каталог. Вы можете обновить Индекс и снова выполнить `git commit`, таким образом добиваясь того же, что делает команда `git commit --amend` (смотрите “Изменение последней фиксации”).

## ШАГ 2: ОБНОВЛЕНИЕ ИНДЕКСА (--MIXED)

Заметьте, если сейчас вы выполните `git status`, то увидите отмеченные зеленым цветом изменения между Индексом и новым HEAD.

Следующим, что сделает `reset`, будет обновление Индекса содержимым того снимка, на который указывает HEAD.

FIGURE 7-11



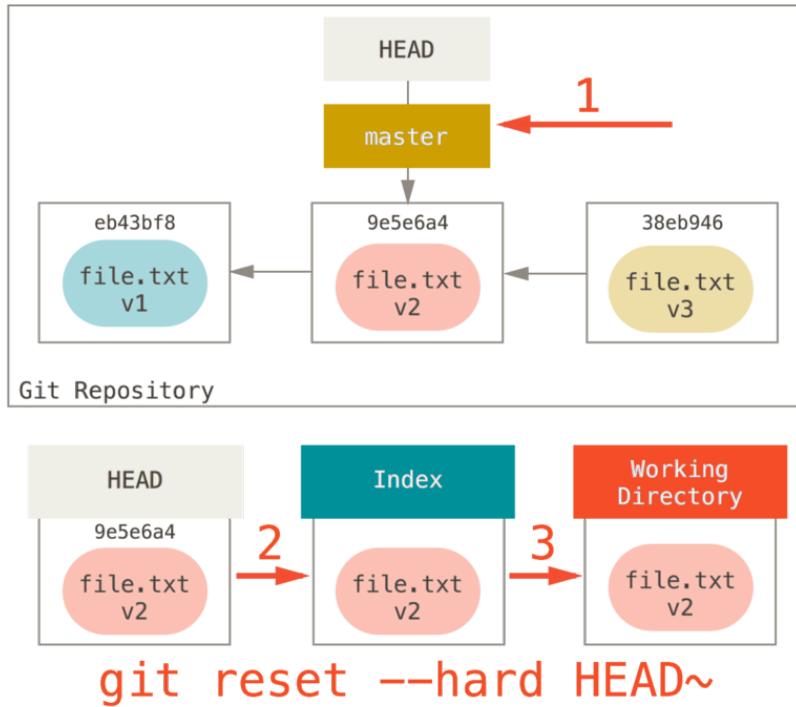
Если вы указали опцию `--mixed`, выполнение `reset` остановится на этом шаге. Такое поведение также используется по умолчанию, поэтому если вы не указали совсем никаких опций (в нашем случае `git reset HEAD~`), выполнение команды также остановится на этот шаге.

Снова взгляните на диаграмму и постараитесь разобраться, что произошло: отменен не только ваш последний commit, но также и **добавление в индекс** всех файлов. Вы откатились назад до момента выполнения команд `git add` и `git commit`.

### ШАГ 3: ОБНОВЛЕНИЕ РАБОЧЕГО КАТАЛОГА (--HARD)

Третье, что сделает `reset` – это приведение вашего Рабочего Каталога к тому же виду, что и Индекс. Если вы используете опцию `--hard`, то выполнение команды будет продолжено до этого шага.

FIGURE 7-12



Давайте разберемся, что сейчас случилось. Вы отменили ваш последний коммит, результаты выполнения команд `git add` и `git commit`, а также **все** изменения, которые вы сделали в рабочем каталоге.

Важно отметить, что только указание этого флага (`--hard`) делает команду `reset` опасной, это один из немногих случаев, когда Git действительно удаляет данные. Все остальные вызовы `reset` легко отменить, но при указании опции `--hard` команда принудительно перезаписывает файлы в Рабочем Каталоге. В данном конкретном случае, версия **v3** нашего файла всё еще остается в коммите внутри базы данных Git и мы можем вернуть ее, просматривая наш `reflog`, но

если вы не коммитили эту версию, Git перезапишет файл и ее уже нельзя будет восстановить.

## РЕЗЮМЕ

Команда `reset` в заранее определенном порядке перезаписывает три дерева Git, останавливаясь тогда, когда вы ей скажите:

1. Перемещает ветку, на которую указывает HEAD (*останавливается на этом, если указана опция `--soft`*)
2. Делает Индекс таким же как и HEAD (*останавливается на этом, если не указана опция `--hard`*)
3. Делает Рабочий Каталог таким же как и Индекс.

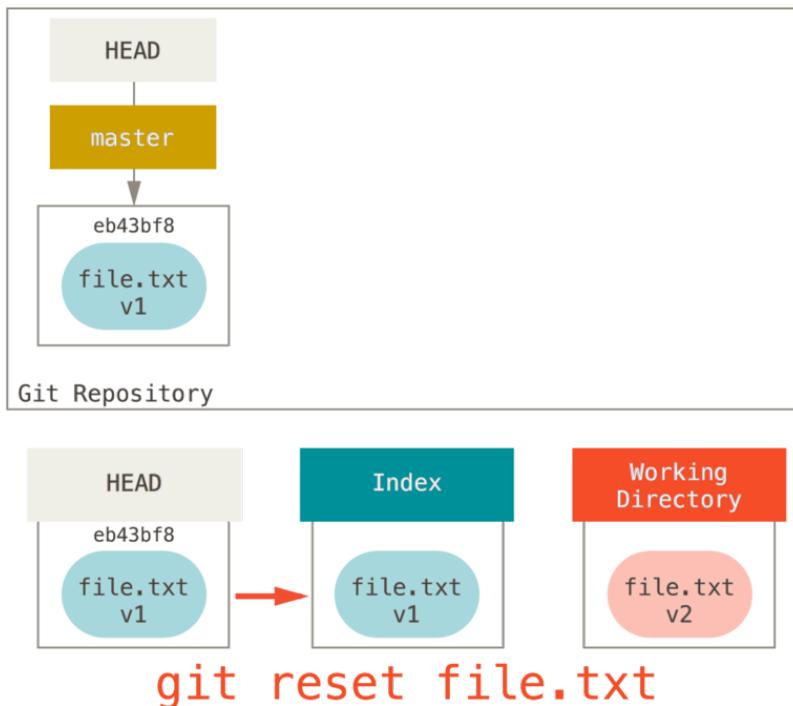
### **Reset с указанием пути**

Основной форме команды `reset` (без опций `--soft` и `--hard`) вы также можете передавать путь, с которым она будет оперировать. В этом случае, `reset` пропустит первый шаг, а на остальных будет работать только с указанным файлом или набором файлов. Первый шаг пропускается, так как HEAD является указателем и не может ссылаться частично на один коммит, а частично на другой. Но Индекс и Рабочий Каталог *могут* быть изменены частично, поэтому `reset` выполняет шаги 2 и 3.

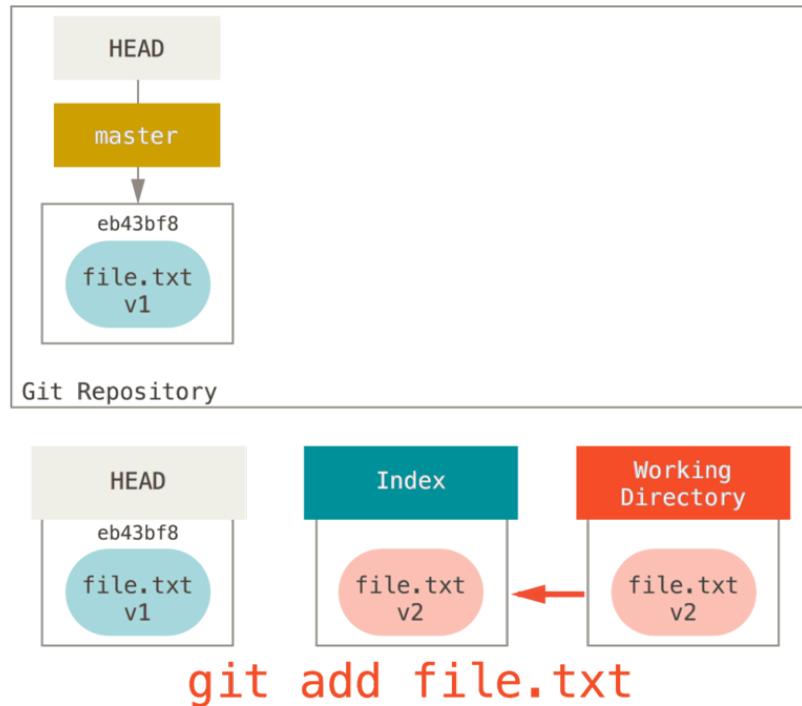
Итак, предположим вы выполнили команду `git reset file.txt`. Эта форма записи (так как вы не указали ни SHA-1 коммита, ни ветку, ни опций `--soft` или `--hard`) является сокращением для `git reset --mixed HEAD file.txt`, которая:

1. Перемещает ветку, на которую указывает HEAD (*будет пропущено*)
2. Делает Индекс таким же как и HEAD (*остановится здесь*)

То есть, фактически, она копирует файл `file.txt` из HEAD в Индекс.

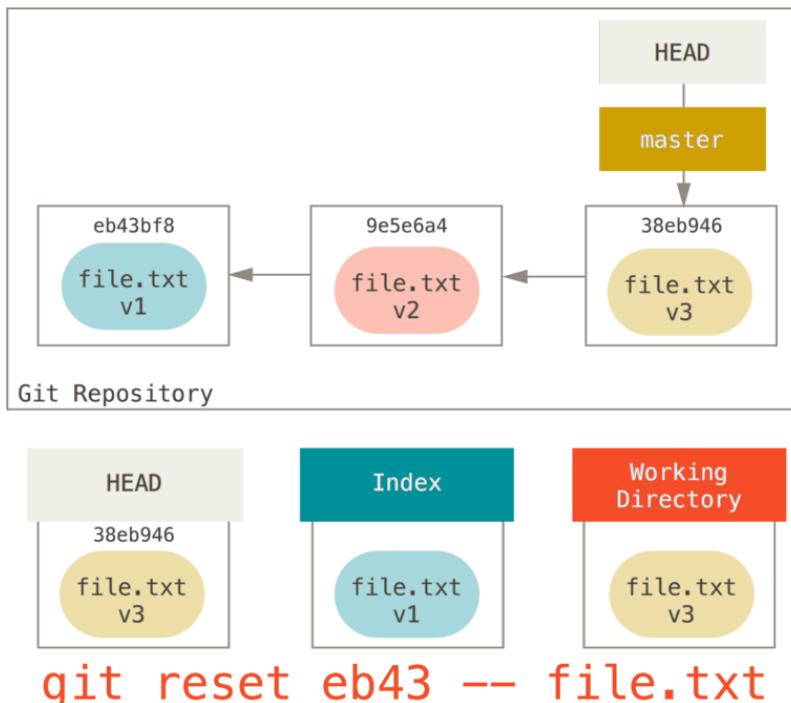
**FIGURE 7-13**

Это создает эффект *отмены индексации* файла. Если вы посмотрите на диаграммы этой команды и команды `git add`, то увидите, что их действия прямо противоположные.

**FIGURE 7-14**

Именно поэтому в выводе `git status` предлагается использовать такую команду для отмены индексации файла. (Смотрите подробности в “Отмена подготовки файла”.)

Мы легко можем заставить Git “брать данные не из HEAD”, указав коммит, из которого нужно взять версию этого файла. Для этого мы должны выполнить следующее `git reset eb43bf file.txt`.

**FIGURE 7-15**

Можно считать, что, фактически, мы в Рабочем Каталоге вернули содержимое файла к версии **v1**, выполнили для него `git add`, а затем вернули содержимое обратно к версии **v3** (в действительности все эти шаги не выполняются). Если сейчас мы выполним `git commit`, то будут сохранены изменения, которые возвращают файл к версии **v1**, но при этом файл в Рабочем Каталоге никогда не возвращается к такой версии.

Заметим, что как и команде `git add, reset` можно указывать опцию `--patch` для отмены индексации части содержимого. Таким способом вы можете избирательно отменять индексацию или откатывать изменения.

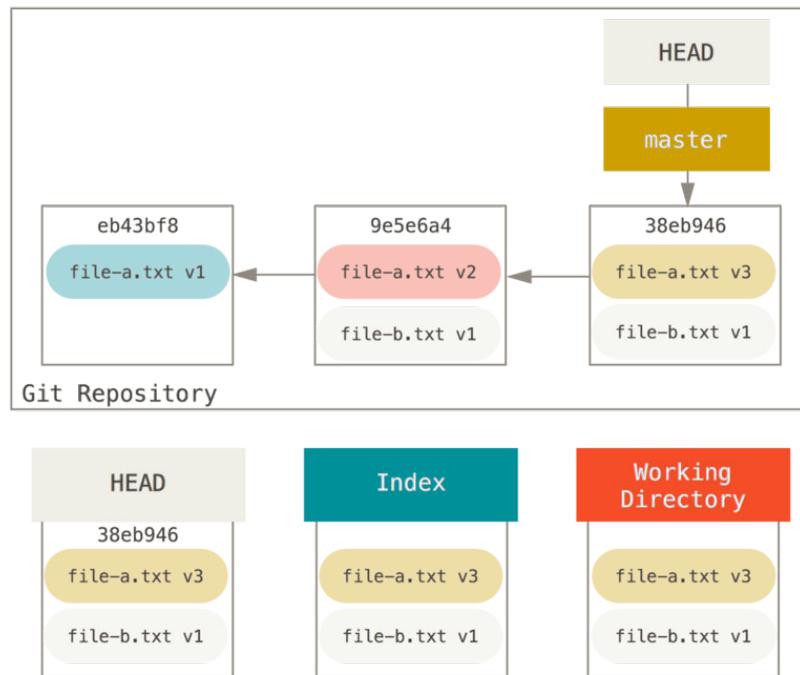
## Слияние коммитов

Давайте посмотрим, как, используя вышеизложенное, сделать кое-что интересное – слияние коммитов.

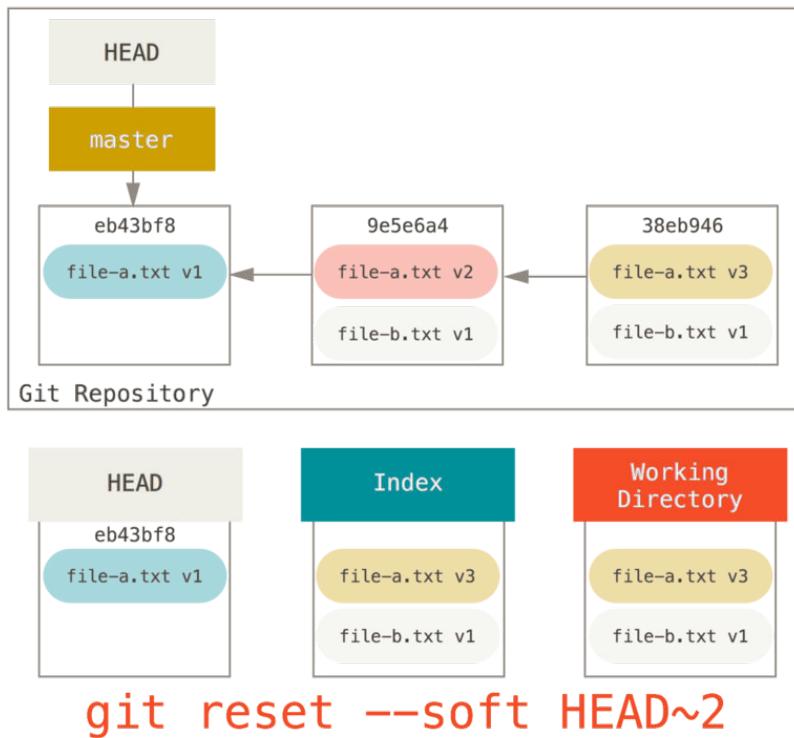
Допустим, у вас есть последовательность коммитов с сообщениями вида “ups.”, “В работе” и “позабыл этот файл”. Вы можете использовать `reset` для того, чтобы просто и быстро слить их в один. (В “Объединение фиксаций” представлен другой способ сделать то же самое, но в данном примере проще воспользоваться `reset`.)

Предположим, у вас есть проект, в котором первый коммит содержит один файл, второй коммит добавляет новый файл и изменяет первый, а третий коммит снова изменяет первый файл. Второй коммит был сделан в процессе работы и вы хотите слить его со следующим.

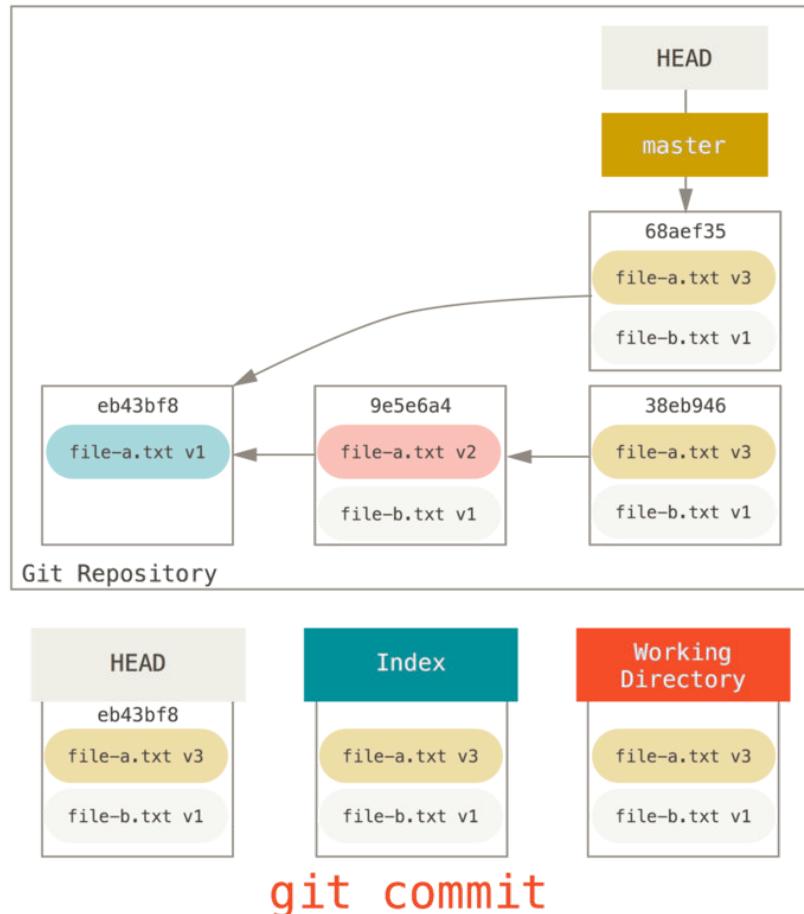
FIGURE 7-16



Вы можете выполнить `git reset --soft HEAD~2`, чтобы вернуть ветку HEAD на какой-то из предыдущих коммитов (на первый коммит, который вы хотите оставить):

**FIGURE 7-17**

Затем просто снова выполните `git commit`:

**FIGURE 7-18**

Теперь вы можете видеть, что ваша “доступимая” история (история, которую вы впоследствии отправите на сервер), сейчас выглядит так – у вас есть первый коммит с файлом `file-a.txt` версии **v1**, и второй, который изменяет файл `file-a.txt` до версии **v3** и добавляет `file-b.txt`. Коммита, который содержал файл версии **v2** не осталось в истории.

### Сравнение с `checkout`

Наконец, вы можете задаться вопросом, в чем же состоит отличие между `checkout` и `reset`. Как и `reset`, команда `checkout` управляет

тремя деревьями Git, и также ее поведение зависит от того указали ли вы путь до файла или нет.

## БЕЗ УКАЗАНИЯ ПУТИ

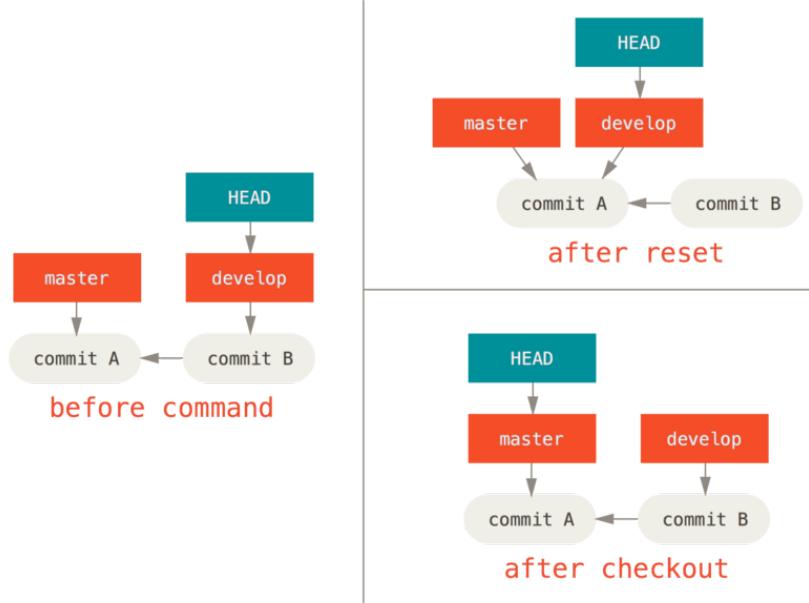
Команда `git checkout [branch]` очень похожа на `git reset --hard [branch]`, в процессе их выполнения все три дерева изменяются так, чтобы выглядеть как `[branch]`. Но между этими командами есть два важных отличия.

Во-первых, в отличии от `reset --hard`, команда `checkout` бережно относится к рабочему каталогу, и проверяет, что она не трогает файлы, в которых есть изменения. В действительности, эта команда поступает немного умнее – она пытается выполнить в Рабочем Каталоге простые слияния так, чтобы все файлы, которые вы *не* изменили, были обновлены. С другой стороны, команда `reset --hard` просто заменяет всё целиком, не выполняя проверок.

Второе важно отличие заключается в том, как эти команды обновляют HEAD. В то время как `reset` перемещает ветку, на которую указывает HEAD, команда `checkout` перемещает сам HEAD так, чтобы он указывал на другую ветку.

Например, пусть у нас есть ветки `master` и `develop`, которые указывают на разные коммиты и мы сейчас находимся на ветке `develop` (то есть HEAD указывает на нее). Если мы выполним `git reset master`, сама ветка `develop` станет ссылаться на тот же коммит, что и `master`. Если мы выполним `git checkout master`, то `develop` не изменится, но изменится HEAD. Он станет указывать на `master`.

Итак, в обоих случаях мы перемещаем HEAD на коммит A, но важное отличие состоит в том, *как* мы это делаем. Команда `reset` переместит также и ветку, на которую указывает HEAD, а `checkout` перемещает только сам HEAD.

**FIGURE 7-19**

### С УКАЗАНИЕМ ПУТИ

Другой способ выполнить `checkout` состоит в том, чтобы указать путь до файла. В этом случае, как и для команды `reset`, `HEAD` не перемещается. Эта команда как и `git reset [branch] file` обновляет файл в индексе версией из коммита, но дополнительно она обновляет и файл в рабочем каталоге. То же самое сделала бы команда `git reset --hard [branch] file` (если бы `reset` можно было бы так запускать) – это небезопасно для рабочего каталога и не перемещает `HEAD`.

Также как `git reset` и `git add`, команда `checkout` принимает опцию `--patch` для того, чтобы позволить вам избирательно откатить изменения содержимого файла по частям.

### Заключение

Надеюсь, вы разобрались с командой `reset` и можете ее спокойно использовать. Но, возможно, вы всё еще немного путаетесь, чем именно она отличается от `checkout`, и не запомнили всех правил, используемых в различных вариантах вызова.

Ниже приведена памятка того, как эти команды воздействуют на каждое из деревьев. В столбце “HEAD” указывается “REF” если эта команда перемещает ссылку (ветку), на которую HEAD указывает, и “HEAD” если перемещается только сам HEAD. Обратите особое внимание на столбец “Сохранность РК?” – если в нем указано **NO**, то хорошоенько подумайте прежде чем выполнить эту команду.

	<b>HEAD</b>	<b>Индекс</b>	<b>Рабочий Каталог</b>	<b>Сохранность РК?</b>
<b>На уровне коммитов (без указания путей)</b>				
<code>reset --soft [commit]</code>	REF	NO	NO	YES
<code>reset [commit]</code>	REF	YES	NO	YES
<code>reset --hard [commit]</code>	REF	YES	YES	<b>NO</b>
<code>checkout [commit]</code>	HEAD	YES	YES	YES
<b>На уровне файлов (с указанием путей)</b>				
<code>reset (commit) [file]</code>	NO	YES	NO	YES
<code>checkout (commit) [file]</code>	NO	YES	YES	<b>NO</b>

## Продвинутое слияние

Обычно выполнять слияния в Git довольно легко. Git упрощает повторные слияния с одной и той же веткой, таким образом, позволяя вам иметь очень долго живущую ветку, и вы можете сохранять ее всё это время в актуальном состоянии, часто разрешая маленькие конфликты, а не доводить дело до одного большого конфликта по завершению всех изменений.

Однако, иногда все же будут возникать сложные конфликты. В отличии от других систем управления версиями, Git не пытается быть слишком умным при разрешении конфликтов слияния. Философия Git заключается в том, чтобы быть умным, когда слияние разрешается однозначно, но если возникает конфликт, он не пытается сумничать и разрешить его автоматически. Поэтому, если вы слишком долго откладываете слияние двух быстрорастущих веток, вы можете столкнуться с некоторыми проблемами.

В этом разделе мы рассмотрим некоторые из возможных проблем и инструменты, которые предоставляет Git, чтобы помочь вам справиться с этими более сложными ситуациями. Мы также рассмотрим некоторые другие нестандартные типы слияний, которые вы можете выполнять, и вы узнаете как можно откатить уже выполненные слияния.

## Конфликты слияния

Мы рассказали некоторые основы разрешения конфликтов слияния в “Основные конфликты слияния”, для работы с более сложными конфликтами Git предоставляет несколько инструментов, которые помогут вам понять, что произошло и как лучше обойтись с конфликтом.

Во-первых, если есть возможность, перед слиянием, в котором может возникнуть конфликт, позаботьтесь о том, чтобы ваша рабочая директория была без локальных изменений. Если у вас есть несохраненные наработки, либо приберегите их, либо сохраните их во временной ветке. Таким образом, вы сможете легко отменить **любые** изменения, которые сделаете в рабочей директории. Если при выполнении слияния вы не сохраните сделанные в рабочей директории изменения, то некоторые из описанных ниже приемов могут привести к утрате этих наработок.

Давайте рассмотрим очень простой пример. Допустим, у нас есть файл с исходниками на Ruby, выводящими на экран строку *hello world*.

```
#!/usr/bin/env ruby

def hello
 puts 'hello world'
end

hello()
```

В нашем репозитории, мы создадим новую ветку по имени `whitespace` и выполним замену всех окончаний строк в стиле Unix на окончания строк в стиле DOS. Фактически, изменения будут внесены в каждую строку, но изменятся только пробельные символы. Затем мы заменим строку “*hello world*” на “*hello mundo*”.

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'
```

```
$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -w
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

def hello
- puts 'hello world'
+ puts 'hello mundo'^M
end

hello()

$ git commit -am 'hello mundo change'
[whitespace 6d338d2] hello mundo change
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Теперь мы переключимся обратно на ветку master и добавим к функции некоторую документацию.

```
$ git checkout master
Switched to branch 'master'

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
def hello
 puts 'hello world'
end

$ git commit -am 'document the function'
```

```
[master bcc6336] document the function
 1 file changed, 1 insertion(+)
```

Теперь мы попытаемся слить в текущую ветку ветку whitespace и в результате получим конфликты, так как изменились пробельные символы.

```
$ git merge whitespace
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

## ПРЕРЫВАНИЕ СЛИЯНИЯ

В данный момент у нас есть несколько вариантов дальнейших действий. Во-первых, давайте рассмотрим как выйти из этой ситуации. Если вы, возможно, не были готовы к конфликтам и на самом деле не хотите связываться с ними, вы можете просто отменить попытку слияния, используя команду `git merge --abort`.

```
$ git status -sb
master
UU hello.rb

$ git merge --abort

$ git status -sb
master
```

Эта команда пытается откатить ваше состояние до того, что было до запуска слияния. Завершиться неудачно она может только в случаях если перед запуском слияния у вас были не припрятанные, не зафиксированные изменения в рабочей директории, во всех остальных случаях все будет хорошо.

Если по каким-то причинам вы обнаружили себя в ужасном состоянии и хотите просто начать все сначала, вы можете также выполнить `git reset --hard HEAD` (либо вместо HEAD указав то, куда вы хотите откатиться). Но помните, что это откатит все изменения в рабочей директории, поэтому удостоверьтесь, что никакие из них вам не нужны.

## ИГНОРИРОВАНИЕ ПРОБЕЛЬНЫХ СИМВОЛОВ

В данном конкретном случае конфликты связаны с пробельными символами. Мы знаем это, так как это простой пример, но в реальных ситуациях это также легко определить при изучении конфликта, так как каждая строка в нем будет удалена и добавлена снова. По умолчанию Git считает все эти строки измененными и поэтому не может слить файлы.

Стратегии слияния, используемой по умолчанию, можно передать аргументы, и некоторые из них предназначены для соответствующей настройки игнорирования изменений пробельных символов. Если вы видите, что множество конфликтов слияния вызваны пробельными символами, то вы можете прервать слияние и запустить его снова, но на этот раз с опцией `-Xignore-all-space` или `-Xignore-space-change`. Первая опция игнорирует изменения в любом **количестве** существующих пробельных символов, вторая игнорирует вообще все изменения пробельных символов.

```
$ git merge -Xignore-all-space whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
hello.rb | 2 ++
1 file changed, 1 insertion(+), 1 deletion(-)
```

Поскольку в этом примере реальные изменения файлов не конфликтуют, то при игнорировании изменений пробельных символов все сольется хорошо.

Это значительно облегчает жизнь, если кто-то в вашей команде любит временами заменять все пробелы на табуляции или наоборот.

## РУЧНОЕ СЛИЯНИЕ ФАЙЛОВ

Хотя Git довольно хорошо обрабатывает пробельные символы, с другими типами изменений он не может справиться автоматически, но существуют другие варианты исправления. Например, представим, что Git не умеет обрабатывать изменения пробельных символов и нам нужно сделать это вручную.

То что нам действительно нужно – это перед выполнением самого слияния прогнать сливаляемый файл через программу `dos2unix`. Как мы будем делать это?

Во-первых, мы перейдем в состояние конфликта слияния. Затем нам необходимо получить копии *нашей* версии файла, *их* версии

файла (из ветки, которую мысливаем) и *общей* версии (от которой ответвились первые две). Затем мы исправим либо их версию, либо нашу и повторим слияние только для этого файла.

Получить эти три версии файла, на самом деле, довольно легко. Git хранит все эти версии в индексе в разных “состояниях”, каждое из которых имеет ассоциированный с ним номер. Состояние 1 – это общий предок, состояние 2 – ваша версия и состояния 3 взято из MERGE\_HEAD – версия, которую высливаете (“их” версия).

Вы можете извлечь копию каждой из этих версий конфликтующего файла с помощью команды `git show` и специального синтаксиса.

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

Если вы хотите что-то более суровое, то можете также воспользоваться служебной командой `ls-files -u` для получения SHA-1 хешей для каждого из этих файлов.

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1 hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2 hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3 hello.rb
```

Выражение `:1:hello.rb` является просто сокращением для поиска такого SHA-1 хеша.

Теперь, когда в нашей рабочей директории присутствует содержимое всех трех состояний, мы можем вручную исправить их, чтобы устранить проблемы с пробельными символами и повторно выполнить слияние с помощью малоизвестной команды `git merge-file`, которая делает именно это.

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...

$ git merge-file -p \
 hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -w
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
```

```

+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
#! /usr/bin/env ruby

+# prints out a greeting
def hello
- puts 'hello world'
+ puts 'hello mundo'
end

hello()

```

Теперь у нас есть корректно слитый файл. На самом деле, данный способ лучше, чем использование опции `ignore-all-space`, так как в его рамках вместо игнорирования изменений пробельных символов перед слиянием выполняется корректное исправление таких изменений. При слиянии с `ignore-all-space` мы в результате получим несколько строк с окончаниями в стиле DOS, то есть в одном файле смешаются разные стили окончания строк.

Если перед фиксацией изменений вы хотите посмотреть какие в действительности были различия между состояниями, то можете воспользоваться командой `git diff`, сравнивающей содержимое вашей рабочей директории, которое будет зафиксировано как результат слияния, с любым из трех состояний. Давайте посмотрим на все эти сравнения.

Чтобы сравнить результат слияния с тем, что было в вашей ветке до слияния, или другими словами увидеть, что привнесло данное слияние, вы можете выполнить `git diff --ours`

```

$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@
prints out a greeting
def hello
- puts 'hello world'
+ puts 'hello mundo'
end

hello()

```

Итак, здесь мы можем легко увидеть что же произошло с нашей веткой, какие изменения в действительности внесло слияние в данный файл – изменение только одной строки.

Если вы хотите узнать чем результат слияния отличается от сливаемой ветки, то можете выполнить команду `git diff --theirs`. В этом и следующем примере мы используем опцию `-w` для того, чтобы не учитывать изменения в пробельных символах, так как мы сравниваем результат с тем, что есть в Git, а не с нашим исправленным файлом `hello.theirs.rb`.

```
$ git diff --theirs -w
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
def hello
 puts 'hello mundo'
end
```

И, наконец, вы можете узнать как изменился файл по сравнению сразу с обеими ветками с помощью команды `git diff --base`.

```
$ git diff --base -w
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
 #! /usr/bin/env ruby

+# prints out a greeting
def hello
- puts 'hello world'
+ puts 'hello mundo'
end

hello()
```

В данный момент мы можем использовать команду `git clean` для того, чтобы удалить не нужные более дополнительные файлы, созданные нами для выполнения слияния.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

## ИСПОЛЬЗОВАНИЕ CHECKOUT В КОНФЛИКТАХ

Возможно, нас по каким-то причинам не устраивает необходимость выполнения слияния в текущий момент, или мы не можем хорошо исправить конфликт и нам необходимо больше информации.

Давайте немного изменим пример. Предположим, что у нас есть две долгоживущих ветки, каждая из которых имеет несколько фиксаций, что при слиянии приводит к справедливому конфликту.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) update README
* 9af9d3b add a README
* 694971d update phrase to hola world
| * e3eb223 (mundo) add more tests
| * 7cff591 add testing script
| * c3ffff1 changed text to hello mundo
|/
* b7dcc89 initial hello world code
```

У нас есть три уникальных фиксации, которые присутствуют только в ветке `master` и три других, которые присутствуют в ветке `mundo`. Если мы попытаемся слить ветку `mundo`, то получим конфликт.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

Мы хотели бы увидеть в чем состоит данный конфликт. Если мы откроем конфликтующий файл, то увидим нечто подобное:

```
#!/usr/bin/env ruby
```

```

def hello
<<<<< HEAD
 puts 'hola mundo'
=====
 puts 'hello mundo'
>>>>> mundo
end

hello()

```

В обеих сливаемых ветках в этот файл было добавлено содержимое, но в некоторых фиксациях изменились одни и те же строки, что и привело к конфликту.

Давайте рассмотрим несколько находящихся в вашем распоряжении инструментов, которые позволяют определить как возник этот конфликт. Возможно, не понятно как именно вы должны исправить конфликт и вам требуется больше информации.

Полезным в данном случае инструментом является команда `git checkout` с опцией `--conflict`. Она заново выкачет файл и заменит маркеры конфликта. Это может быть полезно, если вы хотите восстановить маркеры конфликта и попробовать разрешить его снова.

В качестве значения опции `--conflict` вы можете указывать `diff3` или `merge` (последнее значение используется по умолчанию). Если вы укажете `diff3`, Git будет использовать немного другую версию маркеров конфликта – помимо “нашей” и “их” версий файлов будет также отображена “базовая” версия, и таким образом вы получите больше информации.

```
$ git checkout --conflict=diff3 hello.rb
```

После того, как вы выполните эту команду, файл будет выглядеть так:

```

#!/usr/bin/env ruby

def hello
<<<<< ours
 puts 'hola mundo'
||||||| base
 puts 'hello mundo'
=====
 puts 'hello mundo'
>>>>> theirs

```

```
end

hello()
```

Если вам нравится такой формат вывода, то вы можете использовать его по умолчанию для будущих конфликтов слияния, установив параметру `merge.conflictstyle` значение `diff3`.

```
$ git config --global merge.conflictstyle diff3
```

Команде `git checkout` также можно передать опции `--ours` и `--theirs`, которые позволяют действительно быстро выбрать одну из версий файлов, не выполняя слияния совсем.

Это может быть действительно полезным при возникновении конфликтов в бинарных файлах (в этом случае вы можете просто выбрать одну из версий), или при необходимости слить из другой ветки только некоторые файлы (в этом случае вы можете выполнить слияние, а затем перед фиксацией переключить нужные файлы на требуемые версии).

## ИСТОРИЯ ПРИ СЛИЯНИИ

Другой полезный инструмент при разрешении конфликтов слияния – это команда `git log`. Она поможет вам получить информацию о том, что могло привести к возникновению конфликтов. Временами может быть очень полезными просмотреть историю, чтобы понять почему в двух ветках разработки изменялась одна и та же область кода.

Для получения полного списка всех уникальных фиксаций, которые были сделаны в любой из сливаемых веток, мы можем использовать синтаксис “трех точек”, который мы изучили в “Три точки”.

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 update README
< 9af9d3b add a README
< 694971d update phrase to hola mundo
> e3eb223 add more tests
> 7cff591 add testing script
> c3ffff1 changed text to hello mundo
```

Это список всех шести фиксаций, включенных в слияния, с указанием также ветки разработки, в которой находится каждая из фиксаций.

Мы также можем сократить его, попросив предоставить нам более специализированную информацию. Если мы добавим опцию `--merge` к команде `git log`, то она покажет нам только те фиксации, в которых изменялся конфликтующий в данный момент файл.

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3ffff1 changed text to hello mundo
```

Если вы выполните эту команду с опцией `-r`, то получите только список изменений файла, на котором возник конфликт. Это может быть **действительно** полезным для быстрого получения информации, которая необходима, чтобы понять почему что-либо конфликтует и как наиболее правильно это разрешить.

## КОМБИНИРОВАННЫЙ ФОРМАТ ИЗМЕНЕНИЙ

Так как Git добавляет в индекс все успешные результаты слияния, то при вызове `git diff` в состоянии конфликта слияния будет отображено только то, что сейчас конфликтует. Это может быть полезно, так как вы сможете увидеть какие еще конфликты нужно разрешить.

Если вы выполните `git diff` сразу после конфликта слияния, то получите информацию в довольно своеобразном формате.

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@@
 #! /usr/bin/env ruby

 def hello
++<<<<< HEAD
+ puts 'hola world'
+=====
+ puts 'hello mundo'
++>>>>>
 end

 hello()
```

Такой формат называется “комбинированным” (“Combined Diff”), для каждого различия в нем содержится два раздела с информацией. В первом разделе отображены различия строки (добавлена она или удалена) между “вашей” веткой и содержимым вашей рабочей директории, а во втором разделе содержится тоже самое, но между “их” веткой и рабочей директорией.

Таким образом, в данном примере вы можете увидеть строки <<<<< и >>>>> в файле в вашей рабочей директории, хотя они отсутствовали в сливаемых ветках. Это вполне оправдано, потому что, добавляя их, инструмент слияния предоставляет вам дополнительную информацию, но предполагается, что мы удалим их.

Если мы разрешим конфликт и снова выполним команду `git diff`, то получим ту же информацию, но в немного более полезном представлении.

```
$ vim hello.rb
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
#! /usr/bin/env ruby

def hello
- puts 'hola world'
- puts 'hello mundo'
++ puts 'hola mundo'
end

hello()
```

В этом выводе указано, что строка “*hola world*” при слиянии присутствовала в “нашей” ветке, но отсутствовала в рабочей директории, строка “*hello mundo*” была в “их” ветке, но не в рабочей директории, и, наконец, “*hola mundo*” не была ни в одной из сливаемых веток, но сейчас присутствует в рабочей директории. Это бывает полезно просмотреть перед фиксацией разрешения конфликта.

Такую же информацию вы можете получить и после выполнения слияния с помощью команды `git log`, узнав таким образом как был разрешен конфликт. Git выводит информацию в таком формате, если вы выполните `git show` для фиксации слияния или вызовете команду

`git log -p` с опцией ‘`--cc`’ (без нее данная команда не показывает изменения для фиксаций слияния).

```
$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date: Fri Sep 19 18:14:49 2014 +0200

 Merge branch 'mundo'

Conflicts:
 hello.rb

diff --cc hello.rb
index 0399cd5..59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
 #! /usr/bin/env ruby

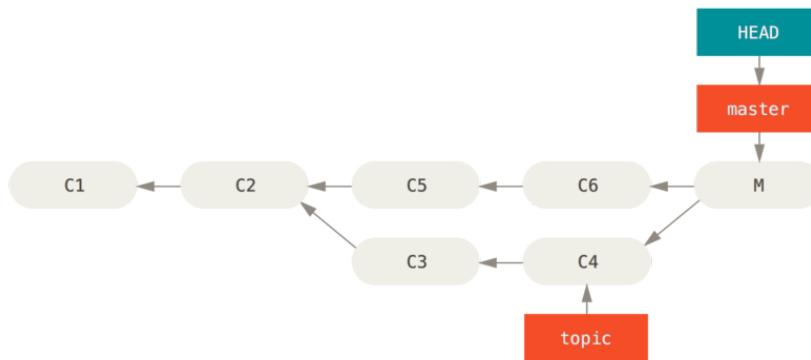
 def hello
- puts 'hola mundo'
- puts 'hello mundo'
++ puts 'hola mundo'
 end

 hello()
```

## Отмена слияний

Теперь когда вы знаете как создать фиксацию слияния, вы можете сделать ее по ошибке. Одна из замечательных вещей в работе с Git – это то, что ошибки совершать не страшно, так как есть возможность исправить их (и в большом количестве случаев сделать это просто).

Фиксации слияния не исключение. Допустим, вы начали работать в тематической ветке, случайно слили ее в `master`, и теперь ваша история фиксаций выглядит следующим образом:

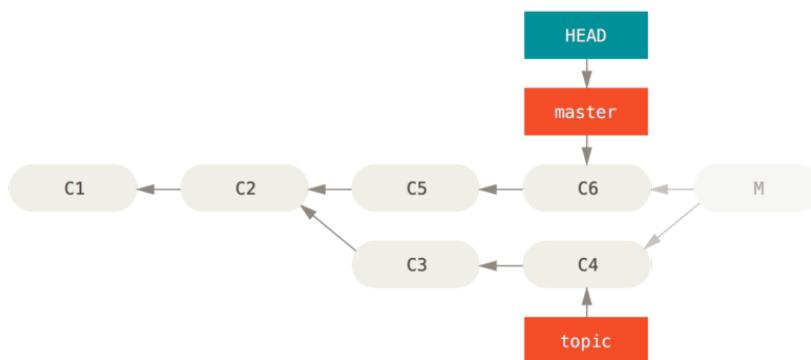
**FIGURE 7-20**

*Случайная  
фиксация слияния*

Есть два подхода к решению этой проблемы, в зависимости от того, какой результат вы хотите получить.

### ИСПРАВЛЕНИЕ ССЫЛОК

Если нежелаемая фиксация слияния существует только в вашем локальном репозитории, то простейшее и лучшее решение состоит в перемещении веток так, чтобы они указывали туда куда вам нужно. В большинстве случаев, если вы после случайного `git merge` выполните команду `git reset --hard HEAD~`, то указатели веток восстановятся так, что будут выглядеть следующим образом:

**FIGURE 7-21**

*История после git  
reset --hard  
HEAD~*

Мы рассматривали команду `reset` ранее в “Раскрытие тайн `reset`”, поэтому вам должно быть не сложно разобраться с тем, что здесь происходит. Здесь небольшое напоминание: `reset --hard` обычно выполняет три шага:

1. Перемещает ветку, на которую указывает `HEAD`. В данном случае мы хотим переместить `master` туда, где она была до фиксации слияния (C6).
2. Приводит индекс к такому же виду что и `HEAD`.
3. Приводит рабочую директорию к такому же виду, что и индекс.

Недостаток этого подхода состоит в изменении истории, что может привести к проблемам в случае совместно используемого репозитория. Загляните в “[The Perils of Rebasing](#)”, чтобы узнать что именно может произойти; кратко говоря, если у других людей уже есть какие-то из изменяемых вами фиксаций, вы должны отказаться от использования `reset`. Этот подход также не будет работать, если после слияния уже была сделана хотя бы одна фиксация; перемещение ссылки фактически приведет к потере этих изменений.

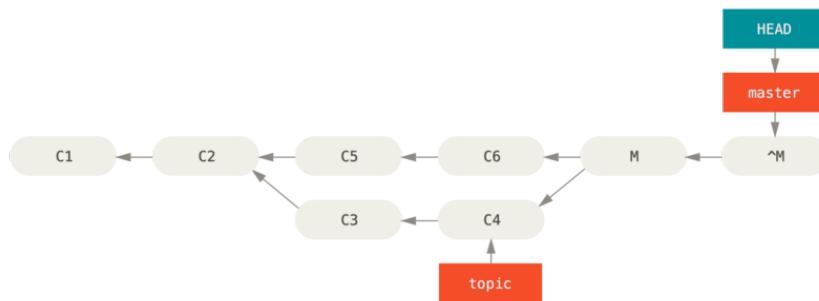
## ОТМЕНА ФИКСАЦИИ

Если перемещение указателей ветки вам не подходит, Git предоставляет возможность сделать новую фиксацию, которая откатывает все изменения, сделанные в другой. Git называет эту операцию “восстановлением” (“revert”), в данном примере вы можете вызвать ее следующим образом:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

Опция `-m 1` указывает какой родитель является “основной веткой” и должен быть сохранен. Когда вы выполняете слияние в `HEAD` (`git merge topic`), новая фиксация будет иметь двух родителей: первый из них `HEAD` (C6), а второй – вершина ветки, которую сливают с текущей (C4). В данном случае, мы хотим отменить все изменения, внесенные слиянием родителя #2 (C4), и сохранить при этом всё содержимое из родителя #1 (C6).

История с фиксацией восстановления (отменой фиксации слияния) выглядит следующим образом:

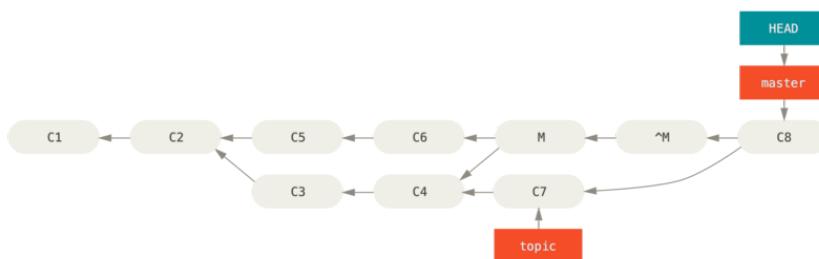
**FIGURE 7-22**

История после `git revert -m 1`

Новая фиксация  $\text{^M}$  имеет точно такое же содержимое как  $\text{C}_6$ , таким образом, начиная с нее всё выглядит так, как будто слияние никогда не выполнялось, за тем лишь исключением, что “теперь уже не слитые” фиксации всё также присутствуют в истории HEAD. Git придет в замешательство, если вы вновь попытаетесь слить `topic` в ветку `master`:

```
$ git merge topic
Already up-to-date.
```

В ветке `topic` нет ничего, что еще недоступно из ветки `master`. Плохо, что в случае добавления новых наработок в `topic`, при повторении слияния Git добавит *только те изменения, которые были сделаны после отмены слияния*:

**FIGURE 7-23**

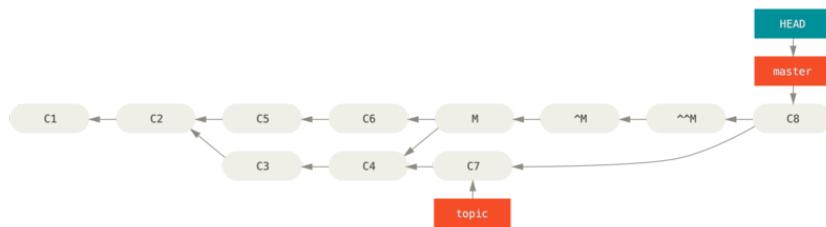
История с плохим слиянием

Лучшим решением данной проблемы является откат фиксации отмены слияния, так как теперь вы хотите внести изменения, которые были отменены, а **затем** создание новой фиксации слияния:

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'''"
$ git merge topic
```

**FIGURE 7-24**

*История после повторения отмененного слияния*



В этом примере, M и ^M отменены. В фиксации ^^M, фактически, сливаются изменения из C3 и C4, а в C8 – изменения из C7, таким образом, ветка topic полностью слита.

## Другие типы слияний

До этого момента мы рассматривали типичные слияния двух веток, которые обычно выполняются с использованием стратегии слияния, называемой “рекурсивной”. Но существуют и другие типы слияния веток. Давайте кратко рассмотрим некоторые из них.

### ВЫБОР “НАШЕЙ” ИЛИ “ИХ” ВЕРСИЙ

Во-первых, существует еще один полезный прием, который мы можем использовать в обычном “рекурсивном” режиме слияния. Мы уже видели опции `ignore-all-space` и `ignore-space-change`, которые передаются с префиксом `-X`, но мы можем также попросить Git при возникновении конфликта использовать ту или иную версию файлов.

По умолчанию, когда Git при слиянии веток замечает конфликт, он добавляет в код маркеры конфликта, отмечает файл как конфликтующий и позволяет вам разрешить его. Если же вместо ручного разрешения конфликта вы хотите, чтобы Git просто

использовал какую-то определенную версию файла, а другую игнорировал, то вы можете передать команде `merge` одну из двух опций `-Xours` или `-Xtheirs`.

В этом случае Git не будет добавлять маркеры конфликта. Все неконфликтующие изменения он сольет, а для конфликтующих он целиком возьмет ту версию, которую вы указали (это относится и к бинарным файлам).

Если мы вернемся к примеру “hello world”, который использовали раньше, то увидим, что попытка слияния в нашу ветку приведет к конфликту.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

Однако, если мы выполним слияние с опцией `-Xours` или `-Xtheirs`, конфликта не будет.

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
hello.rb | 2 ++
test.sh | 2 ++
2 files changed, 3 insertions(+), 1 deletion(-)
create mode 100644 test.sh
```

В этом случае, вместо добавления в файл маркеров конфликта с “hello mundo” в качестве одной версии и с “hola world” в качестве другой, Git просто выберет “hola world”. Однако, все другие неконфликтующие изменения будут слиты успешно.

Такая же опция может быть передана команде `git merge-file`, которую мы обсуждали ранее, то есть для слияния отдельных файлов можно использовать команду `git merge-file --ours`.

На случай если вам нужно нечто подобное, но вы хотите, чтобы Git даже не пытался сливать изменения из другой версии, существует более суровый вариант – *стратегия слияния “ours”*. Важно отметить, что это не тоже самое что *опция “ours”* рекурсивной стратегии слияния.

Фактически, эта стратегия выполнять ненастоящее слияние. Она создаст новую фиксацию слияния, у которой родителями будут обе

ветки, но при этом данная стратегия даже не взглянет на ветку, которую вы сливаете. В качестве результата слияния она просто оставляет тот код, который находится в вашей текущей ветке.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

Вы можете видеть, что между веткой, в которой мы были, и результатом слияния нет никаких отличий.

Это часто бывает полезно, когда нужно заставить Git считать, что ветка уже слита, а реальное слияние отложить на потом. Для примера предположим, что вы создали ветку “release” и проделали в ней некоторую работу, которую когда-то впоследствии захотите спить обратно в “master”. Тем временем в “master” были сделаны некоторые исправления, которые необходимо перенести также в вашу ветку “release”. Вы можете спить ветку с исправлениями в `release`, а затем выполнить `merge -s ours` этой ветки в `master` (хотя исправления в ней уже присутствуют), так что позже, когда вы будете снова сливать ветку `release`, не возникнет конфликтов, связанных с этими исправлениями.

## СЛИЯНИЕ СУБДЕРЕВЬЕВ

Идея слияния субдеревьев состоит в том, что у вас есть два проекта и один из проектов отображается в субдиректорию другого. Когда вы выполняете слияние субдеревьев, Git в большинстве случаев способен понять, что одно из них является субдеревом другого и выполнить слияние подходящим способом.

Далее мы рассмотрим пример добавления в существующий проект другого проекта и последующее слияние кода второго проекта в субдиректорию первого.

Первым делом мы добавим в наш проект приложение Rack. Мы добавим Rack в наш собственный проект, как удаленный репозиторий, а затем выгрузим его в отдельную ветку.

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
```

```

remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
 * [new branch] build -> rack_remote/build
 * [new branch] master -> rack_remote/master
 * [new branch] rack-0.4 -> rack_remote/rack-0.4
 * [new branch] rack-0.9 -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"

```

Таким образом, теперь у нас в ветке `rack_branch` находится основная ветка проекта Rack, а в ветке `master` – наш собственный проект. Если вы переключитесь сначала на одну ветку, а затем на другую, то увидите, что они имеют абсолютно разное содержимое:

```

$ ls
AUTHORS KNOWN-ISSUES Rakefile contrib lib
COPYING README bin example test
$ git checkout master
Switched to branch "master"
$ ls
README

```

Может показаться странным, но, на самом деле, ветки в вашем репозитории не обязаны быть ветками одного проекта. Это мало распространено, так как редко бывает полезным, но иметь ветки, имеющие абсолютно разные истории, довольно легко.

В данном примере, мы хотим выгрузить проект Rack в субдиректорию нашего основного проекта. В Git мы можем выполнить это с помощью команды `git read-tree`. Вы узнаете больше о команде `read-tree` и её друзьях в [Chapter 10](#), сейчас же вам достаточно знать, что она считывает содержимое некоторой ветки в ваш текущий индекс и рабочий каталог. Мы просто переключимся обратно на ветку `master` и выгрузим ветку `rack` в субдиректорию `rack` ветки `master` нашего основного проекта:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Когда мы будем выполнять коммит, он будет выглядеть так, как будто все файлы проекта Rack были добавлены в эту субдиректорию –

например, мы скопировали их из архива. Важно отметить, что слить изменения одной из веток в другую довольно легко. Таким образом, если проект Rack обновился, мы можем получить изменения из его репозитория просто переключившись на соответствующую ветку и выполнив операцию `git pull`:

```
$ git checkout rack_branch
$ git pull
```

Затем мы можем слить эти изменения обратно в нашу ветку `master`. Мы можем использовать `git merge -s subtree` и это будет прекрасно работать; но Git также сольёт вместе истории проектов, а этого мы, возможно, не хотим. Для того, чтобы получить изменения и заполнить сообщение коммита используйте опции `--squash` и `--no-commit`, вместе с опцией `-Xsubtree` рекурсивной стратегии слияния. Вообще-то, по умолчанию используется именно рекурсивная стратегия слияния, но мы указали и её тоже для пущей ясности.

```
$ git checkout master
$ git merge --squash -s recursive -Xsubtree=rack --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Все изменения из проекта Rack слиты и подготовлены для локального выполнения коммита. Вы также можете поступить наоборот – сделать изменения в субдиректории `rack` вашей основной ветки и затем слить их в вашу ветку `rack_branch`, чтобы позже передать их ответственным за проекты или отправить их в вышестоящий репозиторий проекта Rack.

Таким образом, слияние субдеревьев дает нам возможность использовать рабочий процесс в некоторой степени похожий на рабочий процесс с субмодулями, но при этом без использования субмодулей (которые мы рассмотрим в “Подмодули”). Мы можем держать ветки с другими связанными проектами в нашем репозитории и периодически сливать их как субдеревья в наш проект. С одной стороны это удобно, например, тем, что весь код хранится в одном месте. Однако, при этом есть и некоторые недостатки – субдеревья немного сложнее, проще допустить ошибки при повторной интеграции изменений или случайно отправить ветку не в тот репозиторий.

Другая небольшая странность состоит в том, что для получения различий между содержимым вашей субдиректории `rack` и содержимого ветки `rack_branch` – для того, чтобы увидеть необходимо ли выполнять слияния между ними – вы не можете использовать обычную команду `diff`. Вместо этого вы должны выполнить команду `git diff-tree`, указав ветку, с которой вы ходите выполнить сравнение:

```
$ git diff-tree -r rack_branch
```

Для сравнения содержимого вашей субдиректории `rack` с тем, что находилось в ветке `master` сервера, когда вы последний раз извлекали из него изменения, вы можете выполнить:

```
$ git diff-tree -r rack_remote/master
```

## Rerere

Функциональность `git gegrge` – частично скрытый компонент Git. Ее имя является сокращением для “reuse recorded resolution” (“повторное использование сохраненных разрешений конфликтов”). Как следует из имени, эта функциональность позволяет попросить Git запомнить то, как вы разрешили некоторую часть конфликта, так что в случае возникновения такого же конфликта, Git сможет его разрешить автоматически.

Существует несколько ситуаций, в которых данная функциональность может быть действительно удобна. Один из примеров, упомянутый в документации, состоит в том, чтобы обеспечить в будущем простоту слияния некоторой долгоживущей ветки, не создавая при этом набор промежуточных коммитов слияния. При использовании `gegrge` вы можете время от времени выполнять слияния, разрешать конфликты, а затем откатывать слияния. Если делать это постоянно, то итоговое слияние должно пройти легко, так как `gegrge` сможет разрешить все конфликты автоматически.

Такая же тактика может быть использована, если вы хотите сохранить ветку легко перебазируемой, то есть вы не хотите сталкиваться с одними и теми же конфликтами каждый раз при перебазировании. Или, например, вы решили ветку, которую ужесливали и разрешали при этом некоторые конфликты, вместо слияния

перебазировать – наврядли вы захотите разрешать те же конфликты еще раз.

Другая ситуация возникает, когда вы изредка сливаете несколько веток, относящихся к еще разрабатываемым задачам, в одну тестовую ветку, как это часто делается в проекте самого Git. Если тесты завершатся неудачей, вы можете откатить все слияния и повторить их, исключив из них ветку, которая поломала тесты, при этом не разрешая конфликты снова.

Для того, чтобы включить функциональность гегеге, достаточно изменить настройки следующим образом:

```
$ git config --global gogoge.enabled true
```

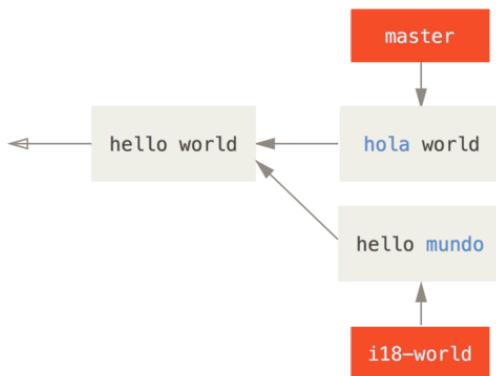
Также вы можете включить ее, создав каталог `.git/gg-cache` в нужном репозитории, но включение через настройки понятнее и может быть сделано глобально.

Давайте рассмотрим простой пример, подобный используемому ранее. Предположим, у нас есть файл вида:

```
#!/usr/bin/env ruby

def hello
 puts 'hello world'
end
```

Как и ранее, в одной ветке мы изменили слово “hello” на “hola”, а в другой – слово “world” на “mundo”.

**FIGURE 7-25**

Когда мы будем сливать эти две ветки в одну, мы получим конфликт:

```
$ git merge i18n-world
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Recorded preimage for 'hello.rb'
Automatic merge failed; fix conflicts and then commit the result.
```

Вы должно быть заметили в выводе новую строку `Recorded pre-image for FILE`. Во всем остальном вывод такой же, как при обычном конфликте слияния. В этот момент гегеге может сообщить нам несколько вещей. Обычно в такой ситуации вы можете выполнить `git status`, чтобы увидеть в чем заключается конфликт:

```
$ git status
On branch master
Unmerged paths:
(use "git reset HEAD <file>..." to unstage)
(use "git add <file>..." to mark resolution)
#
both modified: hello.rb
#
```

Однако, с помощью команды `git gegrere status` вы также можете узнать, для каких файлов `git gegrere` сохранил снимки состояния, в котором они были до начала слияния:

```
$ git gegrere status
hello.rb
```

А команда `git gegrere diff` показывает текущее состояние разрешения конфликта – то, с чего вы начали разрешать конфликт, и то, как вы его разрешили (фактически, патч, который в дальнейшем можно использовать для разрешения такого же конфликта).

```
$ git gegrere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,11 @@
#! /usr/bin/env ruby

def hello
-<<<<<
- puts 'hello mundo'
-=====
+<<<<< HEAD
 puts 'hola world'
->>>>>
+=====
+ puts 'hello mundo'
+>>>>> i18n-world
end
```

Также (и это уже не относится к `gegrere`), вы можете использовать команду `ls-files -u`, чтобы увидеть конфликтующие файлы, их общую родительскую версию и обе сливаемых версии:

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1 hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2 hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3 hello.rb
```

Теперь вы можете разрешить конфликт, используя `puts 'hola mundo'`, и снова выполнить команду `gegrere diff`, чтобы увидеть, что именно `gegrere` запомнит:

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
#! /usr/bin/env ruby

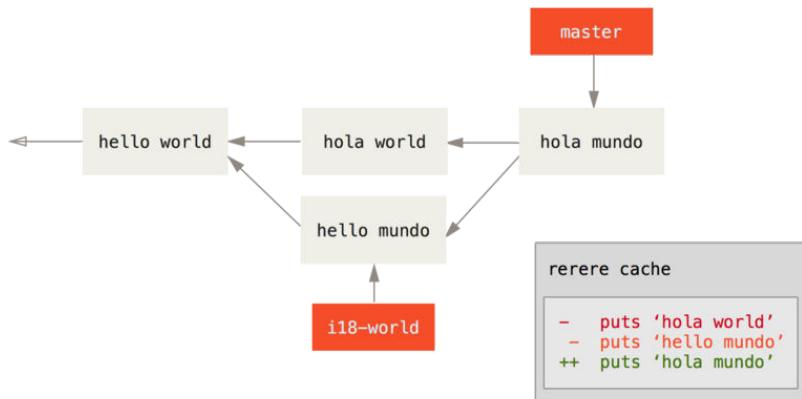
def hello
-<<<<<
- puts 'hello mundo'
=====
- puts 'hola world'
->>>>>
+ puts 'hola mundo'
end
```

То есть, когда Git увидит в файле `hello.rb` конфликт, в котором с одной стороны стоит “`hello mundo`” и “`hola world`” с другой, он разрешит его как “`hola mundo`”.

Теперь мы можем отметить конфликт как разрешенный и закоммитить его:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

Как вы видите, при этом было “сохранено разрешение конфликта для ФАЙЛА” (“Recorded resolution for FILE”).

**FIGURE 7-26**

Теперь давайте отменим это слияние и перебазируем ветку `i18n-world` поверх `master`. Как мы видели в “Раскрытие тайн `reset`”, мы можем переместить нашу ветку назад, используя команду `reset`.

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

Наше слияние отменено. Теперь давайте перебазируем ветку `i18n-world`.

```
$ git checkout i18n-world
Switched to branch 'i18n-world'

$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: i18n one word
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Failed to merge in the changes.
Patch failed at 0001 i18n one word
```

При этом мы получили ожидаемый конфликт слияния, но обратите внимание на строку `Resolved FILE using previous resolution.`. Если мы посмотрим на содержимое файла, то увидим, что конфликт уже был разрешен, и в файле отсутствуют маркеры конфликта слияния.

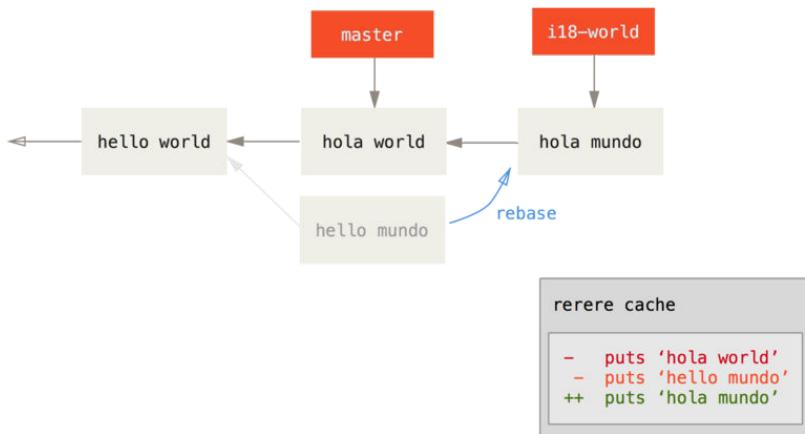
```
$ cat hello.rb
#!/usr/bin/env ruby

def hello
 puts 'hola mundo'
end
```

При этом команда `git diff` покажет вам *как именно* этот конфликт был автоматически повторно разрешен:

```
$ git diff
diff --cc hello.rb
index a440db6,54336ba..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
 #! /usr/bin/env ruby

 def hello
- puts 'hola world'
- puts 'hello mundo'
++ puts 'hola mundo'
 end
```

**FIGURE 7-27**

С помощью команды `checkout` вы можете вернуть этот файл назад в конфликтующее состояние:

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#!/usr/bin/env ruby

def hello
<<<<< ours
 puts 'hola mundo'
=====
 puts 'hello mundo'
>>>>> theirs
end
```

Мы видели пример этого в “Продвинутое слияние”. Теперь давайте повторно разрешим конфликт используя `git merge`:

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#!/usr/bin/env ruby

def hello
 puts 'hola mundo'
end
```

Мы автоматически повторно разрешили конфликт, используя сохраненный гегеге вариант разрешения. Теперь вы можете добавить файл в индекс и продолжить перебазирование ветки.

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

Итак, если вы выполняете много повторных слияний или хотите сохранять тематическую ветку в состоянии, актуальном вашей основной ветке, без множества слияний в истории, или часто перебазируете ветки, то вы можете включить гегеге. Это, в какой-то мере, упростит вам жизнь.

## Обнаружение ошибок с помощью Git

Git предоставляет несколько инструментов, которые помогут вам найти и устранить проблемы в ваших проектах. Так как Git рассчитан на работу с проектом почти любого типа, эти инструменты имеют довольно обобщенные возможности, но часто они могут помочь вам отловить ошибку или ее виновника.

### Аннотация файла

Если вы обнаружили ошибку в вашем коде и хотите знать, когда она была добавлена и почему, то в большинстве случаев аннотация файла будет лучшим инструментом для этого. С помощью нее для любого файла можно увидеть, каким коммитом последний раз изменили каждую из строк. Поэтому если вы видите, что некоторый метод в вашем коде работает неправильно, вы можете с помощью команды `git blame` снабдить файл аннотацией, и таким образом увидеть, когда каждая строка метода была изменена последний раз и кем. В следующем примере используется опция `-L`, чтобы ограничить вывод строками с 12 по 22:

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree = 'master')
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13) command("git show #{tree}")
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree = 'master')
```

```

79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17) command("git log #{tree}")
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21) command("git blame #{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end

```

Обратите внимание, что первое поле – это неполная SHA-1 сумма последнего коммита, который изменял соответствующую строку. Следующими двумя полями являются значения, извлеченные из этого коммита – имя автора и время создания коммита – таким образом, вы можете легко увидеть кто изменял строку и когда. После этого следуют номер строки и содержимое файла. Обратите внимание на строки со значением ^4832fe2 в поле коммита, так обозначаются те строки, которые были в первом коммите этого файла. Этот коммит был сделан, когда данный файл был впервые добавлен в проект и с тех пор эти строки не были изменены. Немного сбивает с толку то, что вы уже видели в Git, по крайней мере, три различных варианта использования символа ^ для изменения SHA-1 коммита, в данном случае этот символ имеет такое значение.

Другая отличная вещь в Git – это то, что он не отслеживает явно переименования файлов (пользователю не нужно явно указывать какой файл в какой был переименован). Он сохраняет снимки и уже после выполнения самого переименования неявно попытается выяснить, что было переименовано. Одна из интересных возможностей, вытекающих из этого – это то, что вы также можете попросить Git выявить перемещения кода всех других видов. Если передать опцию -C команде git blame, Git проанализирует аннотируемый файл и пытается выяснить откуда изначально появились фрагменты кода, если они, конечно же, были откуда-то скопированы. Например, предположим при реорганизации кода в файле GITServerHandler.m вы разнесли его по нескольким файлам, один из которых GITPackUpload.m. Вызывая git blame с опцией -C для файла GITPackUpload.m, вы можете увидеть откуда изначально появились разные фрагменты этого файла.

```

$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144) //NSLog(@"%@", GATHER COMM
ad11ac80 GITPackUpload.m (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 146) NSString *parentSha;

```

```

ad11ac80 GITPackUpload.m (Scott 2009-03-24 147) GITCommit *commit = [g
ad11ac80 GITPackUpload.m (Scott 2009-03-24 148) //NSLog(@"GATHER COMM
ad11ac80 GITPackUpload.m (Scott 2009-03-24 149) if(commit) {
ad11ac80 GITPackUpload.m (Scott 2009-03-24 150) [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)

```

Это, действительно, полезно. Обычно вы получаете в качестве изначального коммит, в котором вы скопировали код, так как это первый коммит, в котором вы обращаетесь к этим строкам в *этом* файле. Но в данном случае Git сообщает вам первый коммит, в котором эти строки были написаны, даже если это было сделано в другом файле.

## Бинарный поиск

Аннотирование файла помогает, если вы знаете, где находится проблема и можете начать исследование с этого места. Если вы не знаете, что сломано, а с тех пор как код работал, были сделаны десятки или сотни коммитов, вы вероятно воспользуетесь командой `git bisect`. Эта команда выполняет бинарный поиск по истории коммитов для того, чтобы помочь вам как можно быстрее определить коммит, который создал проблему.

Допустим, вы только что развернули некоторую версию вашего кода в боевом окружении и теперь получаете отчеты о некоторой ошибке, которая не возникала в вашем разработческом окружении, и вы не можете представить, почему код ведет себя так. Вы возвращаетесь к вашему коду и выясняете, что можете воспроизвести проблему, но всё еще не понимаете, что работает неверно. Вы можете воспользоваться бинарным поиском, чтобы выяснить это. Во-первых, выполните команду `git bisect start` для запуска процесса поиска, а затем используйте `git bisect bad`, чтобы сообщить Git, что текущий коммит сломан. Затем, используя `git bisect good [good_commit]`, вы должны указать, когда было последнее известное рабочее состояние:

```

$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo

```

Git выяснил, что произошло около 12 коммитов между коммитом, который вы отметили как последний хороший коммит (v1.0), и текущим плохим коммитом, и выгрузил вам один из середины. В этот момент вы можете запустить ваши тесты, чтобы проверить присутствует ли проблема в этом коммите. Если это так, значит она была внесена до выгруженного промежуточного коммита, если нет, значит проблема была внесена после этого коммита. Пусть в данном коммите проблема не проявляется, вы сообщаете об этом Git с помощью `git bisect good` и продолжаете ваше путешествие:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Теперь вы оказались на другом коммите, расположенному посередине между только что протестированным и плохим коммитами. Вы снова выполняете ваши тесты, обнаруживаете, что текущий коммит сломан, и сообщаете об этом Git с помощью команды `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

Этот коммит хороший и теперь Git имеет всю необходимую информацию для определения того, где была внесена ошибка. Он сообщает вам SHA-1 первого плохого коммита и отображает некоторую информацию о коммите и файлах, которые были изменены в этом коммите, так, чтобы вы смогли разобраться что же случилось, что могло привнести эту ошибку:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date: Tue Jan 27 14:48:32 2009 -0800

 secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfcc639b1a3814550e62d60b8e68a8e4 M config
```

Когда вы закончили бинарный поиск, нужно выполнить `git bisect reset` для того, чтобы вернуть HEAD туда, где он был до начала поиска, иначе вы останетесь в, довольно, причудливом состоянии:

```
$ git bisect reset
```

Это мощный инструмент, который помогает вам за считанные минуты проверить сотни коммитов на возможность внесения ошибки. В действительности, если у вас есть скрипт, который будет возвращать 0 если проект находится в рабочем состоянии и любое другое число в обратном случае, то вы можете полностью автоматизировать `git bisect`. Сперва, вы снова сообщаете границы бинарного поиска, указывая известные плохие и хорошие коммиты. Вы можете сделать это, передавая их команде `bisect start` – первым аргументом известный плохой коммит, а вторым известный хороший коммит:

```
$ git bisect start HEAD v1.0
$ git bisect run test-errgor.sh
```

Это приведет к автоматическому выполнению `test-errgor.sh` на каждый выгруженный коммит до тех пор, пока Git не найдет первый сломанный коммит. Вы также можете использовать что-то вроде `make` или `make tests`, или что-то еще, что у вас есть для запуска автоматизированных тестов.

## Подмодули

Часто при работе над одним проектом, возникает необходимость использовать в нем другой проект. Возможно, это библиотека, разрабатываемая сторонними разработчиками или вами, но в рамках отдельного проекта, и используемая в нескольких других проектах. Типичная проблема, возникающая при этом – вы хотите продолжать работать с двумя проектами по отдельности, но при этом использовать один из них в другом.

Приведем пример. Предположим, вы разрабатываете веб-сайт и создаете ленту в формате Atom. Вместо написания собственного генератор Atom, вы решили использовать библиотеку. Вы, вероятно, должны либо включить нужный код из разделяемой библиотеки,

например, модуля CPAN или Ruby gem, либо скопировать исходный код библиотеки в ваш проект. Проблема с использованием библиотеки состоит в сложной адаптации библиотеки под свои нужды и часто более сложным ее распространением, так как вам нужно быть уверенным, что каждому клиенту доступна такая библиотека. При включении кода библиотеки в свой проект проблема будет заключаться в сложном объединении ваших собственных изменений с изменениями в вышестоящем репозитории.

Git решает эту проблему, предоставляя функционал подмодулей. Подмодули позволяют вам сохранить один Git-репозиторий как поддиректорию другого Git-репозитория. Это дает вам возможность склонировать в ваш проект другой репозиторий, но фиксации при этом хранить отдельно.

## Начало работы с подмодулями

Далее мы рассмотрим процесс разработки простого проекта, разбитого на один главный проект и несколько подпроектов.

Давайте начнем с добавления существующего Git-репозитория, в качестве подмодуля репозитория, в котором мы работаем. Для добавления нового подмодуля используйте команду `git submodule add` с URL проекта, который вы хотите начать отслеживать. В данном примере мы добавим библиотеку “DbConnector”.

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

По умолчанию подмодули добавляют подпроекты в директории, называемые так же, как и соответствующие репозитории, в нашем примере – “DbConnector”.

Если в данный момент вы выполните `git status`, то заметите несколько моментов.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
```

```
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

 new file: .gitmodules
 new file: DbConnector
```

Во-первых, вы должны заметить новый файл `.gitmodules`. Это конфигурационный файл, в котором хранится соответствие между URL проекта и локальной поддиректорией, в которую вы его выкачали:

```
$ cat .gitmodules
[submodule "DbConnector"]
path = DbConnector
url = https://github.com/chaconinc/DbConnector
```

Если у вас несколько подмодулей, то и в этом файле у вас будет несколько записей. Важно заметить, что этот файл добавлен под управление Git так же, как и другие ваши файлы, например, ваш файл `.gitignore`. Этот файл можно получить или отправить на сервер вместе с остальными файлами проекта. Благодаря этому другие люди, который клонируют ваш проект, узнают откуда взять подмодули проекта.

---

Поскольку другие люди первым делом будут пытаться выполнить команды `clone/fetch` по URL, указанным в файле `.gitmodules`, старайтесь проверять, что URL будут им доступны. Например, если вы выполняете отправку по URL отличному от того, по которому другие люди получают данные, то используйте URL, к которому у других участников будет доступ. Вы можете изменить это значение локально только для себя с помощью команды `git config submodule.DbConnector.url PRIVATE_URL`.

---

Следующим элементом вывода `git status` является сама директория проекта. Если вы выполните `git diff` для нее, то увидите кое-что интересное:

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 000000..c3f01dc
--- /dev/null
+++ b/DbConnector
```

```
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

Хотя DbConnector — является поддиректорией вашей рабочей директории, Git распознает ее как подмодуль и не отслеживает ее содержимое, когда вы не находитесь в этой директории. Вместо этого, Git видит ее как некоторую отдельную фиксацию из этого репозитория.

Если вам нужен немного более понятный вывод, то можете передать команде `git diff` опцию `--submodule`.

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+ path = DbConnector
+ url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 000000...c3f01dc (new submodule)
```

Когда вы выполните фиксацию, то увидите следующее:

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
2 files changed, 4 insertions(+)
create mode 100644 .gitmodules
create mode 160000 DbConnector
```

Обратите внимание на права доступа 160000 у DbConnector. Это специальные права доступа в Git, которые, по сути, означают, что вы сохраняете фиксацию как элемента каталога, а не как поддиректорию или файл.

## Клонирование проекта с подмодулями

Далее мы рассмотрим клонирование проекта, содержащего подмодули. Когда вы клонируете такой проект, по умолчанию вы получите директории, содержащие подмодули, но ни одного файла в них не будет:

```
$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x 9 schacon staff 306 Sep 17 15:21 .
drwxr-xr-x 7 schacon staff 238 Sep 17 15:21 ..
drwxr-xr-x 13 schacon staff 442 Sep 17 15:21 .git
-rw-r--r-- 1 schacon staff 92 Sep 17 15:21 .gitmodules
drwxr-xr-x 2 schacon staff 68 Sep 17 15:21 DbConnector
-rw-r--r-- 1 schacon staff 756 Sep 17 15:21 Makefile
drwxr-xr-x 3 schacon staff 102 Sep 17 15:21 includes
drwxr-xr-x 4 schacon staff 136 Sep 17 15:21 scripts
drwxr-xr-x 4 schacon staff 136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
$
```

Директория DbConnector присутствует, но они пустая. Вы должны выполнить две команды: `git submodule init` – для инициализации локального конфигурационного файла, и `git submodule update` – для извлечения всех данных этого проекта и переключения на соответствующую фиксацию, указанную в вашем основном проекте.

```
$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbConnector'
$ git submodule update
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Теперь ваша директория DbConnector находятся в точно таком же состоянии, как и ранее при выполнении фиксации.

Однако, существует другой немного более просто вариант сделать тоже самое. Если вы передадите опцию `--recursive` команде `git`

`clone`, то она автоматически инициализирует и обновит каждый подмодуль в этом репозитории.

```
$ git clone --recursive https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29b'
```

## Работа над проектом с подмодулями

Теперь у нас есть копия проекта с подмодулями. Давайте рассмотрим, как мы будем работать совместно с нашими коллегами над основным проектом и над подпроектом.

### ПОЛУЧЕНИЕ ИЗМЕНЕНИЙ ИЗ ВЫШЕСТОЯЩЕГО РЕПОЗИТОРИЯ

Простейший вариант использования подмодулей в проекте состоит в том, что вы просто получаете сам подпроект и хотите периодически получать обновления, но в своей копии проекта ничего не изменяете. Давай рассмотрим этот простой пример.

Если вы хотите проверить наличие изменений в подмодуле, вы можете перейти в его директорию, выполнить `git fetch` и затем `git merge` для обновления локальной версии из вышестоящего репозитория.

```
$ git fetch
From https://github.com/chaconinc/DbConnector
 c3f01dc..d0354fc master -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
 scripts/connect.sh | 1 +
```

```
src/db.c | 1 +
2 files changed, 2 insertions(+)
```

Теперь если вы вернетесь в основной проект и выполните `git diff --submodule`, то сможете увидеть, что подмодуль обновился, и получить список новых фиксаций. Если вы не хотите каждый раз при вызове `git diff` указывать опцию `--submodule`, то можете установить такой формат вывода по умолчанию, задав параметру `diff.submodule` значение “`log`”.

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
 > more efficient db routine
 > better connection routine
```

Если в данный момент вы создадите фиксацию, то таким образом сделаете доступным новый код в подмодуле для других людей.

Если вы не хотите вручную извлекать и сливать изменения в поддиректорию, то для вас существует более простой способ сделать тоже самое. Если вы выполните `git submodule update --remote`, то Git сам перейдет в ваши подмодули, заберет изменения и обновит их для вас.

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
 3f19983..d0354fc master -> origin/master
Submodule path 'DbConnector': checked out 'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

Эта команда по умолчанию предполагает, что вы хотите обновить локальную копию до состояния ветки `master` из репозитория подмодуля. Однако, по желанию вы можете изменить это. Например, если вы хотите, чтобы подмодуль `DbConnector` отслеживал ветку “`stable`” репозитория, то вы можете установить это либо в файле `.gitmodules` (тогда и другие люди также будут отслеживать эту ветку), либо в вашем локальном файле `.git/config`. Давайте настроим это в файле `.gitmodules`:

```
$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
 27cf5d3..c87d55d stable -> origin/stable
Submodule path 'DbConnector': checked out 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae668'
```

Если вы уберете `-f .gitmodules`, то команда сделает изменения локально только у вас, но, кажется, имеет смысл всё же отправлять эту информацию в репозиторий, так чтобы и все остальные участники имели к ней доступ.

Если в данный момент мы выполним `git status`, то Git покажет нам, что у нас есть “новые фиксации” в подмодуле.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/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: .gitmodules
 modified: DbConnector (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
```

Если вы установите в настройках параметр `status.submodulesummary`, то Git будет также отображать краткое резюме об изменениях в ваших подмодулях:

```
$ git config status.submodulesummary 1

$ git status
On branch master
Your branch is up-to-date with 'origin/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: .gitmodules
modified: DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c3f01dc...c87d55d (4):
 > catch non-null terminated lines

```

Если сейчас вы выполните `git diff`, то сможете увидеть, что изменился наш файл `.gitmodules`, а также, что существует несколько полученных вами фиксаций, которые готовы для фиксации в проекте вашего подмодуля.

```

$ git diff
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
 path = DbConnector
 url = https://github.com/chaconinc/DbConnector
+
 branch = stable
Submodule DbConnector c3f01dc..c87d55d:
 > catch non-null terminated lines
 > more robust error handling
 > more efficient db routine
 > better connection routine

```

Здорово, что мы можем увидеть список подготовленных фиксаций в нашем подмодуле. Но после создания фиксации, вы также можете получить эту информацию, если выполните команду `git log -p`.

```

$ git log -p --submodule
commit 0a24cf121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
Date: Wed Sep 17 16:37:02 2014 +0200

 updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules

```

```

@@ -1,3 +1,4 @@
[submodule "DbConnector"]
 path = DbConnector
 url = https://github.com/chaconinc/DbConnector
+
 branch = stable
Submodule DbConnector c3f01dc..c87d55d:
 > catch non-null terminated lines
 > more robust error handling
 > more efficient db routine
 > better connection routine

```

По умолчанию при выполнении команды `git submodule update --remote` Git будет пытаться обновить **все** ваши подмодули, поэтому если у вас их много, вы можете указать имя подмодуля, который вы хотите попробовать обновить.

## РАБОТА С ПОДМОДУЛЕМ

Весьма вероятно, что вы используете подмодули, потому что хотите работать над кодом подмодуля (или нескольких подмодулей) во время работы над кодом основного проекта. Иначе бы вы, скорее всего, предпочли использовать более простую систему управления зависимостями (такую как Maven или Rubygems).

Давайте теперь рассмотрим пример, в котором мы одновременно с изменениями в основном проекте внесем изменения в подмодуль, зафиксировав и опубликовав все эти изменения в одно и тоже время.

До сих пор, когда мы выполняли команду `git submodule update` для извлечения изменений из репозиториев подмодуля, Git получал изменения и обновлял файлы в поддиректории, но оставлял подрепозиторий в состоянии, называемом “отделенный HEAD” (“detached HEAD”). Это значит, что локальная рабочая ветка (такая, например, как “`master`”), отслеживающая изменения, отсутствует. Таким образом, любые вносимые вами изменения не будет нормально отслеживаться.

Для упрощения работы с подмодулями вам необходимо сделать две вещи. Вам нужно перейти в каждый подмодуль и переключиться на ветку, в которой будете в дальнейшем работать. Затем вам необходимо сообщить Git, что ему делать если вы внесли изменения, а затем командой `git submodule update --remote` получаете новые изменения из репозитория. Возможны два варианта – вы можете слить их в вашу локальную версию или попробовать перебазировать ваши локальные наработки поверх новых изменений.

Первым делом, давайте перейдем в директорию нашего подмодуля и переключимся на нужную ветку.

```
$ git checkout stable
Switched to branch 'stable'
```

Давайте попробуем воспользоваться опцией “merge” (“слияния”). Для того, чтобы задать ее вручную, мы можем просто добавить опцию `--merge` в наш вызов команды `update`.

```
$ git submodule update --remote --merge
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
 c87d55d..92c7337 stable -> origin/stable
Updating c87d55d..92c7337
Fast-forward
 src/main.c | 1 +
 1 file changed, 1 insertion(+)
Submodule path 'DbConnector': merged in '92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

Если мы перейдем в директорию `DbConnector`, то увидим, что новые изменения уже слиты в нашу локальную ветку `stable`. Теперь давайте посмотрим, что случится, когда мы внесем свои собственные локальные изменения в библиотеку, а кто-то другой в это же время отправит другие изменения в вышестоящий репозиторий.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'unicode support'
[stable f906e16] unicode support
 1 file changed, 1 insertion(+)
```

Теперь если мы обновим наш подмодуль, то сможем увидеть, что случится, когда мы сделали локальные изменения, а вышестоящий репозиторий также имеет изменения, которые мы должны объединить.

```
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
```

```
Applying: unicode support
Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da9'
```

Если вы забудете указать опцию `--rebase` или `--merge`, то Git просто обновит ваш подмодуль, до состояния, что есть на сервере, и установит ваш проект в состояние отделенного HEAD.

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da9'
```

Не беспокойтесь, если такое случится, вы можете просто вернуться в директорию, переключиться обратно на вашу ветку (которая все еще будет содержать ваши наработки) и слить или перебазировать ветку `origin/stable` (или другую нужную вам удаленную ветку) вручную.

Если вы не зафиксировали ваши изменения в подмодуле и выполнили его обновление, то это приведет к проблемам – Git извлечет изменения из вышестоящего репозитория, но не затрет несохраненные наработки в директории вашего подмодуля.

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
 5d60ef9..c75e92a stable -> origin/stable
error: Your local changes to the following files would be overwritten by checkout:
 scripts/setup.sh
Please, commit your changes or stash them before you can switch branches.
Aborting
Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path 'DbConnector'
```

Если вы сделали изменения, которые конфликтуют с какими-то изменениями в вышестоящем репозитории, то Git сообщит вам об этом, когда вы запустите операцию обновление.

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
```

```
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path 'DbConnector'
```

Вы можете перейти в директорию подмодуля и исправить конфликт обычным образом.

## ПУБЛИКАЦИЯ ИЗМЕНЕНИЙ В ПОДМОДУЛЕ

Теперь у нас есть некоторые изменения в директории нашего подмодуля. Некоторые из них мы получили при обновлении из вышестоящего репозитория, а другие были сделаны локально и пока никому не доступны, так как мы их еще никуда не отправили.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
> Merge from origin/stable
> updated setup script
> unicode support
> remove unnessesary method
> add new option for conn pooling
```

Если мы создадим фиксацию в основном проекте и отправим ее на сервер, не отправив при этом изменения в подмодуле, то другие люди, которые попытаются использовать наши изменения, столкнутся с проблемами, так как у них не будет возможности получить требуемые изменения подмодуля. Эти изменения будут присутствовать только в нашей локальной копии.

Для того, чтобы гарантированно избежать этой проблемы, вы можете перед отправкой основного проекта попросить Git проверить, что все наши подмодули сами были корректно отправлены на серверы. Команда `git push` принимает аргумент `--recurse-submodules`, который может принимать значения “`check`” или “`on-demand`”. Использование значения “`check`” придет к тому, что `push` просто завершится неудачей, если какой-то из зафиксированных подмодулей не был отправлен на сервер.

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
 DbConnector

Please try
```

```
git push --recurse-submodules=on-demand

or cd to the path and use

 git push

to push them to a remote.
```

Как видите, эта команда также дает нам некоторые полезные советы о том, что мы могли бы делать дальше. Самый простой вариант – это пройти по всем подмодулям и вручную отправить изменения на серверы, чтобы гарантировать доступность изменений другим людям, а зачем повторять первоначальную команду `push`.

Другой вариант – это использовать значение “`on-demand`”, которое попытается сделать это все за вас.

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
 c75e92a..82d2ad3 stable -> stable
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/MainProject
 3d6d338..9a377d1 master -> master
```

Как видите, перед отправкой на сервер основного проекта Git перешел в директорию модуля `DbConnector` и отправил на сервер его. Если отправка подмодуля по каким-то причинам завершилась неудачей, то и отправка основного проекта также завершится неудачей.

## ОБЪЕДИНЕНИЕ ИЗМЕНЕНИЙ ПОДМОДУЛЯ

Если вы измените ссылку на подмодуль одновременно с кем-то еще, то вы можете столкнуться с некоторыми проблемами. Такое

случается если истории подмодуля разошлись и они зафиксированы в разошедшихся ветках основного проекта. Для исправления такой ситуации потребуются некоторые дополнительные действия.

Если одна фиксация является прямым предком другой (слияние может быть выполнено перемоткой вперед), то Git просто выберет последнюю для выполнения слияния, то есть все отработает хорошо.

Однако, Git не будет пытаться выполнить даже простейшего слияния. Если фиксации подмодуля разошлись и слияние необходимо, вы получите нечто подобное:

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
From https://github.com/chaconinc/MainProject
 9a377d1..eb974f8 master -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits not found)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

Здесь говорится о том, что Git понял, что в этих двух ветках содержатся указатели на разошедшиеся записи в истории подмодуля и их необходимо слить. Git поясняет это как “merge following commits not found”, что несколько обескураживает, но мы объясним почему так происходит.

Для решения этой проблемы, мы должны разобраться в каком состоянии должен находиться подмодуль. Странно, но Git не предоставляет вам для этого никакой вспомогательной информации, даже SHA-1 хешей фиксаций с обеих сторон истории. К счастью, получить эту информации несложно. Если вы выполните `git diff`, то получите SHA-1 хеши фиксаций из обеих сливаемых веток.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

Так, в данном примере `eb41d76` является нашей фиксацией в подмодуле, а `c771610` – фиксацией из вышестоящего репозитория. Если мы перейдем в директорию нашего подмодуля, то он должен быть на фиксации `eb41d76`, так как операция слияния его не изменяла. Если по каким-то причинам это не так, то вы можете просто переключиться на ветку (создав ее при необходимости), указывающую на эту фиксацию.

Куда более важным является SHA-1 хеш фиксации другой стороны, которую мы должны будем спить. Вы можете либо просто выполнить слияние, указав непосредственно этот SHA-1 хеш, либо вы можете создать с ним отдельную ветку и затем уже сливать эту ветку. Мы предлагаем использовать последний вариант, хотя бы только из-за того, что сообщение фиксации слияния получается более читаемым.

Итак, перейдите в директорию нашего подмодуля, создайте ветку на основе второго SHA-1 хеша из `git diff` и выполните слияние вручную.

```
$ cd DbConnector
$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

Мы получили настоящий конфликт слияния, поэтому если мы разрешим его и создадим фиксацию, то, используя результат, сможем просто обновить основной проект.

```
$ vim src/main.c ①
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ cd .. ②
$ git diff ③
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
```

```

+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
- Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector ④

$ git commit -m "Merge Tom's Changes" ⑤
[master 10d2c60] Merge Tom's Changes

```

- ❶ Во-первых, мы разрешили конфликт
- ❷ Затем мы вернулись в директорию основного проекта
- ❸ Мы снова проверили SHA-1 хеши
- ❹ Разрешили сам конфликтовавший подмодуль
- ❺ Зафиксировали наше слияния

Это может немного запутать, но на самом деле здесь нет ничего сложного.

Интересно, что существует еще один случай, который Git обрабатывает. Если существует какая-то фиксация слияния подмодуля, которая содержит в своей истории **обе** первоначальные фиксации, то Git предложит ее вам как возможное решение. Он видит, что в какой-то момент в подмодуле, кто-то уже слил ветки, содержащие эти две фиксации, поэтому, может быть, это то, что вы хотите.

Именно поэтому выше сообщение об ошибке содержало “merge following commits not found” – Git не смог сделать **это** (найти такую фиксацию). Оно обескураживает – кто мог ожидать, что Git **пытается** сделать это?

Если удастся найти единственную приемлемую фиксацию, то вы увидите нечто подобное:

```

$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a "DbConnector"

```

```
which will accept this suggestion.
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

Здесь предполагается, что вы обновите индекс, выполнив команду `git add`, которая очищает список конфликтов и затем создает фиксацию. Хотя вы, наверное, не обязаны делать так. Вы можете также легко перейти в директорию подмодуля, просмотреть изменения, выполнить перемотку вперед до этой фиксации, выполнить необходимые проверки, а затем создать фиксацию.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forwarded to a common submodule child'
```

В этом случае выполняются те же вещи, что и в предыдущем, но так по завершению перемотки вы хотя бы сможете проверить, что все работает и вы получили правильный код в директории подмодуля.

## Полезные советы для работы с подмодулями

Существует несколько хитростей, которые могут немного упростить вашу работу с подмодулями.

### FOREACH для подмодулей

Существует команда `foreach`, которая позволяет выполнить произвольную команду в каждом подмодуле. Это может быть, действительно, полезным если у вас в одном проекте присутствует большое количество подмодулей.

Например, допустим, мы хотим начать работу над какой-то новой функциональностью или исправить какую-то ошибку и наша работа будет происходить в нескольких подмодулях. Мы можем легко прибречь все наработки во всех наших подмодулях.

```
$ git submodule foreach 'git stash'
Entering 'CryptoLibrary'
No local changes to save
Entering 'DbConnector'
Saved working directory and index state WIP on stable: 82d2ad3 Merge from origin/stable
HEAD is now at 82d2ad3 Merge from origin/stable
```

Затем мы можем создать новую ветку и переключиться на нее во всех наших подмодулях.

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Switched to a new branch 'featureA'
Entering 'DbConnector'
Switched to a new branch 'featureA'
```

Подкинем вам еще одну идею. Действительно, полезная вещь, которую вы можете сделать с помощью этой команды – это создать комплексную дельту того, что изменилось в вашем основном проекте, а также и во всех подпроектах.

```
$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)

commit_page_choice();

+
url = url_decode(url_orig);
+
/* build alias_argv */
alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
 return url_decode_internal(&url, len, NULL, &out, 0);
}
```

```
+char *url_decode(const char *url)
+{
+ return url_decode_mem(url, strlen(url));
+}
+
char *url_decode_parameter_name(const char **query)
{
 struct strbuf out = STRBUF_INIT;
```

Здесь видно, что мы определили в подмодуле функцию и вызываем ее в основном проекте. Это, конечно, упрощенный пример, но надеемся, что мы смогли донести до вас всю полезность этой функции.

## ПОЛЕЗНЫЕ ПСЕВДОНИМЫ

Возможно, вы захотите настроить псевдонимы для некоторых из этих команд, так как они могут быть, довольно, длинными, и вы не можете задать для большинства из их параметров значения по умолчанию. Мы рассмотрели настройку псевдонимов Git в “[Псевдонимы в Git](#)”, но ниже приведен пример того, что вы можете захотеть настроить, если планируете часто работать с подмодулями Git.

```
$ git config alias.sdiff '!"git diff && git submodule foreach \'git diff\'"
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'
```

Таким образом при необходимости обновить ваши подмодули вы можете просто выполнить команду `git supdate`, а для отправки изменений с проверкой зависимостей подмодулей – команду `git spush`.

## Проблемы с подмодулями

Однако, использование подмодулей не обходится без небольших проблем.

Например, переключение веток при использовании подмодулей может оказаться, довольно, запутанным. Если вы создадите новую ветку, добавите в нее подмодуль, а затем переключитесь обратно на ветку без подмодуля, то у вас все же останется директория подмодуля, как неотслеживаемая директория:

```
$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'adding crypto library'
[add-crypto 4445836] adding crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
 (use "git add <file>..." to include in what will be committed)

 CryptoLibrary/

nothing added to commit but untracked files present (use "git add" to track)
```

Удалить директорию не сложно, но может показаться странным, что она вообще оказалась там. Если вы удалите директорию и переключитесь на ветку с подмодулем, то вам потребуется выполнить `submodule update --init` для повторного создания директории.

```
$ git clean -fdx
Removing CryptoLibrary/

$ git checkout add-crypto
Switched to branch 'add-crypto'

$ ls CryptoLibrary/
Makefile includes scripts src

$ git submodule update --init
Submodule path 'CryptoLibrary': checked out 'b8ddda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile includes scripts src
```

И снова это, на самом деле, не сильно сложно, но может немного сбивать с толку.

Другая большая проблема возникает, когда люди переходят от использования поддиректорий к использованию подмодулей. Если у вас были отслеживаемые файлы в вашем проекте и вы хотите переместить их в подмодуль, то вы должны быть осторожны, иначе Git будет ругаться на вас. Предположим, у вас есть файлы в какой-то директории вашего проекта, и вы хотите переместить их в подмодуль. Если вы удалите поддиректорию, а затем выполните `submodule add`, то Git заругается на вас:

```
$ rm -rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

Сначала, вы должны удалить директорию `CryptoLibrary` из индекса. Затем вы можете добавить подмодуль:

```
$ git rm -r CryptoLibrary
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Предположим, что вы сделали это в какой-то ветке. Если вы попробуете переключиться обратно на ветку, где эти файлы все еще находятся в основном проекте, а не в подмодуле, то вы получите ошибку:

```
$ git checkout master
error: The following untracked working tree files would be overwritten by checkout
 CryptoLibrary/Makefile
 CryptoLibrary/includes/crypto.h
 ...
Please move or remove them before you can switch branches.
Aborting
```

Вы все же можете переключить ветку принудительно, используя команду `checkout -f`, но удостоверьтесь, что у вас отсутствуют

несохраненные изменения, так как они могут быть затерты этой командой.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Когда в дальнейшем вы переключитесь обратно, то по некоторой причине получите пустую директорию `CryptoLibrary` и команда `git submodule update` не сможет этого исправить. Вам может потребоваться перейти в директорию подмодуля и выполнить `git checkout ..`, чтобы вернуть все ваши файлы. Для того, чтобы запустить эту команду для нескольких подмодулей, вы можете выполнять ее, используя `submodule foreach`.

Важно отметить, что подмодули в данный момент сохраняют все служебные данные в директории `.git` основного проекта, поэтому в отличии от более старых версий Git, удаление директории подмодуля не приведет к потери каких-либо фиксаций или веток, которые у вас были.

Все эти инструменты делают подмодули довольно простым и эффективным методом работы одновременно над несколькими связанными, но пока разделенными проектами.

## Создание пакетов

Помимо рассмотренных ранее основных способов передачи данных Git по сети (HTTP, SSH и т.п.), существует еще один способ, который обычно не используется, но в некоторых случаях может быть весьма полезным.

Git умеет “упаковывать” свои данные в один файл. Это может быть полезным в разных ситуациях. Может быть, ваша сеть не работает, а вы хотите отправить изменения своим коллегам. Возможно, вы работаете откуда-то извне офиса и не имеете доступа к локальной сети по соображениям безопасности. Может быть, ваша карта беспроводной/проводной связи просто сломалась. Возможно, у вас в данный момент нет доступа к общему серверу, а вы хотите отправить кому-нибудь по электронной почте обновления, но передавать 40 коммитов с помощью `format-patch` не хотите.

В этих случаях вам может помочь команда `git bundle`. Она упакует все, что в обычной ситуации было бы отправлено по сети

командой `git push`, в бинарный файл, который вы можете передать кому-нибудь по электронной почте или поместить на флешку и затем распаковать в другом репозитории.

Рассмотрим простой пример. Допустим, у вас есть репозиторий с двумя коммитами:

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date: Wed Mar 10 07:34:10 2010 -0800

 second commit

commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date: Wed Mar 10 07:34:01 2010 -0800

 first commit
```

Если вы хотите отправить кому-нибудь этот репозиторий, но не имеете доступа на запись к общей копии репозитория или просто не хотите его настраивать, то вы можете упаковать его командой `git bundle create`.

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

В результате вы получили файл `repo.bundle`, в котором содержатся все данные, необходимые для воссоздания ветки `master` репозитория. Команде `bundle` необходимо передать список или диапазон коммитов, которые вы хотите добавить в пакет. Если вы намереваетесь использовать пакет для того, чтобы склонировать репозиторий где-нибудь еще, вы должны добавить в этот список `HEAD`, как это сделали мы.

Вы можете отправить файл `repo.bundle` кому-нибудь по электронной почте или скопировать его на USB-диск, тем самым легко решив исходную проблему.

С другой стороны, допустим, вы получили файл `hero.bundle` и хотите поработать над этим проектом. Вы можете склонировать репозиторий из бинарного файла в каталог, почти также как вы делаете это при использовании URL.

```
$ git clone hero.bundle hero
Initialized empty Git repository in /private/tmp/bundle/hero/.git/
$ cd hero
$ git log --oneline
9a466c5 second commit
b1ec324 first commit
```

Если при создании пакета вы не указали в списке ссылок HEAD, то при распаковке вам потребуется указать `-b master` или какую-либо другую ветку, включенную в пакет, иначе Git не будет знать, на какую ветку ему следует переключиться.

Теперь предположим, что вы сделали три коммита и хотите отправить их обратно в виде пакета на USB-флешке или по электронной почте.

```
$ git log --oneline
71b84da last commit - second hero
c99cf5b fourth commit - second hero
7011d3d third commit - second hero
9a466c5 second commit
b1ec324 first commit
```

Во-первых, нам нужно определить диапазон коммитов, которые мы хотим включить в пакет. В отличии от сетевых протоколов, которые сами выясняют минимальный набор данных, который нужно передать по сети, в данном случае мы должны сделать это сами вручную. В данном примере вы можете сделать, как раньше и упаковать полностью весь репозиторий, но будет лучше упаковать только изменения – три коммита, сделанные локально.

Для того, чтобы сделать это, вы должны вычислить различия. Как мы рассказывали в “Диапазоны фиксаций”, вы можете указать диапазон коммитов несколькими способами. Для того, чтобы получить три коммита из нашей основной ветки, которые отсутствовали в изначально склонированной ветке, мы можем использовать запись вида `origin/master..master` или `master ^origin/master`. Вы можете проверить ее с помощью команды `log`.

```
$ git log --oneline master ^origin/master
71b84da last commit - second гепо
c99cf5b fourth commit - second гепо
7011d3d third commit - second гепо
```

Так что теперь у нас есть список коммитов, которые мы хотим включить в пакет, давайте упакуем их. Сделаем мы это с помощью команды `git bundle create`, указав имя выходного пакета и диапазон коммитов, которые мы хотим включить в него.

```
$ git bundle create commits.bundle master ^9a466c5
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 775 bytes, done.
Total 9 (delta 0), reused 0 (delta 0)
```

В результате в нашем каталоге появился файл `commits.bundle`. Если мы отправим его нашему коллеге, то он сможет импортировать пакет в исходный репозиторий, даже если в репозитории была проделана некоторая работа параллельно с нашей.

При получении пакета коллега перед импортом его в свой репозиторий может проверить пакет, просмотрев его содержимое. Лучшей командой для этого является `bundle verify`, которая может проверить, что файл действительно является корректным Git-пакетом и что у вас есть все необходимые предки коммитов для правильного его восстановления.

```
$ git bundle verify ./commits.bundle
The bundle contains 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
The bundle requires these 1 ref
9a466c572fe88b195efd356c3f2bbeccdb504102 second commit
./commits.bundle is okay
```

Если автор создал пакет только с последними двумя коммитами, которые он сделал, а не со всеми тремя, то исходный репозиторий не сможет импортировать этот пакет, так как у него отсутствует необходимая история. В таком случае команда `verify` вернет нечто подобное:

```
$ git bundle verify ../commits-bad.bundle
еггог: Repository lacks these prerequisite commits:
еггог: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second гero
```

Однако, наш первый пакет корректен, поэтому мы можем извлечь коммиты из него. На случай если вы захотите увидеть ветки пакета, которые могут быть импортированы, существует команда для отображения только списка веток:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

Подкоманда `verify` также выводит список веток. Если цель состоит в том, чтобы увидеть, что может быть извлечено из пакета, то вы можете использовать команды `fetch` или `pull` для импорта коммитов. Ниже мы ветку `master` из пакета извлекаем в ветку `other-master` нашего репозитория:

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
 * [new branch] master -> other-master
```

Теперь мы можем увидеть, какие коммиты мы импортировали в ветку `other-master` так же, как и любые коммиты, которые мы сделали в то же время в нашей собственной ветке `master`.

```
$ git log --oneline --decorate --graph --all
* 8255d41 (HEAD, master) third commit - first гero
| * 71b84da (other-master) last commit - second гero
| * c99cf5b fourth commit - second гero
| * 7011d3d third commit - second гero
|/
* 9a466c5 second commit
* b1ec324 first commit
```

Таким образом, команда `git bundle` может быть, действительно, полезной для организации совместной работы или для выполнения сетевых операций, когда у вас нет доступа к соответствующей сети или общему репозиторию.

## Замена

Объекты в Git неизменяемы, но он предоставляет интересный способ эмулировать замену объектов в своей базе другими объектами.

Команда `replace` позволяет вам указать объект Git и сказать “каждый раз, когда встречается этот объект, заменяй его другим”. В основном, это бывает полезно для замены одного коммита в вашей истории другим.

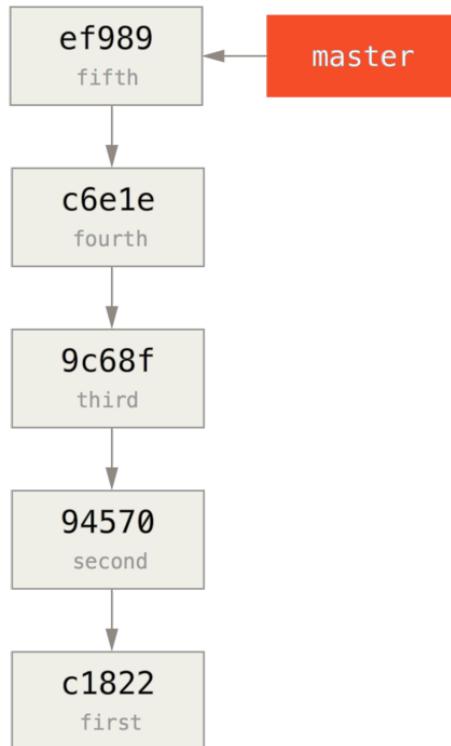
Например, допустим в вашем проекте огромная история изменений и вы хотите разбить ваш репозиторий на два – один с короткой историей для новых разработчиков, а другой с более длинной историей для людей, интересующихся анализом истории. Вы можете пересадить одну историю на другую, заменяя самый первый коммит в короткой истории последним коммитом в длинной истории. Это удобно, так как вам не придется по-настоящему изменять каждый коммит в новой истории, как это вам бы потребовалось делать в случае обычного объединения историй (так как родословная коммитов влияет на SHA-1).

Давайте испробуем как это работает, возьмем существующий репозиторий и разобьем его на два – один со свежими правками, а другой с историческими, и затем посмотрим как мы можем воссоединить их с помощью операции `replace`, не изменяя при этом значений SHA-1 в свежем репозитории.

Мы будем использовать простой репозиторий с пятью коммитами:

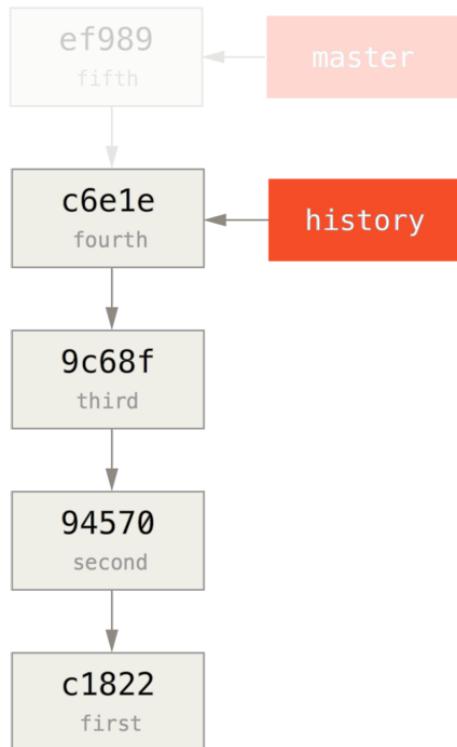
```
$ git log --oneline
ef989d8 fifth commit
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Мы хотим разбить его на два семейства историй. Одно семейство, которое начинается от первого коммита и заканчивается четвертым, будет историческим. Второе, состоящее пока только из четвертого и пятого коммитов – будет семейством со свежей историей.

**FIGURE 7-28**

Создать историческое семейство легко, мы просто создаем ветку с вершиной на нужном коммите и затем отправляем эту ветку как мастер в новый удаленный репозиторий.

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

**FIGURE 7-29**

Теперь мы можем отправить только что созданную ветвь `history` в ветку `master` нашего нового репозитория:

```
$ git remote add project-history https://github.com/schacon/project-history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
To git@github.com:schacon/project-history.git
 * [new branch] history -> master
```

Таким образом, наша история опубликована, а мы теперь зайдем более сложной частью – усечем свежую историю. Нам необходимо перекрытие, так чтобы мы смогли заменить коммит из одного части коммитом из другой, то есть мы будем обрезать историю, оставив четвертый и пятый коммиты (таким образом четвертый коммит будет входить в пересечение).

```
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

В данном случае будет полезным создать базовый коммит, содержащий инструкции о том как раскрыть историю, так другие разработчики будут знать что делать, если они столкнулись с первым коммитом урезанной истории и нуждаются в остальной истории. Итак, далее мы создадим объект заглавного коммита, представляющий нашу отправную точку с инструкциями, а затем перебазируем оставшиеся коммиты (четвертый и пятый) на этот коммит.

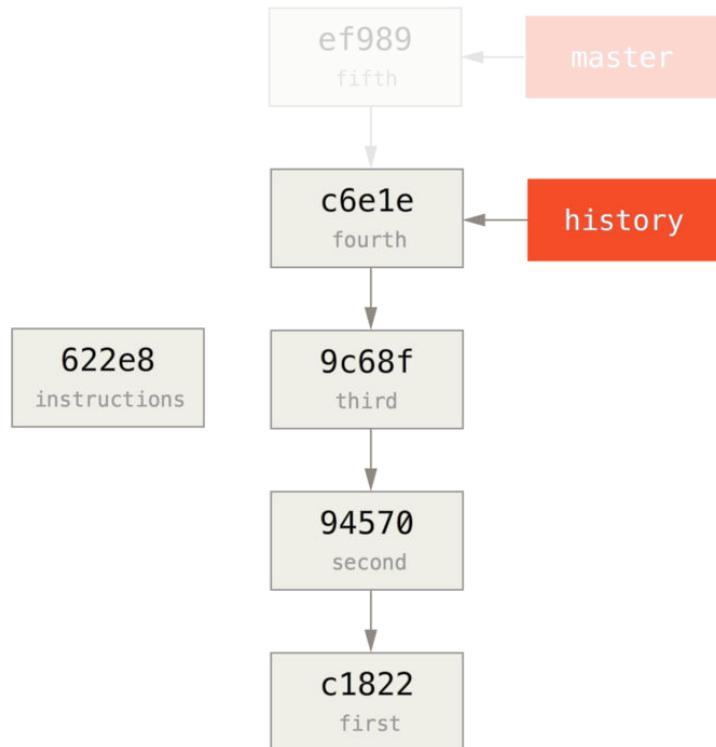
Для того, чтобы сделать это, нам нужно выбрать точку разбиения, которой для нас будет третий коммит, хеш которого 9c68fdc. Таким образом, наш базовый коммит будет основываться на этом дереве. Мы можем создать наш базовый коммит, используя команду `commit-tree`, которая просто берет дерево и возвращает SHA-1 объекта, представляющего новый сиротский коммит.

```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

---

Команда `commit-tree` входит в набор команд, которые обычно называются **сантехническими**. Это команды, которые обычно не предназначены для непосредственного использования, но вместо этого используются **другими** командами Git для выполнения небольших задач. Периодически, когда мы занимаемся странными задачами подобными текущей, эти команды позволяют нам делать низкоуровневые вещи, но все они не предназначены для повседневного использования. Вы можете прочитать больше о сантехнических командах в “**Сантехника и Фарфор**”.

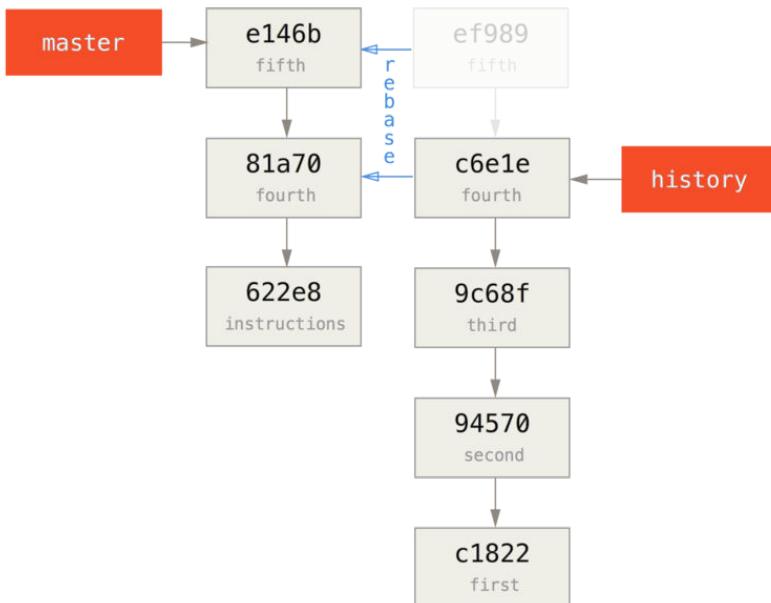
---

**FIGURE 7-30**

Хорошо. Теперь когда у нас есть базовый коммит, мы можем перебазировать нашу оставшуюся историю на этот коммит используя `git rebase --onto`. Значением аргумента `--onto` будет SHA-1 хеш коммита, которую мы только что получили от команды `commit-tree`, а перебазируемой точкой будет третий коммит (родитель первого коммита, который мы хотим сохранить, `9c68fdc`):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```

**FIGURE 7-31**



Таким образом, мы переписали нашу свежую историю поверх вспомогательного базового коммита, который теперь содержит инструкции о том, как при необходимости восстановить полную историю. Мы можем отправить эту историю в новый проект и теперь, когда люди клонируют его репозиторий, они будут видеть только два свежих коммита и после них базовый коммит с инструкциями.

Давайте представим себя на месте кого-то, кто впервые склонировал проект и хочет получить полную историю. Для получения исторических данных после клонирования усеченного репозитория, ему нужно добавить в список удаленных репозиториев исторический репозиторий и извлечь из него данные:

```
$ git clone https://github.com/schacon/project
$ cd project

$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah
```

```
$ git remote add project-history https://github.com/schacon/project-history
$ git fetch project-history
From https://github.com/schacon/project-history
 * [new branch] master -> project-history/master
```

Теперь у этого пользователя его собственные свежие коммиты будут находиться в ветке `master`, а исторические коммиты в ветке `project-history/master`.

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git log --oneline project-history/master
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Для объединения этих веток вы можете просто вызывать `git replace`, указав коммит, который вы хотите заменить, и коммит, которым вы хотите заменить первый. Так мы хотим заменить “четвертый” коммит в основной ветке “четвертым” коммитом из ветки `project-history/master`:

```
$ git replace 81a708d c6e1e95
```

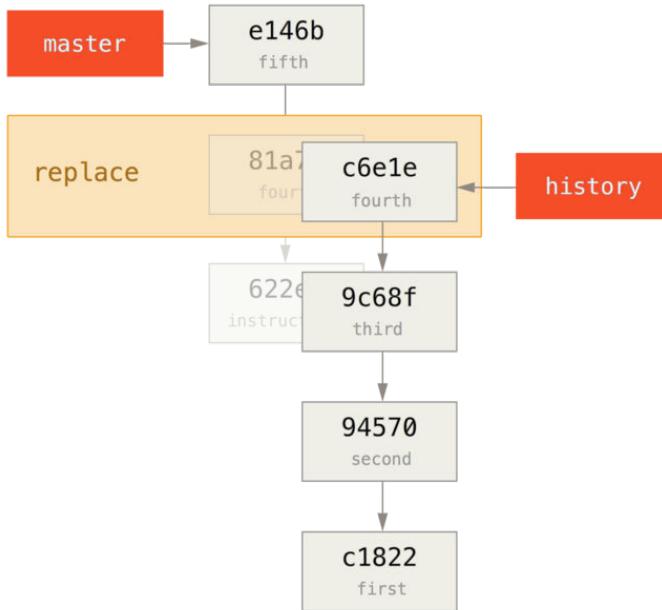
Если теперь вы посмотрите историю ветки `master`, то должны увидеть нечто подобное:

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Здорово, не правда ли? Не изменяя SHA-1 всех коммитов семейства, мы можем заменить один коммит в нашей истории

совершенно другим коммитом и все обычные утилиты (`bisect`, `blame` и т.д.) будут работать как от них это и ожидается.

**FIGURE 7-32**



Интересно, что для четвертого коммита SHA-1 хеш выводится равной 81a708d, хотя в действительности он содержит данные коммита сбе1e95, которым мы его заменили. Даже если вы выполните команду типа `cat-file`, она отобразит замененные данные:

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

Помните, что настоящим родителем коммита 81a708d был наш вспомогательный базовый коммит (622e88e), а не 9c68fdce как это отмечено здесь.

Другое интересное замечание состоит в том, что информация о произведенной замене сохранена у нас в ссылках:

```
$ git for-each-ref
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/remotes/history/master
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/replace/81a708dd0e167a3f69154
```

Следовательно можно легко поделиться заменами – для этого мы можем отправить их на наш сервер, а другие люди могут легко скачать их оттуда. Это не будет полезным в случае если вы используете `replaced` для пересадки истории (так как в этом случае все люди будут скачивать обе истории, тогда зачем мы разделяли их?), но это может быть полезным в других ситуациях.

## Хранилище учетных данных

Если для подключения к удаленным серверам вы используете SSH-транспорт, то вы можете использовать ключ без пароля, что позволит вам безопасно передавать данные без ввода логина и пароля. Однако, это невозможно при использовании HTTP-протоколов – каждое подключение требует пары логин, пароль. Все ещё сложнее для систем с двухфакторной аутентификацией, когда выражение, которое вы используете в качестве пароля, генерируется случайно и его сложно воспроизвести.

К счастью, в Git есть система управления учетными данными, которая может помочь в этом. В Git “из коробки” есть несколько опций:

- По умолчанию Git не кеширует учетные данные совсем. Каждое подключение будет запрашивать у вас логин и пароль.
- В режиме “cache” учетные данные сохраняются в памяти в течении определенного периода времени. Ни один из паролей никогда не сохраняется на диск и все они удаляются из кеша через 15 минут.
- В режиме “store” учетные данные сохраняются на неограниченное время в открытом виде в файле на диске. Это

значит что, до тех пор пока вы не измените пароль к Git-серверу, вам не потребуется больше вводить ваши учетные данные. Недостатком такого подхода является то, что ваш пароль хранится в открытом виде в файле в вашем домашнем каталоге.

- На случай если вы используете Mac, в Git есть режим “osxkeychain”, при использовании которого учетные данные хранятся в защищенном хранилище, привязанному к вашему системному аккаунту. В этом режиме учетные данные сохраняются на диск на неограниченное время, но они шифруются с использованием той же системы, с помощью которой сохраняются HTTPS-сертификаты и автозаполнения для Safari.
- В случае если вы используете Windows, вы можете установить помощник, называемый “winstore”. Он похож на “osxkeychain”, описанный выше, но для управления секретной информацией использует Windows Credential Store. Найти его можно по ссылке <https://gitcredentialstore.codeplex.com>.

Мы можем выбрать один из этих методов, изменив настройки Git:

```
$ git config --global credential.helper cache
```

Некоторые из этих помощников имеют опции. Помощник “store” может принимать аргумент `--file <path>`, который определяет где будет хранится файл с открытыми учетными данными (по умолчанию используется `~/.git-credentials`). Помощник “cache” принимает опцию `--timeout <seconds>`, которая изменяет промежуток времени, в течение которого демон остается запущенным (по умолчанию “900”, или 15 минут). Ниже приведен пример как вы можете настроить помощник “store” на использование определенного файла:

```
$ git config --global credential.helper store --file ~/.my-credentials
```

Git позволяет настраивать сразу несколько помощников. При поиске учетных данных для конкретного сервера, Git будет по порядку запрашивать у них учетный данные и остановится при получении первого ответа. При сохранении учетных данных, Git отправит их **всем** помощникам в списке, которые уже в свою очередь могут решить, что с этими данными делать. Ниже приведено как будет выглядеть `.gitconfig`, если у вас есть файл с учетными данными на

флэш-диске, но, на случай его отсутствия, вы ходите дополнительно использовать кеширование в оперативной памяти.

```
[credential]
helper = store --file /mnt/thumbdrive/.git-credentials
helper = cache --timeout 30000
```

## Под капотом

Как же это все работает? Корневой командой Git для системы помощников авторизации является `git credential`, которая принимает команду через аргумент, а все остальные входные данные через стандартный поток ввода.

Возможно, это проще понять на примере. Допустим, помощник авторизации был настроен и в нем сохранены учетные данные для `mygithost`. Ниже приведена рабочая сессия, в которой используется команда “`fill`”, вызываемая Git при попытке найти учетные данные для сервера:

```
$ git credential fill ❶
protocol=https ❷
host=mygithost
❸
protocol=https ❹
host=mygithost
username=bob
password=s3cre7
$ git credential fill ❺
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

❶ Это команда, которая начинает взаимодействие.

❷ После этого Git-credential ожидает данные из стандартного потока ввода. Мы передаем ему то, что знаем: протокол и имя сервера.

- ③ Пустая строка обозначает, что ввод закончен и система управления учетными данными должна ответить, что ей известно.
- ④ После этого Git-credential выполняет какую-то работу и выводит обнаруженную информацию.
- ⑤ Если учетные данные не найдены, Git спрашивает у пользователя логин/пароль, и выводит их обратно в задействованный поток вывода (в данном примере это одна и та же консоль).

В действительности, система управления учетными данными вызывает программы, отделенные от самого Git; какие и как зависит в том числе и от настроек `credential.helper`. Существует несколько вариантов вызова:

Настройки	Поведение
<code>foo</code>	Выполняется <code>git-credential-foo</code>
<code>foo -a --opt=bcd</code>	Выполняется <code>git-credential-foo -a --opt=bcd</code>
<code>/absolute/path/foo -xyz</code>	Выполняется <code>/absolute/path/</code> <code>foo -xyz</code>
<code>!f() { echo "pass-word=s3cr3t"; }; f</code>	Код после символа ! выполняется в шелле

Итак, помощники, описанные выше на самом деле называются `git-credential-cache`, `git-credential-store` и тд. и мы может настроить их на прием аргументов командной строки. Общая форма для этого `git-credential-foo [args] <action>`. Протокол ввода/вывода такой же как и у `git-credential`, но они используют немного другой набор операций:

- `get` запрос логина и пароля.
- `store` запрос на сохранение учетных данных в памяти помощника.
- `erase` удаляет учетные данные для заданных параметров из памяти используемого помощника.

Для операций `store` и `erase` не требуется ответа (в любом случае Git его игнорирует). Однако, для Git очень важно, что помощник ответит на операцию `get`. Если помощник не знает что-либо

полезного, он может просто завершить работу не выводя ничего, но если знает – он должен добавить к введенной информации имеющуюся у него информацию. Вывод обрабатывается как набор операций присваивания; выведенные значения заменят те, что Git знал до этого.

Ниже приведет пример, используемый ранее, но вместо `git-credential` напрямую вызывается `git-credential-store`:

```
$ git credential-store --file ~/git.store store ❶
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get ❷
protocol=https
host=mygithost

username=bob ❸
password=s3cre7
```

- ❶ Здесь мы просим `git-credential-store` сохранить некоторые учетные данные: логин “`bob`” и пароль “`s3cre7`”, которые будут использоваться при доступе к `https://mygithost`.
- ❷ Теперь мы извлечем эти учетные данные. Мы передаем часть уже известных нам параметров подключения (`https://mygithost`) и пустую строку.
- ❸ `git-credential-store` возвращает логин и пароль, которые мы сохранили ранее.

Ниже приведено содержимое файла `~/git.store`:

```
https://bob:s3cre7@mygithost
```

Это просто набор строк, каждая из которых содержит URL, включающий в себя учетные данные. Помощники `osxkeychain` и `winstore` используют формат, лежащих в их основе хранилищ, а `cache` использует его собственный формат хранения во внутренней памяти (который другие процессы прочитать не могут).

## Собственное хранилище учетных данных

Поскольку `git-credential-store` и подобные ей утилиты являются отдельными от Git программами, не сложно сделать так, чтобы *любая* программа могла быть помощником авторизации Git. Помощники предоставляемые Git покрывают наиболее распространенные варианты использования, но не все. Для примера допустим, что ваша команда имеет некоторые учетные данные, совместно используемые всей командой, например, для развертывания. Эти данные хранятся в общедоступной директории, но вы не хотите копировать их в ваше собственное хранилище учетных данных, так как они часто изменяются. Ни один из существующих помощников не покрывает этот случай; давайте посмотрим, что будет стоить написать свой собственный. Есть несколько ключевых особенностей, которым должна удовлетворять эта программа:

1. Мы должны уделить внимание только одной операции `get`; `store` и `erase` являются операциями записи, поэтому мы не будем ничего делать при их получении.
2. Формат файла с совместно используемыми учетными данными такой же как и у `git-credential-store`.
3. Расположение этого файла более-менее стандартное, но, на всякий случай, мы должны позволять пользователям передавать свой собственный путь.

Мы снова напишем расширение на Ruby, но подойдет любой язык, так как Git может использовать всё, что сможет запустить на выполнение. Ниже приведен полный исходный код нашего нового помощника авторизации:

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' ❶
OptionParser.new do |opts|
 opts.banner = 'USAGE: git-credential-read-only [options] <action>'
 opts.on('-f', '--file PATH', 'Specify path for backing store') do |argpath|
 path = File.expand_path argpath
 end
end.parse!

exit(0) unless ARGV[0].downcase == 'get' ❷
exit(0) unless File.exists? path
```

```

known = {} ③
while line = STDIN.gets
 break if line.strip == ''
 k,v = line.strip.split '=', 2
 known[k] = v
end

File.readlines(path).each do |fileline| ④
 prot,user,pass,host = fileline.scan(/^(.*?):\/\/(.*):(.*?)@(.*)$/).first
 if prot == known['protocol'] and host == known['host'] then
 puts "protocol=#{prot}"
 puts "host=#{host}"
 puts "username=#{user}"
 puts "password=#{pass}"
 exit(0)
 end
end

```

- ① Здесь мы разбираем аргументы командной строки, позволяя указывать пользователям входной файл. По умолчанию это `~/.git-credentials`.
- ② Эта программа отвечает только если операцией является `get` и файл хранилища существует.
- ③ В циклечитываются данные из стандартного ввода, до тех пор пока не будет прочитана пустая строка. Введенные данные для дальнейшего использования сохраняются в отображении `known`.
- ④ Этот цикл читает содержимое файла хранилища, выполняя поиск соответствия. Если протокол и сервер из `known` соответствуют текущей строке, программа выводит результат и завершает работу.

Мы сохраним нашего помощника как `git-credential-read-only`, расположим его в одной из директорий из `PATH` и сделаем его исполняемым. Ниже приведено на что будет похож сеанс взаимодействия:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
```

```
username=bob
password=s3cre7
```

Так как его имя начинается с “git-”, мы можем использовать простой синтаксис для настройки:

```
$ git config --global credential.helper read-only --file /mnt/shared/creds
```

Как вы видите, расширять эту систему довольно просто и это позволяет решить некоторые общие проблемы, которые могут возникнуть у вас и вашей команды.

## Заключение

Вы познакомились с множеством продвинутых инструментов, которые позволяют вам более точно управлять вашими коммитами и областью подготовленных изменений. Когда вы столкнетесь с какими-то проблемами, вы должны легко выяснить, каким коммитом они были добавлены, когда и кем. На случай, если в вашем проекте вы захотите использовать подпроекты, вы уже изучили как этого можно добиться. Таким образом, к этому моменту вы должны уметь выполнять в командной строке большинство вещей, необходимых при повседневной работе с Git, и при этом чувствовать себя уверенно.



# Настройка Git

До этого момента мы описывали основы того, как Git работает, и как его использовать. Также мы познакомились с некоторыми предоставляемыми Git'ом инструментами, которые делают его использование простым и эффективным. В этой главе мы пройдёмся по некоторым действиям, которые вы можете предпринять, чтобы заставить Git работать в нужной именно вам манере. Мы рассмотрим несколько важных настроек и систему перехватчиков (hook). С их помощью легко сделать так, чтобы Git работал именно так как вам, вашей компании или вашей группе нужно.

## Git Configuration

As you briefly saw in [Chapter 1](#), you can specify Git configuration settings with the `git config` command. One of the first things you did was set up your name and e-mail address:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Now you'll learn a few of the more interesting options that you can set in this manner to customize your Git usage.

First, a quick review: Git uses a series of configuration files to determine non-default behavior that you may want. The first place Git looks for these values is in an `/etc/gitconfig` file, which contains values for every user on the system and all of their repositories. If you pass the option `--system` to `git config`, it reads and writes from this file specifically.

The next place Git looks is the `~/.gitconfig` (or `~/.config/git/config`) file, which is specific to each user. You can make Git read and write to this file by passing the `--global` option.

Finally, Git looks for configuration values in the configuration file in the Git directory (`.git/config`) of whatever repository you’re currently using. These values are specific to that single repository.

Each of these “levels” (system, global, local) overwrites values in the previous level, so values in `.git/config` trump those in `/etc/gitconfig`, for instance.

---

Git’s configuration files are plain-text, so you can also set these values by manually editing the file and inserting the correct syntax. It’s generally easier to run the `git config` command, though.

---

## Basic Client Configuration

The configuration options recognized by Git fall into two categories: client-side and server-side. The majority of the options are client-side – configuring your personal working preferences. Many, *many* configuration options are supported, but a large fraction of them are only useful in certain edge cases. We’ll only be covering the most common and most useful here. If you want to see a list of all the options your version of Git recognizes, you can run

```
$ man git-config
```

This command lists all the available options in quite a bit of detail. You can also find this reference material at <http://git-scm.com/docs/git-config.html>.

### CORE.EDITOR

By default, Git uses whatever you’ve set as your default text editor (`$VISUAL` or `$EDITOR`) or else falls back to the `vi` editor to create and edit your commit and tag messages. To change that default to something else, you can use the `core.editor` setting:

```
$ git config --global core.editor emacs
```

Now, no matter what is set as your default shell editor, Git will fire up Emacs to edit messages.

## COMMIT.TEMPLATE

If you set this to the path of a file on your system, Git will use that file as the default message when you commit. For instance, suppose you create a template file at `~/.gitmessage.txt` that looks like this:

```
subject line
what happened
[ticket: X]
```

To tell Git to use it as the default message that appears in your editor when you run `git commit`, set the `commit.template` configuration value:

```
$ git config --global commit.template ~/.gitmessage.txt
$ git commit
```

Then, your editor will open to something like this for your placeholder commit message when you commit:

```
subject line
what happened
[ticket: X]
Please enter the commit message for your changes. Lines starting
with '#' will be ignored, and an empty message aborts the commit.
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

modified: lib/test.rb

~
~
.git/COMMIT_EDITMSG" 14L, 297C
```

If your team has a commit-message policy, then putting a template for that policy on your system and configuring Git to use it by default can help increase the chance of that policy being followed regularly.

**CORE.PAGER**

This setting determines which pager is used when Git pages output such as `log` and `diff`. You can set it to `more` or to your favorite pager (by default, it's `less`), or you can turn it off by setting it to a blank string:

```
$ git config --global core.pager ''
```

If you run that, Git will page the entire output of all commands, no matter how long they are.

**USER.SIGNINGKEY**

If you're making signed annotated tags (as discussed in “[Подпись результатов вашей работы](#)”), setting your GPG signing key as a configuration setting makes things easier. Set your key ID like so:

```
$ git config --global user.signingkey <gpg-key-id>
```

Now, you can sign tags without having to specify your key every time with the `git tag` command:

```
$ git tag -s <tag-name>
```

**CORE.EXCLUDESFILE**

You can put patterns in your project's `.gitignore` file to have Git not see them as untracked files or try to stage them when you run `git add` on them, as discussed in “[Ignoring Files](#)”.

But sometimes you want to ignore certain files for all repositories that you work with. If your computer is running Mac OS X, you're probably familiar with `.DS_Store` files. If your preferred editor is Emacs or Vim, you know about files that end with a `~`.

This setting lets you write a kind of global `.gitignore` file. If you create a `~/ .gitignore_global` file with these contents:

```
*~
.DS_Store
```

...and you run `git config --global core.excludesfile ~/.gitignore_global`, Git will never again bother you about those files.

## HELP.AUTOCORRECT

If you mistype a command, it shows you something like this:

```
$ git chekcout master
git: 'chekcout' is not a git command. See 'git --help'.

Did you mean this?
 checkout
```

Git helpfully tries to figure out what you meant, but it still refuses to do it. If you set `help.autocorrect` to 1, Git will actually run this command for you:

```
$ git chekcout master
WARNING: You called a Git command named 'chekcout', which does not exist.
Continuing under the assumption that you meant 'checkout'
in 0.1 seconds automatically...
```

Note that “0.1 seconds” business. `help.autocorrect` is actually an integer which represents tenths of a second. So if you set it to 50, Git will give you 5 seconds to change your mind before executing the autocorrected command.

## Colors in Git

Git fully supports colored terminal output, which greatly aids in visually parsing command output quickly and easily. A number of options can help you set the coloring to your preference.

## COLOR.UI

Git automatically colors most of its output, but there's a master switch if you don't like this behavior. To turn off all Git's colored terminal output, do this:

```
$ git config --global color.ui false
```

The default setting is `auto`, which colors output when it's going straight to a terminal, but omits the color-control codes when the output is redirected to a pipe or a file.

You can also set it to `always` to ignore the difference between terminals and pipes. You'll rarely want this; in most scenarios, if you want color codes in your redirected output, you can instead pass a `--color` flag to the Git command to force it to use color codes. The default setting is almost always what you'll want.

#### COLOR.\*

If you want to be more specific about which commands are colored and how, Git provides verb-specific coloring settings. Each of these can be set to `true`, `false`, or `always`:

```
color.branch
color.diff
color.interactive
color.status
```

In addition, each of these has subsettings you can use to set specific colors for parts of the output, if you want to override each color. For example, to set the meta information in your diff output to blue foreground, black background, and bold text, you can run

```
$ git config --global color.diff.meta "blue black bold"
```

You can set the color to any of the following values: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, or `white`. If you want an attribute like `bold` in the previous example, you can choose from `bold`, `dim`, `ul` (underline), `blink`, and `reverse` (swap foreground and background).

## External Merge and Diff Tools

Although Git has an internal implementation of diff, which is what we've been showing in this book, you can set up an external tool instead. You can also set up a graphical merge-conflict-resolution tool instead of having to resolve conflicts manually. We'll demonstrate setting up the Perforce Visual Merge Tool (P4Merge) to do your diffs and merge resolutions, because it's a nice graphical tool and it's free.

If you want to try this out, P4Merge works on all major platforms, so you should be able to do so. We'll use path names in the examples that work on Mac

and Linux systems; for Windows, you'll have to change `/usr/local/bin` to an executable path in your environment.

To begin, download P4Merge from <http://www.perforce.com/downloads/Perforce/>. Next, you'll set up external wrapper scripts to run your commands. We'll use the Mac path for the executable; in other systems, it will be where your `p4merge` binary is installed. Set up a merge wrapper script named `extMerge` that calls your binary with all the arguments provided:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

The diff wrapper checks to make sure seven arguments are provided and passes two of them to your merge script. By default, Git passes the following arguments to the diff program:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Because you only want the `old-file` and `new-file` arguments, you use the wrapper script to pass the ones you need.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[$# -eq 7] && /usr/local/bin/extMerge "$2" "$5"
```

You also need to make sure these tools are executable:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Now you can set up your config file to use your custom merge resolution and diff tools. This takes a number of custom settings: `merge.tool` to tell Git what strategy to use, `mergetool.<tool>.cmd` to specify how to run the command, `mergetool.<tool>.trustExitCode` to tell Git if the exit code of that program indicates a successful merge resolution or not, and `diff.external` to tell Git what command to run for diffs. So, you can either run four config commands

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
'extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"'
```

```
$ git config --global mergetool.extMerge.trustExitCode false
$ git config --global diff.external extDiff
```

or you can edit your `~/.gitconfig` file to add these lines:

```
[merge]
 tool = extMerge
[mergetool "extMerge"]
 cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
 trustExitCode = false
[diff]
 external = extDiff
```

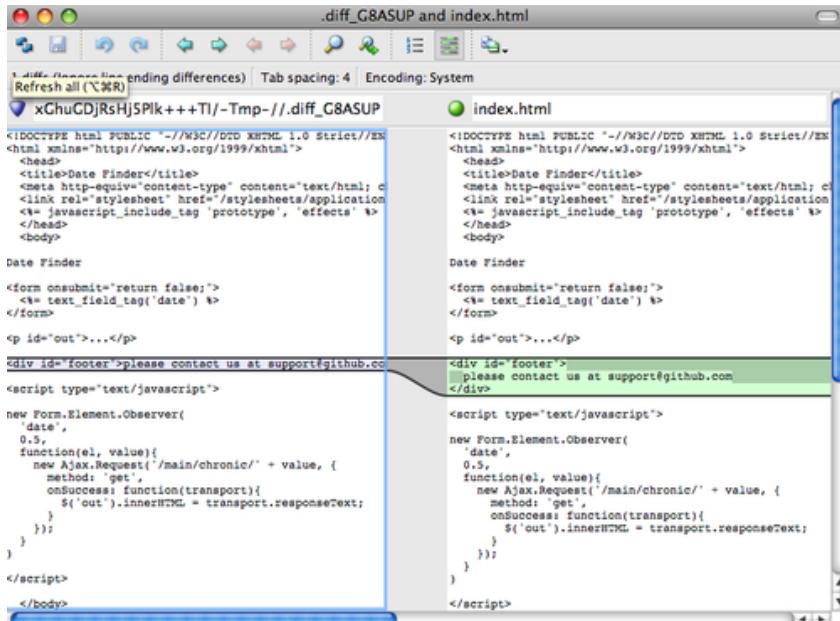
After all this is set, if you run diff commands such as this:

```
$ git diff 32d1776b1^ 32d1776b1
```

Instead of getting the diff output on the command line, Git fires up P4Merge, which looks something like this:

**FIGURE 8-1**

P4Merge.



If you try to merge two branches and subsequently have merge conflicts, you can run the command `git mergetool`; it starts P4Merge to let you resolve the conflicts through that GUI tool.

The nice thing about this wrapper setup is that you can change your diff and merge tools easily. For example, to change your `extDiff` and `extMerge` tools to run the KDiff3 tool instead, all you have to do is edit your `extMerge` file:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Now, Git will use the KDiff3 tool for diff viewing and merge conflict resolution.

Git comes preset to use a number of other merge-resolution tools without your having to set up the cmd configuration. To see a list of the tools it supports, try this:

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
 emerge
 gvimdiff
 gvimdiff2
 opendiff
 p4merge
 vimdiff
 vimdiff2
```

The following tools are valid, but not currently available:

```
 araxis
 bc3
 codecompare
 deltawalker
 diffmerge
 diffuse
 ecmerge
 kdiff3
 meld
 tkdiff
 tortoisemerge
 xxdiff
```

Some of the tools listed above only work in a windowed environment. If run in a terminal-only session, they will fail.

If you're not interested in using KDiff3 for diff but rather want to use it just for merge resolution, and the kdiff3 command is in your path, then you can run

```
$ git config --global merge.tool kdiff3
```

If you run this instead of setting up the extMerge and extDiff files, Git will use KDiff3 for merge resolution and the normal Git diff tool for diffs.

## Formatting and Whitespace

Formatting and whitespace issues are some of the more frustrating and subtle problems that many developers encounter when collaborating, especially cross-platform. It's very easy for patches or other collaborated work to introduce subtle whitespace changes because editors silently introduce them, and if your files ever touch a Windows system, their line endings might be replaced. Git has a few configuration options to help with these issues.

### CORE.AUTOCLRF

If you're programming on Windows and working with people who are not (or vice-versa), you'll probably run into line-ending issues at some point. This is because Windows uses both a carriage-return character and a linefeed character for newlines in its files, whereas Mac and Linux systems use only the linefeed character. This is a subtle but incredibly annoying fact of cross-platform work; many editors on Windows silently replace existing LF-style line endings with CRLF, or insert both line-ending characters when the user hits the enter key.

Git can handle this by auto-converting CRLF line endings into LF when you add a file to the index, and vice versa when it checks out code onto your filesystem. You can turn on this functionality with the `core.autocrlf` setting. If you're on a Windows machine, set it to `true` – this converts LF endings into CRLF when you check out code:

```
$ git config --global core.autocrlf true
```

If you're on a Linux or Mac system that uses LF line endings, then you don't want Git to automatically convert them when you check out files; however, if a file with CRLF endings accidentally gets introduced, then you may want Git to fix it. You can tell Git to convert CRLF to LF on commit but not the other way around by setting `core.autocrlf` to `input`:

```
$ git config --global core.autocrlf input
```

This setup should leave you with CRLF endings in Windows checkouts, but LF endings on Mac and Linux systems and in the repository.

If you're a Windows programmer doing a Windows-only project, then you can turn off this functionality, recording the carriage returns in the repository by setting the config value to `false`:

```
$ git config --global core.autocrlf false
```

## CORE.WHITESPACE

Git comes preset to detect and fix some whitespace issues. It can look for six primary whitespace issues – three are enabled by default and can be turned off, and three are disabled by default but can be activated.

The ones that are turned on by default are `blank-at-eol`, which looks for spaces at the end of a line; `blank-at-eof`, which notices blank lines at the end of a file; and `space-before-tab`, which looks for spaces before tabs at the beginning of a line.

The three that are disabled by default but can be turned on are `indent-with-non-tab`, which looks for lines that begin with spaces instead of tabs (and is controlled by the `tabwidth` option); `tab-in-indent`, which watches for tabs in the indentation portion of a line; and `cr-at-eol`, which tells Git that carriage returns at the end of lines are OK.

You can tell Git which of these you want enabled by setting `core.whitespace` to the values you want on or off, separated by commas. You can disable settings by either leaving them out of the setting string or prepending a `-` in front of the value. For example, if you want all but `cr-at-eol` to be set, you can do this:

```
$ git config --global core.whitespace \
 trailing-space,space-before-tab,indent-with-non-tab
```

Git will detect these issues when you run a `git diff` command and try to color them so you can possibly fix them before you commit. It will also use these values to help you when you apply patches with `git apply`. When you're applying patches, you can ask Git to warn you if it's applying patches with the specified whitespace issues:

```
$ git apply --whitespace=warn <patch>
```

Or you can have Git try to automatically fix the issue before applying the patch:

```
$ git apply --whitespace=fix <patch>
```

These options apply to the `git rebase` command as well. If you've committed whitespace issues but haven't yet pushed upstream, you can run `git rebase --whitespace=fix` to have Git automatically fix whitespace issues as it's rewriting the patches.

## Server Configuration

Not nearly as many configuration options are available for the server side of Git, but there are a few interesting ones you may want to take note of.

### RECEIVE.FSCKOBJECTS

Git is capable of making sure every object received during a push still matches its SHA-1 checksum and points to valid objects. However, it doesn't do this by default; it's a fairly expensive operation, and might slow down the operation, especially on large repositories or pushes. If you want Git to check object consistency on every push, you can force it to do so by setting `receive.fsckObjects` to true:

```
$ git config --system receive.fsckObjects true
```

Now, Git will check the integrity of your repository before each push is accepted to make sure faulty (or malicious) clients aren't introducing corrupt data.

### RECEIVE.DENYNONFASTFORWARDS

If you rebase commits that you've already pushed and then try to push again, or otherwise try to push a commit to a remote branch that doesn't contain the commit that the remote branch currently points to, you'll be denied. This is generally good policy; but in the case of the rebase, you may determine that

you know what you’re doing and can force-update the remote branch with a `-f` flag to your push command.

To tell Git to refuse force-pushes, set `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

The other way you can do this is via server-side receive hooks, which we’ll cover in a bit. That approach lets you do more complex things like deny non-fast-forwards to a certain subset of users.

#### RECEIVE.DENYDELETES

One of the workarounds to the `denyNonFastForwards` policy is for the user to delete the branch and then push it back up with the new reference. To avoid this, set `receive.denyDeletes` to true:

```
$ git config --system receive.denyDeletes true
```

This denies any deletion of branches or tags – no user can do it. To remove remote branches, you must remove the ref files from the server manually. There are also more interesting ways to do this on a per-user basis via ACLs, as you’ll learn in “[An Example Git-Enforced Policy](#)”.

## Git Attributes

Some of these settings can also be specified for a path, so that Git applies those settings only for a subdirectory or subset of files. These path-specific settings are called Git attributes and are set either in a `.gitattributes` file in one of your directories (normally the root of your project) or in the `.git/info/attributes` file if you don’t want the attributes file committed with your project.

Using attributes, you can do things like specify separate merge strategies for individual files or directories in your project, tell Git how to diff non-text files, or have Git filter content before you check it into or out of Git. In this section, you’ll learn about some of the attributes you can set on your paths in your Git project and see a few examples of using this feature in practice.

## Binary Files

One cool trick for which you can use Git attributes is telling Git which files are binary (in cases it otherwise may not be able to figure out) and giving Git special instructions about how to handle those files. For instance, some text files may be machine generated and not diffable, whereas some binary files can be diffed. You'll see how to tell Git which is which.

### IDENTIFYING BINARY FILES

Some files look like text files but for all intents and purposes are to be treated as binary data. For instance, Xcode projects on the Mac contain a file that ends in `.pbxproj`, which is basically a JSON (plain-text Javascript data format) dataset written out to disk by the IDE, which records your build settings and so on. Although it's technically a text file (because it's all UTF-8), you don't want to treat it as such because it's really a lightweight database – you can't merge the contents if two people change it, and diffs generally aren't helpful. The file is meant to be consumed by a machine. In essence, you want to treat it like a binary file.

To tell Git to treat all `.pbxproj` files as binary data, add the following line to your `.gitattributes` file:

```
*.pbxproj binary
```

Now, Git won't try to convert or fix CRLF issues; nor will it try to compute or print a diff for changes in this file when you run `git show` or `git diff` on your project.

### DIFFING BINARY FILES

You can also use the Git attributes functionality to effectively diff binary files. You do this by telling Git how to convert your binary data to a text format that can be compared via the normal diff.

First, you'll use this technique to solve one of the most annoying problems known to humanity: version-controlling Microsoft Word documents. Everyone knows that Word is the most horrific editor around, but oddly, everyone still uses it. If you want to version-control Word documents, you can stick them in a Git repository and commit every once in a while; but what good does that do? If you run `git diff` normally, you only see something like this:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
```

```
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

You can't directly compare two versions unless you check them out and scan them manually, right? It turns out you can do this fairly well using Git attributes. Put the following line in your `.gitattributes` file:

```
*.docx diff=word
```

This tells Git that any file that matches this pattern (`.docx`) should use the “word” filter when you try to view a diff that contains changes. What is the “word” filter? You have to set it up. Here you'll configure Git to use the `docx2txt` program to convert Word documents into readable text files, which it will then diff properly.

First, you'll need to install `docx2txt`; you can download it from <http://docx2txt.sourceforge.net>. Follow the instructions in the `INSTALL` file to put it somewhere your shell can find it. Next, you'll write a wrapper script to convert output to the format Git expects. Create a file that's somewhere in your path called `docx2txt`, and add these contents:

```
#!/bin/bash
docx2txt.pl $1 -
```

Don't forget to `chmod a+x` that file. Finally, you can configure Git to use this script:

```
$ git config diff.word.textconv docx2txt
```

Now Git knows that if it tries to do a diff between two snapshots, and any of the files end in `.docx`, it should run those files through the “word” filter, which is defined as the `docx2txt` program. This effectively makes nice text-based versions of your Word files before attempting to diff them.

Here's an example: Chapter 1 of this book was converted to Word format and committed in a Git repository. Then a new paragraph was added. Here's what `git diff` shows:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 0b013ca..ba25db5 100644
--- a/chapter1.docx
```

```

+++ b/chapter1.docx
@@ -2,6 +2,7 @@
 This chapter will be about getting started with Git. We will begin at the beginning.
 1.1. About Version Control
 What is "version control", and why should you care? Version control is a system to
 +Testing: 1, 2, 3.
 If you are a graphic or web designer and want to keep every version of an image or
 1.1.1. Local Version Control Systems
 Many people's version-control method of choice is to copy files into another dire

```

Git successfully and succinctly tells us that we added the string “Testing: 1, 2, 3.”, which is correct. It’s not perfect – formatting changes wouldn’t show up here – but it certainly works.

Another interesting problem you can solve this way involves diffing image files. One way to do this is to run image files through a filter that extracts their EXIF information – metadata that is recorded with most image formats. If you download and install the `exiftool` program, you can use it to convert your images into text about the metadata, so at least the diff will show you a textual representation of any changes that happened:

```

$ echo '*.*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool

```

If you replace an image in your project and run `git diff`, you see something like this:

```

diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
 ExifTool Version Number : 7.74
 -File Size : 70 kB
 -File Modification Date/Time : 2009:04:21 07:02:45-07:00
 +File Size : 94 kB
 +File Modification Date/Time : 2009:04:21 07:02:43-07:00
 File Type : PNG
 MIME Type : image/png
 -Image Width : 1058
 -Image Height : 889
 +Image Width : 1056
 +Image Height : 827
 Bit Depth : 8
 Color Type : RGB with Alpha

```

You can easily see that the file size and image dimensions have both changed.

## Keyword Expansion

SVN- or CVS-style keyword expansion is often requested by developers used to those systems. The main problem with this in Git is that you can't modify a file with information about the commit after you've committed, because Git checksums the file first. However, you can inject text into a file when it's checked out and remove it again before it's added to a commit. Git attributes offers you two ways to do this.

First, you can inject the SHA-1 checksum of a blob into an `$Id$` field in the file automatically. If you set this attribute on a file or set of files, then the next time you check out that branch, Git will replace that field with the SHA-1 of the blob. It's important to notice that it isn't the SHA-1 of the commit, but of the blob itself:

```
$ echo '*.txt ident' >> .gitattributes
$ echo 'Id' > test.txt
```

The next time you check out this file, Git injects the SHA-1 of the blob:

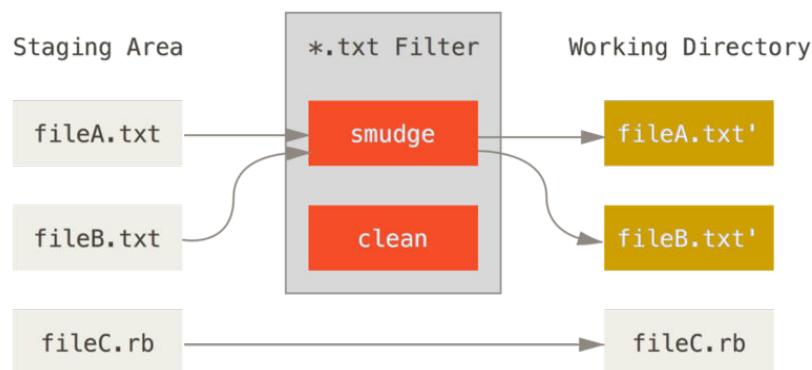
```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

However, that result is of limited use. If you've used keyword substitution in CVS or Subversion, you can include a timestamp – the SHA-1 isn't all that helpful, because it's fairly random and you can't tell if one SHA-1 is older or newer than another just by looking at them.

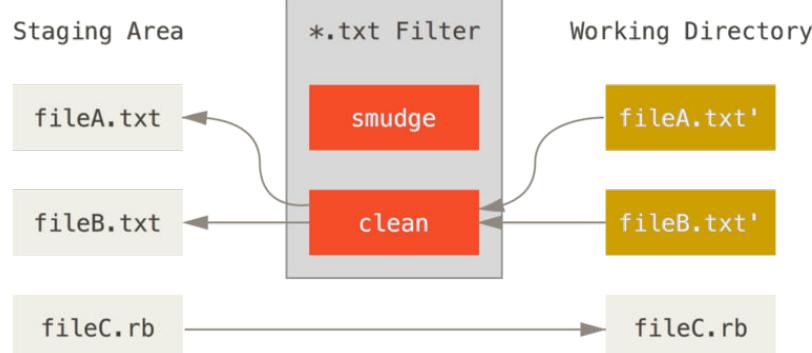
It turns out that you can write your own filters for doing substitutions in files on commit/checkout. These are called “clean” and “smudge” filters. In the `.gitattributes` file, you can set a filter for particular paths and then set up scripts that will process files just before they're checked out (“smudge”, see [Figure 8-2](#)) and just before they're staged (“clean”, see [Figure 8-3](#)). These filters can be set to do all sorts of fun things.

**FIGURE 8-2**

The “smudge” filter is run on checkout.

**FIGURE 8-3**

The “clean” filter is run when files are staged.



The original commit message for this feature gives a simple example of running all your C source code through the `indent` program before committing. You can set it up by setting the `filter` attribute in your `.gitattributes` file to filter `*.c` files with the “`indent`” filter:

```
*.c filter=indent
```

Then, tell Git what the “`indent`” filter does on smudge and clean:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

In this case, when you commit files that match `*.c`, Git will run them through the `indent` program before it stages them and then run them through the `cat` program before it checks them back out onto disk. The `cat` program does essentially nothing: it spits out the same data that it comes in. This combination effectively filters all C source code files through `indent` before committing.

Another interesting example gets `$Date$` keyword expansion, RCS style. To do this properly, you need a small script that takes a filename, figures out the last commit date for this project, and inserts the date into the file. Here is a small Ruby script that does that:

```
#!/usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

All the script does is get the latest commit date from the `git log` command, stick that into any `$Date$` strings it sees in stdin, and print the results – it should be simple to do in whatever language you’re most comfortable in. You can name this file `expand_date` and put it in your path. Now, you need to set up a filter in Git (call it `dater`) and tell it to use your `expand_date` filter to smudge the files on checkout. You’ll use a Perl expression to clean that up on commit:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\$\$Date[^\\\$]*\\\$/\$\$Date\\\$/"'
```

This Perl snippet strips out anything it sees in a `$Date$` string, to get back to where you started. Now that your filter is ready, you can test it by setting up a file with your `$Date$` keyword and then setting up a Git attribute for that file that engages the new filter:

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

If you commit those changes and check out the file again, you see the keyword properly substituted:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
```

```
$ git checkout date_test.txt
$ cat date_test.txt
$Date: Tue Apr 21 07:26:52 2009 -0700$
```

You can see how powerful this technique can be for customized applications. You have to be careful, though, because the `.gitattributes` file is committed and passed around with the project, but the driver (in this case, `dater`) isn't, so it won't work everywhere. When you design these filters, they should be able to fail gracefully and have the project still work properly.

## Exporting Your Repository

Git attribute data also allows you to do some interesting things when exporting an archive of your project.

### EXPORT-IGNORE

You can tell Git not to export certain files or directories when generating an archive. If there is a subdirectory or file that you don't want to include in your archive file but that you do want checked into your project, you can determine those files via the `export-ignore` attribute.

For example, say you have some test files in a `test/` subdirectory, and it doesn't make sense to include them in the tarball export of your project. You can add the following line to your Git attributes file:

```
test/ export-ignore
```

Now, when you run `git archive` to create a tarball of your project, that directory won't be included in the archive.

### EXPORT-SUBST

Another thing you can do for your archives is some simple keyword substitution. Git lets you put the string `$Format:$` in any file with any of the `--pretty=format` formatting shortcodes, many of which you saw in Chapter 2. For instance, if you want to include a file named `LAST_COMMIT` in your project, and the last commit date was automatically injected into it when `git archive` ran, you can set up the file like this:

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
```

```
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

When you run `git archive`, the contents of that file when people open the archive file will look like this:

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

## Merge Strategies

You can also use Git attributes to tell Git to use different merge strategies for specific files in your project. One very useful option is to tell Git to not try to merge specific files when they have conflicts, but rather to use your side of the merge over someone else's.

This is helpful if a branch in your project has diverged or is specialized, but you want to be able to merge changes back in from it, and you want to ignore certain files. Say you have a database settings file called `database.xml` that is different in two branches, and you want to merge in your other branch without messing up the database file. You can set up an attribute like this:

```
database.xml merge=ours
```

And then define a dummy `ours` merge strategy with:

```
$ git config --global merge.ours.driver true
```

If you merge in the other branch, instead of having merge conflicts with the `database.xml` file, you see something like this:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

In this case, `database.xml` stays at whatever version you originally had.

## Git Hooks

Like many other Version Control Systems, Git has a way to fire off custom scripts when certain important actions occur. There are two groups of these hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, while server-side hooks run on network operations such as receiving pushed commits. You can use these hooks for all sorts of reasons

### Installing a Hook

The hooks are all stored in the `hooks` subdirectory of the Git directory. In most projects, that's `.git/hooks`. When you initialize a new repository with `git init`, Git populates the `hooks` directory with a bunch of example scripts, many of which are useful by themselves; but they also document the input values of each script. All the examples are written as shell scripts, with some Perl thrown in, but any properly named executable scripts will work fine – you can write them in Ruby or Python or what have you. If you want to use the bundled hook scripts, you'll have to rename them; their file names all end with `.sample`.

To enable a hook script, put a file in the `hooks` subdirectory of your Git directory that is named appropriately and is executable. From that point forward, it should be called. We'll cover most of the major hook filenames here.

### Client-Side Hooks

There are a lot of client-side hooks. This section splits them into committing-workflow hooks, e-mail-workflow scripts, and everything else.

---

It's important to note that client-side hooks are **not** copied when you clone a repository. If your intent with these scripts is to enforce a policy, you'll probably want to do that on the server side; see the example in “An Example Git-Enforced Policy”.

---

#### COMMITTING-WORKFLOW HOOKS

The first four hooks have to do with the committing process.

The `pre-commit` hook is run first, before you even type in a commit message. It's used to inspect the snapshot that's about to be committed, to see if you've forgotten something, to make sure tests run, or to examine whatever you need to inspect in the code. Exiting non-zero from this hook aborts the commit, although you can bypass it with `git commit --no-verify`. You can

do things like check for code style (run `lint` or something equivalent), check for trailing whitespace (the default hook does exactly this), or check for appropriate documentation on new methods.

The `prepare-commit-msg` hook is run before the commit message editor is fired up but after the default message is created. It lets you edit the default message before the commit author sees it. This hook takes a few parameters: the path to the file that holds the commit message so far, the type of commit, and the commit SHA-1 if this is an amended commit. This hook generally isn't useful for normal commits; rather, it's good for commits where the default message is auto-generated, such as templated commit messages, merge commits, squashed commits, and amended commits. You may use it in conjunction with a commit template to programmatically insert information.

The `commit-msg` hook takes one parameter, which again is the path to a temporary file that contains the commit message written by the developer. If this script exits non-zero, Git aborts the commit process, so you can use it to validate your project state or commit message before allowing a commit to go through. In the last section of this chapter, We'll demonstrate using this hook to check that your commit message is conformant to a required pattern.

After the entire commit process is completed, the `post-commit` hook runs. It doesn't take any parameters, but you can easily get the last commit by running `git log -1 HEAD`. Generally, this script is used for notification or something similar.

## E-MAIL WORKFLOW HOOKS

You can set up three client-side hooks for an e-mail-based workflow. They're all invoked by the `git am` command, so if you aren't using that command in your workflow, you can safely skip to the next section. If you're taking patches over e-mail prepared by `git format-patch`, then some of these may be helpful to you.

The first hook that is run is `applypatch-msg`. It takes a single argument: the name of the temporary file that contains the proposed commit message. Git aborts the patch if this script exits non-zero. You can use this to make sure a commit message is properly formatted, or to normalize the message by having the script edit it in place.

The next hook to run when applying patches via `git am` is `pre-applypatch`. Somewhat confusingly, it is run *after* the patch is applied but before a commit is made, so you can use it to inspect the snapshot before making the commit. You can run tests or otherwise inspect the working tree with this script. If something is missing or the tests don't pass, exiting non-zero aborts the `git am` script without committing the patch.

The last hook to run during a `git am` operation is `post-applypatch`, which runs after the commit is made. You can use it to notify a group or the author of the patch you pulled in that you've done so. You can't stop the patching process with this script.

## OTHER CLIENT HOOKS

The `pre-rebase` hook runs before you rebase anything and can halt the process by exiting non-zero. You can use this hook to disallow rebasing any commits that have already been pushed. The example `pre-rebase` hook that Git installs does this, although it makes some assumptions that may not match with your workflow.

The `post-rewrite` hook is run by commands that replace commits, such as `git commit --amend` and `git rebase` (though not by `git filter-branch`). Its single argument is which command triggered the rewrite, and it receives a list of rewrites on `stdin`. This hook has many of the same uses as the `post-checkout` and `post-merge` hooks.

After you run a successful `git checkout`, the `post-checkout` hook runs; you can use it to set up your working directory properly for your project environment. This may mean moving in large binary files that you don't want source controlled, auto-generating documentation, or something along those lines.

The `post-merge` hook runs after a successful `merge` command. You can use it to restore data in the working tree that Git can't track, such as permissions data. This hook can likewise validate the presence of files external to Git control that you may want copied in when the working tree changes.

The `pre-push` hook runs during `git push`, after the remote refs have been updated but before any objects have been transferred. It receives the name and location of the remote as parameters, and a list of to-be-updated refs through `stdin`. You can use it to validate a set of ref updates before a push occurs (a non-zero exit code will abort the push).

Git occasionally does garbage collection as part of its normal operation, by invoking `git gc --auto`. The `pre-auto-gc` hook is invoked just before the garbage collection takes place, and can be used to notify you that this is happening, or to abort the collection if now isn't a good time.

## Server-Side Hooks

In addition to the client-side hooks, you can use a couple of important server-side hooks as a system administrator to enforce nearly any kind of policy for your project. These scripts run before and after pushes to the server. The pre

hooks can exit non-zero at any time to reject the push as well as print an error message back to the client; you can set up a push policy that's as complex as you wish.

#### PRE-RECEIVE

The first script to run when handling a push from a client is `pre-receive`. It takes a list of references that are being pushed from stdin; if it exits non-zero, none of them are accepted. You can use this hook to do things like make sure none of the updated references are non-fast-forwards, or to do access control for all the refs and files they're modifying with the push.

#### UPDATE

The update script is very similar to the `pre-receive` script, except that it's run once for each branch the pusher is trying to update. If the pusher is trying to push to multiple branches, `pre-receive` runs only once, whereas update runs once per branch they're pushing to. Instead of reading from stdin, this script takes three arguments: the name of the reference (branch), the SHA-1 that reference pointed to before the push, and the SHA-1 the user is trying to push. If the update script exits non-zero, only that reference is rejected; other references can still be updated.

#### POST-RECEIVE

The `post-receive` hook runs after the entire process is completed and can be used to update other services or notify users. It takes the same stdin data as the `pre-receive` hook. Examples include e-mailing a list, notifying a continuous integration server, or updating a ticket-tracking system – you can even parse the commit messages to see if any tickets need to be opened, modified, or closed. This script can't stop the push process, but the client doesn't disconnect until it has completed, so be careful if you try to do anything that may take a long time.

## An Example Git-Enforced Policy

In this section, you'll use what you've learned to establish a Git workflow that checks for a custom commit message format, and allows only certain users to modify certain subdirectories in a project. You'll build client scripts that help the developer know if their push will be rejected and server scripts that actually enforce the policies.

The scripts we'll show are written in Ruby; partly because of our intellectual inertia, but also because Ruby is easy to read, even if you can't necessarily write it. However, any language will work – all the sample hook scripts distributed with Git are in either Perl or Bash, so you can also see plenty of examples of hooks in those languages by looking at the samples.

## Server-Side Hook

All the server-side work will go into the `update` file in your `hooks` directory. The `update` hook runs once per branch being pushed and takes three arguments:

- The name of the reference being pushed to
- The old revision where that branch was
- The new revision being pushed

You also have access to the user doing the pushing if the push is being run over SSH. If you've allowed everyone to connect with a single user (like "git") via public-key authentication, you may have to give that user a shell wrapper that determines which user is connecting based on the public key, and set an environment variable accordingly. Here we'll assume the connecting user is in the `$USER` environment variable, so your `update` script begins by gathering all the information you need:

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev = ARGV[1]
$newrev = ARGV[2]
$user = ENV['USER']

puts "Enforcing Policies..."
puts "($refname) ($oldrev[0..6]) ($newrev[0..6])"
```

Yes, those are global variables. Don't judge – it's easier to demonstrate this way.

## ENFORCING A SPECIFIC COMMIT-MESSAGE FORMAT

Your first challenge is to enforce that each commit message adheres to a particular format. Just to have a target, assume that each message has to include a string that looks like "ref: 1234" because you want each commit to link to a work item in your ticketing system. You must look at each commit being pushed up, see if that string is in the commit message, and, if the string is absent from any of the commits, exit non-zero so the push is rejected.

You can get a list of the SHA-1 values of all the commits that are being pushed by taking the `$newrev` and `$oldrev` values and passing them to a Git plumbing command called `git rev-list`. This is basically the `git log` command, but by default it prints out only the SHA-1 values and no other information. So, to get a list of all the commit SHA-1s introduced between one commit SHA-1 and another, you can run something like this:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cf0e475
```

You can take that output, loop through each of those commit SHA-1s, grab the message for it, and test that message against a regular expression that looks for a pattern.

You have to figure out how to get the commit message from each of these commits to test. To get the raw commit data, you can use another plumbing command called `git cat-file`. We'll go over all these plumbing commands in detail in [Chapter 10](#); but for now, here's what that command gives you:

```
$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

A simple way to get the commit message from a commit when you have the SHA-1 value is to go to the first blank line and take everything after that. You can do so with the `sed` command on Unix systems:

```
$ git cat-file commit ca82a6 | sed '1,/^\$/d'
changed the version number
```

You can use that incantation to grab the commit message from each commit that is trying to be pushed and exit if you see anything that doesn't match. To exit the script and reject the push, exit non-zero. The whole method looks like this:

```
$regex = /\[ref: (\d+)\]/

enforced custom commit message format
def check_message_format
 missed_revs = `git rev-list ${oldrev}..${newrev}`.split("\n")
 missed_revs.each do |rev|
 message = `git cat-file commit ${rev} | sed '1,/^\$/d'`
 if !$regex.match(message)
 puts "[POLICY] Your message is not formatted correctly"
 exit 1
 end
 end
end
check_message_format
```

Putting that in your update script will reject updates that contain commits that have messages that don't adhere to your rule.

## ENFORCING A USER-BASED ACL SYSTEM

Suppose you want to add a mechanism that uses an access control list (ACL) that specifies which users are allowed to push changes to which parts of your projects. Some people have full access, and others can only push changes to certain subdirectories or specific files. To enforce this, you'll write those rules to a file named `acl` that lives in your bare Git repository on the server. You'll have the update hook look at those rules, see what files are being introduced for all the commits being pushed, and determine whether the user doing the push has access to update all those files.

The first thing you'll do is write your ACL. Here you'll use a format very much like the CVS ACL mechanism: it uses a series of lines, where the first field is `avail` or `unavail`, the next field is a comma-delimited list of the users to which the rule applies, and the last field is the path to which the rule applies (blank meaning open access). All of these fields are delimited by a pipe (|) character.

In this case, you have a couple of administrators, some documentation writers with access to the `doc` directory, and one developer who only has access to the `lib` and `tests` directories, and your ACL file looks like this:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

You begin by reading this data into a structure that you can use. In this case, to keep the example simple, you'll only enforce the `avail` directives. Here is a

method that gives you an associative array where the key is the user name and the value is an array of paths to which the user has write access:

```
def get_acl_access_data(acl_file)
 # read in ACL data
 acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
 access = {}
 acl_file.each do |line|
 avail, users, path = line.split('||')
 next unless avail == 'avail'
 users.split(',').each do |user|
 access[user] ||= []
 access[user] << path
 end
 end
 access
end
```

On the ACL file you looked at earlier, this `get_acl_access_data` method returns a data structure that looks like this:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Now that you have the permissions sorted out, you need to determine what paths the commits being pushed have modified, so you can make sure the user who's pushing has access to all of them.

You can pretty easily see what files have been modified in a single commit with the `--name-only` option to the `git log` command (mentioned briefly in Chapter 2):

```
$ git log -1 --name-only --pretty=format:'' 9f585d
README
lib/test.rb
```

If you use the ACL structure returned from the `get_acl_access_data` method and check it against the listed files in each of the commits, you can determine whether the user has access to push all of their commits:

```

only allows certain users to modify certain subdirectories in a project
def check_directory_perms
 access = get_acl_access_data('acl')

 # see if anyone is trying to push something they can't
 new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
 new_commits.each do |rev|
 files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
 files_modified.each do |path|
 next if path.size == 0
 has_file_access = false
 access[$user].each do |access_path|
 if !access_path # user has access to everything
 || (path.start_with? access_path) # access to this path
 has_file_access = true
 end
 end
 if !has_file_access
 puts "[POLICY] You do not have access to push to #{path}"
 exit 1
 end
 end
 end
end

check_directory_perms

```

You get a list of new commits being pushed to your server with `git rev-list`. Then, for each of those commits, you find which files are modified and make sure the user who's pushing has access to all the paths being modified.

Now your users can't push any commits with badly formed messages or with modified files outside of their designated paths.

## TESTING IT OUT

If you run `chmod u+x .git/hooks/update`, which is the file into which you should have put all this code, and then try to push a commit with a non-compliant message, you get something like this:

```

$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)

```

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

There are a couple of interesting things here. First, you see this where the hook starts running.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Remember that you printed that out at the very beginning of your update script. Anything your script echoes to `stdout` will be transferred to the client.

The next thing you'll notice is the error message.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

The first line was printed out by you, the other two were Git telling you that the update script exited non-zero and that is what is declining your push. Lastly, you have this:

```
To git@gitserver:project.git
! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

You'll see a remote rejected message for each reference that your hook declined, and it tells you that it was declined specifically because of a hook failure.

Furthermore, if someone tries to edit a file they don't have access to and push a commit containing it, they will see something similar. For instance, if a documentation author tries to push a commit modifying something in the `lib` directory, they see

```
[POLICY] You do not have access to push to lib/test.rb
```

From now on, as long as that `update` script is there and executable, your repository will never have a commit message without your pattern in it, and your users will be sandboxed.

## Client-Side Hooks

The downside to this approach is the whining that will inevitably result when your users' commit pushes are rejected. Having their carefully crafted work rejected at the last minute can be extremely frustrating and confusing; and furthermore, they will have to edit their history to correct it, which isn't always for the faint of heart.

The answer to this dilemma is to provide some client-side hooks that users can run to notify them when they're doing something that the server is likely to reject. That way, they can correct any problems before committing and before those issues become more difficult to fix. Because hooks aren't transferred with a clone of a project, you must distribute these scripts some other way and then have your users copy them to their `.git/hooks` directory and make them executable. You can distribute these hooks within the project or in a separate project, but Git won't set them up automatically.

To begin, you should check your commit message just before each commit is recorded, so you know the server won't reject your changes due to badly formatted commit messages. To do this, you can add the `commit-msg` hook. If you have it read the message from the file passed as the first argument and compare that to the pattern, you can force Git to abort the commit if there is no match:

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
 puts "[POLICY] Your message is not formatted correctly"
 exit 1
end
```

If that script is in place (in `.git/hooks/commit-msg`) and executable, and you commit with a message that isn't properly formatted, you see this:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

No commit was completed in that instance. However, if your message contains the proper pattern, Git allows you to commit:

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
1 file changed, 1 insertions(+), 0 deletions(-)
```

Next, you want to make sure you aren't modifying files that are outside your ACL scope. If your project's `.git` directory contains a copy of the ACL file you used previously, then the following `pre-commit` script will enforce those constraints for you:

```
#!/usr/bin/env ruby

$user = ENV['USER']

[insert acl_access_data method from above]

only allows certain users to modify certain subdirectories in a project
def check_directory_perms
 access = get_acl_access_data('.git/acl')

 files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
 files_modified.each do |path|
 next if path.size == 0
 has_file_access = false
 access[$user].each do |access_path|
 if !access_path || (path.index(access_path) == 0)
 has_file_access = true
 end
 end
 if !has_file_access
 puts "[POLICY] You do not have access to push to #{path}"
 exit 1
 end
 end
end

check_directory_perms
```

This is roughly the same script as the server-side part, but with two important differences. First, the ACL file is in a different place, because this script runs from your working directory, not from your `.git` directory. You have to change the path to the ACL file from this

```
access = get_acl_access_data('acl')
```

to this:

```
access = get_acl_access_data('.git/acl')
```

The other important difference is the way you get a listing of the files that have been changed. Because the server-side method looks at the log of commits, and, at this point, the commit hasn't been recorded yet, you must get your file listing from the staging area instead. Instead of

```
files_modified = `git log -1 --name-only --pretty=format:'' #[ref]`
```

you have to use

```
files_modified = `git diff-index --cached --name-only HEAD`
```

But those are the only two differences – otherwise, the script works the same way. One caveat is that it expects you to be running locally as the same user you push as to the remote machine. If that is different, you must set the `$user` variable manually.

One other thing we can do here is make sure the user doesn't push non-fast-forwarded references. To get a reference that isn't a fast-forward, you either have to rebase past a commit you've already pushed up or try pushing a different local branch up to the same remote branch.

Presumably, the server is already configured with `receive.denyDeletes` and `receive.denyNonFastForwards` to enforce this policy, so the only accidental thing you can try to catch is rebasing commits that have already been pushed.

Here is an example pre-rebase script that checks for that. It gets a list of all the commits you're about to rewrite and checks whether they exist in any of your remote references. If it sees one that is reachable from one of your remote references, it aborts the rebase.

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
 topic_branch = ARGV[1]
else
 topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
 remote_refs.each do |remote_ref|
 shas_pushed = `git rev-list ^#{sha}@ refs/remotes/#{remote_ref}`
 if shas_pushed.split("\n").include?(sha)
 puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
 end
 end
end
```

```
 exit 1
 end
end
end
```

This script uses a syntax that wasn't covered in the Revision Selection section of Chapter 6. You get a list of commits that have already been pushed up by running this:

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

The SHA<sup>^@</sup> syntax resolves to all the parents of that commit. You're looking for any commit that is reachable from the last commit on the remote and that isn't reachable from any parent of any of the SHA-1s you're trying to push up – meaning it's a fast-forward.

The main drawback to this approach is that it can be very slow and is often unnecessary – if you don't try to force the push with -f, the server will warn you and not accept the push. However, it's an interesting exercise and can in theory help you avoid a rebase that you might later have to go back and fix.

## Заключение

Мы рассмотрели большинство основных способов настройки клиента и сервера Git'а с тем, чтобы он был максимально удобен для ваших проектов и при вашей организации рабочего процесса. Мы узнали о всевозможных настройках, атрибутах файлов и о перехватчиках событий, а также рассмотрели пример настройки сервера с соблюдением политики. Теперь вам должно быть по плечу заставить Git подстроиться под практически любой тип рабочего процесса, который можно вообразить.



# Git и другие системы контроля версий

9

Наш мир несовершенен. Как правило, вы не сможете быстро перевести проект, в котором вы участвуете, на использование Git'a. Иногда вам придётся иметь дело с проектами, использующими другую систему контроля версий, хотя вам и не нравится, что это не Git. В первой части этого раздела вы узнаете о способах использования Git'a в качестве клиента для работы с проектом, размещенном в какой-то другой системе.

В какой-то момент, вы, возможно, захотите перевести свой существующий проект на Git. Во второй части раздела вы узнаете о том, как провести миграцию с некоторых распространённых систем, а также познакомитесь с методом, который будет работать в нестандартных ситуациях, когда готовых инструментов миграции не существует.

## Git как клиент

Git оставляет настолько положительное впечатление на разработчиков, что многие из них придумывают способы, как использовать его на своём компьютере, в случае если остальная часть команды использует другую СКВ. Для этого разработан целый ряд специальных адаптеров, называемых “мостами” (“bridges”). Здесь мы рассмотрим те, с которыми вы, скорее всего, столкнетесь в работе над реальными проектами.

## Git и Subversion

Весомая часть проектов с исходным кодом, равно как и огромное количество корпоративных проектов, до сих пор используют Subversion

для версионирования исходного кода. SVN'у больше десяти лет и большую часть этого срока он оставался *единственным* вариантом СКВ для проектов с открытым исходным кодом. SVN очень похож на CVS, своего предка — “крёстного отца” всех современных СКВ.

Одна из многих замечательных вещей в Git — это поддержка двусторонней интеграции с SVN через `git svn`. Этот инструмент позволяет использовать Git в качестве полноценного SVN клиента; вы можете использовать весь функционал Git для работы с локальным репозиторием, скомпоновать ревизии и отправить их на сервер, словно вы использовали обычный SVN. Да, вы не ослышались: можно создавать локальные ветки, производить слияния, использовать индекс для неполного применения изменений, перемещать фиксации и повторно применять их (*cherry-pick*) и т.д., в то время как ваши коллеги, использующие SVN, застряли в палеолите. Это отличный способ по-партизански внедрить Git в процесс разработки и помочь соратниками стать более продуктивными, а затем потребовать от инфраструктуры полной поддержки Git. `git svn` — это первый укол наркотика “РСКВ”, вызывающего сильнейшее привыкание.

## GIT SVN

Основная команда для работы с Subversion — это `git svn`. Она принимает несколько дополнительных команд, которые мы рассмотрим далее.

Важно понимать, что каждый раз, когда вы используете `git svn`, вы взаимодействуете с Subversion, который работает совсем не как Git. И хотя вы **можете** создавать и сливать локальные ветки, всё же лучше сохранять историю линейной настолько, насколько это возможно, используя перемещение фиксаций. Также избегайте одновременной работы с удалённым Git сервером.

Не изменяйте уже опубликованную историю, и не зеркалируйте изменения в Git репозитории, с которым работают люди, использующие Git (они могут изменить историю). В Subversion может быть только одна линейная история фиксаций. Если в вашей команде часть людей использует SVN, а часть — Git, убедитесь, что все используют SVN сервер для сотрудничества. Это сделает вашу жизнь проще.

## УСТАНОВКА

Чтобы попробовать `git svn` в деле вам понадобится обычный SVN репозиторий с правом на запись. Если вы хотите попробовать

примеры ниже, вам понадобится копия нашего тестового репозитория. К счастью, в Subversion есть инструмент `svnsync`, который упростит перенос. Для тестов мы создали новый Subversion репозиторий на Google Code, являющийся частичной копией проекта `protobuf` — библиотеки для сериализации структурированных данных для передачи по сети.

Если вы с нами, создайте локальный Subversion репозиторий:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Затем, позвольте всем пользователям изменять т.н. `гевргорс`; самый простой способ сделать это — добавить скрипт `рге-гевргорс-change`, всегда возвращающий 0:

```
$ cat /tmp/test-svn/hooks/рге-гевргорс-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/рге-гевргорс-change
```

Теперь вы можете синхронизировать репозиторий на локальной машине, вызвав `svnsync init`, передав входной и выходной репозитории:

```
$ svnsync init file:///tmp/test-svn \
http://progit-example.googlecode.com/svn/
```

Наконец (SVN вам ещё не надоел?), можно запустить саму синхронизацию. Затем можно будет склонировать собственно код, выполнив:

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Transmitting file data[...]
Committed revision 2.
Copied properties for revision 2.
[...]
```

На всё про всё у вас уйдёт несколько минут, но на самом деле вам ещё повезло: если бы вы копировали данные не на свой компьютер, а в другой удалённый репозиторий, понадобился бы почти час, несмотря на то, что в тестовом проекте меньше сотни ревизий. Subversion копирует данные последовательно, скачивая по одной ревизии и отправляя в другой репозиторий — это поразительно неэффективно, но как есть, так есть.

## НАЧАЛО РАБОТЫ

Теперь, когда у вас есть Subversion репозиторий с правами на запись, можно опробовать типичные приёмы работы с ним через `git svn`. Начнём с команды `git svn clone`, которая склонирует Subversion репозиторий целиком в локальный Git репозиторий. Разумеется, при переносе реального Subversion репозитория нужно будет заменить `file:///tmp/test-svn` на настоящий URL:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcbfb5891860124cc2e8cc616cded42624897125 (refs/remotes/origin/trunk)
 A m4/acx_pthread.m4
 A m4/stl_hash.m4
 A java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
 A java/src/test/java/com/google/protobuf/WireFormatTest.java
...
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae (refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-svn/bra
Found branch parent: (refs/remotes/origin/my-calc-branch) 556a3e1e7ad1fde0a32823fc
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-calc-branch)
Checked out HEAD:
 file:///tmp/test-svn/trunk r75
```

Приведённая выше команда является композицией двух других — `git svn init` и `git svn fetch` для указанного URL. Процесс копирования займёт некоторое время. Тестовый проект невелик — всего 75 фиксаций — но Git вынужден последовательно скачивать SVN ревизии и превращать их в Git фиксации по одной за раз. Для проекта с сотней или тысячей ревизий это может занять часы или даже дни!

Параметры `-T trunk -b branches -t tags` говорят Git о том, что клонируемый репозиторий следует стандартному, принятому в Subversion, расположению директорий с транком, ветками и метками. Если

же директории названы по-другому, можно указать их явно, используя эти параметры. Большинство Subversion репозиториев следуют этому соглашению, поэтому для этой комбинации параметров существует сокращение `-s`, что означает “стандартное расположение директорий” Следующая команда эквивалентна приведённой выше:

```
$ git svn clone file:///tmp/test-svn -s
```

На этом этапе у вас должен быть обычный Git репозиторий с импортированными ветками и метками:

```
$ git branch -a
* master
 remotes/origin/my-calc-branch
 remotes/origin/tags/2.0.2
 remotes/origin/tags/release-2.0.1
 remotes/origin/tags/release-2.0.2
 remotes/origin/tags/release-2.0.2rc1
 remotes/origin/trunk
```

Обратите внимание, как `git svn` представляет метки Subversion в виде ссылок. Давайте посмотрим на это повнимательней, используя команду `show-ref`:

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abca9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711fed4 refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

При работе с Git репозиторием Git поступает иначе, вот как выглядит Git репозиторий сразу после клонирования:

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
```

```
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dcb09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
```

Заметили? Git помещает метки прямиком в `refs/tags`, но в случае с Subversion репозиторием они трактуются как удалённые ветки.

## ОТПРАВКА ИЗМЕНЕНИЙ В SUBVERSION

Теперь, когда вы настроили репозиторий, можно проделать некую работу и отправить изменения обратно в Subversion, используя Git как SVN клиент. Если вы отредактируете какой-либо файл и зафиксируете изменения, полученная фиксация будет существовать в локальном Git репозитории, но не на сервере Subversion.

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
 1 file changed, 5 insertions(+)
```

Далее следует отправить изменения на сервер. Обратите внимание как это меняет привычный сценарий работы с Subversion: вы фиксируете изменения без связи с сервером, а затем отправляете их все при удобном случае. Чтобы отправить изменения на Subversion сервер, следует выполнить команду `git svn dcommit`:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
 M README.txt
Committed r77
 M README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Эта команда берёт все изменения, зафиксированные поверх последних ревизий с Subversion сервера, создаёт для каждого новую Subversion ревизию, а затем переписывает их, добавляя уникальный идентификатор. Это важно, потому что изменяются SHA-1 хеши фиксаций. Это одна из причин, почему не рекомендуется смешивать Subversion и Git сервер в одном проекте. Если посмотреть на последнюю фиксацию, вы увидите, что добавилась строка `git-svn-id`:

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date: Thu Jul 24 03:08:36 2014 +0000

 Adding git-svn instructions to the README

git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
```

Обратите внимание, что SHA-1 хеш, прежде начинавшийся с 4af61fd, теперь начинается с 95e0222. Если всё же хотите работать как с Git, так и с Subversion серверами в одном проекте, сначала следует отправлять (`dcommit`) изменения на Subversion сервер, так как это изменяет хеши.

## ПОЛУЧЕНИЕ НОВЫХ ИЗМЕНЕНИЙ

Если вы работаете в команде, рано или поздно кто-то успеет отправить изменения раньше вас и вы не сможете отправить свои изменения на сервер не разрешив возникший конфликт. Ваши фиксации будут попросту отвергаться сервером, пока вы не произведёте слияние. В `git svn` это выглядит следующим образом:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and refs/remotes/origin/trunk differ, using rebase
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145 c80b6127dd04f5fcda218730ddf3a2da4eb3
Current branch master is up to date.
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Чтобы решить эту проблему запустите `git svn rebase`, которая заберёт все ревизии с сервера, которых у вас пока нет, и переместит (`rebase`) ваши локальные наработки на них:

```
$ git svn rebase
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
```

```
Transaction is out of date: File '/trunk/README.txt' is out of date
W: eaa029d99f87c5c822c5c29039d19111ff32ef46 and refs/remotes/origin/trunk differ,
:100644 100644 65536c6e30d263495c17d781962cff12422693a b34372b25ccf4945fe5658fa38
First, rewinding head to replay your work on top of it...
Applying: update foo
Using index info to reconstruct a base tree...
M README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Теперь история линейна и вы можете успешно выполнить `dcommit`:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M README.txt
Committed r85
M README.txt
r85 = 9c29704cc0bbbed7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

В отличие от Git, всегда требующего производить слияние свежих изменений из удалённого репозитория с локальными наработками, перед отправкой на сервер, `git svn` требует слияний только в случае конфликтующих изменений (так работает Subversion). Если кто-нибудь изменил один файл, а затем вы изменяете другой, `dcommit` сработает гладко:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M configure.ac
Committed r87
M autogen.sh
r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs/remotes/origin/trunk)
M configure.ac
r87 = f3653ea40cb4e26b6281cec102e35dcba1fe17c4 (refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fefd2b1d92806 and refs/remotes/origin/trunk differ,
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 e757b59a9439312d80d5d43bb6
First, rewinding head to replay your work on top of it...
```

Важно помнить про это, потому что в результате такого поведения вы получаете непредсказуемое состояние проекта, до этого не существовавшее ни на одном из компьютеров. Если изменения были несовместимы между собой, но не вызывали конфликта слияния (например, логически противоречивые изменения в разных файлах) в результате подобного произвола могут возникнуть труднодиагностируемые проблемы. С Git сервером дела обстоят иначе: перед отправкой изменений в удалённый репозиторий вы можете полностью протестировать проект локально, в то время как в Subversion вы не можете быть уверенными, что состояние проекта до и после фиксации было одинаковым.

Даже если вы не готовы зафиксировать собственные изменения, следует почаще забирать изменения с Subversion сервера. Для синхронизации можно использовать `git svn fetch`, или `git svn rebase`; последняя команда не только забирает все изменения из Subversion, но и переносит ваши локальные фиксации наверх.

```
$ git svn rebase
M autogen.sh
r88 = c9c5f83c64bd755368784b44bc7a0216cc1e17b (refs/remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/origin/trunk.
```

Выполнение `git svn rebase` актуализирует состояние локального репозитория. Для выполнения этой команды ваша рабочая директория не должна содержать незафиксированных изменений. Если это не так, вам следует либо “спрятать” (`stash`) свои наработки, либо на время зафиксировать: иначе `git svn rebase` прекратит выполнение в случае конфликта.

## ПРОБЛЕМЫ С GIT-ВЕТВЛЕНИЕМ

После того как вы привыкните к Git, вам понравится создавать тематические ветки, работать в них и сливать их основную ветку разработки. Если работаете с Subversion сервером через `git svn`, вам придётся перемещать изменения, а не проводить слияния. Причина кроется в линейности истории в Subversion — в нём принята несколько иная концепция ветвления и слияния — так что `git svn` учитывает лишь первого родителя любой фиксации при преобразовании её в SVN формат.

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
 M CHANGES.txt
Committed r89
 M CHANGES.txt
r89 = 89d492c884ea7c834353563d5d913c6adf933981 (refs/remotes/origin/trunk)
 M COPYING.txt
 M INSTALL.txt
Committed r90
 M INSTALL.txt
 M COPYING.txt
r90 = cb522197870e61467473391799148f6721bcf9a0 (refs/remotes/origin/trunk)
No changes between 71af502c214ba13123992338569f4669877f55fd and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Выполнение команды `dcommit` на ветке с уже слитой историей пройдёт успешно, за исключением того момента, что при просмотре истории вы заметите, что фиксации из ветки `experiment` не были переписаны один за другим; вместо этого они схлопнулись в одну фиксацию-слияние.

Когда кто-нибудь склонирует этот репозиторий, всё что он увидит — единственное слияние, в котором собраны все изменения, словно вы выполнили `git merge --squash`; они не увидят кто и когда производил фиксации.

## SUBVERSION-ВЕТВЛЕНИЕ

Итак, ветвление в Subversion отличается от оного в Git; используйте его как можно реже. Тем не менее, используя `git svn`, вы можете создавать Subversion-ветки и фиксировать изменения в них.

## СОЗДАНИЕ НОВЫХ SVN-ВЕТОК

Чтобы создать новую ветку в Subversion, выполните `git svn branch [имя ветки]`:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r90 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-svn/branches/opera...
Found branch parent: (refs/remotes/origin/opera) cb522197870e61467473391799148f6721bcf9a0
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

Это эквивалентно выполнению команды `svn copy trunk branches/орега` в Subversion, при этом действия совершаются на Subversion сервере. Обратите внимание, что создание SVN ветки не переключает вас на неё; если сейчас зафиксировать какие-либо изменения и отправить их на сервер, они попадут в ветку `trunk`, а не `орега`.

## ПЕРЕКЛЮЧЕНИЕ АКТИВНЫХ ВЕТОК

Git определяет ветку, в которую он отправит ваши фиксации при выполнении `dcommit`, ища верхушку Subversion-ветки в вашей истории — она должна быть одна и она должна быть последней в текущей истории веток, имеющей метку `git-svn-id`.

Если вы хотите работать одновременно с несколькими ветками, вы можете настроить локальные ветки на внесение изменений через `dcommit` в конкретные ветки Subversion, отпочковывая их из импортированных SVN-ревизий нужных веток. Если вам нужна ветка `орега`, в которой вы можете поработать отдельно, можете выполнить:

```
$ git branch орега remotes/origin/орега
```

Теперь, если вы захотите слить ветку `орега` в `trunk` (`master`), вы сможете сделать это с помощью обычной команды `git merge`. Однако вам потребуется добавить подробное описание к фиксации (через параметр `-m`), иначе при слиянии комментарий будет иметь вид “Merge branch `орега`” вместо чего-нибудь полезного.

Помните, что хотя вы и используете `git merge` для этой операции, и слияние, скорее всего, произойдёт намного проще, чем в Subversion (потому что Git автоматически определяет подходящую основу для слияния), оно не является обычным слиянием в Git. Вы должны передать данные обратно на сервер в Subversion, который не способен справиться с фиксацией, имеющей более одного родителя; так что после передачи она будет выглядеть как единое целое, куда будут затолканы все изменения из другой ветки. После того как вы сольёте одну ветку в другую, вы не сможете просто так вернуться к работе над ней, как могли бы в Git. Команда `dcommit` удаляет всю информацию о том, какая ветка была влита, так что последующие вычисления базы слияния будут неверными — команда `dcommit` сделает результаты выполнения `git merge` такими же, какими они были бы после выполнения `git merge --squash`. К сожалению, избежать подобной ситуации вряд ли удастся: Subversion не способен сохранять подобную

информацию, так что вы всегда будете связаны этими ограничениями. Во избежание проблем вы должны удалить локальную ветку (в нашем случае `oreg`) после того, как вы вольёте её в `trunk`.

## КОМАНДЫ SUBVERSION

В `git svn` содержится несколько команд для облегчения перехода на Git путём предоставления схожего с Subversion функционала. Ниже приведены несколько команд, которые дают вам то, что вы имели в Subversion.

### SVN

Если вы привыкли к Subversion и хотите просматривать историю в стиле SVN, выполните команду `git svn log`, чтобы просматривать историю в таком же формате, как в SVN:

```
$ git svn log

r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2 lines
autogen change

r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2 lines
Merge branch 'experiment'

r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2 lines
updated the changelog
```

Вы должны знать две важные вещи о команде `git svn log`. Во-первых, для её работы не требуется доступ к сети, в отличие от оригинальной команды `svn log`, которая запрашивает информацию с Subversion сервера. Во-вторых, эта команда отображает только те фиксации, которые были переданы на Subversion сервер. Локальные Git фиксации, которые вы ещё не отправили с помощью `dcommit`, не будут отображаться, равно как и фиксации, отправленные на Subversion сервер другими людьми с момента последнего выполнения `dcommit`. Результат действия этой команды скорее похож на последнее известное состояние изменений на Subversion сервере.

**SVN-**

Так же как команда `git svn log` эмулирует команду `svn log`, эквивалентом команды `svn annotate` является команда `git svn blame [ФАЙЛ]`. Вывод выглядит следующим образом:

```
$ git svn blame README.txt
2 temporal Protocol Buffers - Google's data interchange format
2 temporal Copyright 2008 Google Inc.
2 temporal http://code.google.com/apis/protocolbuffers/
2 temporal
22 temporal C++ Installation - Unix
22 temporal =====
2 temporal
79 schacon Committing in git-svn.
78 schacon
2 temporal To build and install the C++ Protocol Buffer runtime and the Protocol
2 temporal Buffer compiler (protoc) execute the following:
2 temporal
```

Опять же, эта команда не показывает фиксации, которые вы сделали локально в Git или те, что были отправлены на Subversion сервер с момента последней связи с ним.

**SVN-**

Вы можете получить ту же информацию, которую выдаёт `svn info`, выполнив команду `git svn info`:

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

Так же, как `blame` и `log`, эта команда выполняется без доступа к сети и выводит информацию, актуальную на момент последнего обращения к серверу Subversion.

### , Subversion

Если вы клонируете Subversion-репозиторий с установленными svn:ignore свойствами, скорее всего, вы захотите создать соответствующие им файлы .gitignore, чтобы ненароком не зафиксировать лишнего. Для решения этой проблемы в git svn имеется две команды. Первая — git svn create-ignore — автоматически создаст соответствующие файлы .gitignore, которые вы затем можете зафиксировать.

Вторая команда — git svn show-ignore — выводит на стандартный вывод строки, которые следует включить в файл .gitignore; вы можете попросту перенаправить вывод этой команды в файл исключений:

```
$ git svn show-ignore > .git/info/exclude
```

Поступая таким образом, вы не захламляете проект файлами .gitignore. Это хорошее решение в случае если вы являетесь единственным пользователем Git в команде, использующей Subversion, и ваши коллеги выступают против наличия файлов .gitignore в проекте.

## ЗАКЛЮЧЕНИЕ ПО GIT-SVN

Утилиты git svn полезны в том случае, если ваша разработка по каким-то причинам требует наличия рабочего Subversion-сервера. Однако, стоит воспринимать их как “функционально урезанный” Git, ибо при использовании всех возможностей Git вы столкнётесь с проблемами в преобразованиях, которые могут сбить с толку вас и ваших коллег. Чтобы избежать неприятностей, старайтесь следовать следующим рекомендациям:

- Держите историю в Git линейной, чтобы она не содержала слияний, сделанных с помощью git merge. Перемещайте всю работу, которую вы выполняете вне основной ветки обратно в неё; не выполняйте слияний.
- Не устанавливайте отдельный Git-сервер для совместной работы. Можно иметь такой сервер для того, чтобы ускорить клонирование для новых разработчиков, но не отправляйте на него ничего, не имеющего записи git-svn-id. Возможно, стоит даже добавить перехватчик rge-gesceive, который будет

проверять каждое изменение на наличие `git-svn-id` и отклонять фиксации, если они не имеют такой записи.

При следовании этим правилам, работа с Subversion сервером может быть более-менее сносной. Однако, если возможен перенос проекта на нормальный Git-сервер, преимущества от этого перехода дадут вашему проекту намного больше.

## Git и Mercurial

Вселенная распределённых систем контроля версий не заканчивается на Git. На самом деле, существуют и другие системы, каждая со своим подходом к управлению версиями. На втором месте по популярности после Git'a находится Mercurial и у этих систем много общего.

К счастью, если вам нравится Git, но приходится работать с Mercurial-репозиторием, существует способ использовать Git-клиент для работы с Mercurial. Учитывая тот факт, что Git работает с серверами через концепцию “удалённых репозиториев” (remotes), неудивительно, что работа с Mercurial-репозиторием происходит через своего рода обёртку над “удалённым репозиторием”. Проект, добавляющий такую интероперабельность, называется `git-remote-hg` и расположен по адресу <https://github.com/felipec/git-remote-hg>.

### GIT-REMOTE-HG

Для начала необходимо установить `git-remote-hg`. Ничего особенного — просто поместите файл в место, откуда он будет виден другим программам, типа:

```
$ curl -o ~/bin/git-remote-hg \
https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

...предполагая, что `~/bin` включён в `$PATH`. Есть ещё одна зависимость: библиотека `mercurial` для Python. Если у вас установлен Python, просто выполните:

```
$ pip install mercurial
```

(Если же у вас ещё нет Python, пора исправить это: скачайте установщик с <https://www.python.org/>.)

Ну и наконец понадобится сам клиент Mercurial. Если он ещё не установлен — скачайте и установите с <http://mercurial.selenic.com/>.

Теперь можно отжигать! Всё что потребуется — репозиторий Mercurial с которым вы можете работать. К счастью, подойдёт любой, так что мы воспользуемся репозиторием “привет, мир”, используемом для обучения Mercurial’у.

```
$ hg clone http://selenic.com/hero/hello /tmp/hello
```

## ОСНОВЫ

Теперь, когда у нас есть подходящий “серверный” репозиторий, мы готовы разобрать типичные приёмы работы с Mercurial. Как вы увидите, эти две системы очень похожи, так что всё пройдёт гладко.

Как и всегда, вначале мы клонируем репозиторий:

```
$ git clone hg::/tmp/hello /tmp/hello-git
$ cd /tmp/hello-git
$ git log --oneline --graph --decorate
* ac7955c (HEAD, origin/master, origin/branches/default, origin/HEAD, refs/hg/orig
* 65bb417 Create a standard "hello, world" program
```

Наверняка вы обратили внимание, что мы использовали обычновенный `git clone`. Это потому, что `git-remote-hg` работает на довольно низком уровне, подобно тому, как в Git реализован HTTP/S протокол (`git-remote-hg` служит как бы в качестве “помощника” для работы с удалённым репозиторием по новому протоколу (`hg`), расширяя базовые возможности Git). Подобно Git’у, Mercurial рассчитан на то, что каждый клиент хранит полную копию репозитория со всей историей, поэтому приведённая выше команда выполняет полное копирование со всей историей. Стоит отметить, что делает она это достаточно быстро.

`git log` показывает две фиксации, на последнюю из которых указывает довольно много ссылок. На самом деле, не все из них реально существуют. Давайте-ка посмотрим, что хранится внутри директории `.git`:

```
$ tree .git/refs
.git/refs
├── heads
│ └── master
├── hg
│ └── origin
│ ├── bookmarks
│ │ └── master
│ └── branches
│ └── default
├── notes
│ └── hg
└── remotes
 └── origin
 └── HEAD
└── tags

9 directories, 5 files
```

`git-remote-hg` пытается нивелировать различия между Git и Mercurial, преобразовывая форматы за кулисами. Ссылки на объекты в удалённом репозитории хранятся в директории `refs/hg`. Например, `refs/hg/origin/branches/default` — это Git-ссылка, содержащая SHA-1 “`ac7955c`” — фиксация на который ссылается ветка `master`. Таким образом, директория `refs/hg` — это что-то типа `refs/remotes/origin`, с той разницей что здесь же отдельно хранятся Mercurial-закладки и ветки.

Файл `notes/hg` — отправная точка для выяснения соответствия между хешами фиксаций в Git и идентификаторами ревизий в Mercurial. Давайте посмотрим что там:

```
$ cat notes/hg
d4c10386...

$ git cat-file -p d4c10386...
tree 1781c96...
author remote-hg <> 1408066400 -0800
committer remote-hg <> 1408066400 -0800

Notes for master

$ git ls-tree 1781c96...
100644 blob ac9117f... 65bb417...
100644 blob 485e178... ac7955c...
```

```
$ git cat-file -p ac9117f
0a04b987be5ae354b710cefeba0e2d9de7ad41a9
```

Итак, `refs/notes/hg` указывает на дерево, которое содержит список других объектов и имён. Команда `git ls-tree` выводит права доступа, тип, хеш и имя файла для содержимого дерева. Наконец, добравшись до первого элемента дерева, мы обнаружим, что это блоб с названием “`ac9117f`” (SHA-1 фиксации, на которую указывает ветка `master`), содержащий “`0a04b98`” (идентификатор последней ревизии ветки `default` в Mercurial).

Всё это немного запутанно, но хорошие новости в том, что, по большому счёту, нам не нужно беспокоиться об организации данных в `git-remote-hg`. В целом, работа с Mercurial сервером не сильно отличается от работы с Git сервером.

Ещё одна вещь, которую следует учитывать: список игнорируемых файлов. Mercurial и Git используют очень похожие механизмы для таких списков, но всё же хранить `.gitignore` в Mercurial репозитории — не самая удачная идея. К счастью, в Git есть механизм игнорирования специфичных для локальной копии репозитория файлов, а формат списка исключений в Mercurial совместим с Git, так что можно просто скопировать `.hgignore` кое-куда:

```
$ cp .hgignore .git/info/exclude
```

Файл `.git/info/exclude` работает подобно `.gitignore`, но не фиксируется в истории изменений.

## РАБОЧИЙ ПРОЦЕСС

Предположим, вы проделали некую работу, зафиксировали изменения в ветке `master` и готовы отправить изменения в удалённый репозиторий. Вот как выглядит репозиторий сейчас:

```
$ git log --oneline --graph --decorate
* ba04a2a (HEAD, master) Update makefile
* d25d16f Goodbye
* ac7955c (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/br...
* 65bb417 Create a standard "hello, world" program
```

Наша ветка `master` опережает `origin/master` на две фиксации, пока что они есть лишь в локальном репозитории. Давайте посмотрим, вдруг кто-нибудь сделал важные изменения:

```
$ git fetch
From hg::/tmp/hello
 ac7955c..df85e87 master -> origin/master
 ac7955c..df85e87 branches/default -> origin/branches/default
$ git log --oneline --graph --decorate --all
* 7b07969 (refs/notes/hg) Notes for default
* d4c1038 Notes for master
* df85e87 (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default)
| * ba04a2a (HEAD, master) Update makefile
| * d25d16f Goodbye
|
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Из-за того, что мы использовали флаг `--all`, выводятся ссылки “notes”, используемые внутри `git-remote-hg`, можно не обращать на них внимания. В остальном, ничего необычного: `origin/master` продвинулся на одну фиксацию и история разошлась. В отличие от остальных систем контроля версий, рассматриваемых в этой главе, Mercurial умеет работать со слияниями, так что мы не будем вытворять никаких фокусов.

```
$ git merge origin/master
Auto-merging hello.c
Merge made by the 'recursive' strategy.
hello.c | 2 ++
1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline --graph --decorate
* 0c64627 (HEAD, master) Merge remote-tracking branch 'origin/master'
|
| * df85e87 (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default)
| * ba04a2a Update makefile
| * d25d16f Goodbye
|
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Отлично! Мы запустили все тесты, они прошли, так что всё готово для отправки изменений на удалённый сервер:

```
$ git push
To hg::/tmp/hello
 df85e87..0c64627 master -> master
```

Вот и всё! Если теперь посмотреть на Mercurial репозиторий, мы не увидим там ничего необычного:

```
$ hg log -G --style compact
o 5[tip]:4,2 dc8fa4f932b8 2014-08-14 19:33 -0700 ben
|\ Merge remote-tracking branch 'origin/master'
|
| o 4 64f27bcef35 2014-08-14 19:27 -0700 ben
| | Update makefile
|
| o 3:1 4256fc29598f 2014-08-14 19:27 -0700 ben
| | Goodbye
|
@ | 2 7db0b4848b3c 2014-08-14 19:30 -0700 ben
| / Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
| Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
| Create a standard "hello, world" program
```

Набор изменений 2 был произведён Mercurial'ом, а изменения 3 и 4 внесены git-remote-hg после отправки изменений, сделанных через Git.

## ВЕТКИ И ЗАКЛАДКИ

В Git есть только один тип веток: указатель, который передвигается “вперёд” по мере фиксации изменений. В Mercurial такие указатели называются “закладки” и ведут себя схожим с Git образом.

Понятие “ветка” в Mercurial означает немного другое. Название ветки, в которой происходят изменения, записывается *внутри каждого набора изменений* и, таким образом, навсегда остаётся в истории. Например, вот одна из фиксаций, произведённых в ветке `develop`:

```
$ hg log -l 1
changeset: 6:8f65e5e02793
branch: develop
```

```
tag: tip
user: Ben Straub <ben@straub.cc>
date: Thu Aug 14 20:06:38 2014 -0700
summary: More documentation
```

Обратите внимание на строку, начинающуюся с “branch”. Git устроен по-другому (на самом деле, оба типа веток могут быть представлены как ссылки в Git), но git-гемоте-hg вынужден понимать разницу, потому что нацелен на работу с Mercurial.

Создание “закладок” Mercurial не сложнее создания простых веток в Git. Вот что мы делаем в Git:

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg:///tmp/hello
 * [new branch] featureA -> featureA
```

А со стороны Mercurial это выглядит так:

```
$ hg bookmarks
featureA 5:bd5ac26f11f9
$ hg log --style compact -G
@ 6[tip] 8f65e5e02793 2014-08-14 20:06 -0700 ben
| More documentation
|
o 5[featureA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700 ben
|\ Merge remote-tracking branch 'origin/master'
|
| o 4 0434aaa6b91f 2014-08-14 20:01 -0700 ben
| | update makefile
|
| o 3:1 318914536c86 2014-08-14 20:00 -0700 ben
| | goodbye
|
o | 2 f098c7f45c4f 2014-08-14 20:01 -0700 ben
| / Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
| Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
| Create a standard "hello, world" program
```

Обратите внимание на метку [featureA] на пятой ревизии. Таким образом, со стороны Git “закладки” выглядят как обычные ветки с одним лишь исключением: нельзя удалить закладку через Git (это одно из ограничений обёртка для взаимодействия с другими СКВ).

Можно работать и с полноценными ветками Mercurial — просто поместите Git ветку в пространство имён `branches`:

```
$ git checkout -b branches/permanent
Switched to a new branch 'branches/permanent'
$ vi Makefile
$ git commit -am 'A permanent change'
$ git push origin branches/permanent
To hg::/tmp/hello
 * [new branch] branches/permanent -> branches/permanent
```

Вот как это будет выглядеть со стороны Mercurial:

```
$ hg branches
permanent 7:a4529d07aad4
develop 6:8f65e5e02793
default 5:bd5ac26f11f9 (inactive)
$ hg log -G
o changeset: 7:a4529d07aad4
| branch: permanent
| tag: tip
| parent: 5:bd5ac26f11f9
| user: Ben Straub <ben@straub.cc>
| date: Thu Aug 14 20:21:09 2014 -0700
| summary: A permanent change
|
| @ changeset: 6:8f65e5e02793
| / branch: develop
| | user: Ben Straub <ben@straub.cc>
| | date: Thu Aug 14 20:06:38 2014 -0700
| | summary: More documentation
|
o changeset: 5:bd5ac26f11f9
|\ bookmark: featureA
| | parent: 4:0434aaa6b91f
| | parent: 2:f098c7f45c4f
| | user: Ben Straub <ben@straub.cc>
| | date: Thu Aug 14 20:02:21 2014 -0700
| | summary: Merge remote-tracking branch 'origin/master'
[...]
```

Имя ветки “permanent” было записано внутри набора изменений с номером 7.

Итак, со стороны Git работа с обеими типами Mercurial веток выглядит одинаково: переключаемся на ветку, фиксируем изменения, забираем чужие наработки, производим слияния и отправляем изменения в репозиторий как обычно. И ещё: Mercurial не поддерживает изменение истории, только добавление новых изменений. Вот как будет выглядеть Mercurial репозиторий после интерактивного изменения истории и “насильной” её отправки назад:

```
$ hg log --style compact -G
o 10[tip] 99611176cbc9 2014-08-14 20:21 -0700 ben
| A permanent change
|
o 9 f23e12f939c3 2014-08-14 20:01 -0700 ben
| Add some documentation
|
o 8:1 c16971d33922 2014-08-14 20:00 -0700 ben
| goodbye
|
| o 7:5 a4529d07aad4 2014-08-14 20:21 -0700 ben
| | A permanent change
|
| | @ 6 8f65e5e02793 2014-08-14 20:06 -0700 ben
| |/ More documentation
|
| | o 5[featureA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700 ben
| | \ Merge remote-tracking branch 'origin/master'
| |
| | | o 4 0434aaa6b91f 2014-08-14 20:01 -0700 ben
| | | update makefile
| | |
| | | +---o 3:1 318914536c86 2014-08-14 20:00 -0700 ben
| | | | goodbye
| | |
| | o 2 f098c7f45c4f 2014-08-14 20:01 -0700 ben
| |/ Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
| Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
| Create a standard "hello, world" program
```

Были созданы изменения 8, 9 и 10 и теперь они принадлежат ветке permanent, но старые изменения никуда не делись. Это может **очень**

удивить ваших коллег, привыкших к Mercurial, так что старайтесь так не делать.

## ЗАКЛЮЧЕНИЕ

Git и Mercurial довольно похожи, их относительно просто можно “подружить”. Если вы будете избегать изменения уже опубликованной истории (это в целом хорошая идея, не только в контексте взаимодействия с Mercurial), вы даже не заметите что работаете с другой СКВ.

## Git и Perforce

Perforce — очень распространённая система контроля версий в корпоративной среде. Она появилась в 1995 году, что делает её самой старой СКВ, рассматриваемой в этой главе. Perforce разработан в духе тех времён; он предполагает постоянное подключение к центральному серверу, а локально хранится одна-единственная версия файлов. На самом деле, его возможности, как и ограничения, разрабатывались для решения вполне конкретных проблем; хотя многие проекты, использующие Perforce сегодня, выиграли бы от перехода на Git.

Существует два варианта совместного использования Git и Perforce. Первый — Git Fusion от разработчиков Perforce — позволяет выставлять субдеревья Perforce-депо в качестве удалённых Git репозиториев. Второй — `git-p4` — клиентская обёртка над Perforce для Git; она не требует дополнительной настройки Perforce сервера.

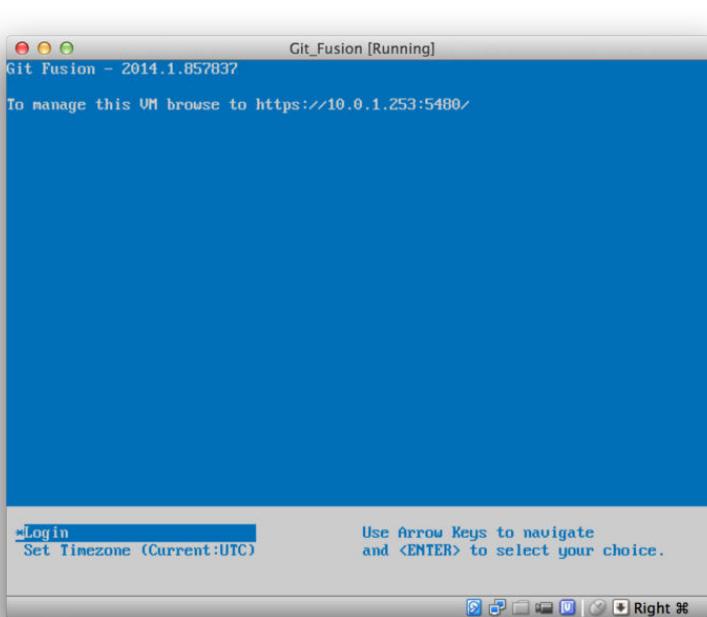
## GIT FUSION

У создателей Perforce есть продукт, именуемый Git Fusion (доступен на <http://www.perforce.com/git-fusion>), который синхронизирует Perforce сервер с Git репозиторием на стороне сервера.

Для примера мы воспользуемся простейшим способом настройки Git Fusion — подготовленным образом для виртуальной машины с предустановленным Perforce демоном и собственно Git Fusion’ом. Вы можете скачать образ на <http://www.perforce.com/downloads/Perforce/20-User>, а затем импортировать его в ваше любимое средство виртуализации (мы будем использовать VirtualBox).

Во время первого запуска вам потребуется сконфигурировать пароли трёх Linux пользователей (`root`, `perforce` и `git`) и имя хоста,

которое будет идентифицировать компьютер в сети. По окончании вы увидите следующее:

**FIGURE 9-1**

Экран виртуальной машины Git Fusion.

Запомните IP адрес, он пригодится в будущем. Далее, создадим пользователя Perforce. Выберите внизу опцию “Login” и нажмите Enter (или используйте SSH) и войдите как пользователь root. Используйте приведённые ниже команды, чтобы создать пользователя:

```
$ p4 -p localhost:1666 -u super user -f john
$ p4 -p localhost:1666 -u john passwd
$ exit
```

Первая команда откроет редактор для уточнения данных пользователя, но вы можете принять настройки по умолчанию, введя :wq и нажав Enter. Вторая команда дважды попросит ввести пароль. Это всё, что требовалось выполнить в оболочке ОС, можете завершить сессию.

Следующим шагом необходимо запретить Git проверять SSL сертификаты. Хотя виртуальная машина Git Fusion поставляется с сертификатом, он не привязан к домену и IP адресу виртуальной машины, так что Git будет отвергать соединения как небезопасные. Если вы собираетесь использовать эту виртуальную машину на постоянной основе, обратитесь к руководству по Git Fusion, чтобы узнать, как установить другой сертификат; для тестов же хватит следующего:

```
$ export GIT_SSL_NO_VERIFY=true
```

Теперь можете проверить что всё работает.

```
$ git clone https://10.0.1.254/Talkhouse
Cloning into 'Talkhouse'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
remote: Counting objects: 630, done.
remote: Compressing objects: 100% (581/581), done.
remote: Total 630 (delta 172), reused 0 (delta 0)
Receiving objects: 100% (630/630), 1.22 MiB | 0 bytes/s, done.
Resolving deltas: 100% (172/172), done.
Checking connectivity... done.
```

На виртуальной машине уже настроен проект, который вы можете клонировать. Мы клонируем репозиторий по HTTPS протоколу, используя ранее созданного пользователя `john`; Git спросит пароль, но менеджер паролей запомнит его для последующих запросов.

#### Fusion

После установки Git Fusion вы, возможно, захотите настроить его. Это относительно несложно сделать, используя ваш любимый Perforce клиент; просто отобразите директорию `//.git-fusion` на Perforce сервере в ваше рабочее пространство. Структура файлов приведена ниже:

```
$ tree
.
├── objects
│ └── repos
│ └── [...]
└── trees
 └── [...]
```

```

|
├── p4gf_config
├── repos
│ └── Talkhouse
│ └── p4gf_config
└── users
 └── p4gf_usermap

498 directories, 287 files

```

Директория `objects` используется Git Fusion для отображения объектов Perforce в Git и наоборот, вам не следует ничего здесь трогать. Внутри расположен глобальный конфигурационный файл `p4gf_config`, а также по одному такому же файлу для каждого репозитория — эти файлы и определяют поведение Git Fusion. Заглянем в тот, что в корне:

```

[repo-creation]
charset = utf8

[git-to-perforce]
change-owner = author
enable-git-branch-creation = yes
enable-swarm-reviews = yes
enable-git-merge-commits = yes
enable-git-submodules = yes
preflight-commit = none
ignore-author-permissions = no
read-permission-check = none
git-merge-avoidance-after-change-num = 12107

[perforce-to-git]
http-url = none
ssh-url = none

[@features]
imports = False
chunked-push = False
matrix2 = False
parallel-push = False

[authentication]
email-case-sensitivity = no

```

Мы не будем вдаваться в назначение каждой опции, просто обратите внимание, что это обычновенный INI файл, подобный тем, что использует Git для конфигурации. Файл, рассмотренный выше,

задаёт глобальные опции, которые могут быть переопределены внутри специфичных для репозитория файлов конфигурации, типа `repos/Talkhouse/p4gf_config`. Если откроете этот файл, увидите секцию `[@геро]`, определяющую некоторые глобальные настройки. Также, внутри есть секции, подобные этой:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

Они задают соответствие между ветками Perforce и ветками Git. Названия таких секций могут быть произвольными; главное, чтобы они оставались уникальными. `git-branch-name` позволяет преобразовать пути внутри депо, которые смотрелись бы непривычно для Git пользователей. Параметр `view` управляет отображением Perforce файлов на Git репозиторий; используется стандартный синтаксис отображения видов. Может быть задано более одного отображения, как в примере ниже:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
 //depot/project2/mainline/... project2/...
```

Таким образом, если ваше отображение включает изменения в структуре директорий, вы можете реплицировать эти изменения здесь.

Последний файл, который мы обсудим, это `users/p4gf_usermap`; в нём задаётся отображение пользователей Perforce на пользователей Git. Возможно, вам не пригодится этот файл.

Когда Git Fusion преобразовывает набор изменений Perforce в Git фиксацию, он находит пользователя в этом файле и использует хранящиеся здесь адрес электронной почты и полное имя для заполнения полей “автор” и “применяющий изменения” в Git. При обратном процессе ищется пользователь Perforce с адресом электронной почты из поля “автор” Git фиксации и используется далее для изменения.

В большинстве случаев это нормальное поведение, но что будет, если соответствия выглядят так:

```
john john@example.com "John Doe"
john johnny@appleseed.net "John Doe"
bob employeeX@example.com "Anon X. Mouse"
joe employeeY@example.com "Anon Y. Mouse"
```

Каждая строка имеет формат <user> <email> "<full name>" и задаёт соответствие для одного пользователя. Первые две строчки отображают два разных адреса электронной почты на одного и того же пользователя. Это может быть полезным если вы фиксировали изменения в Git, используя разные адреса, или если вы поменяли адрес, но хотите отобразить эти изменения на одного и того же Perforce пользователя. При создании Git фиксаций Perforce используется информация из первой совпадшей строки.

Последний две строки скрывают настоящие имена Боба и Джо в созданных Git фиксациях. Это может быть полезным, если вы хотите отдать внутренний проект в open-source, но не хотите раскрывать информацию о сотрудниках. Адреса электронной почты и полные имена должны быть уникальными если вы хотите хоть как-то различать авторов в полученном Git репозитории.

Perforce Git Fusion — это двунаправленный “мост” между Perforce и Git. Давайте посмотрим, как выглядит работа со стороны Git. Предполагается, что мы настроили отображение проекта “Jam”, используя приведённую выше конфигурацию. Тогда мы можем клонировать его:

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://ben@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
 remotes/origin/HEAD -> origin/master
 remotes/origin/master
 remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest metrowerks on Beos
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
[...]
```

В первый раз этот процесс может занять некоторое время. Git Fusion преобразовывает все наборы изменений Perforce в Git фиксации. Данные преобразуются локально на сервере, так что это вполне быстрый процесс; тем не менее, он может слегка затянуться, если у вас большая история изменений. Последующие скачивания требуют лишь инкрементального преобразования данных, таким образом скорость будет сравнима со скоростью работы обычного Git сервера.

Как видите, наш репозиторий выглядит так же, как выглядел бы любой другой Git репозиторий. В нём три ветки и Git предусмотрительно создал локальную ветку `master`, отслеживающую `origin/master`. Давайте немного поработаем и зафиксируем изменения:

```
...
$ git log --oneline --decorate --graph --all
* cfd46ab (HEAD, master) Add documentation for new feature
* a730d77 Whitespace
* d254865 (origin/master, origin/HEAD) Upgrade to latest metrowerks on Beos -- the
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

Теперь у нас две новых фиксации. Проверим, какие изменения внесли другие:

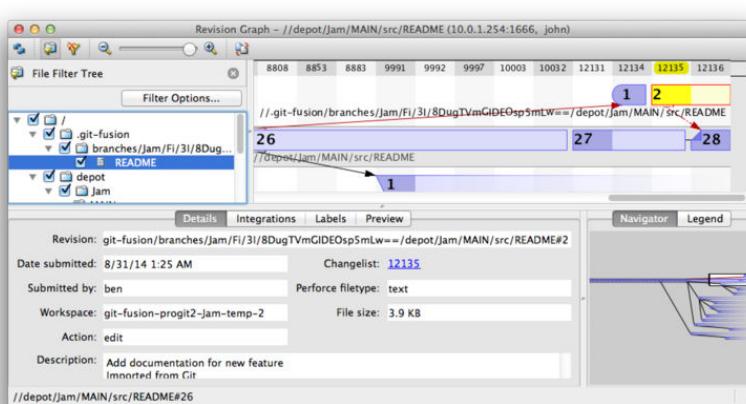
```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://10.0.1.254/Jam
 d254865..6afeb15 master -> origin/master
$ git log --oneline --decorate --graph --all
* 6afeb15 (origin/master, origin/HEAD) Update copyright
| * cfd46ab (HEAD, master) Add documentation for new feature
| * a730d77 Whitespace
|/
* d254865 Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

Кто-то успел отправить свои изменения раньше нас! Конечно, из приведённого вывода команды `git fetch` не видно, но на самом деле фиксация с SHA-1 `6afeb15` была создана Perforce клиентом. Она

выглядит так же, как и любая другая фиксация, и это именно то, для чего создан Git Fusion. Давайте посмотрим, как Perforce обработает фиксацию-слияние:

```
$ git merge origin/master
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 6), reused 0 (delta 0)
remote: Perforce: 100% (3/3) Loading commit tree into memory...
remote: Perforce: 100% (5/5) Finding child commits...
remote: Perforce: Running git fast-export...
remote: Perforce: 100% (3/3) Checking commits...
remote: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
To https://10.0.1.254:Jam
 6afeb15..89cba2b master -> master
```

Со стороны Git всё работает как положено. Давайте посмотрим на историю файла README со стороны Perforce, используя p4v:

**FIGURE 9-2**

*Граф ревизий  
Perforce после  
отправки данных из  
Git.*

Если вы ни разу не работали с Perforce это окно может показаться вам запутанным, но его концепция аналогичная gitk. Мы просматриваем историю файла README, так что дерево с директориями слева вверху показывает этот файл в разных ветках. Справа вверху мы видим граф зависимости разных ревизий файла, справа внизу этот же график показан целиком для быстрого ориентирования. Оставшаяся часть окна отображает детали выбранной ревизии (в нашем случае это ревизия 2).

Граф выглядит в точно как в Git. У Perforce не было именованной ветки для сохранения фиксаций 1 и 2, так что он создал “анонимную” ветку в директории .git-fusion. Git Fusion поступит так же для именованных Git веток не соответствующих веткам в Perforce, но вы можете задать соответствие в конфигурационном файле.

Большинство происходящей магии скрыто от посторонних глаз, а в результате кто-то в команде может использовать Git, кто-то — Perforce и никто не будет подозревать о выборе других.

#### Git-Fusion

Если у вас есть (или вы можете получить) доступ к Perforce серверу, Git Fusion — это прекрасный способ подружить Git и Perforce. Конечно, требуется небольшая работа напильником, но в целом всё довольно интуитивно и просто. Это один из немногих разделов в этой главе, где мы не будем предупреждать вас об опасности использования всего функционала Git. Но Perforce не всеяден: если вы попытаетесь переписать опубликованную историю, Git Fusion отклонит изменения. Тем не менее, Git Fusion будет стараться изо всех сил, чтобы не нарушать ваших привычек при работе с Git. Вы даже можете использовать подмодули Git (хотя они и будут выглядеть странными для Perforce пользователей) и сливать ветки (на стороне Perforce это будет выглядеть как интеграция).

И даже в том случае, если вы не можете уговорить администратора настроить Git Fusion есть способ использовать Git и Perforce вместе.

#### GIT-P4

Git-p4 — это двусторонний мост между Git и Perforce. Он работает на стороне клиента, так что вам не нужен будет доступ к Perforce серверу (разумеется, вам по-прежнему понадобятся логин и пароль). Git-p4 не так гибок и полнофункционален, как Git Fusion, но он позволяет совершать большинство необходимых действий.

---

Исполняемый файл p4 должен быть доступен в PATH для использования git-p4. На момент написания книги он свободно доступен на <http://www.perforce.com/downloads/Perforce/20-User>.

---

Мы будем использовать описанный выше образ виртуальной машины Git Fusion, но мы будем напрямую обращаться к Perforce, минуя Git Fusion.

Для того, чтобы использовать команду p4 (от которой зависит git-p4), вам нужно будет установить следующие переменные окружения:

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

Как обычно при работе с Git, первая команда — это клонирование:

```
$ git p4 clone //depot/www/live www-shallow
Importing from //depot/www/live into www-shallow
Initialized empty Git repository in /private/tmp/www-shallow/.git/
Doing initial import of //depot/www/live/ from revision #head into refs/remotes/p4/master
```

В терминах Git мы получим так называемую “поверхностную” копию: выкачивается лишь последняя ревизия. Помните, Perforce не предназначен для раздачи всех ревизий всем пользователям. Этого достаточно, чтобы использовать Git как Perforce клиент, но этого недостаточно для других задач.

Как только клонирование завершится, у нас будет Git репозиторий:

```
$ cd myproject
$ git log --oneline --all --graph --decorate
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of //depot/www/live/ from the st
```

Обратите внимание на наличие удалённого репозитория “p4”, соответствующего Perforce серверу; всё остальное выглядит как обычный клонированный репозиторий. Но давайте присмотримся повнимательней: на самом деле нет никакого удалённого репозитория!

```
$ git remote -v
```

В этом репозитории нет удалённых серверов. `git-p4` создал несколько ссылок для представления состояния на сервере, и они выглядят как удалённый сервер для `git log`, но таковым не являются и вы не можете отправлять изменения в них.

Что ж, приступим к работе. Предположим, вы сделали несколько изменений для очень важной фичи и готовы показать свои наработки остальным членам команды.

```
$ git log --oneline --all --graph --decorate
* 018467c (HEAD, master) Change page title
* c0fb617 Update link
* 70eaf78 (p4/master, p4/HEAD) Initial import of //depot/www/live/ from the state
```

Мы подготовили две фиксации для отправки на Perforce сервер. Давайте посмотрим, успели ли другие члены команды проделать какую-либо работу:

```
$ git p4 sync
git p4 sync
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12142 (100%)
$ git log --oneline --all --graph --decorate
* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Кажется, успели: `master` и `p4/master` разошлись. Система ветвления в Perforce *абсолютно* непохожа на Git, отправка слияний в Perforce не имеет смысла. `git-p4` рекомендует перемещать фиксации и даже предоставляет команду для этого:

```
$ git p4 rebase
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
No changes to import!
```

```
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
Applying: Update link
Applying: Change page title
index.html | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

Вы, возможно, скажете что `git p4 rebase` это всего лишь сокращение для `git p4 sync` с последующим `git rebase p4/master`. На самом деле, эта команда немного умнее, особенно при работе с несколькими ветками, но догадка вполне верна.

Теперь наша история снова линейна и мы готовы отправить изменения в Perforce. Команда `git p4 submit` попытается создать новые Perforce ревизии для всех фиксаций в Git между `p4/master` и `master`. Её запуск откроет ваш любимый редактор с примерно таким содержимым:

```
A Perforce Change Specification.
#
Change: The change number. 'new' on a new changelist.
Date: The date this specification was last modified.
Client: The client on which the changelist was created. Read-only.
User: The user who created the changelist.
Status: Either 'pending' or 'submitted'. Read-only.
Type: Either 'public' or 'restricted'. Default is 'public'.
Description: Comments about the changelist. Required.
Jobs: What opened jobs are to be closed by this changelist.
You may delete jobs from this list. (New changelists only.)
Files: What opened files from the default changelist are to be added
to this changelist. You may delete files from this list.
(New changelists only.)

Change: new

Client: john_bens-mbp_8487

User: john

Status: new

Description:
 Update link

Files:
 //depot/www/live/index.html # edit
```

```

#####
git author ben@straub.cc does not match your p4 account.
#####
Use option --preserve-user to modify authorship.
#####
Variable git-p4.skipUserNameCheck hides this message.
#####
everything below this line is just the diff
--- //depot/www/live/index.html 2014-08-31 18:26:05.000000000 0000
+++ /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/index.html
@@ -60,7 +60,7 @@
</td>
<td valign=top>
Source and documentation for
-
+
Jam/MR,
a software build tool.
</td>
```

Это практически те же данные, что вы увидели бы, запустив `p4 submit`, за исключением нескольких строк в конце, любезно вставленных `git-p4`. `git-p4` старается учитывать Git и Perforce настройки когда нужно предоставить имя для фиксации, но в некоторых случаях вы захотите изменить его. Например, если фиксация в Git была создана человеком, у которого нет Perforce аккаунта, вы всё равно захотите сделать автором фиксации его, а не себя.

`git-p4` вставил сообщение из фиксации Git в содержимое набора изменений Perforce, так что всё что нам остаётся сделать — это дважды сохранить и закрыть редактор (по одному разу на каждую фиксацию). В результате мы получим такой вывод:

```

$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens
Synchronizing p4 checkout...
... - file(s) up-to-date.
Applying dbac45b Update link
//depot/www/live/index.html#4 - opened for edit
Change 12143 created with 1 open file(s).
Submitting change 12143.
Locking 1 files ...
edit //depot/www/live/index.html#5
Change 12143 submitted.
Applying 905ec6a Change page title
//depot/www/live/index.html#5 - opened for edit
Change 12144 created with 1 open file(s).
Submitting change 12144.
Locking 1 files ...
```

```

edit //depot/www/live/index.html#6
Change 12144 submitted.
All commits applied!
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12144 (100%)
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
$ git log --oneline --all --graph --decorate
* 775a46f (HEAD, p4/master, p4/HEAD, master) Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head

```

Выглядит так, словно мы только что выполнили `git push`, что на самом деле очень близко к тому, что произошло.

Обратите внимание, что во время этого процесса каждая фиксация в Git превращается в отдельный набор изменений Perforce; если вы хотите слепить их воедино, вы можете сделать это с помощью интерактивного переноса фиксаций до выполнения `git p4 submit`. Ещё один важный момент: SHA-1 хеши фиксаций, превращённых в наборы изменений Perforce изменились: это произошло из-за того, что `git-p4` добавил строку в конец каждого сообщения:

```

$ git log -1
commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145
Author: John Doe <john@example.com>
Date: Sun Aug 31 10:31:44 2014 -0800

 Change page title

[git-p4: depot-paths = "//depot/www/live/": change = 12144]

```

Что произойдёт если вы попробуете отправить фиксацию-слияние? Давайте попробуем. Допустим, мы имеем такую историю:

```

$ git log --oneline --all --graph --decorate
* 3be6fd8 (HEAD, master) Correct email address
* 1dcbf21 Merge remote-tracking branch 'p4/master'
\\
| * c4689fc (p4/master, p4/HEAD) Grammar fix
* | cbacd0a Table borders: yes please
* | b4959b6 Trademark

```

```
//
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

История в Git разошлась с Perforce на фиксации 775a46f. В Git мы имеем две дополнительные фиксации, затем слияние с состоянием Perforce, затем ещё одна фиксация. Мы собираемся отправить эту историю в Perforce. Давайте посмотрим, что произойдёт:

```
$ git p4 submit -n
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487
Would apply
b4959b6 Trademark
cbacd0a Table borders: yes please
3be6fd8 Correct email address
```

Флаг `-n` — это сокращение для `--dry-run`, который, в свою очередь, пытается вывести результат выполнения отправки, как если бы отправка на самом деле произошла. В этом случае, похоже мы создадим три ревизии в Perforce, по одной для каждой не являющейся слиянием фиксации в Git. Звучит логично, давайте посмотрим что произойдёт на самом деле:

```
$ git p4 submit
[...]
$ git log --oneline --all --graph --decorate
* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address
* 1b79a80 Table borders: yes please
* 0097235 Trademark
* c4689fc Grammar fix
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

История стала линейной, словно мы переместили изменения перед отправкой (что на самом деле и произошло). Это означает, что вы можете свободно создавать ветки в Git, работать в них, сливать их, не боясь, что ваша история станет несовместима с Perforce. Если вы

можете переместить изменения, вы можете отправить их на Perforce сервер.

Если в вашем Perforce проекте несколько веток, не переживайте: git-p4 может организовать работу с ними, не сложнее, чем с обычными Git ветками. Предположим, ваш Perforce репозиторий имеет следующую структуру:

```
//depot
└── project
 ├── main
 └── dev
```

Также предположим, что ветка dev настроена следующим образом:

```
//depot/project/main/... //depot/project/dev/...
```

git-p4 может автоматически распознать эту ситуацию и выполнить нужные действия:

```
$ git p4 clone --detect-branches //depot/project@all
Importing from //depot/project@all into project
Initialized empty Git repository in /private/tmp/project/.git/
Importing revision 20 (50%)
 Importing new branch project/dev

 Resuming with change 20
Importing revision 22 (100%)
Updated branches: main dev
$ cd project; git log --oneline --all --graph --decorate
* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
| * a43cfae Populate //depot/project/main/... //depot/project/dev/....
| /
* 2b83451 Project init
```

Обратите внимание на “@all” в пути; она говорит git-p4 клонировать не только последнюю ревизию для указанного субдерева, но все ревизии, затрагивающие указанные пути. Это ближе к оригинальной концепции клонирования в Git, но если вы работаете с большим репозиторием, это может занять некоторое время.

Флаг `--detect-branches` указывает git-p4 использовать настройки веток Perforce для отображения на Git ветки. Если же таких настроек на

Perforce сервере нет (что вполне корректно для Perforce), вы можете указать их git-p4 вручную, получив аналогичный результат:

```
$ git init project
Initialized empty Git repository in /tmp/project/.git/
$ cd project
$ git config git-p4.branchList main:dev
$ git clone --detect-branches //depot/project@all .
```

Задав конфигурационный параметр `git-p4.branchList` равным `main:dev` мы указали git-p4, что “main” и “dev” — это ветки, и что вторая является потомком первой.

Если мы теперь выполним `git checkout -b dev p4/project/dev` и зафиксируем в ветке `dev` некоторые изменения, git-p4 будет достаточно смешлённым, чтобы догадаться, в какую ветку отправлять изменения при выполнении `git p4 submit`. К сожалению, git-p4 не позволяет использовать несколько веток в поверхностных копиях репозиториев; если у вас есть большой проект и вы хотите работать более чем в одной ветке, вам придётся выполнять `git p4 clone` для каждой ветки, в которую вы хотите отправлять изменения.

Для создания и интеграции веток вам нужно будет использовать Perforce клиент. git-p4 может только забирать изменения из Perforce и отправлять линейную историю обратно. Если вы произведёте слияние двух веток в Git и отправите изменения в Perforce, сохранятся лишь данные об изменении файлов, все метаданные об исходных ветках, участвующих в интеграции, будут потеряны.

## ЗАКЛЮЧЕНИЕ ПО GIT И PERFORCE

git-p4 позволяет использовать Git для работы с Perforce и он достаточно хорош в этом. Тем не менее, не стоит забывать, что источником данных по-прежнему остаётся Perforce, а Git используется лишь для локальной работы. Будьте осторожны с публикацией Git фиксаций: если у вас есть удалённый репозиторий, который используют другие люди, не публикуйте в нём фиксации, не отправленные на Perforce сервер.

Если вы хотите свободно смешивать Git и Perforce для контроля версий, уговорите администратора установить Git Fusion — он позволяет использовать Git в качестве полноценного клиента для Perforce сервера.

## Git и TFS

Git набирает популярность среди Windows-разработчиков и если вы один из них, то велика вероятность что вы пользовались Microsoft Team Foundation Server (TFS). TFS — это комплексное решение, включающее в себя систему отслеживание ошибок, систему учёта рабочего времени, решения для поддержки Scrum методологии, инструменты для проведения инспекции кода и собственно систему контроля версий. Здесь есть небольшая путаница: **TFS** — это сервер, поддерживающий управление версиями как с помощью Git, так и с помощью собственной СКВ — **TFVC** (Team Foundation Version Control). Поддержка Git появилась в TFS относительно недавно (начиная с 2013-й версии), так что когда идёт речь об управлении версиями в более ранних версиях TFS, имеется в виду именно TFVC.

Если вы оказались в команде, работающей с TFVC, но хотите использовать Git для управления версиями, есть проект, способный вам помочь.

### ИНСТРУМЕНТЫ ДЛЯ РАБОТЫ С TFVC

На самом деле, даже два проекта: `git-tf` и `git-tfs`.

`git-tfs` (расположившийся по адресу <https://github.com/git-tfs/git-tfs>) написан на .NET и (на момент написания этой книги) работает только на Windows. Он использует .NET привязки для библиотеки libgit2 для работы с Git репозиториями; это очень гибкая и эффективная библиотека, позволяющая выполнять множество низкоуровневых операций с Git репозиторием. Но libgit2 не полностью покрывает функционал Git, так что в некоторых случаях `git-tfs` вызывает консольный клиент Git, что делает его возможности по работе с репозиториями практически безграничными. Поддержка TFVC также впечатляет своей полнотой, ведь `git-tfs` использует “родные” .NET-сборки Visual Studio для работы с сервером. И это не означает, что вам нужен будет доступ к этим сборкам! Достаточно лишь установить свежую версию Visual Studio (любую, начиная с 2010-й, включая Express, начиная с 2012-й) или комплект средств разработки для Visual Studio (Visual Studio SDK).

`git-tf` (его можно найти на <https://gittf.codeplex.com>) написан на Java и его можно запустить практически на любом компьютере. Он взаимодействует с Git посредством библиотеки JGit (JVM-имплементация Git), что теоретически означает отсутствие каких-либо ограничение при работе с Git. К сожалению, поддержка TFVC не так полна, как у `git-tfs`: например, не поддерживаются ветки.

Итак, у каждого из двух проектов есть сильные и слабые стороны и существуют ситуации, в которых один окажется предпочтительнее другого. В этой книге мы вкратце рассмотрим каждый из них.

---

Если вы хотите опробовать примеры из книги, вам понадобится доступ к TFVC репозиторию. Они достаточно редко встречаются на просторах Интернета, возможно, вам придётся создать репозиторий самим. Можем посоветовать использовать Codeplex (<https://www.codeplex.com>) или Visual Studio Online (<http://www.visualstudio.com>).

---

## НАЧАЛО РАБОТЫ: GIT-TF

Как и в большинстве других примеров, первым делом мы клонируем репозиторий. Вот как это выглядит с использованием git-tf:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main project_git
```

Первый аргумент — это URL TFVC коллекции, второй представляет собой строку вида `$/project/branch`, и третий — это путь к локальному Git репозиторию, который будет создан (третий параметр опционален). git-tf поддерживает одновременную работу только с одной веткой; если вы хотите работать с разными TFVC ветками, вам потребуется несколько копий репозитория.

Приведённая выше команда создаёт обычный Git репозиторий:

```
$ cd project_git
$ git log --all --oneline --decorate
512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

Это так называемая *поверхностная* копия, что означает, что в ней есть только последняя ревизия проекта. TFVC не предусматривает наличия полной копии репозитория на каждом клиенте, так что git-tf по умолчанию скачивает лишь последнюю ревизию, что намного быстрее.

Если вы никуда не торопитесь, можно выкачать и полную историю проекта, используя опцию `--deep`:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main \
project_git --deep
```

```

Username: domain\user
Password:
Connecting to TFS...
Cloning $/myproject into /tmp/project_git: 100%, done.
Cloned 4 changesets. Cloned last changeset 35190 as d44b17a
$ cd project_git
$ git log --all --oneline --decorate
d44b17a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Goodbye
126aa7b (tag: TFS_C35189)
8f77431 (tag: TFS_C35178) FIRST
0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
Team Project Creation Wizard

```

Обратите внимание на метки типа TFS\_C35189; это помогает проассоциировать Git фиксации с наборами изменений TFVC. Это очень удобно, потому что вы можете узнать, какие из фиксаций ассоциированы со слепком в TFVC с помощью простой команды. Это не обязательное поведение (вы можете выключить его, вызвав `git config git-tf.tag false`) — `git-tf` и так хранит соответствия в файле `.git/git-tf`.

## НАЧАЛО РАБОТЫ: GIT-TFS

Клонирование в `git-tfs` слегка отличается. Взгляните-ка:

```

PS> git tfs clone --with-branches \
 https://username.visualstudio.com/DefaultCollection \
 $/project/Trunk project_git
Initialized empty Git repository in C:/Users/ben/project_git/.git/
C15 = b75da1aba1ffb359d00e85c52acb261e4586b0c9
C16 = c403405f4989d73a2c3c119e79021cb2104ce44a
Tfs branches found:
- $/tfvc-test/featureA
The name of the local branch will be : featureA
C17 = d202b53f67bde32171d5078968c644e562f1c439
C18 = 44cd729d8df868a8be20438fdeeffb961958b674

```

Обратите внимание на флаг `--with-branches`.

`git-tfs` умеет сопоставлять ветки TFVC с ветками в Git и этот флаг говорит ему завести по локальной Git-ветке для каждой ветки в TFVC. Крайне рекомендуется использовать эту опцию, если вы использовали ветки в TFS. Но она не сработает для версий TFS ниже 2010-й: до этого релиза “ветки” были просто директориями и `git-tfs` неспособен отличить их от обычных директорий.

Давайте посмотрим на получившийся репозиторий:

```

PS> git log --oneline --graph --decorate --all
* 44cd729 (tfv/featureA, featureA) Goodbye
* d202b53 Branched from $/tfvc-test/Trunk
* c403405 (HEAD, tfv/default, master) Hello
* b75da1a New project
PS> git log -1
commit c403405f4989d73a2c3c119e79021cb2104ce44a
Author: Ben Straub <ben@straub.cc>
Date: Fri Aug 1 03:41:59 2014 +0000

Hello

git-tfs-id: [https://username.visualstudio.com/DefaultCollection]$/{myproject}/Trunk

```

Видим две локальные ветки — `master` и `featureA` — представляющие соответственно основную ветку разработки (`Trunk` в TFVC) и дочернюю ветку `featureA` в TFVC. Также вы можете видеть, что “удалённый репозиторий” `tfv` имеет две ссылки — `default` и `featureA` — соответствующие тем же веткам в TFVC. `git-tfs` также называет ветку с которой вы инициировали копирование `tfv/default`, имена остальных веток соответствуют таковым в TFVC.

Ещё одна стоящая внимание вещь: строки `git-tfs-id:` в сообщениях фиксаций. `git-tfs` использует их вместо меток для сопоставления наборов изменений из TFVC и фиксаций в Git. Как результат, ваши фиксации будут иметь различные SHA-1 хеши до и после отправки в TFVC.

## РАБОЧИЙ ПРОЦЕСС С GIT-TF[\$]

---

Независимо от того, какой конкретный инструмент для работы с TFVC вы используете, вам следует задать некоторые конфигурационные параметры для избежания проблем.

```
$ git config set --local core.ignorecase=true
$ git config set --local core.autocrlf=false
```

---

Очевидно, после клонирования проекта вам захочется поработать над ним. Но в TFVC и TFS есть несколько особенностей, осложняющих рабочий процесс:

1. Функциональные ветки (feature branches), не представленные на TFVC сервере добавляют сложности. Всё из-за того, что TFVC имеет **совершенно** другую концепцию ветвления, нежели Git.

2. Помните, что TFVC позволяет пользователям запретить изменения файлов другими пользователями. Разумеется, это не помешает вам изменить их в локальном репозитории, но вы не сможете отправить эти изменения на TFVC сервер пока не будет снят запрет.
3. В TFS существует понятие “курируемых” наборов изменений; это означает, что прежде чем изменения будут приняты сервером, они должны успешно пройти фазы сборки и тестирования. При этом используется функционал “откладывания изменений”, не рассматриваемый нами в деталях. Вы можете вручную эмулировать подобное поведение в git-tf, а git-tfs предоставляет специальную команду checkintool, способную работать с “курируемыми” наборами изменений.

Для краткости мы рассмотрим здесь простой сценарий работы, избегающий описанных особенностей.

## РАБОЧИЙ ПРОЦЕСС В GIT-TF

Предположим, вы проделали некую работу, зафиксировали несколько изменений в ветке master и готовы поделиться результатом. Вот как выглядит Git репозиторий:

```
$ git log --oneline --graph --decorate --all
* 4178a82 (HEAD, master) update code
* 9df2ae3 update readme
* d44b17a (tag: TFS_C35190, origin_tfs/tfs) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
 Team Project Creation Wizard
```

Мы хотим взять слепок на момент фиксации 4178a82 и отправить его на TFVC сервер. Но для начала давайте проверим наличие наработок от других членов команды:

```
$ git tf fetch
Username: domain\user
Password:
Connecting to TFS...
Fetching $/myproject at latest changeset: 100%, done.
Downloaded changeset 35320 as commit 8ef06a8. Updated FETCH_HEAD.
$ git log --oneline --graph --decorate --all
```

```
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
| * 4178a82 (HEAD, master) update code
| * 9df2ae3 update readme
|/
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
 Team Project Creation Wizard
```

Похоже, вы работаете над проектом не в одиночку. История разошлась. Git очень хорош в таких ситуациях, но в случае TFVC есть два пути:

1. Как пользователь Git, вы, наверняка, заходите создать фиксацию-слияние (в конце-концов именно так поступает `git pull`). В `git-tf` есть специальная команда для этого: `git tf pull`. Но помните, что TFVC сервер мыслит несколько иначе, и если вы отправите фиксацию-слияние, ваша история станет выглядеть по-разному со стороны Git и TFVC, что может привести к путанице. Хотя, если вы хотите отправить все изменения одним набором, это самый лёгкий способ.
2. Перенос фиксаций сделает историю линейной, а значит мы сможем сопоставить каждой Git-фиксации набор изменений в TFVC. Мы рекомендуем использовать именно этот способ как наиболее гибкий. В `git-tf` для этого служит команда `git tf pull --rebase`.

Выбор за вами. Мы же последуем собственным советам:

```
$ git rebase FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
 Team Project Creation Wizard
```

Теперь всё готово к отправке данных на TFVC сервер. `git-tf` предоставляет вам выбор: собрать все изменения воедино и сделать из них один набор изменений (опция `--shallow`, по умолчанию включённая), или создать отдельный набор изменений для каждой фиксации в Git (опция `--deep`). В этом примере мы создадим один набор изменений:

```
$ git tf checkin -m 'Updating readme and code'
Username: domain\user
Password:
Connecting to TFS...
Checking in to $/myproject: 100%, done.
Checked commit 5a0e25e in as changeset 35348
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, tag: TFS_C35348, origin_tfs/tfs, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
 Team Project Creation Wizard
```

Появилась новая метка `TFS_C35348`, указывающая на то, что TFVC сохранил слепок состояния `5a0e25e` под этим идентификатором. Обращаем ваше внимание, что не каждая фиксация в Git имеет аналог в TFVC; например `6eb3eb5` не представлена на сервере.

Такой вот рабочий процесс. И ещё несколько важных моментов:

- Нет веток. `git-tf` умеет создавать Git-репозитории, соответствующие единственной ветке в TFVC.
- Делитесь наработками либо через TFVC-сервер, либо через Git-сервер, не используйте их одновременно. Разные `git-tf` клонны одного и того же TFVC-репозитория могут иметь различные SHA-1 хеши, что сулит нескончаемую головную боль.
- Если ваш рабочий процесс выстроен таким образом, что вы делитесь наработками через Git и лишь периодически синхронизируетесь с TFVC, не используйте более одного Git репозитория.

## РАБОЧИЙ ПРОЦЕСС В GIT-TFS

Давайте пробежимся по тому же сценарию в git-tfs. Вот новые фиксации в ветке master в нашем Git репозитории:

```
PS> git log --oneline --graph --all --decorate
* c3bd3ae (HEAD, master) update code
* d85e5a2 update readme
| * 44cd729 (tfss/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|
* c403405 (tfss/default) Hello
* b75da1a New project
```

Проверим, что успели сделать другие:

```
PS> git tfs fetch
C19 = ae74a0313de0a391940c999e51c5c381d91d
PS> git log --all --oneline --graph --decorate
* aea74a0 (tfss/default) update documentation
| * c3bd3ae (HEAD, master) update code
| * d85e5a2 update readme
|
| * 44cd729 (tfss/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|
* c403405 Hello
* b75da1a New project
```

В TFVC появились свежие изменения пока мы работали, соответствующие фиксации aea74a0 и удалённая ветка tfss/default передвинулась.

Как и в случае с git-tf, у нас есть два пути решения этой проблемы:

1. Переместить изменения и сделать историю линейной.
2. Произвести слияние, сохранив историческую достоверность.

Мы хотим, чтобы каждой фиксации в Git соответствовал набор изменений в TFVC, так что мы будем перемещать изменения и делать историю линейной.

```
PS> git rebase tfss/default
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
PS> git log --all --oneline --graph --decorate
```

```
* 10a75ac (HEAD, master) update code
* 5cec4ab update readme
* aea74a0 (tfs/default) update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Теперь мы готовы отправить наши наработки на TFVC сервер. Для этого мы используем команду `gcheckin`, которая сопоставляет каждой фиксации в Git новый набор изменений в TFVC (команда `checkin` создала бы только один набор изменений, примерно как опция `squash` при интерактивном перемещении изменений).

```
PS> git tfs gcheckin
Working with tfs remote: default
Fetching changes from TFS to minimize possibility of late conflict...
Starting checkin of 5cec4ab4 'update readme'
 add README.md
C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017
Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto new TFS-commit...
Rebase done successfully.
Starting checkin of b1bf0f99 'update code'
 edit .git\dfs\default\workspace\ConsoleApplication1\ConsoleApplication1\Program.cs
C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b
Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto new TFS-commit...
Rebase done successfully.
No more to gcheckin.
PS> git log --all --oneline --graph --decorate
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Обратите внимание, как после каждой удачной ревизии на TFVC сервере `git-tfs` перемещает оставшиеся к отправке изменения на вновь созданные фиксации. Это необходимо, потому что `git-tfs` добавляет строку `git-tfs-id` к сообщениям фиксаций, меняя, таким образом, их SHA-1 хеши. Это запланированное поведение и вам не о чём беспокоиться, просто помните об этом, особенно если вы публикуете эти фиксации где-либо ещё.

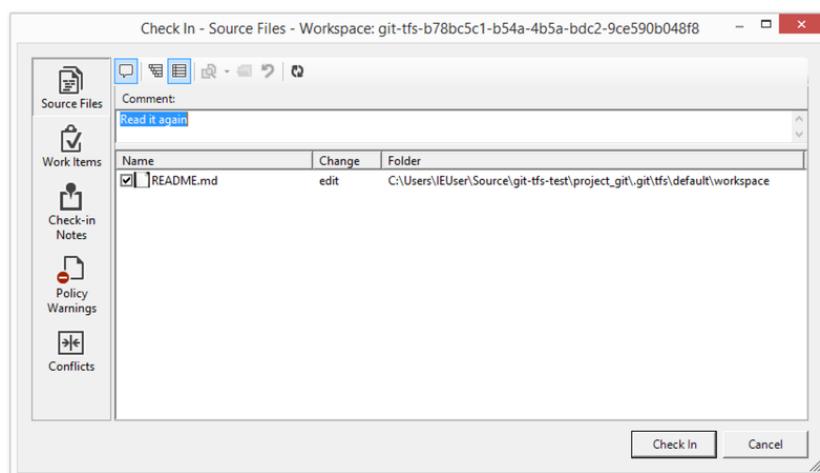
У TFS много козырей в рукаве, типа курируемых наборов изменений, привязки ревизий к задачам, инспекция кода и прочее. Возможно, кому-то покажется сложной работа с этими возможностями через командную строку. К счастью, в git-tfs вы можете использовать графическую утилиту:

```
PS> git tfs checkin tool
PS> git tfs ct
```

Выглядит она примерно так:

**FIGURE 9-3**

Графическая утилита git-tfs.



Она покажется знакомой для пользователей TFS, потому что это тот же самый диалог, что вызывается из Visual Studio.

Также git-tfs позволяет управлять ветками в TFVC из Git репозитория. Например, создадим новую ветку:

```
PS> git tfs branch $/tfvc-test/featureBee
The name of the local branch will be : featureBee
C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5
PS> git log --oneline --graph --decorate --all
* 1d54865 (tfv/featureBee) Creation branch $/myproject/featureBee
* ff04e7c (HEAD, tfv/default, master) update code
* 71a5ddc update readme
*aea74a0 update documentation
| * 44cd729 (tfv/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
```

```
!/
* c403405 Hello
* b75da1a New project
```

Создание ветки в TFVC означает создание новой ревизии, с которой и стартует ветка, в Git это выглядит как очередная фиксация изменений. Обратите внимание, что `git-tfs` **создал** удалённую ветку `tfs/featureBee`, но указатель HEAD всё ещё находится на ветке `master`. Если вы хотите продолжать работу в новой ветке, вам нужно базировать новые изменения на фиксации `1d54865`, создав начиная с неё новую ветку.

## ЗАКЛЮЧЕНИЕ ПО GIT И TFS

`git-tf` и `git-tfs` — отличные инструменты для взаимодействия с TFVC сервером. Они позволяют использовать преимущества Git для работы в локальном репозитории, избегая постоянных взаимодействий с центральным TFVC сервером. Это упрощает вашу жизнь, но не заставляет ваших коллег также переходить на Git. Если вы работаете под Windows (что вполне вероятно, раз уж вы используете TFS), тогда `git-tfs` будет наиболее разумным выбором, так как его функционал наиболее полон; но если вы используете другую платформу, вам придётся использовать более ограниченный `git-tf`. Как и с большинством других описываемых в этой главе инструментов, вам следует выбрать единственный “источник правды”: вы будете делиться наработками либо через Git, либо через TFVC, но никак не через обе системы сразу.

## Миграция на Git

Если у вас уже есть кодовая база в другой СКВ, но вы решили начать использовать Git, вам необходимо так или иначе перенести миграцию проекта. В этом разделе описаны некоторые существующие варианты импорта для распространённых систем, а затем показано, как разрабатывать собственные нестандартные варианты импорта. Вы узнаете, как импортировать данные из некоторых распространённых профессиональных СКВ. Поскольку они используются большинством разработчиков, для них легко найти качественные инструменты миграции.

## Subversion

Если вы читали предыдущий раздел про использование `git svn`, вы уже должны знать, как использовать команду `git svn clone` чтобы склонировать Subversion репозиторий. После этого вы можете прекратить использовать Subversion и перейти на Git. Сразу же после клонирования вам будет доступная вся история репозитория, хотя сам процесс получения копии может затянуться.

В добавок к этому импортирование не идеально, так что вы, возможно, захотите сделать его как можно более правильно с первой попытки. И первая проблема — это информация об авторстве. В Subversion на каждого участника рабочего процесса заведён пользователь, информация о пользователе сохраняется вместе с каждой ревизией. В предыдущем разделе вы могли видеть пользователя `schacon` в некоторых местах, типа вывода команды `blame` или `git svn log`. Если вы хотите видеть подробную информацию об авторстве в Git, вам потребуется задать соответствие между пользователями Subversion и авторами в Git. Создайте файл `users.txt` со следующим содержимым:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Чтобы получить список имён пользователей в SVN, выполните следующее:

```
$ svn log --xml | grep author | sort -u | \
perl -pe 's/.*/>(.*)<.*$/\$1 = /'
```

Эта команда генерирует XML документ, оставляет только строчки с авторами, избавляется от дубликатов, а затем обрезает XML-теги. (Очевидно, она сработает только на компьютерах с установленными `grep`, `sort` и `perl`.) Вы можете направить вывод этой команды в файл `users.txt`, а затем просто дописать Git авторов в каждой строке.

Затем вы можете передать этот файл `git svn`, чтобы тот мог проассоциировать авторов. Также вы можете указать `git svn` не включать метаданные, обычно вставляемые в сообщения фиксаций, передав флаг `--no-metadata` командам `clone` или `init`. Итого, команда `import` примет вид:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
--authors-file=users.txt --no-metadata -s my_project
```

Теперь у вас будет красивая копия Subversion репозитория в директории `my_project`. Вместо фиксаций типа

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date: Sun May 3 00:12:22 2009 +0000

fixed install - go to trunk

git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-
be05-5f7a86268029
```

вы получите следующее:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@gmail.com>
Date: Sun May 3 00:12:22 2009 +0000
```

```
fixed install - go to trunk
```

Теперь не только поле с информацией об авторстве выглядит лучше, но и `git-svn-id` не мозолит глаза.

Также вам следует немного “вычистить” репозиторий сразу после импорта. Во-первых, следует удалить ненужные ссылки, устанавливаемые `git svn`. Вначале переместим метки, чтобы они действительно стали метками, а не странными удалёнными ветками, а затем удалим остальное, сделав все ветки локальными.

Чтобы переместить метки, выполните следующие команды:

```
$ cp -Rf .git/refs/remotes/origin/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/origin/tags
```

Они берут все удалённые ветки, начинавшиеся с `remotes/origin/tags/` и делают из них настоящие легковесные метки.

Далее, сделайте остальные ветки, начинающиеся с `refs/remotes`, локальными, выполнив следующее:

```
$ cp -Rf .git/refs/remotes/* .git/refs/heads/
$ rm -Rf .git/refs/remotes
```

Теперь все ветки и метки из Subversion стали настоящими Git ветками и метками соответственно. Последнее, что нужно сделать — это добавить ваш Git сервер в качестве удалённого репозитория и залить данные на него. Вот пример добавления удалённого репозитория:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Так как вы хотите отправить все ваши ветки и метки, выполните это:

```
$ git push origin --all
```

Наконец, все ваши ветки и метки перенесены на Git сервер и облагорожены!

## Mercurial

Из-за того что Mercurial и Git обладают похожей моделью ветвления, а также из-за того что Git несколько более гибок, перенос репозитория из Mercurial в Git довольно прост; можете использовать инструмент hg-fast-export, который можно найти здесь:

```
$ git clone http://repo.or.cz/r/fast-export.git /tmp/fast-export
```

Первым делом нужно получить полную копию интересующего Mercurial репозитория:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

Следующим шагом создадим файл соответствия авторов. Mercurial менее строг к данным об авторстве коммитов, так что придётся слегка навести порядок. Вот односстрочник для bash, который сгенерирует заготовку:

```
$ cd /tmp/hg-repo
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

Пройдёт несколько секунд, в зависимости от размера репозитория, и вы получите файл `/tmp/authors` со следующим содержимым:

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

В примере выше, один и тот же человек (Боб) вносил изменения под пятью различными именами, лишь одно из которых правильное, а одно и вовсе не соответствует формату Git. `hg-fast-export` позволяет быстро исправить ситуацию, добавив `={new name and email address}` к каждой строке, которую мы хотим изменить; чтобы оставить имя как есть, просто удалите нужные строки. Если же все имена выглядят хорошо, этот файл и вовсе не потребуется. В нашем примере мы хотим чтобы данные выглядели так:

```
bob=Bob Jones <bob@company.com>
bob@localhost=Bob Jones <bob@company.com>
bob jones <bob <AT> company <DOT> com>=Bob Jones <bob@company.com>
bob <bob@company.com>=Bob Jones <bob@company.com>
```

Затем нужно создать Git репозиторий и запустить экспорт:

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

Флаг `-r` указывает на подлежащий конвертации Mercurial репозиторий, а флаг `-A` задаёт файл с соответствиями между авторами. Скрипт пробегается по наборам изменений Mercurial и преобразует их в скрипт для `fast-import` в Git (мы поговорим об этом инструменте чуть позже). Процесс конвертации займет некоторое время (хотя и *намного* меньше, чем при конвертации по сети), а мы пока можем наблюдать за подробным выводом в консоли:

```
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
Loaded 4 authors
master: Exporting full revision 1/22208 with 13/0/0 added/changed/removed files
master: Exporting simple delta revision 2/22208 with 1/1/0 added/changed/removed files
master: Exporting simple delta revision 3/22208 with 0/1/0 added/changed/removed files
```

```
[...]
master: Exporting simple delta revision 22206/22208 with 0/4/0 added/changed/removed
master: Exporting simple delta revision 22207/22208 with 0/2/0 added/changed/removed
master: Exporting thorough delta revision 22208/22208 with 3/213/0 added/changed/removed
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
[...]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
git-fast-import statistics:

Alloc'd objects: 120000
Total objects: 115032 (208171 duplicates))
blobs : 40504 (205320 duplicates 26117 deltas of 39602
trees : 52320 (2851 duplicates 47467 deltas of 47599
commits: 22208 (0 duplicates 0 deltas of 0
tags : 0 (0 duplicates 0 deltas of 0
Total branches: 109 (2 loads)
marks: 1048576 (22208 unique)
atoms: 1952
Memory total: 7860 KiB
pools: 2235 KiB
objects: 5625 KiB

pack_report: getpagesize() = 4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit = 8589934592
pack_report: pack_used_ctr = 90430
pack_report: pack_mmap_calls = 46771
pack_report: pack_open_windows = 1 / 1
pack_report: pack_mapped = 340852700 / 340852700

$ git shortlog -sn
369 Bob Jones
365 Joe Smith
```

Вот, собственно, и всё. Все Mercurial метки были преобразованы в метки Git, а ветки и закладки — в ветки Git. Теперь можно отправить репозиторий на новый Git сервер:

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin --all
```

## Perforce

Следующей системой из которой мы импортируем репозиторий станет Perforce. Вы уже знаете, что существует два способа подружить Git и Perforce: `git-p4` и Git Fusion.

### PERFORCE GIT FUSION

Git Fusion делает процесс переноса вполне безболезненным. Просто настройте проект, соответствия между пользователями и ветки в конфигурационном файле как показано в “[Git Fusion](#)” и клонируйте репозиторий. В результате вы получите настоящий Git репозиторий, который, при желании, можно сразу же отправлять на удалённый Git сервер. Вы даже можете использовать Perforce в качестве такового.

### GIT-P4

`git-p4` также можно использовать для переноса репозитория. В качестве примера мы импортируем проект “Jam” из публичного депо Perforce.

Вначале нужно указать адрес депо в переменной окружения `P4PORT`.

```
$ export P4PORT=public.perforce.com:1666
```

---

Для дальнейших экспериментов вам понадобится доступ к Perforce депо. Мы используем общедоступное депо на [public.perforce.com](http://public.perforce.com), но вы можете взять любое другое, к которому у вас есть доступ.

---

Запустите команду `git p4 clone` чтобы импортировать проект “Jam” с Perforce сервера, передав ей путь к проекту в депо и директорию, в которую хотите импортировать репозиторий:

```
$ git-p4 clone //guest/perforce_software/jam@all p4import
Importing from //guest/perforce_software/jam@all into p4import
Initialized empty Git repository in /private/tmp/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 9957 (100%)
```

Конкретно этот проект имеет одну ветку, но если бы их было несколько, вы бы просто могли передать флаг `--detect-branches` в `git p4 clone`. Перечитайте раздел “Ветвление” для подробностей.

На данном этапе репозиторий почти готов. Если вы зайдёте в директорию `p4import` и выполните `git log`, вы увидите результат:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@perforce.com>
Date: Wed Feb 8 03:13:27 2012 -0800

 Correction to line 355; change to .

[git-p4: depot-paths = "//public/jam/src/": change = 8068]

commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Author: kwirth <kwirth@perforce.com>
Date: Tue Jul 7 01:35:51 2009 -0800

 Fix spelling error on Jam doc page (cumulative -> cumulative).

[git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

`git-p4` оставил идентификаторы в сообщениях всех фиксаций. Ничего страшного нет в том, чтобы оставить всё как есть, особенно если вы захотите сослаться на номер ревизии в Perforce в будущем. Если же вы хотите убрать эти строки, теперь — прежде чем приступить к работе с репозиторием — самое время для этого. Вы можете использовать `git filter-branch` чтобы удалить идентификаторы из всех сообщений одним махом:

```
$ git filter-branch --msg-filter 'sed -e "/^\\[git-p4:/d"
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

Если вы сейчас выполните `git log`, вы увидите, что SHA-1 хеши фиксаций изменились, а строки `git-p4` исчезли из сообщений:

```
$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Author: giles <giles@giles@perforce.com>
Date: Wed Feb 8 03:13:27 2012 -0800
```

```

Correction to line 355; change to .

commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date: Tue Jul 7 01:35:51 2009 -0800

Fix spelling error on Jam doc page (cumulative -> cumulative).
```

Теперь ваш репозиторий готов к отправке на Git сервер.

## TFS

Если вы переходите с TFVC на Git, вам захочется получить как можно более точную копию репозитория. Поэтому, несмотря на то, что мы рассматривали `git-tfs` и `git-tf` в предыдущих разделах, здесь мы сосредоточимся лишь на использовании `git-tfs`, потому что этот инструмент поддерживает ветки, чего нет в `git-tf`.

---

Это дорога в один конец. Получившийся Git репозиторий невозможно будет подключить к TFVC.

---

Первым делом нужно задать соответствия между пользователями. TFVC не следит за данными, сохраняемыми в поле “автор” наборов изменений, Git же ожидает увидеть там человекопонятное имя и адрес электронной почты. Вы можете получить список всех авторов с помощью консольного клиента `tf`:

```
PS> tf history $/myproject -recursive > AUTHORS_TMP
```

Эта команда пробегается по всем ревизиям проекта и сохраняет информацию о них в файл `AUTHORS_TMP`, из которого мы впоследствии вытянем пользователей (2-я колонка). Откройте этот файл и запомните начало и конец колонки с пользователями, а затем используйте следующую команду (параметр `11-20` — это и есть границы колонки с пользователями):

```
PS> cat AUTHORS_TMP | cut -b 11-20 | tail -n+3 | uniq | sort > AUTHORS
```

Команда `cut` оставляет символы с 11-го по 20-й из каждой строки. Команда `tail` пропускает первые две строки с заголовком и ASCII-art’ом. Результат направляется в команду `uniq`, которая избавляется от дубликатов, её вывод сортируется и сохраняется в файл `AUTHORS`.

Далее необходимо поработать руками: для того, чтобы git-tfs распознал записи в этом файле, они должны иметь следующий формат:

```
DOMAIN\username = User Name <email@address.com>
```

Часть слева от знака равенства — это поле “User” из TFVC, а часть справа — соответствующий ему автор в Git.

Как только этот файл готов, необходимо сделать полную копию TFVC проекта:

```
PS> git tfs clone --with-branches --authors=AUTHORS https://username.visualstudio.co
```

Затем вы, возможно, захотите избавиться от строчек с git-tfs-id в сообщениях фиксаций. Следующая команда сделает это:

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$/g"' -- --all
```

Она использует утилиту sed из пакета git-bash чтобы заменить все строки, начинающиеся с git-tfs-id: на пустые, которые Git проигнорирует.

Теперь всё готово. Можете добавить новый удалённый репозиторий, отправить изменения в него и ваша команда может начинать работу с Git.

## A Custom Importer

If your system isn’t one of the above, you should look for an importer online – quality importers are available for many other systems, including CVS, Clear Case, Visual Source Safe, even a directory of archives. If none of these tools works for you, you have a more obscure tool, or you otherwise need a more custom importing process, you should use `git fast-import`. This command reads simple instructions from stdin to write specific Git data. It’s much easier to create Git objects this way than to run the raw Git commands or try to write the raw objects (see [Chapter 10](#) for more information). This way, you can write an import script that reads the necessary information out of the system you’re importing from and prints straightforward instructions to stdout. You can then run this program and pipe its output through `git fast-import`.

To quickly demonstrate, you’ll write a simple importer. Suppose you work in `current`, you back up your project by occasionally copying the directory into a time-stamped `back_YYYY_MM_DD` backup directory, and you want to import this into Git. Your directory structure looks like this:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

In order to import a Git directory, you need to review how Git stores its data. As you may remember, Git is fundamentally a linked list of commit objects that point to a snapshot of content. All you have to do is tell `fast-import` what the content snapshots are, what commit data points to them, and the order they go in. Your strategy will be to go through the snapshots one at a time and create commits with the contents of each directory, linking each commit back to the previous one.

As we did in “[An Example Git-Enforced Policy](#)”, we’ll write this in Ruby, because it’s what we generally work with and it tends to be easy to read. You can write this example pretty easily in anything you’re familiar with – it just needs to print the appropriate information to `stdout`. And, if you are running on Windows, this means you’ll need to take special care to not introduce carriage returns at the end your lines – git `fast-import` is very particular about just wanting line feeds (LF) not the carriage return line feeds (CRLF) that Windows uses.

To begin, you’ll change into the target directory and identify every subdirectory, each of which is a snapshot that you want to import as a commit. You’ll change into each subdirectory and print the commands necessary to export it. Your basic main loop looks like this:

```
last_mark = nil

loop through the directories
Dir.chdir(argv[0]) do
 Dir.glob("*").each do |dir|
 next if File.file?(dir)

 # move into the target directory
 Dir.chdir(dir) do
 last_mark = print_export(dir, last_mark)
 end
 end
end
```

You run `print_export` inside each directory, which takes the manifest and mark of the previous snapshot and returns the manifest and mark of this one; that way, you can link them properly. “Mark” is the `fast-import` term for an identifier you give to a commit; as you create commits, you give each one a

mark that you can use to link to it from other commits. So, the first thing to do in your `print_export` method is generate a mark from the directory name:

```
mark = convert_dir_to_mark(dir)
```

You'll do this by creating an array of directories and using the index value as the mark, because a mark must be an integer. Your method looks like this:

```
$marks = []
def convert_dir_to_mark(dir)
 if !$marks.include?(dir)
 $marks << dir
 end
 ($marks.index(dir) + 1).to_s
end
```

Now that you have an integer representation of your commit, you need a date for the commit metadata. Because the date is expressed in the name of the directory, you'll parse it out. The next line in your `print_export` file is

```
date = convert_dir_to_date(dir)
```

where `convert_dir_to_date` is defined as

```
def convert_dir_to_date(dir)
 if dir == 'current'
 return Time.now().to_i
 else
 dir = dir.gsub('back_', '')
 (year, month, day) = dir.split('_')
 return Time.local(year, month, day).to_i
 end
end
```

That returns an integer value for the date of each directory. The last piece of meta-information you need for each commit is the committer data, which you hardcode in a global variable:

```
$author = 'John Doe <john@example.com>'
```

Now you're ready to begin printing out the commit data for your importer. The initial information states that you're defining a commit object and what branch it's on, followed by the mark you've generated, the committer information and commit message, and then the previous commit, if any. The code looks like this:

```
print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

You hardcode the time zone (-0700) because doing so is easy. If you're importing from another system, you must specify the time zone as an offset. The commit message must be expressed in a special format:

```
data (size)\n(contents)
```

The format consists of the word `data`, the size of the data to be read, a newline, and finally the data. Because you need to use the same format to specify the file contents later, you create a helper method, `export_data`:

```
def export_data(string)
 print "data #[string.size]\n#[string]"
end
```

All that's left is to specify the file contents for each snapshot. This is easy, because you have each one in a directory – you can print out the `deleteall` command followed by the contents of each file in the directory. Git will then record each snapshot appropriately:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
 next if !File.file?(file)
 inline_data(file)
end
```

Note: Because many systems think of their revisions as changes from one commit to another, `fast-import` can also take commands with each commit to specify which files have been added, removed, or modified and what the new contents are. You could calculate the differences between snapshots and provide only this data, but doing so is more complex – you may as well give Git all the data and let it figure it out. If this is better suited to your data, check the `fast-import` man page for details about how to provide your data in this manner.

The format for listing the new file contents or specifying a modified file with the new contents is as follows:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Here, 644 is the mode (if you have executable files, you need to detect and specify 755 instead), and inline says you'll list the contents immediately after this line. Your `inline_data` method looks like this:

```
def inline_data(file, code = 'M', mode = '644')
 content = File.read(file)
 puts "#{code} #{mode} inline #{file}"
 export_data(content)
end
```

You reuse the `export_data` method you defined earlier, because it's the same as the way you specified your commit message data.

The last thing you need to do is to return the current mark so it can be passed to the next iteration:

```
return mark
```

If you are running on Windows you'll need to make sure that you add one extra step. As mentioned before, Windows uses CRLF for new line characters while git fast-import expects only LF. To get around this problem and make git fast-import happy, you need to tell ruby to use LF instead of CRLF:

```
$stdout.binmode
```

That's it. Here's the script in its entirety:

```
#!/usr/bin/env ruby

$stdout.binmode
$author = "John Doe <john@example.com>

$marks = []
def convert_dir_to_mark(dir)
 if !$marks.include?(dir)
 $marks << dir
 end
 ($marks.index(dir)+1).to_s
end

def convert_dir_to_date(dir)
 if dir == 'current'
```

```

 return Time.now().to_i
 else
 dir = dir.gsub('back_', '')
 (year, month, day) = dir.split('_')
 return Time.local(year, month, day).to_i
 end
end

def export_data(string)
 print "data #{string.size}\n#{string}"
end

def inline_data(file, code='M', mode='644')
 content = File.read(file)
 puts "#{code} #{mode} inline #{file}"
 export_data(content)
end

def print_export(dir, last_mark)
 date = convert_dir_to_date(dir)
 mark = convert_dir_to_mark(dir)

 puts 'commit refs/heads/master'
 puts "mark :#{mark}"
 puts "committer ${author} ${date} -0700"
 export_data("imported from #{dir}")
 puts "from :#[last_mark]" if last_mark

 puts 'deleteall'
 Dir.glob("**/*").each do |file|
 next if !File.file?(file)
 inline_data(file)
 end
 mark
end

Loop through the directories
last_mark = nil
Dir.chdir(argv[0]) do
 Dir.glob("*").each do |dir|
 next if File.file?(dir)

 # move into the target directory
 Dir.chdir(dir) do
 last_mark = print_export(dir, last_mark)
 end
 end
end

```

If you run this script, you'll get content that looks something like this:

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer John Doe <john@example.com> 1388649600 -0700
data 29
imported from back_2014_01_02deleteall
M 644 inline README.md
data 28
Hello

This is my readme.
commit refs/heads/master
mark :2
committer John Doe <john@example.com> 1388822400 -0700
data 29
imported from back_2014_01_04from :1
deleteall
M 644 inline main.rb
data 34
#!/bin/env ruby

puts "Hey there"
M 644 inline README.md
(...)
```

To run the importer, pipe this output through `git fast-import` while in the Git directory you want to import into. You can create a new directory and then run `git init` in it for a starting point, and then run your script:

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:

Alloc'd objects: 5000
Total objects: 13 (6 duplicates))
 blobs : 5 (4 duplicates) 3 deltas of
 trees : 4 (1 duplicates) 0 deltas of
 commits: 4 (1 duplicates) 0 deltas of
 tags : 0 (0 duplicates) 0 deltas of
Total branches: 1 (1 loads))
 marks: 1024 (5 unique))
 atoms: 2
Memory total: 2344 KiB
 pools: 2110 KiB
```

```

objects: 234 KiB

pack_report: getpagesize() = 4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit = 8589934592
pack_report: pack_used_ctr = 10
pack_report: pack_mmap_calls = 5
pack_report: pack_open_windows = 2 / 2
pack_report: pack_mapped = 1457 / 1457

```

As you can see, when it completes successfully, it gives you a bunch of statistics about what it accomplished. In this case, you imported 13 objects total for 4 commits into 1 branch. Now, you can run `git log` to see your new history:

```

$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date: Tue Jul 29 19:39:04 2014 -0700

 imported from current

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Author: John Doe <john@example.com>
Date: Mon Feb 3 01:00:00 2014 -0700

 imported from back_2014_02_03
```

There you go – a nice, clean Git repository. It's important to note that nothing is checked out – you don't have any files in your working directory at first. To get them, you must reset your branch to where `master` is now:

```

$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

You can do a lot more with the `fast-import` tool – handle different modes, binary data, multiple branches and merging, tags, progress indicators, and more. A number of examples of more complex scenarios are available in the `contrib/fast-import` directory of the Git source code.

## Заключение

После всего вышесказанного вы должны чувствовать себя уверенно, используя Git как клиент для других СКВ, или, импортируя практически любой существующий репозиторий в Git без потери данных. Следующая глава раскроет перед вами внутреннюю механику Git'a, так что вы будете способны контролировать каждый байт данных, если это потребуется.

# Git изнутри 10

Вы могли прочитать почти всю книгу перед тем, как приступить к этой главе, а могли только часть. Так или иначе, в данной главе рассматриваются внутренние процессы Git и особенности его реализации. На мой взгляд, изучение этого материала это основа понимания того, насколько Git полезный и мощный инструмент. Хотя некоторые утверждают, что изложение этого материала может сбить новичков с толку и оказаться для них неоправданно сложным. Именно поэтому эта глава отнесена в самый конец, так что вы можете начать читать её раньше или позже по ходу обучения. Мы оставляем выбор за вами.

Раз уж вы тут, приступим. Во-первых, напомню, что Git — это, по сути, контентно-адресуемая файловая система с пользовательским интерфейсом системы контроля версий поверх неё. Довольно скоро станет понятнее, что это значит.

На заре развития Git (примерно до версии 1.5) интерфейс был значительно сложнее, поскольку был больше похож на интерфейс доступа к файловой системе, чем на законченную систему контроля версий. За последние годы, интерфейс значительно очищен и упрощен до уровня аналогов; тем не менее, зачастую, сохраняется стереотип о том, что интерфейс у Git чересчур сложен и труден для изучения.

Контентно-адресуемая файловая система — основа Git, невероятно крута, именно её мы рассмотрим в этой главе в первую очередь; затем вы узнаете о транспортных механизмах и инструментах обслуживания репозитория, с которыми вам в своё время, возможно, придется столкнуться.

## Сантехника и Фарфор

В этой книге было описано, как пользоваться Git'ом, применяя примерно три десятка команд, например, `checkout`, `branch`, `remote` и т.п. Но так как сначала Git был скорее инструментарием для создания СКВ, чем СКВ, удобной для пользователей, в нём полно команд, выполняющих низкоуровневые операции, которые спроектированы так, чтобы их можно было использовать в цепочку в стиле UNIX, а также использовать в сценариях. Эти команды, как правило, называют служебными (“plumbing” — трубопровод), а ориентированные на пользователя называют пользовательскими (“porcelain” — фарфор).

Первые девять глав книги были посвящены практически лишь пользовательским командам. В данной главе же рассматриваются именно низкоуровневые служебные команды, дающие контроль над внутренними процессами Git'a и показывающие, как он работает и почему он работает так, а не иначе. Предполагается, что данные команды не будут использоваться напрямую из командной строки, а будут служить в качестве строительных блоков для новых команд и пользовательских сценариев.

Когда вы выполняете `git init` в новой или существовавшей ранее директории, Git создаёт подкаталог `.git`, в котором располагается почти всё, чем он заправляет. Если требуется выполнить резервное копирование или клонирование репозитория, достаточно скопировать всего лишь этот каталог, чтобы получить почти всё необходимое. И данная глава почти полностью посвящена его содержимому. Вот так он выглядит:

```
$ ls -F1
HEAD
config*
description
hooks/
info/
objects/
refs/
```

Там могут быть и другие файлы, но выше приведён листинг свежесозданного репозитория — это то, что вы увидите непосредственно после `git init`. Файл `description` используется только программой GitWeb, не обращайте на него внимание. Файл `config` содержит специфичные для этого репозитория

конфигурационные параметры, а в директории `info` расположен файл с глобальными настройками игнорирования файлов — он позволяет исключить файлы, которые вы не хотите помещать в `.gitignore`. В директории `hooks` располагаются клиентские и серверные триггеры, подробно рассмотренные в главе “[Git Hooks](#)”.

Итак, осталось четыре важных элемента: файлы `HEAD` и `index` (ещё не созданный) и директории `objects` и `refs`. Это ключевые элементы Git’а. В директории `objects` находится, собственно, база данных объектов Git; в `refs` — ссылки на объекты коммитов в этой базе (ветки); файл `HEAD` указывает на текущую ветку, а в файле `index` хранится содержимое индекса. Сейчас мы детально разберёмся с этими элементами, чтобы понять как работает Git.

## Объекты Git

Git — контентно-адресуемая файловая система. Здорово. Что это означает? А означает это, по сути, что Git — простое хранилище ключ-значение. Можно добавить туда любые данные, в ответ будет выдан ключ по которому их можно извлечь обратно. Например, можно воспользоваться служебной командой `hash-object`, добавляющей данные в директорию `.git` и возвращающей ключ. Для начала создадим новый Git-репозиторий и убедимся, что директория `objects` пуста:

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

Git проинициализировал директорию `objects` и создал в нём пустые поддиректории `pack` и `info`. Теперь добавим кое-какое текстовое содержимое в базу Git:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Ключ `-w` указывает команде `hash-object`, что объект необходимо сохранить, иначе команда просто вернёт ключ. Флаг `--stdin` указывает, что данные необходимо считать из потока стандартного ввода, в противном случае `hash-object` ожидает путь к файлу в качестве аргумента. Вывод команды — 40-символьная контрольная сумма. Это хеш SHA-1 — контрольная сумма содержимого и заголовка, который будет рассмотрен позднее. Теперь можно увидеть, в каком виде сохранены ваши данные:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Мы видим новый файл в директории `objects`. Это и есть начальное внутреннее представление данных в Git'е — один файл на единицу хранения с именем, являющимся контрольной суммой содержимого и заголовка. Первые два символа SHA-1 определяют поддиректорию файла внутри `objects`, остальные 38 — собственно, имя.

Получить обратно содержимое объекта можно командой `cat-file`. Она подобна швейцарскому ножу для проверки объектов в Git'е. Ключ `-p` означает автоматическое определение типа содержимого и вывод содержимого на печать в удобном виде:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Теперь вы умеете добавлять данные в Git и извлекать их обратно. То же самое можно делать и с файлами. Например, можно проверсиионировать один файл. Для начала, создадим новый файл и сохраним его в базе данных Git:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Теперь изменим файл и сохраним его в базе ещё раз:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Теперь в базе содержатся две версии файла, а также самый первый сохранённый объект:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Теперь можно откатить файл к его первой версии:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

или ко второй:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Однако запоминать хеш для каждой версии неудобно, к тому же теряется имя файла, сохраняется лишь содержимое. Объекты такого типа называют блобами (англ. blob — binary large object). Имея SHA-1 объекта, можно попросить Git показать нам его тип с помощью команды `cat-file -t`:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

## Деревья

Следующий тип объектов, который мы рассмотрим, — деревья — решают проблему хранения имен файлов, а также позволяют хранить группы файлов вместе. Git хранит данные сходным с файловыми системами UNIX способом, но в немного упрощённом виде. Содержимое хранится в деревьях и блобах, где дерево соответствует директории на файловой системе, а блоб более или менее соответствует inode или содержимому файла. Дерево может содержать одну или более записей, содержащих SHA-1 хеш,

соответствующий блобу или поддереву, права доступа к файлу, тип и имя файла. Например, дерево последнего коммита в проекте может выглядеть следующим образом:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859 README
100644 blob 8f94139338f9404f26296befa88755fc2598c289 Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0 lib
```

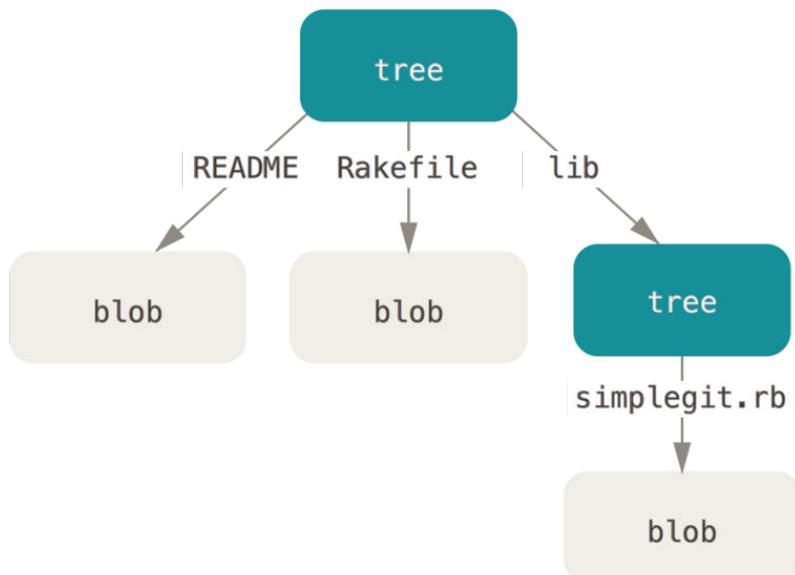
Запись `master^{tree}` указывает на дерево, соответствующее последнему коммиту ветки `master`. Обратите внимание, что поддиректория `lib` — не блоб, а указатель на другое дерево:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b simplegit.rb
```

Концептуально, данные хранятся в Git примерно так:

**FIGURE 10-1**

Упрощённая модель данных Git.



Можно создать дерево самому. Обычно Git создаёт деревья на основе индекса, затем сохраняя их в БД. Поэтому для создания дерева необходимо проиндексировать какие-нибудь файлы. Для создания индекса из одной записи — первой версии файла test.txt — воспользуемся низкоуровневой командой `update-index`. Данная команда может искусственно добавить более раннюю версию test.txt в новый индекс. Необходимо передать опции `--add`, т.к. файл ещё не существует в индексе (да и самого индекса ещё нет), и `--cacheinfo`, т.к. добавляемого файла нет в рабочей директории, но он есть в базе данных. Также необходимо передать права доступа, хеш и имя файла:

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

В данном случае права доступа — 100644, означают обычный файл. Другие возможные варианты: 100755 — исполняемый файл, 120000 — символьическая ссылка. Права доступа в Git'e сделаны по аналогии с режимами доступа в UNIX, но они гораздо менее гибки: данные три режима — единственные доступные для файлов (блобов) в Git'e (хотя существуют и другие режимы, используемые для директорий и дочерних модулей).

Теперь можно воспользоваться командой `write-tree` для сохранения индекса в виде дерева. Здесь опция `-w` не требуется — вызов `write-tree` автоматически создаёт дерево из индекса, если такого дерева ещё не существует:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Можно проверить, что мы действительно создали дерево:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

Давайте создадим новое дерево со второй версией файла test.txt и ещё одним файлом:

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

Теперь в индексе содержится новая версия файла test.txt и новый файл new.txt. Зафиксируем изменения, сохранив состояние индекса в новое дерево, и посмотрим, что из этого вышло:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
```

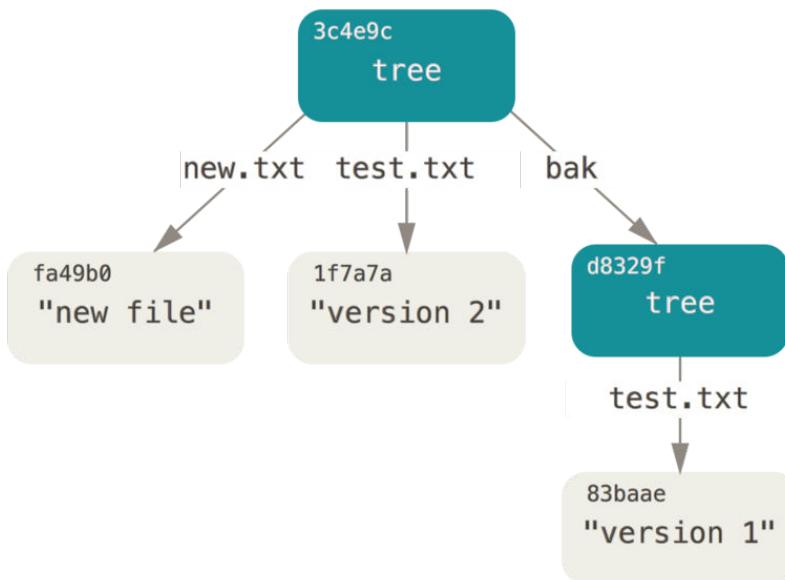
Обратите внимание, что в данном дереве находятся записи для обоих файлов, а также, что хеш файла test.txt это хеш “второй версии” этого файла (1f7a7a). Для интереса, добавим первое дерево как поддиректорию текущего. Вычитать дерево в индекс можно командой `read-tree`. В нашем случае, чтобы прочитать уже существующее дерево в индекс и сделать его поддеревом, необходимо использовать опцию `--prefix`:

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579 bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
```

Если бы вы сейчас вычитали только что сохранённое дерево в рабочую директорию, вы бы увидели два файла в корне рабочей директории и поддиректорию `bak` с первой версией файла `test.txt`. Представьте, что данные хранятся в Git следующим образом:

**FIGURE 10-2**

Структура данных Git для последнего дерева.



## Commit Objects

У вас есть три дерева, соответствующих разным состояниям проекта, но предыдущая проблема с необходимостью запоминать все три значения SHA-1, чтобы иметь возможность восстановить какое-либо из этих состояний, ещё не решена. К тому же у нас нет никакой информации о том, кто, когда и почему сохранил их. Такие данные — основная информация, хранимая в коммите.

Для создания коммита необходимо вызвать команду `commit-tree` и задать SHA-1 нужного дерева и, если необходимо, родительские коммиты. Для начала создадим коммит для самого первого дерева:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Просмотреть вновь созданный коммит можно командой `cat-file`:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
```

```
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700
first commit
```

Формат коммита прост: в нём указано дерево верхнего уровня, соответствующее состоянию проекта на некоторый момент; имена автора и коммиттера (берутся из полей конфигурации `user.name` и `user.email`); временная метка; пустая строка и сообщение коммита.

Далее, создадим ещё два коммита, каждый из которых будет ссылаться на предыдущий:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Каждый из этих коммитов указывает на одно из деревьев состояний проекта. Вы не поверите, но теперь у нас есть полноценная Git-история, которую можно посмотреть командой `git log`, указав хеш последнего коммита:

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:15:24 2009 -0700

 third commit

bak/test.txt | 1 +
1 file changed, 1 insertion(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:14:29 2009 -0700

 second commit

new.txt | 1 +
test.txt | 2 ++
2 files changed, 2 insertions(+), 1 deletion(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:09:34 2009 -0700
```

```
first commit

test.txt | 1 +
1 file changed, 1 insertion(+)
```

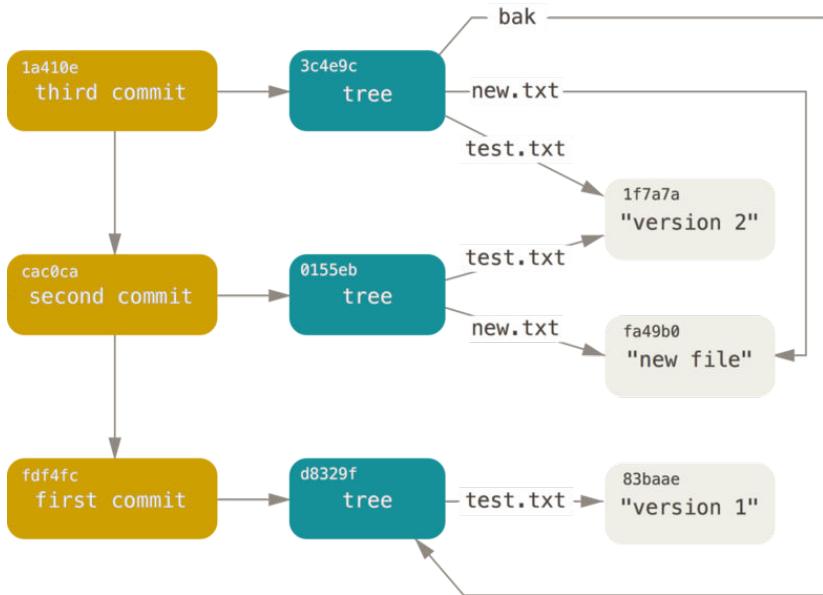
Здорово, правда? Мы только что выполнили несколько низкоуровневых операций и получили “настоящий” Git-репозиторий с историей без единой высокоуровневой команды! Именно так и работает Git, когда выполняются команды `git add` и `git commit` — сохраняет блобы для изменённых файлов, обновляет индекс, записывает его в виде дерева, и, наконец, фиксирует изменения в коммите, ссылающимся на это дерево и предшествующие коммиты. Эти три основных вида объектов Git’а — блоб, дерево и коммит — сохраняются в виде отдельных файлов в директории `.git/objects`. Вот все объекты, которые сейчас находятся в директории, с примером с комментариями чему они соответствуют:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Если пройти по всем внутренним ссылкам, получится граф объектов такой, как на рисунке:

**FIGURE 10-3**

Все объекты в директории Git.



## Хранение объектов

Ранее мы упоминали, что заголовок сохраняется вместе с содержимым. Давайте посмотрим, как Git сохраняет объекты на диске. Мы рассмотрим сохранение блоба — в данном случае это будет строка “как дела, Док?” — на языке Ruby.

Для запуска интерактивного интерпретатора воспользуйтесь командой `irb`:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git создаёт заголовок, начинающийся с типа объекта, в данном случае это блоб. Далее идут пробел, размер содержимого и в конце нулевой байт:

```
>> header = "blob #{content.length}\0"
=> "blob 16\0000"
```

Git склеивает заголовок и содержимое, а потом вычисляет SHA-1 сумму для полученного результата. В Ruby значение SHA-1 для строки можно получить, подключив соответствующую библиотеку командой `require` и затем вызвав `Digest::SHA1hexdigest()`:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git сжимает новые данные при помощи zlib, в Ruby это можно сделать с помощью одноимённой библиотеки. Сперва необходимо подключить её, а затем вызвать `Zlib::Deflate.deflate()`:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\x9C\xCA\xC90R04c(\xC9F,Q\xC8,V(-\xD0QH\xC90\xB6\aa\x00_\x1C\aa\x9D"
```

После этого запишем сжатую zlib'ом строку в объект на диск. Определим путь к файлу, который будет записан (первые два символа хеша используются в качестве названия директории, оставшиеся 38 — в качестве имени файла в ней). В Ruby для безопасного создания нескольких вложенных директорий можно использовать функцию  `FileUtils.mkdir_p()`. Далее, откроем файл вызовом `File.open()` и запишем сжатые данные вызовом `write()` для полученного файлового дескриптора:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

Вот и всё, мы создали корректный блоб Git. Все другие объекты создаются аналогично, меняется лишь запись о типе в заголовке (`blob`, `commit` либо `tree`). Стоит добавить, что хотя в блобе может

храниться почти любое содержимое, содержимое деревьев и коммитов записывается в очень строгом формате.

## Ссылки в Git

Для просмотра истории можно выполнить команду типа `git log 1a410e`, но вам всё ещё придётся запоминать, что именно коммит `1a410e` является последним, чтобы иметь возможность найти все наши объекты. Было бы неплохо, если бы существовал файл с понятным названием, содержащий этот SHA-1, чтобы можно было пользоваться им вместо хеша.

И такие файлы есть в Git! Они называются ссылками (“references” или, сокращённо, “refs”) и расположены в директории `.git/refs`. В нашем проекте эта директория пока пуста, но в ней уже прослеживается некая структура:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

Чтобы создать новую ссылку, которая поможет вам запомнить SHA-1 последнего коммита, по сути, необходимо выполнить примерно следующее:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Теперь в командах Git вместо SHA-1 можно использовать свежесозданную ссылку:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Тем не менее, редактировать файлы ссылок вручную не рекомендуется. Git предоставляет более безопасную и удобную команду — `update-ref` — для изменения ссылок:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

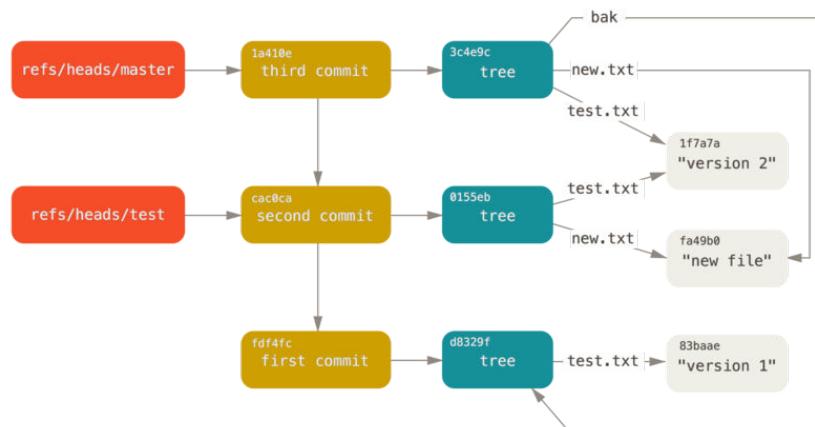
Вот что такое, по сути, ветка в Git — простой указатель (ссылка) на последнюю версию цепочки коммитов. Для создания ветки, соответствующей предыдущему коммиту, можно выполнить следующее:

```
$ git update-ref refs/heads/test cac0ca
```

Данная ветка будет содержать лишь коммиты по указанный, но не те, что были созданы после него:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769ccbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Теперь база данных Git схематично выглядит так, как показано на рисунке:

**FIGURE 10-4**

Объекты в директории .git, а также указатели на вершины веток

Когда выполняется команда `git branch` (имя ветки), Git, по сути, выполняет `update-ref` для добавления хеша последнего коммита текущей ветки под указанным именем в виде новой ссылки.

## HEAD

Как же Git получает хеш последнего коммита при выполнении `git branch` (имя ветки)? Ответ кроется в файле HEAD.

Файл HEAD — это символическая ссылка (не в терминах файловой системы) на текущую ветку. Символическая ссылка отличается от обычной тем, что она содержит не сам хеш SHA-1, а указатель на другую ссылку. Если вы заглянете внутрь HEAD, то увидите следующее:

```
$ cat .git/HEAD
ref: refs/heads/master
```

Если выполнить `git checkout test`, Git обновит содержимое файла:

```
$ cat .git/HEAD
ref: refs/heads/test
```

При выполнении `git commit` Git создаёт коммит, указывая его родителем объект, SHA-1 которого содержится в файле, на который ссылается HEAD.

При желании, можно вручную редактировать этот файл, но лучше использовать команду `symbolic-ref`. Получить значение HEAD этой командой можно так:

```
$ git symbolic-ref HEAD
refs/heads/master
```

Изменить значение HEAD можно так:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

Символическую ссылку на файл вне `.git/refs` поставить нельзя:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

## Метки

Мы рассмотрели три основных типа объектов Git, но есть ещё один. Метка очень похожа на коммит: она содержит имя автора метки, дату, сообщение и указатель. Разница же в том, что метка указывает на коммит, а не на дерево. Она похожа на ветку, которая никогда не перемещается: она всегда указывает на один и тот же коммит, просто давая ему понятное имя.

Как мы знаем из главы [Chapter 2](#), метки бывают двух типов: аннотированные и легковесные. Легковесную метку можно создать следующей командой:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Вот и всё! Легковесная метка — это ветка, которая никогда не перемещается. Аннотированная метка имеет более сложную структуру. При создании аннотированной метки Git создаёт специальный объект, на который будет указывать ссылка, а не просто указатель на коммит. Мы можем увидеть это, создав аннотированную метку (-a задаёт аннотированные метки):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Вот значение SHA-1 созданного объекта:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Теперь выполним `cat-file` для этого хеша:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700
```

```
test tag
```

Обратите внимание, в поле `object` записан SHA-1 помеченного коммита. Также стоит отметить, что это поле не обязательно должно указывать на коммит; вы можете пометить любой объект в Git. Например, в исходниках Git мейнтайнер добавил свой публичный GPG-ключ в блоб и пометил его. Увидеть этот ключ можно, выполнив команду:

```
$ git cat-file blob junio-gpg-pub
```

В репозитории ядра Linux также есть метка, указывающая не на коммит: самая первая метка указывает на дерево первичного импорта.

## Ссылки на удалённые ветки

Третий тип ссылок, который мы рассмотрим — ссылки на удалённые ветки. Если вы добавили удалённый репозиторий и отправили в него какие-нибудь изменения, Git сохранит последнее отправленное значение SHA-1 в директории `refs/remotes` для каждой отправленной ветки. Например, можно добавить удалённый репозиторий `origin` и отправить туда ветку `master`:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
 a11bef0..ca82a6d master -> master
```

Позже вы сможете посмотреть, где находилась ветка `master` с сервера `origin` во время последней синхронизации с ним, заглянув в файл `refs/remotes/origin/master`:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Ссылки на удалённые ветки отличаются от веток (ссылки в `refs/heads`) тем, что они считаются неизменяемыми. Это означает, что вы можете переключиться на любую из таких ссылок с помощью `git checkout`, но Git не установит HEAD на такую ссылку, а значит вы не сможете фиксировать свои изменения с помощью `git commit`. Git поддерживает удалённые ветки в качестве закладок на определённые состояния в удалённом репозитории во время последнего контакта с сервером.

## Pack-файлы

Рассмотрим объекты, хранящиеся в базе данных тестового Git репозитория. К этому моменту их должно быть 11 штук: 4 блоба, 3 дерева, 3 коммита и одна метка:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769ccbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git использует zlib для сжатия содержимого этих файлов; к тому же у нас не так уж и много данных, поэтому все эти файлы вместе занимают всего 925 байт. Для того, чтобы продемонстрировать одну интересную особенность Git, добавим файл побольше. Добавим файл `repo.rb` из библиотеки Grit — он занимает примерно 22 Кб:

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb > repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 709 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

Если мы посмотрим на полученное дерево, мы увидим значение SHA-1 блоба для файла `геро.rb`:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 геро.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b test.txt
```

Посмотрим, сколько этот объект занимает места на диске, используя `git cat-file`:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Теперь немного изменим этот файл и посмотрим на результат:

```
$ echo '# testing' >> геро.rb
$ git commit -am 'modified геро а bit'
[master 2431da6] modified геро.rb a bit
 1 file changed, 1 insertion(+)
```

Взглянув на дерево, полученное в результате коммита, мы увидим любопытную вещь:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e геро.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b test.txt
```

Теперь файлу `геро.rb` соответствует совершенно другой блоб. Это означает, что всего одна единственная строка, добавленная в конец 400-строчного файла, требует создания абсолютно нового объекта:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

Итак, мы имеем два почти одинаковых объекта занимающих по 22 Кб на диске. Было бы здорово, если бы Git сохранял только один объект целиком, а другой как разницу между ним и первым объектом.

Оказывается, Git так и делает. Первоначальный формат для сохранения объектов в Git называется “рыхлым” форматом (loose format). Однако, время от времени Git упаковывает несколько таких объектов в один pack-файл (pack в пер. с англ. — упаковывать, уплотнять) для сохранения места на диске и повышения эффективности. Это происходит, когда “рыхлых” объектов становится слишком много, а также при вызове `git gc` вручную, и при отправке изменений на удалённый сервер. Чтобы посмотреть, как происходит упаковка, можно выполнить команду `git gc`:

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

Если вы загляните в директорию с объектами, вы обнаружите, что большая часть объектов исчезла, зато появились два новых файла:

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

Оставшиеся объекты — это блобы, на которые не указывает ни один коммит. В нашем случае это созданные ранее объекты: содержащий строку “what is up, doc?”, и “test content”. В силу того, что ни в одном коммите данные файлы не присутствуют, они считаются “висячими” и не упаковываются.

Остальные файлы — это pack-файл и его индекс. Pack-файл — это файл, который теперь содержит все удалённые объекты. Индекс — это файл, в котором записаны смещения прежних объектов в pack-файле для быстрого поиска. Упаковка данных положительно повлияла на общий размер файлов: если до вызова `gc` они занимали примерно 22 Кб, то pack-файл занимает всего 7 Кбайт. Мы только что освободили  $\frac{2}{3}$  занимаемого дискового пространства!

Как Git это делает? При упаковке Git ищет похожие по имени и размеру файлы и сохраняет только разницу между соседними версиями. Можно заглянуть в pack-файл чтобы понять, какие действия

выполняются при сжатии. Для просмотра содержимого упакованного файла существует служебная команда `git verify-pack`:

```
$ git verify-pack -v .git/objects/pack/pack-978e03944f5c581011e6998cd0e9e300009055
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
d368d0ac0678cbe6cce505be58126d3526706e54 tag 130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree 136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree 36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree 136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree 6 17 1314 1 \
 deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree 8 19 1331 1 \
 deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree 71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob 10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob 9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob 22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob 9 20 7262 1 \
 b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob 10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

Здесь блоб 033b4, который, как мы помним, был первой версией файла `hero.gb`, ссылается на блоб b042a, который хранит вторую его версию. Третья колонка в выводе — это размер содержимого объекта. Как видите, b042a занимает 22 Кб, а 033b4 — всего 9 байт. Что интересно, вторая версия файла сохраняется “как есть”, а первая — в виде дельты: ведь скорее всего вам понадобится быстрый доступ к самым последним версиям файла.

Также здорово, что переупаковку можно выполнять в любое время. Время от времени Git будет выполнять её автоматически, чтобы сэкономить место на диске, но всегда можно инициировать упаковку, выполнив `git gc`.

## Спецификации ссылок

На протяжении всей книги мы использовали довольно простые соответствия между локальными ветками и ветками в удалённых репозиториях, но всё может быть чуть сложнее. Допустим, вы добавили удалённый репозиторий:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

Эта команда добавляет секцию в файл `.git/config`, в которой заданы имя удалённого репозитория (`origin`), его URL и спецификация ссылок для извлечения данных:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit
fetch = +refs/heads/*:refs/remotes/origin/*
```

Формат спецификации следующий: опциональный `+`, далее пара `<src>:<dst>`, где `<src>` — шаблон ссылок в удалённом репозитории, а `<dst>` — соответствующий шаблон локальных ссылок. Символ `+` сообщает Git, что обновление необходимо выполнять даже в том случае, если оно не является перемоткой (fast-forward).

По умолчанию, после выполнения `git remote add`, Git забирает все ссылки из `refs/heads/` на сервере, и записывает их в `refs/remotes/origin/` локально. Таким образом, если на сервере есть ветка `master`, журнал данной ветки можно получить, вызвав:

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Все эти команды эквивалентны, так как Git развернёт каждую запись до `refs/remotes/origin/master`.

Если хочется, чтобы Git забирал при обновлении только ветку `master`, а не все доступные на сервере, можно изменить соответствующую строку в конфигурации:

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Эта настройка будет использоваться по умолчанию при вызове `git fetch` для данного удалённого репозитория. Если же вам нужно изменить спецификацию всего раз, можно задать `refspec` в командной строке. Например, чтобы получить данные из ветки `master` из удалённого репозитория в локальную `origin/mymaster`, можно выполнить:

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Можно задать несколько и спецификаций за один раз. Получить данные нескольких веток из командной строки можно так:

```
$ git fetch origin master:refs/remotes/origin/mymaster \
 topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
! [rejected] master -> origin/mymaster (non fast forward)
* [new branch] topic -> origin/topic
```

В данном случае слияние ветки `master` выполнить не удалось, поскольку слияние не было “быстрой перемоткой”. Такое поведение можно изменить, добавив перед спецификацией знак `+`.

В конфигурационном файле также можно задавать несколько спецификаций для получения обновлений. Чтобы каждый раз получать обновления веток `master` и `experiment`, добавьте две такие строки:

```
[remote "origin"]
 url = https://github.com/schacon/simplegit-progit
 fetch = +refs/heads/master:refs/remotes/origin/master
 fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Задавать маску частично в спецификации нельзя, следующая запись неверна:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Тем не менее, можно использовать пространства имён для получения схожего результата. Если ваша QA команда использует несколько веток для своей работы, и вы хотите получать только ветку `master` и все ветки команды QA, то можно добавить в конфигурацию следующее:

```
[remote "origin"]
 url = https://github.com/schacon/simplegit-progit
 fetch = +refs/heads/master:refs/remotes/origin/master
 fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

Если у вас сложный рабочий процесс при котором все команды — разработчики, QA и специалисты по внедрению — ведут работы в одном репозитории, вы можете разграничить их с помощью пространств имён.

## Спецификации ссылок для отправки данных на сервер

Здорово, что можно получать данные по ссылкам в отдельных пространствах имён, но нам же ещё надо сделать так, чтобы команда QA сначала смогла отправить свои ветки в пространство имён qa/. Мы решим эту задачу, используя спецификации ссылок для команды push.

Если команда QA хочет отправлять изменения из локальной ветки master в qa/master на удалённом сервере, они могут использовать такой приём:

```
$ git push origin master:refs/heads/qa/master
```

Если же они хотят, чтобы Git автоматически делал так при вызове `git push origin`, можно добавить в конфигурационный файл значение для `push`:

```
[remote "origin"]
 url = https://github.com/schacon/simplegit-progit
 fetch = +refs/heads/*:refs/remotes/origin/*
 push = refs/heads/master:refs/heads/qa/master
```

Опять же, это приведёт к тому, что при вызове `git push origin` локальная ветка `master` будет по умолчанию отправляться в удалённую ветку `qa/master`.

## Удаление ссылок

Кроме того, спецификации ссылок можно использовать для удаления ссылок на удалённом сервере:

```
$ git push origin :topic
```

Так как спецификация ссылки задаётся в виде `<src>:<dst>`, то, пропуская `<src>`, мы указываем Git, что указанную ветку на удалённом сервере надо сделать пустой, что приводит к её удалению.

## Протоколы передачи данных

Git умеет передавать данные между репозиториями двумя способами: используя “глупый” и “умный” протоколы. В этой главе мы рассмотрим, как они работают.

### Глупый протокол

Если вы разрешили доступ на чтение к вашему репозиторию через HTTP, то скорее всего будет использован “глупый” протокол. Протокол назвали глупым, потому что для его работы не требуется выполнение специфичных для Git операций на стороне сервера: весь процесс получения данных представляет собой серию HTTP GET запросов. При этом клиент ожидает обнаружить определённые файлы на сервере по заданным путём.

---

Глупый протокол довольно редко используется в наши дни. При использовании глупого протокола сложно обеспечить безопасность передачи и приватность данных, поэтому большинство Git серверов (как облачных, так и тех, что требуют установки) откажутся работать через него. Рекомендуется использовать умный протокол, который мы рассмотрим далее.

---

Давайте рассмотрим процесс получения данных из репозитория `simplegit-progit`:

```
$ git clone http://server/simplegit-progit.git
```

Первым делом будет загружен файл `info/refs`. Данный файл записывается командой `update-server-info`, поэтому для корректной работы HTTP-транспорта необходимо выполнять её в `post-receive` триггере.

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949 refs/heads/master
```

Теперь у нас имеется список удалённых веток и их хеши. Далее, надо посмотреть, куда ссылается HEAD, чтобы знать на что переключиться после завершения работы команды.

```
=> GET HEAD
ref: refs/heads/master
```

Итак, нужно переключится на ветку master после окончания работы. На данном этапе можно начинать обход репозитория. Начальной точкой является коммит ca82a6, о чём мы узнали из файла info/refs, и мы начинаем с его загрузки:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

Объект получен, он был в рыхлом формате на сервере, и мы получили его по HTTP, используя GET-запрос. Теперь можно его разархивировать, обрезать заголовок и посмотреть на содержимое:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Далее, необходимо загрузить ещё два объекта: дерево cfda3b — содержимое только что загруженного коммита, и 085bb3 — родительский коммит:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

Вот мы и получили следующий объект-коммит. Теперь содержимое коммита:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Упс, похоже, этого дерева нет на сервере в рыхлом формате, поэтому мы получили ответ 404. Возможны два варианта: объект в другом репозитории, или в упакованном файле текущего репозитория. Сначала Git проверяет список альтернативных репозиториев:

```
=> GET objects/info/http-alternates
(empty file)
```

Если бы этот запрос вернул непустой список альтернатив, Git проверил бы указанные репозитории на наличие файла в “рыхлом” формате – довольно полезная фишка для проектов-форков, позволяющая устраниТЬ дублирование. Так как в данном случае альтернатив нет, объект, должно быть, упакован в pack-файл. Чтобы посмотреть доступные на сервере pack-файлы, нужно скачать файл `objects/info/packs`, содержащий их список. Этот файл тоже обновляется командой `update-server-info`:

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

На сервере имеется только один pack-файл, поэтому объект точно там, но необходимо проверить индексный файл, чтобы в этом убедиться. Если бы на сервере было несколько pack-файлов, загрузив сначала индексы, мы смогли бы определить, в каком именно pack-файле находится нужный нам объект:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Теперь, когда мы получили индекс pack-файла, можно проверить, содержится ли в нём наш объект. Это возможно благодаря тому, что в индексе хранятся SHA-1 объектов, содержащихся внутри pack-файла, а также их смещения. Наш объект там присутствует, так что продолжим и скачаем весь pack-файл:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

Итак, мы получили наше дерево, можно продолжить обход списка коммитов. Все они содержатся внутри свежескаченного pack-файла, так что снова обращаться к серверу не надо. Git извлекает рабочую копию ветки `master`, на которую ссылается `HEAD` (не забыли, для чего мы скачивали файл `info/refs` в самом начале?).

## Умный протокол

Глупый протокол прост, но неэффективен и не позволяет производить запись в удалённые репозитории. Гораздо чаще для обмена данными используют “умный” протокол. Но это требует запуска на сервере специального процесса, знающего о структуре Git репозитория, умеющего выяснить, какие данные необходимо отправить клиенту и генерирующего отдельный pack-файл с недостающими изменениями для него. Работу умного протокола обеспечивают несколько процессов: два для отправки данных на сервер и два для загрузки с него.

### ЗАГРУЗКА ДАННЫХ НА СЕРВЕР

Для загрузки данных на удалённый сервер используются процессы `send-pack` и `receive-pack`. Процесс `send-pack` запускается на клиенте и подключается к `receive-pack` на сервере.

#### SSH

Допустим, вы выполняете `git push origin master` и `origin` задан как URL, использующий протокол SSH. Git запускает процесс `send-pack`, который устанавливает соединение с сервером по протоколу SSH. Он пытается запустить команду на удалённом сервере через вызов `ssh` команды, который выглядит следующим образом:

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status \
 delete-refs side-band-64k quiet ofs-delta \
 agent=git/2:2.1.1+github-607-gfba4028 delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

Команда `git-receive-pack` тут же посыпает в ответ по одной строке на каждую из имеющихся в наличии ссылок — в данном случае только ветку `master` и её SHA-1. Первая строка также содержит список возможностей сервера (здесь это `report-status`, `delete-refs` и парочка других, включая версию используемого процесса).

Каждая строка начинается с 4-байтового шестнадцатеричного значения, содержащего длину оставшейся части строки. Первая строка начинается с `005b`, это `91` в десятичной системе счисления, значит в этой строке ещё `91` байт. Следующая строка начинается с `003e` (`62`), то есть надо прочитать ещё `62` байта. Далее следует `0000`, означающая конец списка ссылок.

Теперь, когда `send-pack` выяснил состояние сервера, он определяет коммиты, которые есть локально, но отсутствующие на сервере. Для каждой ссылки, которая будет обновлена командой `push`, процесс `send-pack` передаёт процессу `receive-pack` эти данные. Например, если мы обновляем ветку `master`, и добавляем ветку `experiment`, ответ `send-pack` будет выглядеть следующим образом:

Git посыпает по строке, содержащей собственную длину, старый хэш, новый хэш и имя ссылки; для каждой обновляемой ссылки. В первой строке также посыпаются возможности клиента. Хэш, состоящий из нулей, говорит о том, что раньше такой ссылки не было – вы ведь добавляете новую ветку `experiment`. При удалении ветки всё было бы наоборот: нули были бы справа.

Затем клиент посыпает pack-файл с объектами, которых нет на сервере. В конце сервер передаёт статус операции – успех или ошибка:

000Aunpack ok

## HTTP(S)

Этот процесс похож на HTTP, но установка соединения слегка отличается. Всё начинается с такого запроса:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack
001f# service=git-receive-pack
000000ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master \
 report-status delete-refs side-band-64k quiet ofs-delta \
 agent=git/2:2.1.1~vmg-bitmaps-bugaloo-608-g116744e
0000
```

Это всё, что передаётся в ответ на первый запрос. Затем клиент делает второй запрос, на этот раз POST, передавая данные, полученные от команды `git-upload-pack`.

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

Этот запрос включает в себя результаты send-pack и собственно pack-файлы. Сервер, используя код состояния HTTP, возвращает результат операции.

## СКАЧИВАНИЕ ДАННЫХ

Для получения данных из удалённых репозиториев используются процессы fetch-pack и upload-pack. Клиент запускает процесс fetch-pack, который подключается к процессу upload-pack на сервере для определения подлежащих передаче данных.

### SSH

Если вы работаете через SSH, fetch-pack выполняет примерно такую команду:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

Как только fetch-pack подключается к upload-pack, тот отсылает обратно следующее:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEADmulti_ack thin-pack \
 side-band side-band-64k ofs-delta shallow no-progress include-tag \
 multi_ack_detailed symref=HEAD:refs/heads/master \
 agent=git/2:2.1.1+github-607-gfba4028
003fc82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

Это очень похоже на ответ receive-pack, но только возможности другие. Вдобавок upload-pack отсылает обратно ссылку HEAD, чтобы клиент понимал, на какую ветку переключиться, если выполняется клонирование.

На данном этапе процесс fetch-pack смотрит на объекты, имеющиеся в наличии, и для недостающих объектов отвечает словом “want” + соответствующий SHA-1. Для уже имеющихся объектов процесс отправляет их хеши со словом “have”. В конце списка он пишет “done”, и это даёт понять процессу upload-pack, что пора начинать отправлять упакованный pack-файл с необходимыми данными:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bc608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

## HTTP(S)

“Рукопожатие” для процесса получения недостающих данных занимает два HTTP запроса. Первый — это GET запрос на тот же URL, что и в случае глупого протокола:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
000000e7ca82a6dff817ec66f44342007202690a93763949 HEADmulti_ack thin-pack \
 side-band side-band-64k ofs-delta shallow no-progress include-tag \
 multi_ack_detailed no-done symref=HEAD:refs/heads/master \
 agent=git/2:2.1.1+github-607-gfba4028
003fcfa82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

Это очень похоже на использование git-upload-pack по SSH, вот только обмен данными производится отдельным запросом:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fd93eb2908e52742248faf0ee993
0000
```

Используется тот же формат что и ранее: В ответ сервер посыпает статус операции и генерированный pack-файл.

## Заключение

В этом разделе мы вкратце рассмотрели протоколы передачи данных. Протоколы обмена данных в Git включают в себя множество фич — типа `multi_ack` или `side-band` — рассмотрение которых выходит за пределы этой книги. Мы описали формат сообщений между клиентом и сервером не вдаваясь в детали, если хотите покопаться в этой теме глубже — обратитесь к исходному коду Git.

## Уход за репозиторием и восстановление данных

Изредка вам потребуется делать “уборку” — сделать репозиторий более компактным, очистить импортированный репозиторий от лишних файлов или восстановить потерянные данные. Данный раздел охватывает некоторые из этих сценариев.

## Уход за репозиторием

Время от времени Git выполняет автоматическую сборку мусора. Чаще всего эта команда ничего не делает. Однако, если у вас накопилось слишком много “рыхлых” объектов (не в pack-файлах), или, наоборот, отдельных pack-файлов, Git запускает полноценный сборщик — `git gc`. Здесь “`gc`” это сокращение от “garbage collect”, что означает “сборка мусора”. Эта команда выполняет несколько действий: собирает все рыхлые объекты и упаковывает их в pack-файлы; объединяет несколько упакованных файлов в один большой; удаляет объекты, недостижимые ни из одной фиксации и хранящиеся дольше нескольких месяцев.

Можно запустить сборку мусора вручную:

```
$ git gc --auto
```

Опять же, как правило, эта команда ничего не делает. Нужно иметь примерно 7000 несжатых объектов или более 50 pack-файлов, чтобы запустился настоящий `gc`. Эти значения можно изменить с помощью параметров `gc.auto` и `gc.autopacklimit` соответственно.

Ещё одно действие, выполняемое `gc` — упаковка ссылок в единый файл. Предположим, репозиторий содержит следующие ветки и теги:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

Если выполнить `git gc`, эти файлы будут удалены из директории `refs`. Git перенесёт их в файл `.git/packed-refs` в угоду эффективности:

```
$ cat .git/packed-refs
pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

При обновлении ссылки Git не будет редактировать этот файл, а добавит новый файл в `refs/heads`. Для получения хеша, соответствующего нужной ссылке, Git сначала проверит наличие файла ссылки в директории `refs`, а к файлу `packed-refs` обратится только в случае отсутствия оного. Так что, если вы не можете найти ссылку в директории `refs`, скорее всего она упакована в файле `packed-refs`.

Обратите внимание, последняя строка файла начинается с `^`. Это означает, что метка на предыдущей строке является аннотированной меткой и данная строка — это фиксация, на которую аннотированная метка указывает.

## Восстановление данных

В какой-то момент при работе с Git вы можете нечаянно потерять фиксацию. Как правило, такое случается, когда вы удаляете ветку, в которой находились некоторые наработки, а потом оказывается, что они всё-таки были нужными; либо вы выполнили `git reset --hard`, тем самым отказавшись от фиксаций, которые затем понадобились. Как же в таком случае заполучить свои фиксации обратно?

Ниже приведён пример, в котором мы сбрасываем ветку `master` с потерей данных до более раннего состояния, а затем восстанавливаем потерянные фиксации. Для начала, давайте посмотрим, как сейчас выглядит история изменений:

```
$ git log --pretty=oneline
ab1afe80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbdde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Теперь сбросим ветку `master` на третью фиксацию:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbdde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Итак, теперь две последних фиксации по-настоящему потеряны — они не достижимы ни из одной ветки. Необходимо найти SHA-1 последней фиксации и создать ветку, указывающую на неё. Сложность в том, чтобы узнать этот самый SHA-1, ведь вряд ли вы его запомнили, да?

Зачастую самый быстрый способ — использование команды `git reflog`. Дело в том, что во время вашей работы Git записывает все изменения HEAD. Каждый раз при переключении веток и фиксации изменений, добавляется запись в `reflog`. `reflog` также обновляется командой `git update-ref` — это, кстати, хорошая причина использовать именно эту команду, а не вручную записывать SHA-1 в `ref`-файлы, как было показано в “Ссылки в Git”. Вы можете посмотреть где находился указатель HEAD в любой момент времени, запустив `git reflog`:

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: modified repo.rb a bit
484a592 HEAD@{2}: commit: added repo.rb
```

Здесь мы видим две фиксации, на которые когда-то указывал HEAD, однако информации не так уж и много. Более интересные выводы можно получить, используя `git log -g`, что выведет привычный лог, но для записей из `reflog`:

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:22:37 2009 -0700

 third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:15:24 2009 -0700

 modified repo.rb a bit
```

Похоже, что последняя фиксация — это и есть та, которую мы потеряли; и её можно восстановить, создав ветку, указывающую на неё. Например, создадим ветку с именем `recover-branch`, указывающую на эту фиксацию (`ab1afef`):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified hero a bit
484a59275031909e19aadb7c92262719cfcdf19a added hero.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Здорово! Теперь у нас есть ветка `recover-branch`, указывающая туда, куда ранее указывал `master`, тем самым делая потерянные фиксации вновь доступными. Теперь, положим, потерянная ветка по какой-либо причине не попала в `reflog`, для этого удалим восстановленную ветку и весь `reflog`. Теперь две первых фиксации недоступны ниоткуда:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Данные `reflog` хранятся в директории `.git/logs/`, которую мы только что удалили, поэтому теперь у нас нет `reflog`. Как теперь восстановить фиксации? Один из вариантов — использование утилиты `git fsck`, проверяющую внутреннюю базу данных на целостность. Если выполнить её с ключом `--full`, будут показаны все объекты, недостижимые из других объектов:

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

В данном случае потерянная фиксация указана после слов “`dangling commit`” (“висячий коммит”). Его можно восстановить аналогичным образом, добавив ветку, указывающую на этот SHA-1.

## Removing Objects

Git — замечательный инструмент с кучей классных фич, но некоторые из них способны и навредить. Например, команда `git clone` загружает проект вместе со всей историей, включая все версии всех файлов. Это нормально, если в репозитории хранится только исходный код, так как Git хорошо оптимизирован под такой тип данных и может эффективно сжимать их. Однако, если когда-либо в проект был добавлен большой файл, каждый, кто потом захочет клонировать проект, будет вынужден скачивать этот файл, даже если он был удалён в следующей же фиксации. Он будет в базе всегда, просто потому, что он доступен в истории.

Это может стать большой проблемой при конвертации Subversion или Perforce репозиториев в Git. В этих системах вам не нужно загружать всю историю, поэтому добавление больших файлов не имеет там особых последствий. Если при импорте из другой системы или при каких-либо других обстоятельствах стало ясно, что ваш репозиторий намного больше, чем он должен быть, то как раз сейчас мы расскажем, как можно найти и удалить большие объекты.

**Предупреждаем: дальнейшие действия разрушат историю изменений.** Каждая фиксация, начиная с самой ранней, из которой нужно удалить большой файл, будет переписана. Если сделать это непосредственно после импорта, пока никто ещё не работал с репозиторием, то всё окей, иначе придётся сообщать всем участникам о необходимости перемещения их правок на новые фиксации.

Для примера добавим большой файл в тестовый репозиторий, удалим его в следующей фиксации, а потом найдём и удалим его полностью из базы. Для начала добавим большой файл в нашу историю:

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz > git.tgz
$ git add git.tgz
$ git commit -m 'add git tarball'
[master 7b30847] add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

Упс, мы нечаянно. Нам лучше избавится от этого файла:

```
$ git rm git.tgz
rm 'git.tgz'
```

```
$ git commit -m 'oops - removed large tarball'
[master dadf725] oops - removed large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Теперь запустим сборщик мусора и посмотрим, сколько места мы занимаем:

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

Чтобы быстро узнать, сколько места занято, можно воспользоваться командой `count-objects`:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

Строка `size-pack` — это размер pack-файлов в килобайтах, то есть всего занято почти 5 МБ. Перед последней фиксаций использовалось около 2 КБ — очевидно, удаление файла не удалило его из истории. Всякий раз, когда кто-либо захочет склонировать этот репозиторий, ему придётся скачивать все 5 МБ для того, чтобы заполучить этот крошечный проектик, просто потому, что однажды вы имели неосторожность добавить большой блоб! Давайте же исправим это!

Для начала найдём проблемный файл. В данном случае, мы заранее знали, что это за файл. Но если бы не знали, как можно было бы определить, какие файлы занимают много места? При вызове `git gc` все объекты упаковываются в один pack-файл, но, несмотря на это, определить самые крупные файлы можно, запустив служебную команду `git verify-pack`, и отсортировав её вывод по третьей колонке, в которой записан размер файла. Так как нас интересуют

самые крупный файлы, оставим три последние строки с помощью `tail`:

```
$ git verify-pack -v .git/objects/pack/pack-29..69.idx \
| sort -k 3 -n \
| tail -3
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob 22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob 4975916 4976258 1438
```

Большой объект в самом внизу, его размер — 5 МБ. Для того чтобы узнать, что это за файл, воспользуемся командой `rev-list`, которая уже упоминалась в главе “**Enforcing a Specific Commit-Message Format**”. Если передать ей ключ `--objects`, она выдаст хеши всех фиксаций, а также хеши объектов и соответствующие им имена файлов. Воспользуемся этим для определения имени выбранного объекта:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Теперь необходимо удалить данный файл из всех деревьев в прошлом. Легко получить все фиксации, которые меняли данный файл:

```
$ git log --oneline --branches -- git.tgz
dadf725 oops - removed large tarball
7b30847 add git tarball
```

Необходимо переписать все фиксации, начиная с `7b30847` для полного удаления этого файла из истории. Воспользуемся командой `filter-branch`, о которой мы писали в “Исправление истории”:

```
$ git filter-branch --index-filter \
'git rm --ignore-unmatch --cached git.tgz' -- 7b30847^..
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm 'git.tgz'
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)
Ref 'refs/heads/master' was rewritten
```

Опция `--index-filter` похожа на `--tree-filter`, использовавшуюся в главе “Исправление истории”, за исключением того, что вместо передачи команды, модифицирующей файлы на диске, мы используем команду, изменяющую файлы в индексе.

Вместо удаления файла чем-то вроде `rm file`, мы используем `git rm --cached`, так как нам надо удалить файл из индекса, а не с диска. Причина, по которой мы делаем именно так — скорость: нет необходимости извлекать каждую ревизию на диск, чтобы применить фильтр, а это может очень сильно ускорить процесс. Если хотите, можете использовать и `tree-filter` для получения аналогичного результата. Опция `--ignore-unmatch` команды `git rm` отключает вывод сообщения об ошибке в случае отсутствия файлов, соответствующих шаблону. Ещё один момент: мы указали команде `filter-branch` переписывать историю, начиная с фиксации 7b30847, потому что мы знаем, что именно в этом изменении впервые появилась проблема. По умолчанию перезапись начинается с самого первого состояния, что потребовало бы гораздо больше времени.

Теперь история не содержит ссылок на данный файл. Однако, в `reflog` и в новом наборе ссылок, добавленном Git в `.git/refs/original` после выполнения `filter-branch`, ссылки на него всё ещё присутствуют, поэтому необходимо их удалить, а потом переупаковать базу. Необходимо избавиться от всех возможных ссылок на старые фиксации перед переупаковкой:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (15/15), done.
Total 15 (delta 1), reused 12 (delta 0)
```

Проверим, сколько места удалось освободить:

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
```

```
garbage: 0
size-garbage: 0
```

Размер упакованного репозитория сократился до 8 КБ, что намного лучше, чем 5 МБ. Из значения поля size видно, что большой объект всё ещё хранится в одном из ваших “рыхлых” объектов, но, что самое главное, при любой последующей отправке данных наружу (а значит и при последующих клонированиях репозитория) он передаваться не будет. Если очень хочется, можно удалить его навсегда локально, выполнив `git prune --expire`:

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

## Переменные среды

Git всегда запущен в bash и использует некоторые переменные shell среды, чтобы определить, как она себя ведет. Порой удобно знать, какие именно константы используются, чтобы Git работал именно так, как вы хотите. Это не исчерпывающий список переменных среды, которые использует Git, но мы рассмотрим самые полезные.

## Глобальное поведение

Поведение Git как компьютерной программы зависит от параметров среды.

`GIT_EXEC_PATH` определяет где Git будет искать свои подпрограммы. Текущие настройки можно узнать командой `git --exec-path`.

`HOME` обычно не рассматривается в качестве изменяемого параметра (чесчур много вещей от него зависят), но именно тут Git ищет глобальный файл конфигурации. Если вам нужна по-настоящему portable-версия Git с собственной глобальной конфигурацией, можете переопределить `HOME` в shell профиле.

**PREFIX** аналогичная константа, но для общесистемной конфигурации. Git ищет этот файл в `$PREFIX/etc/gitconfig`.

**GIT\_CONFIG\_NOSYSTEM**, если задана, отключает использование файла общесистемной конфигурации. Это пригодится, если ваша системная конфигурация мешает вашим командам, а прав на её редактирование или удаление у вас нет.

**GIT\_PAGER** определяет программу, используемую для отображения многостраничного вывода в командной строке. Если не задана, в качестве запасного варианта используется PAGER.

**GIT\_EDITOR** это редактор, который Git запустит, когда пользователю понадобится отредактировать какой-нибудь текст (например, сообщение коммита). Если не задана, откроется EDITOR.

## Расположение репозитория

Git использует некоторые переменные среды, чтобы определить как она взаимодействует с конкретным репозиторием.

**GIT\_DIR** — это месторасположение директории `.git`. Если эта переменная не задана, Git будетходить вверх по дереву директорий, пока не достигнет `~` (домашней директории пользователя) или `/` (корневой директории), проверяя на каждом шагу наличие директории `.git`.

**GIT\_CEILING\_DIRECTORIES** управляет процессом поиска директории `.git`. Если вы работаете с медленной файловой системой (типа ленточного накопителя или сетевой папки), вы можете запретить Git доступ к `.git` без надобности, например, для построения строки приветствия.

**GIT\_WORK\_TREE** — это путь к рабочей директории для не-серверного репозитория (с непустой рабочей директорией). Если эта переменная не задана, будет использована родительская директория `$GIT_DIR`.

**GIT\_INDEX\_FILE** — это путь к файлу индекса (только для репозиториев с непустой рабочей директорией).

**GIT\_OBJECT\_DIRECTORY** может быть использована для указания директории с объектами вместо `.git/objects`.

**GIT\_ALTERNATE\_OBJECT\_DIRECTORIES** — это список разделённых двоеточием директорий (типа `/dir/one:/dir/two:...`), в которых Git будет пытаться найти объекты, которых нет в `GIT_OBJECT_DIRECTORY`. Это может быть полезным, если вы работаете над несколькими проектами с одинаковым содержимым, чтобы не дублировать файлы.

## Пути к файлам

Эти переменные влияют на то, как Git будет понимать пути к файлам и шаблоны путей. Эти настройки применяются к записям в файлах `.gitignore` и к путям, переданным в командной строке (`git add *.c`).

`GIT_GLOB_PATHSPECS` и `GIT_NOGLOB_PATHSPECS` управляют поведением шаблонов путей к файлам. Если переменная `GIT_GLOB_PATHSPECS` установлена в 1, то специальные символы интерпретируются стандартным для шаблонов способом; если же `GIT_NOGLOB_PATHSPECS` установлена в 1, то специальные символы обрабатываются буквально, это означает, что, например, запись `*.c` будет обозначать лишь единственный файл с именем `"*.c"`, а не все файлы с расширением `".c"`. Это поведение можно переопределить в каждом конкретном случае, приписывая к путям строки `:(glob)` или `:(literal)`, например `:(glob)*.c`.

`GIT_LITERAL_PATHSPECS` отключает шаблоны в путях: ни специальные символы, ни специальные префиксы не будут работать.

`GIT_ICASE_PATHSPECS` делает все пути регистронезависимыми.

## Фиксация изменений

Окончательное создание объекта-коммита обычно производится командой `git-commit-tree`, которая использует приведённые ниже переменные окружения в качестве источника информации. И лишь в случае, если эти переменные не заданы, она будет использовать данные из файлов конфигурации.

`GIT_AUTHOR_NAME` используется для указания автора коммита.

`GIT_AUTHOR_EMAIL` задаёт адрес электронной почты автора коммита.

`GIT_AUTHOR_DATE` время создания коммита.

`GIT_COMMITTER_NAME` используется для указания человека, применившего коммит.

`GIT_COMMITTER_EMAIL` задаёт адрес электронной почты человека, применившего коммит.

`GIT_COMMITTER_DATE` время применения коммита.

`EMAIL` используется, как запасное значение, если конфигурационный параметр `user.email` не задан. Если же и эта переменная не задана, Git будет использовать идентификатор пользователя в системе и имя хоста.

## Работа с сетью

Git использует библиотеку `curl` для работы с сетью через HTTP. Задание переменной `GIT_CURL_VERBOSE` указывает Git'у выводить все сообщения, генерируемые этой библиотекой. Это похоже на использование `curl` с флагом `-v` в командной строке.

`GIT_SSL_NO_VERIFY` отключает проверку SSL сертификатов. Это может пригодиться если вы используете самоподписанные сертификаты для работы репозиториев через HTTPS, или если вы настраиваете Git сервер и ещё не установили необходимые сертификаты.

Если на протяжении более чем `GIT_HTTP_LOW_SPEED_TIME` секунд скорость передачи данных не поднималась выше `GIT_HTTP_LOW_SPEED_LIMIT` байт в секунду, Git прервёт операцию. Эти переменные замещают значения конфигурационных параметров `http.lowSpeedLimit` и `http.lowSpeedTime`.

`GIT_HTTP_USER_AGENT` задаёт заголовок User-Agent при работе через HTTP. По умолчанию используется что-то вроде `git/2.0.0`.

## Сравнение файлов и слияния

`GIT_DIFF_OPTS` — слегка громкое название для этой переменной. Единственными допустимыми значениями являются `-u<n>` и `--unified=<n>`, задающие количество контекстных строк, показываемых командой `git diff`.

`GIT_EXTERNAL_DIFF` замещает конфигурационный параметр `diff.external`. Если значение задано, Git вызовет указанную программу вместо `git diff`.

`GIT_DIFF_PATH_COUNTER` и `GIT_DIFF_PATH_TOTAL` используются внутри программы, заданной через `GIT_EXTERNAL_DIFF` или `diff.external`. Первая содержит порядковый номер сравниваемого на данный момент файла (начиная с 1), вторая — полное количество файлов, подлежащих сравнению.

`GIT_MERGE_VERBOSITY` задаёт уровень детализированности вывода при рекурсивном слиянии. Возможные значения перечислены ниже:

- 0 не выводить ничего, кроме единственного сообщения об ошибке..
- 1 выводить только конфликты.
- 2 также выводить изменения файлов.

- 3 показывать пропущенные неизменённые файлы.
- 4 выводить все пути при обработке.
- 5 и выше выводят даже отладочную информацию.

По умолчанию значение полагается равным 2.

## Отладка

Хотите знать что *на самом деле* делает Git? Git ведёт достаточно подробный лог выполняемых действий и всё что вам нужно — включить его. Возможные значения приведённых ниже переменных следующие:

- “true”, “1”, или “2” – вывод осуществляется в стандартный поток ошибок (stderr).
- Абсолютный путь, начинающийся с / – вывод будет производиться в указанный файл.

**GIT\_TRACE** задаёт журналирование действий, не подпадающий под какую-либо определённую категорию. Это включает в себя разворачивание алиасов и вызовы внешних программ.

```
$ GIT_TRACE=true git lga
20:12:49.877982 git.c:554 trace: exec: 'git-lga'
20:12:49.878369 run-command.c:341 trace: run_command: 'git-lga'
20:12:49.879529 git.c:282 trace: alias expansion: lga => 'log' '--graph' '--pr...
20:12:49.879885 git.c:349 trace: built-in: git 'log' '--graph' '--pretty=oneli...
20:12:49.899217 run-command.c:341 trace: run_command: 'less'
20:12:49.899675 run-command.c:192 trace: exec: 'less'
```

**GIT\_TRACE\_PACK\_ACCESS** задаёт журналирование обращений к pack-файлам. При этом первое выводимое значение – файл, к которому происходит обращение, а второе значение – смещение внутри этого файла.

```
$ GIT_TRACE_PACK_ACCESS=true git status
20:10:12.081397 sha1_file.c:2088 .git/objects/pack/pack-c3fa...291e.pack 12
20:10:12.081886 sha1_file.c:2088 .git/objects/pack/pack-c3fa...291e.pack 34662
20:10:12.082115 sha1_file.c:2088 .git/objects/pack/pack-c3fa...291e.pack 35175
[...]
20:10:12.087398 sha1_file.c:2088 .git/objects/pack/pack-e80e...e3d2.pack 56914983
20:10:12.087419 sha1_file.c:2088 .git/objects/pack/pack-e80e...e3d2.pack 14303666
On branch master
```

```
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

**GIT\_TRACE\_PACKET** задаёт журналирование пакетов при операциях с сетью.

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46 packet: git< # service=git-upload
20:15:14.867071 pkt-line.c:46 packet: git< 0000
20:15:14.867079 pkt-line.c:46 packet: git< 97b8860c071898d9e162
20:15:14.867088 pkt-line.c:46 packet: git< 0f20ae29889d61f2e93a
20:15:14.867094 pkt-line.c:46 packet: git< 36dc827bc9d17f80ed4f
[...]
```

**GIT\_TRACE\_PERFORMANCE** задаёт журналирование данных о производительности. Вывод показывает, как долго выполнялись те или иные действия.

```
$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414 performance: 0.374835000 s: git command: 'gc'
20:18:19.845585 trace.c:414 performance: 0.343020000 s: git command: 'gc'
Counting objects: 170994, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (43413/43413), done.
Writing objects: 100% (170994/170994), done.
Total 170994 (delta 126176), reused 170524 (delta 125706)
20:18:23.567927 trace.c:414 performance: 3.715349000 s: git command: 'gc'
20:18:23.584728 trace.c:414 performance: 0.000910000 s: git command: 'gc'
20:18:23.605218 trace.c:414 performance: 0.017972000 s: git command: 'gc'
20:18:23.606342 trace.c:414 performance: 3.756312000 s: git command: 'gc'
Checking connectivity: 170994, done.
20:18:25.225424 trace.c:414 performance: 1.616423000 s: git command: 'gc'
20:18:25.232403 trace.c:414 performance: 0.001051000 s: git command: 'gc'
20:18:25.233159 trace.c:414 performance: 6.112217000 s: git command: 'gc'
```

**GIT\_TRACE\_SETUP** задаёт журналирование информации о репозитории и окружении, в котором выполняется сам Git.

```
$ GIT_TRACE_SETUP=true git status
20:19:47.086765 trace.c:315 setup: git_dir: .git
20:19:47.087184 trace.c:316 setup: worktree: /Users/ben/src/git
20:19:47.087191 trace.c:317 setup: cwd: /Users/ben/src/git
20:19:47.087194 trace.c:318 setup: prefix: (null)
On branch master
```

```
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

## Разное

**GIT\_SSH** – указанная программа (если значение задано) будет использоваться вместо ssh когда Git работает по SSH протоколу. Порядок вызова этой программы будет таков: \$GIT\_SSH [имя пользователя@]хост [-р <порт>] <команда>. На самом деле, это не самый простой способ настроить поведение ssh: дополнительные параметры командной строки не поддерживаются, и вам, скорее всего, придётся писать скрипт-обёртку и указать GIT\_SSH на него. Возможно, проще будет использовать ~/.ssh/config.

**GIT\_ASKPASS** заменяет значение конфигурационного параметра core.askpass. Это программа вызывается Git'ом каждый раз, когда требуется запросить у пользователя пароль. Стока с текстом запроса передаётся этой программе в качестве аргумента командной строки, а вывод значения она должна осуществлять в стандартный поток вывода (stdout). (Читайте подробнее в главе “Хранилище учетных данных”.)

**GIT\_NAMESPACE** управляет доступом к ссылкам внутри пространств имён аналогично параметру --namespace. Чаще всего эта переменная используется на стороне сервера когда вы хотите хранить несколько форков одного репозитория, разделяя лишь ссылки.

**GIT\_FLUSH** заставляет Git отключить буферизацию при записи в стандартный поток вывода (stdout). Git будет чаще сбрасывать данные на диск если значение выставлено в 1, если же оно равно 0, весь вывод будет буферизоваться. Значение, используемое по умолчанию (если ничего не задано) выбирается в зависимости от выполняемых действий и способа вывода данных.

**GIT\_REFLOG\_ACTION** задаёт подробное описание, записываемое в reflog. Например:

```
$ GIT_REFLOG_ACTION="my action" git commit --allow-empty -m 'my message'
[master 9e3d55a] my message
$ git reflog -1
9e3d55a HEAD@{0}: my action: my message
```

## Заключение

Теперь вы довольно хорошо понимаете, что Git делает за кулисами и, в некоторой степени, как он устроен. В данной главе мы рассмотрели несколько служебных команд — более низкоуровневых и простых, чем обычные пользовательские команды, описанные в остальной части книги.

Понимание принципов работы Git на низком уровне поможет вам понять работу Git в целом и даст возможность написать собственные утилиты и сценарии для организации специфического процесса работы с Git.

Git как контентно-адресуемая файловая система — очень мощный инструмент, который можно использовать как нечто большее, чем просто систему контроля версий. Надеемся, полученное знание внутренней реализации Git поможет вам написать своё крутое приложение, использующее эти технологии, и позволит вам чувствовать себя свободнее с Git даже в продвинутых вещах.



# Git в других окружениях

Если вы прочитали всю книгу, то много узнали об использовании Git в командной строке. Вы можете работать с локальными файлами, синхронизировать свой репозиторий с чужими по сети и эффективно работать с другими. Но это ещё не всё; Git обычно используется как часть большей экосистемы и терминал это не всегда лучший способ работы с ним. Рассмотрим несколько других окружений, где Git может быть полезен, и как другие приложения (включая ваши) работают с Git'ом.

## Графические интерфейсы

Родная среда обитания Git — это терминал. Новые фичи вначале доступны только там, и лишь терминал поможет вам полностью контролировать всю мощь Git. Но текстовый интерфейс — не лучший выбор для всех задач; иногда графическое представление более предпочтительно, а некоторые пользователи чувствуют себя комфортней, орудуя мышкой.

Также стоит понимать, что разные интерфейсы служат разным целям. Некоторые Git клиенты ограничиваются лишь тем функционалом, который их автор считает наиболее востребованным или эффективным. Учитывая это, ни один из представленных ниже инструментов не может быть “лучше” остальных: они просто заточены под разные задачи. Также стоит помнить, что всё, что можно сделать с помощью графического интерфейса, может быть выполнено и из консоли. Командная строка — по прежнему место где у вас больше всего мозги и контроля над репозиторием.

## gitk и git-gui

Установив Git, вы также получаете два графических инструмента: `gitk` и `git-gui`.

`gitk` — это графический просмотрщик истории. Что-то типа улучшенных `git log` и `git gper`. Это тот инструмент, который вы будете использовать для поиска событий и визуализации истории.

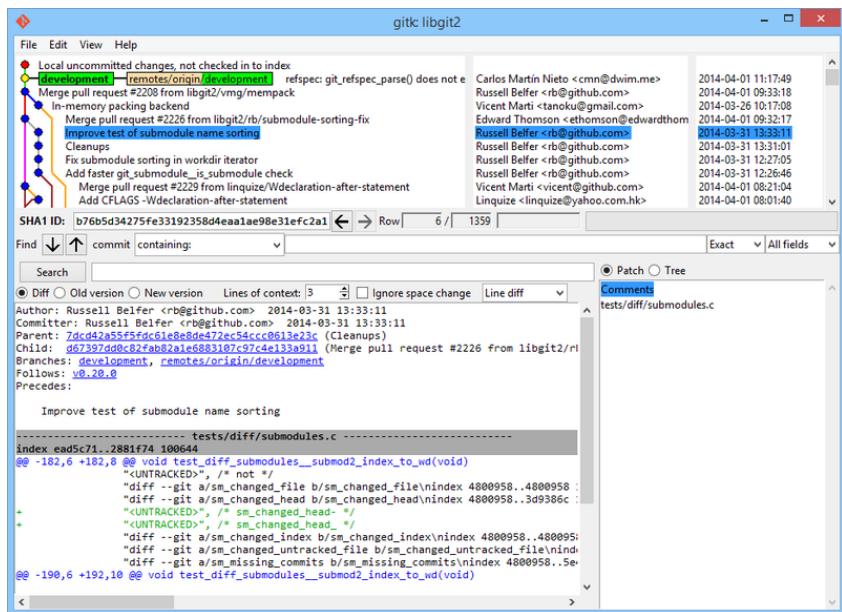
Проще всего вызвать `Gitk` из командной строки: Просто перейдите в директорию с репозиторием и наберите:

```
$ gitk [git log options]
```

`Gitk` принимает кучу различных опций, большинство из которых передаются в `git log` (который, в свою очередь, используется в `Gitk`). Возможно, наиболее используемая опция — `--all`, которая указывает `Gitk` выводить коммиты, доступные из *любой* ссылки, а не только `HEAD`. Интерфейс `Gitk` похож на этот:

Figure 1-1.

`gitk`- инструмент для просмотра истории.

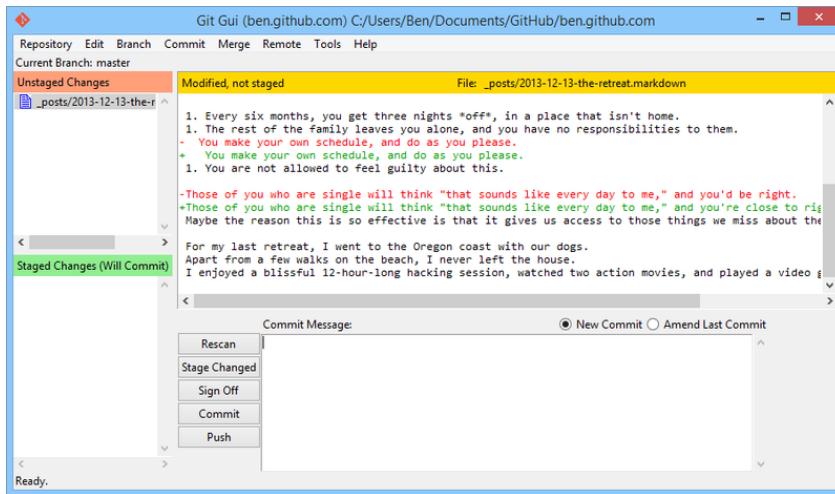


Интерфейс на картинке похож на вывод `git log --graph`: каждая точка соответствует коммиту, линии отражают родство коммитов, а ссылки изображены цветными прямоугольниками. Жёлтая точка обозначает HEAD, а красная — изменения, которые попадут в следующий коммит. Внизу экрана расположены элементы интерфейса для просмотра выделенного коммита: слева показаны изменения и комментарий, а справа — общая информация по изменённым файлам. В центре экрана расположены элементы для поиска по истории.

`git-gui`, в отличие от `gitk` — это инструмент редактирования отдельных коммитов. Его тоже очень просто вызвать из консоли:

```
$ git gui
```

И его интерфейс похож на этот:



**Figure 1-2.**

`git gui` —  
инструмент  
редактирования  
коммитов.

Слева находится область редактирования Git индекса: изменения в рабочей директории наверху, добавленные в индекс изменения — снизу. Вы можете перемещать файлы целиком между двумя состояниями, кликнув на иконки, или же вы можете просмотреть изменения в конкретном файле, выбрав его имя.

Справа вверху расположена область просмотра изменений выделенного файла. Можно добавлять отдельный кусочки или строки в индекс из контекстного меню в этой области.

Справа снизу находится область для ввода сообщения коммита и несколько кнопок. Введите сообщение и нажмите кнопку “Commit” чтобы выполнить коммит. Также можно изменить предыдущий коммит, выбрав радиокнопку “Amend”. Это действие также обновить область добавленных в индекс изменений содержимое предыдущего коммита. После этого вы можете как обычно добавлять или удалять файлы, а также изменить сообщение коммита. По нажатию кнопки “Commit” новый коммит затрёт предыдущий.

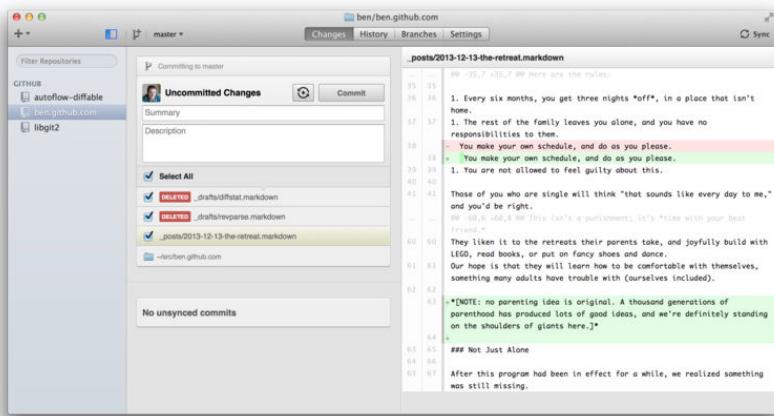
gitk и git-gui — это примеры инструментов, ориентированных на задачи. Каждый из них заточен под определённую задачу (просмотр истории или создание коммитов, соответственно) и не поддерживает фичи Git, ненужные для этой задачи.

## GitHub для Mac и Windows

Компания GitHub выпустила два инструмента, ориентированных на рабочий процесс, а не на конкретные задачи: один для Windows, второй — для Mac. Эти клиенты — хороший пример процесс-ориентированного ПО: вместо предоставления доступа ко всему функционалу Git, они концентрируются на небольшом наборе фич, работающих вместе для достижения цели. Выглядят они примерно так:

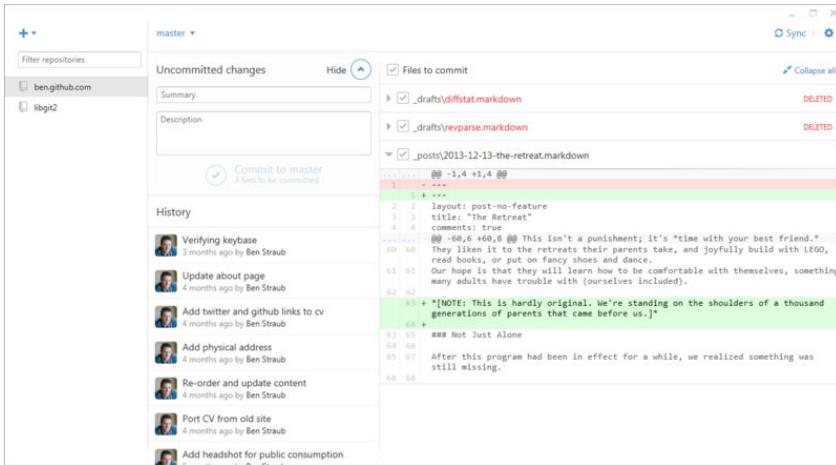
Figure 1-3.

GitHub для Mac.



**Figure 1-4.**

GitHub для Windows.



Они спроектированы по одному шаблону, поэтому мы будем рассматривать их как один продукт в этой главе. Мы не будем разбирать по косточкам эти инструменты (в конце-концов у них есть документация), а лишь быстренько взглянем на экран изменений (место, где вы будете зависать больше всего).

- Слева расположен список отслеживаемых репозиториев; можно добавить репозиторий (склонировав его, либо указав путь к существующей копии) нажатием кнопки “+” над списком.
- В центре экрана расположена область редактирования коммита: тут можно ввести сообщение коммита и выбрать файлы для включение в него. (На Windows, история коммитов расположена под этой областью, на Mac это отдельная вкладка.)
- Справа — просмотр изменений: что изменилось в рабочей директории, какие изменения войдут в коммит.
- И стоит обратить внимание на кнопку “Sync” справа вверху, которая используется для синхронизации по сети.

---

Необязательно регистрироваться на GitHub, чтобы работать с этими инструментами. Хотя они навязывают использование GitHub, оба инструмента прекрасно работают с любым другим Git сервером.

---

## УСТАНОВКА

GitHub для Windows можно скачать на <https://windows.github.com>, а для Mac — на <https://mac.github.com>. При первом запуске обе программы проведут первоначальную настройку Git, например, сконфигурируют ваше имя и email, а также устанавливают разумные значения по умолчанию для распространённых опций типа CRLF-поведение и хранилище паролей.

Оба инструмента поддерживают автообновление в фоне — это означает, что у вас всегда будет последняя версия. Это также относится к поставляемому в комплекте с ними Git'у — вам никогда не придётся обновлять его вручную. На Windows вы также получаете ярлык для запуска PowerShell с Posh-git, который мы рассмотрим далее в этой главе.

Следующий шаг — скормить программе парочку репозиториев для работы. Клиент для GitHub показывает список репозиториев, доступных вам на GitHub, и вы можете склонировать любой в один клик. Если же у вас уже есть склонированный репозиторий, просто перетяните его из окна Finder (или Windows Explorer) в окно клиента GitHub, и он будет включён в список репозиториев слева.

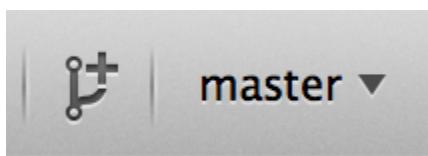
## РЕКОМЕНДУЕМЫЙ РАБОЧИЙ ПРОЦЕСС

После установки GitHub клиент можно использовать для решения кучи стандартных задач. Рекомендуемый ниже подход к работе иногда называют “GitHub Flow”. Мы рассмотрели этот рабочий процесс в главе “[The GitHub Flow](#)”, но вкратце, важны два момента: (а) вы коммитите в отдельные ветки и (б) вы регулярно забираете изменения с удалённого репозитория.

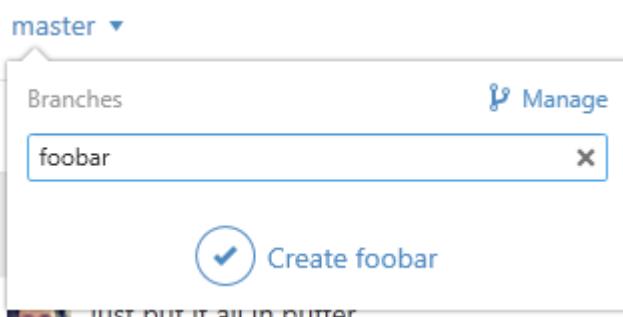
Управление ветками слегка отличается на Mac и Windows. В Mac версии для создания ветки есть кнопка вверху окна:

**Figure 1-5.**

Кнопка создания ветки на Mac.



На Windows создание ветки происходит путём ввода её имени в переключатель веток:



**Figure 1-6.**

Создание ветки в Windows.

После создания ветки добавление коммитов в неё довольно тривиально. Измените что-нибудь в рабочей директории и, переключившись в окно клиента GitHub, вы увидите свои изменения. Введите сообщение коммита, выберете файлы для включения в коммит и нажмите кнопку “Commit” (ctrl-enter or ⌘-enter).

Взаимодействие с удалёнными репозиториями происходит в первую очередь посредством кнопки “Sync”. В Git есть отдельные команды для отправки изменений на сервер, слияния изменений воедино и перемещения веток друг относительно друга, но клиент GitHub совмещает все эти команды в одну. Вот что происходит когда вы жмёте “Sync”:

1. `git pull --rebase`. Если эта команда выполнится с ошибкой, будет выполнено `git pull --no-rebase`.
2. `git push`.

Это довольно привычный, но рутинный процесс при работе по “GitHub Flow”, совмещение команд воедино действительно экономит время.

## ЗАКЛЮЧЕНИЕ

Перечисленные инструменты отлично решают поставленные перед ними задачи. С их помощью разработчики (и не только) могут начать совместную работу над проектами в считанные минуты, причём с настроенным рабочим процессом. Но если вы придерживаетесь иных подходов к использованию Git, или если вам нужно больше контроля над происходящим, мы рекомендуем вам присмотреться к другим клиентам, а то и вовсе к командной строке.

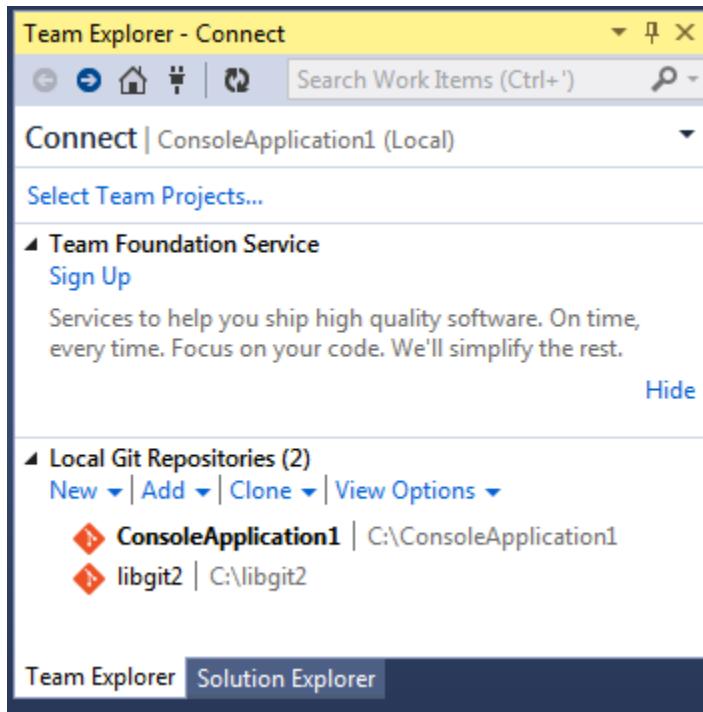
## Другие инструменты

Существует огромное множество других графических инструментов для работы с Git, начиная от специализированных, выполняющих одну задачу, заканчивая “комбайнами” покрывающими весь функционал Git. На официальном сайте Git поддерживается в актуальном состоянии список наиболее популярных оболочек: <http://git-scm.com/downloads/guis>. Более подробный список доступен на Git вики: [https://git.wiki.kernel.org/index.php/Interfaces,\\_frontends,\\_and\\_tools#Graphical\\_Interfaces](https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces).

## Git в Visual Studio

Начиная с Visual Studio 2013 Update 1, пользователям Visual Studio доступен Git-клиент, встроенный непосредственно в IDE. Visual Studio уже в течение достаточно долгого времени имеет встроенные функции управления исходным кодом, но они были ориентированы на централизованные системы с блокировкой файлов, и Git не очень хорошо вписывался в такой рабочей процесс. Поддержка Git в Visual Studio 2013 была существенно переработана по сравнению со старой версией, и в результате удалось добиться лучшей интеграции Visual Studio и Git.

Чтобы воспользоваться этим функционалом, откройте проект, который управляет Git (или выполните `git init` для существующего проекта) и выберите пункты View (Вид) > Team Explorer (Командный обозреватель) в главном меню. В результате откроется окно “Connect” (“Подключить”), которое выглядит примерно вот так:



**Figure 1-7.**

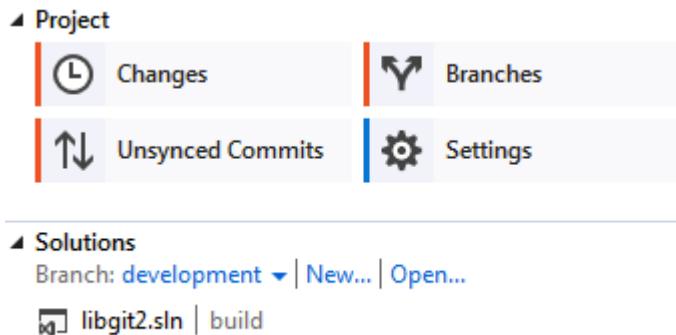
Подключение к Git-репозиторию из окна Team Explorer (Командный обозреватель).

Visual Studio запоминает все проекты, управляемые с помощью Git, которые Вы открыли, и они доступны в списке в нижней части окна. Если в списке нет проекта, который вам нужен, нажмите кнопку “Add” (“Добавить”) и укажите путь к рабочей директории. Двойной клик по одному из локальных Git-репозиториев откроет главную страницу репозитория, которая выглядит примерно так **Figure A-8**.

Это центр управления Git; когда вы пишете код, вы, вероятно, проводите большую часть своего времени на странице “Changes” (“Изменения”), но когда приходит время получать изменения, сделанные вашими коллегами по работе, вам необходимо использовать страницы “Unsynced Commits” (“Несинхронизированные фиксации”) и “Branches” (“Ветви”).

**Figure 1-8.**

“Home”  
(“Главная”)  
страница Git-  
репозитория в  
Visual Studio.



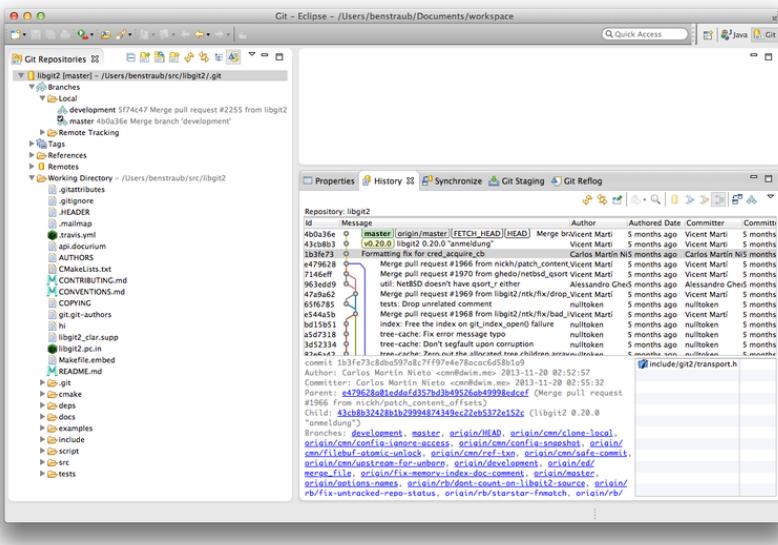
В настоящее время Visual Studio имеет мощный задача-ориентированный графический интерфейс для Git. Он включает в себя возможность линейного представления истории, различные средства просмотра, средства выполнения удаленных команд и множество других возможностей. Для просмотра полной документации по данному функционалу (которая здесь не представлена), перейдите на <http://msdn.microsoft.com/en-us/library/hh850437.aspx>.

## Git в Eclipse

В составе IDE Eclipse есть плагин под названием Egit, который предоставляет довольно полнофункциональный интерфейс для операций с Git. Воспользоваться им можно, включив Git-перспективу (Window > Open Perspective > Other..., и выбрать Git).

**Figure 1-9.**

EGit в Eclipse.



Egit поставляется с неплохой документацией, доступной меню через Help > Help Contents в разделе Egit Documentation.

## Git в Bash

Если вы используете Bash, то можете задействовать некоторые из его фишек для облегчения работы с Git. Вообще-то, Git поставляется с плагинами для нескольких шеллов, но они выключены из коробки.

Для начала, скачайте файл contrib/completion/git-completion.bash из репозитория с исходным кодом Git. Поместите его в укромное место — например, в вашу домашнюю директорию — и добавьте следующие строки в .bashrc:

```
. ~/git-completion.bash
```

Как только закончите с этим, перейдите в директорию с Git репозиторием и наберите:

```
$ git chec<tab>
```

...и Bash дополнит строку до `git checkout`. Эта магия работает для всех Git команд, их параметров, удалённых репозиториев и имён ссылок там, где это возможно.

Возможно, вам также пригодится отображение информации о репозитории, расположенному в текущей директории. Вы можете выводить сколь угодно сложную информацию, но обычно достаточно названия текущей ветки и статуса рабочей директории. Чтобы снабдить строку приветствия этой информацией, скачайте файл `contrib/completion/git-prompt.sh` из репозитория с исходным кодом Git и добавьте примерно такие строки в `.bashrc`:

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(__git_ps1 " (%s)")$\ '$
```

Часть `\w` означает текущую рабочую директорию, `\$` — индикатор суперпользователя (обычно `$` или `#`), а `__git_ps1 " (%s)"` вызывает функцию, объявленную в `git-prompt.sh`, с аргументом `' (%s)'` — строкой форматирования. Теперь ваша строка приветствия будет похожа на эту, когда вы зайдёте в директорию с Git репозиторием:

**Figure 1-10.**

Кастомизированная строка приветствия bash.



~/src/libgit2 (development \*)\$

Оба вышеперечисленных скрипта снабжены полезной документацией, загляните внутрь `git-completion.bash` и `git-prompt.sh` чтобы узнать больше.

## Git в Zsh

Git поставляется с поддержкой автодополнения для Zsh. Просто скопируйте файл `contrib/completion/git-completion.zsh` в вашу

домашнюю директорию и добавьте его в конфигурацию `.zshrc`. Интерфейс Zsh круче оного в Bash:

```
$ git che<tab>
check-attr -- display gitattributes information
check-ref-format -- ensure that a reference name is well formed
checkout -- checkout branch or paths to working tree
checkout-index -- copy files from index to working directory
cherry -- find commits not merged upstream
cherry-pick -- apply changes introduced by some existing commits
```

Возможные варианты автодополнения не просто перечислены; они снабжены полезными описаниями и вы можете выбрать нужный вариант, нажав Tab несколько раз. Это работает не только для команд Git, но и для их аргументов, названий объектов (типа ссылок и удалённых репозиториев), а также для имён файлов и других вещей.

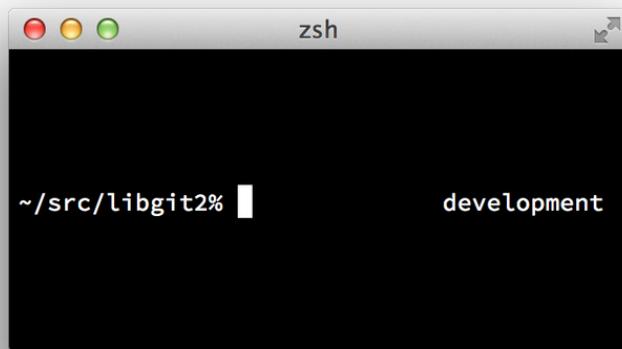
Настройка строки приветствия в Zsh похожа на таковую в Bash, но в Zsh вы можете установить дополнительную строку приветствия справа. Чтобы отобразить имя текущей ветки в правой строке приветствия, добавьте следующие строки в ваш `~/.zshrc`:

```
setopt prompt_subst
. ~/git-prompt.sh
export RPROMPT='${__git_ps1 "%s"}'
```

В результате вы будете видеть имя текущей ветки в правой части окна терминала каждый раз, как перейдёте внутрь Git репозитория. Это выглядит примерно так:

**Figure 1-11.**

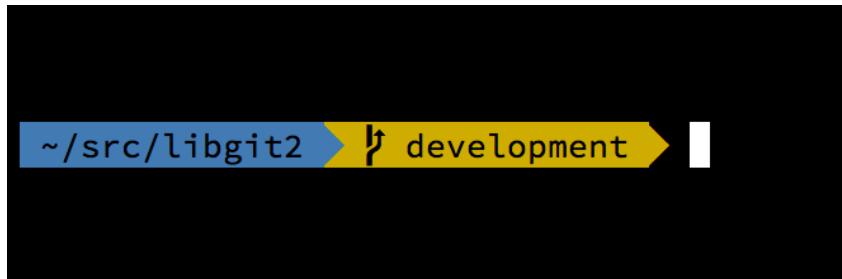
Кастомизированная строка приветствия в zsh.



Zsh настолько конфигурируем, что существуют целые фреймворки, посвящённые его улучшению. Пример такого проекта, называемый “oh-my-zsh”, расположен на <https://github.com/robbyrussell/oh-my-zsh>. Система плагинов этого проекта включает в себя мощнейший набор правил автодополнения для Git, а многие “темы” (служащие для настройки строк приветствия) отображают информацию из различных систем контроля версий. Вот пример настройки Zsh для комфортной работы с Git **Figure A-12**.

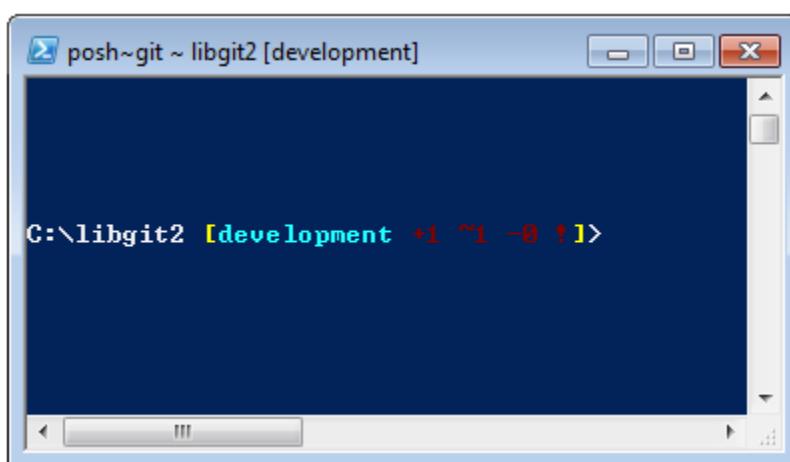
**Figure 1-12.**

Пример темы oh-my-zsh.



## Git в Powershell

Стандартный терминал командной строки Windows (cmd.exe), на самом деле, не предназначен для специализированного использования Git, но если вы используете Powershell, то это меняет дело. Пакет Posh-Git (<https://github.com/dahlbyk/posh-git>) предоставляет мощные средства завершения команд, а также расширенные подсказки, что поможет вам поддерживать состояние вашего репозитория на высоком уровне. Выглядит это примерно так:



**Figure 1-13.**

Powershell с Posh-git.

Если Вы установили приложение GitHub для Windows, то Posh-Git уже включен по умолчанию, и все, что вам остается сделать, это добавить в файл `profile.ps1` (который обычно расположен в `C:\Users\<username>\Documents\WindowsPowerShell`) следующие строки:

```
. (Resolve-Path "$env:LOCALAPPDATA\GitHub\shell.ps1")
. $env:github_posh_git\profile.example.ps1
```

Если же вы не используете GitHub для Windows, просто загрузите последнюю версию пакета Posh-Git с (<https://github.com/dahlbyk/posh-git>), и распакуйте его в директорию `WindowsPowerShell`. После этого запустите Powershell с правами администратора и выполните следующие команды:

```
> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser -Confirm
> cd ~\Documents\WindowsPowerShell\posh-git
> .\install.ps1
```

Это добавит необходимые строки в ваш файл `profile.ps1`, и Posh-Git станет доступным при следующем запуске терминала.

## Заключение

Теперь вы знаете, как использовать мощь Git'a внутри инструментов, используемых вами каждый день и как получить доступ к репозиториям из ваших собственных программ.

# Встраивание Git'а в ваши приложения

B

Если вы пишете приложение для разработчиков, с высокой вероятностью оно выиграет от интеграции с системой управления версиями. Даже приложения для обычных пользователей — например, текстовые редакторы — могут извлечь пользу из систем управления версиями. Git хорошо работает во многих сценариях.

Если вам нужно интегрировать Git в ваше приложение, у вас есть три варианта: запуск шелла и выполнение в нем Git команд, Libgit2 или JGit.

## Git из командной строки

Первый вариант встраивания Git'а — порождение шелла и использование Git из него для выполнения задач. Плюсом данного подхода является каноничность и поддержка всех возможностей Git. Это наиболее простой подход, так как большинство сред исполнения предоставляют достаточно простые средства вызова внешних процессов с параметрами командной строки. Тем не менее, у этого подхода есть некоторые недостатки.

Первый — результат выполнения команд представлен в виде простого текста. Это означает, что вам придётся анализировать вывод команд (который может поменяться со временем) чтобы получить результат выполнения, что неэффективно и подвержено ошибкам.

Следующий недостаток — отсутствие восстановления после ошибок. Если репозиторий был повреждён, или если пользователь указал неверный параметр конфигурации, Git просто откажется выполнять большинство операций.

Ещё одним недостатком является необходимость управления порождённым процессом. При таком использовании Git требует

выделения в отдельный процесс с шеллом, что может добавить сложностей. Попытка скоординировать множество таких процессов (особенно при работе с одним репозиторием из нескольких процессов) может оказаться нетривиальной задачей.

## Libgit2

Другой доступный вам вариант — это использование библиотеки Libgit2. Libgit2 — это свободная от внешних зависимостей реализация Git, фокусирующаяся на предоставлении приятного API другим программам. Вы можете найти её на <http://libgit2.github.com>.

Для начала, давайте посмотрим на что похож C API. Вот краткий обзор:

```
// Открытие репозитория
git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// Получение HEAD коммита
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// Вывод некоторых атрибутов коммита на печать
printf("%s", git_commit_message(commit));
const git_signature *author = git_commit_author(commit);
printf("%s <%s>\n", author->name, author->email);
const git_oid *tree_id = git_commit_tree_id(commit);

// Очистка
git_commit_free(commit);
git_repository_free(repo);
```

Первая пара строк открывают Git репозиторий. Тип `git_repository` представляет собой ссылку на репозиторий с кешем в памяти. Это самый простой метод, его можно использовать если вы знаете точный путь к рабочей директории репозитория или к `.git` директории. Существует расширенный вариант — `git_repository_open_ext` — который принимает набор параметров для поиска репозитория. Функция `git_clone` с компаньонами используется для клонирования удалённого репозитория. И, наконец, `git_repository_init` используется для создания нового репозитория с нуля.

Следующий кусок кода использует `rev-parse` синтаксис (см. “Ссылки на ветки”) чтобы получить коммит на который указывает

HEAD. Возвращаемый тип — это указатель на структуру `git_object`, которая представляет любой объект, хранящийся во внутренней БД Git. `git_object` — родительский тип для нескольких других; внутренняя структура всех этих типов одинаковая, так что вы можете относительно безопасно преобразовывать типы друг в друга. В нашем случае `git_object_type(head_commit)` вернёт `GIT_OBJ_COMMIT`, так что мы вправе привести типы для `git_commit`.

Затем мы получаем некоторые свойства коммита. Последняя строчка в этом фрагменте кода использует тип `git_oid` — это внутреннее представление SHA-1 в Libgit2.

Глядя на этот пример, можно сделать несколько выводов:

- Если вы объявили указатель и передали его в одну из функций Libgit2, она, возможно, вернёт целочисленный код ошибки. Значение 0 означает успешное выполнение операции, всё что меньше — означает ошибку.
- Если Libgit2 возвращает вам указатель, вы ответственны за очистку ресурсов
- Если Libgit2 возвращает `const`-указатель, вам не нужно заботится о его очистке, но он может оказаться невалидным, если объект на который он ссылается будет уничтожен.
- Писать на C — сложно.

Последний пункт намекает на маловероятность использования С при работе с Libgit2. К счастью, существует ряд обёрток над Libgit2 для различных языков, которые позволяют довольно удобно работать с Git репозиториями, используя ваш язык программирования и среду исполнения. Давайте взглянем на пример ниже, написанный с использованием Ruby и обёртки над Libgit2 для него под названием Rugged, которую можно найти на <https://github.com/libgit2/rugged>.

```
repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"
tree = commit.tree
```

Как видите, код гораздо менее загромождён. Во-первых, Rugged использует исключения: он может кинуть ошибку типа `ConfigError` или `ObjectError` чтобы просигнализировать о сбое. Во-вторых, нет необходимости явно подчищать ресурсы, потому что в Ruby есть сборщик мусора. Давайте посмотрим на более сложный пример — создание коммита с нуля:

```

blob_id = repo.write("Blob contents", :blob) ❶

index = repo.index
index.read_tree(repo.head.target.tree)
index.add(:path => 'newfile.txt', :oid => blob_id) ❷

sig = {
 :email => "bob@example.com",
 :name => "Bob User",
 :time => Time.now,
}

commit_id = Rugged::Commit.create(repo,
 :tree => index.write_tree(repo), ❸
 :author => sig, ❹
 :committer => sig, ❺
 :message => "Add newfile.txt", ❻
 :parents => repo.empty? ? [] : [repo.head.target].compact, ❼
 :update_ref => 'HEAD', ❽
)
commit = repo.lookup(commit_id) ❾

```

- ❶ Создание нового blob'a, содержащего файл.
- ❷ Заполнение индекса содержимым дерева HEAD и добавление нового файла newfile.txt.
- ❸ Создание нового дерева в ODB и использование его для нового коммита.
- ❹ Мы используем одну и ту же сигнатуру для автора и коммиттера.
- ❺ Сообщение в коммите.
- ❼ При создании коммита нужно указать его предков. Для этих целей мы используем HEAD как единственного родителя.
- ❽ Rugged (как и Libgit2) дополнительно могут обновить HEAD-указатель.
- ❾ Результирующее значение — это SHA-1 хэш нового коммита, по которому его можно вычитать из репозитория для получения объекта типа Commit.

Код на Ruby приятен и чист, а благодаря тому что Libgit2 делает основную работу ещё и выполняется довольно быстро. На случай

если вы пишете не на Ruby, мы рассмотрим другие обёртки над Libgit2 в “Обёртки для других языков”.

## Расширенная функциональность

У Libgit2 есть несколько фич, выходящих за рамки стандартного Git. Одна из таких фич — расширяемость: Libgit2 позволяет использовать нестандартные “бэкэнды” для некоторых операций; таким образом вы можете хранить объекты по-иному, нежели это делает Git из коробки. Например, Libgit2 позволяет использовать нестандартные хранилища для конфигурации, ссылок и внутренней базы данных объектов.

Давайте взглянем, как это работает. Код ниже заимствован из примеров, написанных командой разработчиков Libgit2, вы можете ознакомиться с ними на <https://github.com/libgit2/libgit2-backends>. Вот как можно использовать нестандартное хранилище объектов:

```
git_odb *odb;
int error = git_odb_new(&odb); ①

git_odb_backend *my_backend;
error = git_odb_backend_mine(&my_backend, /*...*/); ②

error = git_odb_add_backend(odb, my_backend, ③

git_repository *repo;
error = git_repository_open(&repo, "some-path");
error = git_repository_set_odb(repo); ④
```

(Заметьте, ошибки перехватываются, но не обрабатываются. Мы надеемся, ваш код лучше нашего.)

- ① Инициализация “фронтэнда” для пустого хранилища объектов (ODB), используемого в качестве контейнера “бэкэндов”, которые будут выполнять работу.
- ② Инициализация произвольного ODB бэкэнда.
- ③ Добавление “бэкэнда” к “фронтэнду”
- ④ Открытие репозитория и указание ему использовать ODB созданный на предыдущем этапе.

Что же скрыто внутри `git_odb_backend_mine`? Это ваша собственная имплементация ODB, и вы можете делать что угодно,

лишь убедитесь в правильности заполнения полей структуры `git_odb_backend`. Например, внутри *может* быть следующий код:

```
typedef struct {
 git_odb_backend parent;

 // Другие поля
 void *custom_context;
} my_backend_struct;

int git_odb_backend_mine(git_odb_backend **backend_out, /*...*/)
{
 my_backend_struct *backend;

 backend = calloc(1, sizeof (my_backend_struct));

 backend->custom_context = ...;

 backend->parent.read = &my_backend_read;
 backend->parent.read_prefix = &my_backend_read_prefix;
 backend->parent.read_header = &my_backend_read_header;
 // ...

 *backend_out = (git_odb_backend *) backend;

 return GIT_SUCCESS;
}
```

Важный момент: первое поле структуры `my_backend_struct` имеет тип `git_odb_backend` — это обеспечивает расположение полей в памяти в формате, ожидаемом Libgit2. Оставшиеся поля можно располагать произвольно; сама структура может быть любого нужного вам размера.

Функция инициализации выделяет память под структуру, устанавливает произвольный контекст и заполняет поля структуры `parent`, которые необходимо поддерживать. Взгляните на файл `include/git2/sys/odb_backend.h` в исходном коде Libgit2 чтобы узнать полный список сигнатур доступных методов; в вашем конкретном случае вы сами решаете, какие из них необходимо имплементировать.

## Обёртки для других языков

У Libgit2 есть привязки для многих языков. Здесь мы приведём лишь парочку небольших примеров; полный список поддерживаемых

языков гораздо шире и включает в себя, среди прочего, C++, Go, Node.js, Erlang и JVM, на разных стадиях зрелости. Официальный список обёрток можно найти на <https://github.com/libgit2>. Примеры кода ниже показывают как получить сообщение HEAD-коммита (что-то типа `git log -l`).

## LIBGIT2SHARP

Если вы пишете под платформы .NET / Mono, LibGit2Sharp (<https://github.com/libgit2/libgit2sharp>) — то, что прописал вам доктор. Эта библиотека написана на C# и все прямые вызовы методов Libgit2 тщательно обёрнуты в управляемый CLR код. Вот как будет выглядеть наш пример:

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

Также существует NuGet пакет для десктопных Windows-приложений, который поможет начать разработку ещё быстрее.

## OBJECTIVE-GIT

Если вы пишете приложение для продукции Apple, то скорее всего оно написано на Objective-C. Обёртка над Libgit2 в этом случае называется Objective-Git: (<https://github.com/libgit2/objective-git>). Пример кода:

```
GTRepository *repo =
 [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"] error:NULL];
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];
```

Objective-git полностью интероперабелен с новым языком Swift, так что не бойтесь переходить на него с Objective-C.

## PYGIT2

Обёртка над Libgit2 для Python называется Pygit2, её можно найти на <http://www.pygit2.org/>. И наш пример будет выглядеть так:

```
pygit2.Repository("/path/to/repo") # открыть репозиторий
 .head.resolve() # получить прямую ссылку
 .get_object().message # получить коммит, прочитать сообщение
pygit2.Repository("/path/to/repo") # открыть репозиторий
 .head # получить текущую ветку
 .peel(pygit2.Commit) # получить последний коммит ветки
 .message # прочитать сообщение
```

## Дальнейшее чтение

Конечно же, покрыть полностью все возможности Libgit2 не в силах этой книги. Если вы хотите подробнее ознакомиться с Libgit2, можете начать с API-документации по адресу <https://libgit2.github.com/libgit2> и с руководства на <https://libgit2.github.com/docs>. Для привязок к другим языкам, загляните в README и тестовые исходники, довольно часто в них встречаются ссылки на полезные материалы по теме.

## JGit

Если вы хотите использовать Git из Java-программ, существует библиотека для работы с Git, называемая JGit. Она достаточно полно реализует функционал Git и написана полностью на Java. Эта библиотека получила широкое распространение в Java-мире. Проект JGit находится под опекой Eclipse и расположен по адресу <http://www.eclipse.org/jgit>.

### Приступая к работе

Существует несколько способов добавить JGit в проект и начать писать код с использованием предоставляемого API. Возможно, самый простой путь — использование Maven. Подключение библиотеки происходит путём добавления следующих строк в секцию `<dependencies>` в вашем pom.xml:

```
<dependency>
 <groupId>org.eclipse.jgit</groupId>
 <artifactId>org.eclipse.jgit</artifactId>
 <version>3.5.0.201409260305-r</version>
</dependency>
```

С момента выхода книги скорее всего появились новые версии JGit, проверьте обновления на <http://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit>. После обновления конфигурации Maven автоматически скачает JGit нужной версии и добавит её к проекту.

Если вы управляете зависимостями вручную, собранные бинарные пакеты JGit доступны на <http://www.eclipse.org/jgit/download>. Использовать их в своём проекте можно следующим способом:

```
javac -cp .:org.eclipse.jgit-3.5.0.201409260305-г.јаг App.java
java -cp .:org.eclipse.jgit-3.5.0.201409260305-г.јаг App
```

## Служебный API

У JGit есть два уровня API: служебный (“plumbing” API, “трубопровод”) и пользовательский (“porcelain” API, “фарфор”). Эта терминология заимствована из самого Git и JGit разделён на две части: “фарфоровый” API предоставляет удобные методы для распространённых задач прикладного уровня (тех, для решения которых вы бы использовали обычные Git-команды) и “сантехнический” API для прямого взаимодействия с низкоуровневыми объектами репозитория.

Начальная точка большинства сценариев использования JGit — класс `Repository` и первое, что необходимо сделать — это создать объект данного класса. Для репозиториев основанных на файловой системе (да, JGit позволяет использовать другие модели хранения) эта задача решается с помощью класса `FileRepositoryBuilder`:

```
// Создание нового репозитория; директория должна существовать
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
 new File("/tmp/new_геро/.git"));

// Открыть существующий репозиторий
Repository existingRepo = new FileRepositoryBuilder()
 .setGitDir(new File("my_геро/.git"))
 .build();
```

Вызовы методов билдера можно объединять в цепочку чтобы указать всю информацию для поиска репозитория независимо от того, знает ли ваша программа его точное месторасположение или нет. Можно читать системные переменные (`.readEnvirement()`), начать поиск с произвольного места в рабочей директории (`.setWorkTree(...).findGitDir()`), или просто открыть директорию `.git` по указанному пути.

После создания объекта типа `Repository`, вам будет доступен широкий набор операций над ним. Краткий пример:

```
// Получение ссылки
Ref master =repo.getRef("master");

// Получение объекта, на который она указывает
ObjectId masterTip = master.getObjectId();
```

```

// Использование rev-parse выражений
ObjectId obj = repo.resolve("HEAD^{tree}");

// Получение "сырых" данных
ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// Создание ветки
RefUpdate createBranch1 = repo.updateRef("refs/heads/branch1");
createBranch1.setNewObjectId(masterTip);
createBranch1.update();

// Удаление ветки
RefUpdate deleteBranch1 = repo.updateRef("refs/heads/branch1");
deleteBranch1.setForceUpdate(true);
deleteBranch1.delete();

// Работа с конфигурацией
Config cfg = repo.getConfig();
String name = cfg.getString("user", null, "name");

```

Тут происходит много интересного, давайте разберёмся по порядку.

Первая строка получает указатель на ссылку master. JGit автоматически получает *актуальную* информацию о master, хранимую по пути refs/heads/master, и возвращает объект, предоставляющий доступ к информации о ссылке. Вы можете получить имя (.getName()), а также целевой объект прямой ссылки (.getObjectId()) или ссылку, на которую указывает другая символьная ссылка (.getTarget()). Объекты типа Ref также служат для представления ссылок на теги и самих тегов; вы можете узнать, является ли тег “конечным” (“peeled”), т.е. ссылается ли он на целевой объект потенциально длинной цепи тегов.

Вторая строка получает объект на который указывает ссылка master в виде ObjectId. ObjectId представляют SHA-1 хэш объекта, который, возможно, сохранён внутри базы данных объектов Git. Следующая строка похожа на предыдущую, но используется rev-parse синтаксис (см. детали в “Ссылки на ветки”); вы можете использовать любой, подходящий формат и JGit вернёт либо валидный ObjectId для указанного объекта, либо null.

Следующие две строки показывают, как можно получить содержимое объекта. В этом примере мы используем ObjectLoader .copyTo() чтобы передать содержимое файла прямиком в stdout, но у ObjectLoader есть методы для чтения типа и размера объекта, а

также для считывания объекта в виде массива байтов. Для больших объектов (у которых `.isLarge()` возвращает `true`) можно использовать метод `.openStream()` для открытия потока последовательного чтения объекта без полной загрузки в память.

Следующие строки показывают, как создать новую ветку. Мы создаём объект типа `RefUpdate`, устанавливаем некоторые параметры и вызываем метод `.update()` чтобы инициировать изменение. После этого мы удаляем эту же ветку. Обратите внимание на необходимость вызова `.setForceUpdate(true)` для корректной работы; иначе вызов `.delete()` вернёт `REJECTED` и ничего не произойдёт.

Последний кусок кода показывает как получить параметр `user.name` из файлов конфигурации Git. Созданный объект `Config` будет использовать открытый ранее репозиторий для чтения локальной конфигурации, также он автоматически находит файлы глобальной и системной конфигурации и использует их для чтения значений.

Это лишь малая часть служебного API JGit; в вашем распоряжении окажется гораздо больше классов и методов. Мы не показали как JGit обрабатывает ошибки. JGit использует механизм исключений Java; иногда он бросает стандартные исключения (типа `IOException`), иногда — специфичные для JGit (например `NoRemoteRepositoryException`, `CorruptObjectException` и `NoMergeBaseException`).

## Пользовательский API

Служебные API достаточно всеобъемлющи, но сложны в использовании для простых задач вроде добавления файла в индекс или создания нового коммита. У JGit есть API более высокого уровня, входная точка в который — это класс `Git`:

```
Repository repo;
// создание репозитория...
Git git = new Git(repo);
```

В классе `Git` можно найти отличный набор высокоуровневых “текущих” методов (`builder-style / fluent interface`). Давайте взглянем на пример — результат выполнения этого кода смахивает на `git ls-remote`:

```
CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username", "p4ssw0rd");
Collection<Ref> remoteRefs = git.lsRemote()
.setCredentialsProvider(cp)
```

```

.setRemote("origin")
.setTags(true)
.setHeads(false)
.call();
for (Ref ref : remoteRefs) {
 System.out.println(ref.getName() + " -> " + ref.getObjectId().name());
}

```

Тут показан частый случай использования класса Git: методы возвращают тот же объект, на котором вызваны, что позволяет чередовать их друг за другом, устанавливая параметры. Финальный аккорд — непосредственное выполнение команды с помощью метода `.call()`. В этом примере мы запрашиваем список тегов (но не “головы” веток) с удалённого репозитория `origin`. Обратите внимание на использование класса `CredentialsProvider` для аутентификации.

Множество команд доступно в классе Git, включая такие как `add`, `blame`, `commit`, `clean`, `push`, `rebase`, `revert`, `reset` и другие.

## Дальнейшее чтение

Это лишь небольшой пример всех возможностей JGit. Если вы заинтересованы в более детальной работе с JGit, вот список источников информации для старта:

- Официальная документация по JGit API доступна в Интернете на <http://download.eclipse.org/jgit/docs/latest/apidocs>. Это обычный Javadoc, так что ваша любимая IDE может скачать её и использовать оффлайн.
- “Поваренная книга” JGit, расположенная по адресу <https://github.com/centic9/jgit-cookbook> включает в себя много готовых “рецептов” использования JGit для решения тех или иных задач.
- Вопрос на StackOverflow <http://stackoverflow.com/questions/6861881> содержит несколько полезных ссылок.

# Команды Git



В этой книге было показано больше десятка различных команд Git и мы приложили много усилий, чтобы рассказать вам о них, выстроив некий логический порядок, постепенно внедряя команды в сюжет. Но такой подход “размазал” описания команд по всей книге.

В этом приложении мы пройдёмся по всем командам, о которых шла речь, и сгруппируем их по смыслу. Мы расскажем, что делает каждая команда и укажем главы в книге, где эта команда использовалась.

## Настройка и конфигурация

Две довольно распространённые команды, используемые как сразу после установки Git'a, так и в повседневной практике для настройки и получения помощи — это `config` и `help`.

### `git config`

Сотни вещей в Git'e работают без всякой конфигурации, используя параметры по умолчанию. Для большинства из них вы можете задать иные умолчания, либо вовсе использовать собственные значения. Это включает в себя целый ряд настроек, начиная от вашего имени и заканчивая цветами в терминале и вашим любимым редактором. Команда `config` хранит и читает настройки в нескольких файлах, так что вы можете задавать значения глобально или для конкретных репозиториев.

Команда `git config` используется практически в каждой главе этой книги.

В главе “Первоначальная настройка Git” мы использовали эту команду для указания имени, адреса электронной почты и редактора ещё до того, как начать использовать Git.

В главе “[Псевдонимы в Git](#)” мы показали, как можно использовать её для создания сокращённых вариантов команд с длинными списками опций, чтобы не печатать их все каждый раз.

В главе “[Rebasing](#)” мы использовали `config` чтобы задать поведение `--rebase` по умолчанию для команды `git pull`.

В главе “[Хранилище учетных данных](#)” мы использовали её для задания хранилища ваших HTTP паролей.

В главе “[Keyword Expansion](#)” мы показали как настроить фильтры содержимого для данных, перемещаемых между индексом и рабочей директорией.

Ну и практически вся глава “[Git Configuration](#)” посвящена этой команде.

## **git help**

Команда `git help` служит для отображения встроенной документации Git о других командах. И хотя мы приводим описания самых популярных команд в этой главе, полный список параметров и флагов каждой команды доступен через `git help <command>`.

Мы представили эту команду в главе “[Как получить помощь?](#)” и показали как её использовать, чтобы найти больше информации о команде `git shell` в главе “[Настраиваем сервер](#)”.

## **Клонирование и создание репозиториев**

Существует два способа создать Git репозиторий. Первый — клонировать его из существующего репозитория (например, по сети); второй — создать репозиторий в существующей директории.

## **git init**

Чтобы превратить обычную директорию в Git репозиторий и начать версионировать файлы в ней, просто запустите `git init`.

Впервые мы продемонстрировали эту команду в главе “[Создание Git-репозитория](#)” на примере создания нового репозитория для последующей работы с ним.

Мы немного поговорили о смене названия ветки по умолчанию с “`master`” на что-нибудь другое в главе “[Удалённые ветки](#)”.

Мы использовали эту команду для создания чистого репозитория для работы на стороне сервера в главе “Размещение голого репозитория на сервере”.

Ну и наконец мы немного покопались во внутренностях этой команды в главе “Сантехника и Фарфор”.

## git clone

На самом деле `git clone` работает как обёртка над некоторыми другими командами. Она создаёт новую директорию, переходит внутрь и выполняет `git init` для создания пустого репозитория, затем она добавляет новый удалённый репозиторий (`git remote add`) для указанного URL (по умолчанию он получит имя `origin`), выполняет `git fetch` для этого репозитория и, наконец, обновляет вашу рабочую директорию до последнего коммита, используя `git checkout`.

Команда `git clone` используется в десятке различных мест в этой книге, но мы перечислим наиболее интересные упоминания.

Первоначальное знакомство происходит в главе “Клонирование существующего репозитория”, где мы даём немного объяснений и приводим несколько примеров.

В главе “Установка Git на сервер” мы рассмотрели как использовать опцию `--bare`, чтобы создать копию Git репозитория без рабочей директории.

В главе “Создание пакетов” мы использовали `git clone` для распаковки упакованного с помощью `git bundle` репозитория.

Наконец, в главе “Клонирование проекта с подмодулями” мы научились использовать опцию `--recursive` чтобы упростить клонирование репозитория с субмодулями.

И хотя `git clone` используется во многих других местах в книге, перечисленные выше так или иначе отличаются от других вариантов использования.

## Основные команды

Всего несколько команд нужно для базового варианта использования Git для ведения истории изменений.

## **git add**

Команда `git add` добавляет содержимое рабочей директории в индекс (staging area) для последующего коммита. По умолчанию `git commit` использует лишь этот индекс, так что вы можете использовать `git add` для сборки слепка вашего следующего коммита.

Это одна из ключевых команд Git, мы упоминали о ней десятки раз на страницах книги. Ниже перечислены наиболее интересные варианты использования этой команды.

Знакомство с этой командой происходит в главе “[Tracking New Files](#)”.

О том как использовать `git add` для разрешения конфликтов слияния написано в главе “[Основные конфликты слияния](#)”.

В главе “[Интерактивное индексирование](#)” показано как использовать `git add` для добавления в индекс лишь отдельных частей изменённого файла.

В главе “[Деревья](#)” показано как эта команда работает на низком уровне, чтобы вы понимали, что происходит за кулисами.

## **git status**

Команда `git status` показывает состояния файлов в рабочей директории и индексе: какие файлы изменены, но не добавлены в индекс; какие ожидают коммита в индексе. Вдобавок к этому выводятся подсказки о том, как изменить состояние файлов.

Мы познакомили вас с этой командой в главе “[Checking the Status of Your Files](#)”, разобрали стандартный и упрощённый формат вывода. И хотя мы использовали `git status` повсеместно в этой книге, практически все варианты использования покрыты в указанной главе.

## **git diff**

Команда `git diff` используется для вычисления разницы между любыми двумя Git деревьями. Это может быть разница между вашей рабочей директорией и индексом (собственно `git diff`), разница между индексом и последним коммитом (`git diff --staged`), или между любыми двумя коммитами (`git diff master branchB`).

Мы познакомили вас с основами этой команды в главе “[Viewing Your Staged and Unstaged Changes](#)”, где показали как посмотреть какие изменения уже добавлены в индекс, а какие — ещё нет.

О том как использовать эту команду для проверки на проблемы с пробелами с помощью аргумента `--check` можно почитать в главе “Commit Guidelines”.

Мы показали вам как эффективно сравнивать ветки используя синтаксис `git diff A...B` в главе “Determining What Is Introduced”.

В главе “Продвинутое слияние” показано использование опции `-w` для скрытия различий в пробельных символах, а также рассказано как сравнивать конфликтующие изменения с опциями `--theirs`, `--ours` и `--base`.

Использование этой команды с опцией `--submodule` для сравнения изменений в субмодулях показано в главе “Начало работы с подмодулями”.

## **git difftool**

Команда `git difftool` просто запускает внешнюю утилиту сравнения для показа различий в двух деревьях, на случай если вы хотите использовать что-либо отличное от встроенного просмотрщика `git diff`.

Мы лишь вкратце упомянули о ней в главе “Viewing Your Staged and Unstaged Changes”.

## **git commit**

Команда `git commit` берёт все данные, добавленные в индекс с помощью `git add`, и сохраняет их слепок во внутренней базе данных, а затем сдвигает указатель текущей ветки на этот слепок.

Вы познакомились с основами модели коммитов в главе “Committing Your Changes”. Там же мы продемонстрировали использование опций `-a` для добавления всех изменений в индекс без использования `git add`, что может быть удобным в повседневном использовании, и `-m` для передачи сообщения коммита без запуска полноценного редактора.

В главе “Операции отмены” мы рассказали об опции `--amend`, используемой для изменения последнего совершённого коммита.

В главе “О ветвлении в двух словах” мы более подробно познакомились с тем, что делает команда `git commit` и почему она делает это именно так.

Мы показали вам как подписывать ваши коммиты, используя опцию `-S` в главе “Подпись коммитов”.

И наконец мы заглянули внутрь команды `git commit` в главе “Commit Objects” и узнали что она делает за кулисами.

## **git reset**

Команда `git reset`, как можно догадаться из названия, используется в основном для отмены изменений. Она изменяет указатель HEAD и, опционально, состояние индекса. Также эта команда может изменить файлы в рабочей директории при использовании параметра `--hard`, что может привести к потере наработок при неправильном использовании, так что убедитесь в серьёзности своих намерений прежде чем использовать его.

Мы рассказали об основах использования `git reset` в главе “Отмена подготовки файла”, где эта команда использовалась для удаления файла из индекса, добавленного туда с помощью `git add`.

В главе “Раскрытие тайн `reset`”, полностью посвящённой этой команде, мы разобрались в деталях её использования.

Мы использовали `git reset --hard` чтобы отменить слияние в главе “Прерывание слияния”, там же было продемонстрировано использование команды `git merge --abort` для этих целей, которая работает как обёртка над `git reset`.

## **git rm**

Команда `git rm` используется в Git для удаления файлов из индекса и рабочей директории. Она похожа на `git add` с тем лишь исключением, что она удаляет, а не добавляет файлы для следующего коммита.

Мы немного разобрались с этой командой в главе “Removing Files”, показали как удалять файлы из рабочей директории и индекса и только из индекса, используя флаг `--cached`.

Ещё один вариант использования `git rm` приведён в главе “Removing Objects”, где мы вкратце объяснили как использовать опцию `--ignore-unmatch` при выполнении `git filter-branch`, которая подавляет ошибки удаления несуществующих файлов. Это может быть полезно для автоматически выполняемых скриптов.

## **git mv**

Команда `git mv` — это всего лишь удобный способ переместить файл, а затем выполнить `git add` для нового файла и `git rm` для старого.

Мы лишь вкратце упомянули это команду в главе “[Moving Files](#)”.

## **git clean**

Команда `git clean` используется для удаления мусора из рабочей директории. Это могут быть результаты сборки проекта или файлы конфликтов слияний.

Мы рассмотрели множество опций и сценариев использования этой команды в главе “[Очистка вашей рабочей директории](#)”.

# **Ветвление и слияния**

За создание новых веток и слияние их воедино отвечает несколько Git команд.

## **git branch**

Команда `git branch` — это своего рода “менеджер веток”. Она умеет перечислять ваши ветки, создавать новые, удалять и переименовывать их.

Большая часть главы [Chapter 3](#) посвящена этой команде, она используется повсеместно в этой главе. Впервые команда `branch` была представлена в разделе “[Создание новой ветки](#)”, а большинство таких её фич как перечисление и удаление веток были разобраны в разделе “[Управление ветками](#)”.

В главе “[Отслеживание веток](#)” мы показали как использовать сочетание `git branch -u` для отслеживания веток.

Наконец, мы разобрались что происходит за кулисами этой команды в главе “[Ссылки в Git](#)”.

## **git checkout**

Команда `git checkout` используется для переключения веток и выгрузки их содержимого в рабочую директорию.

Мы познакомились с этой командой в главе “Переключение веток” вместе с `git branch`.

В главе “Отслеживание веток” мы узнали как использовать флаг `--track` для отслеживания веток.

В главе “Использование `checkout` в конфликтах” мы использовали эту команду с опцией `--conflict=diff3` для разрешения конфликтов заново, в случае если предыдущее решение не подходило по некоторым причинам.

Мы рассмотрел детали взаимосвязи этой команды и `git reset` в главе “Раскрытие тайн `reset`”.

Мы исследовали внутренние механизмы этой команды в главе “`HEAD`”.

## **git merge**

Команда `git merge` используется для слияния одной или нескольких веток в текущую. Затем она устанавливает указатель текущей ветки на результирующий коммит.

Мы познакомили вас с этой командой в главе “Основы ветвления”. И хотя `git merge` встречается в этой книге повсеместно, практически все использования имеют вид `git merge <branch>` с указанием единственной ветки для слияния.

Мы узнали как делать “сплющенные” слияния (когда Git делает слияние в виде нового коммита, без сохранения всей истории работы) в конце главы “Forked Public Project”.

В главе “Продвинутое слияние” мы глубже разобрались с процессом слияния и этой командой, включая флаги `-Xignore-all-whitespace` и `--abort`, используемый для отмены слияния в случае возникновения проблем.

Мы научились проверять криптографические подписи перед слияниями если ваш проект использует GPG в главе “Подпись коммитов”.

Ну и наконец в главе “Слияние субдеревьев” мы познакомились со слиянием поддеревьев.

## **git mergetool**

Команда `git mergetool` просто вызывает внешнюю программу слияний, в случае если у вас возникли проблемы слияния.

Мы вкратце упомянули о ней в главе “Основные конфликты слияния” и рассказали как настроить свою программу слияния в главе “External Merge and Diff Tools”.

## git log

Команда `git log` используется для просмотра истории коммитов, начиная с самого свежего и уходя к истокам проекта. По умолчанию, она показывает лишь историю текущей ветки, но может быть настроена на вывод истории других, даже нескольких сразу, веток. Также её можно использовать для просмотра различий между ветками на уровне коммитов.

Практически во всех главах книги эта команда используется для демонстрации истории проекта.

Мы познакомились с `git log` и некоторыми её деталями в главе “Просмотр истории коммитов”. Там мы видели использование опций `-r` и `--stat` для получения представления об изменениях в каждом коммите, а также `--pretty` and `--oneline` для настройки формата вывода этой команды — более полным и подробным или кратким.

В главе “Создание новой ветки” мы использовали опцию `--decorate` чтобы отобразить указатели веток на истории коммитов, а также `--graph` чтобы просматривать историю в виде дерева.

В главах “Private Small Team” и “Диапазоны фиксаций” мы рассмотрели синтаксис `branchA..branchB` для просмотра уникальных для заданной ветки коммитов. Мы часто использовали этот приём в “Диапазоны фиксаций”.

В главах “История при слиянии” и “Три точки” мы рассмотрели синтаксис `branchA..branchB` и опцию `--left-right` для просмотра что находится в первой, либо второй ветке, но не сразу в обеих. Также в главе “История при слиянии” рассмотрели опцию `--merge`, которая может быть полезной при разрешении конфликтов, а также `--cc` для просмотра конфликтов слияния в истории проекта.

В главе “RefLog-сокращения” мы использовали опцию `-g` для вывода `git reflog`, используя `git log`.

В главе “Поиск” мы рассмотрели использование опций `-S` и `-L` для поиска событий в истории проекта, например, истории развития какой-либо фичи.

В главе “Подпись коммитов” мы показали, как использовать опцию `--show-signature` для отображения строки валидации подписи для каждого коммита в `git log`.

## **git stash**

Команда `git stash` используется для временного сохранения всех незакоммиченных изменений для очистки рабочей директории без необходимости коммитить незавершённую работу в новую ветку.

Эта команда практически целиком раскрыта в главе “Прибережение и очистка”.

## **git tag**

Команда `git tag` используется для задания постоянной метки на какой-либо момент в истории проекта. Обычно она используется для релизов.

Мы познакомились и разобрались с ней в главе “Работа с метками” и использовали на практике в “Tagging Your Releases”.

Мы научились создавать подписанные с помощью GPG метки, используя флаг `-s`, и проверять их, используя флаг `-v`, в главе “Подпись результатов вашей работы”.

## **Совместная работа и обновление проектов**

Не так уж много команд в Git требуют сетевого подключения для своей работы, практически все команды оперируют с локальной копией проекта. Когда вы готовы поделиться своими наработками, всего несколько команд помогут вам работать с удалёнными репозиториями.

## **git fetch**

Команда `git fetch` связывается с удалённым репозиторием и забирает из него все изменения, которых у вас пока нет и сохраняет их локально.

Мы познакомились с ней в главе “Fetching and Pulling from Your Remotes” и продолжили знакомство в “Удалённые ветки”.

Мы использовали эту команду в нескольких примерах из главы “Contributing to a Project”.

Мы использовали её для скачивания запросов на слияние (pull request) из других репозиториев в главе “Pull Request Refs”, также мы рассмотрели использование `git fetch` для работы с упакованными репозиториями в главе “Создание пакетов”.

Мы рассмотрели тонкую настройку `git fetch` в главе и “Спецификации ссылок”.

## **git pull**

Команда `git pull` работает как комбинация команд `git fetch` и `git merge`, т.е. Git вначале забирает изменения из указанного удалённого репозитория, а затем пытается слить их с текущей веткой.

Мы познакомились с ней в главе “Fetching and Pulling from Your Remotes” и показали как узнать, какие изменения будут приняты в случае применения в главе “Inspecting a Remote”.

Мы также увидели как она может оказаться полезной для разрешения сложностей при перемещении веток в главе “Rebase When You Rebase”.

Мы показали как можно использовать только URL удалённого репозитория без сохранения его в списке удалённых репозиториев в главе “Checking Out Remote Branches”.

И наконец мы показали как проверять криптографические подписи полученных коммитов, используя опцию `--verify-signatures` в главе “Подпись коммитов”.

## **git push**

Команда `git push` используется для установления связи с удалённым репозиторием, вычисления локальных изменений отсутствующих в нём, и собственно их передачи в вышеупомянутый репозиторий. Этой команде нужно право на запись в репозиторий, поэтому она использует аутентификацию.

Мы познакомились с этой командой в главе “Pushing to Your Remotes”. Там мы рассмотрели основы обновления веток в удалённом репозитории. В главе “Отправка изменений” мы подробнее познакомились с этой командой, а в “Отслеживание веток” мы узнали как настроить отслеживание веток для автоматической передачи на удалённый репозиторий. В главе “Удаление веток на удалённом сервере” мы использовали флаг `--delete` для удаления веток на сервере, используя `git push`.

На протяжении главы “Contributing to a Project” мы показали несколько примеров использования `git push` для совместной работы в нескольких удалённых репозиториях одновременно.

В главе “Публикация изменений в подмодуле” мы использовали опцию `--recurse-submodules` чтобы удостовериться, что все

субмодули будут опубликованы перед отправкой на проекта на сервер, что может быть реально полезным при работе с репозиториями, содержащими субмодули.

В главе “*Other Client Hooks*” мы поговорили о триггере `pre-push`, который может быть выполнен перед отправкой данных, чтобы проверить возможность этой отправки.

Наконец, в главе “*Спецификации ссылок для отправки данных на сервер*” мы рассмотрели передачу данных с полным указанием передаваемых ссылок, вместо использования распространённых сокращений. Это может быть полезным если вы хотите очень точно указать, какими изменениями хотите поделиться.

## **git remote**

Команда `git remote` служит для управления списком удалённых репозиториев. Она позволяет сохранять длинные URL репозиториев в виде понятных коротких строк, например “`origin`”, так что вам не придётся забивать голову всякой ерундой и набирать её каждый раз для связи с сервером. Вы можете использовать несколько удалённых репозиториев для работы и `git remote` поможет добавлять, изменять и удалять их.

Эта команда детально рассмотрена в главе “*Working with Remotes*”, включая вывод списка удалённых репозиториев, добавление новых, удаление или переименование существующих.

Она используется практически в каждой главе, но всегда в одном и том же виде: `git remote add <имя> <URL>`.

## **git archive**

Команда `git archive` используется для упаковки в архив указанных коммитов или всего репозитория.

Мы использовали `git archive` для создания тарбала (`tar.gz` файла) всего проекта для передачи по сети в главе “*Preparing a Release*”.

## **git submodule**

Команда `git submodule` используется для управления вложенными репозиториями. Например, это могут быть библиотеки или другие, используемые не только в этом проекте ресурсы. У команды `submod-`

ule есть несколько под-команд — add, update, sync и др. — для управления такими репозиториями.

Эта команда упомянута и полностью раскрыта в главе “Подмодули”.

## Осмотр и сравнение

### git show

Команда git show отображает объект в простом и человекопонятном виде. Обычно она используется для просмотра информации о метке или коммите.

Впервые мы использовали её для просмотра информации об аннотированной метке в главе “Аннотированные метки”.

В главе “Выбор ревизии” мы использовали её для показа коммитов, подпадающих под различные селекторы диапазонов.

Ещё одна интересная вещь, которую мы проделывали с помощью git show в главе “Ручное слияние файлов” — это извлечение содержимого файлов на различных стадиях во время конфликта слияния.

### git shortlog

Команда git shortlog служит для подведения итогов команды git log. Она принимает практически те же параметры, что и git log, но вместо простого листинга всех коммитов, они будут сгруппированы по автору.

Мы показали, как можно использовать эту команду для создания классных списков изменений (changelogs) в главе “The Shortlog”.

### git describe

Команда git describe принимает на вход что угодно, что можно трактовать как коммит (ветку, тег) и выводит более-менее человекочитаемую строку, которая не изменится в будущем для данного коммита. Это может быть использовано как более удобная, но по-прежнему уникальная, замена SHA-1.

Мы использовали git describe в главах “Generating a Build Number” и “Preparing a Release” чтобы сгенерировать название для архивного файла с релизом.

## Отладка

В Git есть несколько команд, используемых для нахождения проблем в коде. Это команды для поиска места в истории, где проблема впервые проявилась и собственно виновника этой проблемы.

### git bisect

Команда `git bisect` — это чрезвычайно полезная утилита для поиска коммита в котором впервые проявился баг или проблема с помощью автоматического бинарного поиска.

О ней упоминается только в главе “Бинарный поиск”, где она полностью и раскрыта.

### git blame

Команда `git blame` выводит перед каждой строкой файла SHA-1 коммита, последний раз менявшего эту строку и автора этого коммита. Это помогает в поисках человека, которому нужно задавать вопросы о проблемном куске кода.

Эта команда полностью разобрана в главе “Аннотация файла”.

### git grep

Команда `git grep` используется для поиска любой строки или регулярного выражения в любом из файлов вашего проекта, даже в более ранних его версиях.

Она полностью разобрана в разделе “Git Grep” и упоминается лишь там.

## Внесение исправлений

Некоторые команды в Git основываются на подходе к рассмотрению коммитов в терминах внесённых ими изменений, т.е. рассматривают историю коммитов как цепочку патчей. Ниже перечислены эти команды.

## **git cherry-pick**

Команда `git cherry-pick` используется для того чтобы взять изменения, внесённые каким-либо коммитом, и попытаться применить их заново в виде нового коммита наверху текущей ветки. Это может оказаться полезным чтобы забрать парочку коммитов из другой ветки без полного слияния с той веткой.

Мы продемонстрировали работу этой команды в главе “[Rebasing and Cherry Picking Workflows](#)”.

## **git rebase**

`git rebase` — это “автоматизированный” `cherry-pick`. Он выполняет ту же работу, но для цепочки коммитов, тем самым как бы перенося ветку на новое место.

Мы в деталях разобрались с механизмом переноса веток в главе “[Rebasing](#)”, включая рассмотрение потенциальных проблем переноса опубликованных веток при совместной работе.

Мы использовали эту команду на практике для разбиения истории на два репозитория в главе “[Замена](#)”, наряду с использованием флага `--onto`.

В главе “[Rerere](#)” мы рассмотрели случай возникновения конфликта во время переноса коммитов.

Также мы познакомились с интерактивным вариантом `git rebase`, включающимся с помощью опции `-i`, в главе “[Изменение сообщений нескольких фиксаций](#)”.

## **git revert**

Команда `git revert` — полная противоположность `git cherry-pick`. Она создаёт “антикоммит” для указанного коммита, таким образом отменяя изменения, внесённые в нём..

Мы использовали её в главе “[Отмена фиксации](#)” чтобы отменить коммит слияния (merge commit).

## **Работа с помощью электронной почты**

Множество проектов, использующих Git (включая сам Git), активно используют списки рассылок для координирования процесса разработки. В Git есть несколько команд, помогающих в этом, начиная

от генерации патчей, готовых к пересылке по электронной почте, заканчивая применением таких патчей прямиком из папки “входящие”.

## **git apply**

Команда `git apply` применяет патч, сформированный с помощью команды `git diff` или `GNU diff`. Она делает практически то же самое, что и команда `patch`.

Мы продемонстрировали использование этой команды в главе “[Applying Patches from E-mail](#)” и описали случаи, когда вы возможно захотите ею воспользоваться.

## **git am**

Команда `git am` используется для применения патчей из ящика входящих сообщений электронной почты, в частности, тех что используют формат `mbox`. Это используется для простого получения изменений через `email` и применения их к проекту.

Мы рассмотрели использование этой команды в главе “[Applying a Patch with am](#)”, включая такие опции как `--resolved`, `-i` и `-3`.

Существует набор триггеров, которые могут оказаться полезными при использовании `git am` для процесса разработки. О них рассказано в главе “[E-mail Workflow Hooks](#)”.

Также мы использовали `git am` для применения сформированного из Github'овского запроса на слияние patch-файла в главе “[Email Notifications](#)”.

## **git format-patch**

Команда `git format-patch` используется для создания набора патчей в формате `mbox` которые можно использовать для отправки в список рассылки.

Мы рассмотрели процесс отсылки изменений в проект, использующий `email` для разработки в главе “[Public Project over E-Mail](#)”.

## **git send-email**

Команда `git send-email` используется для отсылки патчей, сформированных с использованием `git format-patch`, по электронной почте.

Процесс отсылки изменений по электронной почте в проект рассмотрен в главе “[Public Project over E-Mail](#)”.

## **git request-pull**

Команда `git request-pull` используется для генерации примерного текста сообщения для отсылки кому-либо. Если у вас есть ветка, хранящаяся на публичном сервере, и вы хотите чтобы кто-либо забрал эти изменения без возни с отсылкой патчей по электронной почте, вы можете выполнить эту команду и послать её вывод тому человеку.

Мы показали, как пользоваться этой командой в главе “[Forked Public Project](#)”.

## **Внешние системы**

В Git есть несколько стандартных команд для работы с другими системами контроля версий.

### **git svn**

Команда `git svn` используется для работы с сервером Subversion. Это означает, что вы можете использовать Git в качестве SVN клиента, забирать изменения и отправлять свои собственные на сервер Subversion.

Мы разобрались с этой командой в главе “[Git и Subversion](#)”.

### **git fast-import**

Для других систем контроля версий, либо для импорта произвольно форматированных данных, вы можете использовать `git fast-import`, которая умеет преобразовывать данные в формат, понятный Git’у.

Мы детально рассмотрели эту команду в главе “[A Custom Importer](#)”.

## **Администрирование**

Если вы администрируете Git репозиторий или вам нужно исправить что-либо, Git предоставляет несколько административных команд вам в помощь.

### **git gc**

Команда `git gc` запускает сборщик мусора в вашем репозитории, который удаляет ненужные файлы из хранилища объектов и эффективно упаковывает оставшиеся файлы.

Обычно, эта команда выполняется автоматически без вашего участия, но, если пожелаете, можете вызвать её вручную. Мы рассмотрели некоторые примеры её использования в главе “Уход за репозиторием”.

### **git fsck**

Команда `git fsck` используется для проверки внутренней базы данных на предмет наличия ошибок и несоответствий.

Мы лишь однажды использовали её в главе “Восстановление данных” для поиска более недостижимых (*dangling*) объектов.

### **git reflog**

Команда `git reflog` просматривает историю изменения голов веток на протяжении вашей работы для поиска коммитов, которые вы могли внезапно потерять, переписывая историю.

В основном, мы рассматривали эту команду в главе “RefLog-сокращения”, где мы показали пример использования этой команды, а также как использовать `git log -g` для просмотра той же информации, используя `git log`.

Мы на практике рассмотрели восстановление потерянной ветки в главе “Восстановление данных”.

### **git filter-branch**

Команда `git filter-branch` используется для переписывания содержимого коммитов по заданному алгоритму, например, для полного удаления файла из истории или для вычленения истории

лишь части файлов в проекте для вынесения в отдельный репозиторий.

В главе “Удаление файла из каждой фиксации” мы объяснили механизм работы этой команды и рассказали про использование опций `--commit-filter`, `--subdirectory-filter` и `--tree-filter`.

В главах “Git-p4” и “TFS” мы использовали эту команду для исправления импортированных репозиториев.

## Низкоуровневые команды

Также в этой книге встречались некоторые низкоуровневые (“сантехнические”) команды.

Первая из них — это `ls-remote`, с которой мы столкнулись в главе “Pull Request Refs” и использовали для просмотра ссылок на сервере.

В главах “Ручное слияние файлов”, “Rerere” и “Индекс” мы использовали команду `ls-files` чтобы просмотреть “сырые” данные в индексе.

Мы также упоминали о команде `rev-parse` в главе “Ссылки на ветки”, используемой для превращения практически произвольно отформатированных строк в SHA-1 указатели.

Так или иначе, большинство низкоуровневых команд собрано в главе **Chapter 10**, которая на них и сосредоточена. Мы старались избегать этих команд в других местах в этой книге.



# Index

## Symbols

- \$EDITOR, **404**
- \$VISUAL
  - see \$EDITOR, **404**
- .gitignore, **406**
- .NET, **577**
- @{upstream}, **117**
- @{u}, **117**

## A

- aliases, **82**
- Apache, **148**
- Apple, **577**
- archiving, **422**
- attributes, **415**
- autocorrect, **407**

## B

- bash, **565**
- binary files, **416**
- bitnami, **152**
- branches, **85**
  - basic workflow, **93**
  - creating, **88**
  - deleting remote, **119**
  - diffing, **192**
  - long-running, **105**
  - managing, **103**
  - merging, **98**
  - remote, **108, 191**
  - switching, **89**
  - topic, **106, 187**
  - tracking, **116**
  - upstream, **116**
- build numbers, **201**

## C

- C#, **577**
- Cocoa, **577**
- color, **407**
- commit templates, **405**
- contributing, **163**
  - private managed team, **173**
  - private small team, **166**
  - public large project, **183**
  - public small project, **179**
- credential caching, **38**
- credentials, **394**
- CRLF, **38**
- crlf, **412**
- CVS, **27**

## D

- difftool, **408**
- distributed git, **159**

## E

- Eclipse, **564**
- editor
  - changing default, **56**
- email, **185**
  - applying patches from, **187**
- excludes, **406, 509**

## F

- files
  - moving, **60**
  - removing, **58**
- forking, **161, 212**

## G

- Git as a client, **439**

git commands  
  add, 49, 49, 50  
  am, 188  
  apply, 187  
  archive, 202  
  branch, 88, 103  
  checkout, 89  
  cherry-pick, 198  
  clone, 46  
    bare, 139  
    commit, 56, 86  
    config, 40, 42, 56, 82, 185, 403  
    credential, 394  
    daemon, 146  
    describe, 201  
    diff, 53  
      check, 164  
    fast-import, 498  
    fetch, 74  
    fetch-pack, 537  
    filter-branch, 496  
    format-patch, 184  
    gitk, 556  
    gui, 556  
    help, 42, 146  
    http-backend, 148  
    init, 46, 49  
      bare, 140, 144  
    instaweb, 150  
    log, 61  
    merge, 96  
      squash, 183  
    mergetool, 102  
    p4, 470, 495  
    pull, 74  
    push, 75, 81, 114  
    rebase, 120  
    receive-pack, 535  
    remote, 72, 73, 75, 76  
    request-pull, 180  
    rerere, 199  
    send-pack, 535  
    shortlog, 202  
    show, 79  
    show-ref, 443  
    status, 48, 56  
    svn, 440  
    tag, 77, 78, 80  
    upload-pack, 537

git-svn, 440  
git-tf, 479

git-tfs, 479  
GitHub, 205  
  API, 256  
  Flow, 213  
  organizations, 247  
  pull requests, 216  
  user accounts, 205  
GitHub for Mac, 558  
GitHub for Windows, 558  
gitk, 556  
GitLab, 152  
GitWeb, 150  
GPG, 406  
Graphical tools, 555  
GUIs, 555

## H

hooks, 424  
  post-update, 135

## I

ignoring files, 52  
Importing  
  from Mercurial, 492  
  from others, 498  
  from Perforce, 495  
  from Subversion, 490  
  from TFS, 497

integrating work, 193

Interoperation with other VCSs  
  Mercurial, 453  
  Perforce, 462  
  Subversion, 439  
  TFS, 479

## J

java, 578  
jgit, 578

## K

keyword expansion, 419

## L

libgit2, 572  
line endings, 412  
Linus Torvalds, 30  
Linux, 30

installing, 37  
log filtering, 68  
log formatting, 64

## M

Mac  
installing, 37  
maintaining a project, 186  
master, 87  
Mercurial, 453, 492  
mergetool, 408  
merging, 98  
  conflicts, 100  
  strategies, 423  
  vs. rebasing, 129  
Migrating to Git, 489  
Mono, 577

## O

Objective-C, 577  
origin, 109

## P

pager, 406  
Perforce, 27, 31, 462, 495  
  Git Fusion, 462  
policy example, 427  
posh-git, 569  
Powershell, 38  
powershell, 569  
protocols  
  dumb HTTP, 135  
  git, 137  
  local, 132  
  smart HTTP, 134  
  SSH, 136  
pulling, 118  
pushing, 114  
Python, 577

## R

rebasing, 119  
  perils of, 125  
  vs. merging, 129  
references  
  remote, 108  
releasing, 202  
rerere, 199

Ruby, 573

## S

serving repositories, 131  
  git protocol, 146  
  GitLab, 152  
  GitWeb, 150  
  HTTP, 148  
  SSH, 141  
SHA-1, 33  
shell prompts  
  bash, 565  
  powershell, 569  
  zsh, 566  
SSH keys, 142  
  with GitHub, 206  
staging area  
  skipping, 57  
Subversion, 27, 31, 160, 439, 490

## T

tab completion  
  bash, 565  
  powershell, 569  
  zsh, 566  
tags, 77, 200  
  annotated, 78  
  lightweight, 79  
  signing, 200  
TFS, 479, 497  
TFVC (see TFS)

## V

version control, 25  
  centralized, 27  
  distributed, 28  
  local, 26  
Visual Studio, 562

## W

whitespace, 412  
Windows  
  installing, 38  
workflows, 159  
  centralized, 159  
  dictator and lieutenants, 161  
  integration manager, 160  
  merging, 194

merging (large), **196**  
rebasing and cherry-picking, **198**

**Z**  
**zsh, 566**

**X**  
**Xcode, 37**