THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

EVERYTHING YOU NEED TO KNOW ABOUT GIT

Scott Chacon and Ben Straub

Apress®

Pro Git

Scott Chacon, Ben Straub

Версия 2.1.124-4-gbef0ce5, 24.08.2026

Содержание

Лицензия . . . 1
Предисловие . . . 2
Пролог от Бена Страуба . . . 4
Посвящается . . . 5
Участники . . . 6
Вступление . . . 8
Введение . . . 10
- О системе контроля версий . . . 10
- Краткая история Git . . . 14
- Что такое Git? . . . 14
- Командная строка . . . 18
- Установка Git . . . 19
- Первоначальная настройка Git . . . 22
- Как получить помощь? . . . 25
- Заключение . . . 26

Основы Git . . . 27
- Создание Git-репозитория . . . 27
- Запись изменений в репозиторий . . . 29
- Просмотр истории коммитов . . . 42
- Операции отмены . . . 49
- Работа с удалёнными репозиториями . . . 53
- Работа с тегами . . . 58
- Псевдонимы в Git . . . 64
- Заключение . . . 65

Ветвление в Git . . . 66
- О ветвлении в двух словах . . . 66
- Основы ветвления и слияния . . . 73
- Управление ветками . . . 82
- Работа с ветками . . . 86
- Удалённые ветки . . . 89
- Перебазирование . . . 99
- Заключение . . . 109

Git на сервере . . . 110
- Протоколы . . . 110
- Установка Git на сервер . . . 115
- Генерация открытого SSH ключа . . . 118
- Настраиваем сервер . . . 119
- Git-демон . . . 122

Умный HTTP . . . 123
GitWeb . . . 125
GitLab . . . 127
Git-хостинг . . . 131
Заключение . . . 131
Распределённый Git . . . 132
Распределённый рабочий процесс . . . 132
Участие в проекте . . . 136
Сопровождение проекта . . . 160
Заключение . . . 175
GitHub . . . 176
Настройка и конфигурация учётной записи . . . 176
Внесение собственного вклада в проекты . . . 182
Сопровождение проекта . . . 202
Управление организацией . . . 217
Создание сценариев GitHub . . . 220
Заключение . . . 231
Инструменты Git . . . 232
Выбор ревизии . . . 232
Интерактивное индексирование . . . 240
Припрятывание и очистка . . . 245
Подпись . . . 251
Поиск . . . 255
Перезапись истории . . . 259
Раскрытие тайн reset . . . 267
Продвинутое слияние . . . 290
Rerere . . . 308
Обнаружение ошибок с помощью Git . . . 314
Подмодули . . . 318
Создание пакетов . . . 340
Замена . . . 343
Хранилище учётных данных . . . 352
Заключение . . . 357
Настройка Git . . . 358
Конфигурация Git . . . 358
Атрибуты Git . . . 369
Хуки в Git . . . 377
Пример принудительной политики Git . . . 381
Заключение . . . 390
Git и другие системы контроля версий . . . 391
Git как клиент . . . 391

Переход на Git . . . 428
Заключение . . . 445
Git изнутри . . . 446
Сантехника и Фарфор . . . 446
Объекты Git . . . 447
Ссылки в Git . . . 458
Pack-файлы . . . 462
Спецификации ссылок . . . 465
Протоколы передачи данных . . . 468
Обслуживание репозитория и восстановление данных . . . 474
Переменные окружения . . . 481
Заключение . . . 487
Приложение A: Git в других окружениях . . . 488
Графические интерфейсы . . . 488
Git в Visual Studio . . . 493
Git в Visual Studio Code . . . 495
Git в Eclipse . . . 495
Git в IntelliJ / PyCharm / WebStorm / PhpStorm / RubyMine . . . 496
Git в Sublime Text . . . 497
Git в Bash . . . 497
Git в Zsh . . . 498
Git в PowerShell . . . 500
Заключение . . . 502
Приложение B: Встраивание Git в ваши приложения . . . 503
Git из командной строки . . . 503
Libgit2 . . . 503
JGit . . . 508
go-git . . . 512
Dulwich . . . 514
Приложение C: Команды Git . . . 516
Настройка и конфигурация . . . 516
Клонирование и создание репозиториев . . . 518
Основные команды . . . 519
Ветвление и слияния . . . 522
Совместная работа и обновление проектов . . . 524
Осмотр и сравнение . . . 527
Отладка . . . 527
Внесение исправлений . . . 528
Работа с помощью электронной почты . . . 529
Внешние системы . . . 530
Администрирование . . . 530

Низкоуровневые команды 531
Индекс 532

Лицензия

Это произведение распространяется по свободной лицензии Creative Commons Attribution-NonCommercial-ShareAlike 3.0. Ознакомиться с текстом лицензии вы можете на сайте https://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 четырёх 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 четырёхлетней давности ничего подобного не подозревал. Одна из главных целей издания — затронуть в сообществе Git эти рубежи.

Сообщество свободного программного обеспечения тоже испытало 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 GitHub. На момент публикации у сайта было лишь несколько тысяч пользователей и четверо разработчиков. Когда же я пишу это предисловие, GitHub объявляет о десяти миллионах размещённых проектов, около пяти миллионах аккаунтов разработчиков и более 230 сотрудниках. Его можно любить или ненавидеть — в любом случае GitHub 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 GitHub в качестве примера хостинга, с которым мне никогда не было особо удобно работать. Мне не сильно нравилось писать то, что, по-моему, было ресурсом сообщества, а также упоминать в нём о моей компании. Меня по-прежнему волнует это противоречие, но важность GitHub в сообществе Git бесспорна. Вместо некоего примера Git-хостинга я решил посвятить этот раздел книги детальному описанию сути GitHub и его эффективному использованию. Если вы собираетесь узнать, как пользоваться Git, то умение пользоваться GitHub даст вам возможность поучаствовать в огромном сообществе, ценном вне зависимости от выбранного вами Git-хостинга.

Другим изменением с момента первой публикации стала разработка и развитие протокола HTTP для сетевых Git-транзакций. Из соображений упрощения большинство примеров из книги были переделаны с SSH на HTTP.

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.

Пролог от Бена Страуба

Я подсел на Git после первого издания этой книги. Это изменило моё представление о стиле создания программного обеспечения, которое было более естественным, чем всё, что я видел до этого. К тому моменту я уже несколько лет был разработчиком, но это позволило мне пойти по другому, гораздо более интересному пути.

Сейчас, годы спустя, я принимаю участие в реализации основных изменений в Git, работал в крупнейшей Git-хостинг-компании, путешествовал по миру и учил людей Git. Я долго не думал, когда Скот спросил меня, хочу ли я поучаствовать в работе над вторым изданием.

Работать над этой книгой было большим удовольствием и честью для меня. Надеюсь, она поможет вам так же, как и мне.

Посвящается

Моей жене Бекки, без которой это приключение никогда бы не началось. — Бен

Это издание посвящено моим девочкам. Моей жене Джессике, которая поддерживала меня все эти годы, и моей дочери Жозефине, которая поддержит меня, когда я стану слишком стар, чтобы понимать, что происходит. — Скотт

Участники

Поскольку эта книга Open Source, на протяжении многих лет мы получаем замечания об опечатках и другие доработки. Здесь собраны все люди, которые внесли свой вклад в русскую версию Pro Git как проекта с открытым исходным кодом. Спасибо всем за то, что помогли сделать эту книгу лучше.

```
Adrien Ollier           Gennadii K              Rikudo
Aleh Suprunovich        Grigory Tumakov         Robin Kuzmin
Aleksandr Nokhrin       Haruo Nakayama          Robo Burned
Aleksey Zanin           Hokid                   Roma Zvarich
Alex Chumakov           Igor Bakutin            Sanders Kleinfeld
Alex Dudarik            Igor Bereza             Sarah Schneider
Alexander Ilyin         JFF                     Sergei Zelenyi
Alexander Krasnikov     Jean-Noël Avila         Sergey Khrebtov
Alexander Osadchy       Jon Forrest             Sergey Kuznetsov
Alexandre Garnier       Justin Clift            Sergey Matveev
Alexey Sosnovsky        Konstantin              Sergey Romanov
Andre Polykanine        Leo Sadovsky            Sergey Sarkisyan
Andrei Babintsev        Leonid Kozarin          Sergey Solomein
Andrei Dascalu          Logan Hasson            Sevak Avetisyan
Andrew MacFie           LoginovIlya             Siarhei Krukau
Anna Borzenko           Louise Corrigan         Stephan van Maris
Anton Kreis             Luc Morin               Steven Roddis
Anton Petrunin          Masood Fallahpoor       TheDmitry
Anton Tsyganenko        Matthew Miner           Thomas Ackermann
Antonino Ingargiola     Mattias Vannergard      V. Oz
Arkadij Chistyj         Max Lazarev             Vasya-Chajko
Arun Sathiya            Maxim Kostrikin         Viktor Dmitriyev
Bogdan Doroschenko      Michael MacAskill       Viktor Yegay
Carlos Martín Nieto     Mihail Verenich         Vitalii Elenhaupt
Changwoo Park           Mike Charles            Vladimir Nikinorov
Christoph Prokop        Mike Thibodeau          Vladislav Meshochkin
Christopher Wilson      Mikhail Filippov        Vladislav Osmanov
Damien Tournoud         Mikhail Sudarkin        alcomy
Dan Schmidt             Murad                   bkarpov
Denis Kubashevskiy      Nail Samatov            d355
Denis Savitskiy         Niels Widger            dualsky
Dexter Morganov         Nikita Belevich         fgsfds
Dimio                   Nikolai Shabalin        fidomax
Dmitri Gouliaev         Obasado                 foobar47
Dmitri Karpovich        Oleg Karnauhov          ioannkozlitin
Dmitri Tikhonov         Oleg Zolotarev          iwasborn
Dmitriy Kindeev         Pablo Schläpfer         kastolom
Dmitriy Krasilnikov     Paciupa                 lyobzik
Dmitry Artemov          Pavel Akhrameev         mailer21
Dzmitry Aryka           Pavel Dovlatov          nktaa
EXL                     Pavel Janík             petsuter
Eldar Khayrullin        Pavel Tolstolytko       realbigest
```

```
Eplehova                Pavel Zhdanov           rutriff
Eugene Rexilier         Peter Kokot             slollo
Evgenii Terechkov       Philippe Miossec
Evgeny Petrov           Richard Hoyle
```

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 трёх приложений данной книги.

В **Главе 1** мы охватим системы контроля версий (Version Control System, VCS) и азы Git. Никаких технических штучек, только то, что, собственно, такое Git, почему он пришёл 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. Также мы разберём различные настройки хостинга, если вы предпочитаете держать сервер у кого-нибудь другого.

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

Глава 6 посвящена хостингу GitHub и его инструментам. Мы разберём регистрацию, управление учётной записью, создание и использование Git репозиториев, как вносить вклад в чужие проекты и как принимать чужой вклад в собственный проект, а также программный интерфейс GitHub 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. По прочтению этой главы 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 разберётся 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. Также 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 виртуозно 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 A** мы рассмотрим примеры использования Git в различных окружениях, разберём варианты с различными 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 B** мы изучим скрипты и расширения для Git с помощью libgit2 и JGit. Если вы заинтересованы в написании сложных и быстрых инструментов и нуждаетесь в низкоуровневом доступе к Git, вы найдёте здесь необходимую информацию.

Наконец, в **Приложении C** мы заново пройдёмся 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.

Введение

Эта глава о том, как начать работу с Git. Вначале изучим основы систем 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 времени и позволяющая вернуться позже к определённой 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 VCS) — 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 вызвал проблему, кто поставил задачу и когда и многое другое. Использование VCS также значит, в целом, что если вы сломали что-то или потеряли файлы, то спокойно можете всё исправить. В дополнение ко всему вы получите всё это без каких-либо дополнительных усилий.

Локальные системы контроля версий

Многие люди в качестве метода контроля версий применяют копирование файлов в отдельный каталог (возможно даже, каталог с отметкой по времени, если они достаточно сообразительны). Данный подход очень распространён из-за его простоты, однако он невероятно сильно подвержен появлению ошибок. Можно легко забыть, в каком каталоге 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 VCS с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль версий.

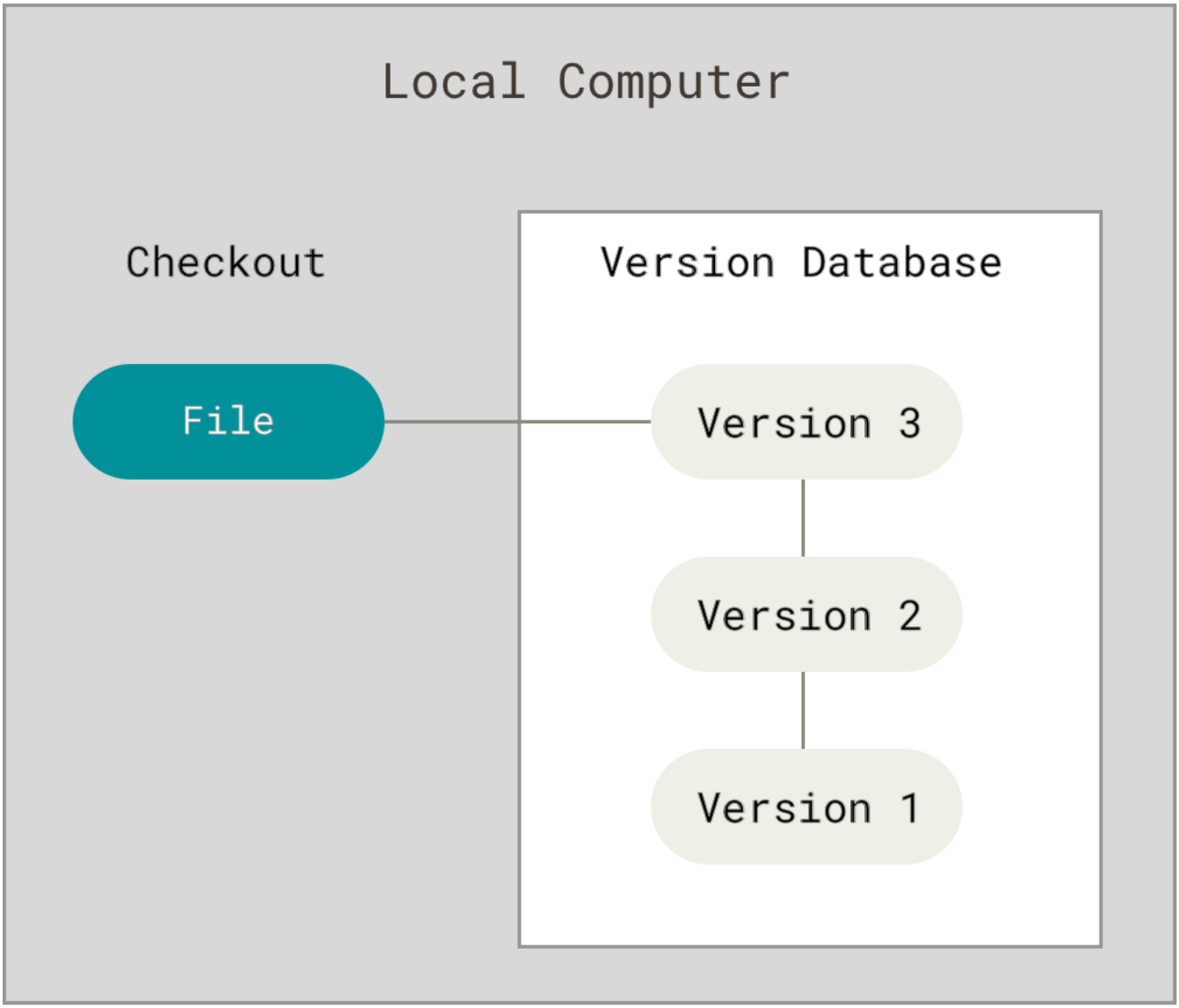


Рисунок 1. Локальный контроль версий

Одной из популярных VCS была система RCS, которая и сегодня распространяется со многими компьютерами. 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 (Centralized Version Control System, далее CVCS). Такие системы как CVS, Subversion и Perforce используют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Применение CVCS являлось стандартом на протяжении многих лет.

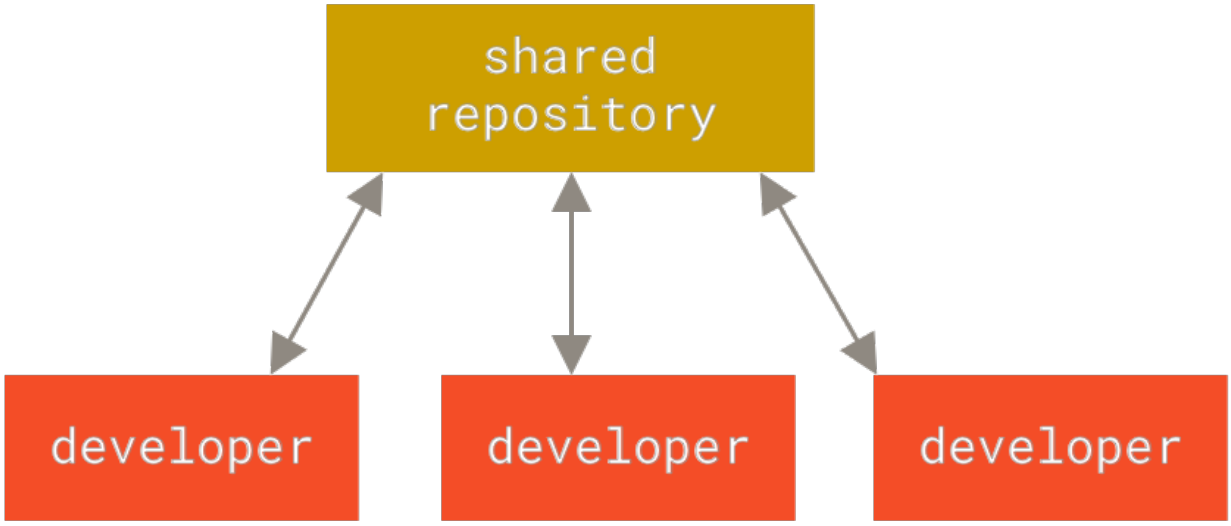


Рисунок 2. Централизованный контроль версий

Такой подход имеет множество преимуществ, особенно перед локальными VCS. 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 CVCS, 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 ни сохранить сделанные изменения, ни обменяться ими с другими разработчиками. Если жёсткий диск, на котором хранится центральная база данных, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные VCS страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

Распределённые системы контроля версий

Здесь в игру вступают распределённые системы контроля версий (Distributed Version Control System, далее DVCS). В DVCS (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 полной копией всех данных.

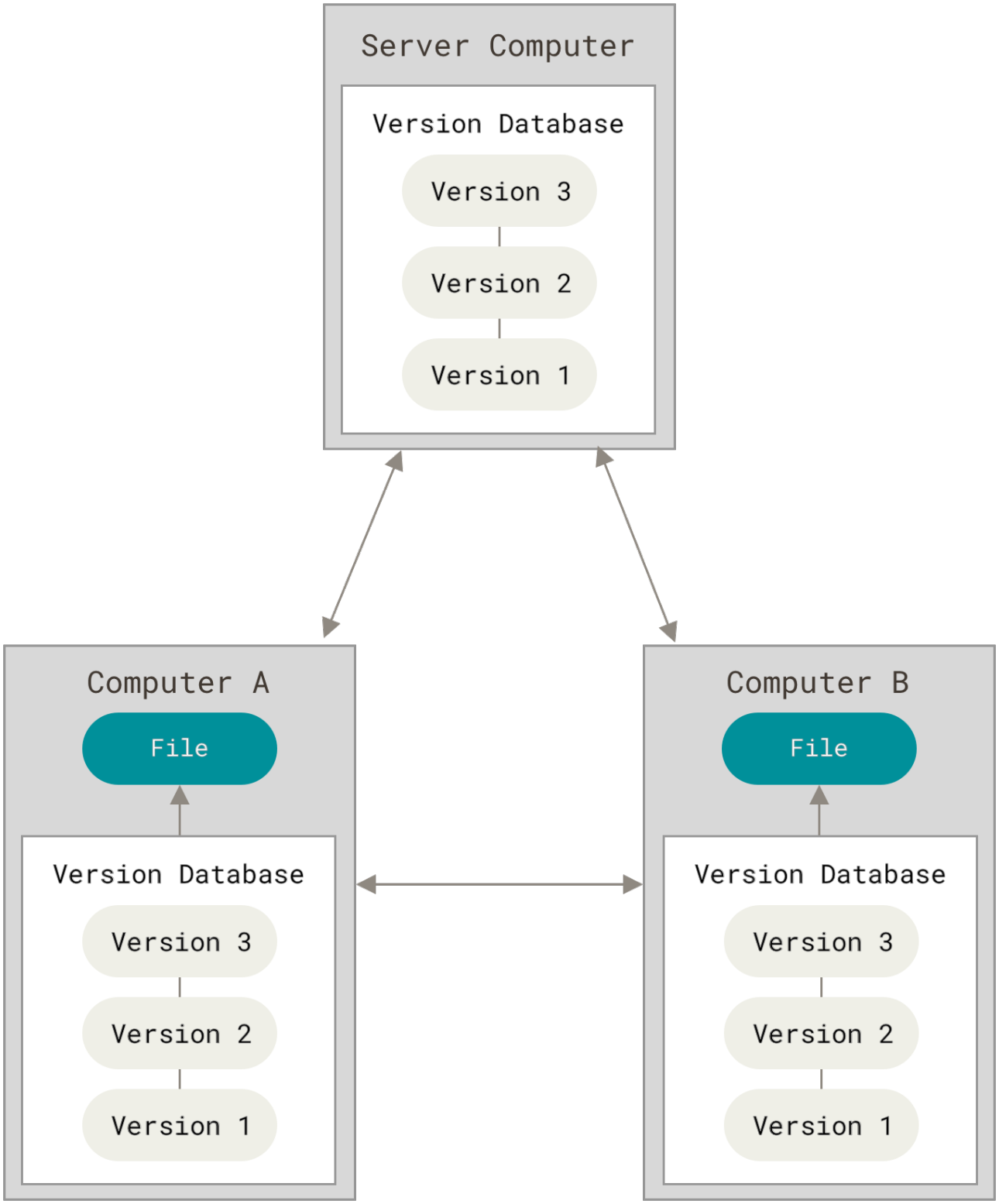


Рисунок 3. Распределённый контроль версий

Более того, многие DVCS 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 систему контроля версий 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. главу Ветвление в Git).

Что такое 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 системах контроля версий, таких как Subversion и Perforce. Это позволит вам избежать определённых проблем при использовании инструмента. 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 от любой другой системы контроля версий (включая Subversion и её собратьев) — это подход 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 хранимую информацию в виде набора файлов и изменений, сделанных в каждом файле, по времени (обычно это называют контролем версий, *основанным на различиях*).

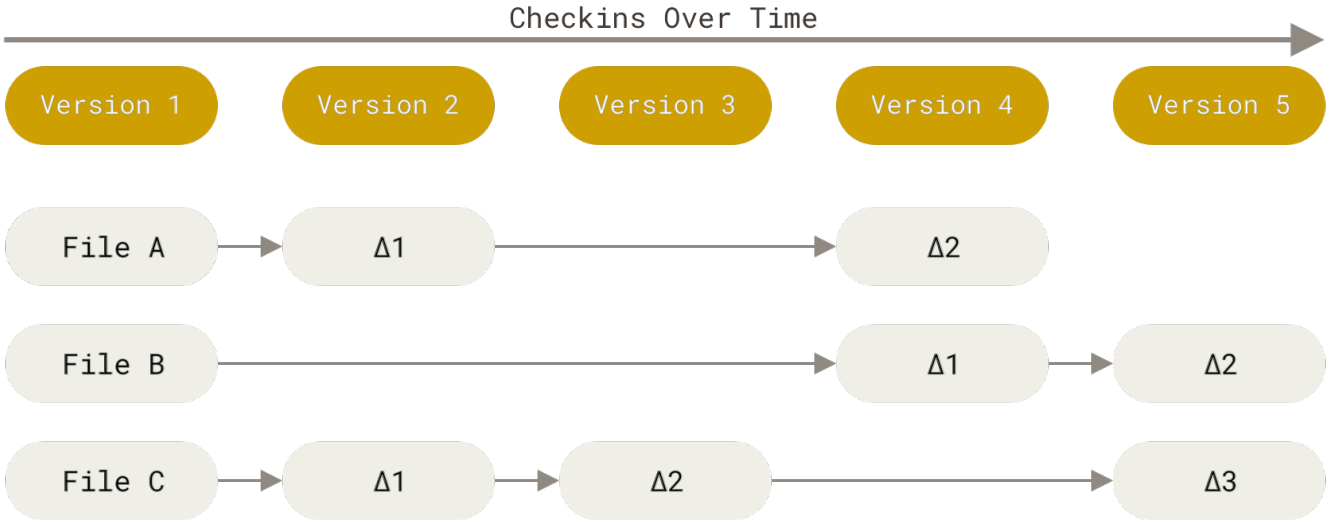


Рисунок 4. Хранение данных как набора изменений относительно первоначальной версии каждого из файлов

Git не хранит и не обрабатывает данные таким способом. Вместо этого, подход Git 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, 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**.

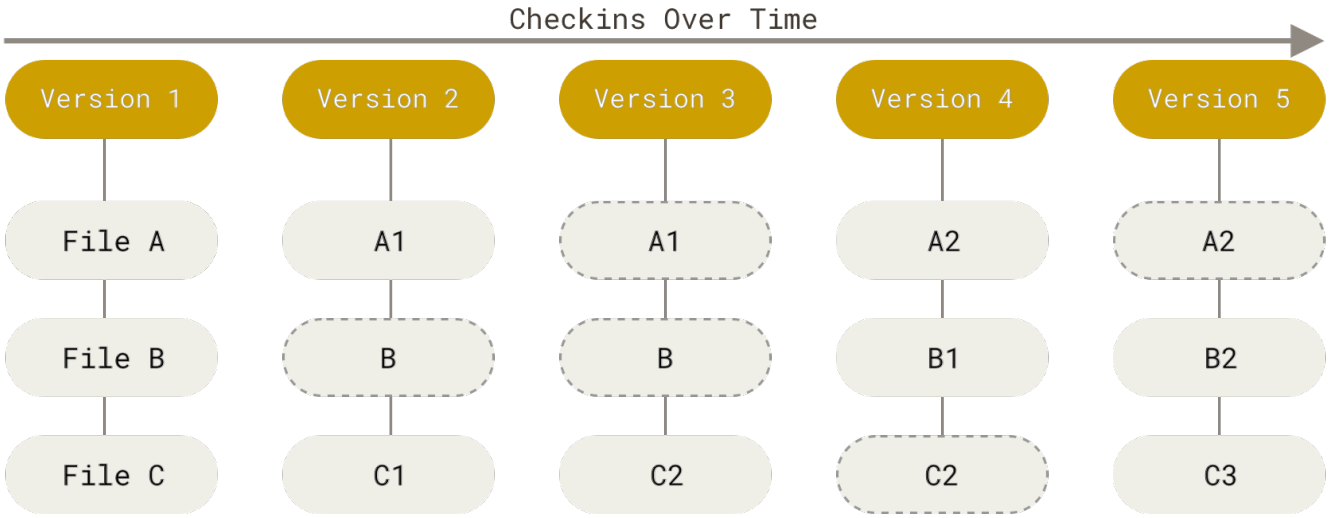


Рисунок 5. Хранение данных как снимков проекта во времени

Это очень важное различие между Git и почти любой другой системой контроля версий. 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 VCS. Когда мы будем рассматривать управление ветками в главе Ветвление в Git, мы увидим, какие преимущества вносит такой подход к работе с данными в Git.

Почти все операции выполняются локально

Для выполнения 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 централизованным системам контроля версий, где большинство операций страдают от задержек из-за работы с сетью, то этот аспект Git 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 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 просить 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 оффлайн или не имеете доступа к VPN в данный момент. Если вы в самолёте или в поезде и хотите немного поработать, вы сможете создавать коммиты без каких-либо проблем (в вашу *локальную* копию, помните?): когда будет возможность подключиться к сети, все изменения можно будет синхронизировать. Если вы ушли домой и не можете подключиться через VPN, 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 многого; 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 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 каталога 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 хеш SHA-1. Это строка длиной в 40 шестнадцатеричных символов (0-9 и a-f), которая вычисляется на основе содержимого файла или структуры каталога. Хеш SHA-1 выглядит примерно так:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Вы будете постоянно встречать хеши в Git, 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 обычно 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 системе контроля версий, вы можете потерять или испортить свои изменения, пока они не зафиксированы, но после того, как вы зафиксируете 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. У Git есть три основных состояния, в которых могут находиться ваши файлы: *изменён* (modified), *индексирован* (staged) и *зафиксирован* (committed):

- К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы.
- Индексированный — это изменённый файл в его текущей версии, отмеченный для включения в следующий коммит.
- Зафиксированный значит, что файл уже сохранён в вашей локальной базе.

Мы подошли к трём основным секциям проекта Git: рабочая копия (working tree), область индексирования (staging area) и каталог Git (Git directory).

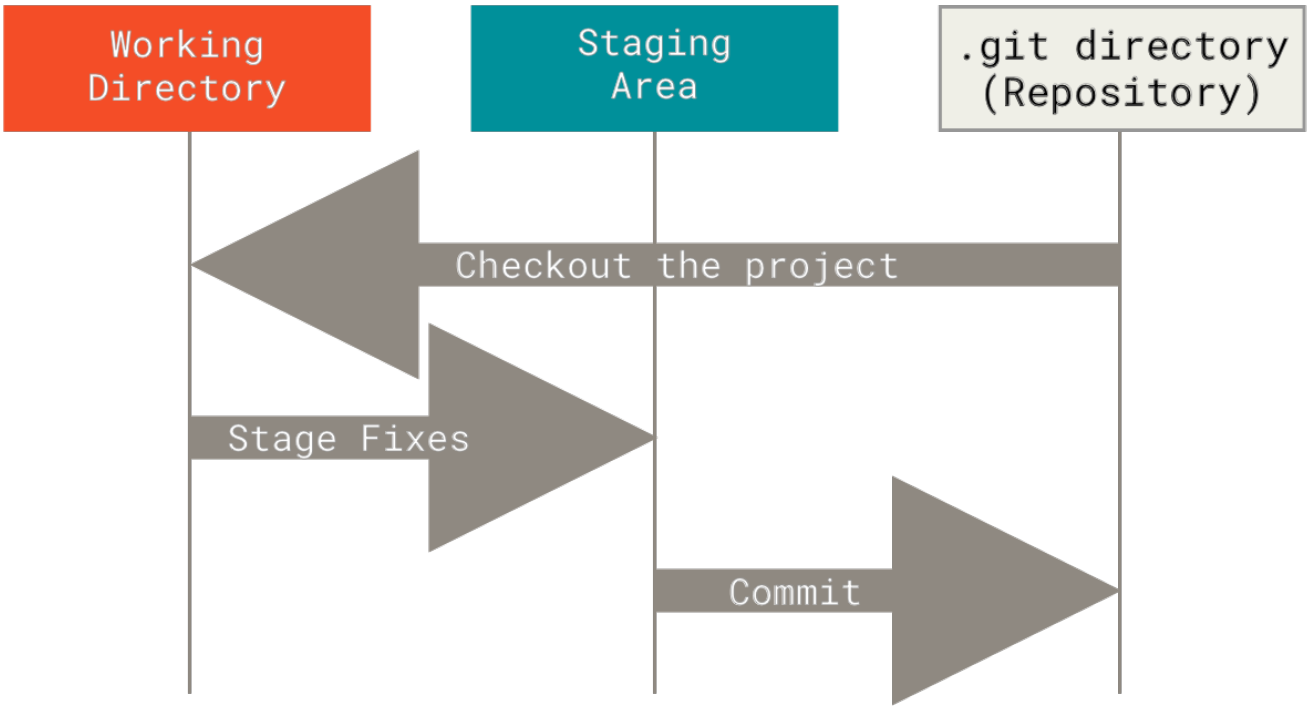


Рисунок 6. Рабочая копия, область индексирования и каталог Git

Рабочая копия является снимком одной версии проекта. Эти файлы извлекаются из сжатой базы данных в каталоге Git и помещаются на диск, для того чтобы их можно было использовать или редактировать.

Область индексирования — это файл, обычно находящийся в каталоге Git. В нём содержится информация о том, что попадёт в следующий коммит. Её техническое название на языке Git — «индекс», но фраза «область индексирования» также работает.

Каталог Git — 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.

Базовый подход в работе с Git выглядит так.

1. Вы изменяете файлы вашей рабочей копии.
2. Выборочно добавляете в индекс только те изменения, которые должны попасть в следующий коммит, добавляя тем самым снимки *только* этих изменений в индекс.
3. Когда вы делаете коммит, используются файлы из индекса как есть, и этот снимок сохраняется в каталог Git.

Если определённая версия файла есть в каталоге Git, эта версия считается *зафиксированной* (committed). Если файл был изменён и добавлен в индекс, значит, он *индексирован* (staged). И если файл был изменён с момента последнего распаковывания из репозитория, но не был добавлен в индекс, он считается *изменённым* (modified). В главе Основы Git вы узнаете больше об этих состояниях и о том, какую пользу вы можете извлечь из них, или как полностью пропустить этап индексирования.

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, командная строка — это единственное место, где вы можете запустить **все** команды Git, так как большинство клиентов с графическим интерфейсом реализуют для простоты только некоторую часть функциональности 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.

Поэтому мы предполагаем, что вы знаете, как открыть терминал в Mac или командную строку, или PowerShell в Windows. Если вам не понятно, о чем мы здесь говорим, то вам, возможно, придётся 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 своём 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 **2.8.0**. Хотя большинство используемых нами команд должны работать даже в старых версиях Git, некоторые из них могут не работать или действовать немного иначе, если вы используете старую версию. Поскольку Git отлично справляется с сохранением обратной совместимости, любая версия после 2.8 должна работать нормально.

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 (или другой похожий дистрибутив, такой как RHEL или CentOS), можно воспользоваться `dnf`:

```
$ sudo dnf install git-all
```

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

```
$ sudo apt install git
```

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

Установка на Mac

Существует несколько способов установки Git 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.

```
$ git --version
```

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 https://git-scm.com/download/mac.

Рисунок 7. OS X инсталлятор Git

Установка в Windows

Для установки Git в Windows также имеется несколько способов. Официальная сборка доступна для скачивания на официальном сайте Git. Просто перейдите на страницу https://git-scm.com/download/win, и загрузка запустится автоматически. Обратите внимание, что это отдельный проект, называемый Git для Windows; для получения дополнительной информации о нём перейдите на https://gitforwindows.org.

Для автоматической установки вы можете использовать пакет Git Chocolatey. Обратите внимание, что пакет Chocolatey поддерживается сообществом.

Установка из исходников

Многие предпочитают устанавливать Git из исходников, поскольку такой способ позволяет получить самую свежую версию. Обновление бинарных инсталляторов, как правило, немного отстаёт, 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: autotools, curl, zlib, openssl, expat и libiconv. Например, если в вашей системе используется `dnf` (Fedora) или `apt-get` (системы на базе Debian), вы можете использовать одну из следующих команд для установки всех зависимостей, используемых для сборки и установки бинарных файлов Git:

```
$ sudo dnf install dh-autoreconf curl-devel expat-devel gettext-devel \
  openssl-devel perl-devel zlib-devel
$ sudo apt-get install dh-autoreconf libcurl4-gnutls-dev libexpat1-dev \
  gettext libz-dev libssl-dev
```

Для того, чтобы собрать документацию в различных форматах (doc, html, info), понадобится установить дополнительные зависимости:

```
$ sudo dnf install asciidoc xmlto docbook2X
$ sudo apt-get install asciidoc xmlto docbook2x
```

Пользователи RHEL и производных от неё (таких как CentOS или Scientific Linux) должны подключить репозиторий EPEL для корректной установки пакета `docbook2X`

Если вы используете систему на базе Debian (Debian/Ubuntu/Ubuntu-производные), вам так же понадобится установить пакет `install-info`:

```
$ sudo apt-get install install-info
```

Если вы используете систему на базе RPM (Fedora/RHEL/RHEL-производные), вам так же понадобится установить пакет `getopt`, который уже установлен в системах на базе Debian:

```
$ sudo dnf install getopt
```

К тому же из-за различий имён бинарных файлов вам понадобится сделать следующее:

```
$ sudo ln -s /usr/bin/db2x_docbook2texi /usr/bin/docbook2x-texi
```

Когда все необходимые зависимости установлены, вы можете пойти 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-2.8.0.tar.gz
$ cd git-2.8.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

После этого вы можете получать обновления Git посредством самого 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 под себя. Это нужно сделать только один раз — при обновлении версии Git настройки сохранятся. Но, при необходимости, вы можете поменять их в любой момент, выполнив те же команды снова.

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

1. Файл `[path]/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. Так как этот файл является системным, то вам потребуются права суперпользователя для внесения изменений в него.
2. Файл `~/.gitconfig` или `~/.config/git/config` хранит настройки конкретного пользователя. Этот файл используется при указании параметра `--global` и применяется ко *всем* репозиториям, с которыми вы работаете в текущей системе.
3. Файл `config` в каталоге Git (т. е. `.git/config`) репозитория, который вы используете в данный момент, хранит настройки конкретного репозитория. Вы можете заставить Git читать и писать в этот файл с помощью параметра `--local`, но на самом деле это значение по умолчанию. Неудивительно, что вам нужно находиться где-то в репозитории Git, чтобы эта опция работала правильно.

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

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

Если вы используете Git для Windows версии 2.x или новее, то так же обрабатывается файл конфигурации уровня системы, который имеет путь `C:\Documents and Settings\All Users\Application Data\Git\config` в Windows XP или `C:\ProgramData\Git\config` в Windows Vista и новее. Этот файл может быть изменён только командой `git config -f <file>`, запущенной с правами администратора.

Чтобы посмотреть все установленные настройки и узнать где именно они заданы, используйте команду:

```
$ git config --list --show-origin
```

Имя пользователя

Первое, что вам следует сделать после установки Git — указать ваше имя и адрес электронной почты. Это важно, потому что каждый коммит в Git 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. 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
```

В системе Windows следует указывать полный путь к исполняемому файлу при установке другого текстового редактора по умолчанию. Пути могут отличаться в зависимости от того, как работает инсталлятор.

В случае с Notepad++, популярным редактором, скорее всего вы захотите установить 32-битную версию, так как 64-битная версия ещё не поддерживает все плагины. Если у вас 32-битная Windows или 64-битный редактор с 64-битной системой, то выполните следующее:

```
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe'
-multiInst -notabbar -nosession -noPlugin"
```

Vim, Emacs и Notepad++ — популярные текстовые редакторы, которые часто используются разработчиками как в Unix-подобных системах, таких как Linux и Mac, так и в Windows. Если вы используете другой редактор или его 32-битную версию, то обратитесь к разделу Команды git config core.editor за

дополнительными инструкциями как использовать его совместно с Git.

В случае, если вы не установили свой редактор и не знакомы с Vim или Emacs, вы можете попасть в затруднительное положение, когда какой-либо из них будет запущен. Например, в Windows может произойти преждевременное прерывание команды Git при попытке вызова редактора.

Настройка ветки по умолчанию

Когда вы инициализируете репозиторий командой `git init`, Git создаёт ветку с именем *master* по умолчанию. Начиная с версии 2.28, вы можете задать другое имя для создания ветки по умолчанию.

Например, чтобы установить имя *main* для вашей ветки по умолчанию, выполните следующую команду:

```
$ git config --global init.defaultBranch main
```

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 отображаться несколько раз, потому что Git читает настройки из разных файлов (например, из `/etc/gitconfig` и `~/.gitconfig`). В таком 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
```

Так как Git читает значение настроек из нескольких файлов, возможна ситуация когда Git использует не то значение, которое вы ожидали. В таком случае вы можете спросить Git об *origin* этого значения. Git выведет имя файла, из которого значение для настройки было взято последним:

```
$ git config --show-origin rerere.autoUpdate
file:/home/johndoe/.gitconfig   false
```

Как получить помощь?

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

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

Например, так можно открыть руководство по команде `git config`

```
$ git help config
```

Эти команды хороши тем, что ими можно пользоваться всегда, даже без подключения к сети. Если руководства и этой книги недостаточно и вам нужна персональная помощь, вы можете попытаться поискать её на каналах `#git` и `#github` IRC сервера Libera Chat, который доступен по адресу https://libera.chat/. Обычно там сотни людей, отлично знающих Git, которые могут помочь.

Так же, если вам нужно посмотреть только список опций и вы не хотите читать полную документацию по команде, вы можете использовать опцию `-h` для вывода краткой инструкции по использованию:

```
$ git add -h
usage: git add [<options>] [--] <pathspec>...

    -n, --dry-run         dry run
    -v, --verbose         be verbose

    -i, --interactive     interactive picking
    -p, --patch           select hunks interactively
    -e, --edit            edit current diff and apply
    -f, --force           allow adding otherwise ignored files
    -u, --update          update tracked files
    --renormalize         renormalize EOL of tracked files (implies -u)
    -N, --intent-to-add   record only the fact that the path will be added later
    -A, --all             add changes from all tracked and untracked files
    --ignore-removal      ignore paths removed in the working tree (same as --no-all)
    --refresh             don't add, only refresh the index
    --ignore-errors       just skip files which cannot be added because of errors
    --ignore-missing      check if - even missing - files are ignored in dry run
    --chmod (+|-)x        override the executable bit of the listed files
```

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

Если вы хотите начать работать с Git, 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. 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 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

Обычно вы получаете репозиторий Git одним из двух способов:

1. Вы можете взять локальный каталог, который в настоящее время не находится под версионным контролем, и превратить его в репозиторий Git, либо
2. Вы можете *клонировать* существующий репозиторий Git из любого места.

В обоих случаях вы получите готовый к работе Git репозиторий на вашем компьютере.

Создание репозитория в существующем каталоге

Если у вас уже есть проект в каталоге, который не находится под версионным контролем Git, то для начала нужно перейти в него. Если вы не делали этого раньше, то для разных операционных систем это выглядит по-разному:

для Linux:

```
$ cd /home/user/my_project
```

для macOS:

```
$ cd /Users/user/my_project
```

для Windows:

```
$ cd C:/Users/user/my_project
```

а затем выполните команду:

```
$ git init
```

Эта команда создаёт в текущем каталоге новый подкаталог с именем `.git`, содержащий все необходимые файлы репозитория — структуру Git репозитория. На этом этапе ваш проект ещё не находится под версионным контролем. Подробное описание файлов, содержащихся в только что созданном вами каталоге `.git`, приведено в главе Git изнутри

Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит добавить их в индекс 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 разберём, 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 забирается (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 хуков (server-side hooks) и т. п., но все данные, помещённые под версионный контроль, будут сохранены, подробнее об этом смотрите в разделе Установка Git на сервер главы 4).

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 каталог `libgit2`, инициализирует в нём подкаталог `.git`, скачивает все данные для этого репозитория и извлекает рабочую копию последней версии. Если вы перейдёте в только что созданный каталог `libgit2`, то увидите в нём файлы проекта, готовые для работы или использования. Для того, чтобы клонировать репозиторий в каталог 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 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 разделе Установка Git на сервер главы 4 мы познакомимся со всеми доступными вариантами конфигурации сервера для обеспечения доступа к вашему Git репозиторию, а также рассмотрим их достоинства и недостатки.

Запись изменений в репозиторий

Итак, у вас имеется настоящий Git-репозиторий и рабочая копия файлов для некоторого проекта. Вам нужно делать некоторые изменения и фиксировать «снимки» состояния (*snapshots*) этих изменений в вашем репозитории каждый раз, когда проект достигает состояния, которое вам хотелось бы сохранить.

Запомните, каждый файл в вашем рабочем каталоге может находиться в одном из двух состояний: под контролем версий (отслеживаемые) и нет (неотслеживаемые). Отслеживаемые файлы — это те файлы, которые были в последнем снимке состояния проекта; они могут быть неизменёнными, изменёнными или подготовленными к коммиту. Если кратко, то отслеживаемые файлы — это те файлы, о которых знает Git.

Неотслеживаемые файлы — это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний снимок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что Git только что их извлёк и вы ничего пока не редактировали.

Как только вы отредактируете файлы, Git будет рассматривать их как изменённые, так как вы изменили их с момента последнего коммита. Вы индексируете эти изменения, затем фиксируете все проиндексированные изменения, а затем цикл повторяется.

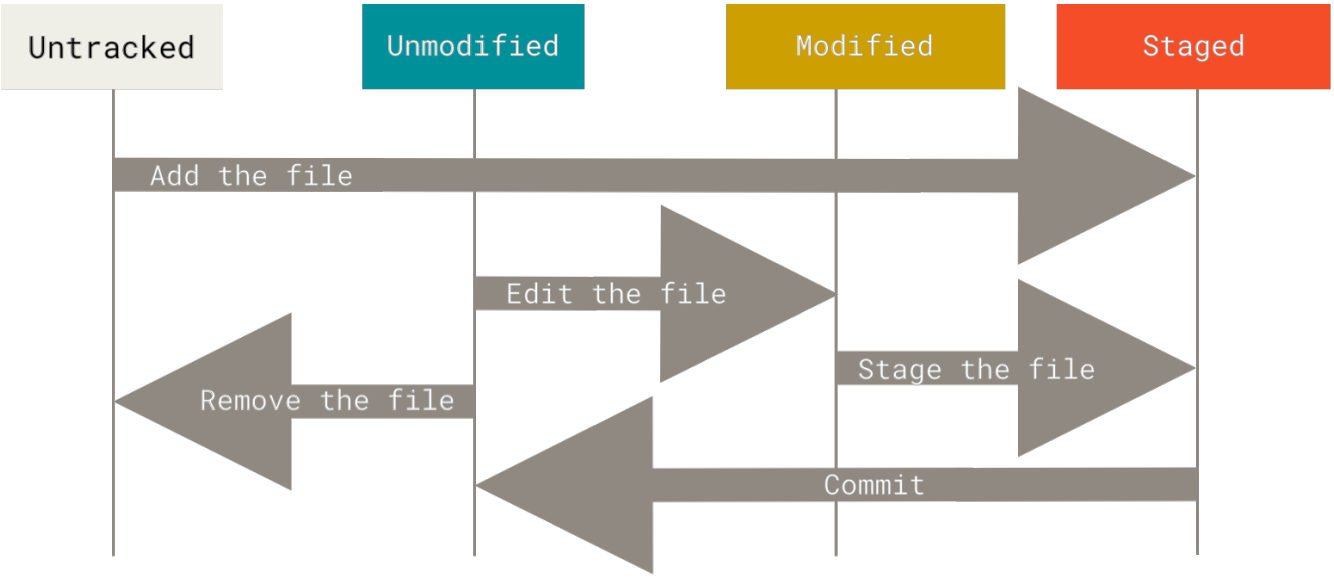


Рисунок 8. Жизненный цикл состояний файлов

Определение состояния файлов

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся, — это команда `git status`. Если вы выполните эту команду сразу после клонирования, вы увидите что-то вроде этого:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

Это означает, что у вас чистый рабочий каталог, другими словами — в нём нет отслеживаемых изменённых файлов. Git также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь. Наконец, команда сообщает вам, на какой ветке вы находитесь и что она не расходится с веткой на сервере. Пока что это всегда ветка `master`, ветка по умолчанию; в этой главе это не важно. В главе Ветвление в Git ветки и ссылки будут рассмотрены более детально.

В 2020 году GitHub изменил имя ветки по умолчанию с `master` на `main`, другие же Git-хостинги последовали этому примеру. Поэтому вы можете обнаружить, что ветка по умолчанию для новых репозиториев — `main`, а не `master`. Более того, имя ветки по умолчанию можно изменить (как вы видели в Настройка ветки по умолчанию), поэтому вам может встретиться и другое имя. При этом Git продолжает использовать имя `master`, поэтому далее в книге мы используем именно его.

Предположим, вы добавили в свой проект новый файл, простой файл `README`. Если этого файла раньше не было и вы выполните `git status`, то увидите свой неотслеживаемый файл:

```
$ echo 'My Project' > README
$ 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)

    README

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

Понять, что новый файл `README` неотслеживаемый, можно по тому, что он находится в секции «Untracked files» в выводе команды `status`. Статус `Untracked` означает, что Git видит файл, которого не было в предыдущем снимке состояния (коммите); Git не станет добавлять его в ваши коммиты, пока вы его явно об этом не попросите. Это предохранит вас от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые вы и не думали добавлять. Мы хотели добавить README, так давайте сделаем это.

Отслеживание новых файлов

Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда `git add`. Чтобы начать отслеживание файла `README`, вы можете выполнить следующее:

```
$ git add README
```

Если вы снова выполните команду `status`, то увидите, что файл `README` теперь отслеживаемый и добавлен в индекс:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)

    new file:   README
```

Вы можете видеть, что файл проиндексирован, так как он находится в секции «Changes to be committed». Если вы выполните коммит в этот момент, то версия файла, существовавшая на момент выполнения вами команды `git add`, будет добавлена в историю снимков состояния. Как вы помните, когда вы ранее выполнили `git init`, затем вы выполнили `git add (файлы)` — это было сделано для того, чтобы добавить файлы в каталоге в систему контроля версий. Команда `git add` принимает в качестве параметра путь к файлу или каталогу; если это каталог, команда рекурсивно добавляет все файлы из указанного каталога в индекс.

Индексация изменённых файлов

Давайте модифицируем файл, уже находящийся в системе контроля версий. Если вы измените отслеживаемый файл `CONTRIBUTING.md` и после этого снова выполните команду `git status`, то результат будет примерно следующим:

```
$ 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:   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
```

Файл `CONTRIBUTING.md` находится в секции «Changes not staged for commit» — это означает, что отслеживаемый файл был изменён в рабочем каталоге, но пока не проиндексирован. Чтобы проиндексировать его, необходимо выполнить команду `git add`. Это многофункциональная команда, она используется для добавления под версионный контроль новых файлов, для индексации изменений, а также для других целей, например для указания файлов с исправленным конфликтом слияния. Вам может быть понятнее, если вы будете думать о смысле `git add` как «добавить этот файл в следующий коммит», а не как «добавить этот файл в проект». Выполним `git add`, чтобы проиндексировать `CONTRIBUTING.md`, а затем снова выполним `git status`:

```
$ git add CONTRIBUTING.md
$ 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:   README
    modified:   CONTRIBUTING.md
```

Теперь оба файла проиндексированы и войдут в следующий коммит. В этот момент вы, предположим, вспомнили одно небольшое изменение, которое вы хотите сделать в `CONTRIBUTING.md` до коммита. Вы открываете файл, вносите и сохраняете необходимые изменения и вроде бы готовы к коммиту. Но давайте-ка ещё раз выполним `git status`:

```
$ vim CONTRIBUTING.md
$ 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:   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
```

Что за чёрт? Теперь `CONTRIBUTING.md` отображается как проиндексированный и непроиндексированный одновременно. Как такое возможно? Такая ситуация наглядно демонстрирует, что Git индексирует файл в точности в том состоянии, в котором он находился, когда вы выполнили команду `git add`. Если вы выполните коммит сейчас, то файл `CONTRIBUTING.md` попадёт в коммит в том состоянии, в котором он находился, когда вы последний раз выполняли команду `git add`, а не в том, в котором он находится в вашем

рабочем каталоге в момент выполнения `git commit`. Если вы изменили файл после выполнения `git add`, вам придётся снова выполнить `git add`, чтобы проиндексировать последнюю версию файла:

```
$ git add CONTRIBUTING.md
$ 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:   README
    modified:   CONTRIBUTING.md
```

Сокращённый вывод статуса

Вывод команды `git status` довольно всеобъемлющий и многословный. Git также имеет флаг вывода сокращённого статуса, так что вы можете увидеть изменения в более компактном виде. Если вы выполните `git status -s` или `git status --short` вы получите гораздо более упрощённый вывод:

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

Новые неотслеживаемые файлы помечены `??` слева от них, файлы добавленные в отслеживаемые помечены `A`, отредактированные файлы помечены `M` и так далее. В выводе содержится два столбца: в левом указывается статус файла, а в правом — имя файла. К примеру, в нашем выводе файл `README` модифицирован в рабочем каталоге, но не проиндексирован, а файл `lib/simplegit.rb` модифицирован и проиндексирован. Файл `Rakefile` модифицирован, проиндексирован и ещё раз модифицирован — таким образом, на данный момент у него есть те изменения, которые попадут в коммит, и те, которые не попадут.

Игнорирование файлов

Зачастую у вас имеется группа файлов, которые вы не только не хотите автоматически добавлять в репозиторий, но и видеть в списках неотслеживаемых. К таким файлам обычно относятся автоматически генерируемые файлы (различные логи, результаты сборки программ и т. п.). В таком случае, вы можете создать файл `.gitignore`. с перечислением шаблонов соответствующих таким файлам. Вот пример файла `.gitignore`:

```
$ cat .gitignore
```

```
*.[oa]
*~
```

Первая строка предписывает Git игнорировать любые файлы заканчивающиеся на «`.o`» или «`.a`» — объектные и архивные файлы, которые могут появиться во время сборки кода. Вторая строка предписывает игнорировать все файлы заканчивающиеся на тильду (`~`), которая используется во многих текстовых редакторах, например Emacs, для обозначения временных файлов. Вы можете также включить каталоги `log`, `tmp` или `pid`; автоматически создаваемую документацию; и т. д. и т. п. Хорошая практика заключается в настройке файла `.gitignore` до того, как начать серьёзно работать, это защитит вас от случайного добавления в репозиторий файлов, которых вы там видеть не хотите.

К шаблонам в файле `.gitignore` применяются следующие правила:

- Пустые строки, а также строки, начинающиеся с `#`, игнорируются.
- Стандартные шаблоны являются глобальными и применяются рекурсивно для всего дерева каталогов.
- Чтобы избежать рекурсии, используйте слеш (`/`) в начале шаблона.
- Чтобы исключить каталог, добавьте слеш (`/`) в конец шаблона.
- Можно инвертировать шаблон, использовав восклицательный знак (`!`) в качестве первого символа.

Glob-шаблоны представляют собой упрощённые регулярные выражения, используемые командными интерпретаторами. Символ (`*`) соответствует 0 или более символам; последовательность `[abc]` — любому символу из указанных в скобках (в данном примере `a`, `b` или `c`); знак вопроса (`?`) соответствует одному символу; и квадратные скобки, в которые заключены символы, разделённые дефисом (`[0-9]`), соответствуют любому символу из интервала (в данном случае от `0` до `9`). Вы также можете использовать две звёздочки, чтобы указать на вложенные каталоги: `a/**/z` соответствует `a/z`, `a/b/z`, `a/b/c/z`, и так далее.

Вот ещё один пример файла `.gitignore`:

```
# Исключить все файлы с расширением .a
*.a

# Но отслеживать файл lib.a даже если он подпадает под исключение выше
!lib.a

# Исключить файл TODO в корневом каталоге, но не файл в subdir/TODO
/TODO

# Игнорировать все файлы в каталоге build/
build/

# Игнорировать файл doc/notes.txt, но не файл doc/server/arch.txt
doc/*.txt
```

```
# Игнорировать все .txt файлы в каталоге doc/
doc/**/*.txt
```

GitHub поддерживает довольно полный список примеров `.gitignore` файлов для множества проектов и языков https://github.com/github/gitignore это может стать отправной точкой для `.gitignore` в вашем проекте.

В простейшем случае репозиторий будет иметь один файл `.gitignore` в корневом каталоге, правила из которого будут рекурсивно применяться ко всем подкаталогам. Также возможно использовать `.gitignore` файлы в подкаталогах. Правила из этих файлов будут применяться только к каталогам, в которых они находятся. Например, репозиторий исходного кода ядра Linux содержит 206 файлов `.gitignore`.

Детальное рассмотрение использования нескольких `.gitignore` файлов выходит за пределы этой книги; детали доступны в справке `man gitignore`.

Просмотр индексированных и неиндексированных изменений

Если результат работы команды `git status` недостаточно информативен для вас — вам хочется знать, что конкретно поменялось, а не только какие файлы были изменены, — вы можете использовать команду `git diff`. Позже мы рассмотрим команду `git diff` подробнее; скорее всего, вы будете использовать эту команду для получения ответов на два вопроса: что вы изменили, но ещё не проиндексировали; и что вы проиндексировали и собираетесь включить в коммит. Если `git status` отвечает на эти вопросы в самом общем виде, перечисляя имена файлов, `git diff` показывает вам непосредственно добавленные и удалённые строки — патч как он есть.

Допустим, вы снова изменили и проиндексировали файл `README`, а затем изменили файл `CONTRIBUTING.md` без индексирования. Если вы выполните команду `git status`, вы опять увидите что-то вроде

```
$ 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)

    modified:   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
```

Чтобы увидеть, что же вы изменили, но пока не проиндексировали, наберите `git diff` без

аргументов:

```
$ 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
```

Эта команда сравнивает содержимое вашего рабочего каталога с содержимым индекса. Результат показывает ещё не проиндексированные изменения.

Если вы хотите посмотреть, что вы проиндексировали и что войдёт в следующий коммит, вы можете выполнить `git diff --staged`. Эта команда сравнивает ваши проиндексированные изменения с последним коммитом:

```
$ 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
```

Важно отметить, что `git diff` сама по себе не показывает все изменения, сделанные с последнего коммита — только те, что ещё не проиндексированы. Такое поведение может сбивать с толку, так как если вы проиндексируете все свои изменения, то `git diff` ничего не вернёт.

Другой пример: вы проиндексировали файл `CONTRIBUTING.md` и затем изменили его, вы можете использовать `git diff` для просмотра как проиндексированных изменений в этом файле, так и тех, что пока не проиндексированы. Пусть окружение выглядит вот так:

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ 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)

    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
```

Используем `git diff` для просмотра непроиндексированных изменений:

```
$ 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
```

а также `git diff --cached` для просмотра проиндексированных изменений (`--staged` и `--cached` — это синонимы):

```
$ 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 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
```

Git Diff во внешних инструментах

Мы будем продолжать использовать команду `git diff` различными способами на протяжении всей книги. Существует ещё один способ

просматривать эти изменения, если вы предпочитаете графический вывод или внешнюю программу просмотра различий вместо консоли. Выполнив команду `git difftool` вместо `git diff`, вы сможете просмотреть изменения в файле с помощью таких программ как emerge, vimdiff и других (включая коммерческие продукты). Выполните `git difftool --tool-help`, чтобы увидеть какие из них уже установлены в вашей системе.

Коммит изменений

Теперь, когда ваш индекс находится в таком состоянии, как вам и хотелось, вы можете зафиксировать свои изменения. Запомните: всё, что до сих пор не проиндексировано — любые файлы, созданные или изменённые вами, и для которых вы не выполнили `git add` после редактирования, — не войдут в этот коммит. Они останутся изменёнными файлами на вашем диске. В нашем случае, когда вы в последний раз выполняли `git status`, вы видели что всё проиндексировано — и вот, вы готовы к коммиту. Простейший способ зафиксировать изменения — это набрать `git commit`:

```
$ git commit
```

Эта команда откроет выбранный вами текстовый редактор.

Редактор устанавливается переменной окружения `EDITOR` — обычно это Vim или Emacs, хотя вы можете установить любой другой с помощью команды `git config --global core.editor`, как было показано в главе Введение).

В редакторе будет отображён следующий текст (это пример окна Vim):

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

Вы можете видеть, что комментарий по умолчанию для коммита содержит закомментированный результат работы команды `git status` и ещё одну пустую строку сверху. Вы можете удалить эти комментарии и набрать своё сообщение или же оставить их для напоминания о том, что вы фиксируете.

Для ещё более подробного напоминания, что же именно вы поменяли, можете передать аргумент `-v` в команду `git commit`. Это приведёт к тому, что в комментарий будет также помещена дельта / diff изменений. Таким образом, вы сможете точно увидеть все изменения которые вы совершили.

Когда вы выходите из редактора, Git создаёт для вас коммит с этим сообщением, удаляя комментарии и вывод команды `diff`.

Есть и другой способ — вы можете набрать свой комментарий к коммиту в командной строке вместе с командой `commit`, указав его после параметра `-m`, как в следующем примере:

```
$ 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
```

Итак, вы создали свой первый коммит! Вы можете видеть, что коммит вывел вам немного информации о себе: на какую ветку вы выполнили коммит (`master`), какая контрольная сумма SHA-1 у этого коммита (`463dc4f`), сколько файлов было изменено, а также статистику по добавленным / удалённым строкам в этом коммите.

Запомните, что коммит сохраняет снимок состояния *проиндексированных* файлов. Всё, что вы не проиндексировали, так и висит в рабочем каталоге как изменённое; вы можете сделать ещё один коммит, чтобы добавить эти изменения в репозиторий. Каждый раз, когда вы делаете коммит, вы сохраняете снимок состояния вашего проекта, который позже вы можете восстановить или с которым можно сравнить текущее состояние.

Игнорирование индексации

Несмотря на то, что индекс может быть удивительно полезным для создания коммитов именно такими, как вам и хотелось, он временами несколько сложнее, чем вам нужно в процессе работы. Если у вас есть желание пропустить этап индексирования, Git предоставляет простой способ. Добавление параметра `-a` в команду `git commit` заставляет Git автоматически индексировать каждый уже отслеживаемый на момент коммита файл, позволяя вам обойтись без `git add`:

```
$ 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:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'Add new benchmarks'
[master 83e38c7] Add new benchmarks
```

```
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Обратите внимание, что в данном случае перед коммитом вам не нужно выполнять `git add` для файла `CONTRIBUTING.md`, потому что флаг `-a` включает все файлы. Это удобно, но будьте осторожны: флаг `-a` может включить в коммит нежелательные изменения.

Удаление файлов

Для того чтобы удалить файл из Git, вам необходимо удалить его из отслеживаемых файлов (точнее, удалить его из вашего индекса) а затем выполнить коммит. Это позволяет сделать команда `git rm`, которая также удаляет файл из вашего рабочего каталога, так что в следующий раз вы не увидите его как «неотслеживаемый».

Если вы просто удалите файл из своего рабочего каталога, он будет показан в секции «Changes not staged for commit» (изменённые, но не проиндексированные) вывода команды `git status`:

```
$ 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")
```

Затем, если вы выполните команду `git rm`, удаление файла попадёт в индекс:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ 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)

    deleted:    PROJECTS.md
```

После следующего коммита файл исчезнет и больше не будет отслеживаться. Если вы изменили файл и уже проиндексировали его, вы должны использовать принудительное удаление с помощью параметра `-f`. Это сделано для повышения безопасности, чтобы предотвратить ошибочное удаление данных, которые ещё не были записаны в снимок состояния и которые нельзя восстановить из Git.

Другая полезная штука, которую вы можете захотеть сделать, — это удалить файл из индекса, оставив его при этом в рабочем каталоге. Другими словами, вы можете захотеть оставить файл на жёстком диске, но перестать отслеживать изменения в нём. Это особенно полезно, если вы забыли добавить что-то в файл `.gitignore` и по ошибке проиндексировали, например, большой файл с логами, или кучу промежуточных файлов компиляции. Чтобы сделать это, используйте опцию `--cached`:

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

В команду `git rm` можно передавать файлы, каталоги или шаблоны. Это означает, что вы можете сделать что-то вроде

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

Обратите внимание на обратный слеш (`\`) перед `*`. Он необходим из-за того, что Git использует свой собственный обработчик имён файлов вдобавок к обработчику вашего командного интерпретатора. Эта команда удаляет все файлы, имеющие расширение `.log` и находящиеся в каталоге `log/`. Или же вы можете сделать вот так:

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

Эта команда удаляет все файлы, имена которых заканчиваются на `~`.

Перемещение файлов

В отличие от многих других систем контроля версий, Git не отслеживает перемещение файлов явно. Когда вы переименовываете файл в Git, в нём не сохраняется никаких метаданных, говорящих о том, что файл был переименован. Однако, Git довольно умён в плане обнаружения перемещений постфактум — мы рассмотрим обнаружение перемещения файлов чуть позже.

Таким образом, наличие в Git команды `mv` выглядит несколько странным. Если вам хочется переименовать файл в Git, вы можете сделать что-то вроде

```
$ git mv file_from file_to
```

и это отлично сработает. На самом деле, если вы выполните что-то вроде этого и посмотрите на вывод команды `git status`, вы увидите, что Git считает, что произошло переименование файла:

```
$ git mv README.md README
$ 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)

    renamed:    README.md -> README
```

Однако это эквивалентно выполнению следующих команд:

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

Git неявно определяет, что произошло переименование, поэтому неважно, переименуете вы файл так или используя команду `mv`. Единственное отличие состоит лишь в том, что `mv` — это функция для удобства: одна команда вместо трёх. Важнее другое — вы можете использовать любой удобный способ для переименования файла, а затем воспользоваться командами `add` или `rm` перед коммитом.

Просмотр истории коммитов

После того, как вы создали несколько коммитов или же клонировали 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». Чтобы клонировать проект, используйте команду:

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

Если вы запустите команду `git log` в каталоге клонированного 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

    Change version number

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

    Remove unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
```

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

    Initial commit
```

По умолчанию (без аргументов) `git log` перечисляет коммиты, сделанные в репозитории в обратном к хронологическому порядке — последние коммиты находятся вверху. Из примера можно увидеть, что данная команда перечисляет коммиты с их SHA-1 контрольными 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. Рассмотрим наиболее популярные из них.

Одним из самых полезных аргументов является `-p` или `--patch`, который показывает разницу (выводит *патч*), внесённую в каждый коммит. Так же вы можете ограничить количество записей в выводе команды; используйте параметр `-2` для вывода только двух записей:

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

    Change 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

    Remove 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
```

Эта опция отображает аналогичную информацию но содержит разницу для каждой записи. Очень удобно использовать данную опцию для код ревью или для быстрого просмотра серии внесённых изменений. Так же есть возможность использовать серию опций для обобщения. Например, если вы хотите увидеть сокращённую 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

    Change 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

    Remove 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

    Initial 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 изменённых 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. Опция `oneline` выводит 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 Change version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 Remove unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 Initial commit
```

Наиболее интересной опцией является `format`, которая позволяет указать 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 изменён даже после обновления Git:

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

Полезные опции для `git log --pretty=format` отображает наиболее полезные опции для изменения формата.

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

Опция	Описания вывода
`%H`	Хеш коммита
`%h`	Сокращённый хеш коммита
`%T`	Хеш дерева
`%t`	Сокращённый хеш дерева
`%P`	Хеш родителей
`%p`	Сокращённый хеш родителей
`%an`	Имя автора
`%ae`	Электронная почта автора
`%ad`	Дата автора (формат даты можно задать опцией --date=option)
`%ar`	Относительная дата автора
`%cn`	Имя коммитера
`%ce`	Электронная почта коммитера
`%cd`	Дата коммитера
`%cr`	Относительная дата коммитера

Опция	Описания вывода
`%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. Более детально мы рассмотрим разницу в главе Распределённый Git.

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 Add 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 ветвление и слияние.

Мы рассмотрели только несколько простых опций для форматирования вывода с помощью команды `git log` — на самом деле их гораздо больше. Наиболее распространённые опции для команды `git log` содержит описание как уже рассмотренных, так и нескольких новых опций, которые могут быть полезными в зависимости от нужного формата вывода.

Таблица 2. Наиболее распространённые опции для команды `git log`

Опция	Описание
`-p`	Показывает патч для каждого коммита.
`--stat`	Показывает статистику изменённых файлов для каждого коммита.
`--shortstat`	Отображает только строку с количеством изменений/вставок/удалений для команды --stat.
`--name-only`	Показывает список изменённых файлов после информации о коммите.
`--name-status`	Показывает список файлов, которые добавлены/изменены/удалены.
`--abbrev-commit`	Показывает только несколько символов SHA-1 чек-суммы вместо всех 40.

Опция	Описание
`--relative-date`	Отображает дату в относительном формате (например, «2 weeks ago») вместо стандартного формата даты.
`--graph`	Отображает ASCII граф с ветвлениями и историей слияний.
`--pretty`	Показывает коммиты в альтернативном формате. Возможные варианты опций: oneline, short, full, fuller и format (с помощью последней можно указать свой формат).
`--oneline`	Сокращение для одновременного использования опций `--pretty=oneline --abbrev-commit`.

Ограничение вывода

В дополнение к опциям форматирования вывода, команда `git log` принимает несколько опций для ограничения вывода — опций, с помощью которых можно увидеть определённое 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 и представляет собой `n` последних коммитов. На практике вы не будете часто использовать эту опцию, потому что Git по умолчанию использует постраничный вывод и вы будете видеть только одну страницу за раз.

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 определённую дату вида `2008-01-15` или же относительную дату, например `2 years 1 day 3 minutes ago`.

Также вы можете фильтровать список коммитов по заданным параметрам. Опция `--author` даёт возможность фильтровать по автору коммита, а опция `--grep` искать по ключевым словам в сообщении коммита.

> Допускается указывать несколько параметров `--author` и `--grep` для поиска, которые позволят найти коммиты, соответствующие *любому* указанному `--author` и *любому* указанному `--grep` шаблону; однако, применение опции `--all-match` заставит искать коммиты соответствующие *всем* указанным `--grep` шаблонам.

Следующим действительно полезным фильтром является опция `-S`, которая принимает аргумент в виде строки 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 определённой функции, вы можете запустить команду:

```
$ git log -S function_name
```

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

```
$ git log -- path/to/file
```

В таблице Опции для ограничения вывода команды `git log` вы можете увидеть эти и другие распространённые опции.

Таблица 3. Опции для ограничения вывода команды `git log`

Опция	Описание
`-(n)`	Показывает только последние n коммитов.
`--since`, `--after`	Показывает только те коммиты, которые были сделаны после указанной даты.
`--until`, `--before`	Показывает только те коммиты, которые были сделаны до указанной даты.
`--author`	Показывает только те коммиты, в которых запись author совпадает с указанной строкой.
`--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 исходном коде Git в октябре 2008 года, автором которых был Junio Hamano и которые не были коммитами слияния, вы можете запустить следующую команду:

```
$ git log --pretty="%h - %s" --author='Junio C Hamano' --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
branch
```

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

Предотвращение отображения коммитов слияния

В зависимости от используемого порядка работы, история коммитов в вашем репозитории может содержать большое количество коммитов слияния, которые сами по себе не очень информативны. Чтобы исключить их из вывода команды `git log` используйте опцию `--no-merges`.

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, 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 — внесите необходимые изменения, добавьте их в индекс и сделайте коммит ещё раз, указав параметр `--amend`:

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

Эта команда использует область подготовки (индекс) для внесения правок в коммит. 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 всё что вы сможете изменить — это ваше сообщение к коммиту.

Запустится тот же редактор, только он уже будет содержать сообщение предыдущего коммита. Вы можете редактировать сообщение как обычно, однако, оно заменит сообщение предыдущего коммита.

Например, если вы сделали коммит и поняли, что забыли проиндексировать изменения в файле, который хотели добавить в коммит, то можно сделать следующее:

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

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

Очень важно понимать, что когда вы вносите правки в последний коммит, вы не столько исправляете его, сколько *заменяете* новым, который полностью его перезаписывает. В результате всё выглядит так, будто первоначальный коммит никогда не существовал, а так же он больше не появится в истории вашего репозитория.

> Очевидно, смысл изменения коммитов в добавлении незначительных правок в последние коммиты и, при этом, в избежании засорения истории сообщениями вида «Ой, забыл добавить файл» или «Исправление грамматической ошибки».

Отмена индексации файла

Следующие два раздела демонстрируют как работать с индексом и изменениями в рабочем каталоге. Радует, что команда, которой вы определяете состояние этих областей, также подсказывает вам как отменять изменения в них. Например, вы изменили два файла и хотите добавить их в разные коммиты, но случайно выполнили команду `git add *` и добавили в индекс оба. Как исключить из индекса один 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: используйте `git reset HEAD <file>…` для исключения из индекса. 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 больше не добавлен в индекс.

Команда `git reset` *может* быть опасной если вызвать её с параметром

> `--hard`. В приведённом примере файл не был затронут, следовательно команда относительно безопасна.

На текущий момент этот магический вызов — всё, что вам нужно знать о команде `git reset`. Мы рассмотрим в деталях что именно делает `reset` и как с её помощью делать действительно интересные вещи в разделе Раскрытие тайн reset главы 7.

Отмена изменений в файле

Что делать, если вы поняли, что не хотите сохранять свои изменения файла `CONTRIBUTING.md`? Как можно просто отменить изменения в нём — вернуть к тому состоянию, которое было в последнем коммите (или к начальному после клонирования, или ещё как-то полученному)? Нам повезло, что `git status` подсказывает и это тоже.

В выводе команды из последнего примера список изменений 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 явно сказано как отменить существующие 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. Все локальные изменения в файле пропадут — Git просто заменит его версией из последнего коммита. Ни в коем случае не используйте эту команду, если вы не уверены, что изменения в файле вам не нужны.

Если вы хотите сохранить изменения в файле, но прямо сейчас их нужно отменить, то есть способы получше, такие как ветвление и припрятывание — мы рассмотрим их в главе Ветвление в Git.

Помните, всё что попало в *коммит* почти всегда Git может восстановить. Можно восстановить даже коммиты из веток, которые были удалены, или коммиты, перезаписанные параметром `--amend` (см. Восстановление данных). Но всё, что не было включено в коммит и потеряно — скорее всего, потеряно навсегда.

Отмена действий с помощью git restore

Git версии 2.23.0 представил новую команду: `git restore`. По сути, это альтернатива git reset, которую мы только что рассмотрели. Начиная с версии 2.23.0, Git будет использовать `git restore` вместо `git reset` для многих операций отмены.

Давайте проследим наши шаги и отменим действия с помощью `git restore` вместо `git reset`.

Отмена индексации файла с помощью git restore

В следующих двух разделах показано, как работать с индексом и изменениями рабочей копии с помощью `git restore`. Приятно то, что команда, которую вы используете для определения состояния этих двух областей, также напоминает вам, как отменить изменения в них. Например, предположим, что вы изменили два файла и хотите зафиксировать их как два отдельных изменения, но случайно набираете `git add *` и индексируете их оба. Как вы можете убрать из индекса один из двух? Команда `git status` напоминает вам:

```
$ git add *
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   CONTRIBUTING.md
    renamed:    README.md -> README
```

Прямо под текстом «Changes to be committed», написано использовать `git restore --staged <file> …` для отмены индексации файла. Итак, давайте воспользуемся этим советом, чтобы убрать из индекса файл `CONTRIBUTING.md`:

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

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   CONTRIBUTING.md
```

Файл `CONTRIBUTING.md` изменен, но снова не индексирован.

Откат изменённого файла с помощью git restore

Что, если вы поймёте, что не хотите сохранять изменения в файле `CONTRIBUTING.md`? Как легко его откатить — вернуть обратно к тому, как он выглядел при последнем коммите (или

изначально клонирован, или каким-либо образом помещён в рабочий каталог)? К счастью, `git status` тоже говорит, как это сделать. В выводе последнего примера, неиндексированная область выглядит следующим образом:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   CONTRIBUTING.md
```

Он довольно недвусмысленно говорит, как отменить сделанные вами изменения. Давайте сделаем то, что написано:

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

Важно понимать, что `git restore <file>` — опасная команда. Любые локальные изменения, внесённые в этот файл, исчезнут — Git просто заменит файл последней зафиксированной версией. Никогда не используйте эту команду, если точно не знаете, нужны ли вам эти несохранённые локальные изменения.

Работа с удалёнными репозиториями

Для того, чтобы внести вклад в какой-либо Git-проект, вам необходимо уметь работать с удалёнными репозиториями. Удалённые репозитории представляют собой версии вашего проекта, сохранённые в интернете или ещё где-то в сети. У вас может быть несколько удалённых репозиториев, каждый из которых может быть доступен для чтения или для чтения-записи. Взаимодействие с другими пользователями предполагает управление удалёнными репозиториями, а также отправку и получение данных из них. Управление репозиториями включает в себя как умение добавлять новые, так и умение удалять устаревшие репозитории, а также умение управлять различными удалёнными ветками, объявлять их отслеживаемыми или нет и так далее. В данном разделе мы рассмотрим некоторые из этих навыков.

Удалённый репозиторий может находиться на вашем локальном компьютере.

Вполне возможно, что удалённый репозиторий будет находиться на том же компьютере, на котором работаете вы. Слово «удалённый» не означает, что репозиторий обязательно должен быть где-то в сети или Интернет, а значит только — где-то ещё. Работа с таким удалённым репозиторием подразумевает выполнение стандартных операций отправки и получения, как и с любым другим удалённым репозиторием.

Просмотр удалённых репозиториев

Для того, чтобы просмотреть список настроенных удалённых репозиториев, вы можете запустить команду `git remote`. Она выведет названия доступных удалённых репозиториев. Если вы клонировали репозиторий, то увидите как минимум `origin` — имя по умолчанию, которое Git даёт серверу, с которого производилось клонирование:

```
$ 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
```

Вы можете также указать ключ `-v`, чтобы просмотреть адреса для чтения и записи, привязанные к репозиторию:

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

Если у вас больше одного удалённого репозитория, команда выведет их все. Например, для репозитория с несколькими настроенными удалёнными репозиториями в случае совместной работы нескольких пользователей, вывод команды может выглядеть примерно так:

```
$ 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)
```

Это означает, что мы можем легко получить изменения от любого из этих пользователей. Возможно, что некоторые из репозиториев доступны для записи и в них можно отправлять свои изменения, хотя вывод команды не даёт никакой информации о правах доступа.

Обратите внимание на разнообразие протоколов, используемых при указании адреса удалённого репозитория; подробнее мы рассмотрим протоколы в разделе Установка Git на сервер главы 4.

Добавление удалённых репозиториев

В предыдущих разделах мы уже упоминали и приводили примеры добавления удалённых репозиториев, сейчас рассмотрим эту операцию подробнее. Для того, чтобы добавить удалённый репозиторий и присвоить ему имя (shortname), просто выполните команду `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)
```

Теперь вместо указания полного пути вы можете использовать `pb`. Например, если вы хотите получить изменения, которые есть у Пола, но нету у вас, вы можете выполнить команду `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
```

Ветка `master` из репозитория Пола сейчас доступна вам под именем `pb/master`. Вы можете слить её с одной из ваших веток или переключить на неё локальную ветку, чтобы просмотреть содержимое ветки Пола. Более подробно работа с ветками рассмотрена в главе Ветвление в Git.

Получение изменений из удалённого репозитория — Fetch и Pull

Как вы только что узнали, для получения данных из удалённых проектов, следует выполнить:

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

Данная команда связывается с указанным удалённым проектом и забирает все те данные

проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта, которые вы можете просмотреть или слить в любой момент.

Когда вы клонируете репозиторий, команда `clone` автоматически добавляет этот удалённый репозиторий под именем «origin». Таким образом, `git fetch origin` извлекает все наработки, отправленные на этот сервер после того, как вы его клонировали (или получили изменения с помощью fetch). Важно отметить, что команда `git fetch` забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы.

Если ветка настроена на отслеживание удалённой ветки (см. следующий раздел и главу Ветвление в Git чтобы получить больше информации), то вы можете использовать команду `git pull` чтобы автоматически получить изменения из удалённой ветки и слить их со своей текущей. Этот способ может для вас оказаться более простым или более удобным. К тому же, по умолчанию команда `git clone` автоматически настраивает вашу локальную ветку `master` на отслеживание удалённой ветки `master` на сервере, с которого вы клонировали репозиторий. Название веток может быть другим и зависит от ветки по умолчанию на сервере. Выполнение `git pull`, как правило, извлекает (fetch) данные с сервера, с которого вы изначально клонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете.

Начиная с версии 2.27, команда `git pull` выдаёт предупреждение, если настройка `pull.rebase` не установлена. Git будет выводить это предупреждение каждый раз пока настройка не будет установлена.

Если хотите использовать поведение Git по умолчанию (простое смещение вперёд если возможно — иначе создание коммита слияния): `git config --global pull.rebase "false"`

Если хотите использовать перебазирование при получении изменений: `git config --global pull.rebase "true"`

Отправка изменений в удалённый репозиторий (Push)

Когда вы хотите поделиться своими наработками, вам необходимо отправить их в удалённый репозиторий. Команда для этого действия простая: `git push <remote-name> <branch-name>`. Чтобы отправить вашу ветку `master` на сервер `origin` (повторимся, что клонирование обычно настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки ваших коммитов:

```
$ git push origin master
```

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и если никто другой с тех пор не выполнял команду `push`. Если вы и кто-то ещё одновременно клонируете, затем он выполняет команду `push`, а после него выполнить команду `push` попытаетесь вы, то ваш `push` точно будет отклонён. Вам придётся

сначала получить изменения и объединить их с вашими и только после этого вам будет позволено выполнить `push`. Обратитесь к главе Ветвление в Git для более подробного описания, как отправлять изменения на удалённый сервер.

Просмотр удалённого репозитория

Если хотите получить побольше информации об одном из удалённых репозиториев, вы можете использовать команду `git remote show <remote>`. Выполнив эту команду с некоторым именем, например, `origin`, вы получите следующий результат:

```
$ 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)
```

Она выдаёт URL удалённого репозитория, а также информацию об отслеживаемых ветках. Эта команда любезно сообщает вам, что если вы, находясь на ветке `master`, выполните `git pull`, ветка `master` с удалённого сервера будет автоматически влита в вашу сразу после получения всех необходимых данных. Она также выдаёт список всех полученных ею ссылок.

Это был пример для простой ситуации и вы наверняка встречались с чем-то подобным. Однако, если вы используете Git более интенсивно, вы можете увидеть гораздо большее количество информации от `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                     (up to
date)
    markdown-strip                 pushes to markdown-strip                 (up to
date)
    master                         pushes to master                         (up to
date)
```

Данная команда показывает какая именно локальная ветка будет отправлена на удалённый сервер по умолчанию при выполнении `git push`. Она также показывает, каких веток с удалённого сервера у вас ещё нет, какие ветки всё ещё есть у вас, но уже удалены на сервере, и для нескольких веток показано, какие удалённые ветки будут в них влиты при выполнении `git pull`.

Удаление и переименование удалённых репозиториев

Для переименования удалённого репозитория можно выполнить `git remote rename`. Например, если вы хотите переименовать `pb` в `paul`, вы можете это сделать при помощи `git remote rename`:

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

Стоит упомянуть, что это также изменит имена удалённых веток в вашем репозитории. То, к чему вы обращались как `pb/master`, теперь стало `paul/master`.

Если по какой-то причине вы хотите удалить удалённый репозиторий — вы сменили сервер или больше не используете определённое зеркало, или кто-то перестал вносить изменения — вы можете использовать `git remote rm`:

```
$ git remote remove paul
$ git remote
origin
```

При удалении ссылки на удалённый репозиторий все отслеживаемые ветки и настройки, связанные с этим репозиторием, так же будут удалены.

Работа с тегами

Как и большинство других систем контроля версий, Git имеет возможность помечать определённые моменты в истории как важные. Как правило, эта функциональность используется для отметки моментов выпуска версий (v1.0, и т. п.). Такие пометки в Git называются тегами. В этом разделе вы узнаете, как посмотреть имеющиеся теги, как

создать новые или удалить существующие, а также какие типы тегов существуют в Git.

Просмотр списка тегов

Просмотреть список имеющихся тегов в Git можно очень просто. Достаточно набрать команду `git tag` (параметры `-l` и `--list` опциональны):

```
$ git tag
v1.0
v2.0
```

Данная команда перечисляет теги в алфавитном порядке; порядок их отображения не имеет существенного значения.

Так же можно выполнить поиск тега по шаблону. Например, репозиторий Git содержит более 500 тегов. Если вы хотите посмотреть теги выпусков 1.8.5, то выполните следующую команду:

```
$ git tag -l "v1.8.5*"
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Для отображение тегов согласно шаблону требуются параметры `-l` или `--list`

Если вы хотите посмотреть весь список тегов, запуск команды `git tag` неявно подразумевает это и выводит полный список; использование параметров `-l` или `--list` в этом случае опционально.

Если вы хотите отфильтровать список тегов согласно шаблону, использование параметров `-l` или `--list` становится обязательным.

Создание тегов

Git использует два основных типа тегов: легковесные и аннотированные.

Легковесный тег — это что-то очень похожее на ветку, которая не изменяется — просто указатель на определённый коммит.

А вот аннотированные теги хранятся в базе данных Git как полноценные объекты. Они имеют контрольную сумму, содержат имя автора, его e-mail и дату создания, имеют

комментарий и могут быть подписаны и проверены с помощью GNU Privacy Guard (GPG). Обычно рекомендуется создавать аннотированные теги, 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.

Аннотированные теги

Создание аннотированного тега в Git выполняется легко. Самый простой способ — это указать `-a` 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, которое будет храниться вместе с тегом. Если не указать сообщение, то Git запустит редактор, чтобы вы смогли его ввести.

С помощью команды `git show` вы можете посмотреть данные тега вместе с коммитом:

```
$ 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

    Change version number
```

Здесь приведена информация об авторе тега, дате его создания и аннотирующее сообщение перед информацией о коммите.

Легковесные теги

Легковесный тег — это ещё один способ пометить коммит. По сути, это контрольная сумма коммита, сохранённая в файл — больше никакой информации не хранится. Для создания легковесного тега не передавайте опций `-a`, `-s` и `-m`, укажите только название:

```
$ 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` для этого тега вы не увидите дополнительной информации. Команда просто покажет коммит:

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

    Change version number
```

Отложенная расстановка тегов

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

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 Create write support
0d52aaab4479697da7686c15f77a3d64d9165190 One more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc Add commit function
4682c3261057305bdd616e23b64b0857d832627b Add todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a Create write support
9fceb02d0ae598e95dc970b74767f19372d61af8 Update rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc Commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a Update readme
```

Теперь предположим, что вы забыли отметить версию проекта v1.2, которая была там, где находится коммит «Update rakefile». Вы можете добавить тег и позже. Для отметки коммита укажите его контрольную сумму (или её часть) как параметр команды:

```
$ git tag -a v1.2 9fceb02
```

Проверим, что коммит 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 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    Update rakefile
...
```

Обмен тегами

По умолчанию, команда `git push` не отправляет теги на удалённые сервера. После создания теги нужно отправлять явно на удалённый сервер. Процесс аналогичен отправке веток — достаточно выполнить команду `git push origin <tagname>`.

```
$ 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 много тегов, и вам хотелось бы отправить всё за один раз, то можно использовать опцию `--tags` для команды `git push`. В таком случае все ваши теги 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 клонирует (clone) или выполнит `git pull` из вашего репозитория, то он получит вдобавок к остальному и ваши метки.

`git push` *отправляет оба типа тегов*

Отправка тегов командой `git push <remote> --tags` не различает аннотированные и легковесные теги. В настоящее время не существует

> опции чтобы отправить только легковесные теги, но если использовать команду `git push <remote> --follow-tags`, то отправятся только аннотированные теги.

Удаление тегов

Для удаления тега в локальном репозитории достаточно выполнить команду `git tag -d <tagname>`. Например, удалить созданный ранее легковесный тег можно следующим образом:

```
$ git tag -d v1.4-lw
Deleted tag 'v1.4-lw' (was e7d5add)
```

Обратите внимание, что при удалении тега не происходит его удаления с внешних серверов. Существует два способа изъятия тега из внешнего репозитория.

Первый способ — это выполнить команду `git push <remote> :refs/tags/<tagname>`:

```
$ git push origin :refs/tags/v1.4-lw
To /git@github.com:schacon/simplegit.git
 - [deleted]         v1.4-lw
```

Это следует понимать как обновление внешнего тега пустым значением, что приводит к его удалению.

Второй способ убрать тег из внешнего репозитория более интуитивный:

```
$ git push origin --delete <tagname>
```

Переход на тег

Если вы хотите получить версии файлов, на которые указывает тег, то вы можете сделать `git checkout` для тега. Однако, это переведёт репозиторий в состояние «detached HEAD», которое имеет ряд неприятных побочных эффектов.

```
$ git checkout v2.0.0
Note: switching to 'v2.0.0'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>
```

```
Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 99ada87... Merge pull request #89 from schacon/appendix-final

$ git checkout v2.0-beta-0.1
Previous HEAD position was 99ada87... Merge pull request #89 from schacon/appendix-
final
HEAD is now at df3f601... Add atlas.json and cover image
```

Если в состоянии «detached HEAD» внести изменения и сделать коммит, то тег не изменится, при этом новый коммит не будет относиться ни к какой из веток, а доступ к нему можно будет получить только по его хешу. Поэтому, если вам нужно внести изменения — исправить ошибку в одной из старых версий — скорее всего вам следует создать ветку:

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

Если сделать коммит в ветке `version2`, то она сдвинется вперёд и будет отличаться от тега `v2.0.0`, так что будьте с этим осторожны.

Псевдонимы в Git

Прежде, чем закончить эту главу по основам Git, рассмотрим ещё одну маленькую хитрость, которая поможет сделать использование Git 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 не полностью. Если вы не хотите печатать каждую команду для Git 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 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. 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: 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 — его модель ветвления.

Ветвление в Git

Почти каждая система контроля версий в той или иной форме поддерживает ветвление. Используя ветвление, Вы отклоняетесь от основной линии разработки и продолжаете работу независимо от неё, не вмешиваясь в основную линию. Во многих системах контроля версий создание веток — это очень затратный процесс, часто требующий создания новой копии каталога с исходным кодом, что может занять много времени для большого проекта.

Некоторые люди, говоря о модели ветвления Git, называют её «киллер-фича», что выгодно выделяет Git на фоне остальных систем контроля версий. Что в ней такого особенного? Ветвление Git очень легковесно: операция создания ветки выполняется почти мгновенно, переключение между ветками туда-сюда, обычно, также быстро. В отличие от многих других систем контроля версий, Git поощряет процесс работы, при котором ветвление и слияние выполняется часто, даже по несколько раз в день. Понимание и владение этой функциональностью даёт вам уникальный и мощный инструмент, который может полностью изменить привычный процесс разработки.

О ветвлении в двух словах

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

Как вы можете помнить из Что такое Git?, 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 двух и более веток.

Предположим, у вас есть каталог с тремя файлами и вы добавляете их все в индекс и создаёте коммит. Во время индексации вычисляется контрольная сумма каждого файла (SHA-1 как мы узнали из Что такое Git?), затем каждый файл сохраняется в репозиторий (Git называет такой файл *блоб* — большой бинарный объект), а контрольная сумма попадёт в индекс:

```
$ git add README test.rb LICENSE
$ git commit -m 'Initial commit'
```

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

Ваш репозиторий Git теперь хранит пять объектов: три блоб объекта (по одному на каждый файл), объект *дерева* каталогов, содержащий список файлов и соответствующих им блобов, а так же объект *коммита*, содержащий метаданные и указатель на объект дерева каталогов.

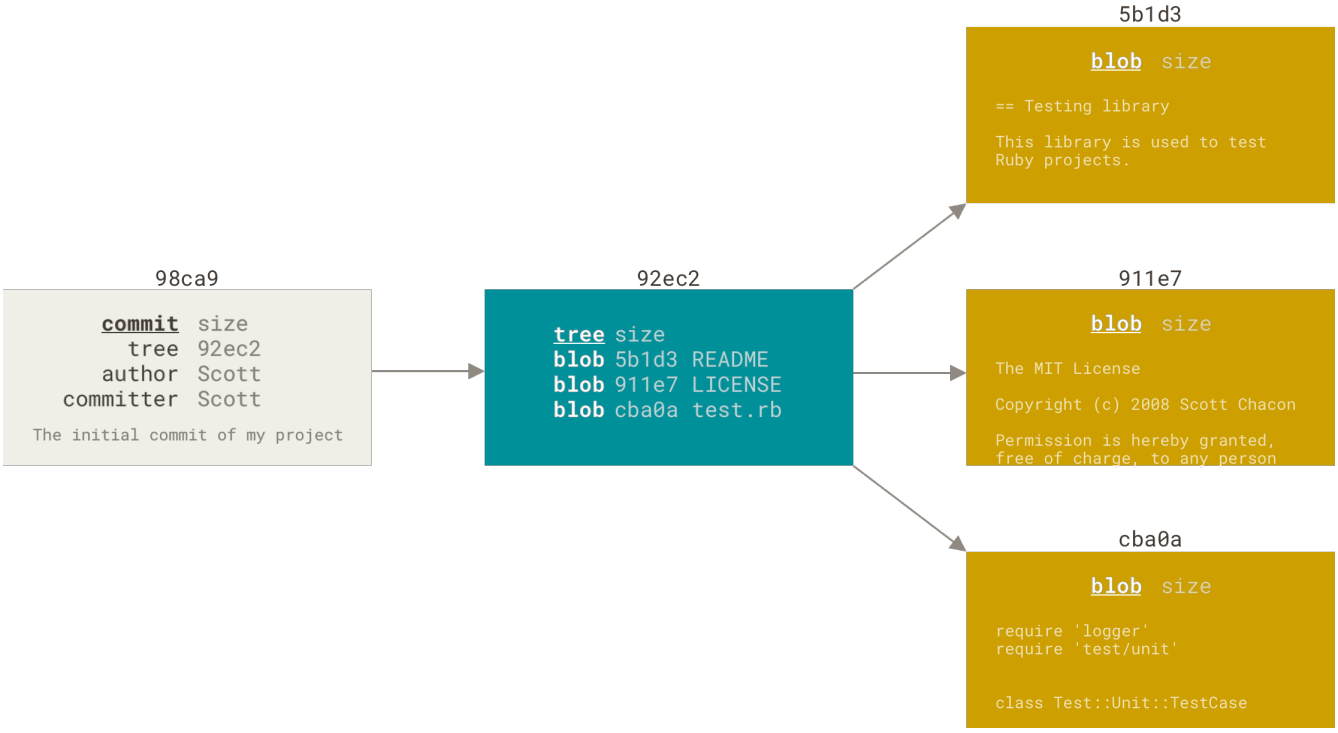


Рисунок 9. Коммит и его дерево

Если вы сделаете изменения и создадите ещё один коммит, то он будет содержать указатель на предыдущий коммит.

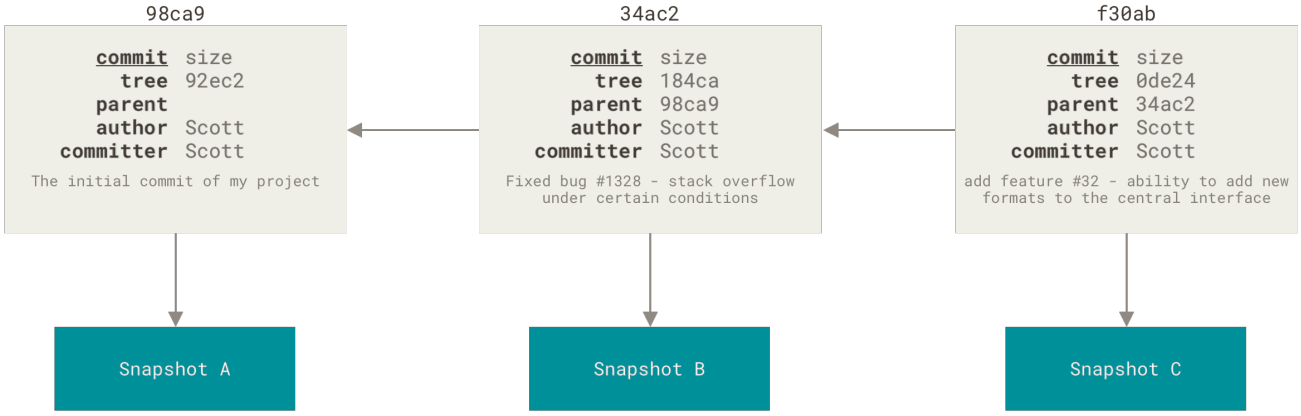


Рисунок 10. Коммит и его родители

Ветка в Git — это простой перемещаемый указатель на один из таких коммитов. По умолчанию, имя основной ветки в Git — `master`. Как только вы начнёте создавать коммиты, ветка `master` будет всегда указывать на последний коммит. Каждый раз при создании коммита указатель ветки `master` будет передвигаться на следующий коммит автоматически.

Ветка «master» в Git — это не какая-то особенная ветка. Она точно такая же, как и все остальные ветки. Она существует почти во всех репозиториях только лишь потому, что её создаёт команда `git init`, а большинство людей

не меняют её название.

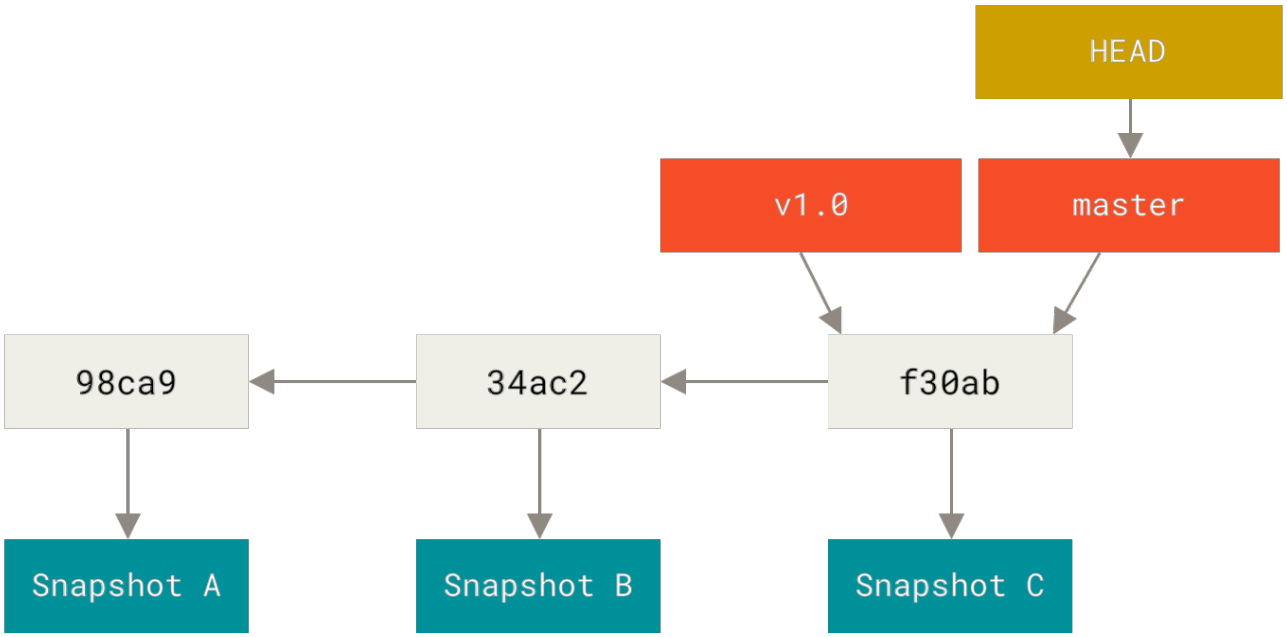


Рисунок 11. Ветка и история коммитов

Создание новой ветки

Что же на самом деле происходит при создании ветки? Всего лишь создаётся 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 создаётся новый указатель на текущий коммит.

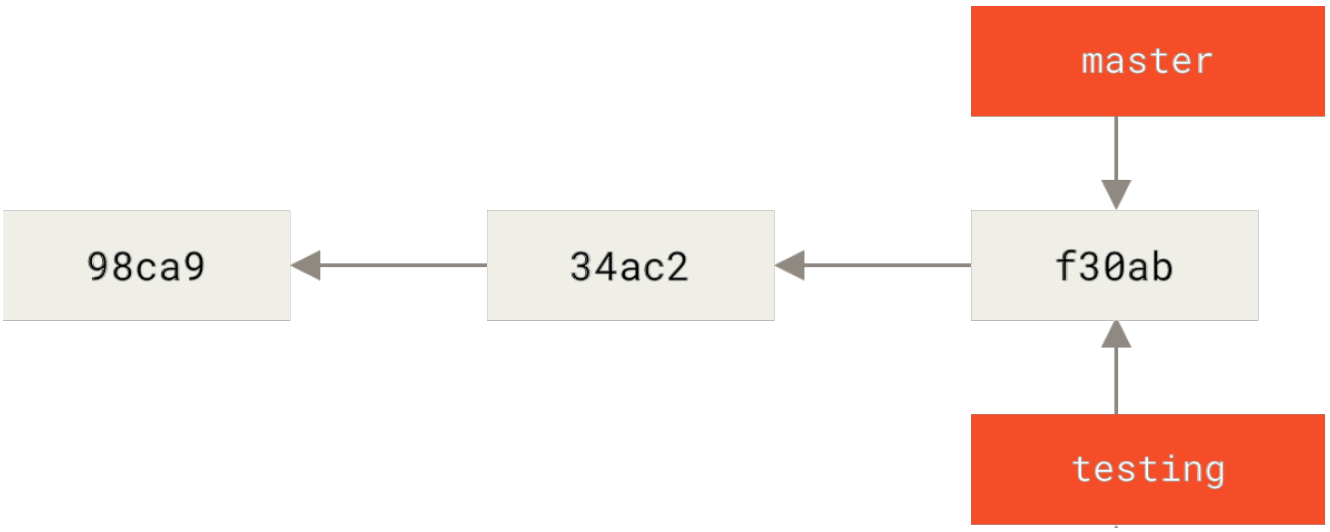


Рисунок 12. Две ветки указывают на одну и ту же последовательность коммитов

Как Git определяет, в какой ветке вы находитесь? Он хранит специальный указатель `HEAD`. Имейте в виду, 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 текущую локальную ветку. В нашем случае мы всё ещё находимся в ветке `master`. Команда `git branch` только *создаёт* новую ветку, но не переключает на неё.

Рисунок 13. HEAD указывает на ветку

Вы можете легко это увидеть при помощи простой команды `git log`, которая 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 formats to the
central interface
34ac2 Fix bug #1328 - stack overflow under certain conditions
98ca9 Initial commit
```

Здесь можно увидеть указывающие на коммит `f30ab` ветки: `master` и `testing`.

Переключение веток

Для переключения на существующую ветку выполните команду `git checkout`. Давайте переключимся на ветку `testing`:

```
$ git checkout testing
```

В результате указатель `HEAD` переместится на ветку `testing`.

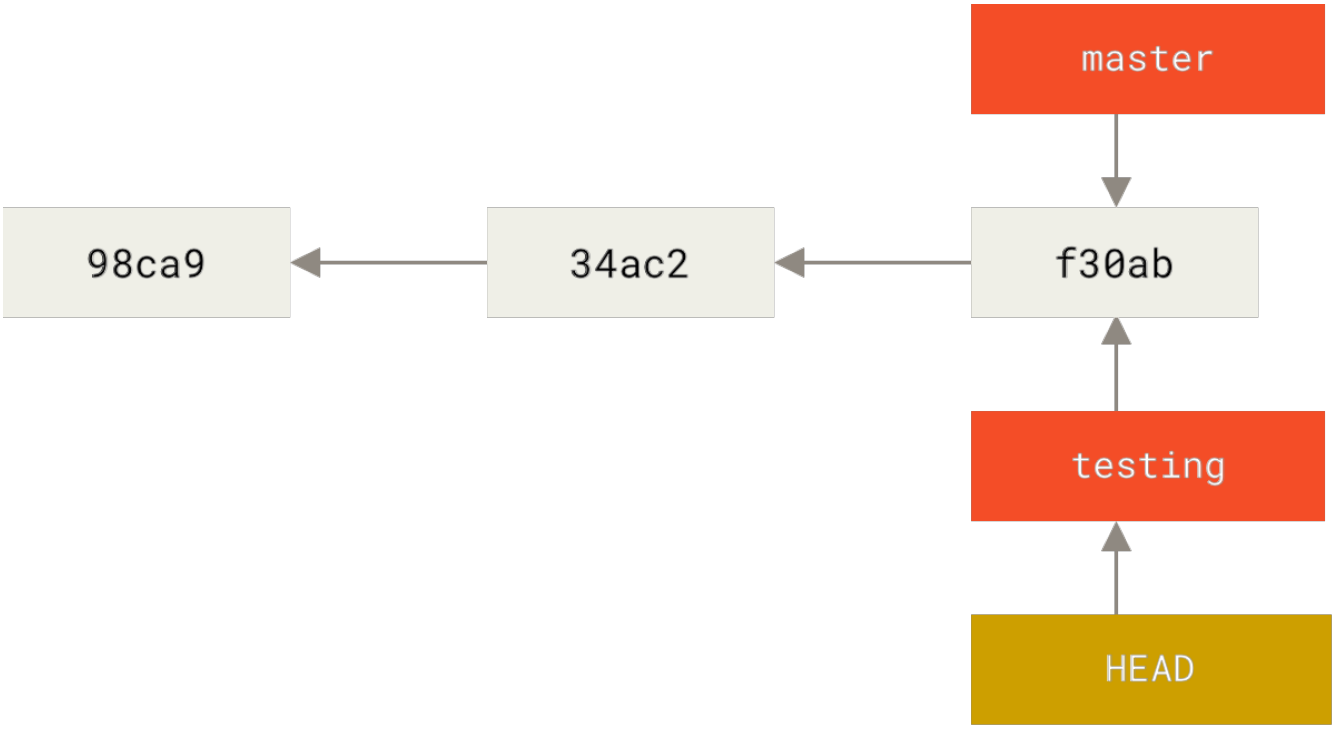


Рисунок 14. HEAD указывает на текущую ветку

Какой в этом смысл? Давайте сделаем ещё один коммит:

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

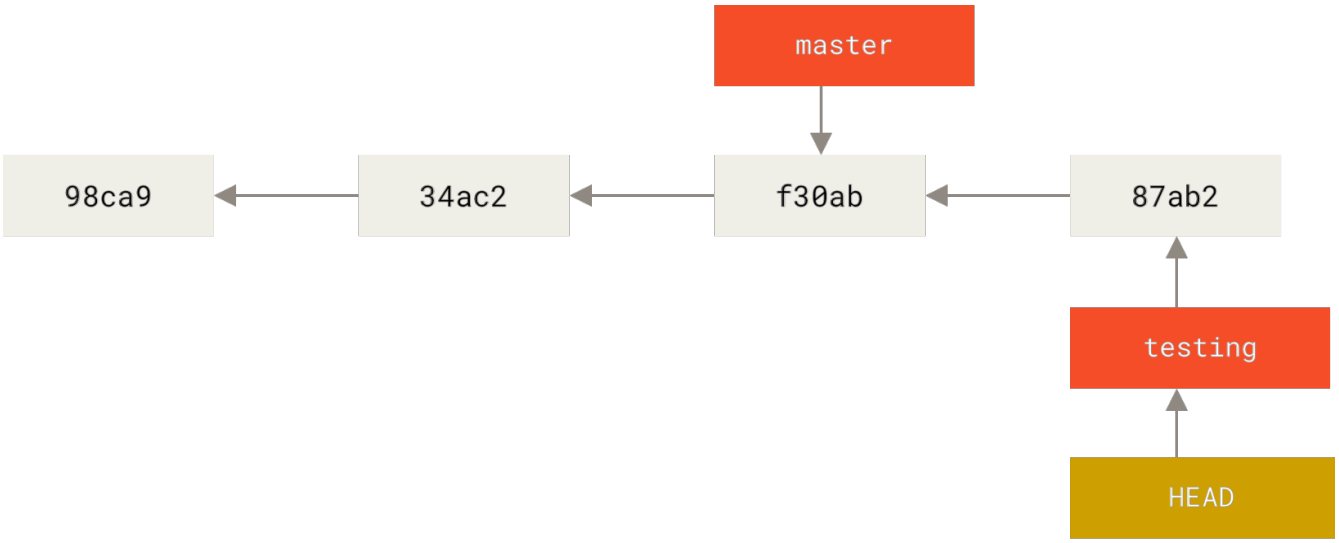


Рисунок 15. Указатель на ветку HEAD переместился вперёд после коммита

Интересная ситуация: указатель на ветку `testing` переместился вперёд, а `master` указывает на тот же коммит, где вы были до переключения веток командой `git checkout`. Давайте переключимся назад на ветку `master`:

```
$ git checkout master
```

`git log` не показывает все ветки по умолчанию

Если выполнить команду `git log` прямо сейчас, то в её выводе только что созданная ветка «testing» фигурировать не будет.

Ветка никуда не исчезла; просто Git не знает, что именно она вас интересует, и выводит наиболее полезную по его мнению информацию. Другими словами, по умолчанию `git log` отобразит историю коммитов только для текущей ветки.

Для просмотра истории коммитов другой ветки необходимо явно указать её имя: `git log testing` Чтобы посмотреть историю по всем веткам — выполните команду с дополнительным флагом: `git log --all`.

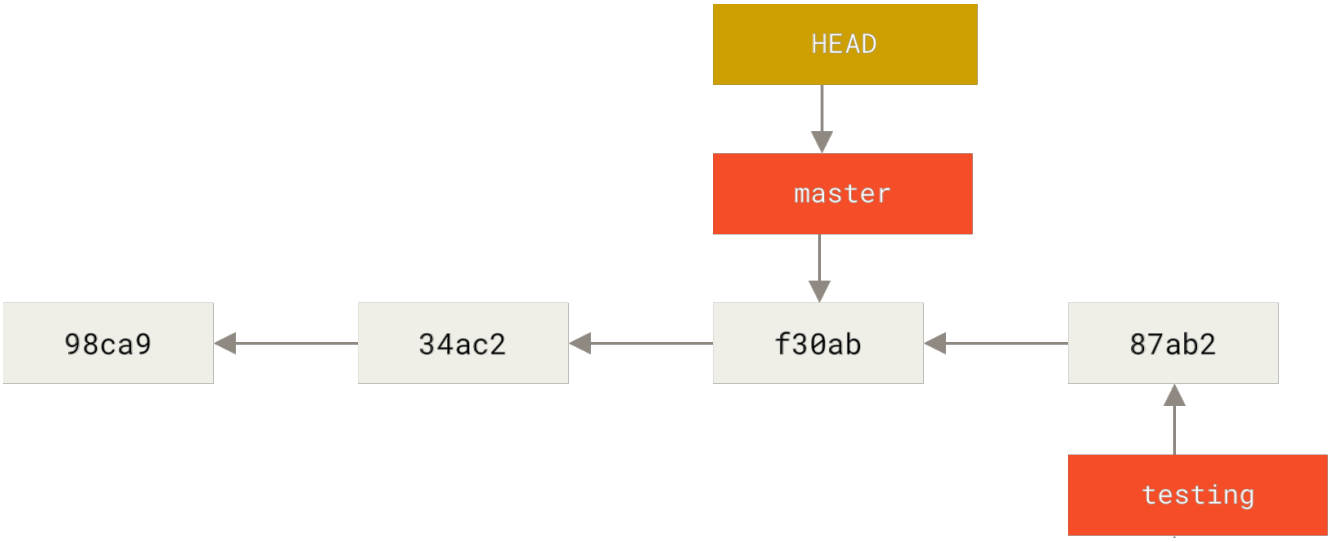


Рисунок 16. HEAD перемещается когда вы делаете checkout

Эта команда сделала две вещи: переместила указатель `HEAD` назад на ветку `master` и вернула файлы в рабочем каталоге в то состояние, на снимок которого указывает `master`. Это также означает, что все вносимые с этого момента изменения будут относиться к старой версии проекта. Другими словами, вы откатили все изменения ветки `testing` и можете продолжать в другом направлении.

Переключение веток меняет файлы в рабочем каталоге

Важно запомнить, что при переключении веток в Git происходит изменение файлов в рабочем каталоге. 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 вообще.

Давайте сделаем ещё несколько изменений и создадим очередной коммит:

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

Теперь история вашего проекта разошлась (см Разветвлённая история). Вы создали ветку и переключились на неё, 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 понадобится — объединить их. И всё это делается простыми командами: `branch`, `checkout` и `commit`.

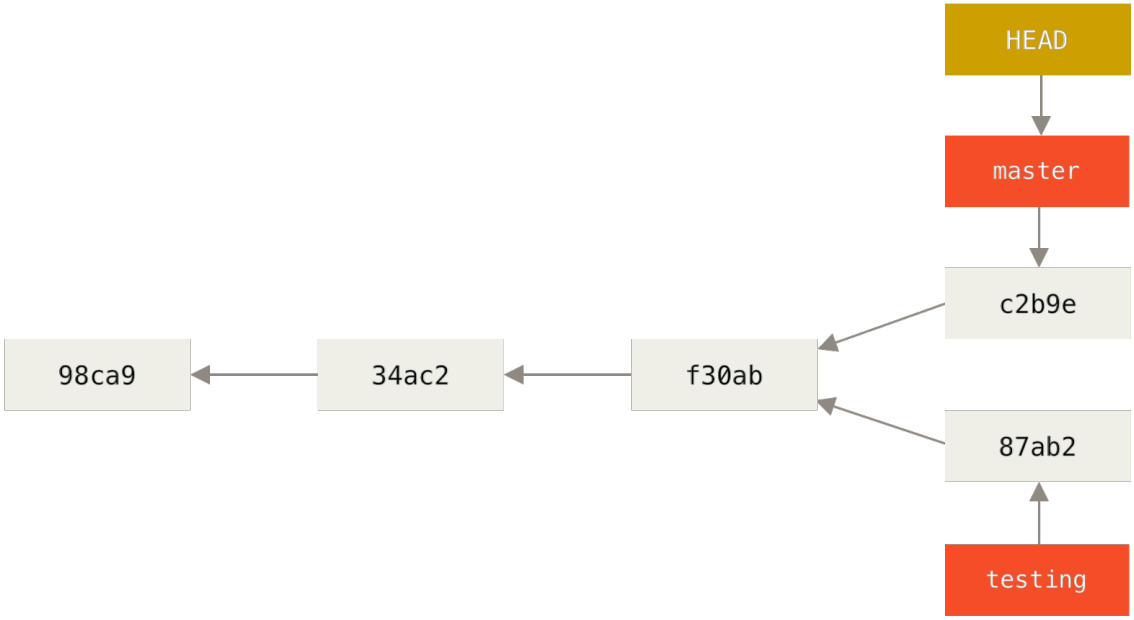


Рисунок 17. Разветвлённая история

Все описанные действия можно визуализировать с помощью команды `git log`. Для отображения истории коммитов, текущего положения указателей веток и истории ветвления выполните команду `git log --oneline --decorate --graph --all`.

```
$ 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 central interface
* 34ac2 Fix bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Ветка в Git — это простой файл, содержащий 40 символов контрольной суммы SHA-1 коммита, на который она указывает; поэтому операции с ветками являются дешёвыми с точки зрения потребления ресурсов или времени. Создание новой ветки в Git происходит так же быстро и просто как запись 41 байта в файл (40 знаков и перевод строки).

Это принципиально отличает процесс ветвления в Git от более старых систем контроля версий, где все файлы проекта копируются в другой подкаталог. В зависимости от размера проекта, операции ветвления в таких системах могут занимать секунды или даже минуты, когда в Git эти операции мгновенны. Поскольку при коммите мы сохраняем указатель на

родительский коммит, то поиск подходящей базы для слияния веток делается автоматически и, в большинстве случаев, очень прост. Эти возможности побуждают разработчиков чаще создавать и использовать ветки.

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

Одновременное создание новой ветки и переключение на неё

Как правило, при создании новой ветки вы хотите сразу на неё переключиться — это можно сделать используя команду `git checkout -b <newbranchname>`.

Начиная с Git версии 2.23, вы можете использовать `git switch` вместо `git checkout`, чтобы:

- Переключиться на существующую ветку: `git switch testing-branch`.
- Создать новую ветку и переключиться на неё: `git switch -c new-branch`. Флаг `-c` означает создание, но также можно использовать полный формат:` --create`.
- Вернуться к предыдущей извлечённой ветке: `git switch -`.

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 создаёте ветку для реализации новой функциональности в соответствии с пользовательской историей.
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 ветку для реализации пользовательской истории и продолжить работать.

Основы ветвления

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

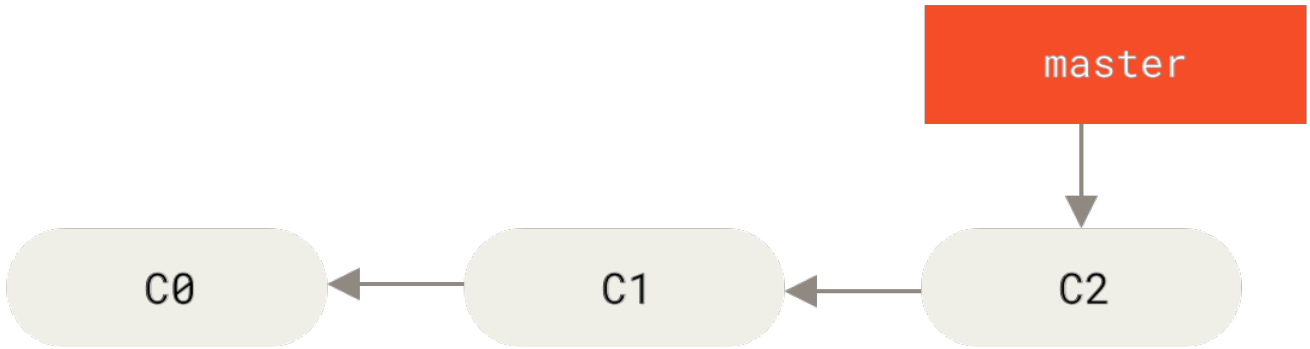


Рисунок 18. Простая история коммитов

Вы выбрали задачу #53 из какая-там-у-вас-система-отслеживания-задач. Чтобы создать ветку и сразу переключиться на неё, можно выполнить команду `git checkout` с параметром `-b`:

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

Это то же самое, что и:

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

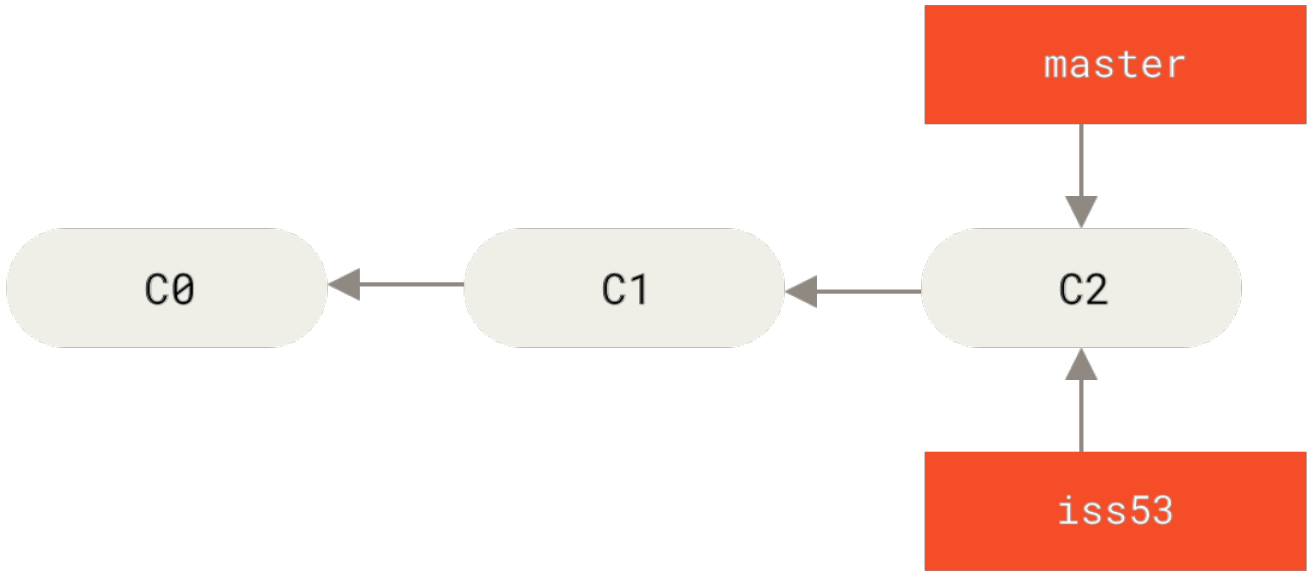


Рисунок 19. Создание нового указателя ветки

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

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

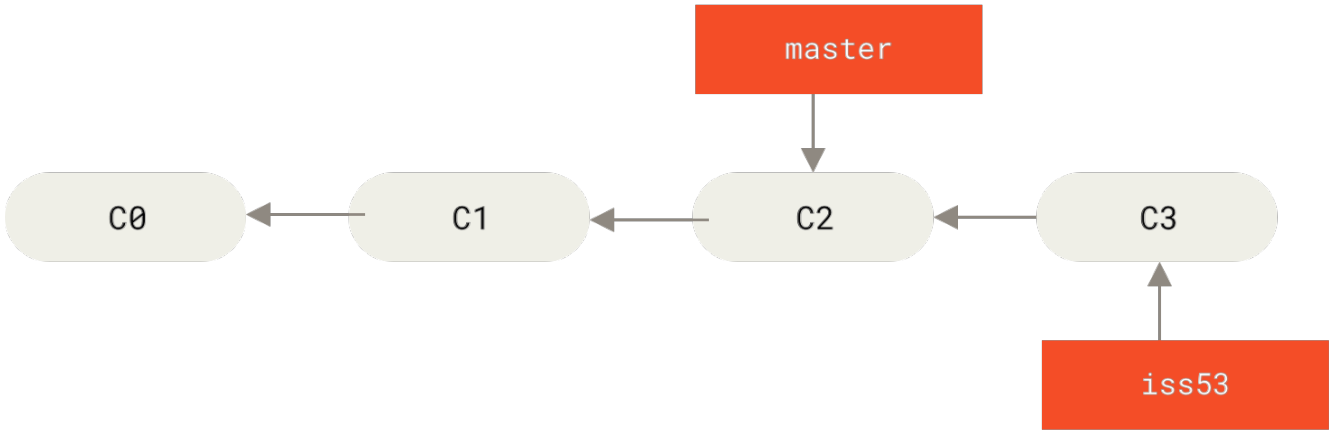


Рисунок 20. Ветка iss53 движется вперёд

И тут вы получаете сообщение об обнаружении на сайте уязвимости, и эту уязвимость устранить нужно немедленно. Благодаря Git вам не придётся ни пытаться реализовать исправление вместе с изменениями, которые вы сделали в ходе разработки `iss53`, ни прилагать усилия для отката этих изменений и возвращения к исходному состоянию перед началом разработки исправления. Всё, что вам нужно — переключиться на ветку `master`.

Имейте в виду, что если рабочий каталог или индекс содержат незафиксированные изменения, конфликтующие с веткой, на которую вы хотите переключиться, то Git не позволит переключить ветки. Лучше всего переключаться из чистого рабочего состояния проекта: все изменённые файлы добавить в индекс и сделать коммит. Есть способы обойти это (припрятать изменения (stash) или добавить их в последний коммит (amend)), но об этом мы поговорим позже в разделе Припрятывание и очистка главы 7. Теперь предположим, что вы зафиксировали все свои изменения и можете переключиться на ветку `master`:

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

С этого момента ваш рабочий каталог имеет точно такой же вид, какой был перед началом работы над задачей #53, и 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 переключаемую 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, в которой реализуем исправление.

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'Fix broken email address'
[hotfix 1fb7853] Fix broken email address
```

```
 1 file changed, 2 insertions(+)
```

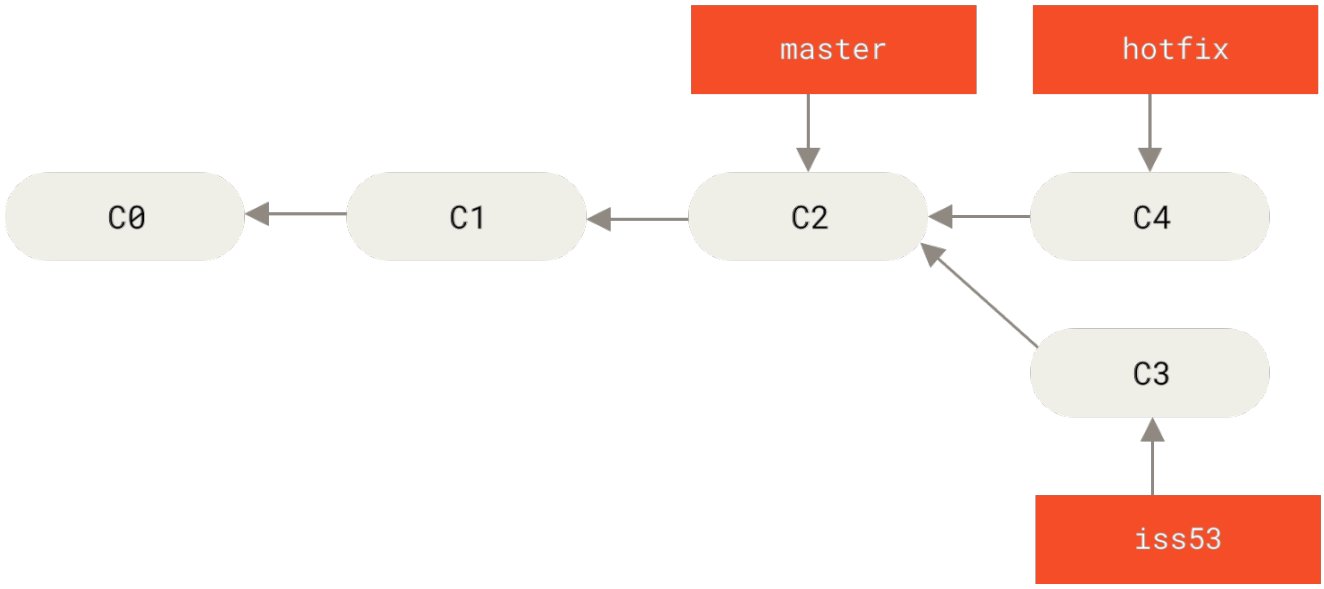


Рисунок 21. Ветка hotfix основана на ветке `master`

Вы можете прогнать тесты, чтобы убедиться, что ваша уязвимость в самом деле исправлена. И если это так — выполнить слияние ветки `hotfix` с веткой `master` для включения изменений 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? Git просто переместил указатель ветки вперёд, потому что коммит `C4`, на который указывает слитая ветка `hotfix`, был прямым потомком коммита `C2`, на котором вы находились до этого. Другими словами, если коммит сливается с тем, до которого можно добраться, двигаясь по истории вперёд, Git упрощает слияние, просто перенося указатель ветки вперёд, потому что в этом случае нет никаких разнонаправленных изменений, которые нужно было бы свести воедино. Это называется «fast-forward».

Теперь ваши изменения включены в коммит, на который указывает ветка `master`, и исправление можно внедрять.

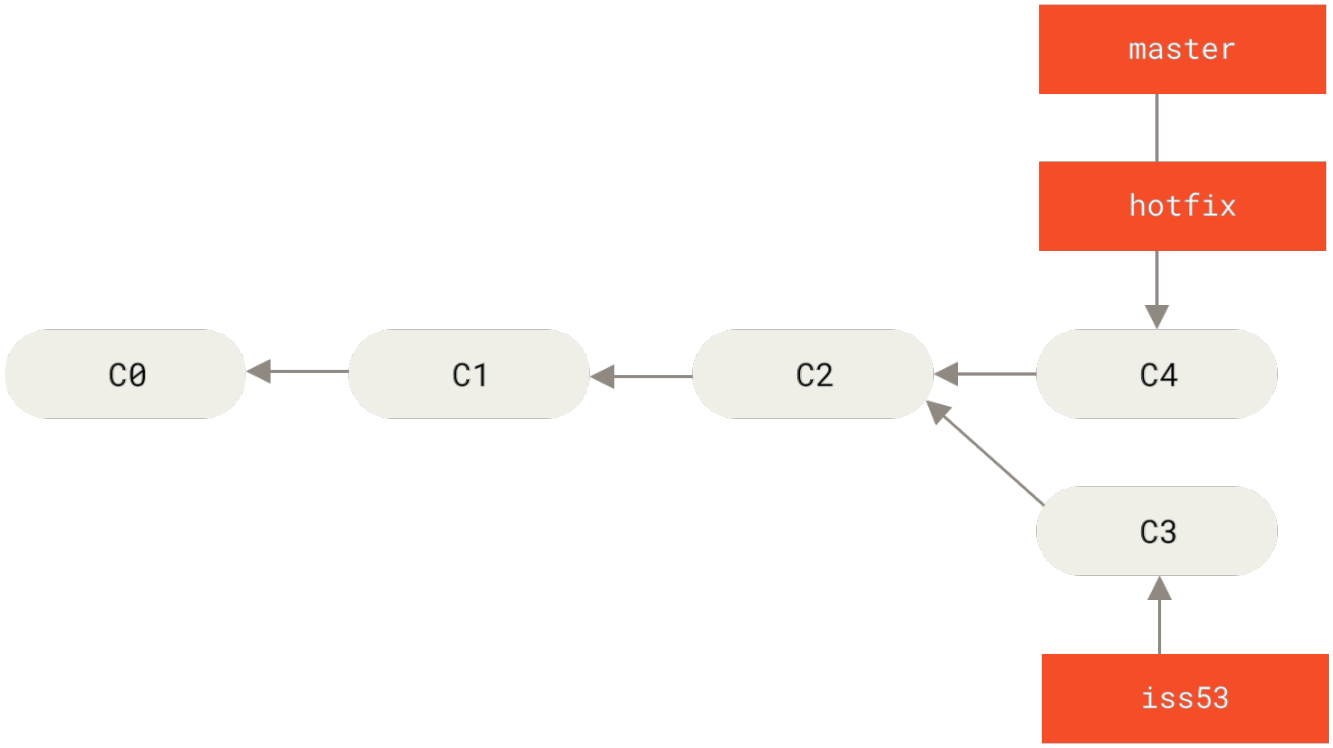


Рисунок 22. `master` *перемотан до* `hotfix`

После внедрения вашего архиважного исправления вы готовы вернуться к работе над тем, что были вынуждены отложить. Но 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 переключиться обратно на ветку `iss53` и продолжить работу над задачей #53:

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

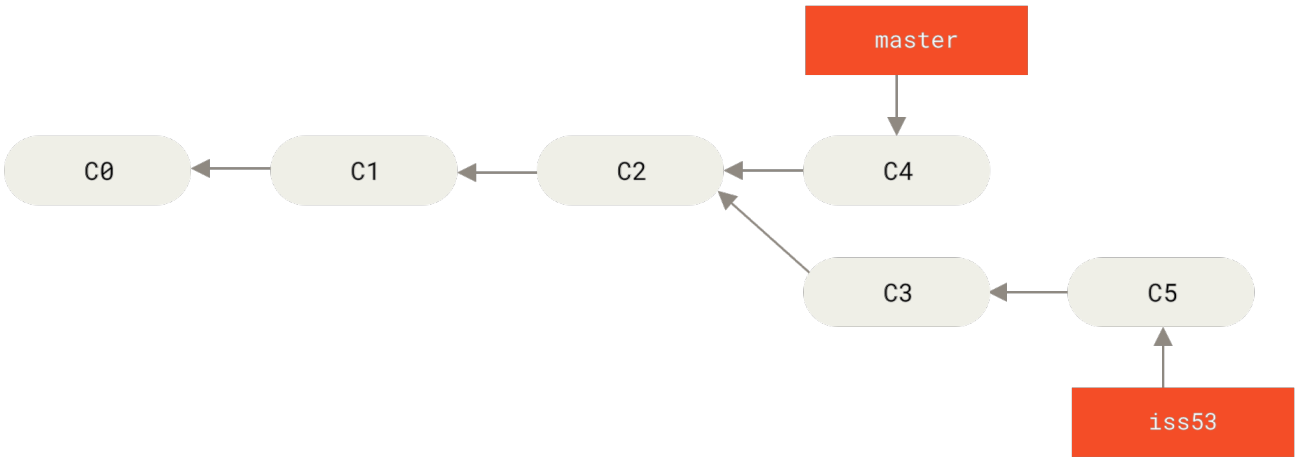


Рисунок 23. Продолжение работы над `iss53`

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`, а 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 её можно влить в ветку `master`. Для этого нужно выполнить слияние ветки `iss53` точно так же, как вы делали это с веткой `hotfix` ранее. Всё, 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 процесс разработки ответвился в более ранней точке. Так как коммит, на котором мы находимся, не является прямым родителем ветки, с которой мы выполняем слияние, Git придётся немного потрудиться. В этом случае Git выполняет простое трёхстороннее слияние, используя последние коммиты объединяемых веток и общего для них родительского коммита.

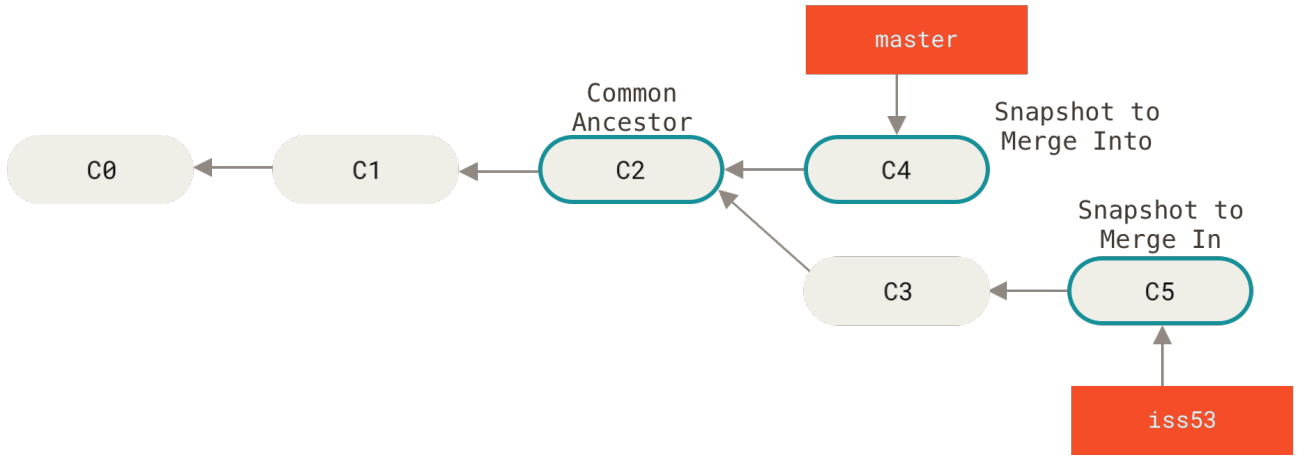


Рисунок 24. Использование трёх снимков при слиянии

Вместо того, чтобы просто передвинуть указатель ветки вперёд, Git создаёт новый результирующий снимок трёхстороннего слияния, а затем автоматически делает коммит. Этот особый коммит называют коммитом слияния, так как у него более одного предка.

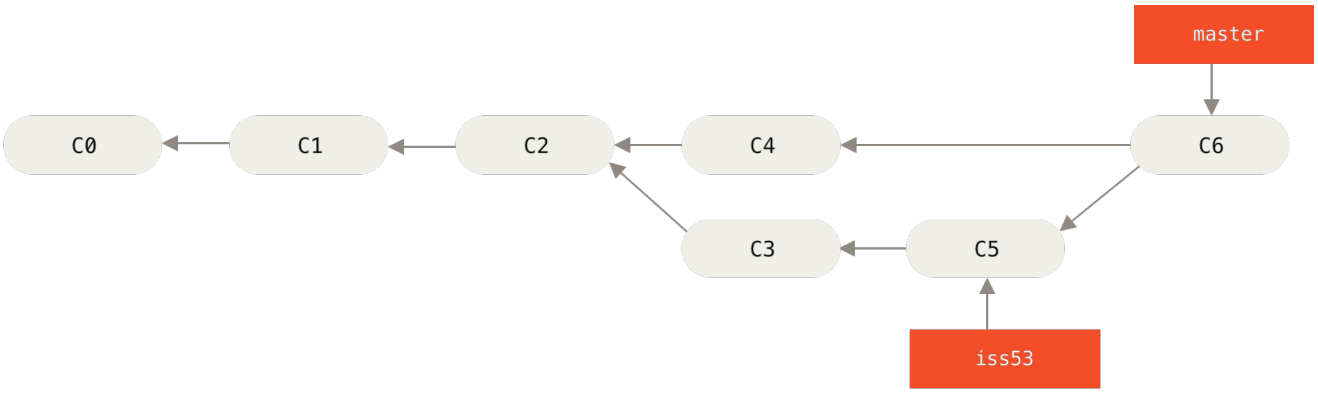


Рисунок 25. Коммит слияния

Теперь, когда изменения слиты, ветка `iss53` больше не нужна. Вы можете закрыть задачу 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")
```

Всё, где есть неразрешённые конфликты слияния, перечисляется как неслитое. В конфликтующие файлы Git добавляет специальные маркеры конфликтов, чтобы вы могли исправить их вручную. 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 её вы извлекли перед запуском команды слияния) — это верхняя часть блока (всё, что над `=======`), а версия из вашей ветки `iss53` представлена в нижней части. Чтобы разрешить конфликт, придётся выбрать один из вариантов, либо объединить содержимое по-своему. Например, вы можете разрешить конфликт, заменив весь блок следующим:

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

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

каждого файла, чтобы отметить конфликт как решённый. Добавление файла в индекс означает для Git, что все конфликты в нём исправлены.

Если вы хотите использовать графический инструмент для разрешения конфликтов, можно запустить `git mergetool`, который проведёт 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
p4merge araxis bc3 codecompare vimdiff emerge
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 инструмент 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 хотите использовать.

Мы рассмотрим более продвинутые инструменты для разрешения сложных конфликтов слияния в разделе Продвинутое слияние главы 7.

После выхода из инструмента слияния Git спросит об успешности процесса. Если вы ответите скрипту утвердительно, то он добавит файл в индекс, чтобы отметить его как разрешённый. Теперь можно снова запустить `git status`, чтобы убедиться в отсутствии конфликтов:

```
$ 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 файлы, где были конфликты, добавлены в индекс — выполните команду `git commit` для создания коммита слияния. Комментарий к коммиту слияния 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
#
```

Если вы считаете, что коммит слияния требует дополнительных пояснений — опишите как были разрешены конфликты и почему были применены именно такие изменения, если это не очевидно.

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 сделаете 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 readme
```

Опции `--merged` и `--no-merged` могут отфильтровать этот список для вывода только тех веток, которые слиты или ещё не слиты в текущую ветку. Чтобы посмотреть те ветки, которые вы уже слили с текущей, можете выполнить команду `git branch --merged`:

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

Ветка `iss53` присутствует в этом списке потому что вы ранее слили её в `master`. 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` приведёт 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 вместе со всеми наработками, используйте опцию `-D`, как указано в подсказке.

Если в качестве аргумента не указан коммит или ветка, то опции `--merged` и `--no-merged` покажут что уже слито или не слито с вашей *текущей* веткой соответственно.

Вы всегда можете указать дополнительный аргумент для вывода той же информации, но относительно указанной ветки предварительно не извлекая и не переходя на неё.

```
$ git checkout testing
$ git branch --no-merged master
  topicA
  featureB
```

Переименование ветки

Не переименовывайте ветки, которые всё ещё используются другими участниками. Не переименовывайте ветку в master/main/mainline, не прочитав раздел Изменение_имени_главной_ветки.

Предположим, у вас есть ветка с именем `bad-branch-name`, и вы хотите изменить её на `corrected-branch-name`, сохранив при этом всю историю. Вместе с этим, вы также хотите изменить имя ветки на удалённом сервере (GitHub, GitLab или другой сервер). Как это сделать?

Переименуйте ветку локально с помощью команды `git branch --move`:

```
$ git branch --move bad-branch-name corrected-branch-name
```

Ветка `bad-branch-name` будет переименована в `corrected-branch-name`, но это изменение пока только локальное. Чтобы все остальные увидели исправленную ветку в удалённом репозитории, отправьте её туда:

```
$ git push --set-upstream origin corrected-branch-name
```

Теперь проверим, где мы сейчас находимся:

```
$ git branch --all
* corrected-branch-name
  main
  remotes/origin/bad-branch-name
  remotes/origin/corrected-branch-name
  remotes/origin/main
```

Обратите внимание, что текущая ветка `corrected-branch-name`, которая также присутствует и на удалённом сервере. Однако, старая ветка всё ещё по-прежнему там, но её можно удалить с помощью команды:

```
$ git push origin --delete bad-branch-name
```

Теперь старое имя ветки полностью заменено исправленным.

Изменение имени главной ветки

Изменение имени ветки, например master/main/mainline/default, сломает интеграции, службы, вспомогательные утилиты и скрипты сборки, которые использует ваш репозиторий. Прежде чем сделать это, обязательно проконсультируйтесь с коллегами. Также убедитесь, что вы выполнили тщательный поиск в своём репозитории и обновили все ссылки на старое

> имя ветки в вашем коде или скриптах.

Переименуйте локальную ветку `master` в `main` с помощью следующей команды:

```
$ git branch --move master main
```

После этого, локальной ветки `master` больше не существует, потому что она была переименована в ветку `main`.

Чтобы все остальные могли видеть новую ветку `main`, вам нужно отправить её в общий репозиторий. Это делает переименованную ветку доступной в удалённом репозитории.

```
$ git push --set-upstream origin main
```

В итоге, состояние репозитория становится следующим:

```
$ git branch --all
* main
  remotes/origin/HEAD -> origin/master
  remotes/origin/main
  remotes/origin/master
```

Ваша локальная ветка `master` исчезла, так как она заменена веткой `main`. Ветка `main` доступна в удалённом репозитории. Старая ветка `master` всё ещё присутствует в удалённом репозитории. Остальные участники будут продолжать использовать ветку `master` в качестве основы для своей работы, пока вы не совершите ряд дополнительных действий.

Теперь, для завершения перехода на новую ветку перед вами стоят следующие задачи:

- Все проекты, которые зависят от текущего, должны будут обновить свой код и/или конфигурацию.
- Обновите конфигурацию всех запускаемых тестов.
- Исправьте скрипты сборки и публикации артефактов.
- Поправьте настройки репозитория на сервере: задайте новую ветку по умолчанию, обновите правила слияния, а также прочие настройки, которые зависят от имени веток.
- Обновите документацию, исправив ссылки, указывающие на старую ветку.
- Слейте или отмените запросы на слияние изменений, нацеленные на старую ветку.

После того, как вы выполнили все эти задачи и уверены, что ветка `main` работает так же, как ветка `master`, вы можете удалить ветку `master`:

```
$ git push origin --delete master
```

Работа с ветками

Теперь, когда вы познакомились с основами ветвления и слияния, возникает вопрос: что можно или нужно делать с этим? В этом разделе мы разберём некоторые основные рабочие процессы, ставшие возможными благодаря облегчённой процедуре ветвления, которые вы возможно захотите применить в собственном цикле разработки.

Долгоживущие ветки

Так как в Git применяется простое трёхстороннее слияние, ничто не мешает многократно объединять ветки в течение длительного времени. Это значит, что у вас может быть несколько постоянно открытых веток, которые вы используете для разных этапов вашего цикла разработки; вы можете регулярно сливать изменения из одной ветки в другую.

Многие разработчики, использующие Git, придерживаются именно такого подхода, оставляя полностью стабильный код только в ветке `master` — возможно, только тот код, который был или будет выпущен. При этом существует и параллельная ветка с именем `develop` или `next`, предназначенная для работы и тестирования стабильности; она не обязательно должна быть всегда стабильной, но при достижении стабильного состояния её содержимое можно слить в ветку `master`. Она используется для слияния завершённых задач из тематических веток (временных веток наподобие `iss53`), чтобы гарантировать, что эти задачи проходят тестирование и не вносят ошибок.

По сути, мы говорим про указатели, перемещающиеся по линии создаваемых вами коммитов. Стабильные ветки находятся в нижнем конце истории коммитов, а самые свежие наработки — ближе к её верхней части

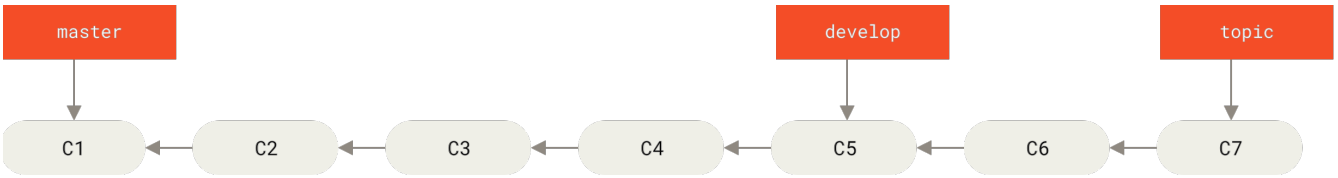


Рисунок 26. Линейное представление повышения стабильности веток

В общем случае это можно представить в виде накопителей, в которых наборы коммитов перемещаются на более стабильный уровень только после полного тестирования.

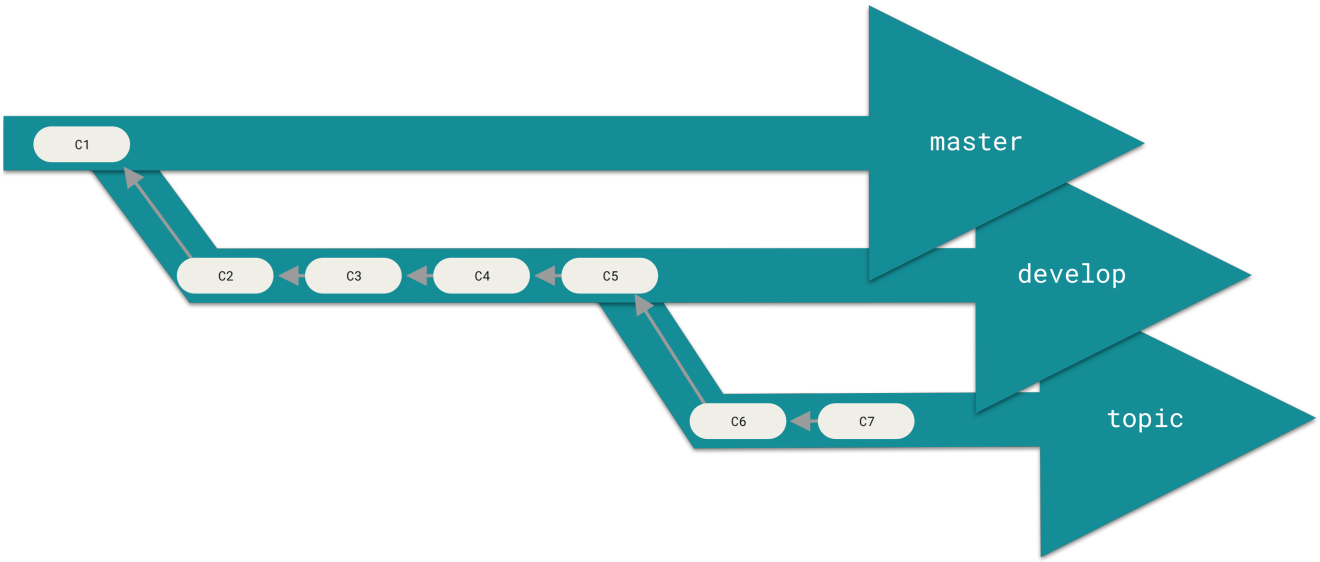


Рисунок 27. Представление диаграммы стабильности веток в виде многоуровневого накопителя

Число уровней стабильности можно увеличить. В крупных проектах зачастую появляется ветка `proposed` или `pu` (предлагаемые обновления), объединяющая ветки с содержимым, которое ещё не готово к включению в ветки `next` или `master`. Идея состоит в том, что каждая ветка представляет собой определённый уровень стабильности; как только он повышается, содержимое сливается в ветку уровнем выше. Разумеется, можно вообще обойтись без долгоживущих веток, но зачастую они имеют смысл, особенно при работе над большими и сложными проектами.

Тематические ветки

А вот такая вещь, как тематические ветки, полезна вне зависимости от величины проекта. Тематической веткой называется временная ветка, создаваемая и используемая для работы над конкретной функциональной возможностью или решения сопутствующих задач. Скорее всего, при работе с другими системами контроля версий вы никогда ничего подобного не делали, так как там создание и слияние веток — затратные операции. Но в Git это обычное дело — много раз в день создавать ветки, работать с ними, сливать их и удалять.

Пример тематических веток вы видели в предыдущем разделе, когда мы создавали ветки `iss53` и `hotfix`. В каждой из них было создано несколько коммитов, после чего, сразу же после слияния с основной веткой, они были удалены. Такая техника позволяет быстро и полностью осуществлять переключения контекста — так как работа разделена по уровням и все изменения в конкретной ветке относятся к определённой теме, что позволяет легко увидеть что именно было сделано во время процедуры просмотра кода или аналогичной. Ветки с внесёнными в них изменениями можно хранить минуты, дни или даже месяцы, а слияние выполнить только когда это действительно потребуется, вне зависимости от порядка их создания.

Предположим, мы работаем в ветке `master`, ответвляемся для решения попутной проблемы (`iss91`), некоторое время занимаемся ею, затем создаём ветку, чтобы попробовать решить эту задачу другим способом (`iss91v2`), возвращаемся в ветку `master` и выполняем там некие действия, затем создаём новую ветку для изменений, в результате которых не уверены

(ветка dumbidea). Результирующая история коммитов будет выглядеть примерно так:

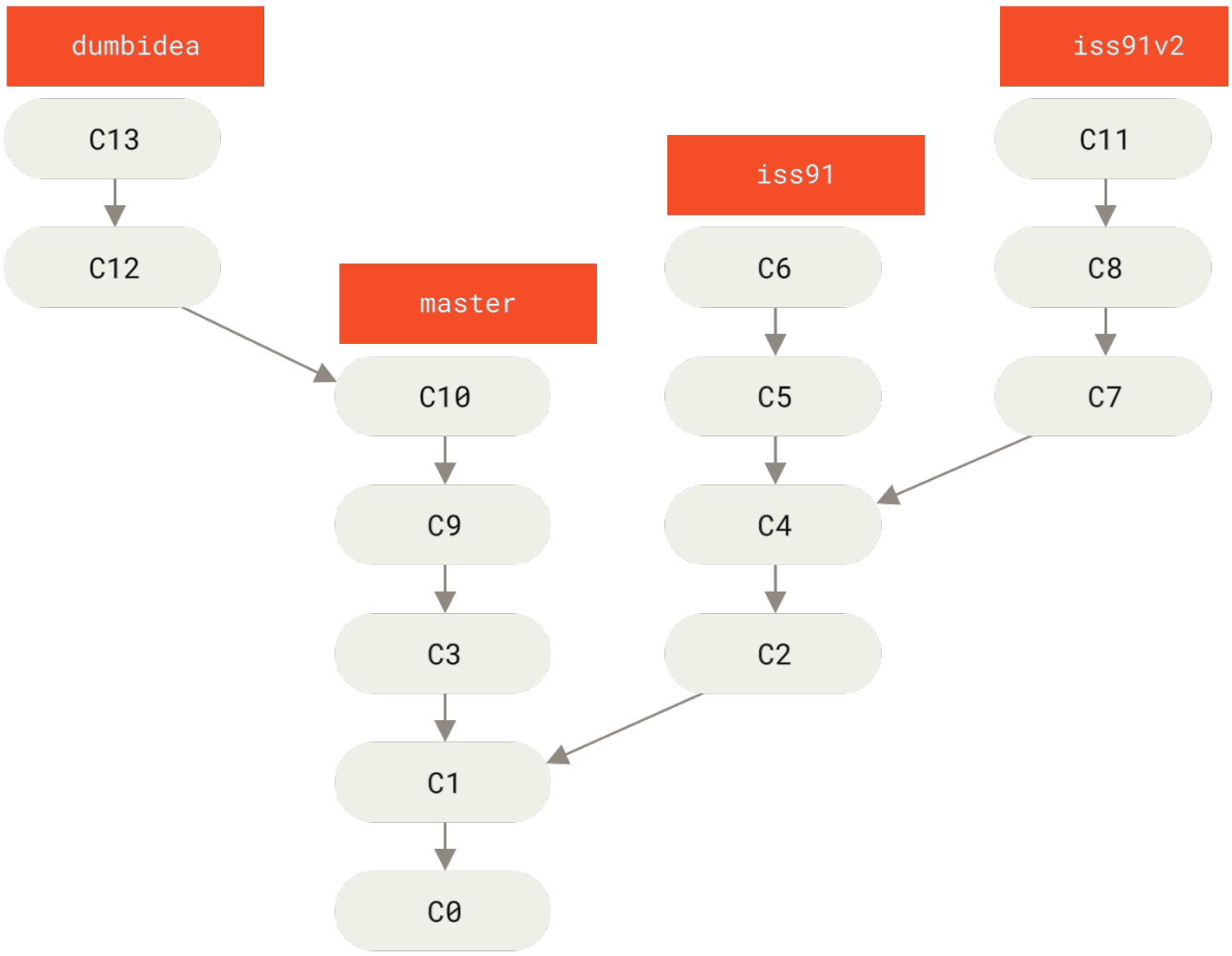


Рисунок 28. Набор тематических веток

Предположим, вам больше нравится второй вариант решения задачи (iss91v2), а ветку dumbidea вы показали коллегам, и оказалось, что там содержится гениальная идея. Фактически вы можете удалить ветку iss91 (потеряв коммиты C5 и C6) и слить две другие ветки. После этого история будет выглядеть так:

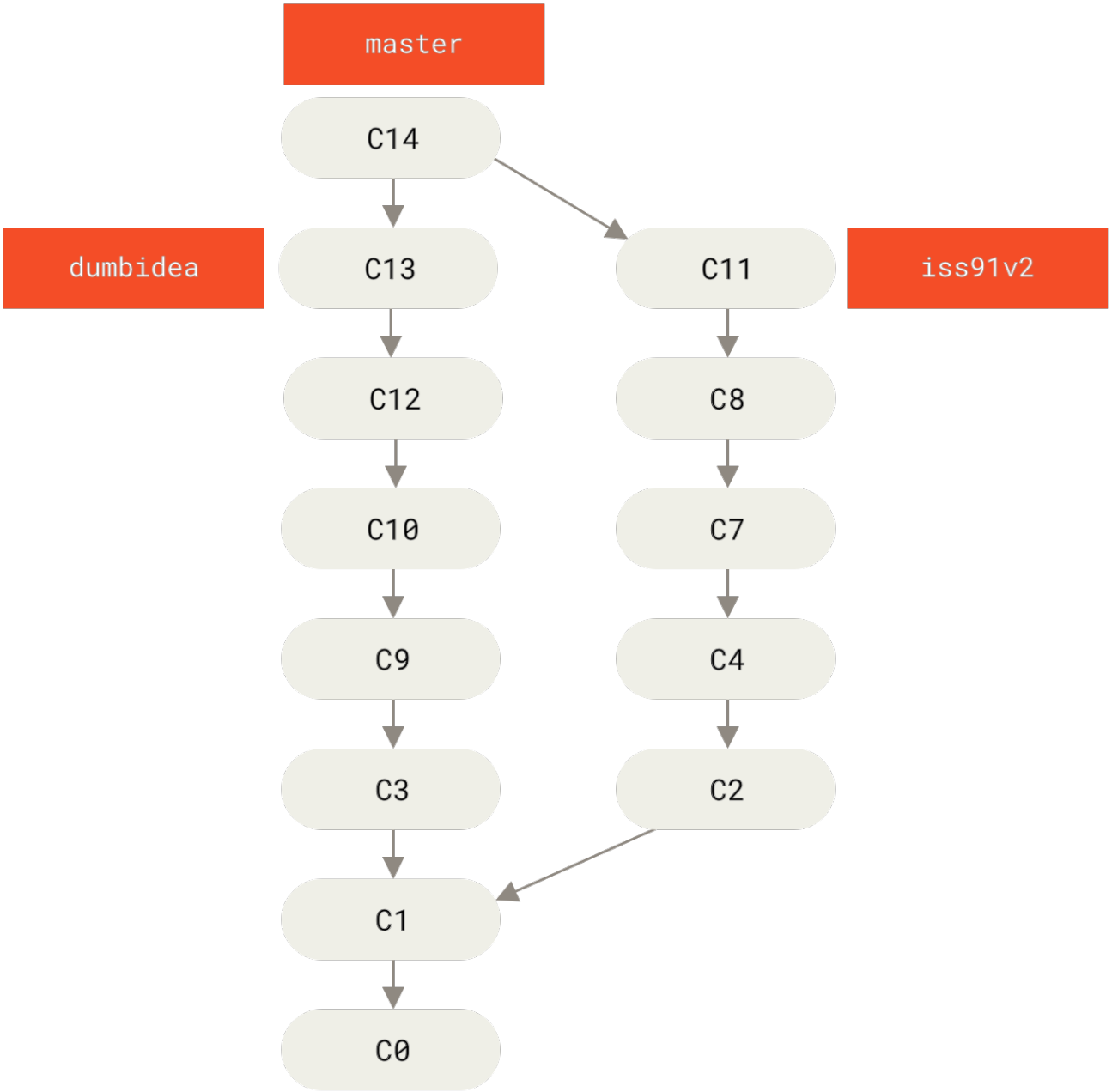


Рисунок 29. История после слияния веток `dumbidea` *и* `iss91v2`

Более подробно возможные варианты рабочих схем для проектов рассматриваются в главе Распределённый Git, поэтому перед выбором схемы обязательно прочитайте эту главу.

Важно помнить, что во время всех этих манипуляций ветки полностью локальны. Ветвления и слияния выполняются только в вашем Git репозитории — связь с сервером не требуется.

Удалённые ветки

Удалённые ссылки — это ссылки (указатели) в ваших удалённых репозиториях, включая ветки, теги и так далее. Полный список удалённых ссылок можно получить с помощью команды `git ls-remote <remote>` или команды `git remote show <remote>` для получения удалённых веток и дополнительной информации. Тем не менее, более распространённым способом является использование веток слежения.

Ветки слежения — это ссылки на определённое состояние удалённых веток. Это локальные ветки, которые нельзя перемещать; Git перемещает их автоматически при любой коммуникации с удалённым репозиторием, чтобы гарантировать точное соответствие с ним. Представляйте их как закладки для напоминания о том, где ветки в удалённых репозиториях находились во время последнего подключения к ним.

Имена веток слежения имеют вид `<remote>/<branch>`. Например, если вы хотите посмотреть, как выглядела ветка `master` на сервере `origin` во время последнего соединения с ним, используйте ветку `origin/master`. Если вы с коллегой работали над одной задачей и он отправил на сервер ветку `iss53`, при том что у вас может быть своя локальная ветка `iss53`, удалённая ветка будет представлена веткой слежения с именем `origin/iss53`.

Возможно, всё это сбивает с толку, поэтому давайте рассмотрим на примере. Скажем, у вас в сети есть свой Git-сервер с адресом `git.ourcompany.com`. Если вы с него что-то клонируете, 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 создаст вам локальную ветку `master`, которая будет начинаться там же, где и ветка `master` в `origin`, так что вам будет с чего начать.

«origin» — это не специальное название

Подобно названию ветки «master», «origin» не имеет какого-либо специального значения в Git. В то время как «master» — это название по умолчанию для ветки при выполнении `git init` только потому, что часто используется, «origin» — это название по умолчанию для удалённого сервера, когда вы запускаете `git clone`. Если вы выполните `git clone -o booyah`, то по умолчанию ветка слежения будет иметь вид `booyah/master`.

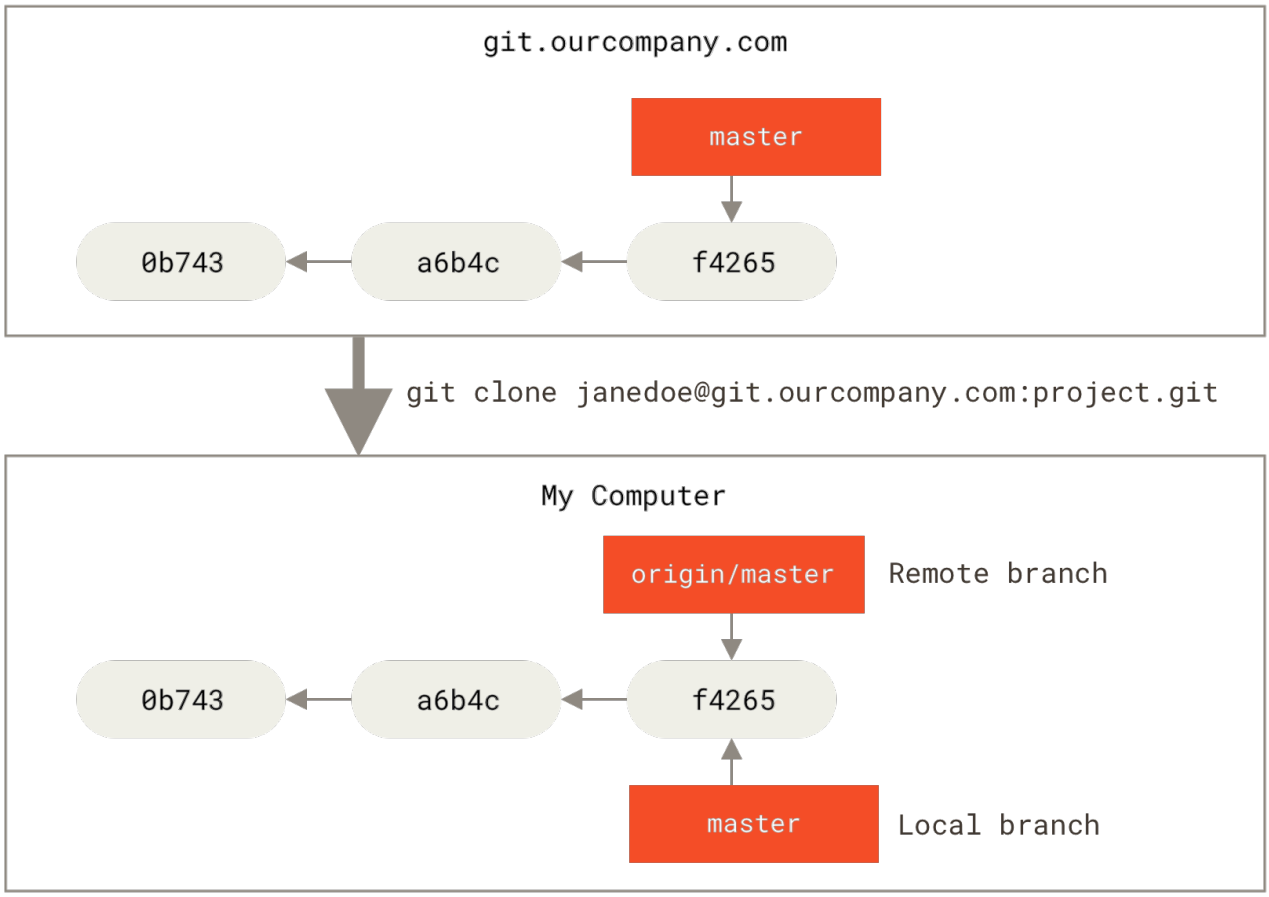


Рисунок 30. Серверный и локальный репозитории после клонирования

Если вы сделаете что-то в своей локальной ветке `master`, а тем временем кто-то отправит изменения на сервер `git.ourcompany.com` и обновит там ветку `master`, то ваши истории продолжатся по-разному. Пока вы не свяжетесь с сервером `origin` ваш указатель `origin/master` останется на месте.

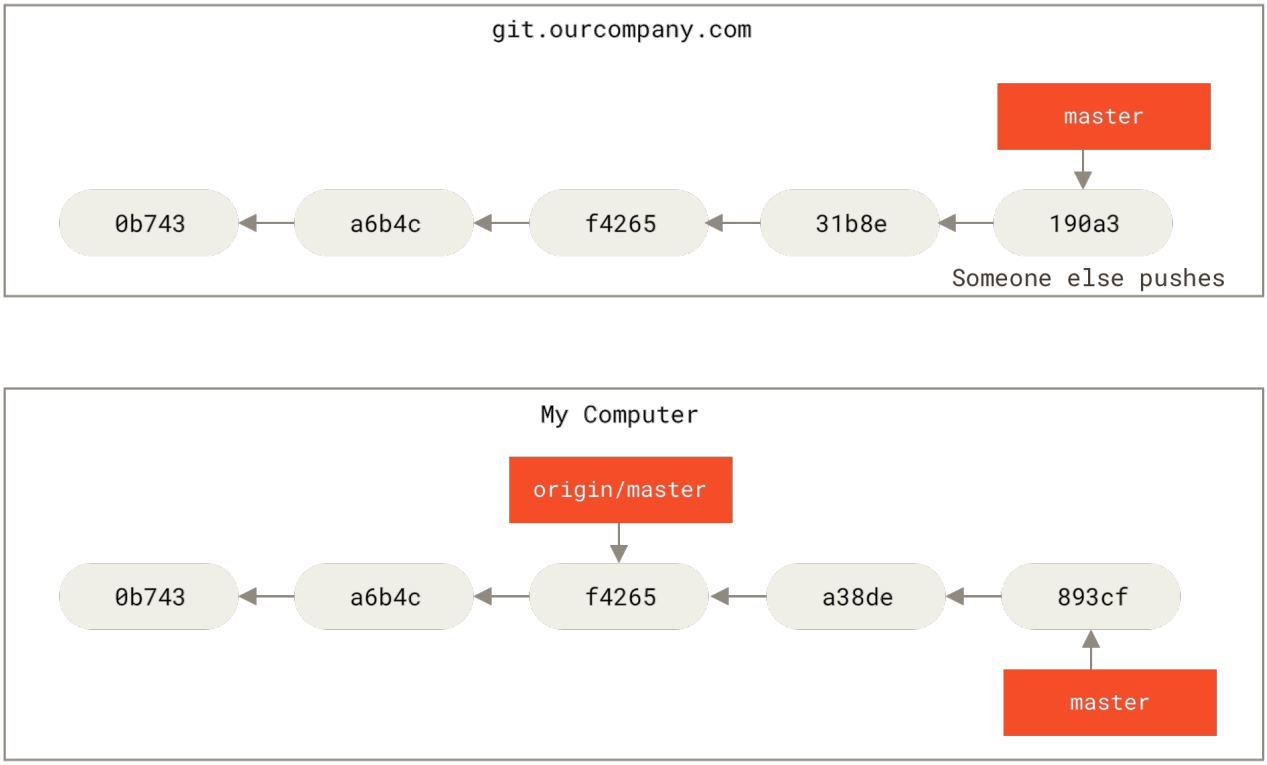


Рисунок 31. Локальная и удалённая работа может расходиться

Для синхронизации ваших изменений с удалённым сервером выполните команду `git fetch <remote>` (в нашем случае `git fetch origin`). Эта команда определяет какому серверу соответствует «origin» (в нашем случае это `git.ourcompany.com`), извлекает оттуда данные, которых у вас ещё нет, и обновляет локальную базу данных, сдвигая указатель `origin/master` на новую позицию.

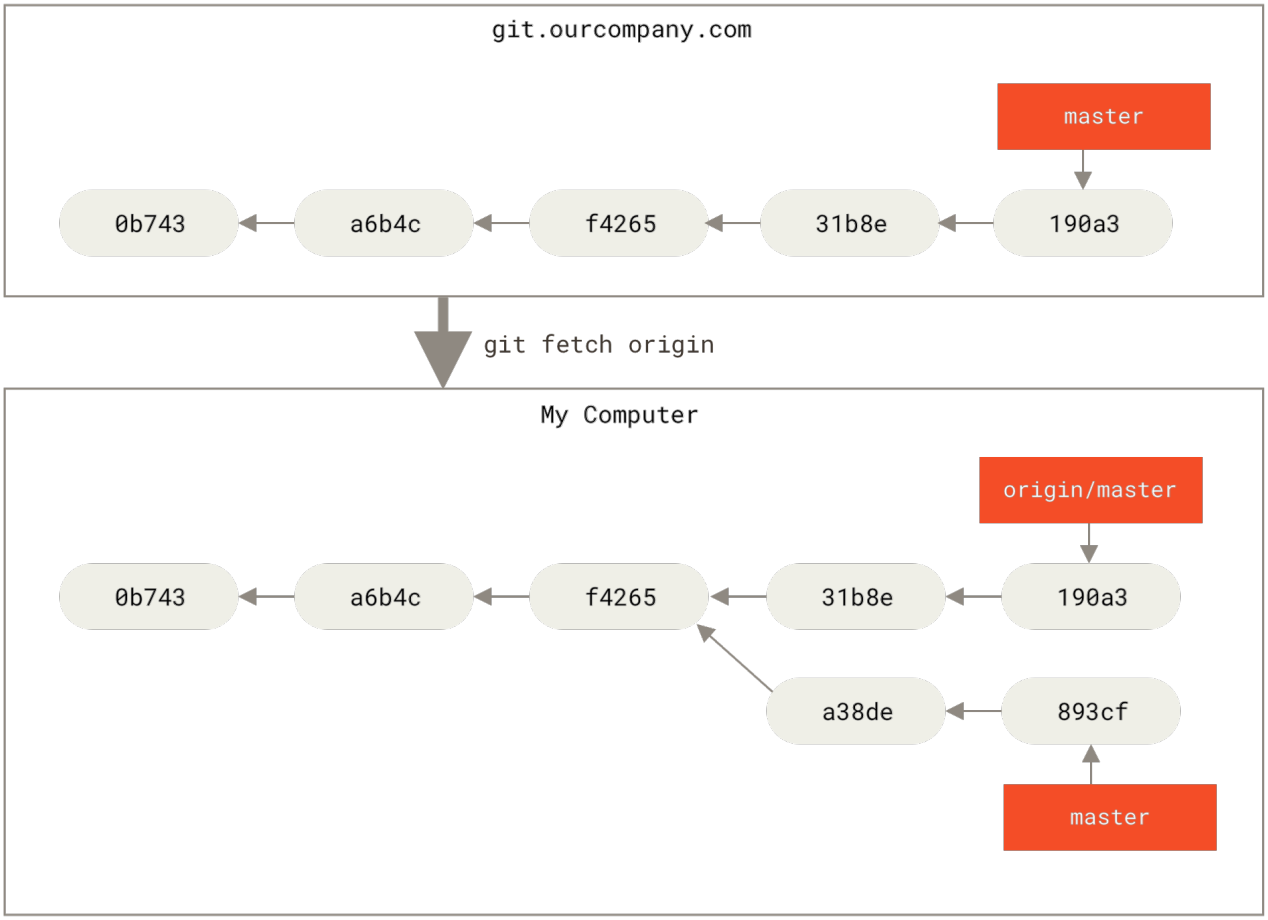


Рисунок 32. `git fetch` *обновляет ветки слежения*

Чтобы продемонстрировать, 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.ourcompany.com`. Вы можете добавить его в качестве новой удалённой ссылки для текущего проекта с помощью команды `git remote add`, как было описано в главе Основы Git. Назовите этот удалённый сервер `teamone` — это имя будет сокращением вместо полного URL.

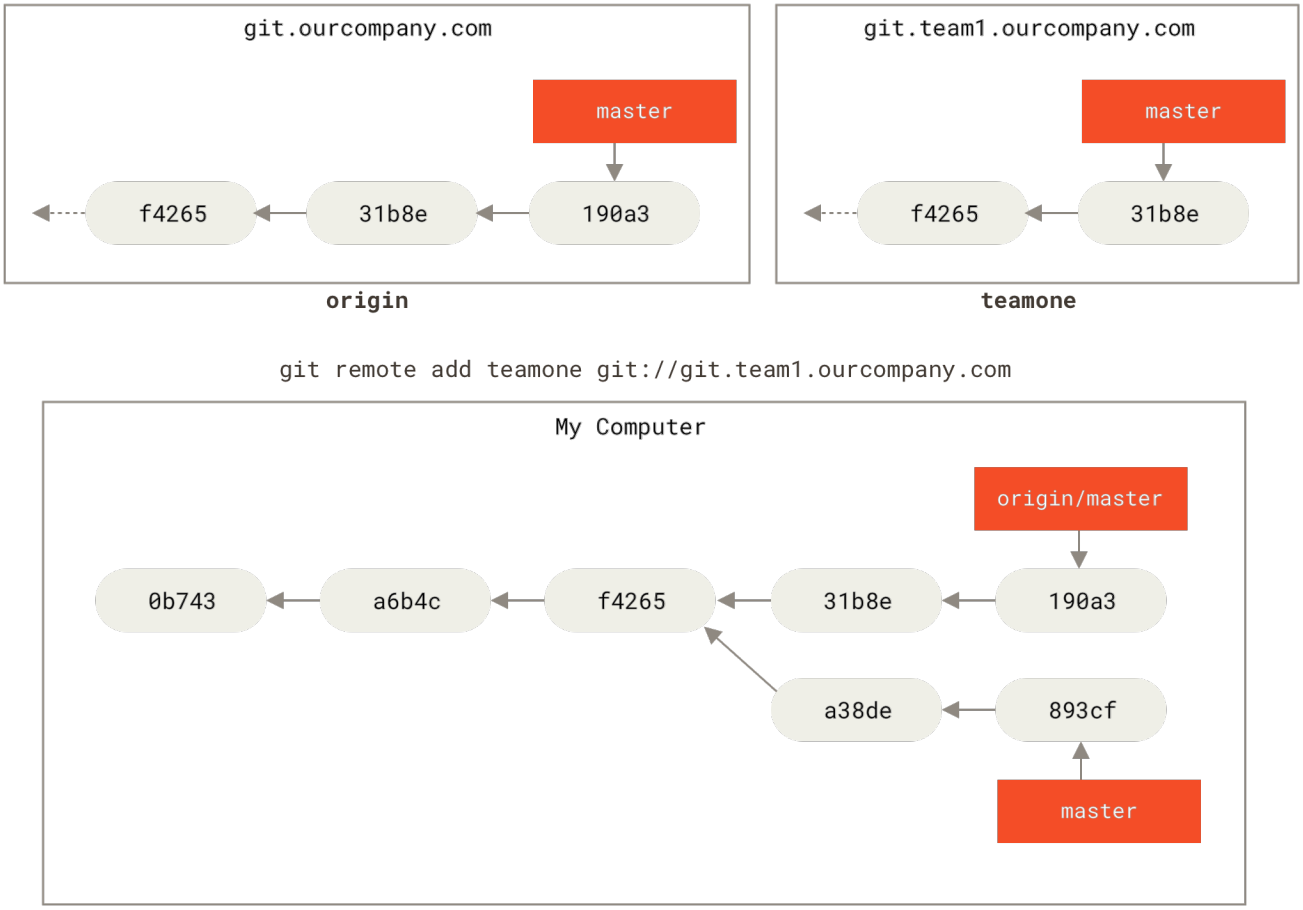


Рисунок 33. Добавление ещё одного сервера в качестве удалённой ветки

Теперь вы можете выполнить команду `git fetch teamone` для получения всех изменений с сервера `teamone`, которых у вас нет локально. Так как в данный момент на этом сервере есть только те данные, что содержит сервер `origin`, Git ничего не получит, но создаст ветку слежения с именем `teamone/master`, которая будет указывать на тот же коммит, что и ветка `master` на сервере `teamone`.

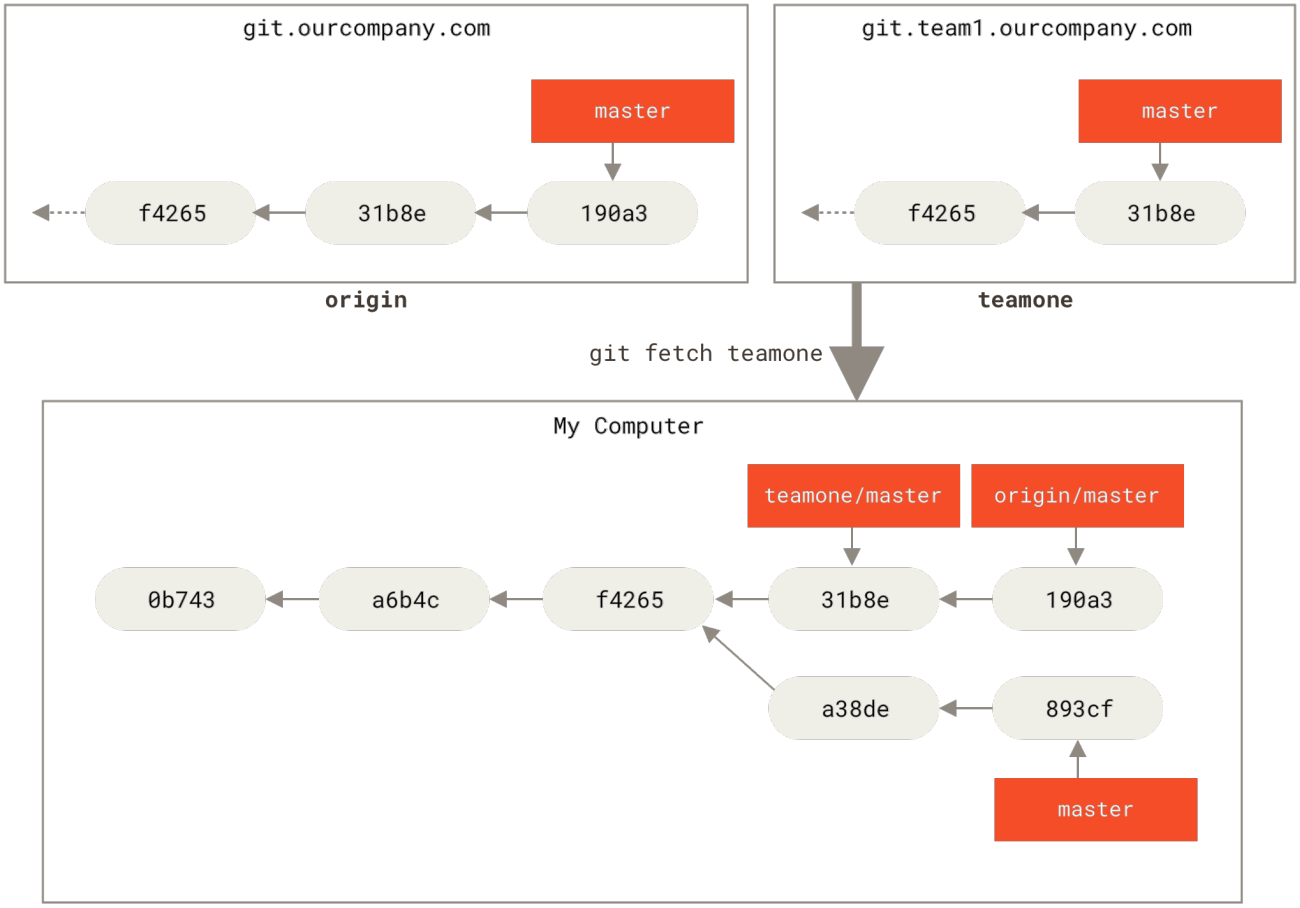


Рисунок 34. Ветка слежения `teamone/master`

Отправка изменений

Когда вы хотите поделиться веткой, вам необходимо отправить её на удалённый сервер, где у вас есть права на запись. Ваши локальные ветки автоматически не синхронизируются с удалёнными при отправке — вам нужно явно указать те ветки, которые вы хотите отправить. Таким образом, вы можете использовать свои личные ветки для работы, которую не хотите показывать, а 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 команду `git push <remote> <branch>`:

```
$ 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 рассмотрим часть с `refs/heads/` в главе Git изнутри, но обычно её можно пропустить. Вы также можете выполнить `git push origin serverfix:serverfix` — произойдёт то же самое — здесь говорится «возьми мою ветку `serverfix` и сделай её удалённой веткой `serverfix`». Можно использовать этот формат для отправки локальной ветки в удалённую ветку с другим именем. Если вы не хотите, чтобы на удалённом сервере ветка называлась `serverfix`, то вместо предыдущей команды выполните `git push origin serverfix:awesomebranch`, которая отправит локальную ветку `serverfix` в ветку `awesomebranch` удалённого репозитория.

Не вводите каждый раз свой пароль

Если вы используете HTTPS URL для отправки изменений, Git-сервер будет спрашивать имя пользователя и пароль для аутентификации. По умолчанию вам будет предложено ввести эти данные в терминале, чтобы сервер мог определить разрешена ли вам отправка изменений.

Если вы не хотите вводить свои данные каждый раз при отправке изменений, вы можете настроить «credential cache». Проще всего держать их в памяти несколько минут, это легко настроить с помощью команды `git config --global credential.helper cache`.

Для получения более подробной информации о различных вариантах кэша учётных данных обратитесь к разделу Хранилище учётных 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
```

Необходимо отметить, что при получении данных создаются ветки слежения, 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 изменять.

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

```
$ 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, в которой можно работать и которая будет начинаться там же, где и `origin/serverfix`.

Отслеживание веток

Получение локальной ветки из удалённой ветки автоматически создаёт то, что называется «веткой слежения» (а ветка, за которой следит локальная называется «upstream branch»). Ветки слежения — это локальные ветки, которые напрямую связаны с удалённой веткой. Если, находясь на ветке слежения, выполнить `git pull`, то Git уже будет знать с какого сервера получать данные и какую ветку использовать для слияния.

При клонировании репозитория, как правило, автоматически создаётся ветка `master`, которая следит за `origin/master`. Однако, при желании вы можете настроить отслеживание и других веток — следить за ветками на других серверах или отключить слежение за веткой `master`. Вы только что видели простейший пример, что сделать это можно с помощью команды `git checkout -b <branch> <remote>/<branch>`. Это часто используемая команда, поэтому Git предоставляет сокращённую форму записи в виде флага `--track`:

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

В действительности, это настолько распространённая команда, что существует сокращение для этого сокращения. Если вы пытаетесь извлечь ветку, которая не существует, но существует только одна удалённая ветка с точно таким же именем, то Git автоматически создаст ветку слежения:

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

Чтобы создать локальную ветку с именем, отличным от имени удалённой ветки, просто укажите другое имя:

```
$ 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 изменения из `origin/serverfix`.

Если у вас уже есть локальная ветка и вы хотите настроить её на слежение за удалённой веткой, которую вы только что получили, или хотите изменить используемую upstream-ветку, то воспользуйтесь параметрами `-u` или `--set-upstream-to` для команды `git branch`,

чтобы явно установить новое значение.

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

Сокращение Upstream

Если у вас настроена отслеживаемая ветка, вы можете ссылаться на неё с помощью сокращений `@{upstream}` или `@{u}`. Итак, если вы находитесь на ветке `master` и она следит за `origin/master`, при желании вы можете использовать `git merge @{u}` вместо `git merge origin/master`.

Если вы хотите посмотреть как у вас настроены ветки слежения, воспользуйтесь опцией `-vv` для команды `git branch`. Это выведет список локальных веток и дополнительную информацию о том, какая из веток отслеживается, отстаёт, опережает или всё сразу относительно отслеживаемой.

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] Add forgotten brackets
  master    1ae2a45 [origin/master] Deploy index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] This should do it
  testing   5ea463a Try something new
```

Итак, здесь мы видим, что наша ветка `iss53` следит за `origin/iss53` и «опережает» её на два изменения — это значит, что у нас есть два локальных коммита, которые не отправлены на сервер. Мы также видим, что наша ветка `master` отслеживает ветку `origin/master` и находится в актуальном состоянии. Далее мы можем видеть, что локальная ветка `serverfix` следит за веткой `server-fix-good` на сервере `teamone`, опережает её на три коммита и отстает на один — это значит, что на сервере есть один коммит, который мы ещё не слили, и три локальных коммита, которые ещё не отправлены на сервер. В конце мы видим, что наша ветка `testing` не отслеживает удалённую ветку.

Важно отметить, что эти цифры описывают состояние на момент последнего получения данных с каждого из серверов. Эта команда не обращается к серверам, а лишь говорит вам о том, какая информация с этих серверов сохранена в локальном кэше. Если вы хотите иметь актуальную информацию об этих числах, вам необходимо получить данные со всех ваших удалённых серверов перед запуском команды. Сделать это можно вот так:

```
$ git fetch --all; git branch -vv
```

Получение изменений

Команда `git fetch` получает с сервера все изменения, которых у вас ещё нет, но не будет изменять состояние вашей рабочей копии. Эта команда просто получает данные и позволяет вам самостоятельно сделать слияние. Тем не менее, существует команда `git pull`, которая в большинстве случаев является командой `git fetch`, за которой непосредственно

следует команда `git merge`. Если у вас настроена ветка слежения как показано в предыдущем разделе, или она явно установлена, или она была создана автоматически командами `clone` или `checkout`, `git pull` определит сервер и ветку, за которыми следит ваша текущая ветка, получит данные с этого сервера и затем попытается слить удалённую ветку.

Обычно, лучше 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 параметр `--delete` для команды `git push`. Для удаления ветки `serverfix` на сервере, выполните следующую команду:

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

Всё, что делает эта строка — удаляет указатель на сервере. Как правило, Git сервер хранит данные пока не запустится сборщик мусора, поэтому если ветка была удалена случайно, чаще всего её легко восстановить.

Перебазирование

В Git есть два способа внести изменения из одной ветки в другую: слияние и перебазирование. В этом разделе вы узнаете, что такое перебазирование, как его осуществлять и в каких случаях этот удивительный инструмент использовать не следует.

Простейшее перебазирование

Если вы вернётесь к более раннему примеру из Основы слияния, вы увидите, что разделили свою работу и сделали коммиты в две разные ветки.

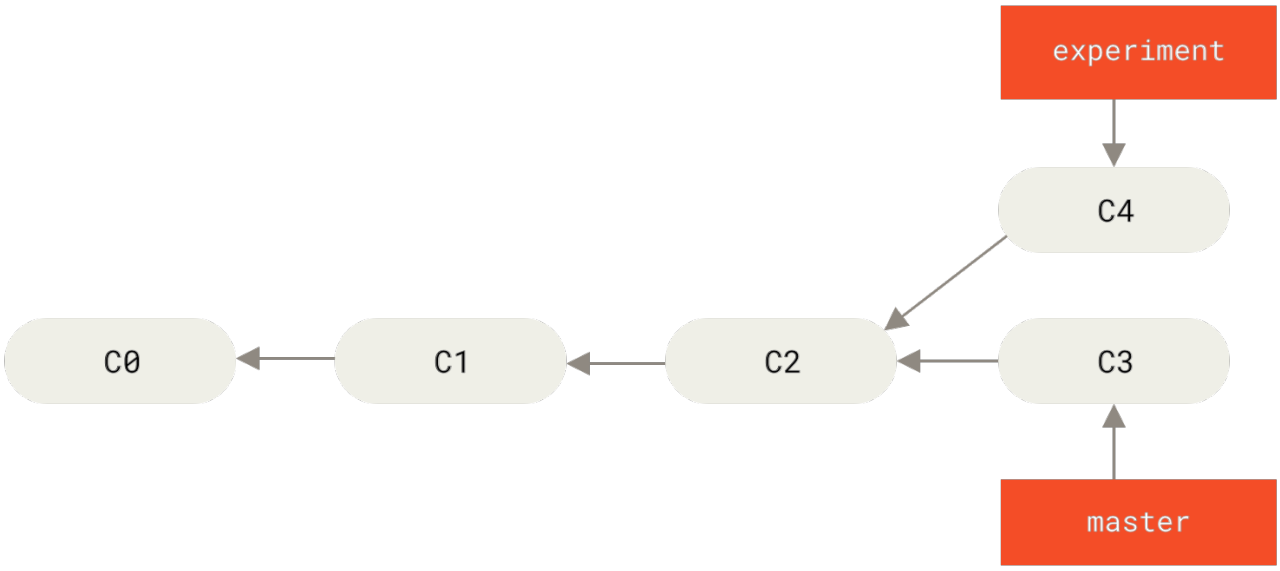


Рисунок 35. История коммитов простого разделения

Как мы выяснили ранее, простейший способ выполнить слияние двух веток — это команда `merge`. Она осуществляет трёхстороннее слияние между двумя последними снимками сливаемых веток (`C3` и `C4`) и самого недавнего общего для этих веток родительского снимка (`C2`), создавая новый снимок (и коммит).

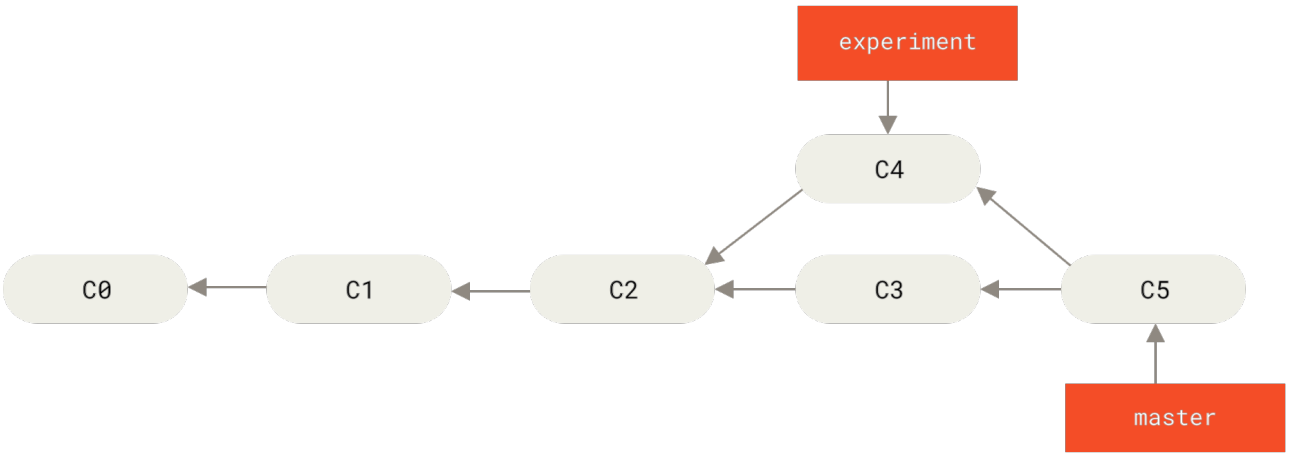


Рисунок 36. Слияние разделённой истории коммитов

Тем не менее есть и другой способ: вы можете взять те изменения, что были представлены в `C4`, и применить их поверх `C3`. В Git это называется *перебазированием*. С помощью команды `rebase` вы можете взять все коммиты из одной ветки и в том же порядке применить их к другой ветке.

В данном примере переключимся на ветку `experiment` и перебазируем её относительно ветки `master` следующим образом:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Это работает следующим образом: берётся общий родительский снимок двух веток (текущей, и той, поверх которой вы выполняете перебазирование), определяется дельта каждого коммита текущей ветки и сохраняется во временный файл, текущая ветка устанавливается на последний коммит ветки, поверх которой вы выполняете перебазирование, а затем по очереди применяются дельты из временных файлов.

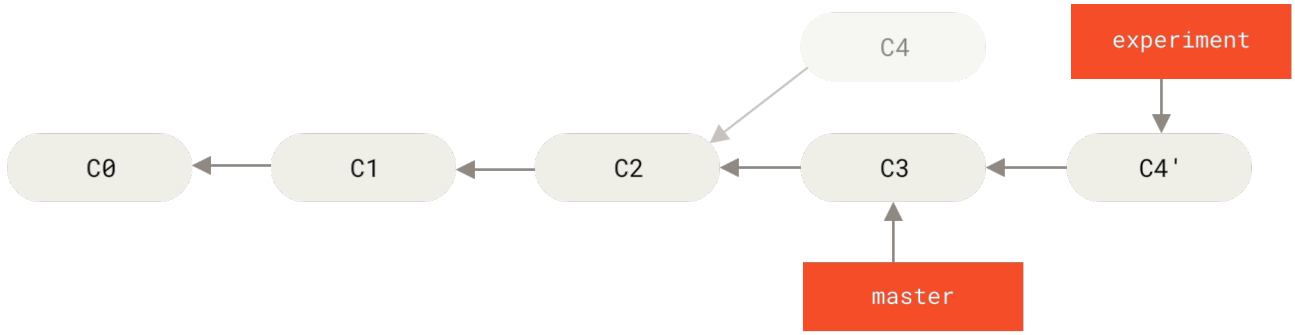


Рисунок 37. Перебазирование изменений из `C4` поверх `C3`

После этого вы можете переключиться обратно на ветку `master` и выполнить слияние перемоткой.

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

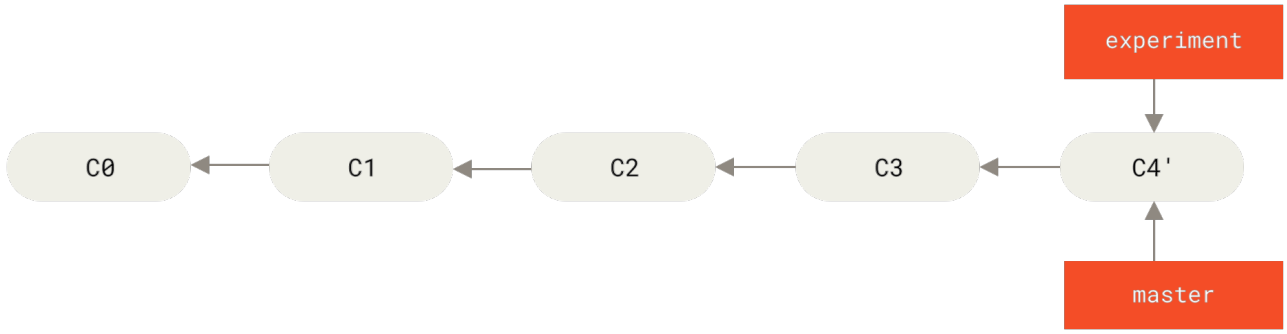


Рисунок 38. Перемотка ветки `master`

Теперь снимок, на который указывает `C4'` абсолютно такой же, как тот, на который указывал `C5` в примере с трёхсторонним слиянием. Нет абсолютно никакой разницы в конечном результате между двумя показанными примерами, но перебазирование делает историю коммитов чище. Если вы взглянете на историю перебазированной ветки, то увидите, что она выглядит абсолютно линейной: будто все операции были выполнены последовательно, даже если изначально они совершались параллельно.

Часто вы будете делать так для уверенности, что ваши коммиты могут быть бесконфликтно слиты в удалённую ветку — возможно, в проекте, куда вы пытаетесь внести вклад, но владельцем которого вы не являетесь. В этом случае вам следует работать в своей ветке и затем перебазировать вашу работу поверх `origin/master`, когда вы будете готовы отправить свои изменения в основной проект. Тогда владельцу проекта не придётся делать никакой лишней работы — всё решится простой перемоткой или бесконфликтным слиянием.

Учтите, что снимок, на который ссылается ваш последний коммит — является ли он последним коммитом после перебазирования или коммитом слияния после слияния — в обоих случаях это один и тот же снимок, отличаются только истории коммитов. Перебазирование повторяет изменения из одной ветки поверх другой в том порядке, в котором эти изменения были сделаны, в то время как слияние берёт две конечные точки и сливает их вместе.

Более интересные перемещения

Также возможно сделать так, чтобы при перебазировании воспроизведение коммитов применялось к совершенно другой ветке. Для примера возьмём История разработки с тематической веткой, ответвлённой от другой тематической ветки. Вы создаёте тематическую ветку `server`, чтобы добавить в проект некоторую функциональность для серверной части, и делаете коммит. Затем вы выполнили ответвление, чтобы сделать изменения для клиентской части, и создали несколько коммитов. Наконец, вы вернулись на ветку `server` и сделали ещё несколько коммитов.

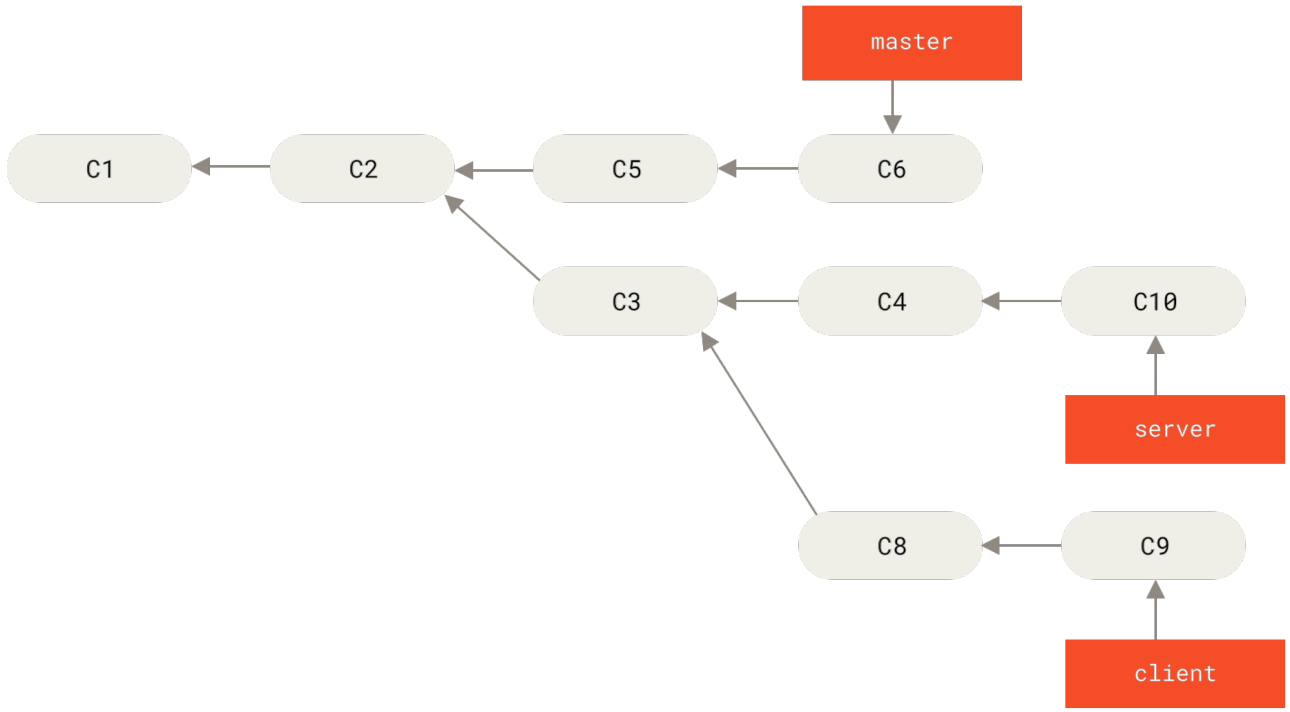


Рисунок 39. История разработки с тематической веткой, ответвлённой от другой тематической ветки

Предположим, вы решили, что хотите внести изменения клиентской части в основную линию разработки для релиза, но при этом не хотите добавлять изменения серверной части до полного тестирования. Вы можете взять изменения из ветки `client`, которых нет в `server` (`C8` и `C9`), и применить их на ветке `master` при помощи опции `--onto` команды `git rebase`:

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

В этой команде говорится: «Переключись на ветку `client`, найди изменения относительно ветки `server` и примени их для ветки `master`». Несмотря на некоторую сложность этого

способа, результат впечатляет.

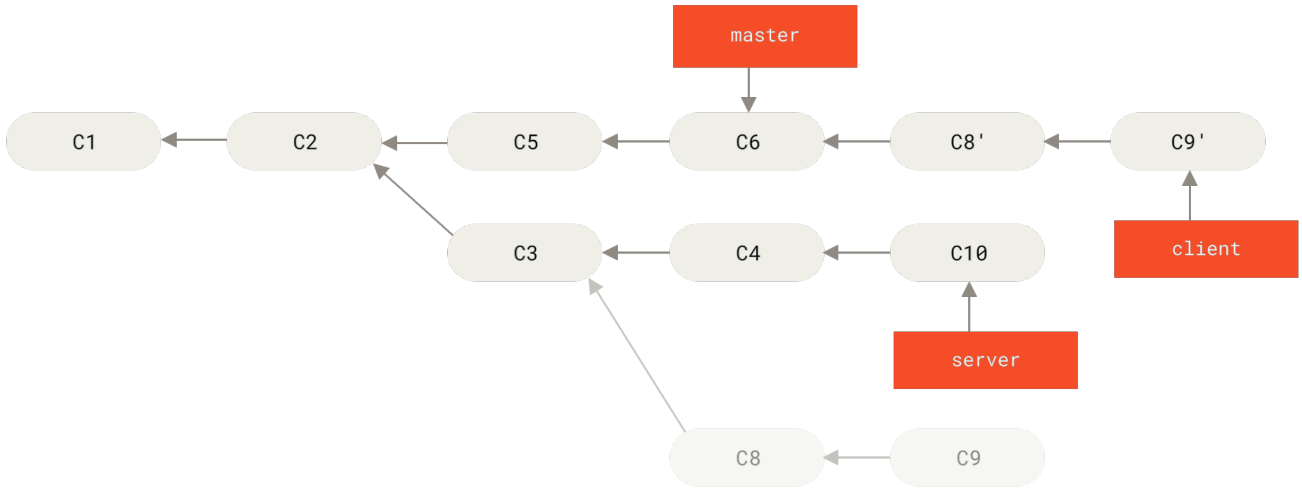


Рисунок 40. Перемещение тематической ветки, ответвлённой от другой тематической ветки

Теперь вы можете выполнить перемотку (fast-forward) для ветки `master` (см Перемотка ветки `master` для добавления изменений из ветки `client`):

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

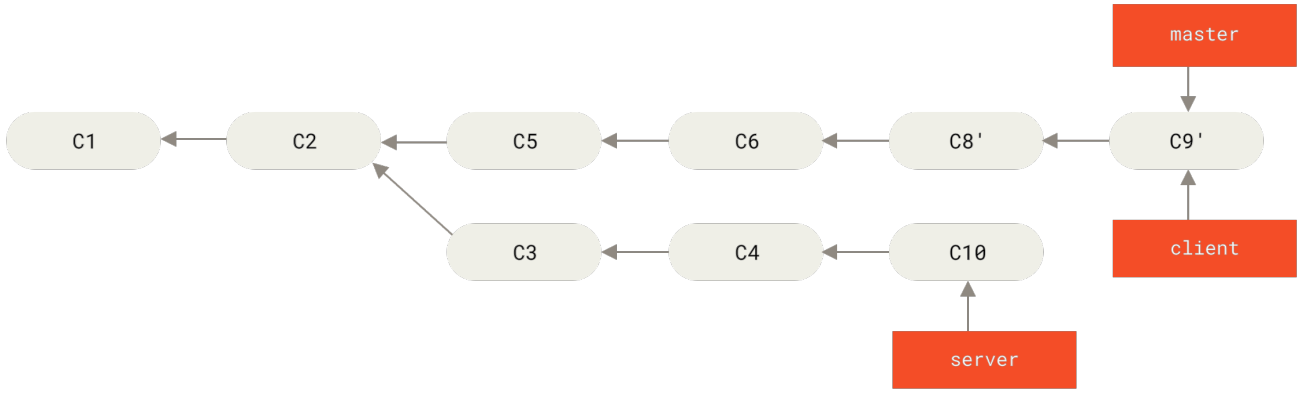


Рисунок 41. Перемотка ветки `master` для добавления изменений из ветки `client`

Представим, что вы решили добавить наработки и из ветки `server`. Вы можете выполнить перебазирование ветки `server` относительно ветки `master` без предварительного переключения на неё при помощи команды `git rebase <basebranch> <topicbranch>`, которая извлечёт тематическую ветку (в данном случае `server`) и применит изменения в ней к базовой ветке (`master`):

```
$ git rebase master server
```

Это повторит работу, сделанную в ветке `server` поверх ветки `master`, как показано на рисунке:

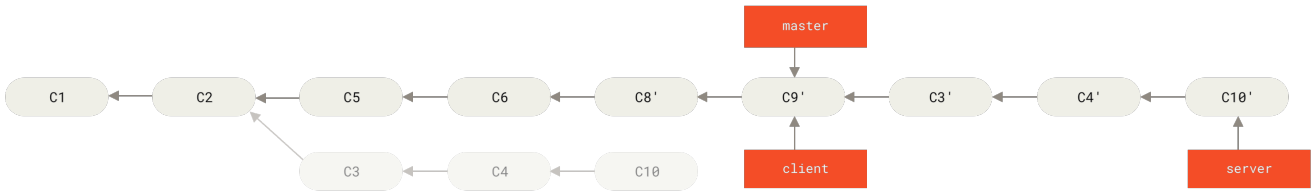


Рисунок 42. Перебазирование ветки `server` *на вершину ветки* `master`

После чего вы сможете выполнить перемотку основной ветки (`master`):

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

Теперь вы можете удалить ветки `client` и `server`, поскольку весь ваш прогресс уже интегрирован и тематические ветки больше не нужны, а полную историю вашего рабочего процесса отражает рисунок Окончательная история коммитов:

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

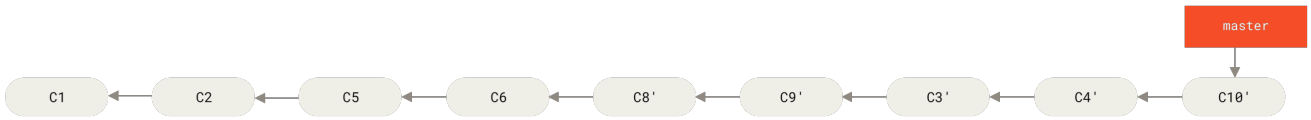


Рисунок 43. Окончательная история коммитов

Опасности перемещения

Но даже перебазирование, при всех своих достоинствах, не лишено недостатков, которые можно выразить одной строчкой:

Не перемещайте коммиты, уже отправленные в публичный репозиторий

Если вы будете придерживаться этого правила, всё будет хорошо. Если не будете, люди возненавидят вас, а ваши друзья и семья будут вас презирать.

Когда вы что-то перемещаете, вы отменяете существующие коммиты и создаёте новые, **похожие** на старые, но являющиеся другими. Если вы куда-нибудь отправляете свои коммиты и другие люди забирают их себе и в дальнейшем основывают на них свою работу, а затем вы переделываете эти коммиты командой `git rebase` и выкладываете их снова, то ваши коллеги будут вынуждены заново выполнять слияние для своих наработок. В итоге, когда вы в очередной раз попытаетесь включить их работу в свою, вы получите путаницу.

Давайте рассмотрим пример того, как перемещение публично доступных наработок может вызвать проблемы. Предположим, вы клонировали репозиторий с сервера и сделали какую-то работу. И ваша история коммитов выглядит так:

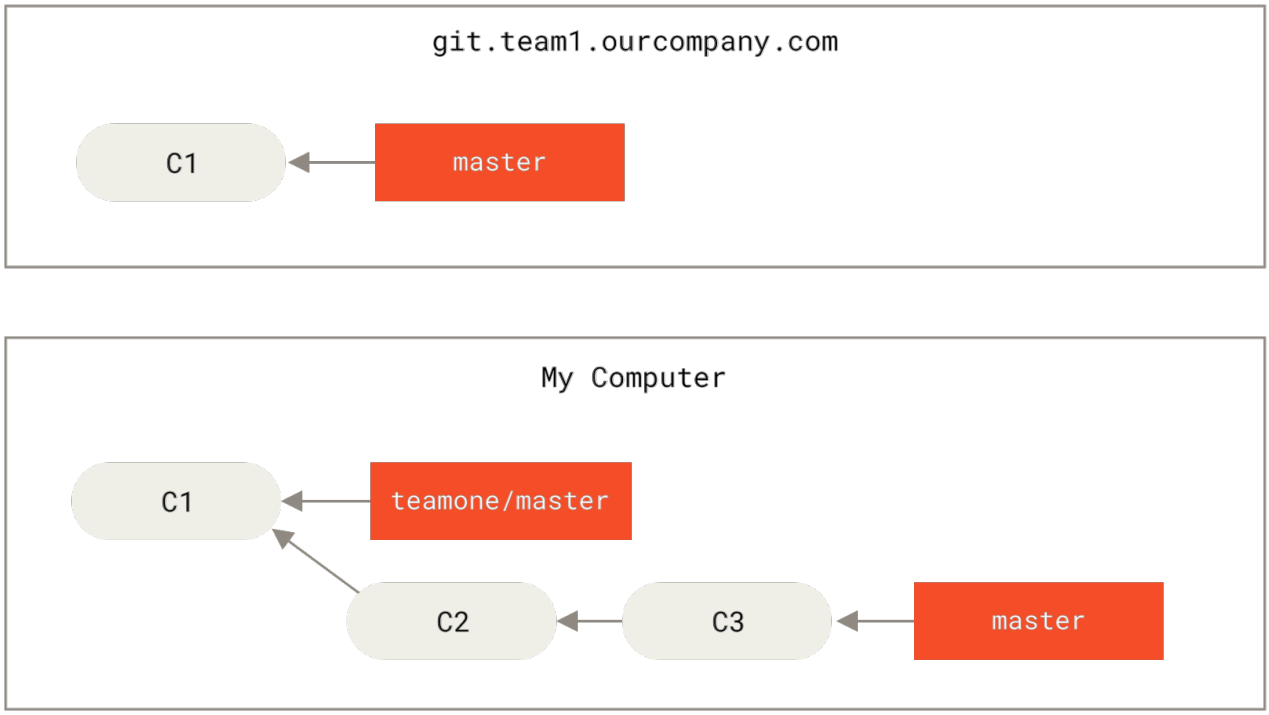


Рисунок 44. Клонирование репозитория и выполнение в нём какой-то работы

Теперь кто-то другой внёс свои изменения, слил их и отправил на сервер. Вы стягиваете их к себе, включая новую удалённую ветку, что изменяет вашу историю следующим образом:

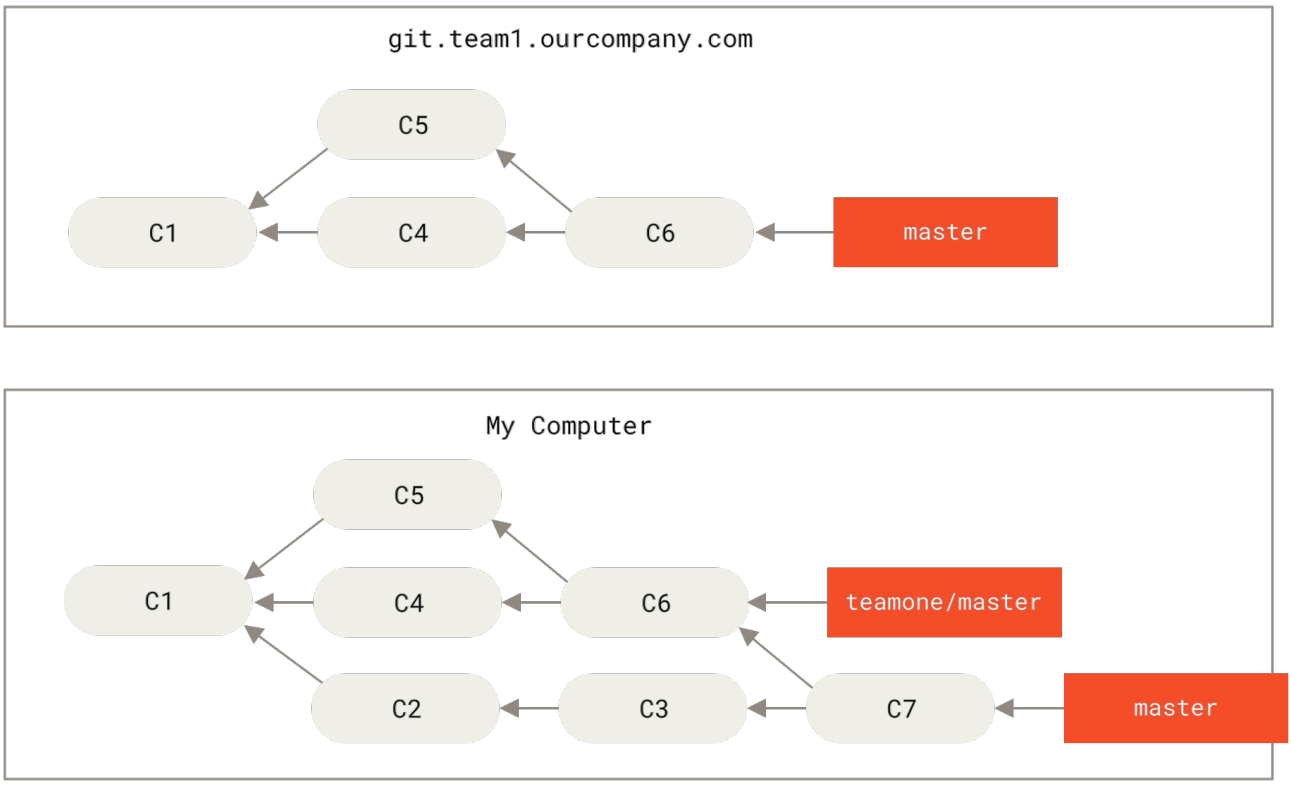


Рисунок 45. Извлекаем ещё коммиты и сливаем их со своей работой

Затем автор коммита слияния решает вернуться назад и перебазировать свою ветку; выполнив `git push --force`, он перезаписывает историю на сервере. При получении

изменений с сервера вы получите и новые коммиты.

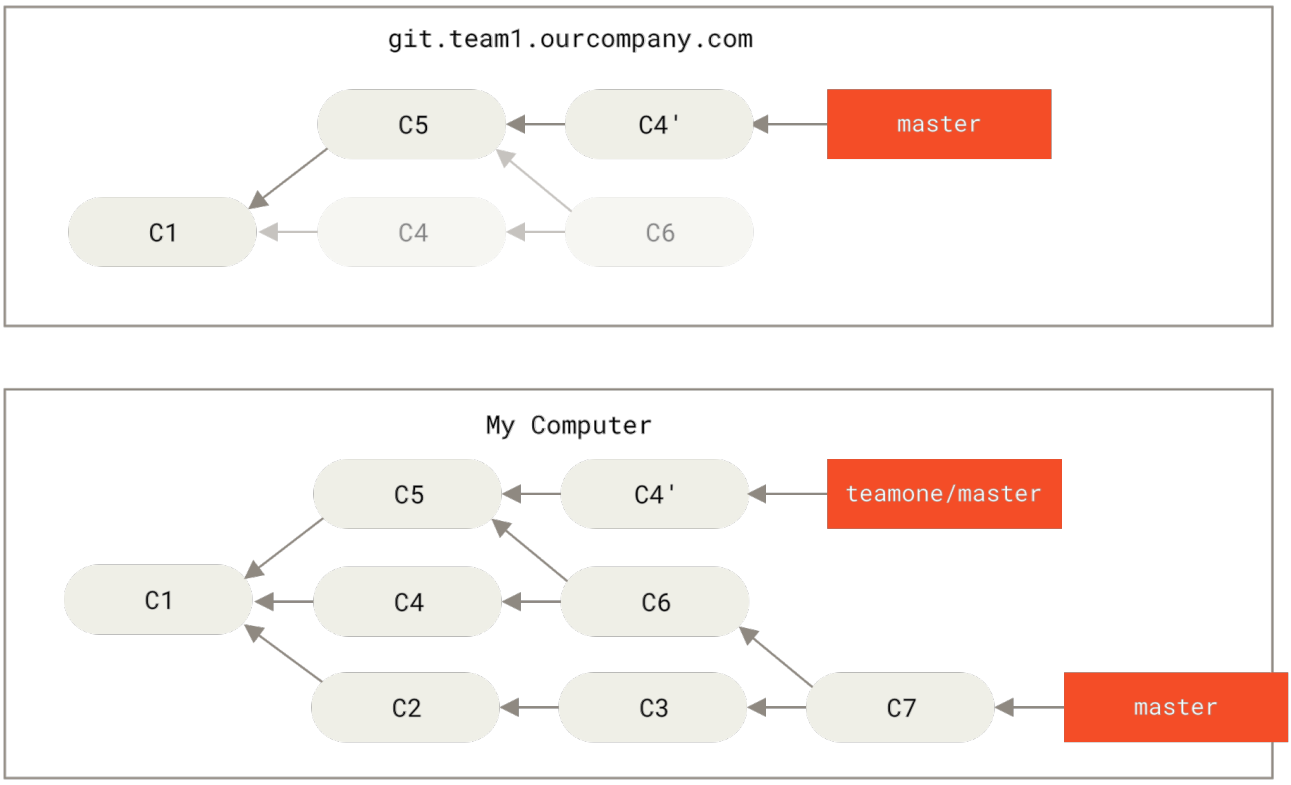


Рисунок 46. Кто-то выложил перебазированные коммиты, отменяя коммиты, на которых основывалась ваша работа

Теперь вы оба в неловком положении. Если вы выполните `git pull`, вы создадите коммит слияния, включающий обе линии истории, и ваш репозиторий будет выглядеть следующим образом:

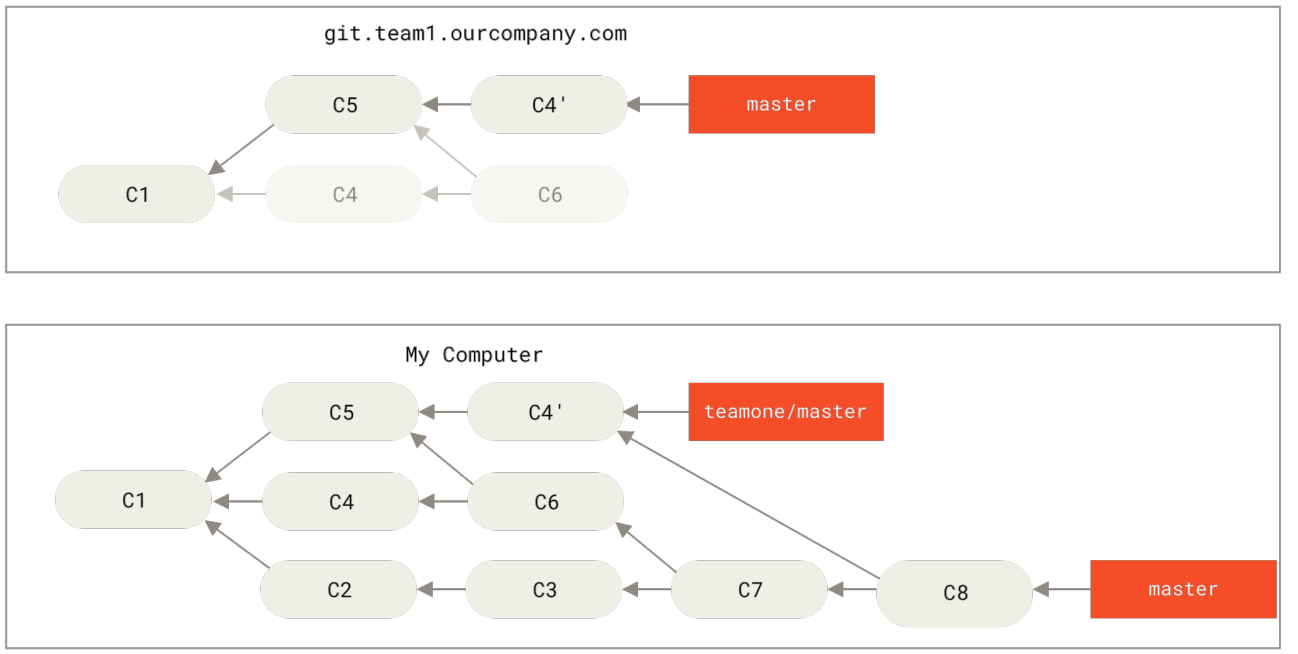


Рисунок 47. Вы снова выполняете слияние для той же самой работы в новый коммит слияния

Если вы посмотрите `git log` в этот момент, вы увидите два коммита с одинаковыми

авторами, датой и сообщением, что может сбить с толку. Помимо этого, если вы отправите свою историю на удалённый сервер в таком состоянии, вы вернёте все эти перебазированные коммиты на сервер, что ещё больше всех запутает. Логично предположить, что разработчик не хочет, чтобы `C4` и `C6` были в истории, и именно поэтому она перебазируется в первую очередь.

Меняя базу, меняй основание

Если вы попали в такую ситуацию, у Git есть особая магия чтобы вам помочь. Если кто-то в вашей команде форсирует отправку изменений на сервер, переписывающих работу, на которых базировалась ваша работа, то ваша задача будет состоять в определении того, что именно было ваше, а что было переписано **ими**.

Оказывается, что помимо контрольной суммы коммита SHA-1, Git также вычисляет контрольную сумму отдельно для патча, входящего в этот коммит. Это контрольная сумма называется «patch-id».

Если вы скачаете перезаписанную историю и перебазируете её поверх новых коммитов вашего коллеги, в большинстве случаев Git успешно определит, какие именно изменения были внесены вами, и применит их поверх новой ветки.

К примеру, если в предыдущем сценарии вместо слияния в Кто-то выложил перебазированные коммиты, отменяя коммиты, на которых основывалась ваша работа мы выполним `git rebase teamone/master`, Git будет:

- Определять, какая работа уникальна для вашей ветки (C2, C3, C4, C6, C7)
- Определять, какие коммиты не были коммитами слияния (C2, C3, C4)
- Определять, что не было перезаписано в основной ветке (только C2 и C3, поскольку C4 — это тот же патч, что и C4')
- Применять эти коммиты к ветке `teamone/master`

Таким образом, вместо результата, который мы можем наблюдать на Вы снова выполняете слияние для той же самой работы в новый коммит слияния, у нас получилось бы что-то вроде Перемещение в начало force-pushed перемещённой работы.

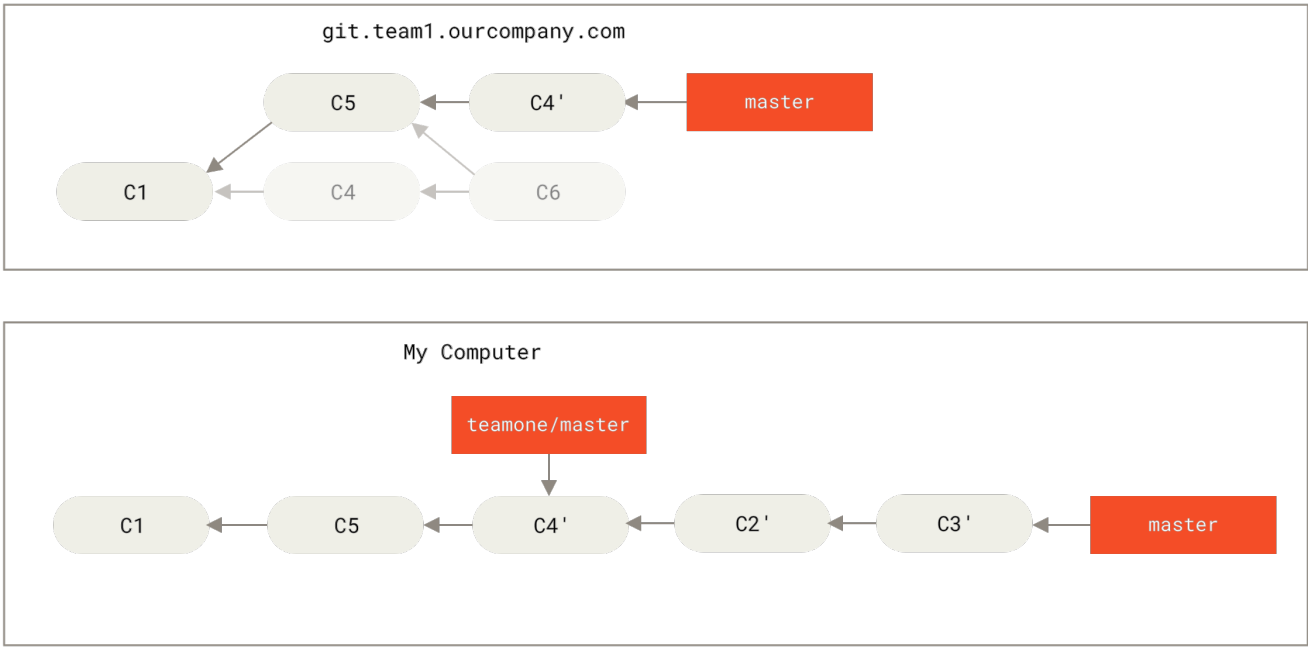


Рисунок 48. Перемещение в начало force-pushed перемещённой работы

Это возможно, если `C4` и `C4'` фактически являются одним и тем же патчем, который был сделан вашим коллегой. В противном случае `rebase` не сможет определить дубликат и создаст ещё один патч, подобный C4 (который с большой вероятностью не удастся применить чисто, поскольку в нём уже присутствуют некоторые изменения).

Вы можете это упростить, применив `git pull --rebase` вместо обычного `git pull`. Или сделать это вручную с помощью `git fetch`, а затем `git rebase teamone/master`.

Если вы используете `git pull` и хотите использовать `--rebase` по умолчанию, вы можете установить соответствующее значение конфигурации `pull.rebase` с помощью команды `git config --global pull.rebase true`.

Если вы рассматриваете перебазирование как способ наведения порядка и работаете с коммитами локально до их отправки или ваши коммиты никогда не будут доступны публично — у вас всё будет хорошо. Однако, если вы перемещаете коммиты, отправленные в публичный репозиторий, и есть вероятность, что работа некоторых людей основывается на этих коммитах, то ваши действия могут вызвать существенные проблемы, а вы — вызвать презрение вашей команды.

Если в какой-то момент вы или ваш коллега находите необходимость в этом, убедитесь, что все знают, как применять команду `git pull --rebase` для минимизации последствий от подобных действий.

Перемещение vs. Слияние

Теперь, когда вы увидели перемещение и слияние в действии, вы можете задаться вопросом, что из них лучше. Прежде чем ответить на этот вопрос, давайте вернёмся немного назад и поговорим о том, что означает история.

Одна из точек зрения заключается в том, что история коммитов в вашем репозитории — это

запись того, что на самом деле произошло. Это исторический документ, ценный сам по себе, и его нельзя подделывать. С этой точки зрения изменение истории коммитов практически кощунственно; вы *лжёте* о том, что на самом деле произошло. Но что, если произошла путаница в коммитах слияния? Если это случается, репозиторий должен сохранить это для потомков.

Противоположная точка зрения заключается в том, что история коммитов — это **история того, как был сделан ваш проект**. Вы не публикуете первый черновик книги или инструкции по поддержке вашего программного обеспечения, так как это нуждается в тщательном редактировании. Сторонники этого лагеря считают использование инструментов `rebase` и `filter-branch` способом рассказать историю проекта наилучшим образом для будущих читателей.

Теперь к вопросу о том, что лучше — слияние или перебазирование: надеюсь, вы видите, что это не так просто. Git — мощный инструмент, позволяющий вам делать многое с вашей историей, однако каждая команда и каждый проект индивидуален. Теперь, когда вы знаете, как работают оба эти приёма, выбор — какой из них будет лучше в вашей ситуации — зависит от вас.

При этом, вы можете взять лучшее от обоих миров: использовать перебазирование для наведения порядка в истории ваших локальных изменений, но никогда не применять его для уже отправленных куда-нибудь изменений.

Заключение

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 ветки на общий сервер, работать с другими людьми над общими ветками и перебазировать ваши ветки до того, как они станут доступны другим разработчикам. Далее мы поговорим о том, что вам необходимо для запуска собственного сервера с хостингом для Git-репозитория.

Git на сервере

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 удалённый 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 надёжного 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, а затем перейти к следующей главе, где мы обсудим различные аспекты работы с распределённой системой контроля версий.

Удалённый 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 — достаточно хранить 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*, для которого удалённый репозиторий — это 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 идеален, поскольку все копии вашего репозитория находятся на одном компьютере, что резко увеличивает вероятность потерять всё.

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 удалённого в существующий проект, используйте путь к репозиторию в качестве URL. Например, для клонирования локального репозитория вы можете выполнить что-то вроде этого:

```
$ git clone /srv/git/project.git
```

Или этого:

```
$ git clone file:///srv/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 жёсткие ссылки или копировать необходимые файлы. Если вы указываете `file://`, Git работает с данными так же, как при использовании сетевых протоколов, что снижает эффективность 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 (задачи по обслуживанию рассмотрены в главе Git изнутри). 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 /srv/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 репозиториев 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 экспортировать 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 находясь дома, вы должны смонтировать удалённый диск, что может оказаться сложнее и медленнее, чем доступ по сети.

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.

Наконец, этот протокол не защищает репозиторий от случайного повреждения. Все пользователи имеют доступ к «удалённому» каталогу и ничего не мешает изменению или удалению внутренних файлов Git и, как следствие, повреждению репозитория.

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. Сначала мы рассмотрим 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 аутентификацию по логину/паролю вместо установки SSH-ключей.

Наверное, сейчас он стал 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 целей, можно использовать один URL адрес для всего. Если вы пытаетесь отослать изменения и 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), можно 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

в корневой каталог HTTP и установить обработчик `post-update` (смотри Хуки в Git). Теперь каждый может клонировать репозиторий, если имеет доступ к веб-серверу, на котором он был размещен. Таким образом, чтобы открыть доступ на чтение к вашему репозиторию посредством 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 получение изменений и клонирование 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 главе Git изнутри приведены детали отдачи таких файлов).

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 аутентификации только при необходимости упрощает работу 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 публичный ключ 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 предоставлять 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 аутентифицированной отправки изменений, ввод учётных данных зачастую сложнее, чем при использовании SSH-ключей. Но есть несколько инструментов для кеширования учётных данных, включая Keychain access на OSX и Credential Manager на Windows, которые вы можете использовать для упрощения процесса. В разделе Хранилище учётных данных главы 7 рассказывается как настроить безопасное кэширование пароля в вашей системе.

Протокол SSH

Часто используемый транспортный протокол для самостоятельного 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 распространённости 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 предустановлены 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 благодаря максимальному сжатию данных 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 придётся иметь дело. Если же вам нужен

анонимный доступ на чтение для своих проектов, придётся 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 отдельный демон, для чего необходимо дополнительно настраивать `xinetd` или `systemd` или им подобные, 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 приводим команды и шаги, необходимые для базовой, упрощённой установки на Linux-сервер, но эти сервисы можно запустить и на MacOS 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 — репозиторий без рабочего каталога. Делается это просто. Чтобы создать новый голый репозиторий — во время клонирования используйте параметр `--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 эквивалентно команде:

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

Тут есть пара небольших различий в файле конфигурации, но в нашем случае эту разницу можно считать несущественной. В этом случае берётся репозиторий Git без рабочего каталога и помещается в отдельный каталог.

Размещение голого репозитория на сервере

Теперь, когда у вас есть голая копия вашего репозитория, осталось поместить её на сервер и настроить протоколы. Предположим, что вы уже настроили сервер `git.example.com`, имеете к нему доступ по SSH и хотите разместить все ваши репозитории Git в каталоге `/srv/git`. Считая, что `/srv/git` уже есть на сервере, вы можете добавить ваш новый репозиторий копированием голого репозитория:

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

Теперь другие пользователи, имеющие доступ к серверу по SSH и права на чтение каталога `/srv/git`, могут клонировать ваш репозиторий выполнив команду:

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

Если у пользователя есть права записи в каталог `/srv/git/my_project.git`, он автоматически получает возможность отправки изменений в репозиторий.

Git автоматически добавит права на запись в репозиторий для группы при запуске команды `git init` с параметром `--shared`. Следует отметить, что при запуске этой команды коммиты, ссылки и прочее удалены не будут.

```
$ ssh user@git.example.com
$ cd /srv/git/my_project.git
```

```
$ git init --bare --shared
```

Видите, как это просто, взять репозиторий Git, создать голую версию и поместить её 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 с возможностью доступа по 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 учётные 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 компания или вы только пробуете использовать Git в вашей организации и у вас небольшое число разработчиков, то всё достаточно просто. Один из наиболее сложных аспектов настройки сервера Git — это управление пользователями. Если вы хотите, чтобы некоторые репозитории были доступны определённым пользователям только на чтение, а остальным на чтение и запись, то настроить доступ и привилегии будет несколько сложнее.

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 сервере, где нет учётных записей для членов команды, которым требуются права на запись, то вы должны настроить доступ по SSH для них. Будем считать, что если у вас для этого есть сервер, то SSH-сервер на нём уже установлен и через него вы получаете доступ.

Есть несколько способов предоставить доступ всем участникам вашей команды. Первый — создать учётные записи для каждого, это просто, но может быть весьма обременительно. Вероятно, вы не захотите для каждого пользователя выполнять `adduser` (или `useradd`) и задавать временные пароли.

Второй способ — это создать на сервере пользователя `git`, попросить всех участников, кому требуется доступ на запись, прислать вам открытый ключ SSH и добавить эти ключи в файл `~/.ssh/authorized_keys` в домашнем каталоге пользователя `git`. Теперь все будут иметь доступ к этой машине используя пользователя `git`. Это никак не повлияет на данные в

коммите — пользователь, под которым вы соединяетесь с сервером по SSH, не воздействует на созданные вами коммиты.

Другой способ сделать это — настроить SSH сервер на использование аутентификации через LDAP-сервер или любой другой имеющийся у вас централизованный сервер аутентификации. Вы можете использовать любой механизм аутентификации на сервере и считать что он будет работать для Git, если пользователь может получить доступ к консоли по SSH.

Генерация открытого SSH ключа

Как отмечалось ранее, многие Git-серверы используют аутентификацию по открытым SSH-ключам. Для того чтобы предоставить открытый ключ, каждый пользователь в системе 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. По умолчанию пользовательские SSH ключи сохраняются в каталоге `~/.ssh` домашнем каталоге пользователя. Вы можете легко проверить наличие ключа перейдя в этот каталог и посмотрев его содержимое:

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

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

```
$ ssh-keygen -o
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 программа попросит указать расположение файла для сохранения ключа (`.ssh/id_rsa`), затем дважды ввести пароль для шифрования. Если вы не хотите вводить пароль каждый раз при использовании ключа, то можете оставить его пустым или использовать программу `ssh-agent`. Если вы решили использовать пароль для приватного ключа, то настоятельно рекомендуется использовать опцию `-o`, которая позволяет сохранить ключ в формате, более устойчивом ко взлому методом подбора, чем стандартный

формат.

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

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

Более подробное руководство по созданию SSH-ключей и конфигурации клиента на различных системах вы можете найти в руководстве GitHub.

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.

Вместо ручного копирования и установки открытых ключей, многое из описанного ниже может быть автоматизировано за счёт использования команды `ssh-copy-id`.

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 открытые SSH-ключи разработчиков в файл `authorized_keys` пользователя `git`. Предположим, у вас уже есть несколько таких ключей и вы сохранили их во временные файлы. Напомним, открытый ключ выглядит примерно так:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
```

```
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

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

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

После этого Джон, Джози или Джессика могут отправить первую версию их проекта в этот репозиторий, добавив его как удалённый и отправив соответствующую 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 и вы создали DNS запись для `gitserver`, указывающую на этот сервер, то можно использовать следующие команды как есть (считая что `myproject` это существующий проект с файлами):

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

Теперь все остальные могут клонировать его и отправлять в него изменения:

```
$ git clone git@gitserver:/srv/git/project.git
$ cd project
$ vim README
$ git commit -am 'Fix for 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 `/etc/passwd`.

Вы можете легко ограничить пользователя `git` только действиями, связанными с Git, с помощью ограниченной оболочки `git-shell`, поставляемой вместе с Git. Если указать её в качестве командного интерпретатора для пользователя `git`, то он не сможет получить доступ к обычной командной оболочке на вашем сервере. Для её использования, укажите `git-shell` вместо bash или csh для пользователя `git`. 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 -e /etc/shells  # и добавим путь к `git-shell` из предыдущей команды
```

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

```
$ sudo chsh git -s $(which git-shell)
```

Теперь пользователь `git` может использовать SSH соединение только для работы с репозиториями Git и не может зайти на машину. Если вы попробуете войти в систему, то вход будет отклонён:

```
$ 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.
```

На текущий момент пользователи всё ещё могут использовать перенаправление порта SSH для доступа к другим Git серверам, к которым текущий может подключиться. Если это нужно отключить, вы можете добавить следующие опции в файл `authorized_keys` перед теми ключами, для которых нужно применить это ограничение:

```
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty
```

В результате файл будет выглядеть следующим образом:

```
$ cat ~/.ssh/authorized_keys
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4LojG6rs6h
PB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4kYjh6541N
YsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9EzSdfd8AcC
IicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myivO7TCUSBd
LQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPqdAv8JggJ
ICUvax2T9va5 gsg-keypair
```

```
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQDEwENNMomTboYI+LJieaAY16qiXiH3wuvENhBG...
```

Теперь сетевые команды Git будут работать, но пользователи не смогут заходить на сервер. Вы также можете создать подкаталог в домашнем каталоге 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. Для получения дополнительной информации по настройке оболочки запустите команду `git help shell`.

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=/srv/git/ /srv/git/
```

Опция `--reuseaddr` позволит серверу перезапуститься без ожидания таймаута существующих подключений, `--base-path` позволит людям не указывать полный путь, чтобы клонировать проект, а путь в конце указывает демону Git где искать экспортируемые 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.

Так как `systemd` является самой распространённой системой инициализации в современных дистрибутивах Linux, вы можете использовать именно её. Просто создайте файл в каталоге `/etc/systemd/system/git-daemon.service` со следующим содержанием:

```
[Unit]
Description=Start Git Daemon

[Service]
ExecStart=/usr/bin/git daemon --reuseaddr --base-path=/srv/git/ /srv/git/
```

```
Restart=always
RestartSec=500ms

StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=git-daemon

User=git
Group=git

[Install]
WantedBy=multi-user.target
```

Как вы могли здесь заметить, Git демон запускается от имени `git`, как пользователя, так и группы. При необходимости укажите другие значения и убедитесь, что указанный пользователь существует в системе. Так же убедитесь, что исполняемый файл Git имеет путь `/usr/bin/git` или укажите соответствующий путь к нему.

Наконец, выполните команду `systemctl enable git-daemon` для запуска сервиса при старте системы; для ручного запуска и остановки сервиса используйте команды `systemctl start git-daemon` и `systemctl stop git-daemon` соответственно.

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

Затем нужно указать Git серверу к каким репозиториям предоставлять доступ без аутентификации. 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 указывает Git, 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 на сервер CGI-скрипта `git-http-backend`, поставляемого вместе с Git. 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 верно 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 (что делает операции чтения обратно совместимыми 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 необходимые для корректной работы модули `mod_cgi`, `mod_alias` и `mod_env`.

Так же необходимо установить Unix пользователя и группу для каталога `/srv/git` в значение `www-data`, чтобы позволить веб-серверу читать из и писать в репозитории, потому что процесс Apache, запускающий CGI скрипт, работает от имени этого пользователя:

```
$ chgrp -R www-data /srv/git
```

Затем добавим некоторые настройки в конфигурационный файл Apache, чтобы запускать `git-http-backend` как обработчик для всех запросов, содержащих `/git`.

```
SetEnv GIT_PROJECT_ROOT /srv/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/lib/git-core/git-http-backend/
```

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

Наконец, нужно разрешить Apache обрабатывать запросы к `git-http-backend`, при этом запросы на запись должны быть авторизованы, для этого можно использовать вот такой блок конфигурации:

```
<Files "git-http-backend">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /srv/git/.htpasswd
    Require expr !(%{QUERY_STRING} -strmatch '*service=git-receive-pack*' ||
%{REQUEST_URI} =~ m#/git-receive-pack$#)
    Require valid-user
</Files>
```

Это потребует создания файла `.htpasswd`, содержащего пароли всех пользователей. Например, добавление пользователя «schacon» в этот файл делается так:

```
$ htpasswd -c /srv/git/.htpasswd schacon
```

Существует множество способов аутентифицировать пользователей в Apache, вам нужно

выбрать и применить хотя бы один из них. Мы привели простейший пример. Скорее всего вы ещё захотите настроить SSL для шифрования трафика.

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 берёт на себя согласование передачи и приёма данных по HTTP. Сам по себе, он не реализует аутентификации, но это легко настраивается на уровне веб-сервера, который его запускает. Вы можете сделать это практически на любом веб-сервере с поддержкой CGI, так что используйте тот, который знаете лучше всего.

За дополнительной информацией по настройке аутентификации в Apache обратитесь к документации: https://httpd.apache.org/docs/current/howto/auth.html

GitWeb

Теперь, когда к вашему проекту настроен доступ на чтение/запись и только на чтение, вероятно, вы захотите настроить простой веб-визуализатор. Для этой цели в комплекте с Git поставляется CGI-сценарий GitWeb.

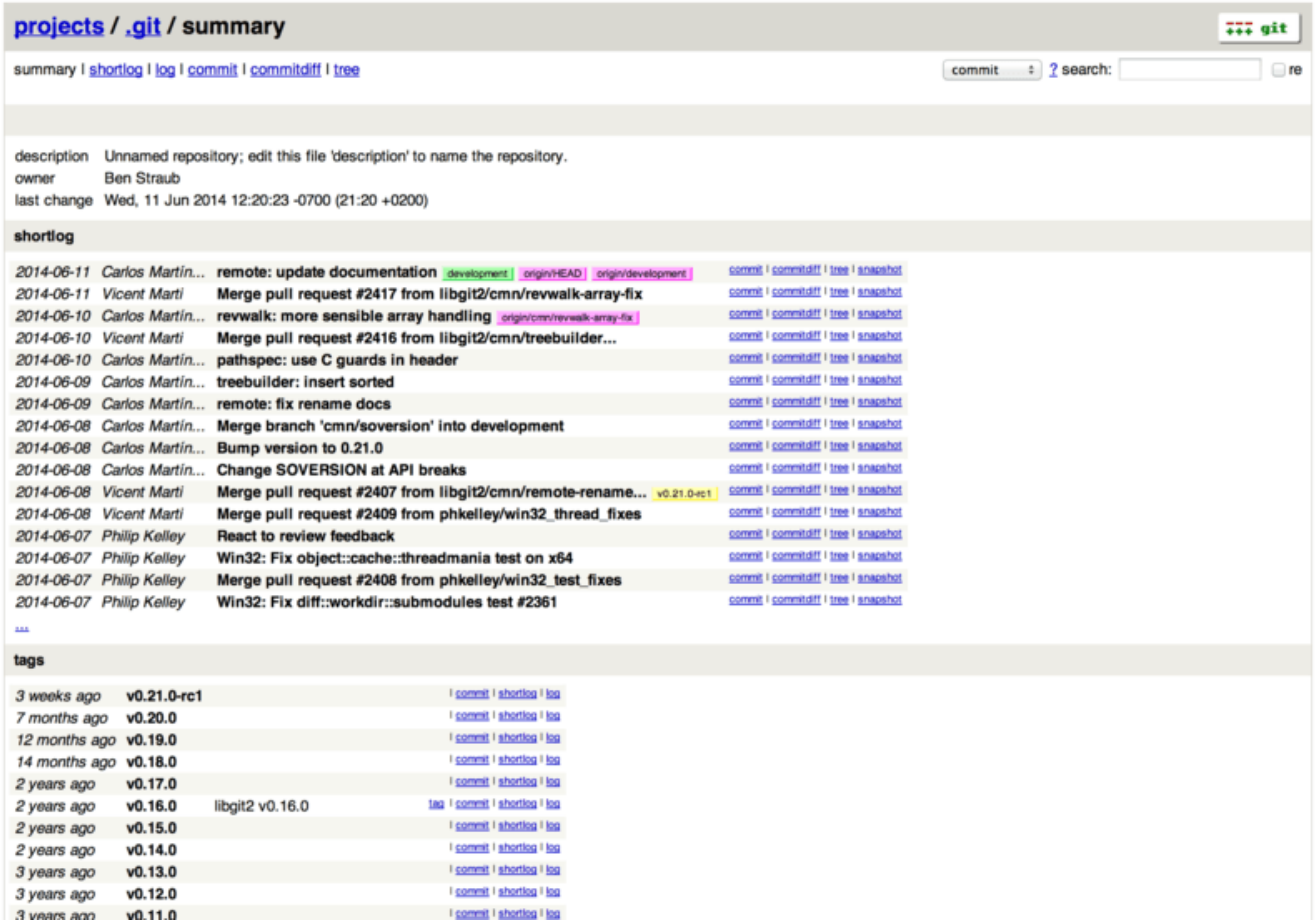


Рисунок 49. Веб-интерфейс GitWeb

Если вы хотите посмотреть как будет выглядеть ваш проект в GitWeb, в Git есть стандартная команда для запуска временного экземпляра, однако она требует наличия установленного веб-сервера, такого как `lighttpd` или `webrick`. Как правило, на машинах с Linux `lighttpd` уже

установлен, поэтому вы сможете его запустить, выполнив команду `git instaweb` в каталоге с вашим проектом. Если вы используете Mac, Leopard поставляется с предустановленным Ruby, поэтому `webrick` может быть лучшим выбором. Чтобы запустить `instaweb` не с ligttpd используйте параметр `--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, а затем откроет эту страницу в веб-браузере. Как видите, нет ничего сложного. Когда вы закончили и хотите остановить сервер, запустите 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 предоставлять хостинг для проекта с открытым кодом, вам необходимо подключить CGI-сценарий на вашем веб-сервере. В некоторых дистрибутивах Linux есть пакет `gitweb`, который вы можете установить, используя `apt` или `dnf`, так что вы можете попробовать сначала этот способ. Мы же вкратце рассмотрим ручную установку GitWeb. Для начала вам нужно скачать 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="/srv/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 функциональность 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 документирован и поддерживается. GitLab настоятельно рекомендует установить GitLab на ваш сервер через официальный пакет Omnibus GitLab.

Другие варианты установки:

- GitLab Helm chart для использования с Kubernetes.
- Официальные образы GitLab для использования с Docker.
- Из исходных файлов.
- Облачный провайдер, такой как AWS, Google Cloud Platform, Azure, OpenShift или Digital Ocean.

Для получения дополнительной информации прочтите GitLab Community Edition (CE) readme.

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.

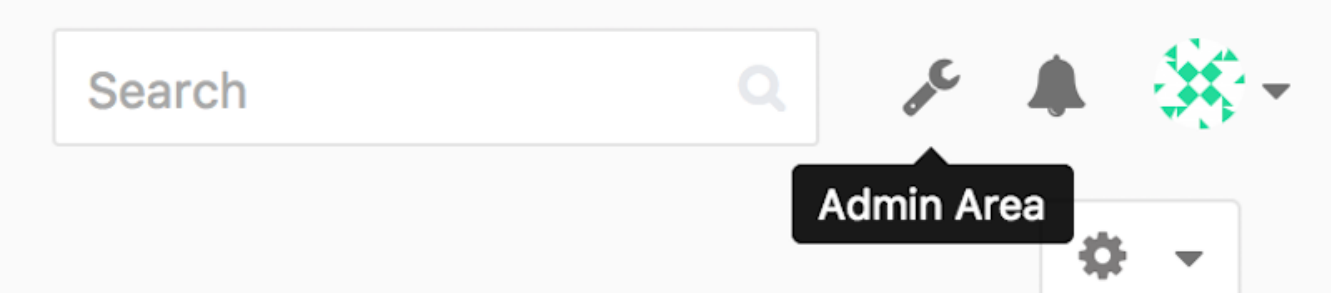


Рисунок 50. Пункт «Административная зона» в меню GitLab

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`.

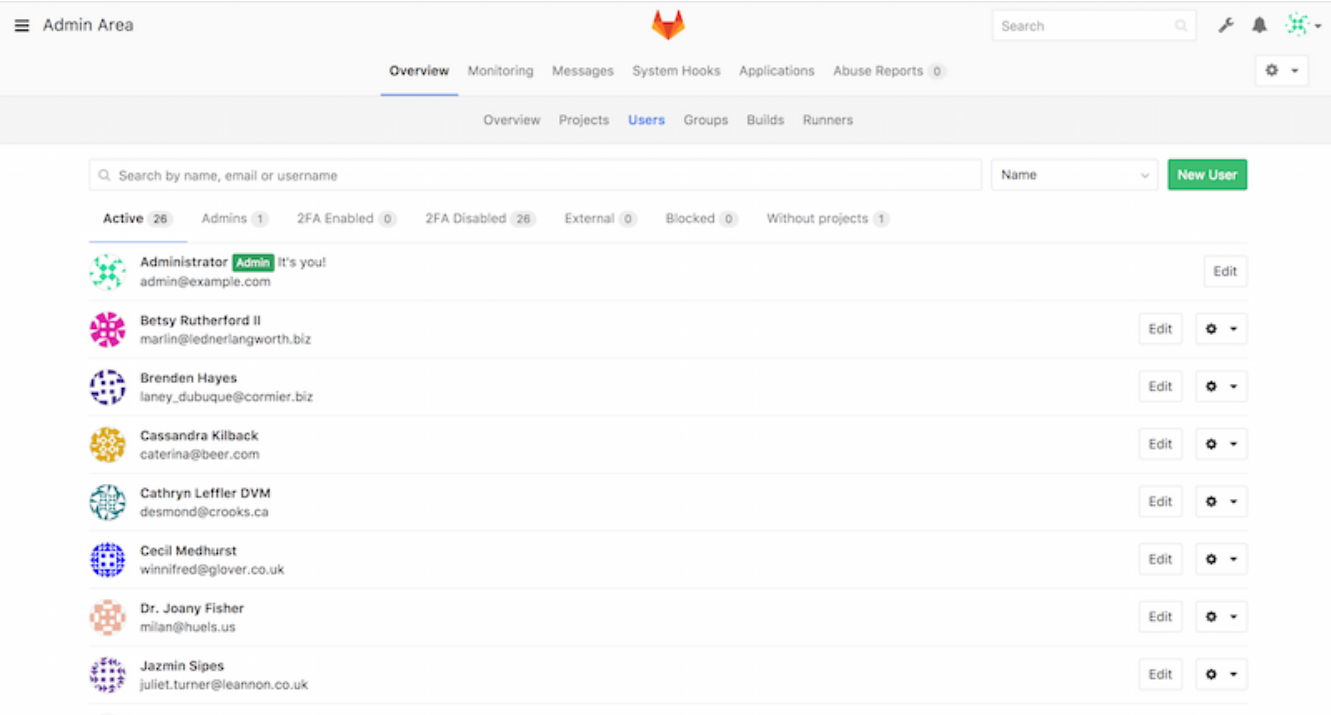


Рисунок 51. Экран управления пользователями GitLab

Удаление пользователя может быть выполнено двумя способами. «Блокирование» («Blocking») пользователя запрещает ему вход в GitLab, но все данные в его пространстве имён 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 имён 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`.

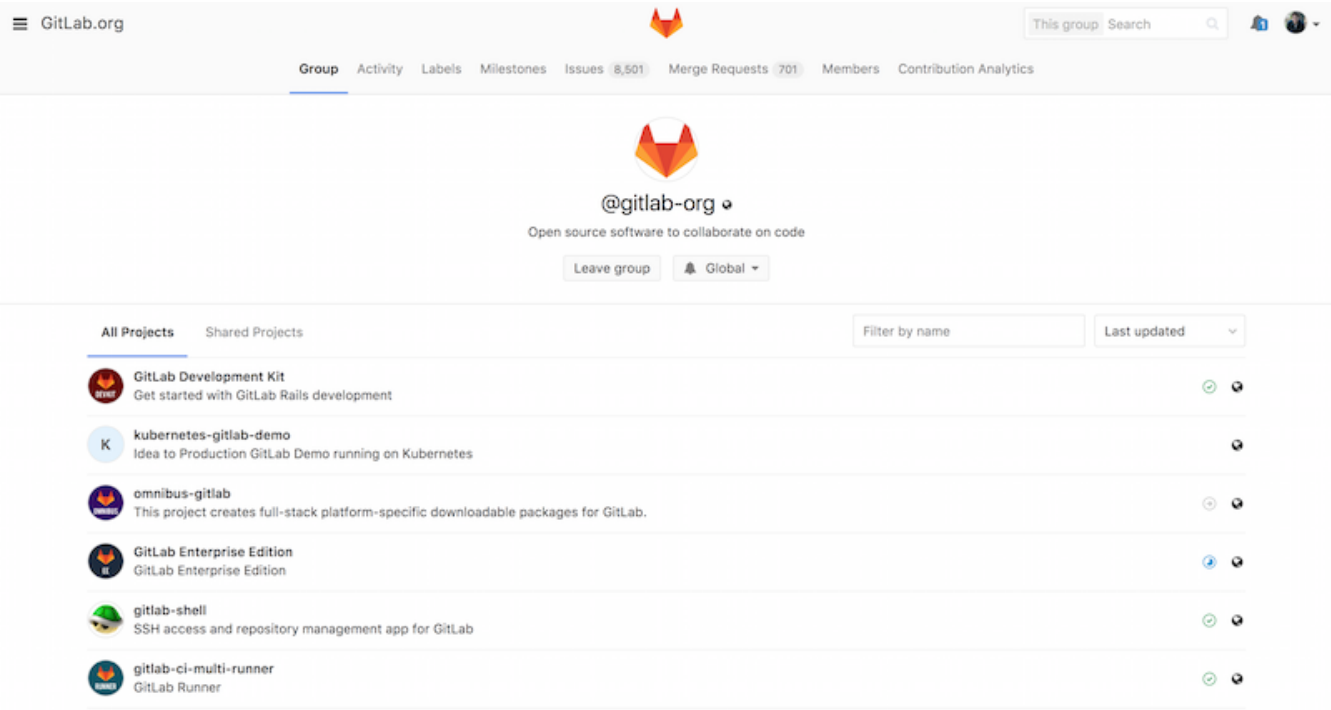


Рисунок 52. Экран управления группами GitLab

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 нескольким 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 беспрепятственно 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: обычно на таких сайтах можно быстро настроить и запустить проект, а так же не требуется никакого мониторинга или поддержки сервера. Даже если вы установили и запустили свой собственный внутренний сервер, возможно, вы захотите использовать сайт на публичном хостинге для ваших проектов с открытым кодом — так сообществу будет проще вас найти и помочь.

В наши дни у вас есть огромный выбор вариантов хостинга, каждый из которых имеет свои преимущества и недостатки. Актуальный список приведён в главной вики Git на странице GitHosting: https://git.wiki.kernel.org/index.php/GitHosting

Мы детально рассмотрим GitHub в главе GitHub, так как это крупнейший Git-хостинг и вам скорее всего понадобится взаимодействовать с проектами, хостящимися на нём; при этом существуют десятки других, что делает необязательным использование собственного сервера.

Заключение

Существует несколько вариантов использования 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 подойдёт вам и вашей организации, не должен вызвать затруднений.

Распределённый 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 для локальной работы, мы рассмотрим, как задействовать некоторые распределённые рабочие процессы, предлагаемые Git.

В этой главе мы рассмотрим работу с Git 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 которых участвует множество разработчиков.

Распределённый рабочий процесс

В отличие от централизованных систем контроля версий, распределённая природа Git позволяет более гибко взаимодействовать разработчикам в рамках проекта. В централизованных системах каждый разработчик представляет собой узел, который более или менее одинаково взаимодействует с центральным хабом. Однако, в Git каждый разработчик это и узел и хаб, то есть каждый разработчик может как отправлять код в другие репозитории, так и поддерживать публичный репозиторий, в который другие разработчики смогут отправлять свой код, взяв его за основу. Это предоставляет огромное количество вариаций для организации рабочего процесса над проектом и/или для команды, поэтому мы расскажем об основных парадигмах, отражающих преимущество гибкости Git. Так же мы рассмотрим сильные и слабые стороны каждой из них; вы сможете выбрать наиболее подходящую или позаимствовать необходимые функции сразу из нескольких.

Централизованная работа

В централизованных системах, как правило, присутствует только одна модель взаимодействия — централизованный рабочий процесс. Центральный хаб или *репозиторий* может принимать код, а все остальные синхронизируют свою работу с ним. Все разработчики являются узлами (пользователями хаба) и синхронизируются только с ним.

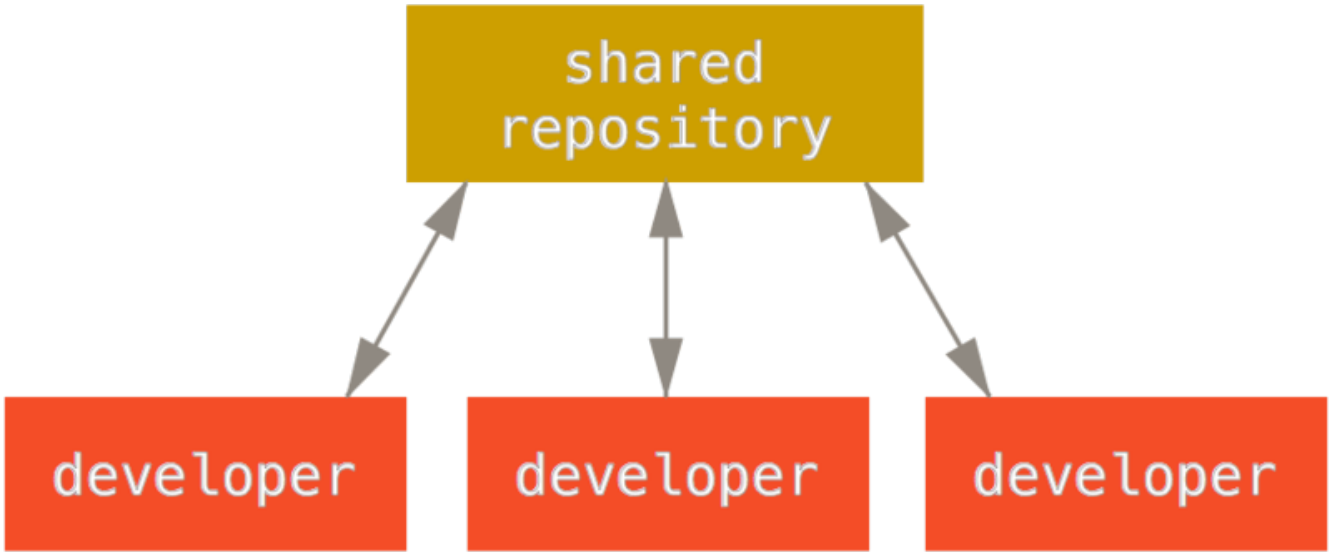


Рисунок 53. Централизованный рабочий процесс

Это означает, что если два разработчика клонируют репозиторий и каждый внесёт изменения, то первый из них сможет отправить свои изменения в репозиторий без проблем. Второй разработчик должен слить изменения, сделанные первым разработчиком, чтобы избежать их перезаписи во время отправки на сервер. Эта концепция верна как для Git, так и для Subversion (или любой другой централизованной системы контроля версий), и прекрасно работает в Git.

Если в вашей организации или команде уже используется централизованный рабочий процесс, то вам ничего не нужно менять для работы с Git. Достаточно создать один репозиторий и предоставить каждому члену команды push-доступ; Git не позволит перезаписать изменения, сделанные другими.

Предположим, Джон и Джессика начинают работать над проектом одновременно. Джон вносит изменения и отправляет их на сервер. Затем Джессика пытается отправить свои изменения, но сервер их отклоняет. Ей говорят, что она пытается отправить изменения, для которых невозможно выполнить быструю перемотку и она не сможет это сделать пока не получит все новые для неё изменения и не сольёт их. Такой рабочий процесс привлекает большинство людей, так как реализует парадигму, с которой они уже знакомы.

Такой подход применим не только к небольшим командам. Используя модель ветвления Git, сотни разработчиков могут одновременно работать над одним проектом, используя при этом десятки веток.

Диспетчер интеграции

Так как Git допускает использование нескольких удалённых репозиториев, то становится возможным организация рабочего процесса, где каждый разработчик имеет доступ на запись в свой публичный репозиторий и доступ на чтение ко всем остальным. При таком сценарии обычно существует канонический репозиторий, который представляет собой «официальный» проект. Для отправки своих наработок в этот проект следует создать его клон и отправить изменения в него. Затем вы отправляете запрос на слияние ваших изменений сопровождающему основного проекта. В свою очередь он может добавить ваш репозиторий как удалённый, протестировать ваши изменения локально, слить их в

соответствующую ветку и отправить в основной репозиторий. Процесс работает в следующей последовательности (смотри Рабочий процесс с менеджером по интеграции):

1. Сопровождающий проекта отправляет изменения в свой публичный репозиторий.
2. Участник клонирует этот репозиторий и вносит изменения.
3. Участник отправляет свои изменения в свой публичный репозиторий.
4. Участник отправляет письмо сопровождающему с запросом на слияние изменений.
5. Сопровождающий добавляет репозиторий участника как удалённый и сливает изменения локально.
6. Сопровождающий отправляет слитые изменения в основной репозиторий.

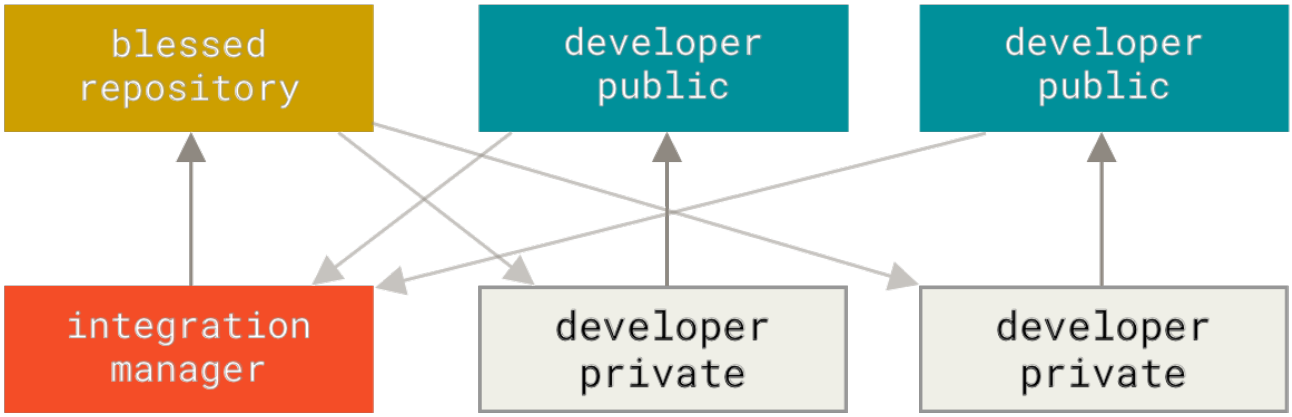


Рисунок 54. Рабочий процесс с менеджером по интеграции

Это достаточно распространённый вид организации рабочего процесса, в котором используются хабы, такие как GitHub или GitLab, где легко создать свой репозиторий как ответвление проекта (fork) и отправить туда свои изменения. Одним из основных преимуществ этого подхода является то, что вы можете продолжать работать, а сопровождающий основного репозитория может слить ваши изменения в любое время. Участники не должны ждать пока основной проект примет их изменения — каждый может работать в своём темпе.

Диктатор и помощники

Это вариант организации рабочего процесса с использованием нескольких репозиториев. В основном такой подход используется на огромных проектах, насчитывающих сотни участников; самый известный пример — ядро Linux. Помощники (lieutenants) — это интеграционные менеджеры, которые отвечают за отдельные части репозитория. Над ними главенствует один диспетчер интеграции, которого называют великодушным диктатором. Репозиторий великодушного диктатора выступает как эталонный (blessed), откуда все участники процесса должны получать изменения. Процесс работает следующим образом (смотри Рабочий процесс с великодушным диктатором):

1. Обычные разработчики работают в своих тематических ветках и перебазируют свою работу относительно ветки `master`. Ветка `master` — это ветка эталонного репозитория в которую имеет доступ только диктатор.
2. Помощники сливают тематические ветки разработчиков в свои ветки `master`.

3. Диктатор сливает ветки `master` помощников в свою ветку `master`.
4. Наконец, диктатор отправляет свою ветку `master` в эталонный репозиторий, чтобы все остальные могли перебазировать свою работу на основании неё.

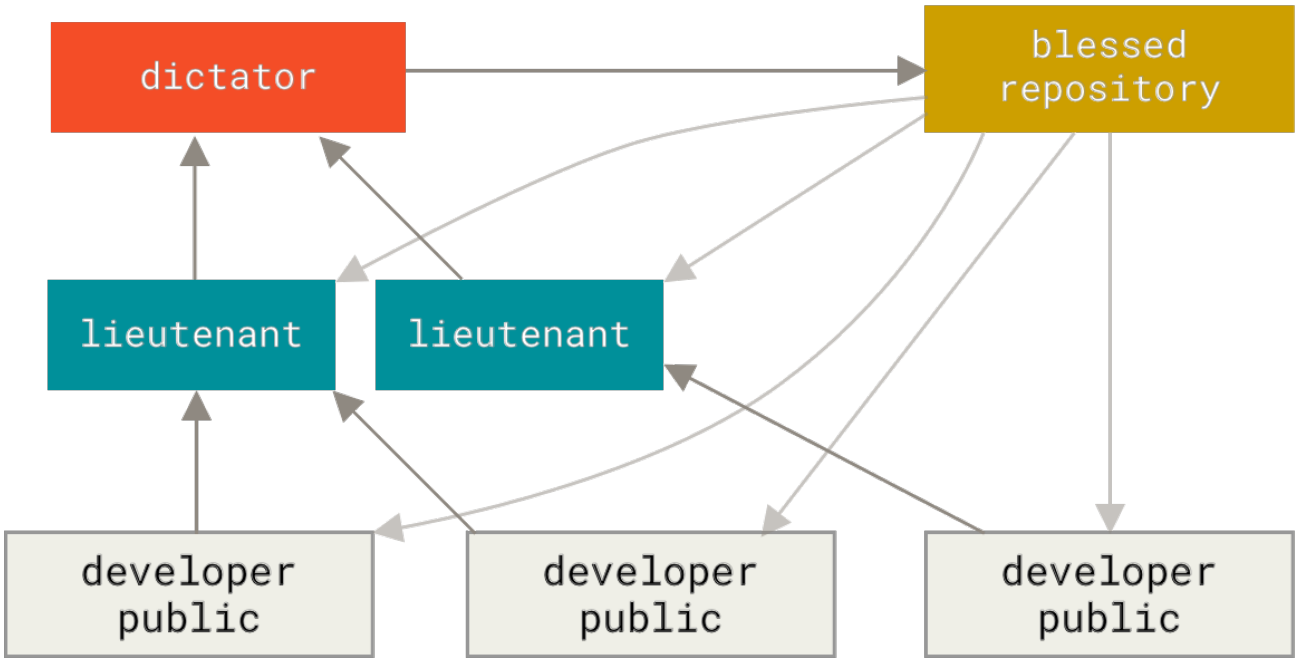


Рисунок 55. Рабочий процесс с великодушным диктатором

Такой вид организации рабочего процесса не является обычным, но может быть применён к очень большим проектам или в сильно иерархической среде. Это позволяет лидеру проекта (диктатору) делегировать бóльшую часть работы и собирать большие подмножества кода в различных точках до того как они будут интегрированы.

Шаблоны для управления ветками исходного кода

Мартин Фаулер сделал руководство «Шаблоны для управления ветками исходного кода». Это руководство охватывает все стандартные рабочие процессы Git и объясняет, как и когда их использовать. Также есть раздел, в котором сравнивается высокая и низкая частота слияния.

https://martinfowler.com/articles/branching-patterns.html

Заключение

Это наиболее часто используемые подходы в организации рабочего процесса на основании распределённых систем, таких как Git. Но, как вы можете видеть, существует множество различных решений для организации рабочего процесса в реальных проектах. Теперь, когда вы определились с комбинацией рабочих процессов, мы рассмотрим ряд более конкретных примеров как использовать основные роли в различных процессах. В следующем разделе вы узнаете о некоторых основных моделях, описывающих процесс участия в проекте.

Участие в проекте

Как именно участвовать в проекте — описать сложно, так как существует очень много различных вариаций как это делать. Так как Git очень гибок, люди могут и работают вместе по-разному. Отсюда и проблема описания участия в проекте — все проекты разные. Переменными здесь являются: количество активных участников, выбранный рабочий процесс, права доступа и, возможно, метод организации внесения вклада в проект извне.

Первая переменная — количество активных участников — подразумевает количество пользователей, которые активно отправляют свой код в проект и как часто они это делают. В большинстве случаев у вас два или три разработчика, которые делают по несколько коммитов в день или меньше, если речь идёт о вялотекущих проектах. В больших компаниях или проектах количество разработчиков может исчисляться тысячами с сотнями тысяч коммитов в день. Это очень важно, так как при увеличении количества разработчиков вы сталкиваетесь со всё большим количеством проблем, связанных со встраиванием или слиянием нового кода. Изменения, которые вы отправляете, могут быть признаны устаревшими или быть серьёзно затронутыми уже применёнными изменениями, пока ваши ожидали одобрения. Как в таком случае можно быть уверенным, что ваш код консистентен и актуален, а ваши коммиты валидны?

Следующая переменная — это используемый рабочий процесс. Централизован ли рабочий процесс и обладают ли все разработчики одинаковыми правами на запись в основную ветку разработки? Существует ли менеджер по интеграции или сопровождающий, кто проверяет все патчи? Все ли патчи проверяются другими разработчиками и проходят одобрение? Вы вовлечены в этот процесс? Существует ли лейтенант, которому следует отправить изменения прежде, чем в основной репозиторий?

Следующая проблема — это уровень доступа. Рабочий процесс, используемый для участия в проекте, может сильно отличаться в зависимости от того, есть ли у вас доступ на запись или нет. Если у вас нет доступа на запись, то как проект принимает изменения? Существует ли вообще политика принятия изменений? Как много изменений вы вносите за раз? Как часто вы это делаете?

Все эти вопросы могут повлиять на эффективность вашего участия в проекте, а так же на то, какие рабочие процессы наиболее предпочтительны или доступны для вас. Мы рассмотрим аспекты каждого из них на примере реальных ситуаций, переходя от более простых к более сложным. На основе этих примеров вы сможете создать реальные рабочие процессы применимые на практике.

Правила создания коммитов

Прежде чем мы начнём рассматривать конкретные варианты использования, давайте вспомним о сообщениях к коммитам. Наличие чётких рекомендаций по созданию коммитов и их соблюдение делают работу с Git и взаимодействие с другими гораздо проще. Проект Git предоставляет документ, в котором содержится ряд полезных советов по созданию коммитов для отправки патчей — вы можете ознакомиться с ними, прочитав файл `Documentation/SubmittingPatches`, находящийся в исходных кодах Git.

Для начала, вам не следует отправлять ненужные пробелы. Git предоставляет простой

способ проверки — перед коммитом выполните команду `git diff --check`, которая выведет список ненужных пробелов.

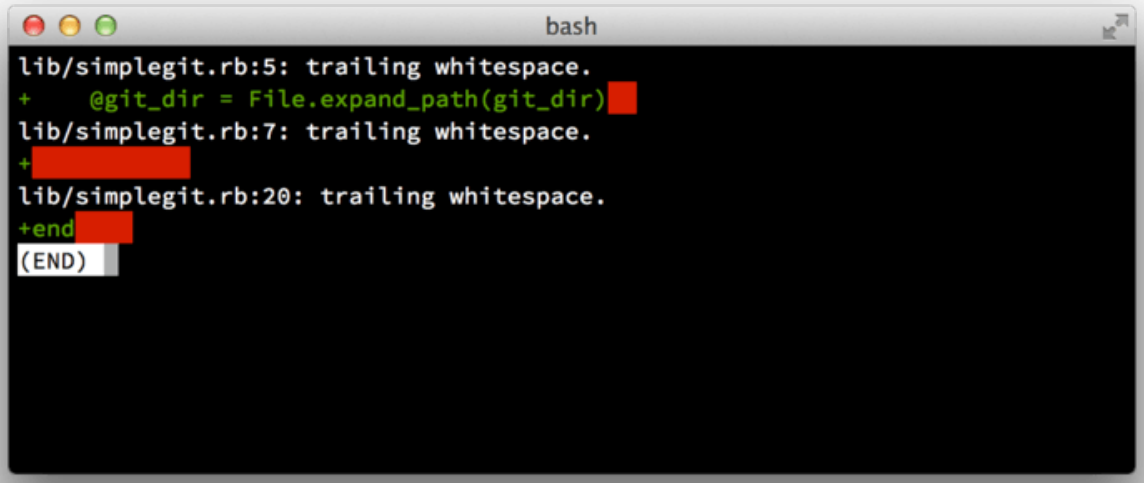


Рисунок 56. Вывод команды `git diff --check`

Выполняя эту команду перед коммитом вы сможете избежать отправки ненужных пробелов, которые могут раздражать других разработчиков.

Далее, постарайтесь делать коммит логически разделённого набора изменений. Если возможно, попытайтесь делать ваши изменения легко понятными — не нужно писать код все выходные, работая над пятью разными задачами, а в понедельник отправлять результат как один большой коммит. Даже если вы не делали коммиты на выходных, то в понедельник используйте область подготовленных файлов для того, чтобы разделить проделанную работу по принципу минимум один коммит на задачу, давая полезные комментарии к каждому из них. Если несколько изменений касаются одного файла, используйте `git add --patch` для частичного добавления файлов в индекс (детально описано в разделе Интерактивное индексирование главы 7). Состояние проекта в конце ветки не зависит от количества сделанных вами коммитов, так как все изменения добавятся в один момент, поэтому постарайтесь облегчить задачу вашим коллегам, когда они будут просматривать ваши изменения.

Такой подход так же облегчает извлечение или отмену отдельных изменений, если это вдруг потребуется в будущем. Раздел Перезапись истории главы 7 описывает ряд полезных трюков Git для переписывания истории изменений и интерактивного индексирования — используйте эти инструменты для создания чистой и понятной истории перед отправкой проделанной работы кому-то ещё.

Последнее, что нужно иметь в виду — это сообщение коммита. Привычка создавать качественные сообщения к коммитам позволяет упростить использование и взаимодействие посредством Git. Как правило, ваши сообщения должны начинаться кратким однострочным описанием не более 50 символов, затем должна идти пустая строка, после которой следует более детальное описание. Проект Git требует, чтобы детальное описание включало причину внесения изменений и сравнение с текущей

реализацией — это хороший пример для подражания. Пишите сообщение коммита в императиве: «Fix bug» а не «Fixed bug» или «Fixes bug». Вот отличный шаблон хорошего сообщения коммита, который мы слегка адаптировали из шаблона, изначально написанного Тимом Поупом:

```
Краткое (не более 50 символов) резюме с заглавной буквы

Более детальный, поясняющий текст, если он требуется.
Старайтесь не превышать длину строки в 72 символа.
В некоторых случаях первая строка подразумевается как тема письма, а всё остальное --
как тело письма.
Пустая строка, отделяющая сводку от тела, имеет решающее значение
(за исключением случаев, когда детального описания нет);
в противном случае такие инструменты, как rebase, могут вас запутать.

Сообщения коммитов следует писать используя неопределённую форму глагола совершенного
вида повелительного наклонения: «Fix bug» (Исправить баг), а не «Fixed bug» (Исправлен
баг) или «Fixes bug» (Исправляет баг).
Это соглашение соответствует сообщениям коммитов, генерируемых такими командами, как
`git merge` и `git revert`.

Последующие абзацы идут после пустых строк.

  - Допускаются обозначения пунктов списка

  - Обычно, элементы списка обозначаются с помощью тире или звёздочки,
    с одним пробелом перед ними, а разделяются пустой строкой, но
    соглашения могут отличаться

  - Допускается обратный абзацный отступ.
```

Вам и вашим разработчикам будет гораздо проще, если все сообщения ваших коммитов будут так выглядеть. В проекте Git все сообщения к коммитам имеют расширенное форматирование — выполните команду `git log --no-merges`, чтобы увидеть как выглядит хорошо отформатированная история коммитов.

Делайте как мы говорим, а не как делаем сами.

Большинство примеров в этой книге не используют расширенного форматирования сообщений коммитов; вместо этого мы просто используем параметр `-m` для команды `git commit`.

Поступайте так как мы говорим сейчас, а не так как делаем мы.

Небольшая команда

Самая простая ситуация, с которой вы можете столкнуться, это частный проект с одним или двумя другими разработчиками. «Частный» — в данном контексте понимается как проект с закрытым исходным кодом, недоступный для внешнего мира. Вы и другие разработчики

имеете права записи в репозиторий.

В такой среде вы можете использовать рабочий процесс, при котором выполняемые действия аналогичны использованию Subversion или другой централизованной системе. Вы всё ещё можете использовать преимущества создания коммитов оффлайн, значительно более простое ветвление и слияние, но процесс будет очень похожим; основное отличие в том, что слияние происходит на стороне клиента, а не на сервере во время коммита. Давайте посмотрим что происходит, когда два разработчика начинают работать вместе и используют общий репозиторий. Первый разработчик Джон клонирует репозиторий, вносит изменения и делает коммит локально. (В последующих примерах сообщения протокола заменены на `⋯` с целью их немного сократить.)

```
# Компьютер Джона
$ git clone john@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'Remove invalid default value'
[master 738ee87] Remove invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Второй разработчик Джессика делает то же самое — клонирует репозиторий и делает коммит:

```
# Компьютер Джессики
$ git clone jessica@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'Add reset task'
[master fbff5bc] Add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Затем Джессика отправляет изменения на сервер:

```
# Компьютер Джессики
$ git push origin master
...
To jessica@githost:simplegit.git
   1edee6b..fbff5bc  master -> master
```

В последней строке примера приведена полезная информация, выводимая после каждой операции отправки изменений. Её базовый формат такой `<oldref>..<newref> fromref → toref`, где `oldref` — коммит, на который указывала ветка до отправки, `newref` — новый коммит, на который ветка указывает сейчас, `fromref` — имя отправленной локальной ветки, `toref` — имя

ветки в удалённом репозитории, в которую были отправлены изменения. Далее вы увидите похожие результаты в выводе команд, поэтому имея общее представление о значении поможет вам лучше понимать различные состояния репозиториев. Дополнительную информацию можно найти в документации к команде git-push.

Возвращаясь к примеру, немного спустя, Джон вносит некоторые изменения, делает коммит и пытается отправить его на тот же сервер:

```
# Компьютер Джона
$ 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'
```

В данном случае изменения Джона отклонены, так как Джессика уже отправила *свои*. Это особенно важно для понимания если вы привыкли к Subversion, потому что, как вы могли заметить, разработчики не редактировали один и тот же файл. Если Subversion автоматически делает слияние на сервере при условии, что редактировались разные файлы, то в Git вы должны *сначала* слить изменения локально. Джон должен получить изменения Джессики и слить их локально, прежде чем сможет отправить свои.

Для начала, Джон получает изменения Джессики (слияния изменений пока что не происходит):

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master     -> origin/master
```

В этот момент локальный репозиторий Джона выглядит примерно так:

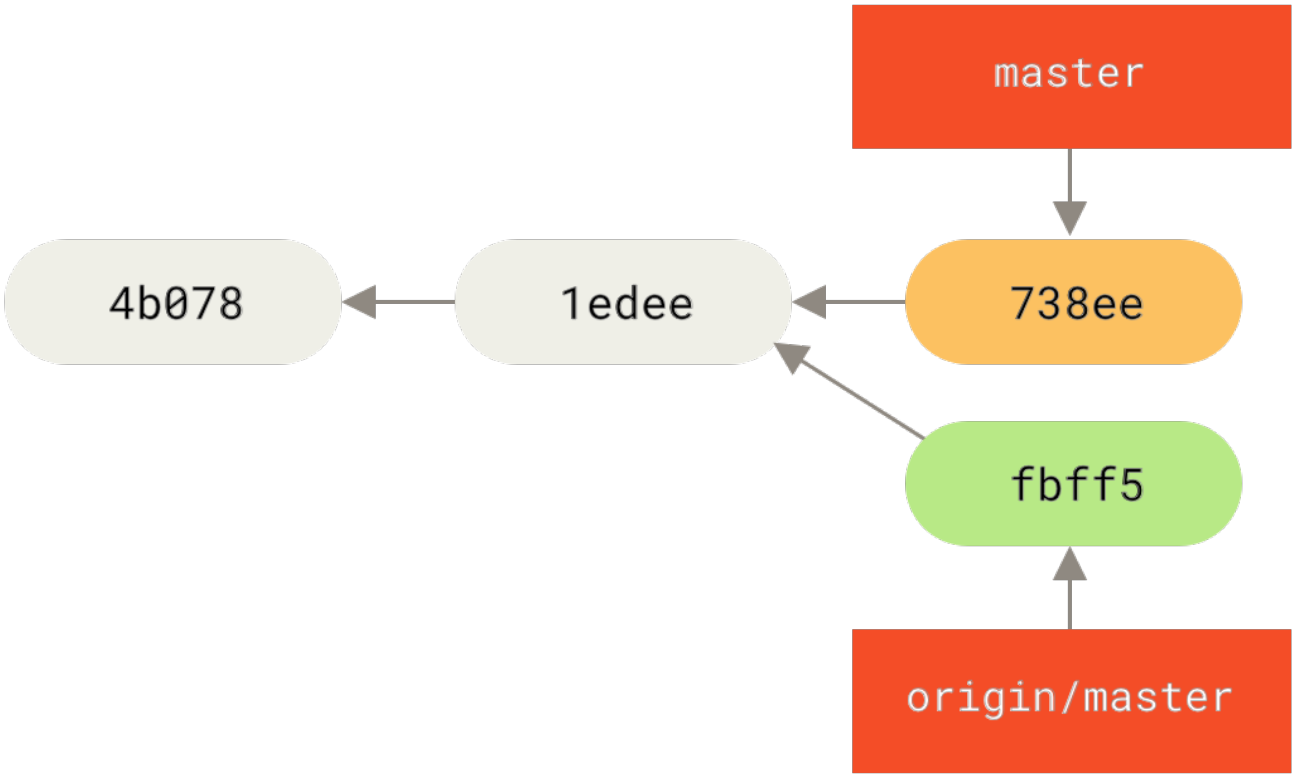


Рисунок 57. Расходящаяся история Джона

Теперь Джон может слить полученные изменения Джессики со своей локальной веткой:

```
$ git merge origin/master
Merge made by the 'recursive' strategy.
 TODO |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Процесс слияния проходит гладко — история коммитов у Джона выглядит примерно так:

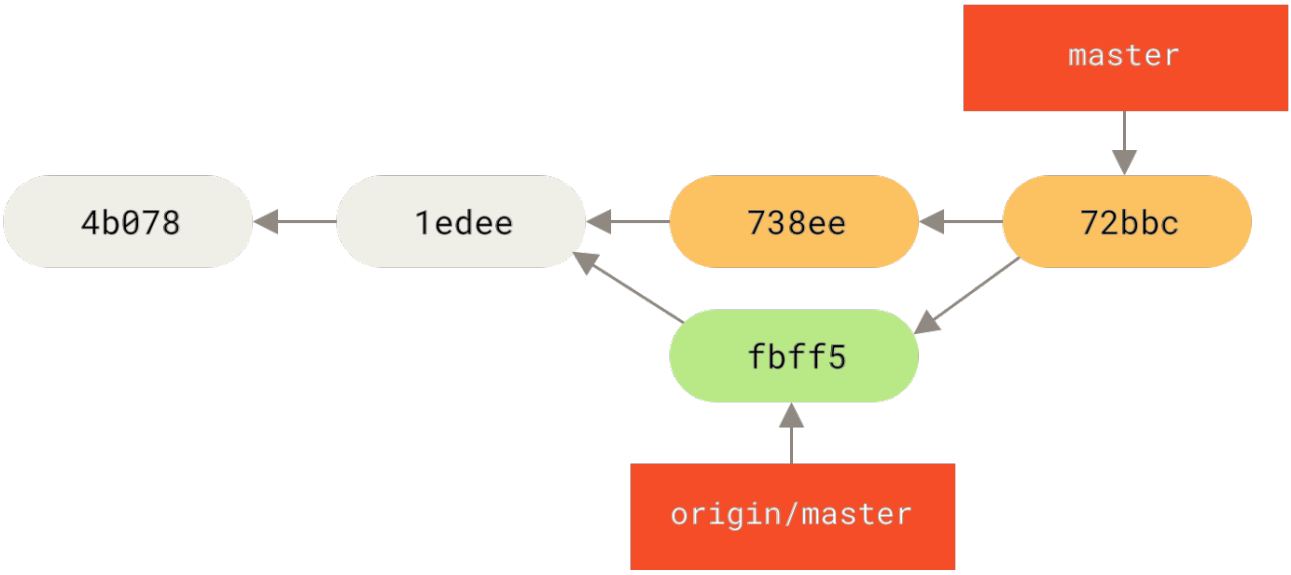


Рисунок 58. Репозиторий Джона после слияния с `origin/master`

Теперь Джон может протестировать новый код, чтобы убедиться в корректной работе

объединённых изменений, после чего он может отправить объединённые изменения на сервер:

```
$ git push origin master
...
To john@githost:simplegit.git
   fbff5bc..72bbc59  master -> master
```

В результате история коммитов у Джона выглядит так:

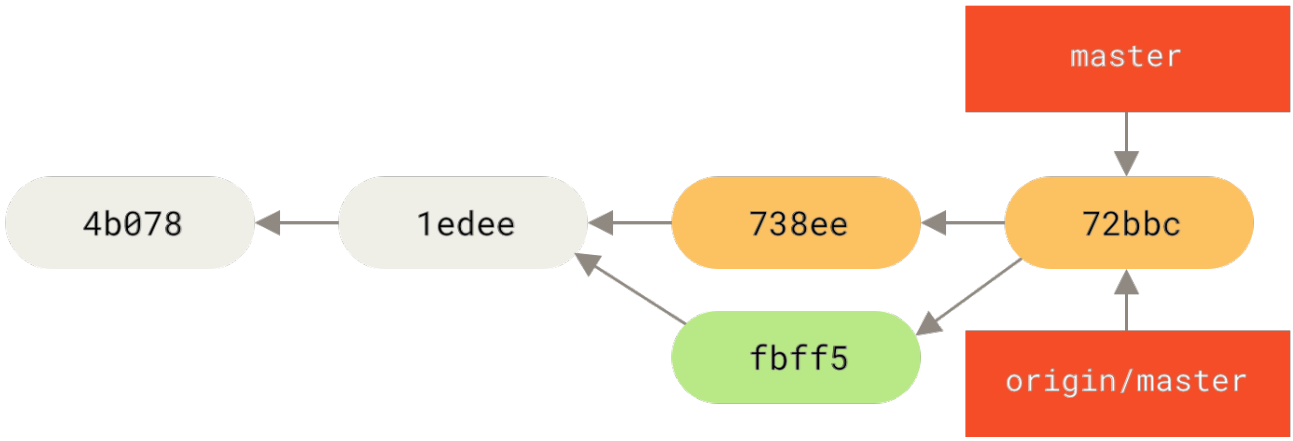


Рисунок 59. История коммитов у Джона после отправки на `origin` *сервер*

Тем временем Джессика создала тематическую ветку с названием `issue54` и сделала в ней три коммита. Она ещё не получила изменения Джона, поэтому история коммитов у неё выглядит следующим образом:

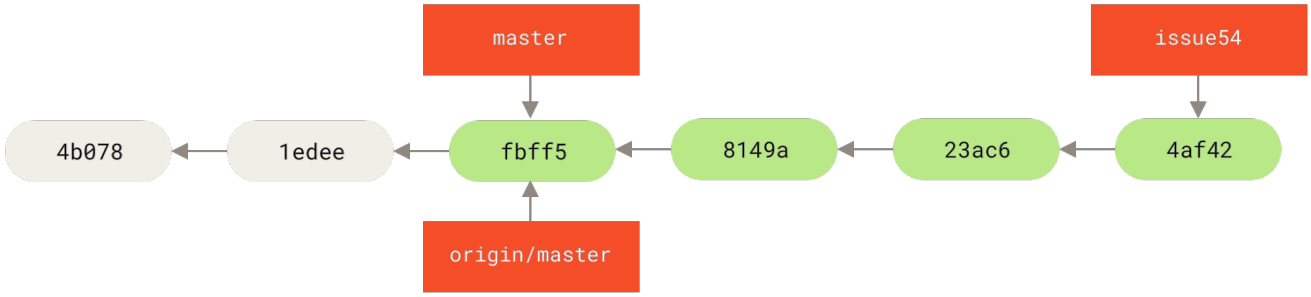


Рисунок 60. Тематическая ветка Джессики

Внезапно Джессика узнаёт, что Джон отправил какие-то изменения на сервер и теперь она хочет на них взглянуть; для этого ей следует получить с сервера все новые изменения:

```
# Компьютер Джессики
$ git fetch origin
...
From jessica@githost:simplegit
   fbff5bc..72bbc59  master     -> origin/master
```

Это приводит к получению изменений, отправленных Джоном в репозиторий. Теперь,

история коммитов у Джессики выглядит так:

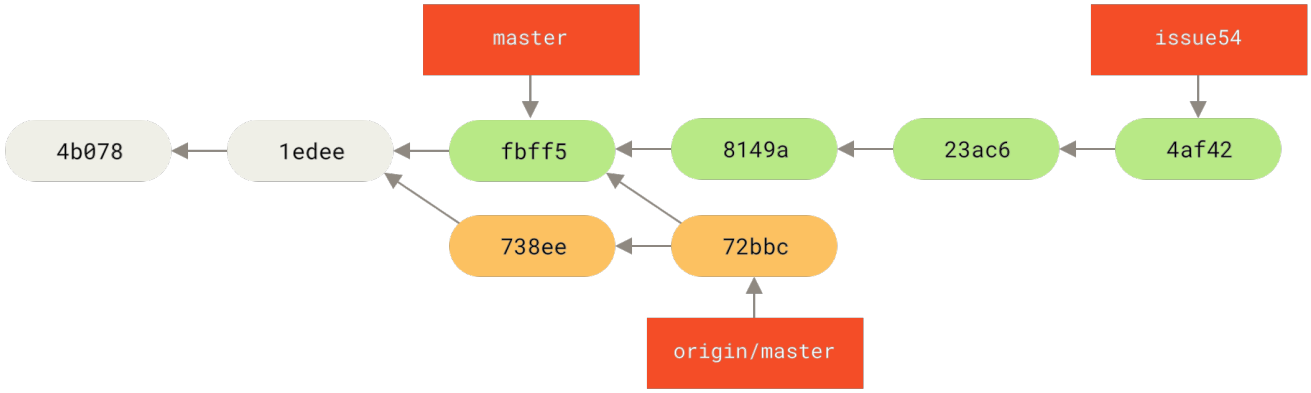


Рисунок 61. История коммитов Джессики после получения изменений Джона

Джессика считает, что её тематическая ветка готова, но так же хочет знать какие изменения следует слить со своей работой перед отправкой на сервер. Для прояснения ситуации она выполняет команду `git log`:

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

    Remove invalid default value
```

`issue54..origin/master` — это синтаксис фильтра, который указывает Git отображать только список коммитов, которые существуют в последней ветке (в данном случае `origin/master`), но отсутствуют в первой (в данном случае `issue54`). Более детально этот синтаксис рассматривается в разделе Диапазоны коммитов главы 7.

В данном случае, в выводе команды мы видим только один коммит, сделанный Джоном и ещё не слитый Джессикой. Если она сольёт `origin/master`, то это будет единственный коммит, который изменит локальное состояние.

Теперь, Джессика может слить изменения тематической ветки и изменения Джона (`origin/master`) в свою локальную ветку `master`, а затем отправить её на сервер.

Для начала (при условии отсутствия изменений в тематической ветке, не включённых в коммит), Джессика переключается на свою ветку `master`:

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

Обе ветки `origin/master` и `issue54` являются отслеживаемыми, поэтому порядок слияния не важен. Конечный результат будет идентичным вне зависимости от порядка слияния, однако история коммитов будет немного отличаться. Джессика решает слить ветку `issue54`

первой:

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

Проблем не возникает; как можно заметить, это простое перемещение вперёд. Теперь Джессика заканчивает процесс локального слияния объединяя полученные ранее изменения Джона, находящиеся в ветке `origin/master`:

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by the 'recursive' strategy.
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Слияние прошло чисто и теперь история коммитов у Джессики выглядит следующим образом:

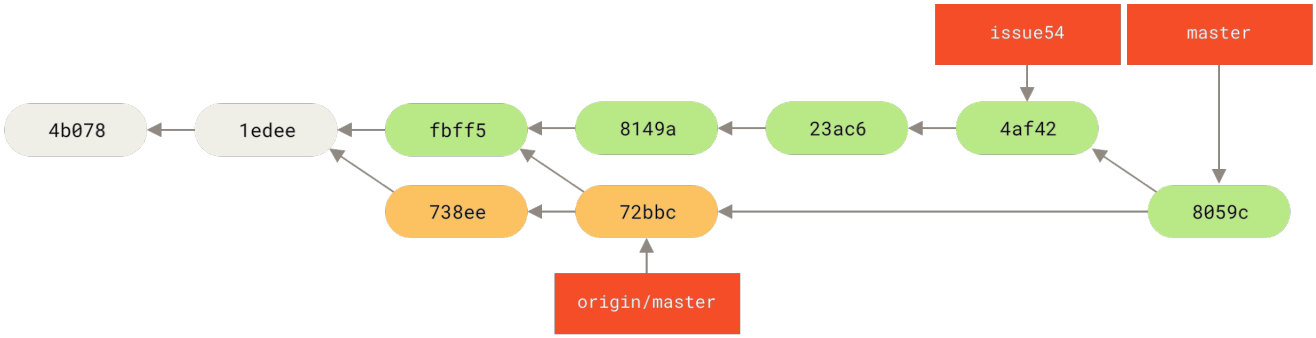


Рисунок 62. История коммитов Джессики после слияния изменений Джона

Теперь Джессика может отправить свою ветку `master` в `origin/master`, при условии что Джон больше не отправлял изменений:

```
$ git push origin master
...
To jessica@githost:simplegit.git
   72bbc59..8059c15  master -> master
```

Каждый разработчик сделал коммиты несколько раз и успешно слил изменения другого.

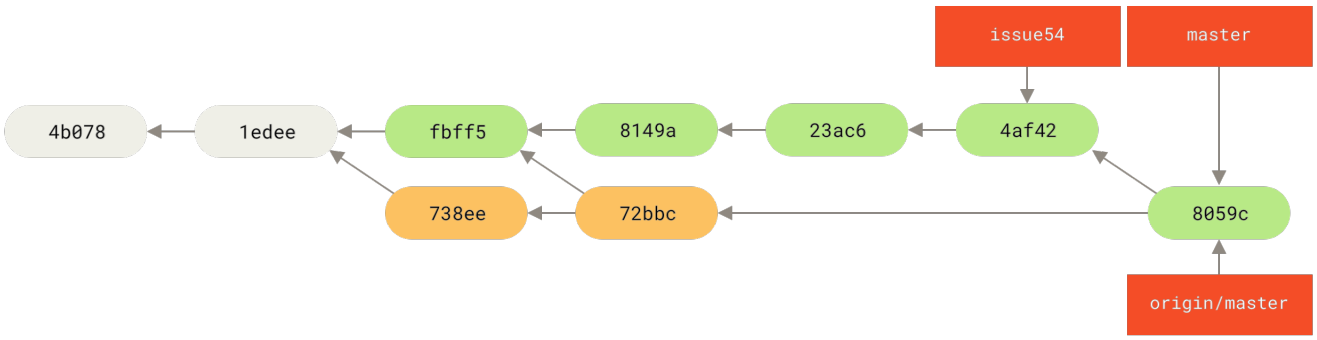


Рисунок 63. История коммитов Джессики после отправки на сервер

Это один из самых простых рабочих процессов. В течение некоторого времени вы работаете в тематической ветке, а затем сливаете изменения в ветку `master` когда всё готово. Чтобы поделиться проделанной работой, вы сливаете её в вашу ветку `master`, затем получаете и сливаете изменения из ветки `origin/master` если таковые имеются, и наконец, отправляете все изменения в ветку `master` на сервере. В общем виде последовательность выглядит так:

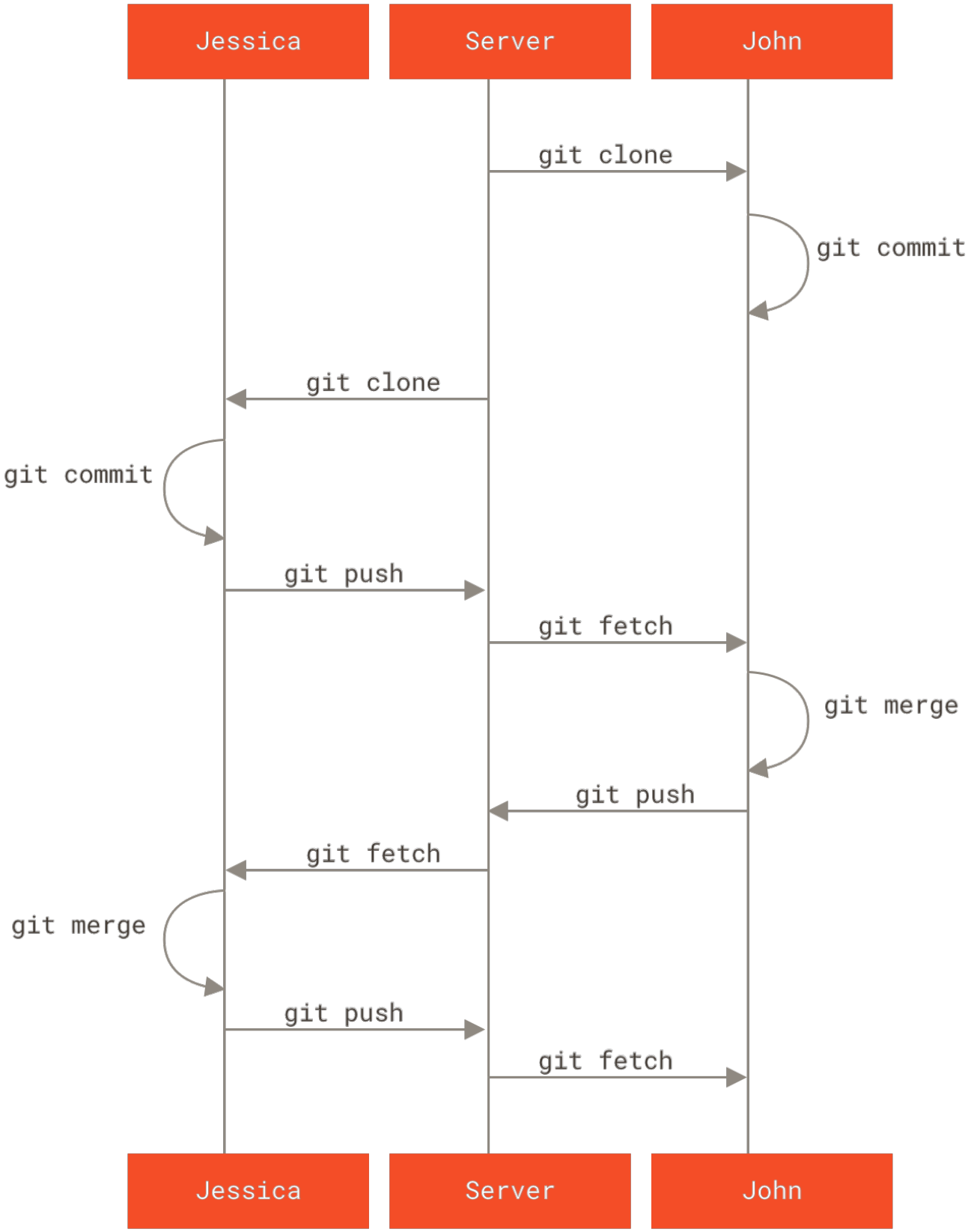


Рисунок 64. Общий вид последовательности событий в рабочем процессе для нескольких разработчиков

Команда с руководителем

В этом сценарии мы рассмотрим роли участников в более крупной частной команде. Вы узнаете как работать в окружении, где мелкие группы совместно работают над

улучшениями, а затем их вклад интегрируется третьей стороной.

Предположим, что Джон и Джессика вместе работают над одной функцией (назовём её «featureA»), при этом Джессика и Джози работают над другой («featureB»). В этом случае компания использует тип рабочего процесса с менеджером по интеграции, при котором работа отдельных групп интегрируется определёнными инженерами, а ветка `master` основного репозитория может быть обновлена только этими инженерами. При таком сценарии вся работа ведётся в отдельных ветках для каждой команды, а затем объединяется интегратором.

Давайте рассмотрим рабочий процесс Джессики, так как она работает над двумя функциями, параллельно сотрудничая с разными разработчиками. Предположим, что репозиторий уже клонирован и она решает работать сначала над функцией `featureA`. Джессика создаёт новую ветку для этой функции и некоторое время работает над ней:

```
# Компьютер Джессики
$ 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(-)
```

В данный момент ей необходимо поделиться проделанной работой с Джоном, поэтому Джессика отправляет ветку `featureA` на сервер. У Джессики нет доступа на запись в ветку `master` (он есть только у интеграторов), поэтому для совместной работы с Джоном она отправляет изменения в другую ветку:

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Джессика отправляет письмо Джону с уведомлением, что внесённые ей изменения уже доступны в ветке `featureA`. Пока Джессика ждёт ответа от Джона, она решает поработать над другой функцией `featureB` вместе с Джози. Для начала, Джесика создаёт новую тематическую ветку, базируясь на состоянии ветки `master` на сервере:

```
# Компьютер Джессики
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

После этого, Джессика делает несколько коммитов в ветке `featureB`:

```
$ vim lib/simplegit.rb
```

```
$ git commit -am 'Make ls-tree function recursive'
[featureB e5b0fdc] Make 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(-)
```

Репозиторий Джессики выглядит следующим образом:

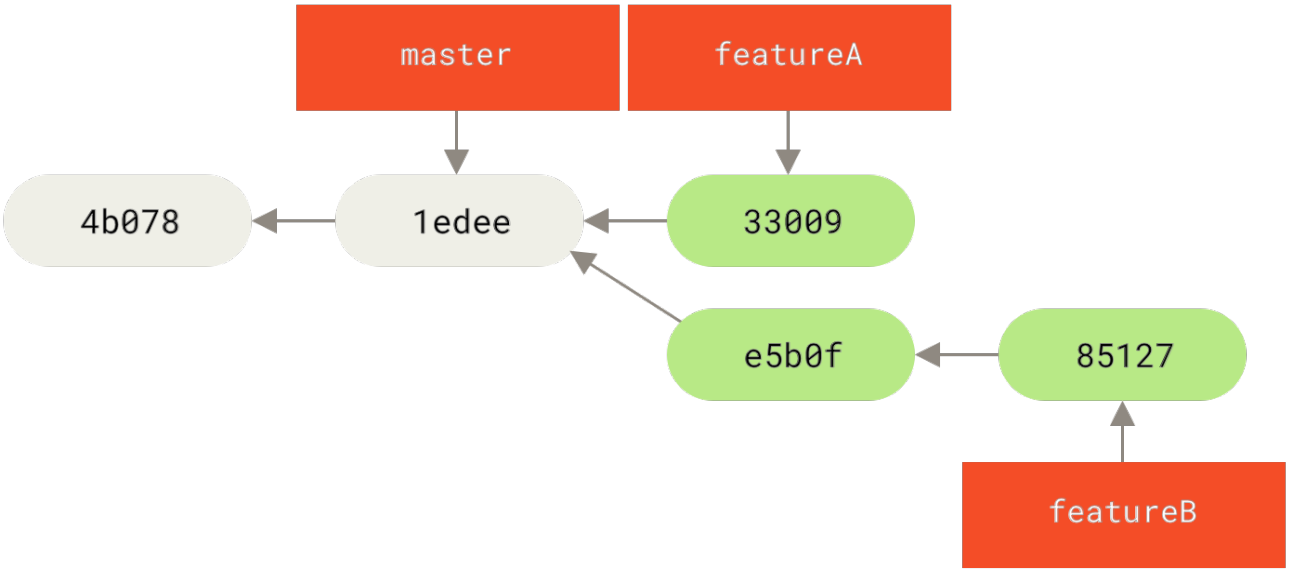


Рисунок 65. Начальное состояние истории коммитов Джессики

Джессика готова отправить свою работу, но получает письмо Джози, что начальная работа уже отправлена на сервер в ветку `featureBee`. Теперь Джессике нужно слить эти изменения со своими перед отправкой на сервер. Изменения Джози она получает командой `git fetch`:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Полагая что Джессика находится в ветке `featureB`, она может слить полученные изменения Джози со своими при помощи команды `git merge`:

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by the 'recursive' strategy.
 lib/simplegit.rb |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

Одна небольшая проблема — ей нужно отправить слитые изменения из локальной ветки `featureB` в ветку `featureBee` на сервере. Для этого в команде `git push` Джессика указывает названия локальной и удалённой веток, разделённых двоеточием:

```
$ git push -u origin featureB:featureBee
...
To jessica@githost:simplegit.git
   fba9af8..cd685d1  featureB -> featureBee
```

Это называется *спецификация ссылок*. В разделе Спецификации ссылок главы 10 приведено более детальное описание спецификаций ссылок Git и различные способы их использования. Так же обратите внимание на флаг `-u`; это сокращение для `--set-upstream`, который настраивает ветки для упрощения отправки и получения изменений в дальнейшем.

После этого, Джессика получает письмо от Джона, в котором он сообщает, что отправил некоторые изменения в ветку `featureA` и просит их проверить. Джесика выполняет команду `git fetch` для получения *всех* новых изменений, включая изменения Джона:

```
$ git fetch origin
...
From jessica@githost:simplegit
   3300904..aad881d  featureA   -> origin/featureA
```

Теперь, она может посмотреть что именно было изменено путём сравнения полученной ветки `featureA` со своей локальной веткой:

```
$ git log featureA..origin/featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

    Increase log output to 30 from 25
```

Если Джессику всё устраивает, то она сливает изменения Джона в свою ветку `featureA`:

```
$ 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(-)
```

Джессика решает немного подправить, делает коммит в локальной ветке `featureA` и отправляет конечный результат на сервер:

```
$ git commit -am 'Add small tweak to merged content'
[featureA 774b3ed] Add small tweak to merged content
```

```
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
   3300904..774b3ed  featureA -> featureA
```

В результате история коммитов у Джессики выглядит так:

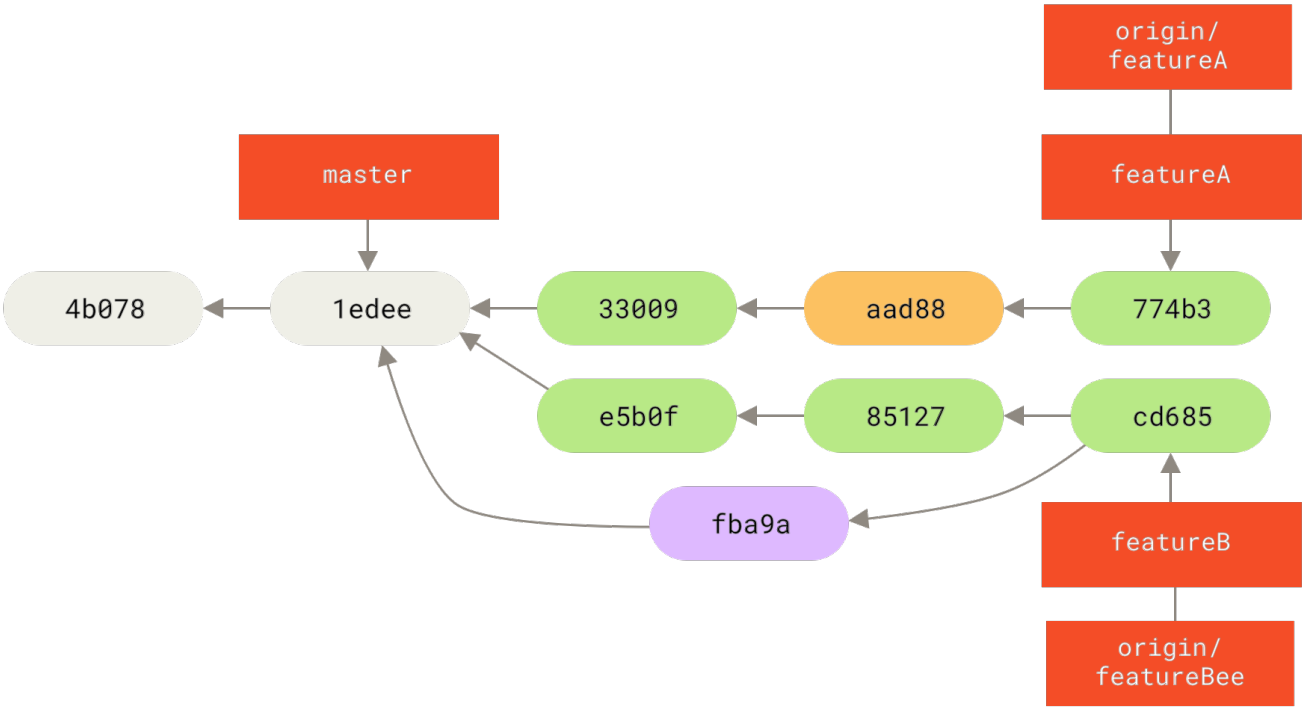


Рисунок 66. История коммитов Джессики после изменений в тематической ветке

Джессика, Джози и Джон информируют интеграторов, что ветки `featureA` и `featureBee` на сервере готовы к слиянию в основную. После того как интеграторы сольют эти ветки в основную, полученные изменения будут содержать коммит слияния, а история коммитов будет иметь вид:

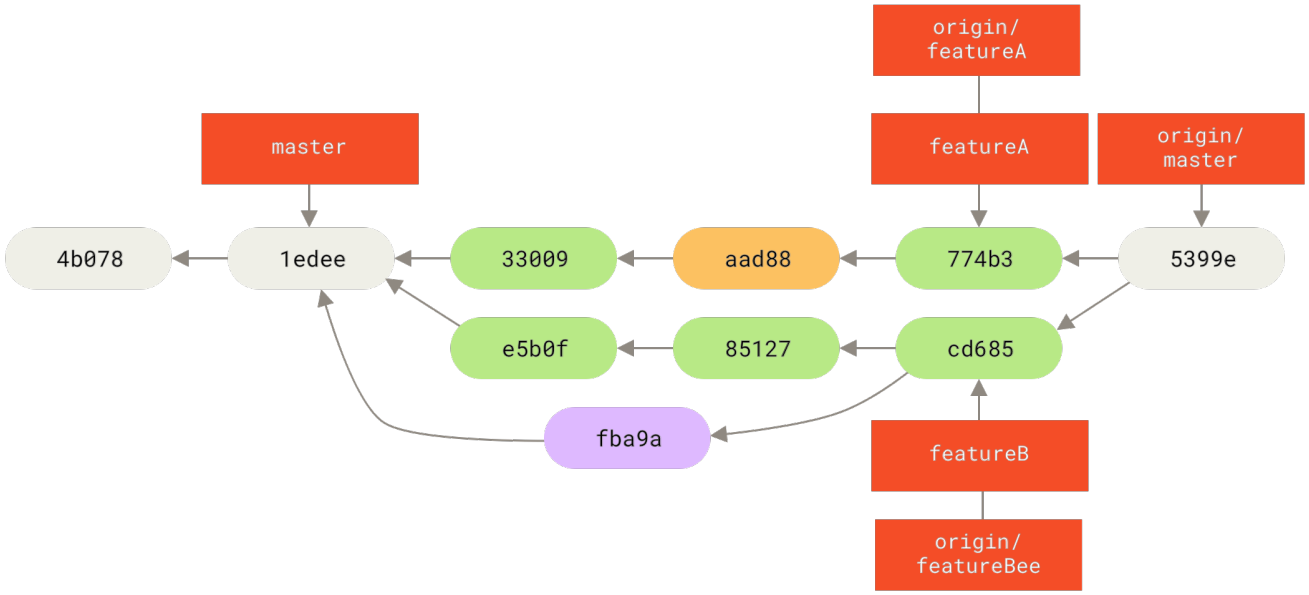


Рисунок 67. История коммитов Джессики после слияния тематических веток

Многие переходят на Git именно из-за возможности параллельной работы нескольких команд в различных направлениях с последующим слиянием проделанной работы. Возможность совместной работы небольших подгрупп команды в удалённых ветках без необходимости вовлекать или мешать всей команде — огромное преимущество Git. Последовательность действий в описанном рабочем процессе выглядит следующим образом:

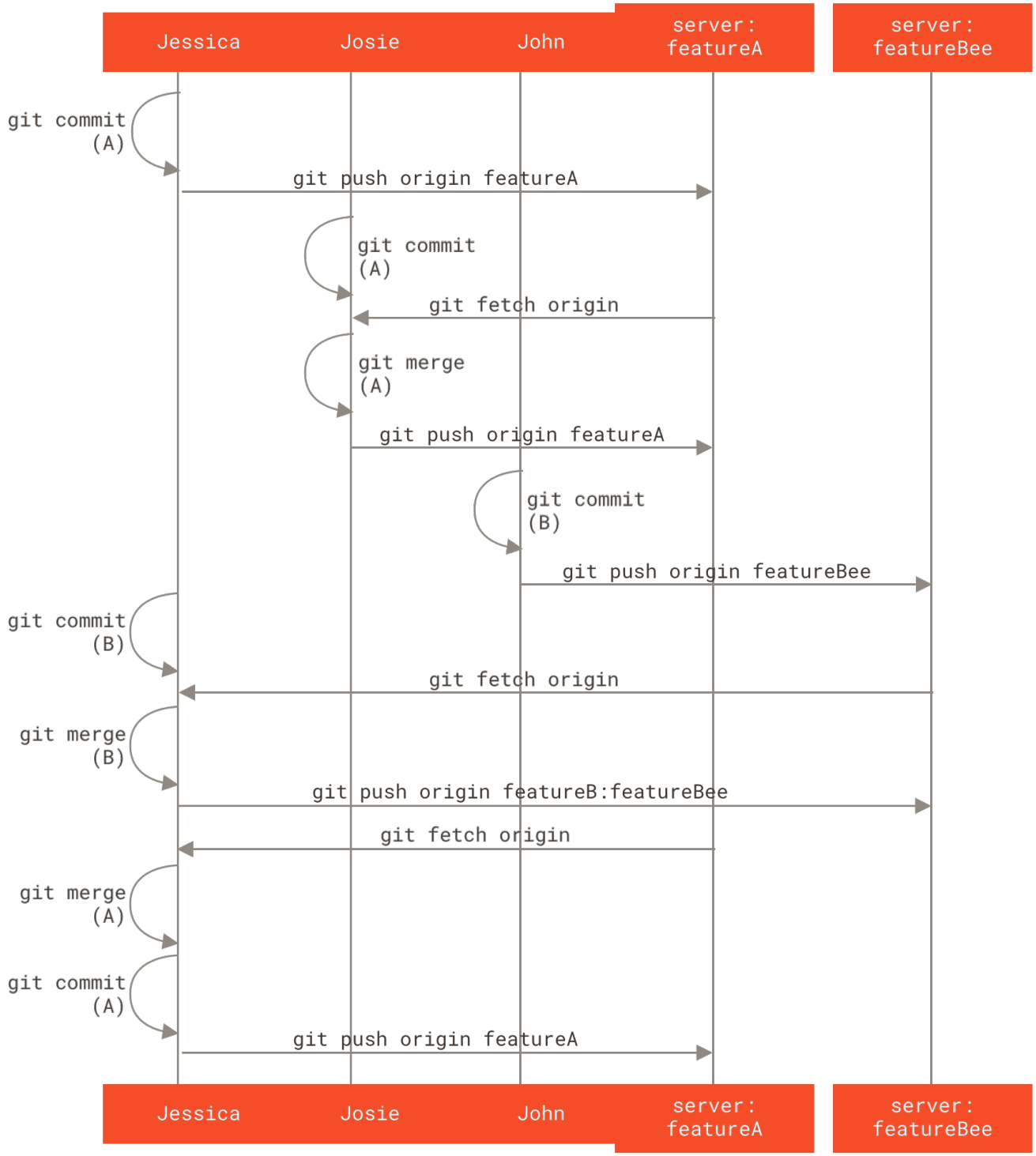


Рисунок 68. Основная последовательность описанного рабочего процесса управляемой команды

Форк публичного проекта

Участие в публичном проекте сильно отличается. Так как у вас нет доступа обновлять ветки проекта напрямую, то передавать проделанную работу следует другим способом. В первом примере рассматривается участие в публичном проекте посредством форка на Git платформах, где возможно его простое создание. Большинство сайтов Git хостинга поддерживают такую функцию (включая GitHub, BitBucket, repo.or.cz и другие), как и большинство тех, кто сопровождает проекты, ожидают такого же стиля участия. Следующий раздел посвящён проектам, которые предпочитают принимать исправления в виде патчей по электронной почте.

Для начала, вам следует клонировать основной репозиторий, создать тематическую ветку для одного или нескольких патчей и работать в ней. Обычно, последовательность действий выглядит так:

```
$ git clone <url>
$ cd project
$ git checkout -b featureA
  ... work ...
$ git commit
  ... work ...
$ git commit
```

Возможно, вы захотите использовать `rebase -i`, чтобы объединить несколько коммитов в один или переставить их местами, чтобы сопровождающему было легче проверять патч — смотрите раздел Перезапись истории для получения детальной информации об интерактивном перебазировании.

Когда работа в тематической ветке завершена и вы готовы передать изменения исходному проекту, перейдите на страницу исходного проекта и нажмите кнопку «Fork», тем самым создавая доступный для записи форк проекта. Затем нужно добавить URL на созданный проект как второй удалённый репозиторий, в нашем случае с именем `myfork`:

```
$ git remote add myfork <url>
```

После этого следует отправить проделанную работу в него. Проще отправить вашу тематическую ветку, в которой велась работа, чем сливать изменения в вашу ветку `master` и отправлять её. Если ваши изменения будут отклонены или какой-то из коммитов будет применён выборочно (команда `cherry-pick` более детально рассматривается в разделе Схема с перебазированием и отбором главы 5), то вы не сможете вернуть состояние вашей ветки `master`. Если менеджер проекта сольёт, перебазирует или выборочно применит ваши изменения, то вы сможете их получить из оригинального репозитория.

В любом случае, отправить свои изменения вы можете командой:

```
$ git push -u myfork featureA
```

Когда ваши изменения отправлены в ваш форк, следует уведомить сопровождающих исходного проекта о том, что у вас есть изменения для интеграции. Обычно, это называется *запросом слияния*, который вы можете создать используя как веб сайт — GitHub использует собственный механизм запросов слияния, который будет рассмотрен в главе GitHub — так и команду `git request-pull`, отправив её вывод по почте.

Команда `git request-pull` принимает в качестве аргументов название базовой ветки, в которую следует влить изменения из вашей тематической ветки, и ссылку на Git репозиторий, из которого следует получать изменения, а результатом будет список всех

изменений, которые вы предлагаете внести. Например, если Джессика хочет отправить Джону запрос слияния и она отправила два коммита в тематическую ветку, то ей следует выполнить команду:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
Jessica Smith (1):
        Create new function

are available in the git repository at:

  git://githost/simplegit.git featureA

Jessica Smith (2):
      Add limit to log function
      Increase log output to 30 from 25

 lib/simplegit.rb |   10 +++++++++-
 1 files changed, 9 insertions(+), 1 deletions(-)
```

Вывод команды можно отправить сопровождающему проекта — в нём говорится с какого момента велась работа, приводится сводка коммитов и указывается откуда можно получить эти изменения.

В проектах, где вы не являетесь сопровождающим, проще держать ветку `master` в соответствии с `origin/master`, а работать в тематических ветках — так вам будет проще отменить изменения, если они будут отклонены. Разделение направлений разработки по изолированным веткам облегчит их перебазирование, когда состояние основного репозитория изменится, а ваши коммиты уже не смогут быть чисто применены. Например, если вы собираетесь отправить исправления на другую тему, не продолжайте работать в той же тематической ветке — создайте новую, базируясь на ветке `master` основного репозитория:

```
$ git checkout -b featureB origin/master
  ... work ...
$ git commit
$ git push myfork featureB
$ git request-pull origin/master myfork
  ... email generated request pull to maintainer ...
$ git fetch origin
```

Теперь, каждая из ваших тематик разработки изолирована — аналогично очереди патчей — каждую из которых можно переписать, перебазировать или исправить без влияния на другие ветки:

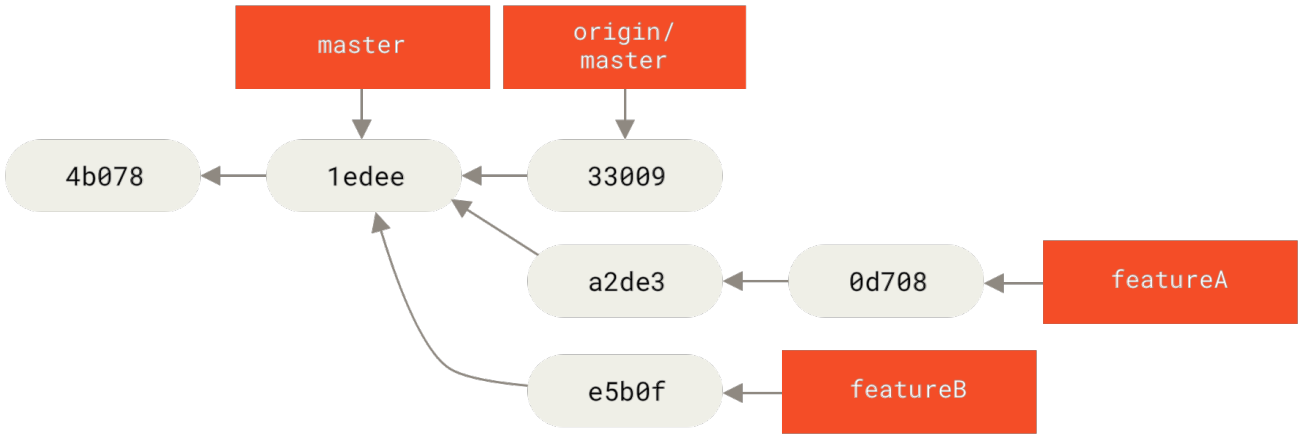


Рисунок 69. История коммитов в начале работы над `featureB`

Предположим, что сопровождающий проекта слил некоторый набор других патчей, а затем пытается применить вашу первую ветку, но она уже не может быть слита без конфликтов. В этом случае вы можете попытаться перебазировать свою ветку относительно `origin/master`, разрешить конфликты и заново отправить свои изменения:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

Эти действия перепишут историю ваших коммитов, которая станет похожа на История коммитов после работы над `featureA`.

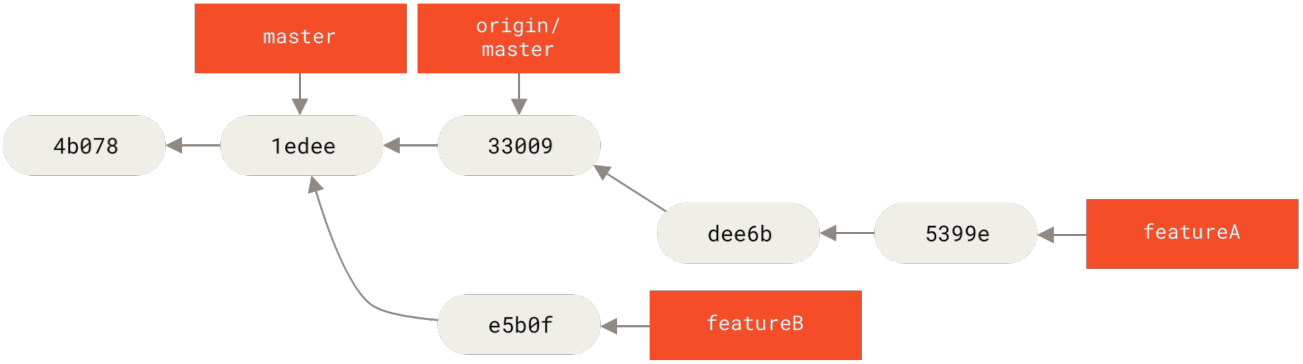


Рисунок 70. История коммитов после работы над `featureA`

Так как вы перебазировали ветку, то должны указать флаг `-f` во время отправки на сервер, чтобы переписать историю ветки `featureA` коммитами, не являющимися её потомками. Альтернативным решением может быть отправка этих исправлений в ветку с другим названием (например, `featureAv2`).

Давайте рассмотрим ещё один возможный сценарий: сопровождающий посмотрел вашу вторую ветку и ему понравилась идея, но он хочет попросить вас изменить некоторые детали. Возможно, вы так же захотите перебазировать эту ветку относительно текущего состояния ветки `master`. Вы создаёте новую ветку базируясь на текущей `origin/master`, сбрасываете все изменения в неё, разрешаете возможные конфликты, делаете изменения в

реализации и отправляете её как новую ветку:

```
$ git checkout -b featureBv2 origin/master
$ git merge --squash featureB
  ... change implementation ...
$ git commit
$ git push myfork featureBv2
```

Опция `--squash` берёт все изменения из указанной ветки, объединяет их и создаёт новый коммит в текущей ветке без создания коммита слияния. Это значит, что новый коммит будет иметь только одного родителя и будет включать все изменения из другой ветки, а так же позволяет внести дополнительные изменения до фактического создания коммита. Опция `--no-commit` указывает Git не создавать новый коммит автоматически.

Теперь можно отправить сопровождающему сообщение, что вы сделали запрошенные изменения и они находятся в вашей ветке `featureBv2`.

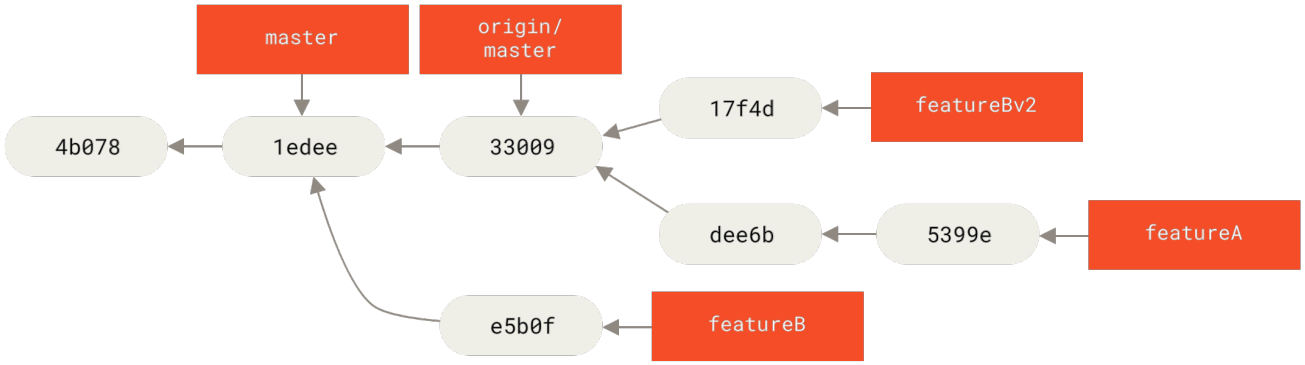


Рисунок 71. История коммитов после работы над `featureBv2`

Публичный проект посредством E-Mail

Много проектов имеют устоявшиеся процедуры по принятию патчей — вам следует ознакомиться с правилами для каждого проекта, так как они могут отличаться. Так как существует несколько больших старых проектов, которые принимают патчи посредством почтовых рассылок, мы рассмотрим такой пример.

Рабочий процесс похож на предыдущий — вы создаёте тематическую ветку для каждого набора патчей, над которыми собираетесь работать. Основное отличие в способе их передачи проекту. Вместо того, чтобы форкнуть проект и отправить в него свои изменения, вы генерируете почтовую версию для каждого набора коммитов с целью отправки её в список рассылки разработчиков:

```
$ git checkout -b topicA
  ... work ...
$ git commit
  ... work ...
```

```
$ git commit
```

Сейчас у вас два коммита, которые вы хотите отправить в почтовую рассылку. Используйте команду `git format-patch` для генерации файлов в формате mbox, которые можно отправить по почте — это обернёт каждый коммит в сообщение e-mail, где первая строка из сообщения коммита будет темой письма, а остальные строки плюс сам патч будут телом письма. Применение патча в формате e-mail, сгенерированного с помощью команды `format-patch`, сохраняет всю информацию о коммите должным образом.

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-increase-log-output-to-30-from-25.patch
```

Команда `format-patch` выводит список имён файлов патчей, которые она создаёт. Флаг `-M` указывает Git искать переименования. В итоге файлы выглядят вот так:

```
$ 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
```

Вы можете редактировать эти файлы, добавляя информацию для списка рассылки, но которую вы не хотите видеть в сообщении к коммиту. Если добавить текст между строкой `---` и началом патча (строка `diff --git`), то разработчики увидят его, но применяться он не

будет.

Для отправки в список рассылки можно либо вставить файлы в почтовую программу, либо отправить их из командной строки. Вставка текста обычно сопровождается проблемами форматирования, особенно при использовании «умных» клиентов, которые не заботятся о переносе строк и пробелах соответствующим образом. К счастью, Git предоставляет утилиту, которая умеет отправлять корректно отформатированные патчи по протоколу IMAP. Позже мы покажем как отправлять патчи через Gmail, так сложилось что мы знаем этот почтовый агент лучше других; вы можете воспользоваться инструкциями по использованию большого числа почтовых программ в вышеупомянутом файле `Documentation/SubmittingPatches` из исходных кодов Git.

Для начала, следует настроить раздел imap в файле `~/.gitconfig`. Каждое отдельное значение можно установить вызовом команды `git config`, а можно указать вручную сразу в файле, но в итоге файл конфигурации должен выглядеть следующим образом:

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = YX]8g76G_2^sFbd
  port = 993
  sslverify = false
```

Если ваш сервер IMAP не использует SSL, то последние две строки не обязательны, а значение host должно быть `imap://` вместо `imaps://`. Как только всё сделано, воспользуйтесь командой `git imap-send` для помещения ваших патчей в папку Drafts на указанном IMAP сервере:

```
$ cat *.patch |git imap-send
Resolving imap.gmail.com... ok
Connecting to [74.125.142.109]:993... ok
Logging in...
sending 2 messages
100% (2/2) done
```

Теперь вы можете перейти в папку Drafts, изменить поле To, указав адрес почтовой рассылки, при необходимости заполнить поле CC, указав адрес сопровождающего или ответственного, и отправить письмо.

Так же вы можете отправить свои патчи используя SMTP сервер. Как и в предыдущем случае, вы можете использовать набор команд `git config` или создать секцию `sendemail` в файле `~/.gitconfig`:

```
[sendemail]
  smtpencryption = tls
  smtpserver = smtp.gmail.com
```

```
  smtpuser = user@gmail.com
  smtpserverport = 587
```

Отправить патчи можно командой `git send-email`:

```
$ git send-email *.patch
0001-add-limit-to-log-function.patch
0002-increase-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
```

Во время выполнения команды, Git выводит много отладочной информации для каждого отправляемого патча, которая выглядит примерно так:

```
(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] Add 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
```

Помощь по конфигурации, дополнительные советы и рекомендации, а также тестовое окружение для отправки патчей по email доступны здесь git-send-email.io.

Заключение

В этом разделе мы рассмотрели ряд общепринятых схем рабочих процессов и поговорили о различиях между работой в составе небольшой команды над проектами с закрытым исходным кодом и участием в большом публичном проекте. Вы знаете, что нужно проверять ошибки в пробелах перед созданием коммита и можете написать отличное сообщение коммита. Вы научились оформлять патчи и отправлять их по электронной почте в список рассылки разработчиков. Работа со слияниями также была покрыта в контексте различных рабочих процессов. Теперь вы хорошо подготовлены для совместной работы над любым проектом.

Далее рассмотрим ситуацию с другой стороны: как сопровождать проект Git. Вы узнаете,

как быть великодушным диктатором или менеджером по интеграции.

Сопровождение проекта

В дополнение к эффективному участию в проекте, было бы неплохо знать как его сопровождать. Сопровождение может включать в себя принятие и применение патчей, сгенерированных с помощью `format-patch` и отправленных вам по почте, или интеграцию изменений в ветках удалённых репозиториев. Независимо от того, поддерживаете ли вы канонический репозиторий или просто хотите помочь в проверке или применении патчей, вам необходимо знать каким образом следует принимать работу, чтобы это было наиболее понятно для других участников и было бы приемлемым для вас в долгосрочной перспективе.

Работа с тематическими ветками

Перед интеграцией новых изменений желательно проверить их в тематической ветке — временной ветке, специально созданной для проверки работоспособности новых изменений. Таким образом, можно применять патчи по одному и пропускать неработающие, пока не найдётся время к ним вернуться. Если вы создадите ветку с коротким и понятным названием, основанным на тематике изменений, например, `ruby_client` или что-то похожее, то без труда можно будет вернуться к ней, если пришлось на какое-то время отказаться от работы с ней. Сопровождающему Git проекта свойственно использовать пространство имён для веток, например, `sc/ruby_client`, где `sc` — это сокращение от имени того, кто проделал работу. Как известно, ветки можно создавать на основании базовой ветки, например:

```
$ git branch sc/ruby_client master
```

Если вы хотите сразу переключиться на создаваемую ветку, то используйте опцию `checkout -b`:

```
$ git checkout -b sc/ruby_client master
```

Теперь вы можете добавить новые изменения в созданную тематическую ветку и определить хотите ли слить эти изменения в ваши долгосрочные ветки.

Применение патчей, полученных по почте

Если вы получили патч по почте и его нужно интегрировать в проект, то следует проанализировать его, применив сначала в тематической ветке. Существует два варианта применения полученного по почте патча: `git apply` или `git am`.

Применение патча командой apply

Если полученный по почте патч был создан командой `git diff` или Unix командой `diff` (что не рекомендуется делать), то применить его можно командой `git apply`. Предположим, патч

сохранён здесь `/tmp/patch-ruby-client.patch`, тогда применить его можно вот так:

```
$ git apply /tmp/patch-ruby-client.patch
```

Это действие модифицирует файлы в вашем рабочем каталоге. Выполнение команды практически эквивалентно выполнению команды `patch -p1`, однако, является более параноидальным и принимает меньше неточных совпадений, чем `patch`. При этом обрабатываются добавления, удаления и переименования файлов, указанные в формате `git diff`, тогда как `patch` этого не делает. Наконец, `git apply` использует модель «применить всё или отменить всё», где изменения либо применяются полностью, либо не применяются вообще, тогда как `patch` может частично применить патч файлы, приведя ваш рабочий каталог в непонятное состояние. В целом, `git apply` более консервативен, чем `patch`. После выполнения команды новый коммит не создаётся и его нужно делать вручную.

Командой `git apply` можно проверить корректность применения патча до его фактического применения, используя `git apply --check`:

```
$ git apply --check 0001-see-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

Если ошибок не выведено, то патч может быть применён без проблем. Так же, в случае ошибки эта команда возвращает отличное от 0 значение, что позволяет использовать её в скриптах.

Применение патча командой `am`

Если участник проекта пользователь Git и умеет пользоваться командой `format-patch` для генерации патчей, то вам будет легче, так как в патч включается информация об авторе и сообщение коммита. Если возможно, требуйте от ваших участников использовать команду `format-patch` вместо `diff` для генерации патчей. Вам останется использовать `git apply` только для устаревших патчей и подобного им.

Для применения патча, созданного с помощью `format-patch`, используйте `git am` (команда названа `am` потому что применяет «apply» набор патчей в формате «mailbox»). С технической точки зрения она просто читает mbox-файл, в котором в виде обычного текста хранится одно или несколько электронных писем. Этот файл имеет следующий вид:

```
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
```

Это начало вывода команды `format-patch`, которая рассматривалась в предыдущем разделе;

это так же представляет собой валидный формат mbox. Если кто-то отправил патч, корректно сформированный командой `git send-email`, и вы сохранили его в формате mbox, то можно указать передать этот файл в качестве аргумента команде `git am`, которая начнёт применять все найденные в файле патчи. Если вы используете почтовый клиент, который умеет сохранять несколько писем в формате mbox, то можно сохранить сразу серию патчей в один файл, а затем применить их за раз, используя `git am`.

Так или иначе, если кто-нибудь загрузит созданный с помощью `format-patch` патч файл в систему управления задачами, то вы сможете сохранить его себе и применить локально с помощью `git am`:

```
$ git am 0001-limit-log-function.patch
Applying: Add limit to log function
```

Как вы могли заметить, патч применился без конфликтов, а так же был создан новый коммит. Информация об авторе была извлечена из заголовков письма `From` и `Date`, а сообщение коммита — из заголовка `Subject` и тела письма (до патча). Например, для применённого патча из примера выше коммит будет выглядеть следующим образом:

```
$ 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 <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

   Add limit to log function

   Limit log functionality to the first 20
```

`Commit` информация указывает на того, кто применил патч и когда это было сделано. `Author` информация указывает на того, кто изначально создал патч и когда это было сделано.

Однако, возможна ситуация, когда патч не может быть бесконфликтно применён. Возможно, ваша основная ветка сильно расходится с той веткой, на основании которой был создан патч, или он зависит от другого, ещё не применённого патча. В таком случае работа `git am` будет прервана, а так же выведена подсказка со списком возможных действий:

```
$ git am 0001-see-if-this-helps-the-gem.patch
Applying: See 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".
```

Эта команда добавит соответствующие маркеры во все файлы где есть конфликты, аналогично конфликтам слияния или перебазирования. Для решения такой проблемы используется аналогичный подход — отредактируйте файлы исправив конфликты, добавьте их в индекс и выполните команду `git am --resolved` для перехода к следующему патчу:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: See if this helps the gem
```

При желании, вы можете указать опцию `-3`, чтобы Git попробовал провести трёхстороннее слияние. Эта опция не включена по умолчанию, так как она не будет работать, если коммит, на который ссылается патч, отсутствует в вашем репозитории. Если у вас есть тот коммит, на который ссылается конфликтующий патч, то использование опции `-3` приводит к более умному применению конфликтующего патча:

```
$ git am -3 0001-see-if-this-helps-the-gem.patch
Applying: See 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.
```

В данном случае, без использования опции `-3` патч будет расценён как конфликтующий. При использовании опции `-3` патч будет применён без конфликтов.

Если вы применяете несколько патчей из файла mbox, то можно запустить `git am` в интерактивном режиме, в котором перед обработкой каждого патча будет задаваться вопрос о дальнейших действиях:

```
$ git am -3 -i mbox
Commit Body is:
--------------------------
See if this helps the gem
--------------------------
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

Это отличная возможность посмотреть содержимое патча перед его применением или пропустить его, если он уже был применён.

Когда все патчи применены и созданы коммиты в текущей ветке, вы уже можете решить стоит ли и как интегрировать их в более долгоживущую ветку.

Извлечение удалённых веток

Если участник проекта создал свой Git репозиторий, отправил в него свои изменения, а затем прислал вам ссылку и название ветки, куда были отправлены изменения, то вы можете добавить этот репозиторий как удалённый и провести слияние локально.

К примеру, Джессика отправила вам письмо, в котором сказано, у неё есть новый функционал в ветке `ruby-client` её репозитория. Добавив удалённый репозиторий и получив изменения из этой ветки, вы можете протестировать изменения извлекая их локально:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Если она снова пришлёт вам письмо с указанием на новый функционал уже в другой ветке, то для его получения достаточно `fetch` и `checkout`, так как удалённый репозиторий уже подключён.

Это очень полезно, когда вы постоянно работаете с этим человеком. Однако, от тех, кто редко отправляет небольшие патчи, будет проще принимать их по почте, чем требовать от всех поддержания собственных серверов с репозиториями, постоянно добавлять их как удалённые, а затем удалять и всё это ради нескольких патчей. Так же вы вряд ли захотите иметь сотни удалённых репозиториев, каждый из которых нужен только для одного или нескольких патчей. К счастью, скрипты и различные сервисы облегчают задачу, но во многом зависят от того как работаете вы и участники вашего проекта.

Отличительным преимуществом данного подхода является получение истории коммитов. Хоть возникновение конфликтов слияния и закономерно, но вы знаете с какого момента это произошло; корректное трёхстороннее слияние более предпочтительно, чем указать опцию `-3` и надеяться, что патч основан на коммите, к которому у вас есть доступ.

Если вы с кем-то не работаете постоянно, но всё равно хотите использовать удалённый репозиторий, то можно указать ссылку на него в команде `git pull`. Это приведёт к однократному выполнению, а ссылка на репозиторий сохранена не будет.

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
 * branch            HEAD       -> FETCH_HEAD
Merge made by the 'recursive' strategy.
```

Определение применяемых изменений

На текущий момент у вас есть тематическая ветка, содержащая предоставленные изменения. Сейчас вы можете определиться что с ними делать. В этом разделе рассматривается набор команд, которые помогут вам увидеть что именно будет интегрировано, если вы решите слить изменения в основную ветку.

Обычно, полезно просмотреть все коммиты текущей ветки, которые ещё не включены в основную. Вы можете исключить коммиты, которые уже есть в вашей основной ветке добавив опцию `--not` перед её названием. Это аналогично указанию использовавшегося ранее формата `master..contrib`. Например, если участник проекта отправил вам два патча, а вы создали ветку с названием `contrib` и применили их, то можно выполнить следующую команду:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    See if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    Update gemspec to hopefully work better
```

Для просмотра изменений, представленных в каждом коммите, можно использовать опцию `-p` команды `git log`, которая выведет разницу по каждому коммиту.

Для просмотра полной разницы того, что произойдёт если вы сольёте изменения в другую ветку, вам понадобится использовать возможно странный способ для получения корректных результатов:

```
$ git diff master
```

Эта команда может вводить в заблуждение, но точно покажет разницу. Если ваша `master` ветка продвинулась вперёд с тех пор как вы создали тематическую ветку, то вы получите на первый взгляд странные результаты. Это происходит потому, что Git непосредственно сравнивает снимки последних коммитов текущей и `master` веток. Например, если вы добавили строку в файл в ветке `master`, то прямое сравнение снимков будет выглядеть как будто тематическая ветка собирается удалить эту строку.

Это не проблема, если ветка `master` является непосредственным родителем вашей тематической ветки, но если история обоих веток изменилась, то разница будет выглядеть как добавление всех изменений из тематической ветки и удаление всего нового из `master` ветки.

Что действительно нужно видеть, так это изменения тематической ветки, которые предстоит слить в `master` ветку. Это можно сделать, сказав Git сравнивать последний коммит тематической ветки с первым общим родителем для обоих веток.

Технически это делается за счёт явного указания общего коммита и применения разницы к нему:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

или более кратко:

```
$ git diff $(git merge-base contrib master)
```

Однако это не удобно, поэтому Git предоставляет более короткий способ: синтаксис троеточия. При выполнении команды `diff`, следует поставить три точки после имени ветки для получения разницы между ней и текущей веткой, относительно общего родителя с другой веткой:

```
$ git diff master...contrib
```

Данная команда отобразит проделанную работу только из тематической ветки, относительно общего родителя с веткой `master`. Полезно запомнить указанный синтаксис.

Интеграция совместной работы

Когда все изменения в текущей тематической ветке готовы к интеграции с основной веткой, возникает вопрос как это сделать. Кроме этого, какой рабочий процесс вы хотите использовать при сопровождении вашего проекта? У вас несколько вариантов, давайте рассмотрим некоторые из них.

Схемы слияния

В простом рабочем процессе проделанная работа просто сливается в ветку `master`. При таком сценарии у вас есть ветка `master`, которая содержит стабильный код. Когда работа в тематической ветке завершена или вы проверили чью-то работу, вы сливаете её в ветку `master` и удаляете, затем процесс повторяется.

Если в репозитории присутствуют две ветки `ruby_client` и `php_client` с проделанной работой, как показано на рисунке История с несколькими тематическими ветками, и вы сначала сливаете ветку `ruby_client`, а затем `php_client`, то состояние вашего репозитория будет выглядеть как показано на рисунке Слияние тематической ветки.

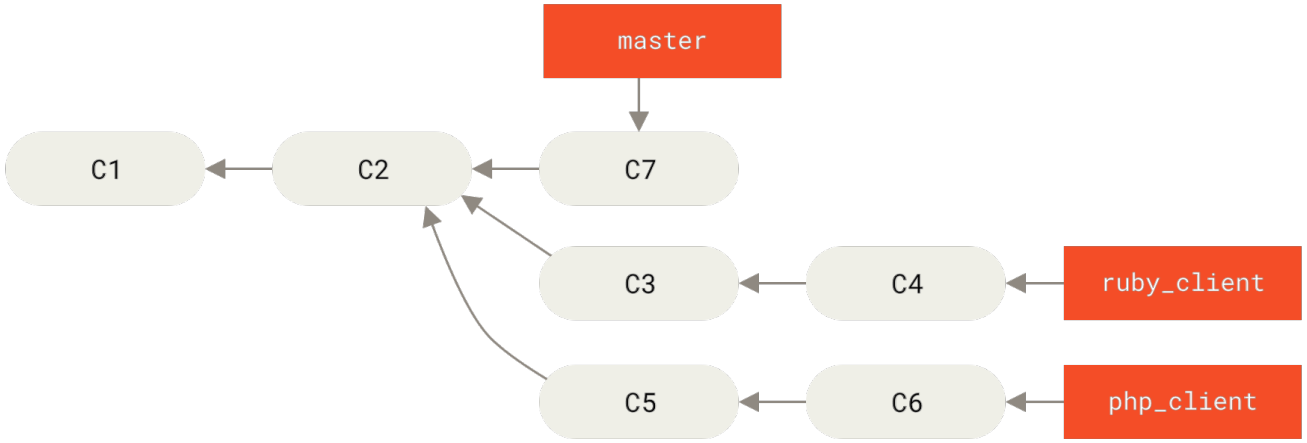


Рисунок 72. История с несколькими тематическими ветками

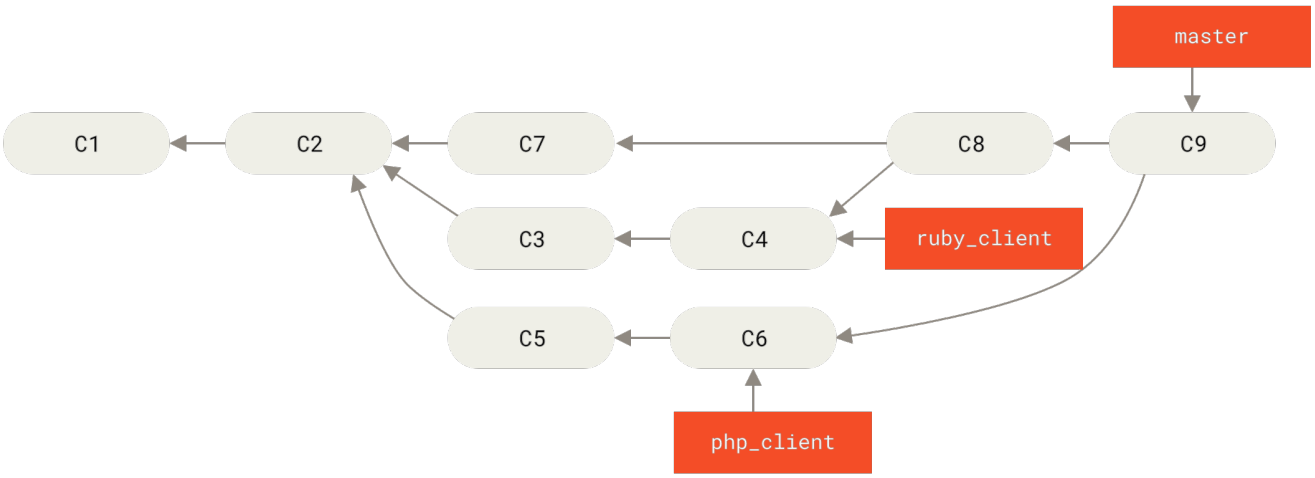


Рисунок 73. Слияние тематической ветки

Это, пожалуй, простейший рабочий процесс и его использование проблематично в больших или более стабильных проектах, где вы должны быть более осторожны с предоставленными изменениями.

Если у вас очень важный проект, то возможно вам стоит использовать двухступенчатый цикл слияния. При таком сценарии у вас имеются две долгоживущие ветки `master` и `develop`, где в `master` сливаются только очень стабильные изменения, а все новые доработки интегрируются в ветку `develop`. Обе ветки регулярно отправляются в публичный репозиторий. Каждый раз, когда новая тематическая ветка готова к слиянию (Перед слиянием тематической ветки), вы сливаете её в `develop` (После слияния тематической ветки); затем, когда вы выпускаете релиз, ветка `master` смещается на стабильное состояние ветки `develop` (После релиза проекта).

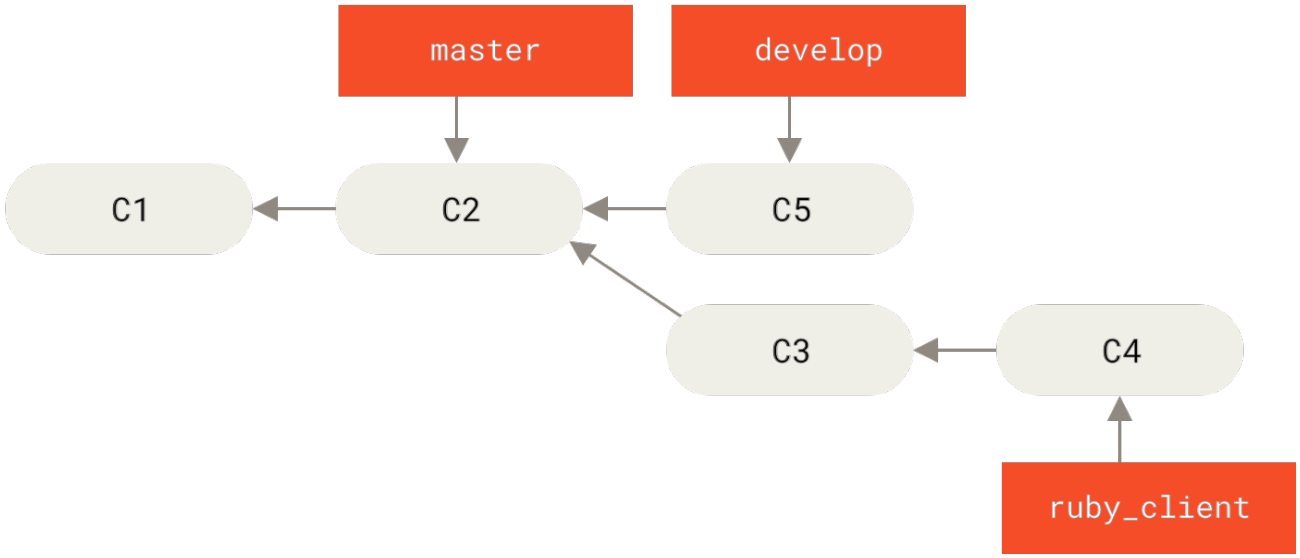


Рисунок 74. Перед слиянием тематической ветки

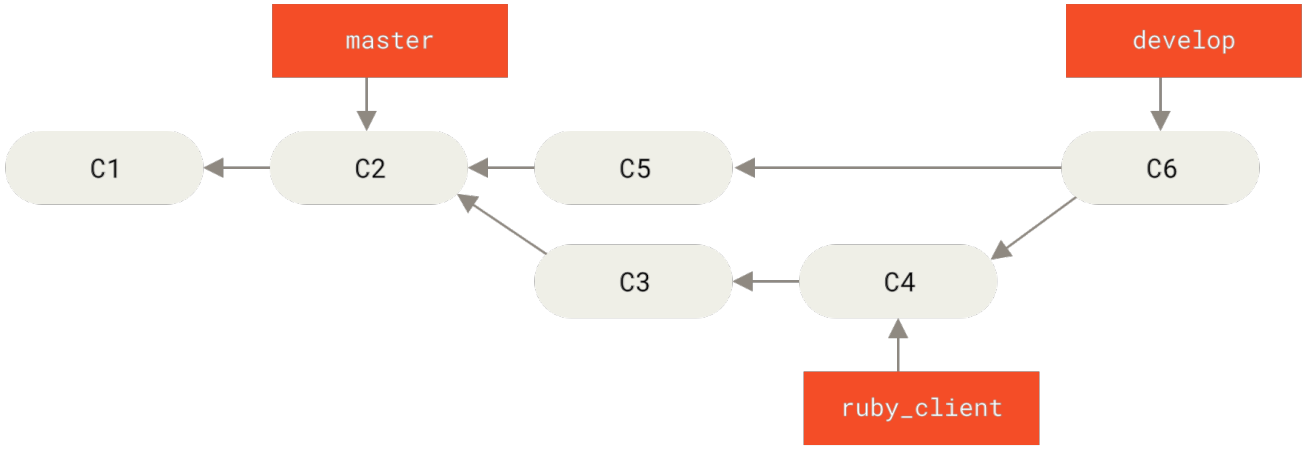


Рисунок 75. После слияния тематической ветки

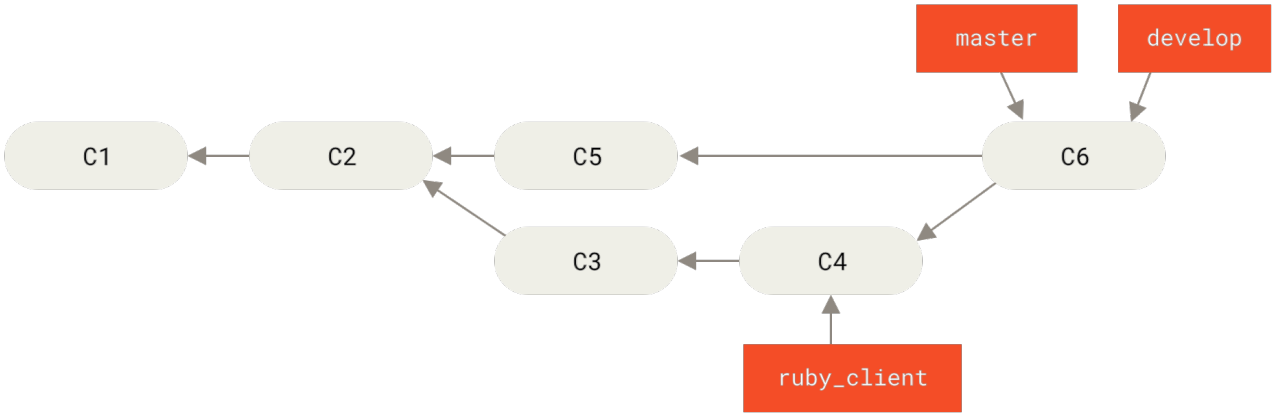


Рисунок 76. После релиза проекта

Таким образом, люди могут клонировать репозиторий вашего проекта и использовать ветку `master` для сборки последнего стабильного состояния и получения актуальных изменений или использовать ветку `develop`, которая содержит самые последние изменения. Вы также можете продолжить эту концепцию, имея интеграционную ветку `integrate`, в которой объединяется вся работа. После того, как кодовая база указанной ветки стабильна и

пройдены все тесты, она сливается в ветку develop, а после того, как стабильность слитых изменений доказана, вы перемещаете состояние ветки master на стабильное.

Схема с большим количеством слияний

В проекте Git присутствуют четыре долгоживущие ветки: master, next, seen (ранее pu — предложенные обновления) для новой работы и maint для поддержки обратной совместимости. Предложенные участниками проекта наработки накапливаются в тематических ветках основного репозитория по ранее описанному принципу (рис. Управление сложным набором параллельно разрабатываемых тематических веток). На этом этапе производится оценка содержимого тематических веток, чтобы определить, работают ли предложенные фрагменты так, как положено, или им требуется доработка. Если всё в порядке, тематические ветки сливаются в ветку next, которая отправляется на сервер, чтобы у каждого была возможность опробовать результат интеграции.

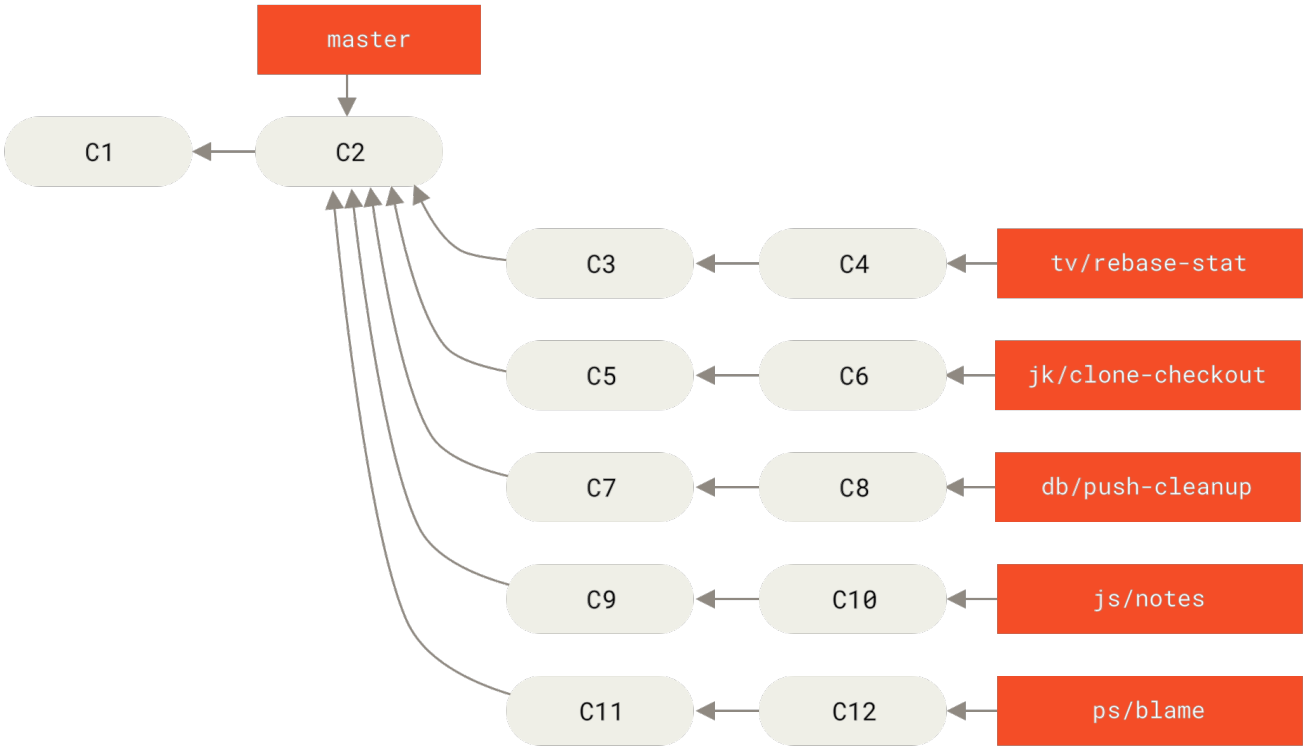


Рисунок 77. Управление сложным набором параллельно разрабатываемых тематических веток

Если содержимое тематических веток требует доработки, оно сливается в ветку seen. Когда выясняется, что предложенный код полностью стабилен, он сливается в ветку master. Затем ветки next и seen перестраиваются на основании master. Это означает, что master практически всегда двигается только вперёд, next время от времени перебазируется, а seen перебазируется ещё чаще:

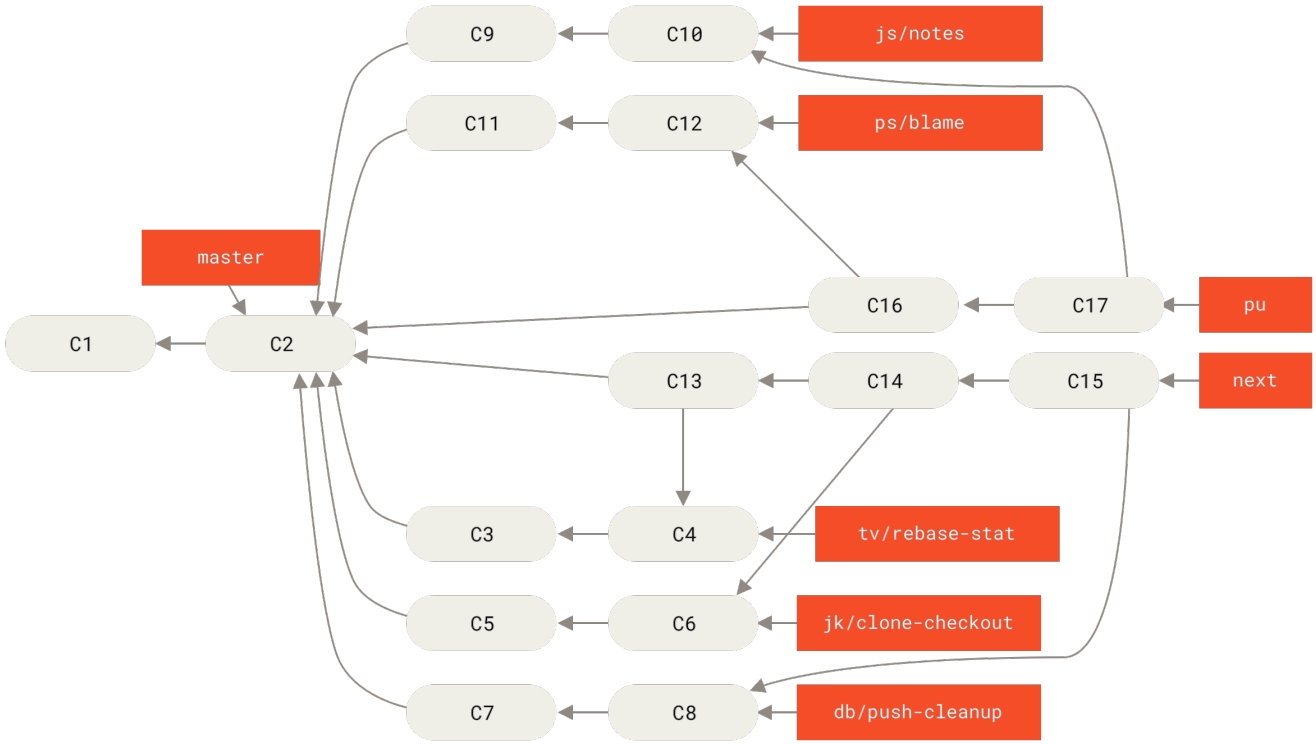


Рисунок 78. Слияние тематических веток в долгоживущие ветки интеграции

После того, как тематическая ветка окончательно слита в `master`, она удаляется из репозитория. Репозиторий также содержит ветку `maint`, которая ответвляется от последнего релиза для предоставления патчей, если требуется поддержка обратной совместимости. Таким образом, после клонирования проекта у вас будет четыре ветки, дающие возможность перейти на разные стадии его разработки, в зависимости от того, на сколько передовым вы хотите быть или как вы собираетесь участвовать в проекте; вместе с этим, рабочий процесс структурирован, что помогает сопровождающему проекта проверять поступающий код. Рабочий процесс проекта Git специфицирован. Для полного понимания процесса обратитесь к Git Maintainer's guide.

Схема с перебазированием и отбором

Некоторые сопровождающие предпочитают перебазировать или выборочно применять (cherry-pick) изменения относительно ветки `master` вместо слияния, что позволяет поддерживать историю проекта в линейном виде. Когда проделанная работа из тематической ветки готова к интеграции, вы переходите на эту ветку и перебазируете её относительно ветки `master` (или `develop` и т. д.). Если конфликты отсутствуют, то вы можете просто сдвинуть состояние ветки `master`, что обеспечивает линейность истории проекта.

Другим способом переместить предлагаемые изменений из одной ветки в другую является их отбор коммитов (cherry-pick). Отбор в Git похож на перебазирование для одного коммита. В таком случае формируется патч для выбранного коммита и применяется к текущей ветке. Это полезно, когда в тематической ветке присутствует несколько коммитов, а вы хотите взять только один из них, или в тематической ветке только один коммит и вы предпочитаете использовать отбор вместо перебазирования. Предположим, ваш проект выглядит так:

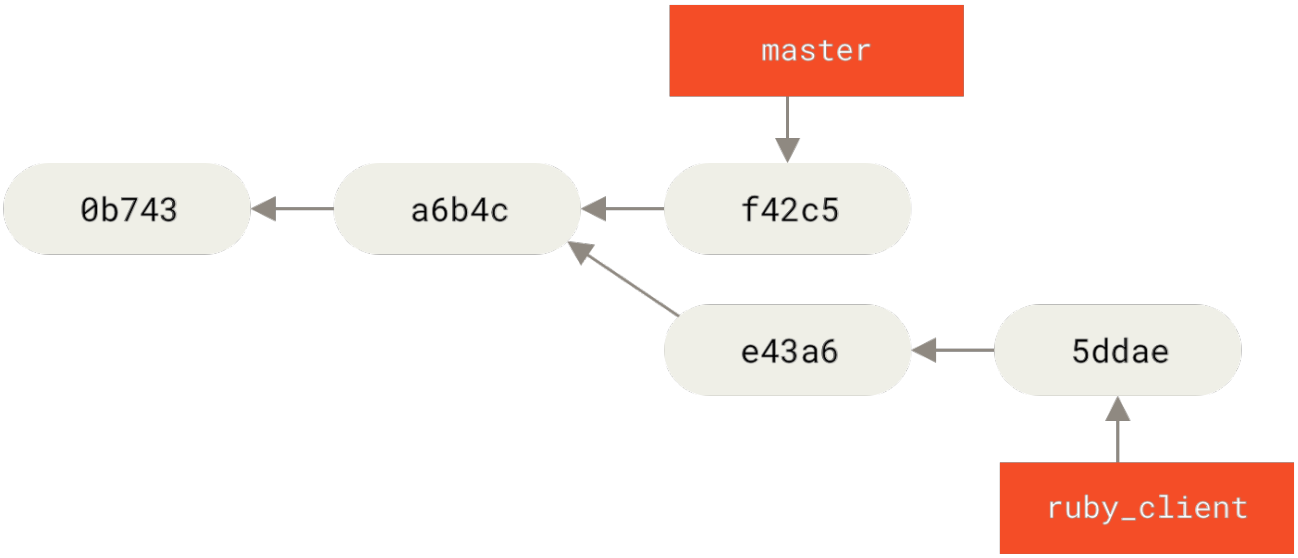


Рисунок 79. Пример истории, из которой нужно отобрать отдельные коммиты

Для применения коммита `e43a6` к ветке `master` выполните команду:

```
$ git cherry-pick e43a6
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

Это действие применит изменения, содержащиеся в коммите `e43a6`, но будет сформирован новый коммит с другим значением SHA-1. После этого история будет выглядеть так:

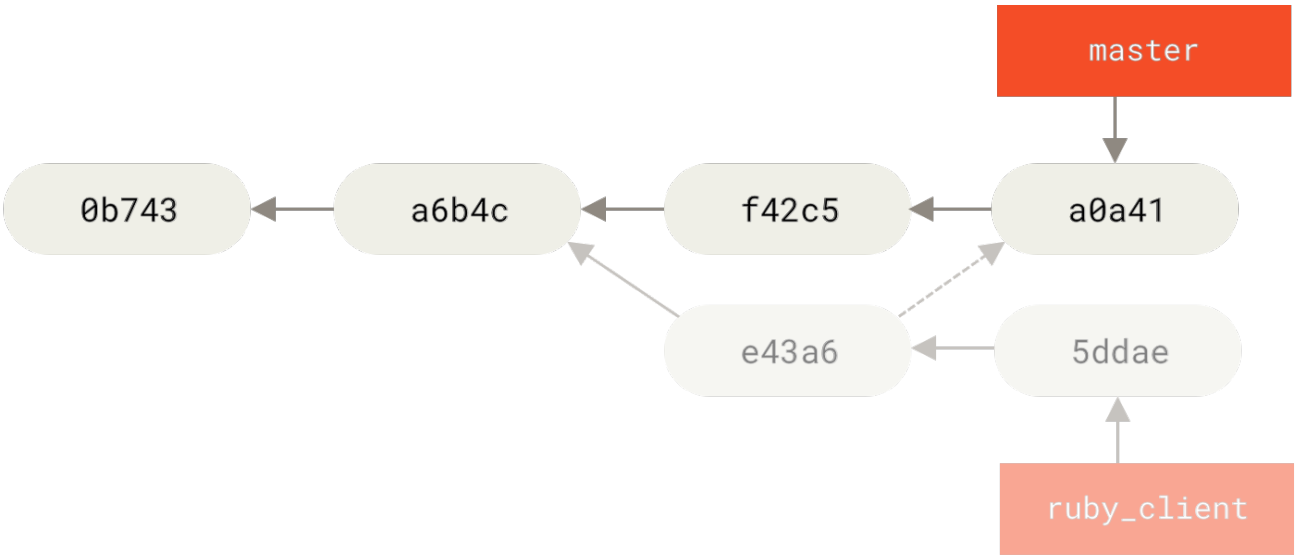


Рисунок 80. История после отбора коммита из тематической ветки

Теперь тематическую ветку можно удалить, отбросив коммиты, которые вы не собираетесь включать в проект.

Возможность «Rerere»

Если вы часто производите перебазирование и слияние или поддерживаете долгоживущие тематические ветки, то в Git есть специальная возможность под названием «rerere», призванная вам помочь.

Rerere означает «reuse recorded resolution» (повторно использовать сохранённое решение) — это способ сокращения количества операций ручного разрешения конфликтов. Когда эта опция включена, Git будет сохранять набор образов до и после успешного слияния, а также разрешать конфликты самостоятельно, если аналогичные конфликты уже были разрешены ранее.

Эта возможность реализована как команда и как параметр конфигурации. Параметр конфигурации называется `rerere.enabled`, который можно включить глобально следующим образом:

```
$ git config --global rerere.enabled true
```

После этого любое разрешение конфликта слияния будет записано на случай повторного использования.

Если нужно, вы можете обращаться к кэшу «rerere» напрямую, используя команду `git rerere`. Когда команда вызвана без параметров, Git проверяет базу данных и пытается найти решение для разрешения текущего конфликта слияния (точно так же как и при установленной настройке `rerere.enabled` в значение `true`). Существует множество дополнительных команд для просмотра, что именно будет записано, удаления отдельных записей из кэша, а так же его полной очистки. Более детально «rerere» будет рассмотрено в разделе Rerere главы 7.

Помечайте свои релизы

После выпуска релиза, возможно, вы захотите пометить текущее состояние так, чтобы можно было вернуться к нему в любой момент. Для этого можно добавить тег, как было описано в главе Основы Git. Кроме этого, вы можете добавить цифровую подпись для тега, выглядеть это будет вот так:

```
$ 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
```

Если вы используете цифровую подпись при расстановке тегов, то возникает проблема распространения публичной части PGP ключа, использованного при создании подписи. Сопровождающий Git проекта может решить эту проблему добавив в репозиторий свой публичный ключ как бинарный объект и установив ссылающийся на него тег. Чтобы это сделать, выберите нужный ключ из списка доступных, который можно получить с помощью команды `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]
```

Затем экспортируйте выбранный ключ и поместите его непосредственно в базу данных Git при помощи команды `git hash-object`, которая создаст новый объект с содержимым ключа и вернёт SHA-1 этого объекта:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Теперь, когда ваш публичный ключ находится в репозитории, можно поставить указывающий на него тег, используя полученное ранее значение SHA-1:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Выполнив команду `git push --tags`, `maintainer-pgp-pub` тег станет общедоступным. Теперь все, кто захочет проверить вашу подпись, могут импортировать ваш публичный ключ, предварительно получив его из репозитория:

```
$ git show maintainer-pgp-pub | gpg --import
```

После этого можно проверять цифровую подпись ваших тегов. Кроме этого, вы можете включить дополнительные инструкции по проверке вашей подписи в сообщение тега, которое будет отображаться каждый раз при вызове команды `git show <tag>`.

Генерация номера сборки

Git не использует монотонно возрастающие идентификаторы для коммитов, поэтому если вы хотите получить читаемые имена коммитов, то воспользуйтесь командой `git describe` для нужного коммита. Git вернёт имя ближайшего тега, количество коммитов после него и частичное значение SHA-1 для указанного коммита (с префиксом в виде буквы «g» — означает Git):

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

Таким образом, вы можете сделать снимок или собрать сборку и дать ей понятное для человека название. К слову, если вы клонируете репозиторий Git и соберёте его из исходного кода, то вывод команды `git --version` будет примерно таким же. Если попытаться получить имя коммита, которому назначен тег, то результатом будет название самого тега.

По умолчанию, команда `git describe` поддерживает только аннотированные теги (созданные с использованием опций `-a` или `-s`); если вы хотите использовать легковесные (не аннотированные) теги, то укажите команде параметр `--tags`. Также это название можно использовать при выполнении команд `git checkout` и `git show`, но в будущем они могут перестать работать из-за сокращённого значения SHA-1. К примеру, ядро Linux недавно перешло к использованию 10 символов в SHA-1 вместо 8 чтобы обеспечить уникальность каждого объекта, таким образом предыдущие результаты `git describe` стали недействительными.

Подготовка релиза

Время делать релиз сборки. Возможно, вы захотите сделать архив последнего состояния вашего кода для тех, кто не использует Git. Для создания архива выполните команду `git archive`:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Открывший этот tarball-архив пользователь получит последнее состояние кода проекта в каталоге `project`. Точно таким же способом можно создать zip-архив, просто добавив опцию `--format=zip` для команды `git archive`:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

В итоге получим tarball- и zip-архивы с релизом проекта, которые можно загрузить на сайт или отправить по почте.

Краткая история (Shortlog)

Сейчас самое время оповестить людей из списка рассылки, которые хотят знать что происходит с вашим проектом. С помощью команды `git shortlog` можно быстро получить список изменений, внесённых в проект с момента последнего релиза или предыдущей рассылки. Она собирает все коммиты в заданном интервале; например, следующая команда выведет список коммитов с момента последнего релиза с названием v1.0.1:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (6):
      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
```

И так, у вас есть готовая к отправке сводка коммитов начиная с версии v1.0.1, сгруппированных по авторам.

Заключение

Теперь вы должны чувствовать себя достаточно свободно как внося свой вклад в проект под управлением Git, так и занимаясь поддержкой своего собственного проекта или интегрированием наработок других пользователей. Поздравляем, вы опытный Git-разработчик! В следующей главе вы узнаете о том, как использовать самый большой и самый популярный Git хостинг — GitHub.

GitHub

GitHub — это крупнейшее хранилище Git репозиториев, а так же центр сотрудничества для миллионов разработчиков и проектов. Огромный процент всех репозиториев хранится на GitHub, а многие проекты с открытым исходным кодом используют его ради Git хостинга, баг-трекера, рецензирования кода и других вещей. Так что, пока всё это не часть открытого Git проекта, наверняка вы захотите, или вам придётся взаимодействовать с GitHub при профессиональном использовании Git.

Эта глава про эффективное использование GitHub. Мы разберём регистрацию, управление учётной 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 GitHub и ещё множество мелочей, который облегчат вам жизнь.

Если вас не интересует использование GitHub для размещения собственных проектов или сотрудничества с другими проектами, размещёнными на нём, вы можете смело перейти к главе Инструменты Git.

> *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 учётной записи

Первым делом нужно создать бесплатную учётную запись. Просто зайдите на https://github.com, выберите имя которое ещё не занято, укажите адрес электронной почты и пароль, а затем нажмите большую зелёную кнопку «Sign up for GitHub».

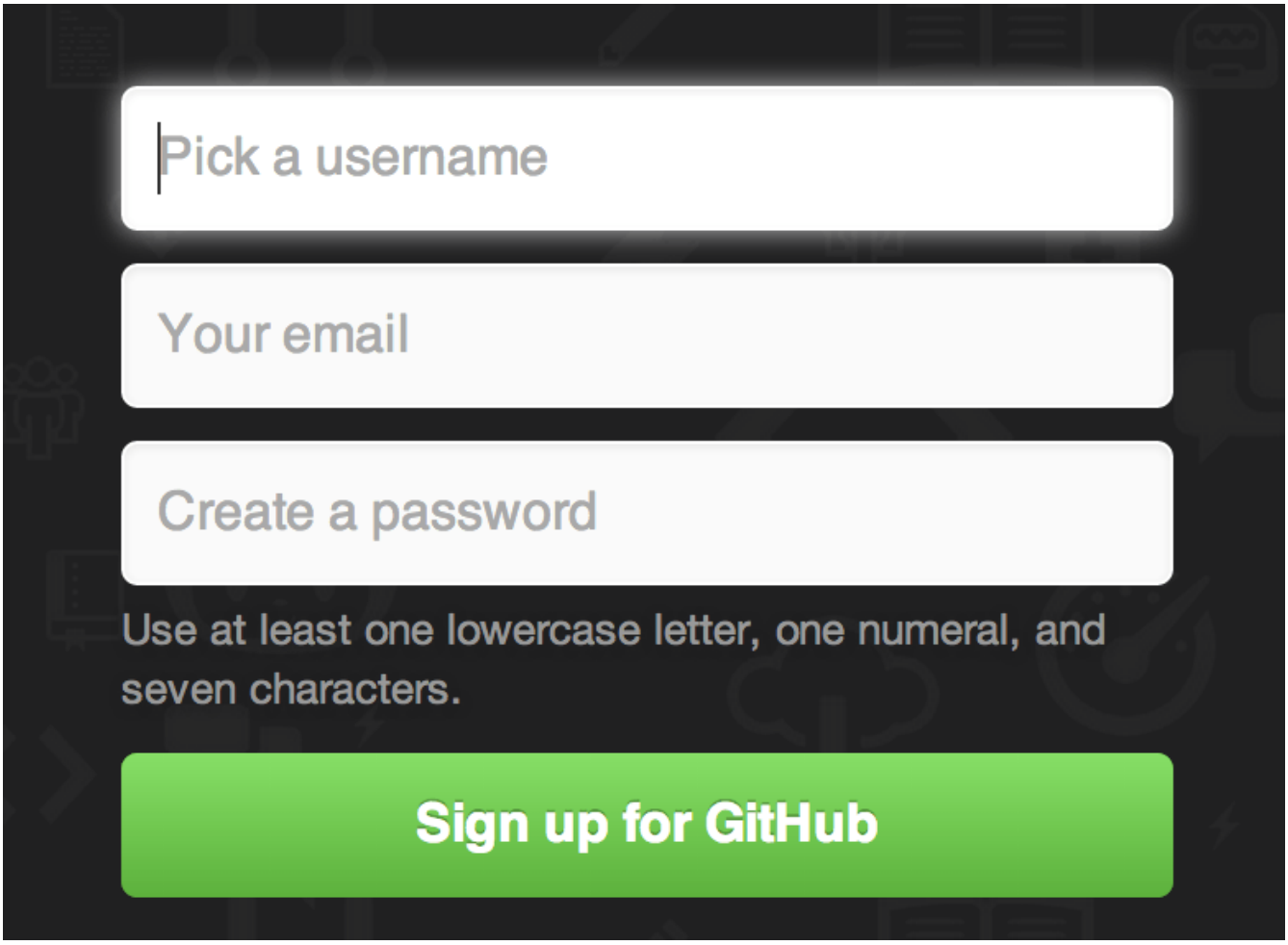


Рисунок 81. Форма регистрации на GitHub

Далее вы попадёте 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 почти все свои функции для бесплатных учётных записей, за исключением некоторых расширенных возможностей. Платные тарифы GitHub включают расширенные инструменты и функции, а также увеличенные лимиты на бесплатные услуги, но мы не будем рассматривать их в этой книге. Для того, чтобы получить более подробную информацию об имеющихся тарифах и их сравнение, посетите https://github.com/pricing.

Клик на расположенном в верхнем левом углу экрана логотипе, изображающем гибрид кота и осьминога (его называют осьмикот), откроет панель управления. Теперь всё готово для работы с GitHub.

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 авторизовываться 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) Откройте настройки вашей учётной записи при помощи ссылки, расположенной в верхнем правом углу окна:

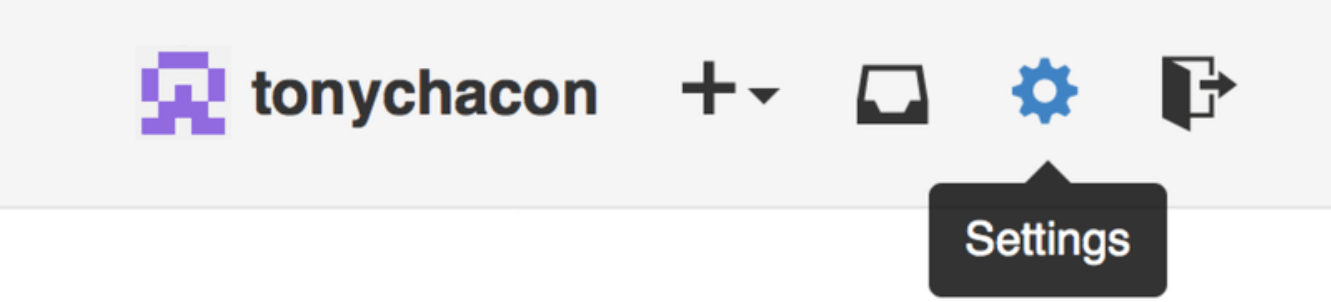


Рисунок 82. Ссылка «Настройка учётной записи» («Account settings»)

Выберите секцию слева под названием «Ключи SSH» («SSH keys»).

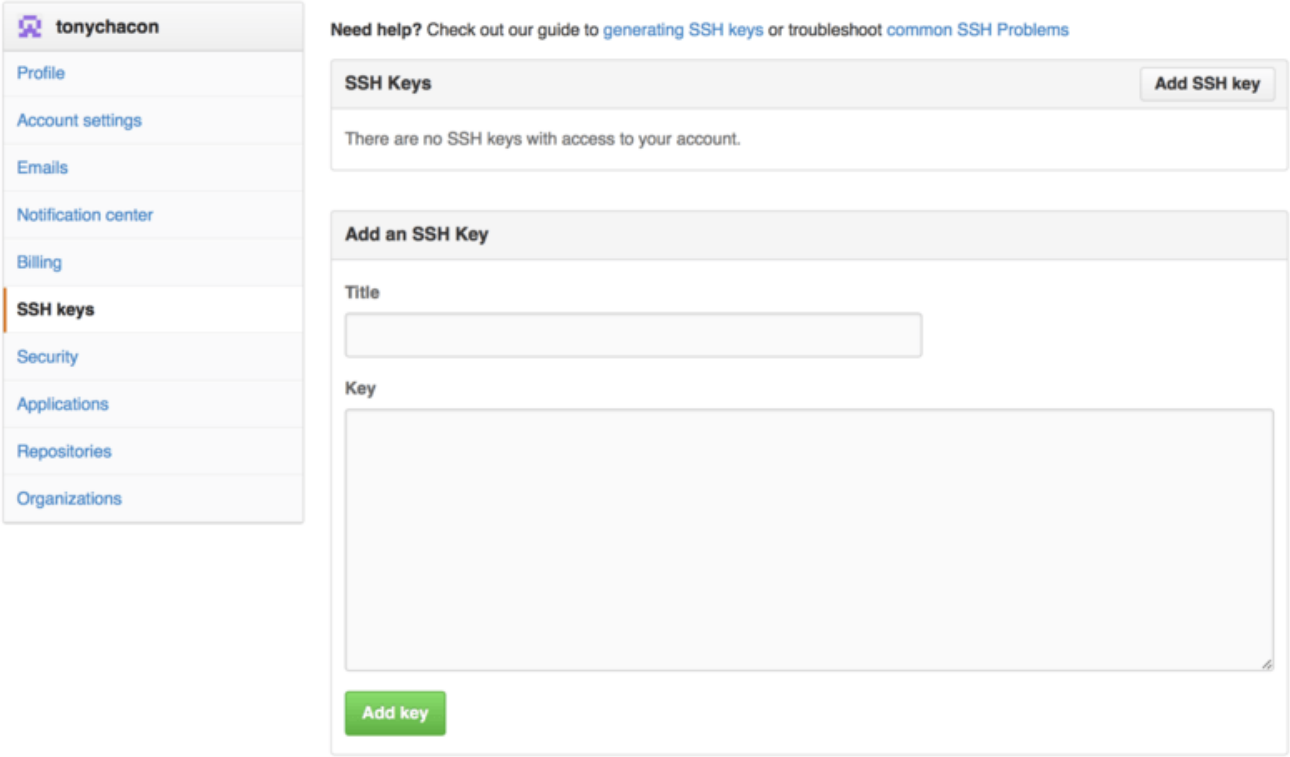


Рисунок 83. Ссылка («SSH keys»)

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»).

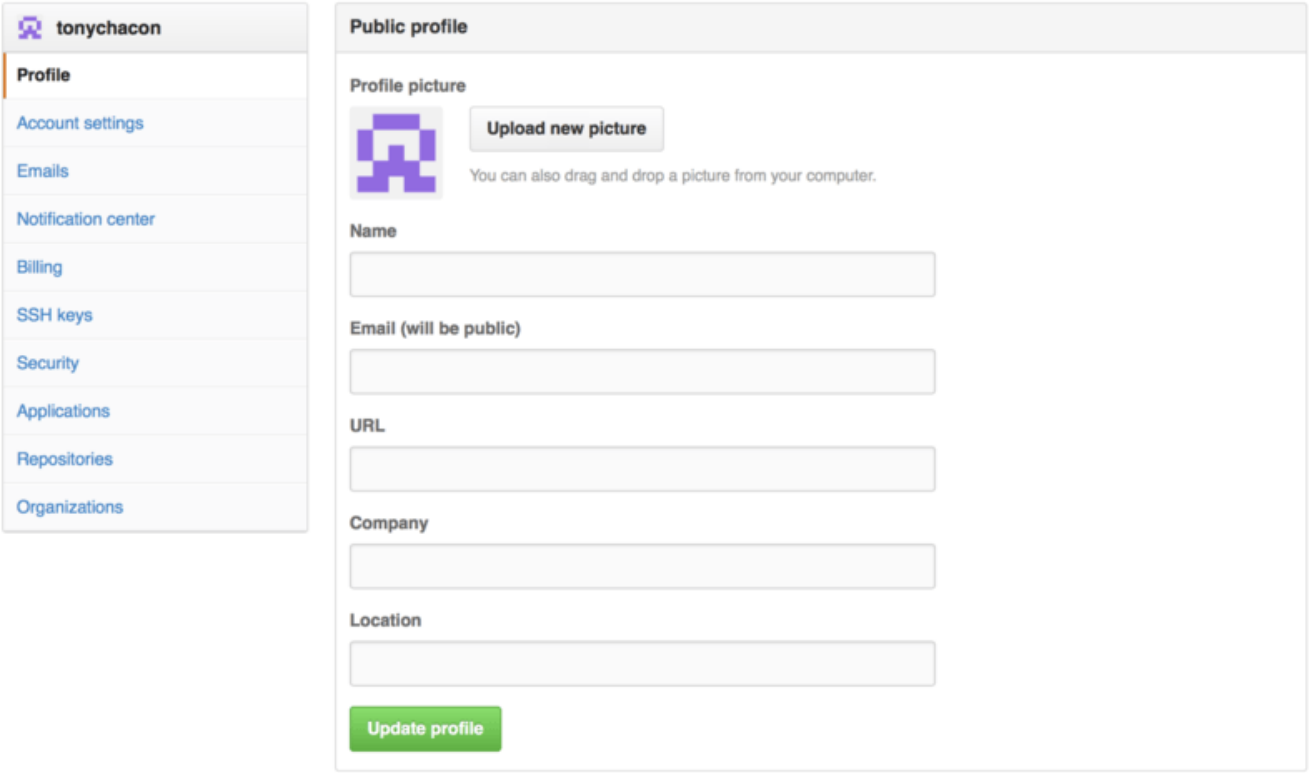


Рисунок 84. Ссылка «Профиль» («Profile»)

Выберем логотип Git с жёсткого диска и отредактируем картинку под желаемый размер.

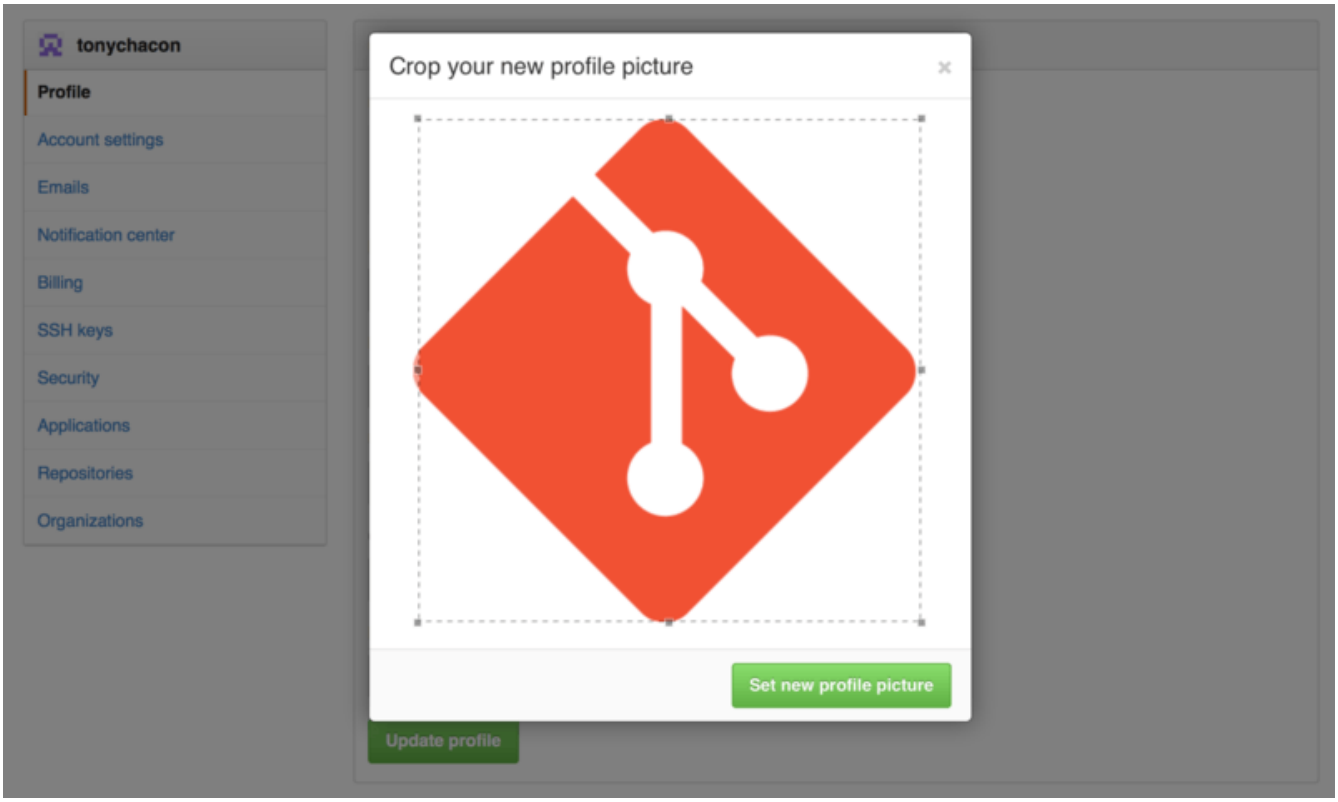


Рисунок 85. Редактирование аватара

После загрузки каждый сможет увидеть ваш аватар рядом с вашим именем пользователя.

Если вы используете такой популярный сервис как Gravatar (часто используется для учётных 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 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»).

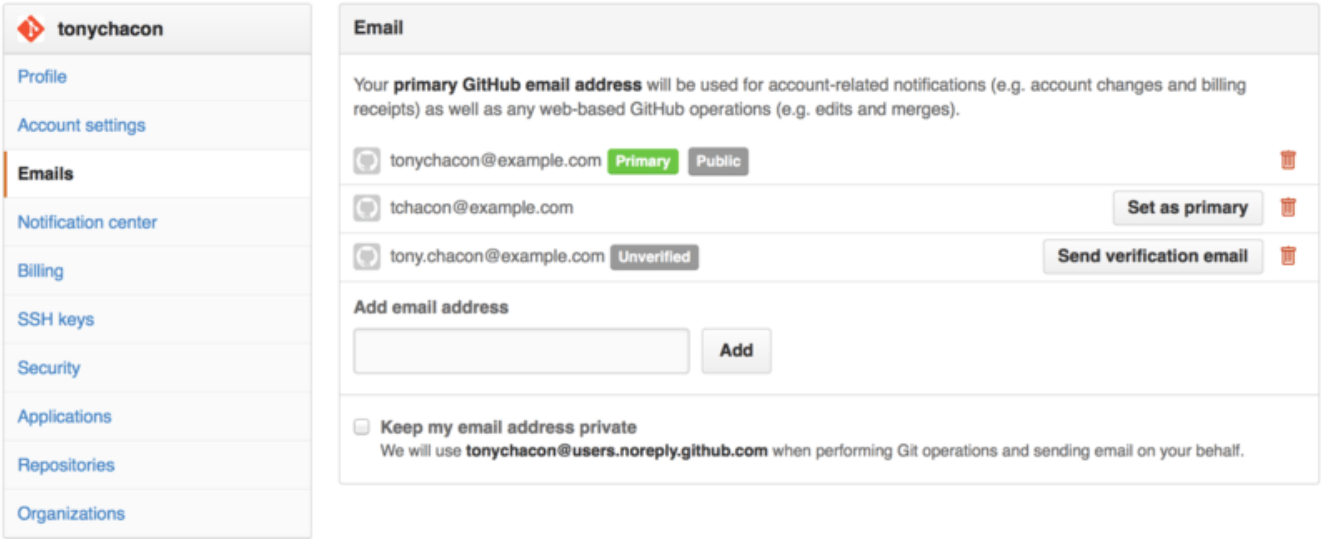


Рисунок 86. Почтовые адреса

Как можно видеть на рисунке Почтовые адреса, у почтовых адресов имеются несколько состояний. Верхний почтовый адрес подтверждён и является основным для пользователя, это тот самый адрес, куда будут направляться оповещения, а также остальные уведомления. Второй адрес тоже подтверждён, и так же может быть назначен в качестве основного. Последний адрес не подтверждён, это значит, что вы не можете использовать его в качестве основного и получать на него уведомления. При отправке коммита в любой из репозиториев, GitHub распознает один из указанных почтовых адресов и автоматически привяжет этот коммит к вашей учётной записи.

Двухфакторная аутентификация

В качестве дополнительной меры безопасности, вы можете настроить «Двухфакторную аутентификацию» («Two-factor Authentication» или «2FA»). Двухфакторная аутентификация — механизм, который становится всё более и более популярным методом по снижению риска скомпрометировать вашу учётную запись в ситуации, когда пароль от вашей учётной записи, по тем или иным причинам, стал известен злоумышленникам. Активация этого механизма заставит GitHub запрашивать у вас оба пароля при авторизации, поэтому даже в ситуациях, когда ваш основной пароль скомпрометирован, злоумышленник всё равно не получит доступ к вашей учётной записи.

Вы сможете найти настройку «Двухфакторной аутентификации» («Two-factor Authentication») в секции «Безопасность» («Security») вкладки «Настройка учётной записи» («Account settings»).

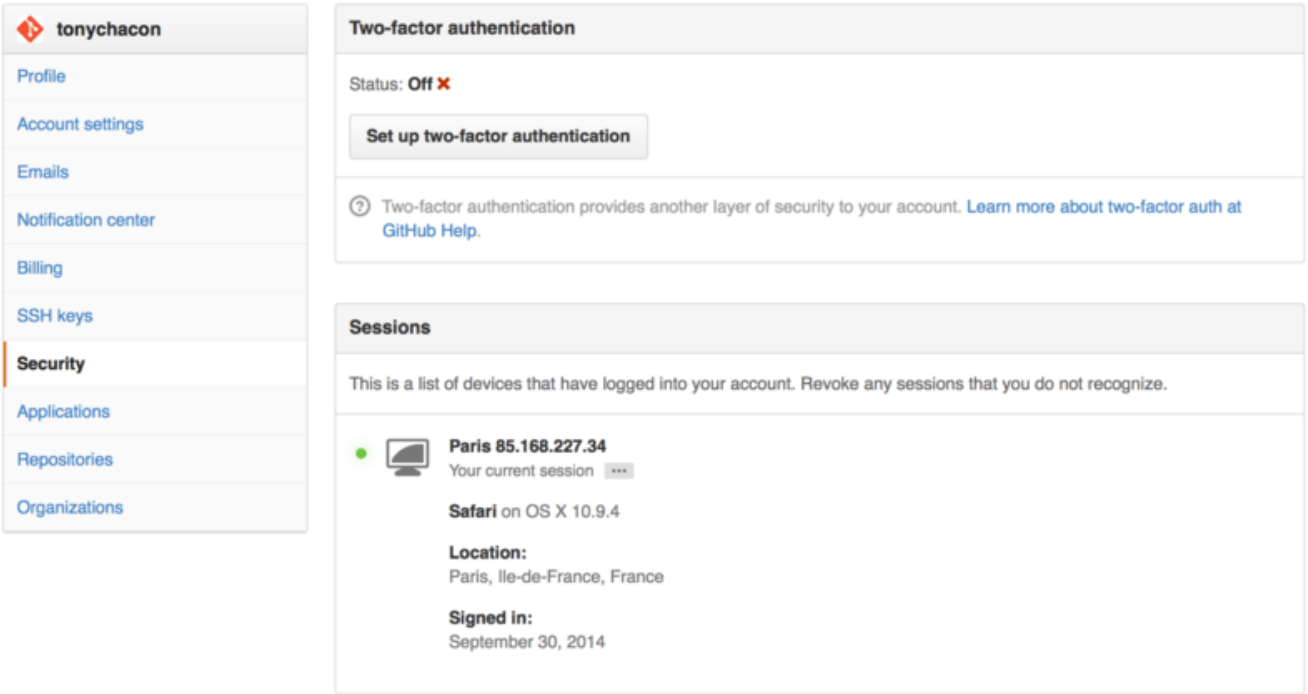


Рисунок 87. Двухфакторная аутентификация («Two-factor Authentication»)

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 предпочитаемый вами метод и выполните предлагаемые инструкции, ваша учётная запись будет в большей безопасности, и вам будет предоставляться дополнительный код во время авторизации на сайте.

Внесение собственного вклада в проекты

Теперь наша учётная запись создана и настроена, давайте же пройдёмся 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 путём отправки (push) изменений, вы можете создать своё собственное ответвление (fork) проекта. Это означает, что GitHub создаст вашу собственную копию проекта, данная копия будет находиться в вашем пространстве имён и вы сможете легко делать изменения путём отправки (push) изменений.

Исторически так сложилось, что англоязычный термин «fork» (создание ветвления проекта) имел негативный контекстный смысл, данный термин означал, что кто-то повёл или ведёт 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 имён, что позволяет вносить публичные изменения и делать свой собственный вклад в более открытом виде.

Таким образом, проекты не обеспокоены тем, чтобы пользователи, которые хотели бы выступать в роли соавторов, имели право на внесение изменений путём их отправки (push). Люди просто могут создавать свои собственные ветвления (fork), вносить туда изменения, а затем отправлять свои внесённые изменения в оригинальный репозиторий проекта путём 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 так же 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, зайдите на страницу проекта и нажмите кнопку «Создать ответвление» («Fork»), которая расположена в правом верхнем углу.

Рисунок 88. 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.

Рабочий процесс с использованием GitHub

GitHub разработан с прицелом на определённый рабочий процесс с использованием запросов на слияния. Этот рабочий процесс хорошо подходит всем: и маленьким, сплочённым вокруг одного репозитория, командам; и крупным распределённым компаниям, и группам незнакомцев, сотрудничающих над проектом с сотней копий. Рабочий процесс GitHub основан на тематических ветках, о которых мы говорили в главе Ветвление в Git.

Вот как это обычно работает:

1. Создайте форк проекта.
2. Создайте тематическую ветку на основании ветки `master`.
3. Создайте один или несколько коммитов с изменениями, улучшающими проект.
4. Отправьте эту ветку в ваш проект на GitHub.
5. Откройте запрос на слияние на GitHub.
6. Обсуждайте его, вносите изменения, если нужно.
7. Владелец проекта принимает решение о принятии изменений, либо об их отклонении.
8. Получите обновлённую ветку `master` и отправьте её в свой форк.

Очень напоминает подход, описанный в разделе Диспетчер интеграции главы 5, но вместо использования электронной почты, команда сотрудничает через веб-интерфейс.

Давайте посмотрим, как можно предложить изменения в проект, размещённый на GitHub.

В большинстве случаев можно использовать официальный инструмент **GitHub CLI** вместо веб-интерфейса GitHub. Инструмент доступен в системах Windows, MacOS и Linux. Посетите страницу GitHub CLI homepage для получения инструкций по установке и использованию.

Создание запроса на слияние

Тони ищет, чего бы запустить на своём новеньком Arduino. Кажется, он нашёл классный пример на https://github.com/schacon/blink.

schacon / blink

Watch 0 Star 0 Fork 0

branch: master blink / blink.ino

schacon on Jun 12 my arduino blinking code (from arduino.cc)

1 contributor

25 lines (20 sloc) 0.71 kb

Raw Blame History

```
1  /*
2    Blink
3    Turns on an LED on for one second, then off for one second, repeatedly.
4
5    This example code is in the public domain.
6   */
7
8  // Pin 13 has an LED connected on most Arduino boards.
9  // give it a name:
10 int led = 13;
11
12 // the setup routine runs once when you press reset:
13 void setup() {
14   // initialize the digital pin as an output.
15   pinMode(led, OUTPUT);
16 }
17
18 // the loop routine runs over and over again forever:
19 void loop() {
20   digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
21   delay(1000);               // wait for a second
22   digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
23   delay(1000);               // wait for a second
24 }
```

Рисунок 89. Проект, над которым мы хотим поработать

Единственная проблема в том, что светодиод моргает слишком быстро; нам кажется, лучше установить задержку в три секунды, не одну. Так давайте исправим это и предложим изменения автору.

Для начала, нажмите кнопку «Fork», как было сказано выше, чтобы заполучить собственную копию проекта. Мы зарегистрированы на GitHub под именем «tonychacon», так что наша копия окажется по адресу `https://github.com/tonychacon/blink`, где мы сможем редактировать её. Мы клонируем её, создадим тематическую ветку, внесём необходимые изменения и, наконец, отправим их на 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 (macOS) ③
# If you're on a Linux system, do this instead:
# $ 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 'Change delay to 3 seconds' ⑤
[slow-blink 5ca509d] Change delay to 3 seconds
 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
```

① Клонируем нашу копию

② Создаём тематическую ветку

③ Вносим свои изменения

④ Проверяем изменения

⑤ Фиксируем изменения в тематической ветку

⑥ Отправляем новую ветку в нашу копию на GitHub

Теперь, если мы зайдём на страничку нашей копии на GitHub, мы увидим, что GitHub заметил наши изменения и предлагает открыть запрос на слияние с помощью большой зелёной кнопки.

Также можно зайти на страницу «Branches», по адресу `https://github.com/<user>/<project>/branches`, найти интересующую ветку и открыть запрос оттуда.

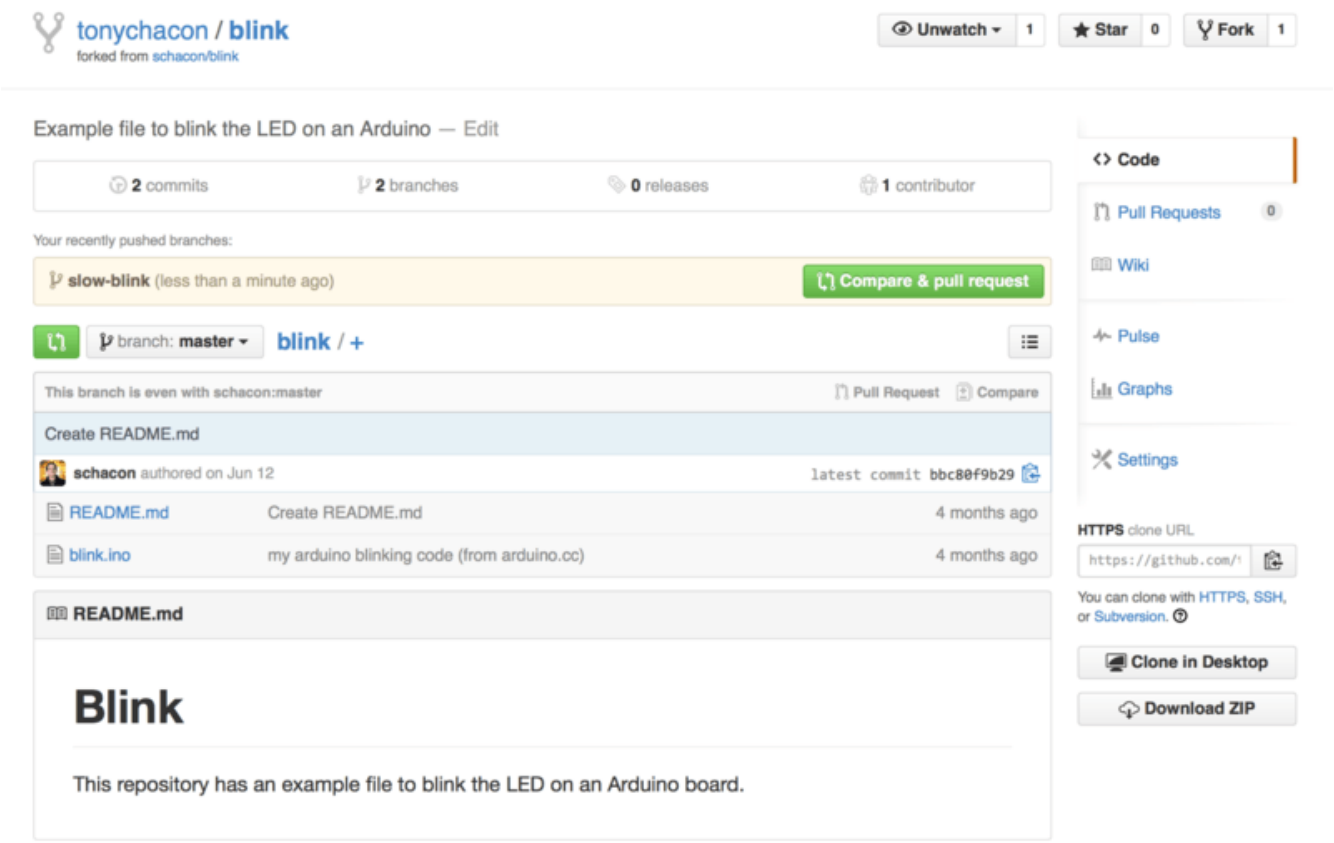


Рисунок 90. Кнопка открытия запроса на слияние

Если нажать на эту кнопку, появится экран ввода заголовка и описания предлагаемых изменений на рассмотрение владельцу проекта. Рекомендуется серьёзно подойти к составлению описания и сделать его максимально информативным, чтобы владелец проекта понимал, зачем эти изменения и какую пользу они принесут.

Также мы видим список коммитов в нашей тематической ветке, «опередивших» ветку `master` (в данном случае всего один коммит) и предпросмотр всех изменений, вносимых этими коммитами.

tonychacon / blink
forked from schacon/blink

Unwatch 1 | Star 0 | Fork 1

schacon:master ... tonychacon:slow-blink | Edit

Three seconds is better

Write | Preview | Parsed as Markdown | Edit in fullscreen

Studies have shown that 3 seconds is a far better LED delay than 1 second.

http://studies.example.com/optimal-led-delays.html

Attach images by dragging & dropping or selecting them.

✔ Able to merge.
These branches can be automatically merged.

Create pull request

1 commit | 1 file changed | 0 commit comments | 1 contributor

Commits on Oct 01, 2014

tonychacon three seconds is better db44c53

Showing 1 changed file with 2 additions and 2 deletions. | Unified | Split

4 blink.ino | View

```
@@ -18,7 +18,7 @@ void setup() {
18 18  // the loop routine runs over and over again forever:
19 19  void loop() {
20 20    digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
21    -  delay(1000);               // wait for a second
   21 +  delay(3000);               // wait for a second
22 22    digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
23    -  delay(1000);               // wait for a second
   23 +  delay(3000);               // wait for a second
24 24  }
```

Рисунок 91. Страница создания запроса на слияние

После создания запроса на слияние (путём нажатия кнопки «Create pull request» на этой странице) владелец форкнутого проекта получит уведомление о предложенных изменениях со ссылкой на страницу с информацией о запросе.

Запросы на слияние широко используются для публичных проектов типа описанного выше, когда участник уже подготовил все изменения для слияния с основным репозиторием. Тем не менее, часто можно встретить использование запросов на слияние во внутренних проектах *в самом начале* цикла разработки. Объяснение простое: вы можете обновлять тематическую ветку **после** открытия запроса на слияние, поэтому сам запрос открывается как можно раньше чтобы отслеживать прогресс разработки.

Обработка запроса на слияние

На этом этапе, владелец проекта может просмотреть предложенные изменения, принять, отклонить или прокомментировать их. Предположим, ему импонирует идея, но он

предпочёл бы большую задержку перед включением или выключением света.

В то время как в главе Распределённый Git обсуждение изменений может производится через электронную почту, на GitHub всё происходит онлайн. Владелец проекта может просмотреть суммарные изменения, вносимые запросом, и прокомментировать любую отдельно взятую строку.

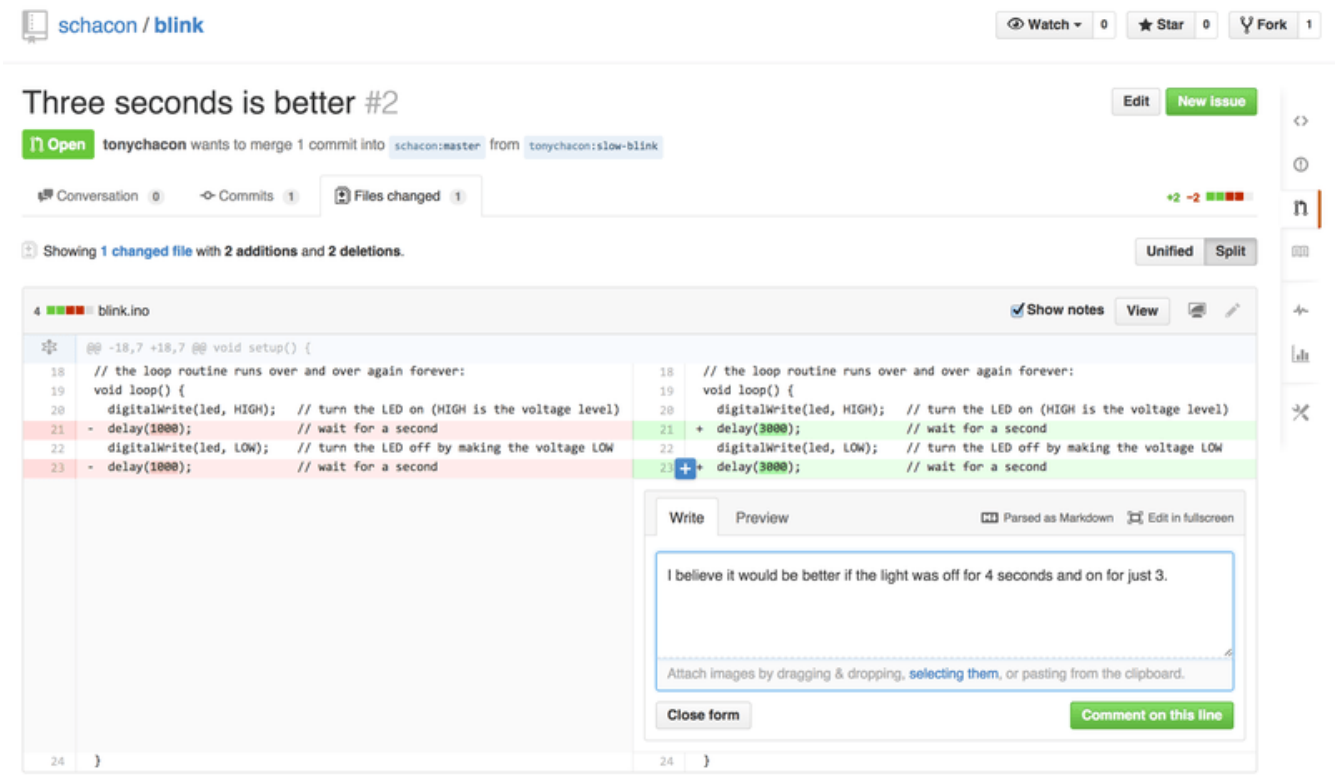


Рисунок 92. Комментирование определённой строки в запросе на слияние

Как только владелец прокомментирует изменения, автор запроса на слияние (а также все подписавшиеся на этот репозиторий) получат уведомления. Далее мы рассмотрим как настроить уведомления, но сейчас, если Тони включил уведомления через электронную почту, он получит следующее письмо:

Рисунок 93. Комментарии, отправленные по электронной почте

Кто угодно может оставлять комментарии к запросу на слияние. На Страница обсуждения запроса на слияние можно увидеть пример, где владелец проекта оставил комментарии как к строке кода, так и в основной секции обсуждения. Как вы могли заметить, комментарии к

коду так же приведены в виде обсуждения.

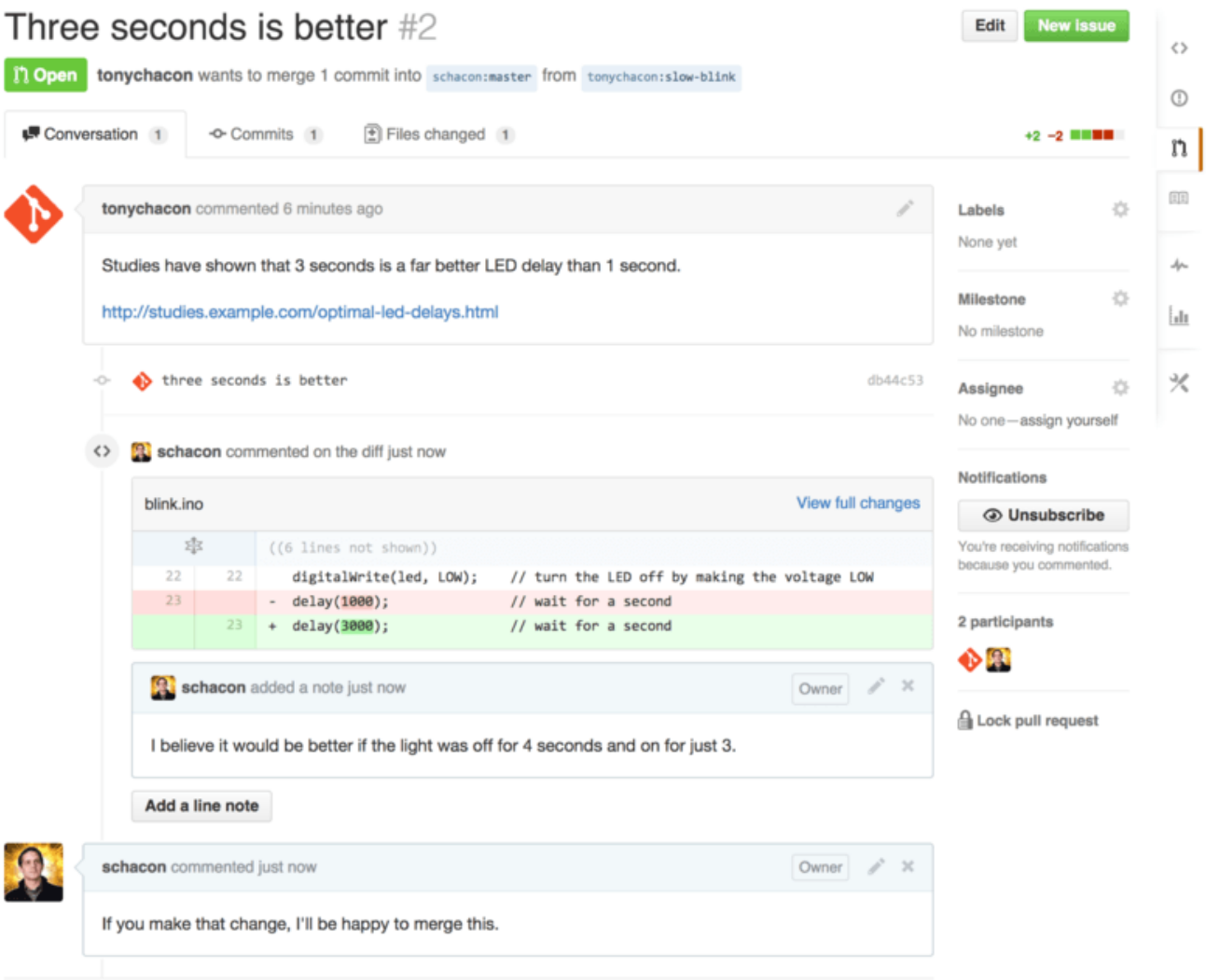


Рисунок 94. Страница обсуждения запроса на слияние

Теперь участник может видеть что ему необходимо сделать для того, чтобы его изменения были приняты. К счастью, это тоже легко сделать. Используя почту, вам потребуется заново отправить свои изменения в список рассылки, а при использовании GitHub вы просто делаете коммит в тематическую ветку и повторяете push, что автоматически обновляет запрос на слияние. На рисунке Финальная стадия запроса на слияние также видно, что в обновлённом запросе на слияние старый комментарий к коду был свёрнут, так как он относится к строке, которая с тех пор изменилась.

Когда участник сделает это, владелец проекта снова получит уведомление, а на странице запроса будет отмечено, что проблема решена. Фактически, как только строка кода имеющая комментарий будет изменена, GitHub заметит это и удалит устаревшее отличие.

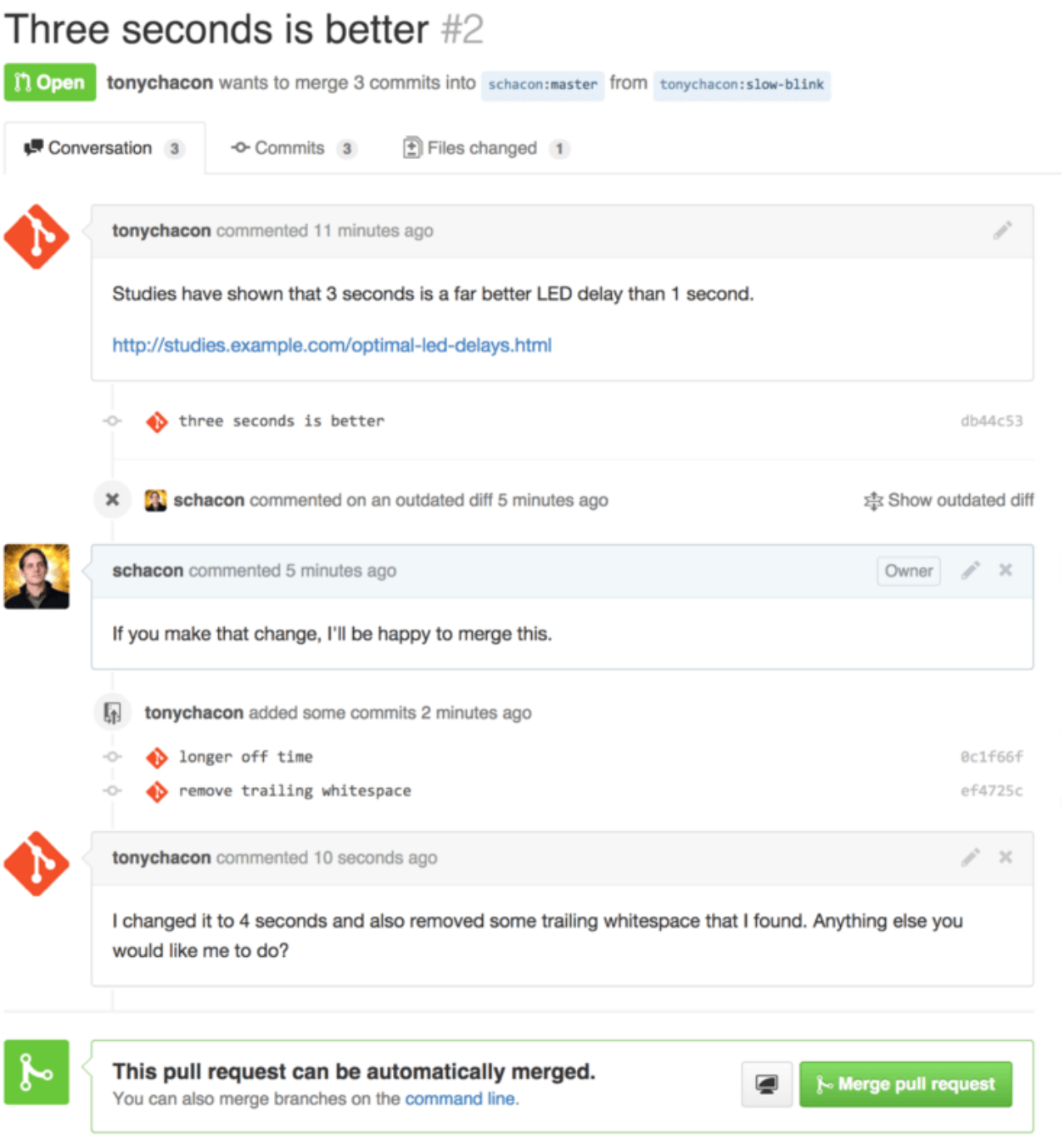


Рисунок 95. Финальная стадия запроса на слияние

Примечательно, что если вы перейдёте на вкладку «Files Changed» в этом запросе на слияние, то увидите «унифицированную» разницу — это суммарные изменения, которые будут включены в основную ветку при слиянии тематической ветки. В терминологии `git diff` это эквивалентно команде `git diff master…<branch>` для ветки, на которой основан этот запрос на слияние. В Определение применяемых изменений детальнее описан данный тип отличий.

GitHub так же проверяет может ли запрос на слияние быть применён без конфликтов и предоставляет кнопку для осуществления слияния на сервере. Эта кнопка отображается только если у вас есть права на запись в репозиторий и возможно простейшее слияние. По нажатию на неё GitHub произведёт «non-fast-forward» слияние, что значит даже если слияние **может** быть осуществлено перемоткой вперёд, всё равно будет создан коммит слияния.

При желании, можно стянуть ветку и произвести слияние локально. Если эта ветка будет слита в `master` ветку и отправлена на сервер, то GitHub автоматически закроет запрос на слияние.

Это основной рабочий процесс, который используется большинством проектов на GitHub. Создаются тематические ветки, открываются запросы на слияние, производится обсуждение, при необходимости производятся доработки в ветке и, наконец, запрос либо закрывается, либо сливается.

> *Не только ответвления*
>
> Важно отметить, что можно открывать запросы на слияние между двумя ветками в одном репозитории. Если вы работаете над функционалом с кем-то ещё и у вас обоих есть права записи, то вы можете отправить свою тематическую ветку в репозиторий и открыть запрос на слияние в `master` ветку в рамках одного проекта, что позволит инициировать процедуру проверки кода и его обсуждения. Создание ответвлений проекта не является обязательным.

Продвинутые запросы на слияние

На текущий момент мы рассмотрели основы участия в проекте на GitHub, давайте рассмотрим некоторые интересные секреты и уловки касательно запросов слияния, чтобы вы могли более эффективно их использовать.

Запросы слияния как Патчи

Важно понимать, что многие проекты не воспринимают запросы слияния как очередь идеальных патчей, которые должны применяться аккуратно и по порядку, как и большинство проектов, участие в которых основывается на отправке набора патчей через списки почтовых рассылок. Большинство проектов на GitHub понимают ветки запросов на слияние как беседу относительно предлагаемого изменения, завершающуюся слиянием унифицированных изменений.

Это важное различие, так как изменение предлагается до того, как код станет считаться идеальным, что гораздо реже происходит с распространяемыми наборами патчей через списки рассылок. Обсуждение происходит на более раннем этапе и выработка правильного решения происходит за счёт усилий сообщества. Когда код предлагается через запрос на слияние и сопровождающий проекта или сообщество предлагает изменения, то не применяется набор патчей, а отправляются результирующие изменения как новый коммит в ветку, двигая обсуждение вперёд и сохраняя уже проделанную работу нетронутой.

Например, если вы вернётесь и посмотрите на Финальная стадия запроса на слияние, то увидите, что участник не делал перебазирование своего коммита и не отправлял новый запрос на слияние. Вместо этого были сделаны новые коммиты и отправлены в существующую ветку. Таким образом, если вы в будущем вернётесь к этому запросу слияния, то легко найдёте весь контекст принятого решения. По нажатию кнопки «Merge» целенаправленно создаётся коммит слияния, который указывает на запрос слияния, оставляя возможность возврата к цепочке обсуждения.

Следование за исходным репозиторием

Если ваш запрос на слияние устарел или не может быть слит без конфликтов, то вам нужно изменить его, чтобы сопровождающий мог просто его слить. GitHub проверит это за вас и под каждым из запросов на слияние отобразит уведомление, можно ли его слить без конфликтов или нет.

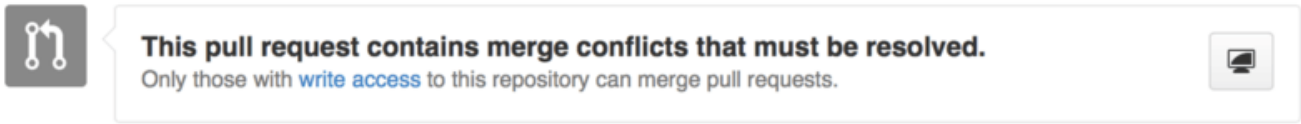


Рисунок 96. Запрос имеет конфликты слияния

Если вы видите что-то вроде Запрос имеет конфликты слияния, то вам следует изменить свою ветку так, чтобы исключить конфликты и сопровождающий не делал лишнюю работу.

Существует два основных варианта это сделать. Вы можете либо перебазировать свою ветку относительно целевой ветки (обычно, относительно `master` ветки исходного репозитория), либо слить целевую ветку в свою.

Большинство разработчиков на GitHub выбирают последний вариант по тем же причинам, что и мы в предыдущем разделе. Важна история и окончательное слияние, а перебазирование не принесёт вам ничего, кроме немного более чистой истории, при этом оно **гораздо** сложнее и может стать источником ошибок.

Если вы хотите сделать запрос на слияние применяемым, то следует добавить исходный репозиторий как новый удалённый, слить изменения из его основной ветки в вашу тематическую, если имеются исправить все проблемы и, наконец, отправить все изменения в ту ветку, на основании которой был открыт запрос на слияние.

Предположим, что в примере «tonychacon», который мы использовали ранее, основной автор сделал изменения, которые конфликтуют с запросом на слияние. Рассмотрим это пошагово.

```
$ 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 Добавляем исходный репозиторий как удалённый с именем `upstream`.

② Получаем последние изменения из него.

③ Сливаем основную ветку в нашу тематическую.

④ Исправляем указанный конфликт.

⑤ Отправляем изменения в ту же тематическую ветку.

Как только это будет сделано, запрос на слияние будет автоматически обновлён и перепроверен на возможность слияния.

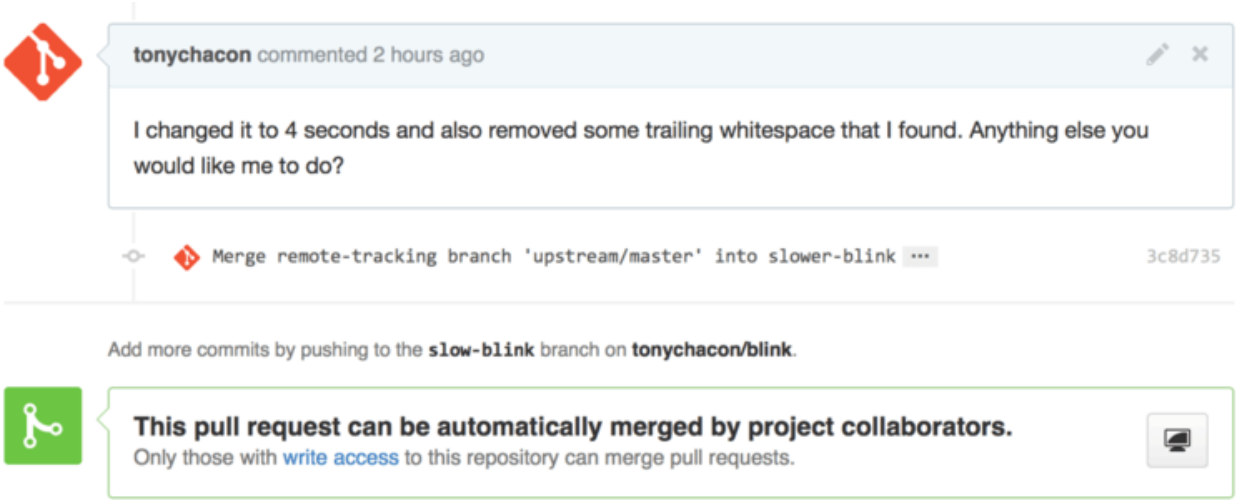


Рисунок 97. Запрос слияния без конфликтов

Одна из замечательных особенностей Git — это то, что вы можете делать это постоянно. Если у вас очень длительный проект, вы можете легко сливать изменения из целевой ветки снова и снова и иметь дело только с конфликтами, возникшими с момента вашего последнего слияния, что делает процесс очень управляемым.

Если вы очень хотите перебазировать ветку, чтобы её почистить, то, конечно, вы можете это сделать, но настоятельно не рекомендуется переписывать ветку, к которой уже открыт запрос на слияние. Если другие люди уже стянули её и проделали много работы, то вы столкнётесь со всеми проблемами, описанными в разделе Опасности перемещения главы 3. Вместо этого, отправьте перебазированную ветку в новую на GitHub и откройте новый

запрос на слияние, который указывает на предыдущий, затем закройте исходный.

Ссылки

Возможно, ваш следующий вопрос будет: «Как мне сослаться на предыдущий запрос слияния?» Оказывается, существует много способов ссылаться на другие вещи практически везде, где у вас есть права записи на GitHub.

Давайте начнём с перекрёстных ссылок для запросов слияния или проблем. Всем запросам слияния и проблемам присваиваются уникальные номера в пределах проекта. Например, у вас не может быть запроса на слияние с номером #3 и проблемы с номером #3. Если вы хотите сослаться на любой запрос слияния или проблему из другого места, просто добавьте `#<num>` в комментарий или описание. Так же можно указывать более конкретно, если проблема или запрос слияния находятся где-то ещё; пишите `username#<num>` если ссылаетесь на проблему или запрос слияния, находящиеся в ответвлённом репозитории, или `username/repo#<num>` если ссылаетесь на другой репозиторий.

Рассмотрим это на примере. Предположим, что мы перебазировали ветку в предыдущем примере, создали новый запрос слияния для неё и сейчас хотим сослаться на предыдущий запрос слияния из нового. Так же мы хотим сослаться на проблему, находящуюся в ответвлённом репозитории, и на проблему из совершенно другого проекта. Мы можем составить описание как указано на Перекрёстные ссылки в запросе слияния.

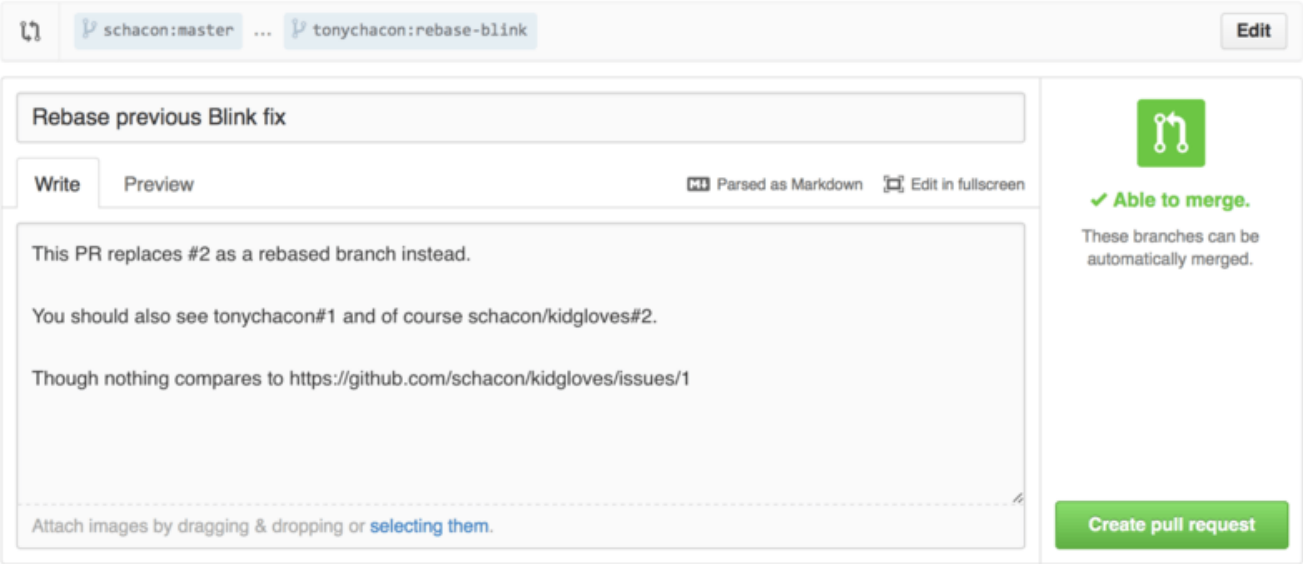


Рисунок 98. Перекрёстные ссылки в запросе слияния

Когда мы отправим запрос на слияние, то увидим что-то вроде Отображение перекрёстных ссылок в запросе слияния.

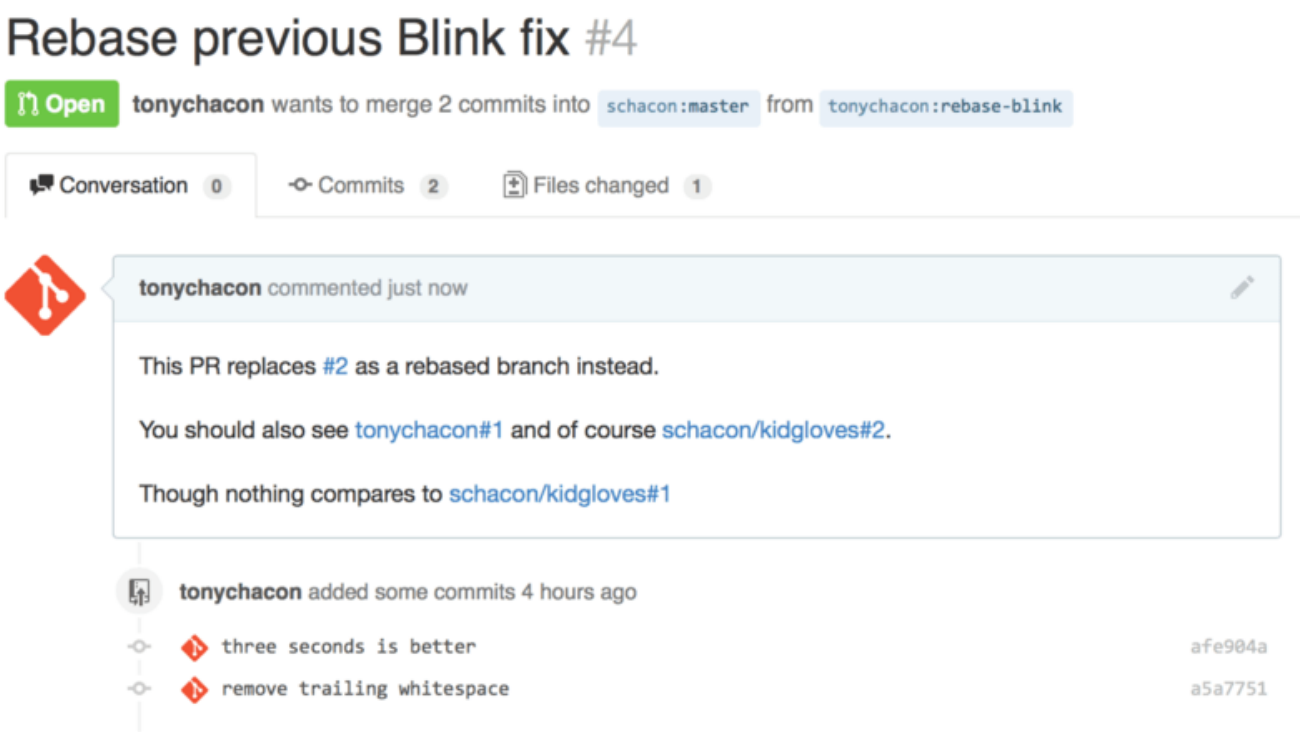


Рисунок 99. Отображение перекрёстных ссылок в запросе слияния

Заметьте, что указанная полная ссылка на GitHub была сокращена до необходимого минимума.

Если Тони сейчас вернётся назад и закроет оригинальный запрос слияния, то мы это увидим, так как он упомянут в новом, а GitHub автоматически создаст отслеживающее событие в хронике запроса слияния. Это значит, что все, кто просматривает закрытый запрос слияния, могут легко перейти к запросу слияния, который его заменил. Ссылка будет выглядеть как указано на Отображение перекрёстных ссылок в закрытом запросе слияния.

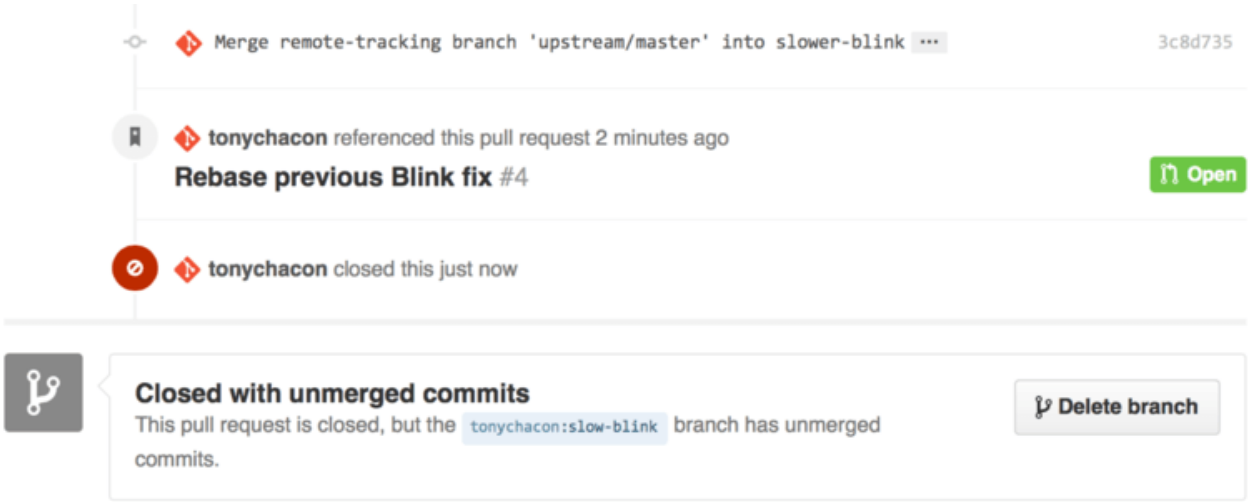


Рисунок 100. Отображение перекрёстных ссылок в закрытом запросе слияния

Кроме идентификационных номеров, можно ссылаться на конкретный коммит используя SHA-1. Следует указывать полный 40 символьный хеш SHA-1, но если GitHub увидит его в комментарии, то автоматически подставит ссылку на коммит. Как было сказано выше, вы можете ссылаться на коммиты как в других, так и в ответвлённых репозиториях точно так же, как делали это с Проблемами.

GitHub-версия разметки Markdown

Ссылки на другие Проблемы — это лишь часть интереснейших вещей, которые вы можете делать в текстовых полях на GitHub. Для «проблемы» или «запроса слияния» в полях описания, комментария, комментария кода и других вы можете использовать так называемую «GitHub-версию разметки Markdown». Разметка похожа на обычный текст, который основательно преобразуется.

Смотрите Пример написания и отображения текста с разметкой для примера как использовать разметку при написании комментариев и текста.

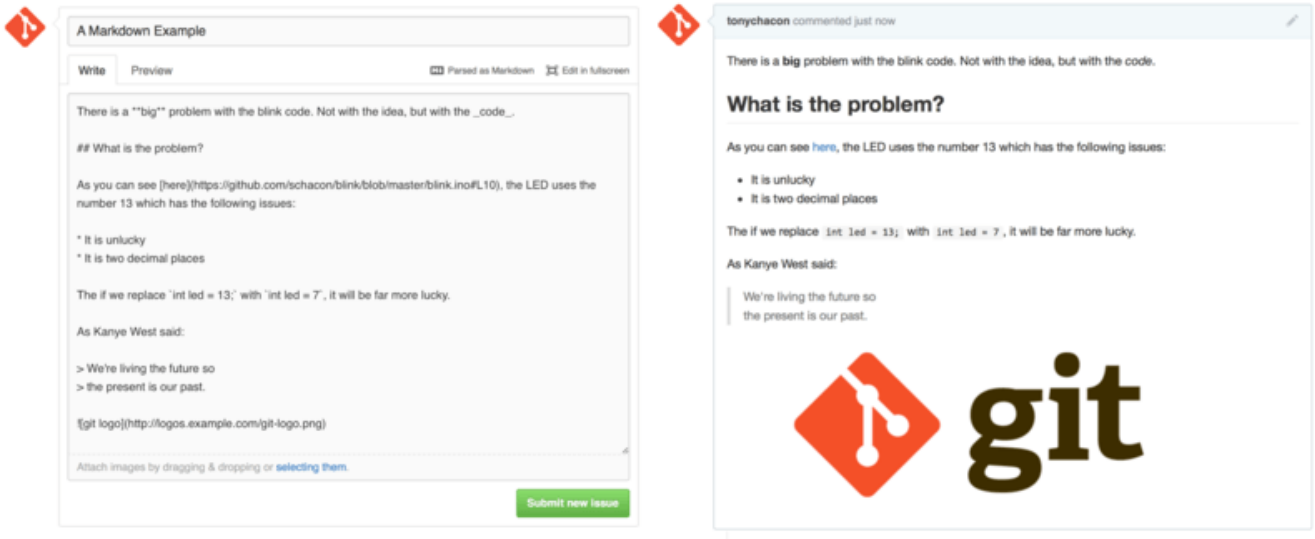


Рисунок 101. Пример написания и отображения текста с разметкой

GitHub расширил возможности обычной разметки. Эти возможности могут быть очень полезными при создании запросов слияния или комментариев и описаний к проблемам.

Списки задач

Список задач — это первая действительно важная возможность специфической разметки GitHub, особенно для запросов слияния. Список задач представляет собой список флажков для задач, которые вы хотите выполнить. Размещение его в описании Проблемы или запроса на слияние обычно указывает на то, что должно быть сделано до того, как проблема будет считаться решённой.

Список задач можно добавить следующим образом:

```
- [X] Write the code
- [ ] Write all the tests
- [ ] Document the code
```

Если добавить этот список в описание запроса на слияние или проблемы, то он будет отображён следующим образом Отображение списка задач в комментарии

Рисунок 102. Отображение списка задач в комментарии

Он часто используется в запросах на слияние для отображения списка того, что вы хотите сделать до того, как запрос будет готов к слиянию. Вы можете просто кликнуть по флажку, чтобы обновить комментарий — не нужно редактировать комментарий вручную, чтобы пометить задачу как выполненную.

Так же GitHub ищет списки задач в запросах на слияние и проблемах и отображает их как метаданные на страницах, где они упоминаются. Например, если в вашем запросе на слияние есть задачи и вы просматриваете список всех запросов, то можно увидеть насколько готов каждый из них. Это позволяет разбивать запрос на слияние на несколько подзадач и помогает другим людям отслеживать прогресс ветки. Пример приведён на Статистика задач в списке запросов слияния.

Рисунок 103. Статистика задач в списке запросов слияния

Такая возможность невероятно полезна когда вы открываете запрос на слияние на раннем этапе реализации и отслеживаете прогресс с помощью него.

Отрывки кода

В комментарии так же можно вставлять отрывки кода. Это особенно полезно когда вы хотите показать что-нибудь, что вы *собираетесь* попробовать сделать, до того, как включить это в вашу ветку. Так же часто применяется для добавления примеров кода, который не работает или мог быть добавлен в запрос на слияние.

Для добавления отрывка кода следует обрамить его обратными кавычками.

```
```java
for(int i=0 ; i < 5 ; i++)
{
   System.out.println("i is : " + i);
}
```
```

```
```

Если вы укажете название языка, как показано на примере, GitHub попробует применить к нему подсветку синтаксиса. Для приведённого примера код будет выглядеть как на Отображение обрамлённого кода.

Рисунок 104. Отображение обрамлённого кода

Цитирование

Если вы отвечаете только на часть большого комментария, то можно цитировать только выбранную часть, предваряя её символом `>`. Это настолько часто используется, что даже существует комбинация клавиш для этого. Если в комментарии выделить текст, на который вы собираетесь ответить, и нажать клавишу `r`, то выделенный текст будет включён как цитата в ваш комментарий.

Цитаты выглядят примерно так:

```
> 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?
```

После обработки комментарий будет выглядеть как Пример отображения цитаты.

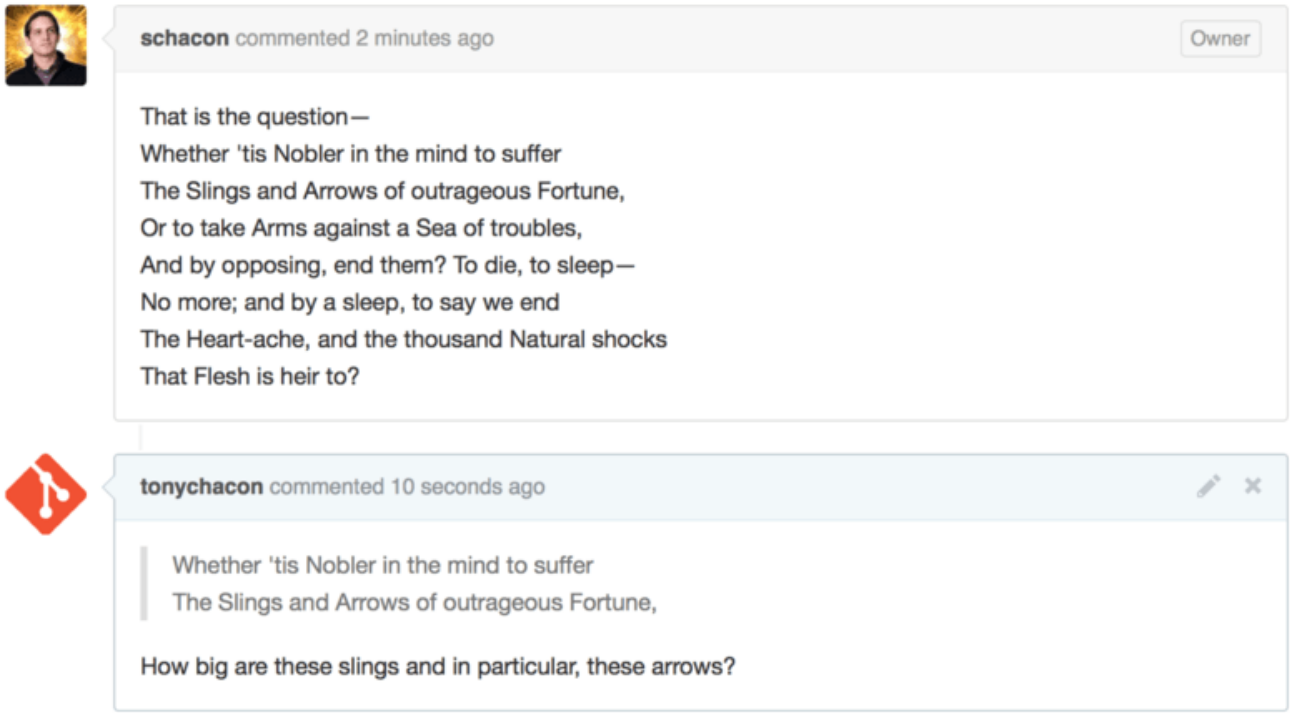


Рисунок 105. Пример отображения цитаты

Смайлики

Наконец, вы можете использовать смайлики. На GitHub вы можете часто встретить их в комментариях или запросах на слияние. Для них есть даже помощник. Например, если при наборе комментария ввести символ двоеточия `:`, то будут предложены варианты автодополнения.

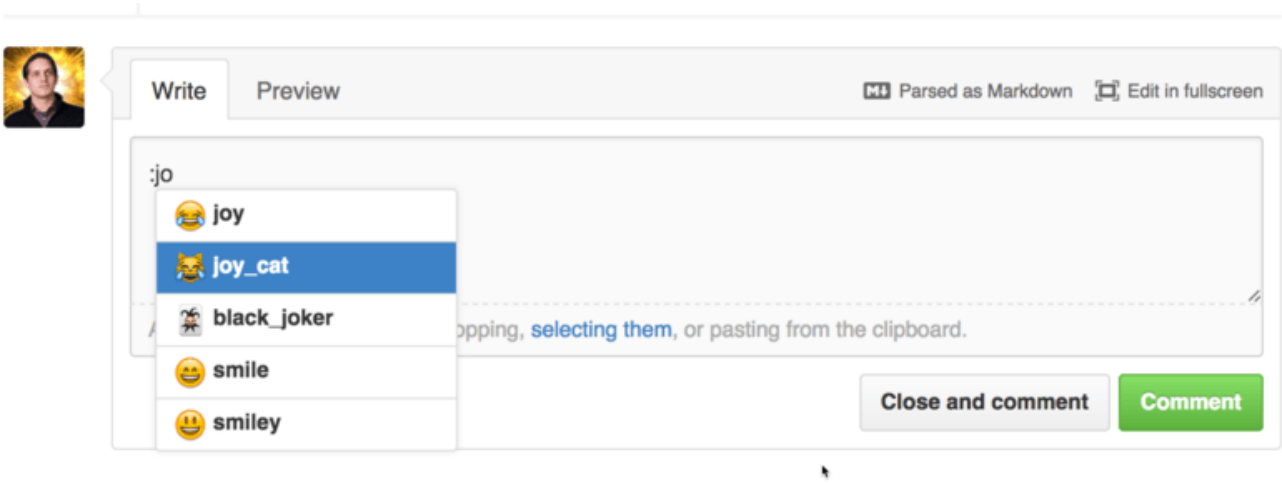


Рисунок 106. Автодополнение для смайлов в действии

Смайлы имеют вид `:<name>:` и могут располагаться в любом месте комментария. Например, вы можете написать что-нибудь вроде этого:

```
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:
```

Такой комментарий будет выглядеть как на Перегруженный смайликами комментарий.

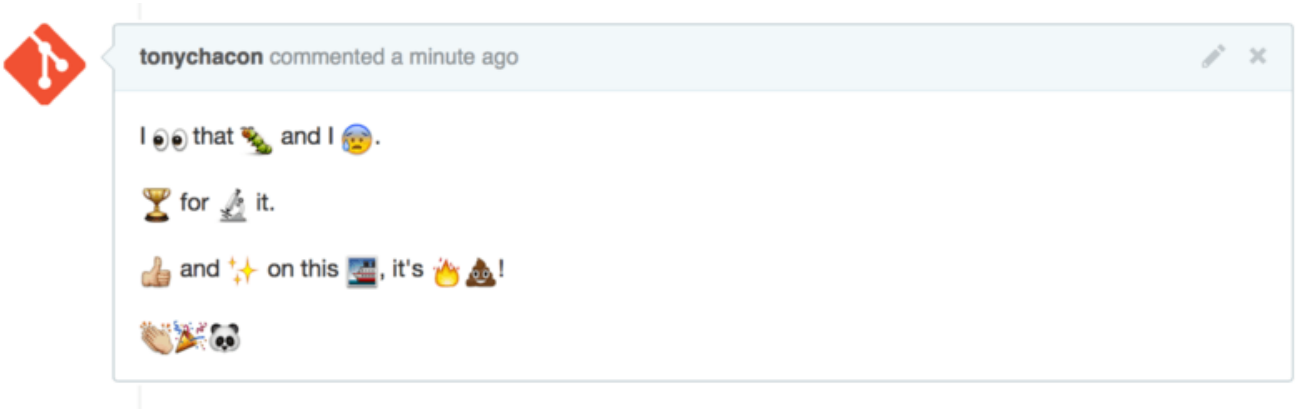


Рисунок 107. Перегруженный смайликами комментарий

Не то чтобы это невероятно полезно, но добавляет немного веселья и эмоций там, где трудно выразить какие-то эмоции.

На текущий момент существует много интернет сервисов, где используются смайлики. Отличную шпаргалку по поиску смайликов, которые выражают нужную вам эмоцию, можно найти здесь:

https://www.webfx.com/tools/emoji-cheat-sheet/

Картинки

Технически, картинки не относятся к разметке GitHub, но их использование очень полезно. В дополнение к ссылкам на картинки в комментариях, GitHub позволяет встраивать картинки в комментарии.

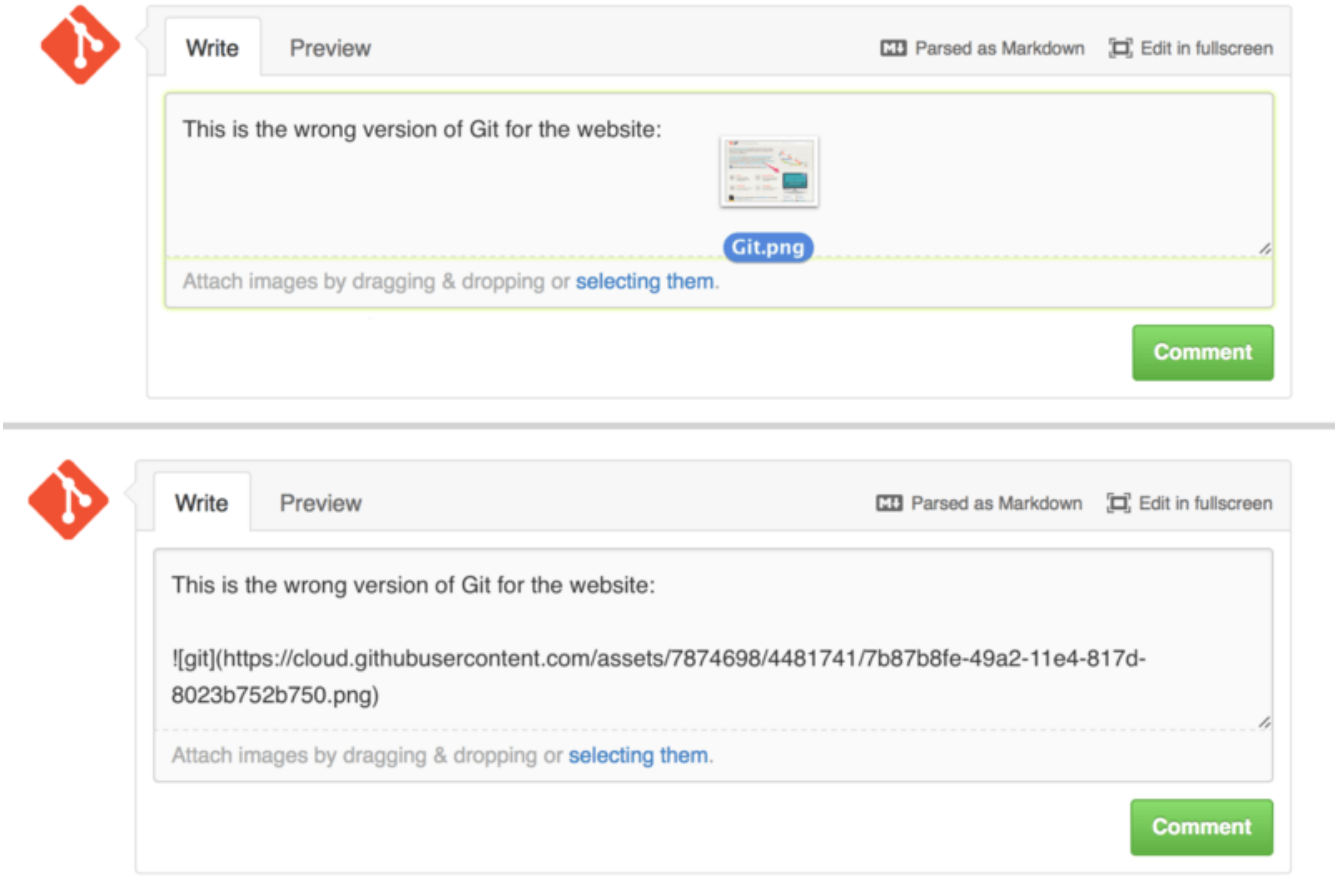


Рисунок 108. Перетаскивание картинки для загрузки и встраивания

Если вернуться немного назад к Перекрёстные ссылки в запросе слияния, то над областью редактирования вы увидите небольшую подсказку «Parsed as Markdown». Нажав не неё, вы получите полную подсказку по использованию GitHub разметки.

Поддержание GitHub репозитория в актуальном состоянии

После создания форка, ваш репозиторий будет существовать независимо от оригинального репозитория. В частности, при появлении в оригинальном репозитории новых коммитов GitHub информирует вас следующим сообщением:

```
This branch is 5 commits behind progit:master.
```

При этом GitHub никогда не обновляет ваш репозиторий — это вы должны делать сами. К счастью, это очень просто сделать.

Первый способ не требует конфигурации. Например, если вы сделали форк репозитория `https://github.com/progit/progit2.git`, то актуализировать ветку `master` можно следующим образом:

```
$ git checkout master ①
$ git pull https://github.com/progit/progit2.git ②
$ git push origin master ③
```

① Если вы находитесь на другой ветке — перейти на ветку `master`.

② Получить изменения из репозитория `https://github.com/progit/progit2.git` и слить их с веткой `master`.

③ Отправить локальную ветку `master` в ваш форк `origin`.

Каждый раз писать URL репозитория для получения изменений достаточно утомительно. Этот процесс можно автоматизировать слегка изменив настройки:

```
$ git remote add progit https://github.com/progit/progit2.git ①
$ git fetch progit ②
$ git branch --set-upstream-to=progit/master master ③
$ git config --local remote.pushDefault origin ④
```

① Добавить исходный репозиторий как удалённый и назвать его `progit`.

② Получить ветки репозитория `progit`, в частности ветку `master`.

③ Настроить локальную ветку `master` на получение изменений из репозитория `progit`.

④ Установить `origin` как репозиторий по умолчанию для отправки.

После этого, процесс обновления становится гораздо проще:

```
$ git checkout master ①
$ git pull ②
$ git push ③
```

① Если вы находитесь на другой ветке — перейти на ветку `master`.

② Получить изменения из репозитория `progit` и слить их с веткой `master`.

③ Отправить локальную ветку `master` в ваш форк `origin`.

Данный подход не лишён недостатков. Git будет молча выполнять указанные действия и не предупредит вас в случае, когда вы добавили коммит в `master`, получили изменения из `progit` и отправили всё вместе в `origin` — все эти операции абсолютно корректны. Поэтому вам стоит исключить прямое добавление коммитов в ветку `master`, поскольку эта ветка фактически принадлежит другому репозиторию.

Сопровождение проекта

Теперь, когда вы комфортно себя чувствуете при участии в проекте, давайте посмотрим на другую сторону вопроса: создание, сопровождение и администрирование вашего собственного проекта.

Создание нового репозитория

Давайте создадим новый репозиторий для распространения кода нашего проекта. В панели управления справа нажмите кнопку «New repository» или воспользуйтесь кнопкой `+` на панели инструментов, рядом с вашим именем пользователя как показано на рисунке

Выпадающее меню «New repository».

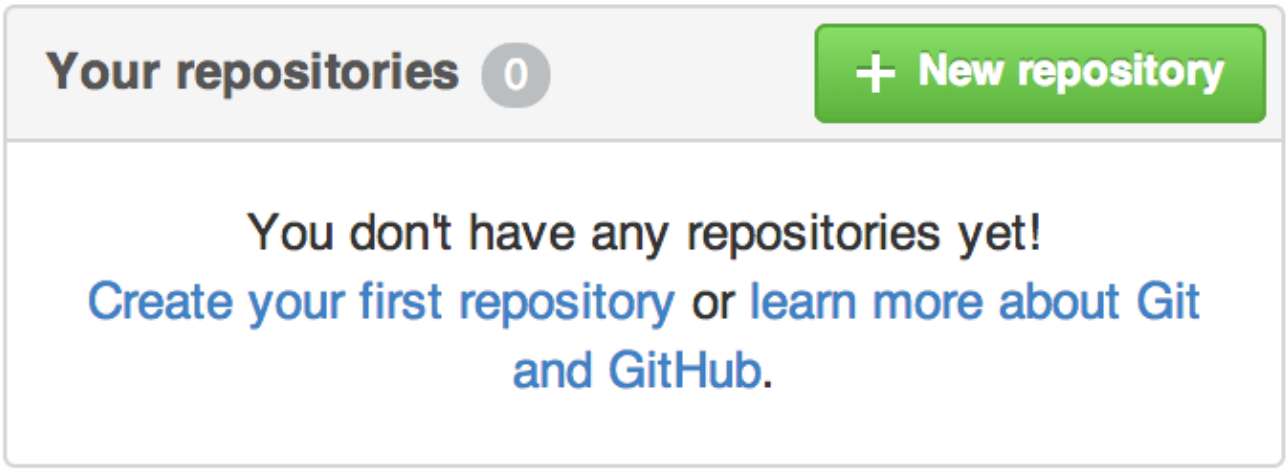


Рисунок 109. Раздел «Your repositories»

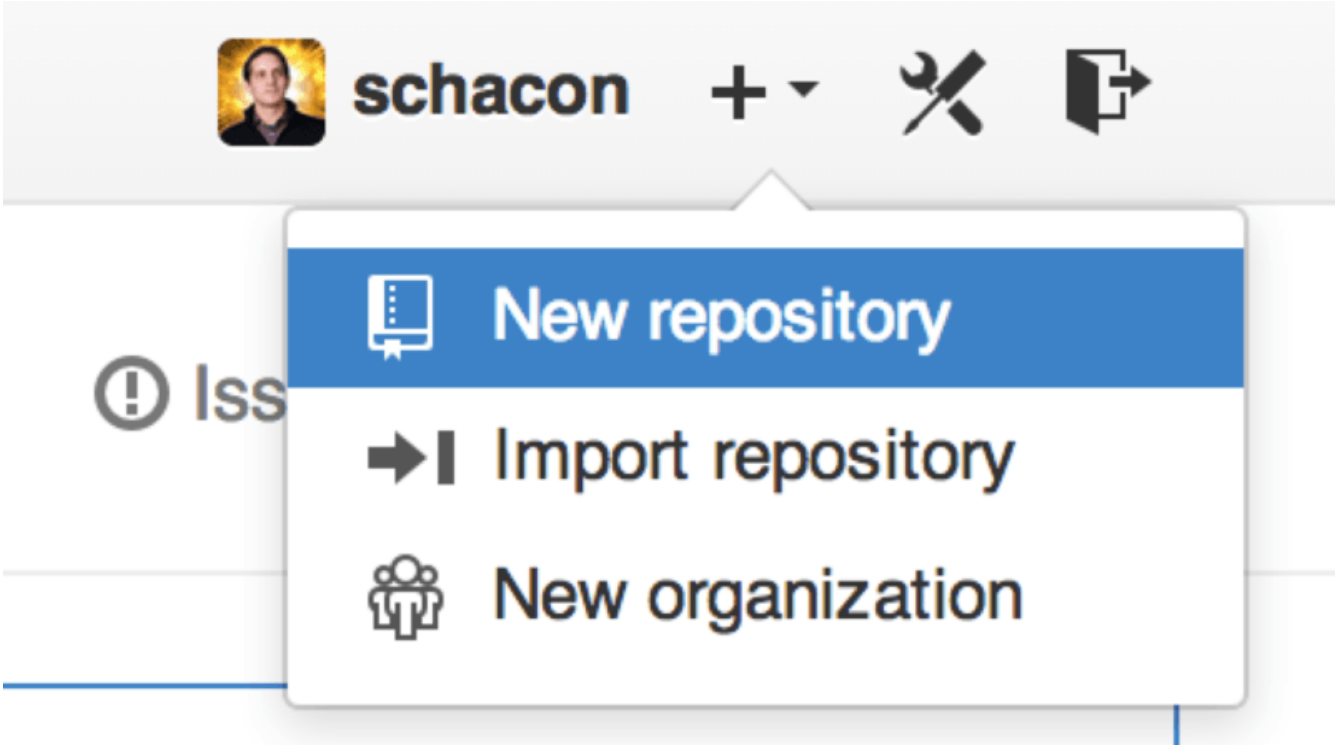


Рисунок 110. Выпадающее меню «New repository»

Это приведёт к открытию формы «New repository»:

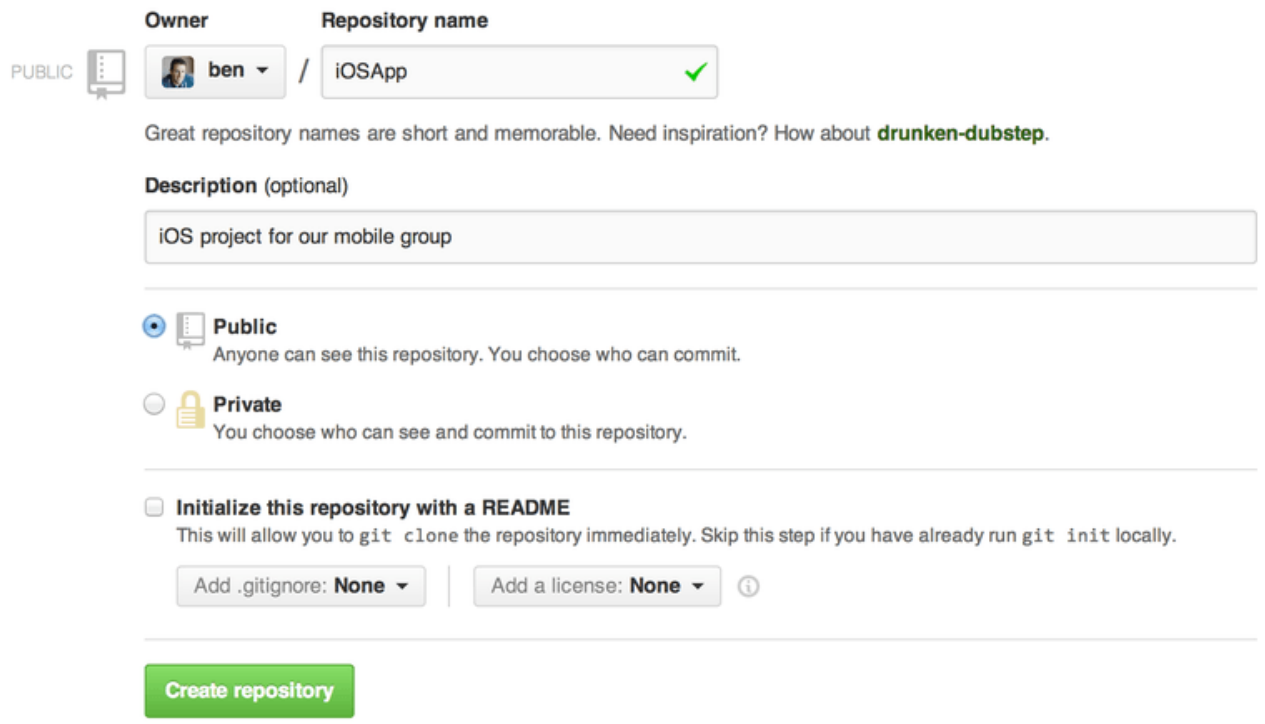


Рисунок 111. Форма «new repository»

Всё, что в действительности нужно сделать, так это указать название проекта, все остальные поля опциональны. Сейчас, просто нажмите кнопку «Create Repository» и ваш новый репозиторий с названием `<пользователь>/<имя_проекта>` готов.

Так как в репозитории ещё нет кода, GitHub отобразит инструкции о том, как создать совершенно новый репозиторий или подключить существующий Git проект. Здесь мы не будем этого делать; если вам нужно освежить память, смотрите главу Основы Git.

Теперь ваш проект хостится на GitHub и вы можете предоставить ссылку на него любому желающему. Все проекты на GitHub доступны как по HTTP `https://github.com/<user>/<project_name>`, так по SSH `git@github.com:<user>/<project_name>`. Git может получать и отправлять изменения по обоим указанным ссылкам, при этом производится контроль доступа на основании учётных данных пользователя, осуществляющего подключение.

> Обычно, для общедоступного проекта предпочтительнее использовать HTTPS ссылки, так как это не требует наличия GitHub аккаунта для клонирования репозитория. При этом, для использования SSH ссылки у пользователя должен быть GitHub аккаунт и его SSH ключ должен быть добавлен в ваш проект. Так же HTTPS ссылка полностью совпадает с URL адресом, который пользователи могут вставить в браузер для просмотра вашего репозитория.

Добавление участников

Если вы работаете с другими людьми, которым вы хотите предоставить доступ для отправки коммитов, то вам следует добавить их как «участников». Если Бен, Джефф и Льюис зарегистрировались на GitHub и вы хотите разрешить им делать «push» в ваш

репозиторий, то добавьте их в свой проект. Это предоставит им «push» доступ; это означает, что они будут иметь права доступа как на чтение, так и на запись в проект и Git репозиторий.

Перейдите по ссылке «Settings» в нижней части панели справа.

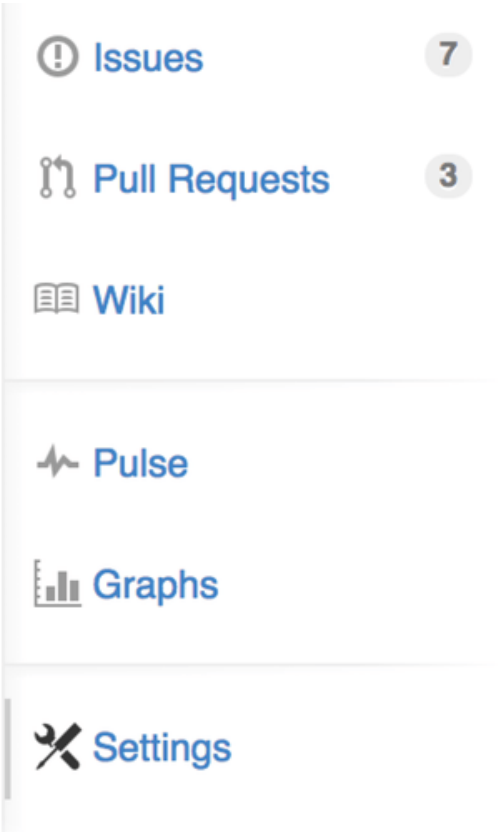


Рисунок 112. Ссылка на настройки репозитория

Затем выберите «Collaborators» в меню слева. Напишите имя пользователя в поле для ввода и нажмите кнопку «Add collaborator». Так вы можете добавить неограниченное количество пользователей. Чтобы отозвать доступ, просто нажмите «X» справа от имени пользователя.

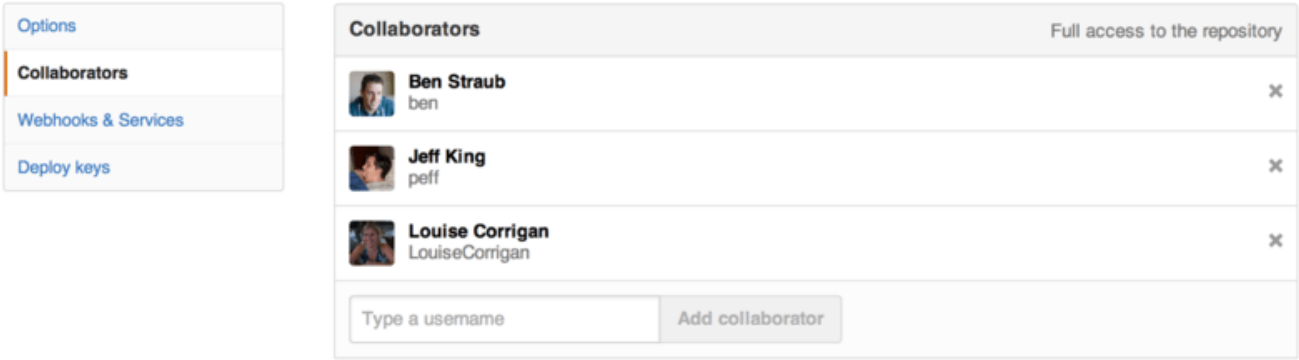


Рисунок 113. Участники проекта

Управление запросами на слияние

Сейчас у вас есть проект с некоторым кодом и, возможно, несколько участников с «push» доступом, давайте рассмотрим ситуацию, когда вы получаете запрос на слияние.

Запрос на слияние может быть как из ветки вашего репозитория, так и из ветки форка вашего проекта. Отличаются они тем, что вы не можете отправлять изменения в ветки ответвлённого проекта, а его владельцы не могут отправлять в ваши, при этом для внутренних запросов на слияние характерно наличие доступа к ветке у обоих пользователей.

Для последующих примеров предположим, что вы «tonychacon» и создали новый проект для Arduino с названием «fade».

Email уведомления

Кто-то вносит изменения в ваш код и отправляет вам запрос на слияние. Вы должны получить письмо с уведомлением о новом запросе слияния, которое выглядит как на Email уведомление о новом запросе слияния.

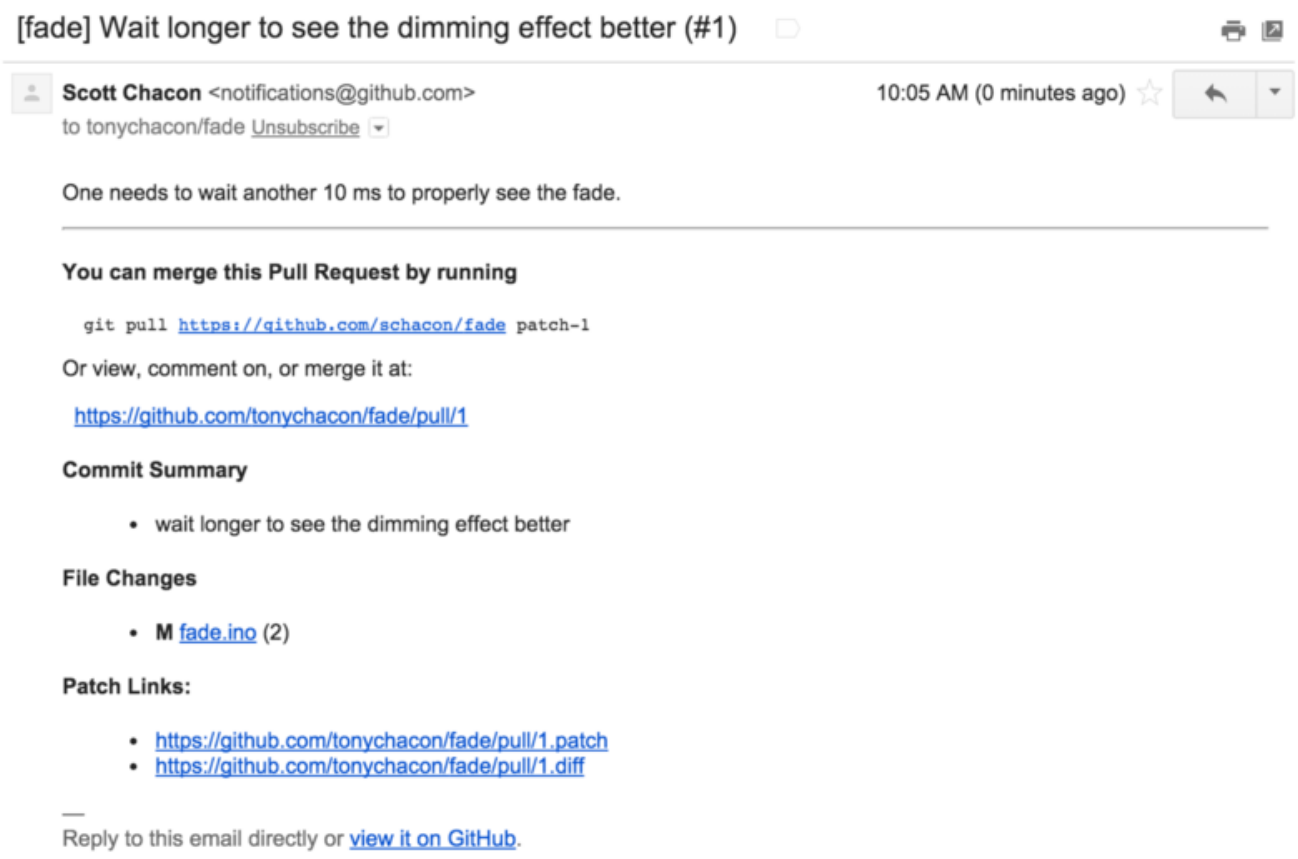


Рисунок 114. Email уведомление о новом запросе слияния

Следует сказать о некоторых особенностях этого уведомления. В нём содержится краткая статистика отличий — количество изменений и список файлов, которые были изменены в этом запросе слияния, ссылка на страницу запроса слияния на GitHub, а так же несколько ссылок, которые вы можете использовать в командной строке.

Если вы видите строку с текстом `git pull <url> patch-1`, то это самый простой способ слить удалённую ветку без добавления удалённого репозитория. Это кратко описывалось в Извлечение удалённых веток. Если хотите, то можно сначала переключиться в тематическую ветку и только потом выполнить эту команду для изменений запроса слияния.

Другие ссылки, которые представляют интерес, это `.diff` и `.patch` ссылки. Как вы догадались, они указывают на версии унифицированной разницы и патча запроса слияния. Технически, вы можете слить изменения из запроса слияния командой:

```
$ curl https://github.com/tonychacon/fade/pull/1.patch | git am
```

Взаимодействие по запросу слияния

Как описано в разделе Рабочий процесс с использованием GitHub главы 6, вы можете общаться с тем, кто открыл запрос на слияние. Вы можете добавлять комментарии к отдельным строкам кода, коммитам или ко всему запросу целиком, используя усовершенствованную разметку GitHub где угодно.

Каждый раз, когда кто-то другой оставляет комментарий к запросу слияния, вы будете получать email уведомления по каждому событию. Каждое уведомление будет содержать ссылку на страницу запроса слияния где была зафиксирована активность и, чтобы оставить комментарий в основной ветке запроса на слияние, вы можете просто ответить на это письмо.

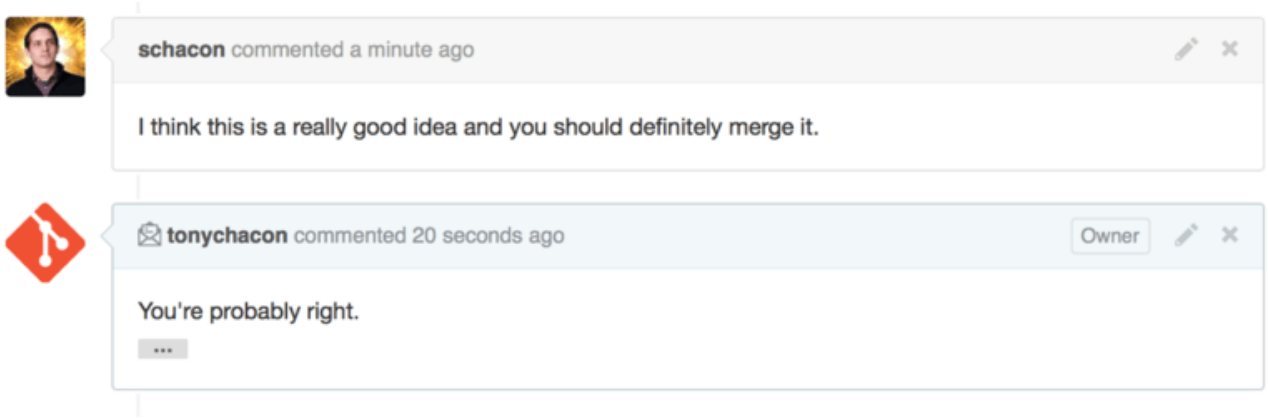


Рисунок 115. Ответы на письма включены в диалог

Когда вы готовы слить код, вы можете стянуть его себе и слить локально, слить используя команду `git pull <url> <branch>`, которую мы видели ранее, или добавив ответвлённый репозиторий как удалённый, получить и слить изменения.

Если слияние тривиально, то можно просто нажать кнопку «Merge» на сайте GitHub. Это всегда приводит с созданию коммита слияния, даже если доступно слияние перемоткой вперёд. Это значит, что в любом случае создаётся коммит слияния, как только вы нажимаете кнопку «Merge». Как можно увидеть на Кнопка Merge и инструкции по ручному слиянию запроса, GitHub отображает информацию об этом при вызове подсказки.

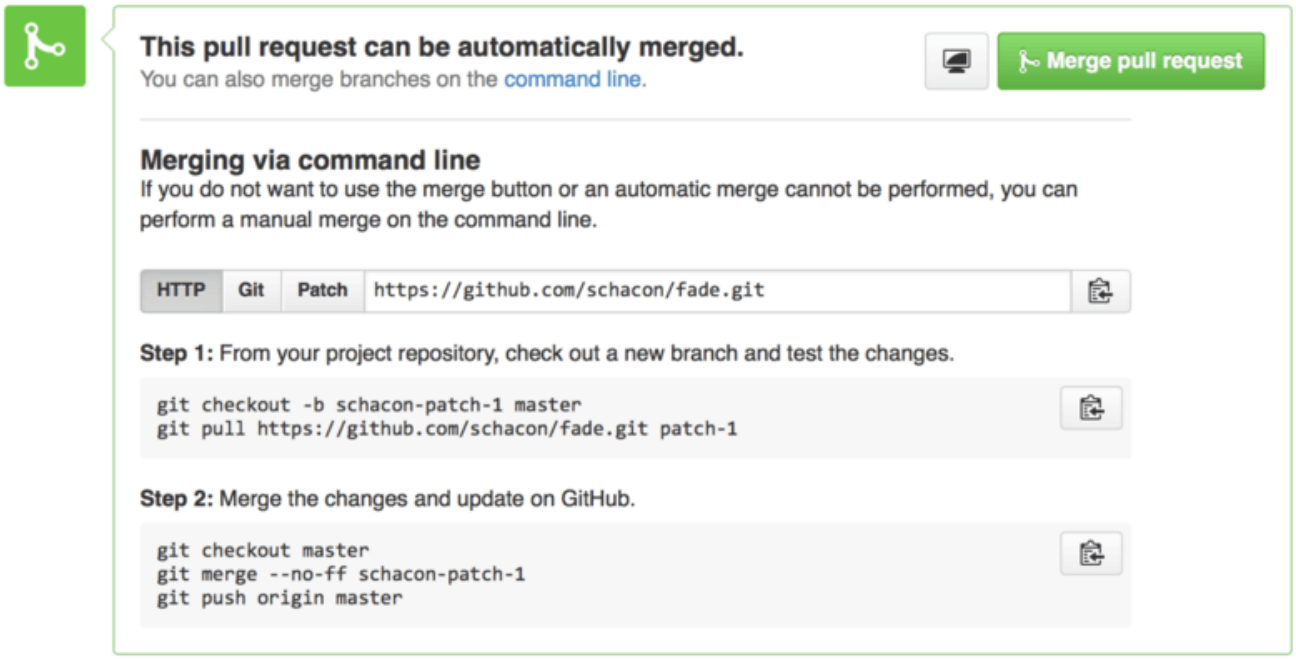


Рисунок 116. Кнопка Merge и инструкции по ручному слиянию запроса

Если вы решаете не сливать запрос, то вы можете просто закрыть запрос на слияние, а открывший его участник будет уведомлён.

Ссылки на запрос слияния

Если у вас **много** запросов слияния и вы не хотите добавлять пачку удалённых репозиториев или постоянно делать однократный «pull», то у GitHub есть хитрый трюк, позволяющий это делать. Этот трюк очень сложный, но полезный и мы рассмотрим его немного позже в Спецификации ссылок.

Фактически, GitHub представляет ветки запросов слияния как псевдоветки на сервере. По умолчанию, они не копируются при клонировании, а существуют в замаскированном виде и вы можете легко получить доступ к ним.

В качестве примера мы используем низкоуровневую команду `ls-remote` (часто упоминается как «plumbing» команда, более подробно о ней будет рассказано в Сантехника и Фарфор). Обычно, эта команда не используется в повседневных Git операциях, но сейчас поможет нам увидеть, какие ссылки присутствуют на сервере.

Если выполнить её относительно использованного ранее репозитория «blink», мы получим список всех веток, тегов и прочих ссылок в репозитории.

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d    HEAD
10d539600d86723087810ec636870a504f4fee4d    refs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6e    refs/pull/1/head
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3    refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfbc2665adec1    refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d    refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a    refs/pull/4/head
```

```
31a45fc257e8433c8d8804e3e848cf61c9d3166c    refs/pull/4/merge
```

Аналогично, если вы, находясь в своём репозитории, выполните команду `git ls-remote origin` или укажете любой другой удалённый репозиторий, то результат будет схожим.

Если репозиторий находится на GitHub и существуют открытые запросы слияния, то эти ссылки будут отображены с префиксами `refs/pull/`. По сути это ветки, но так как они находятся не в `refs/heads/`, то они не копируются при клонировании или получении изменений с сервера — процесс получения изменений игнорирует их по умолчанию.

Для каждого запроса слияния существует две ссылки, одна из которых записана в `/head` и указывает на последний коммит в ветке запроса на слияние. Таким образом, если кто-то открывает запрос на слияние в наш репозиторий из своей ветки `bug-fix`, которая указывает на коммит `a5a775`, то в **нашем** репозитории не будет ветки `bug-fix` (так как она находится в форке), при этом у нас *появится* `pull/<pr#>/head`, которая указывает на `a5a775`. Это означает, что мы можем стянуть все ветки запросов слияния одной командой не добавляя набор удалённых репозиториев.

Теперь можно получить ссылки напрямую.

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch            refs/pull/958/head -> FETCH_HEAD
```

Эта команда указывает Git: «Подключись к `origin` репозиторию и скачай ссылку `refs/pull/958/head`». Git с радостью слушается и выкачивает всё необходимое для построения указанной ссылки, а так же устанавливает указатель на коммит в `.git/FETCH_HEAD`. Далее, вы можете слить изменения в нужную ветку при помощи команды `git merge FETCH_HEAD`, однако, сообщение коммита слияния будет выглядеть немного странно. Так же это становится утомительным, если вы просматриваете **много** запросов на слияние.

Существует способ получать *все* запросы слияния и поддерживать их в актуальном состоянии при подключении к удалённому репозиторию. Откройте файл `.git/config` в текстовом редакторе и обратите внимание на секцию удалённого репозитория `origin`. Она должна выглядеть как-то так:

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2
    fetch = +refs/heads/*:refs/remotes/origin/*
```

Строка, начинающаяся с `fetch =`, является спецификацией ссылок («refspec»). Это способ сопоставить названия в удалённом репозитории с названиями в локальном каталоге `.git`. Конкретно эта строка говорит Git: «все объекты удалённого репозитория из `refs/heads` должны сохраняться локально в `refs/remotes/origin`». Вы можете изменить это поведение добавив ещё одну строку спецификации:

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2.git
    fetch = +refs/heads/*:refs/remotes/origin/*
    fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

Последняя строка говорит Git: «Все ссылки, похожие на `refs/pull/123/head`, должны быть сохранены локально как `refs/remotes/origin/pr/123`». Теперь, если сохранить файл и выполнить команду `git fetch`, вы получите:

```
$ 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
# …
```

Все запросы слияния из удалённого репозитория представлены в локальном репозитории как ветки слежения; они только для чтения и обновляются каждый раз при выполнении `git fetch`. Таким образом, локальное тестирование кода запроса слияния становится очень простым:

```
$ 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'
```

Особо внимательные из вас заметили `head` в конце спецификации, относящейся к удалённому репозиторию. Так же на стороне GitHub существует ссылка `refs/pull/#/merge`, которая представляет коммит, формируемый при нажатии кнопки «merge» на сайте. Это позволяет вам протестировать слияние перед нажатием этой кнопки.

Запросы слияния на запросы слияния

Вы можете открыть запрос слияния не только в ветку `master`, запросы слияния могут указывать на любую ветку любого репозитория в сети. По сути, вы можете даже открыть запрос слияния, указывающий на другой запрос слияния.

Если вы видите толковый запрос слияния и у вас есть идея как его улучшить или вы не уверены, что это хорошая идея, или у вас просто нет прав записи в целевую ветку, то в таком случае вы можете открыть запрос слияния, указывающий на данный запрос.

При открытии запроса на слияние вверху страницы вы увидите меню для выбора целевой и исходной веток. Если нажать кнопку `Edit` справа, то станет доступным выбор не только исходной ветки, а ещё и форка.

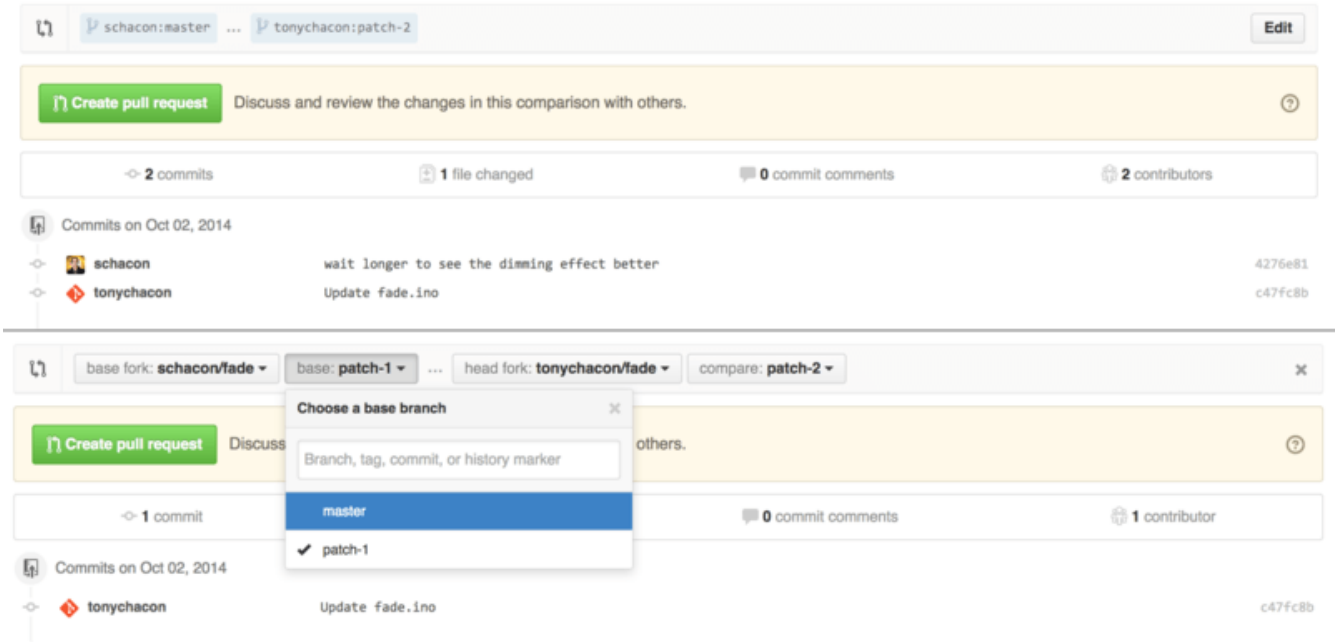


Рисунок 117. Ручное изменение форка и ветки для запроса слияния

Здесь можно указать вашу новую ветку для слияния с другим запросом слияния или другим форком проекта.

Упоминания и уведомления

GitHub обладает отличной встроенной системой уведомлений, которая может пригодиться для решения вопросов или получения обратной связи от конкретных людей или команд.

В любом комментарии можно написать символ `@`, что автоматически вызовет список автодополнения с именами пользователей, которые включены в проект или просто участвуют в нём.

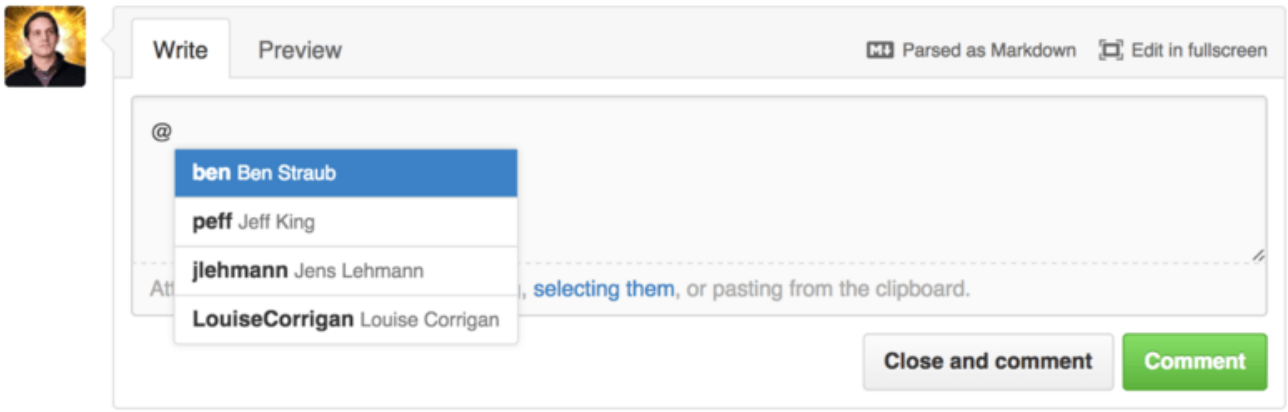


Рисунок 118. Напишите @ для упоминания кого-либо

Так же можно упомянуть пользователя, не указанного в выпадающем списке, но с помощью автодополнения это можно сделать быстрее.

Как только вы оставите комментарий с упоминанием пользователя, ему будет отправлено уведомление. Таким образом, можно более эффективно вовлекать пользователей в обсуждение, не опрашивая их непосредственно. Очень часто в запросах слияния на GitHub пользователи приглашают других людей в свои команды или компании для рецензии

проблем или запросов слияния.

Если кто-то будет упомянут в запросе слияния или проблеме, то он автоматически «подписывается» и будет получать уведомления о последующей активности. Вы так же будете подписаны на некоторые уведомления если просто откроете запрос слияния или проблему, станете отслеживать репозиторий или если оставите комментарий. Для прекращения отправки вам уведомлений нажмите кнопку «Unsubscribe».

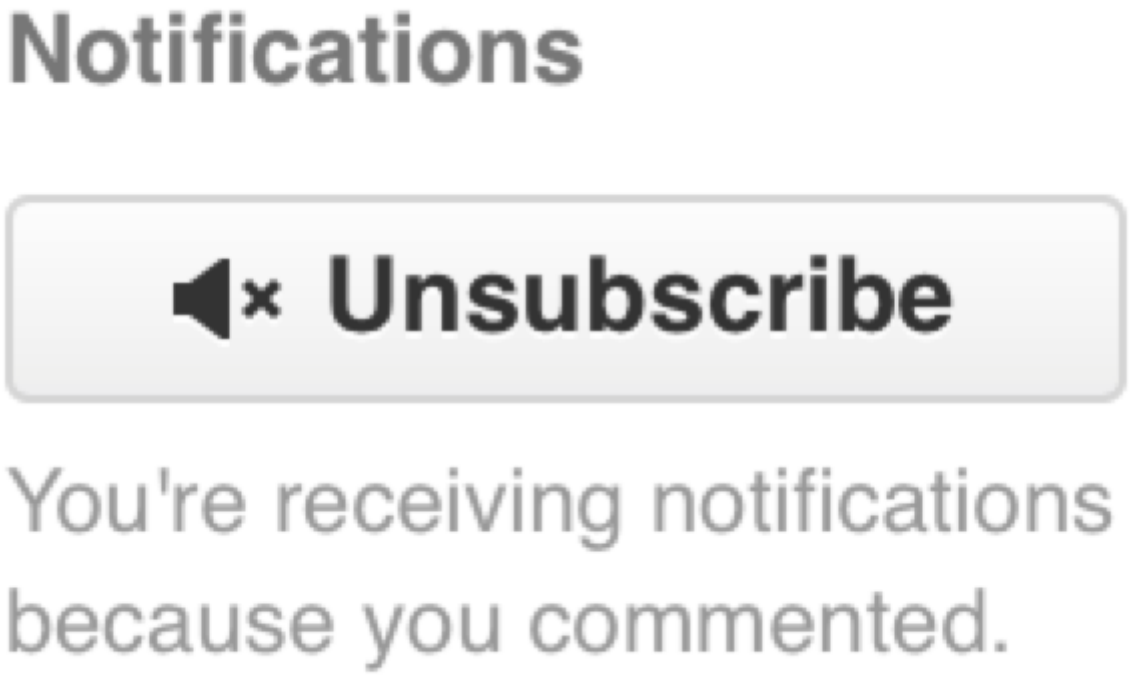


Рисунок 119. Отказ от подписки на проблему или запрос слияния

Страница уведомлений

Когда мы говорим «уведомления» в контексте GitHub, мы имеем в виду способ, которым GitHub пытается связаться с вами в случае возникновения каких-либо событий, настроить который можно несколькими способами. Для просмотра настроек уведомлений перейдите на закладку «Notification center» на странице настроек.

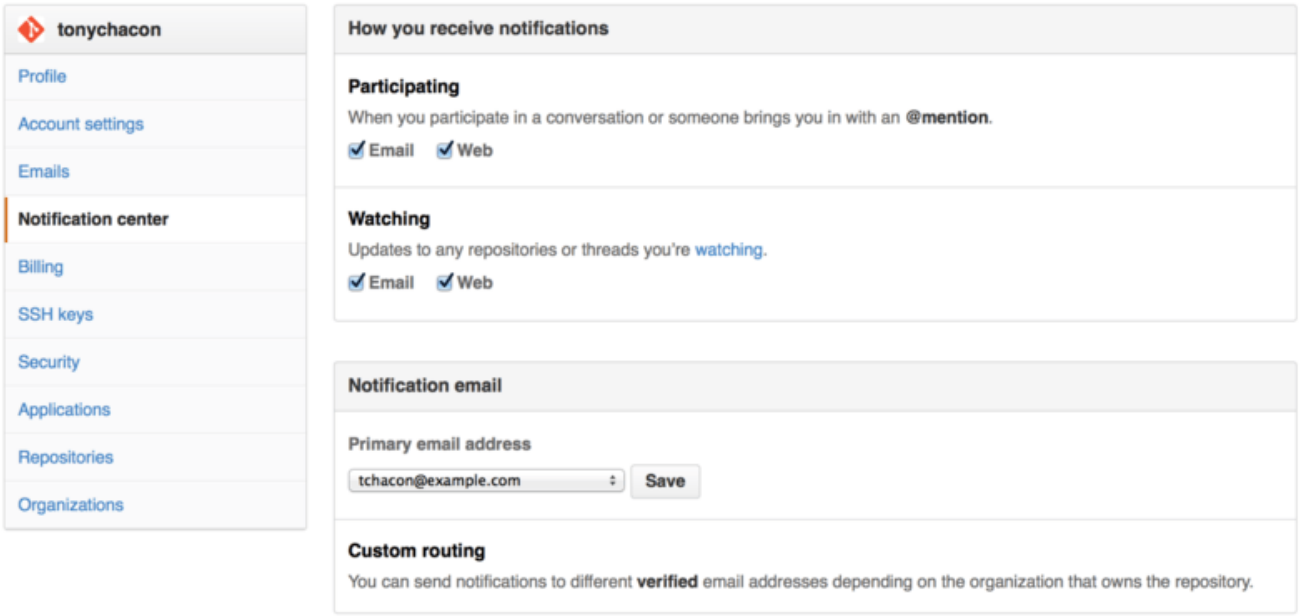


Рисунок 120. Настройки центра уведомлений

Доступны два вида уведомлений: посредствам «Email» и «Web». Вы можете выбрать один, ни одного или оба, если активно участвуете в событиях отслеживаемых репозиториев.

Web уведомления

Такие уведомления существуют только на GitHub и посмотреть их можно только на GitHub. Если эта опция включена у вас в настройках и уведомление сработало для вас, то вы увидите небольшую синюю точку на иконке уведомлений вверху экрана, как показано на рисунке Центр уведомлений.

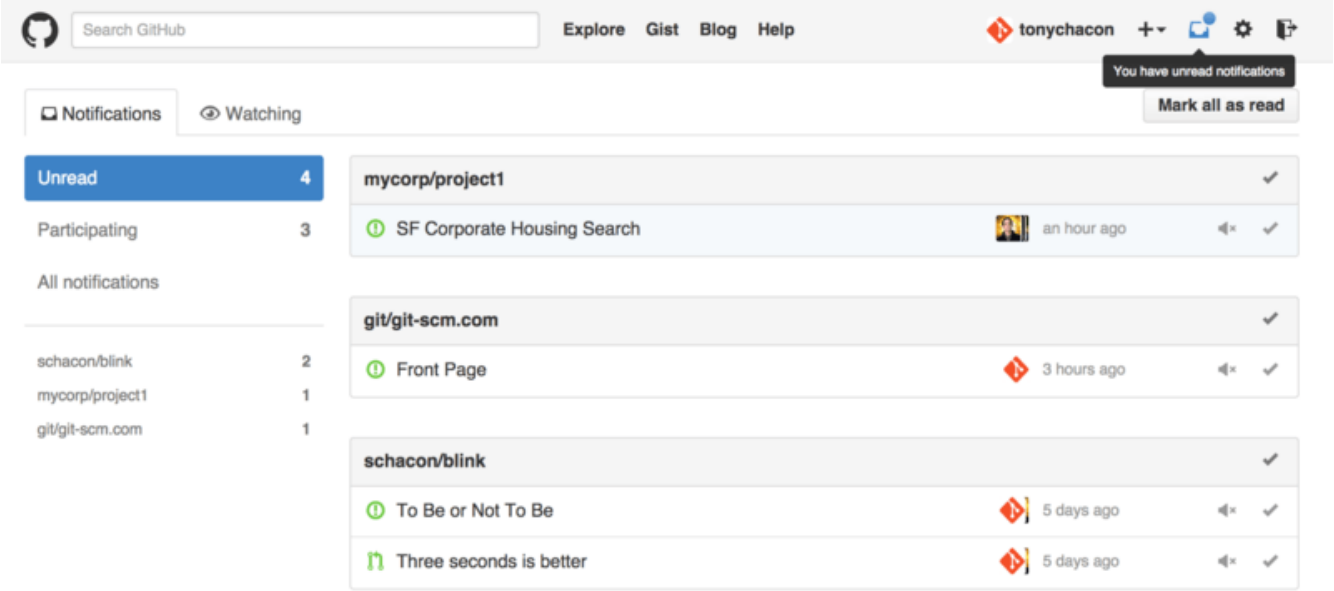


Рисунок 121. Центр уведомлений

Кликнув по иконке, вы увидите список всех уведомлений, сгруппированных по проектам. Вы можете фильтровать уведомления по конкретному проекту, кликнув по его названию на боковой панели слева. Так же вы можете подтверждать получение уведомлений, кликнув

по галочке рядом с любым из уведомлений, или подтвердить *все* уведомления по проекту, кликнув по галочке в шапке группы. После каждой галочки так же есть кнопка отключения, кликнув по которой вы перестанете получать уведомления по данному элементу.

Эти инструменты очень полезны при обработке большого числа уведомлений. Продвинутые пользователи GitHub полностью отключают email уведомления и пользуются этой страницей.

Email уведомления

Email уведомления — это ещё один способ, которым вы можете получать уведомления от GitHub. Если эта опция включена, то вы будете получать по письму на каждое уведомление. Примеры вы видели в разделах Комментарии, отправленные по электронной почте и Email уведомление о новом запросе слияния. Письма объединяются в цепочки, что очень удобно при использовании соответствующего почтового клиента.

GitHub включает много дополнительных метаданных в заголовки каждого письма, что полезно при настройке различных фильтров и правил сортировки.

Например, если взглянуть на заголовки письма, отправленного Тони в примере Email уведомление о новом запросе слияния, то можно увидеть следующее:

```
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
```

Здесь можно увидеть несколько интересных вещей. Если вы хотите выделить или перенаправить письма конкретного проекта или запроса на слияние, то информация, содержащаяся в заголовке `Message-ID`, предоставляет вам соответствующие сведения в формате `<пользователь>/<проект>/<тип>/<идентификатор>`. Для задачи вместо «pull» будет указано «issues».

Заголовки `List-Post` и `List-Unsubscribe`, при наличии у вас почтового клиента, который их понимает, позволяют легко написать в список рассылки или отписаться от неё. Это то же самое, что и нажать кнопку «mute» в веб версии уведомлений или «Unsubscribe» на странице задачи или запроса на слияние.

Если включены оба типа уведомлений и ваш почтовый клиент отображает картинки, то при просмотре email версии уведомления, веб версия так же будет отмечена как прочитанная.

Особенные файлы

Существует несколько особенных файлов, которые GitHub заметит при наличии их в вашем репозитории.

README

Первый — это файл `README`, он может быть в любом формате, который GitHub в состоянии распознать. Например, это может быть `README`, `README.md`, `README.asciidoc` и так далее. Если GitHub увидит такой файл в вашем исходном коде, то отобразит его на заглавной странице проекта.

Большинство команд используют его для поддержания актуальной информации о проекте для новичков. Как правило, он включает следующее:

- Для чего предназначен проект
- Инструкции по конфигурации и установке
- Примеры использования
- Используемую лицензию
- Правила участия в проекте

В этот файл можно встраивать изображения или ссылки для простоты восприятия информации.

CONTRIBUTING

Следующий файл — это `CONTRIBUTING`. Если в вашем репозитории будет файл `CONTRIBUTING` с любым расширением, то GitHub будет показывать ссылку на него при создании любого запроса на слияние.

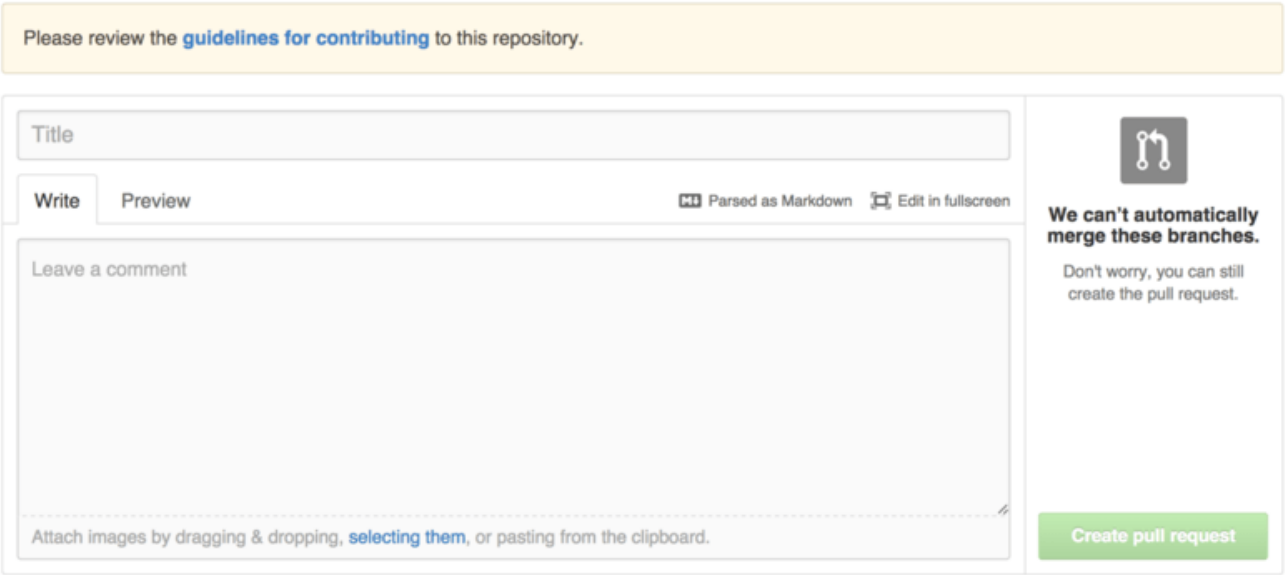


Рисунок 122. Создание запроса на слияние при наличии файла CONTRIBUTING

Идея состоит в том, что вы можете указать конкретные вещи, которые вы хотите или не

хотите видеть в новых запросах на слияние. Таким образом, люди могут ознакомится с руководством перед тем как создавать новый запрос на слияние.

Управление проектом

Для одного проекта не так уж и много администраторских действий, но есть несколько стоящих внимания.

Изменение основной ветки

Если вы используете в качестве основной другую ветку, отличную от «master», и хотите, чтобы пользователи открывали запросы на слияние к ней, то это можно изменить в настройках репозитория на закладке «Options».

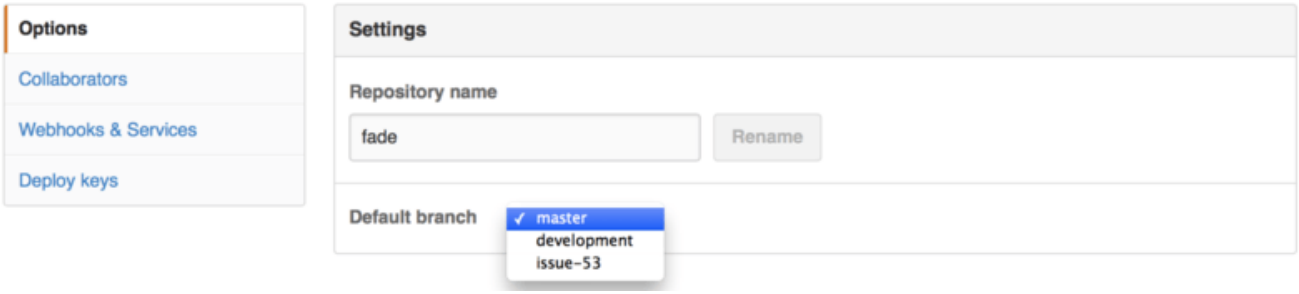


Рисунок 123. Изменение основной ветки проекта

Просто выберите нужную ветку из выпадающего меню и она станет основной для большинства операций, включая извлечение кода при клонировании репозитория.

Передача проекта

Если вы хотите передать проект другому пользователю или организации на GitHub, то это можно сделать нажатием кнопки «Transfer ownership» в настройках репозитория на закладке «Options».

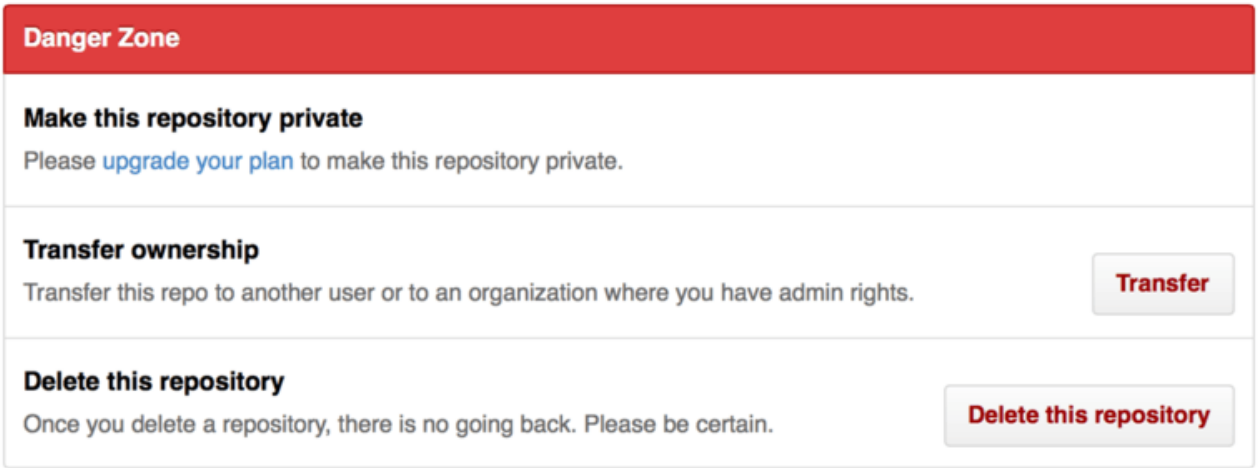


Рисунок 124. Передача проекта другому пользователю или организации на GitHub

Эта опция полезна, когда вы хотите отказаться от проекта, а кто-то другой хочет им заниматься, или когда ваш проект растёт и вы хотите передать его какой-нибудь

организации.

Это действие приведёт не только к передаче репозитория со всеми его подписчиками и звёздами, но и добавит перенаправление с вашего URL на новый. Кроме этого, изменятся ссылки для клонирования и получения изменений из Git, а не только для веб запросов.

Управление организацией

В дополнение к персональным аккаунтам, на GitHub есть организации. Для организаций, как и для персональных аккаунтов, существует окружение, где находятся все проекты, но в остальном много отличий. Такие аккаунты представляют собой группу людей, совместно владеющих проектами, и существует много инструментов для разделения их на подгруппы. Обычно, такие аккаунты используются группами, которые работают с публичными проектами (такими как «perl» или «rails»), или компаниями (такими как «google» или «twitter»).

Основы организаций

Создать новую организацию очень легко; просто нажмите на иконку «+» в правом верхнем углу страницы GitHub и выберите пункт «New organization» из меню.

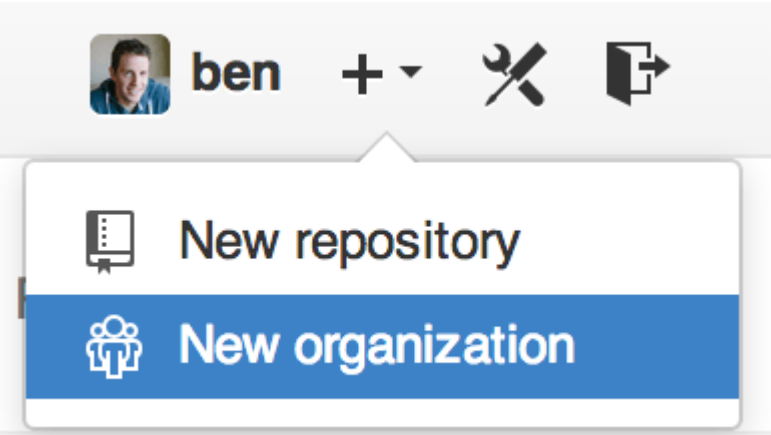


Рисунок 125. Пункт меню «New organization»

Для начала, следует указать название вашей организации и ввести email адрес в качестве основного способа связи. После этого можно приглашать людей в качестве совладельцев аккаунта.

Следуйте инструкциям и в скором времени вы станете владельцем новой компании. Организации, как и персональные аккаунты, бесплатны, если вы планируете работать над проектами с открытым исходным кодом.

Как владельцу организации, при клонировании репозитория вам будет предложено сохранить его в окружение организации. При создании нового репозитория, вы можете его сохранить как в персональном окружении, так и в окружении любой компании, владельцем которой вы являетесь. Так же вы автоматически начинаете отслеживать все создаваемые репозитории в организации.

Как для персонального аккаунта, так и для организации вы можете загрузить отдельную

картинку. Аналогично и для главной страницы, где приводится список доступных репозиториев организации.

Теперь, давайте рассмотрим отличительные черты аккаунтов организаций.

Команды

Организации связаны с отдельными людьми посредством команд, которые представляют собой сгруппированные аккаунты индивидуальных пользователей, репозиториев внутри организации и того, какой доступ эти люди имеют в этих репозиториях.

Например, у вашей компании есть три репозитория: `frontend`, `backend` и `deployscripts`. Вы бы хотели, чтобы ваши разработчики HTML/CSS/JavaScript имели доступ к `frontend` и возможно к `backend`, а ваши администраторы имели доступ к `backend` и `deployscripts`. С помощью команд это легко реализовать, не настраивая доступ к каждому репозиторию для каждого участника.

Страница Организации представляет собой простую панель управления репозиториями, пользователями и командами в пределах данной организации.

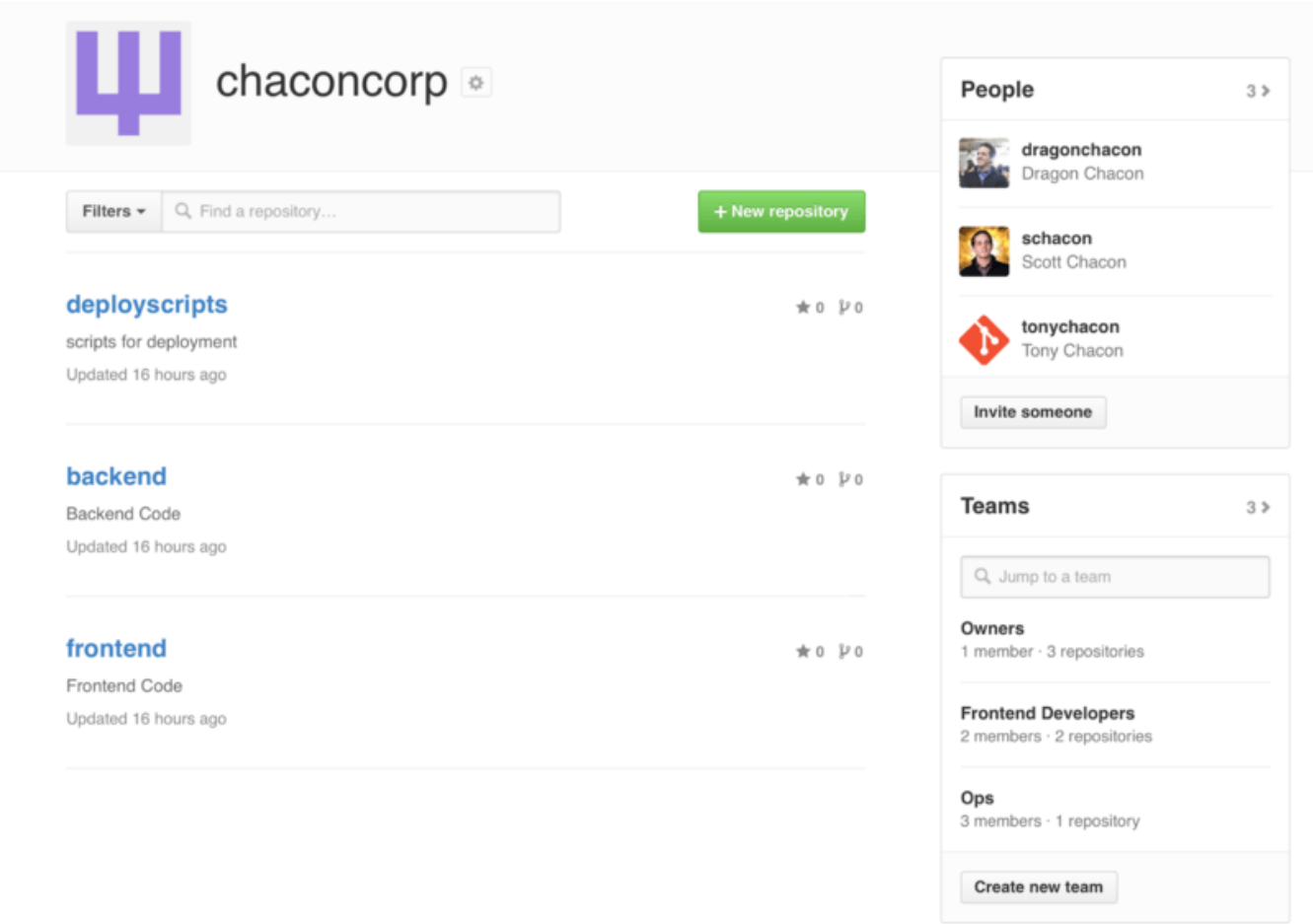


Рисунок 126. Страница организации

Для управления командами нужно перейти на закладку 'Teams' справа вверху на странице Страница организации. Это приведёт вас на страницу где можно добавлять пользователей в команду, добавлять команде репозитории или управлять настройками и правами доступа. Каждая команда может иметь только следующие уровни доступа к репозиториям: «только

чтение», «чтение/запись» или «администратор». Уровень доступа может быть изменён нажатием кнопки «Settings» на странице Страница команды.

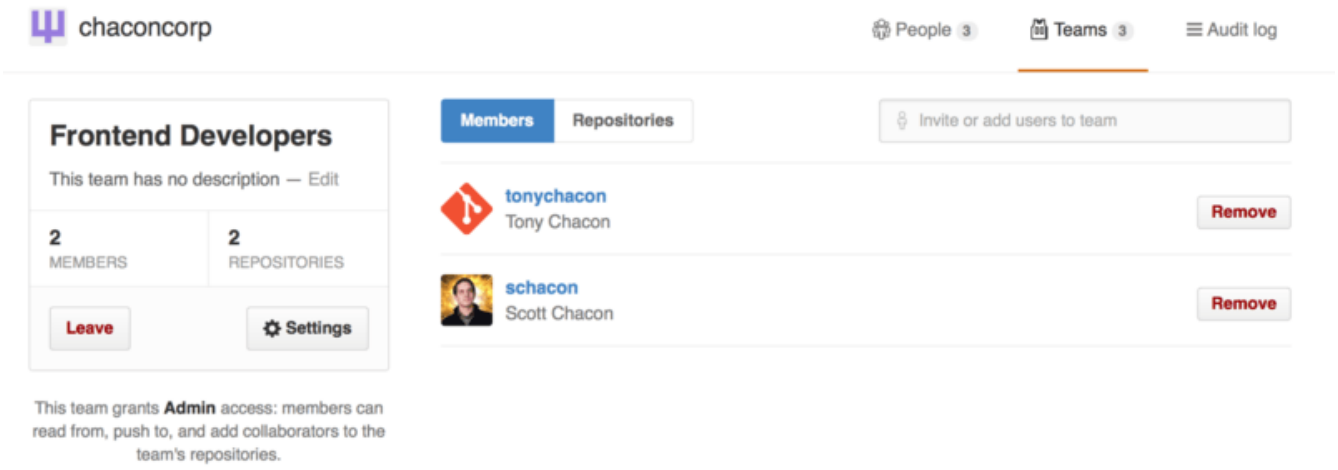


Рисунок 127. Страница команды

Когда вы пригласите кого-нибудь в команду, то будет отправлено письмо с приглашением.

Упоминания команд (`@mentions`), такие как `@acmecorp/frontend`, работают точно так же как и упоминания отдельных пользователей, за исключением того, что уведомляются **все** члены команды. Это полезно, когда вы хотите привлечь внимание кого-нибудь из команды, но точно не знаете, кого спросить.

Пользователь может принадлежать любому числу команд, поэтому не ограничивайте себя командами, разделёнными по уровню доступа. Специализированные команды, такие как `ux`, `css` или `refactoring` полезны для вопросов одной тематики, тогда как команды `legal` и `colorblind` — для вопросов другой тематики.

Журнал аудита

Организации так же предоставляют владельцам информацию о всех происходящих событиях внутри них. Перейдя на закладку 'Audit Log' вы можете увидеть произошедшие события на уровне организации, кто участвовал в них и в какой точке мира они произошли.

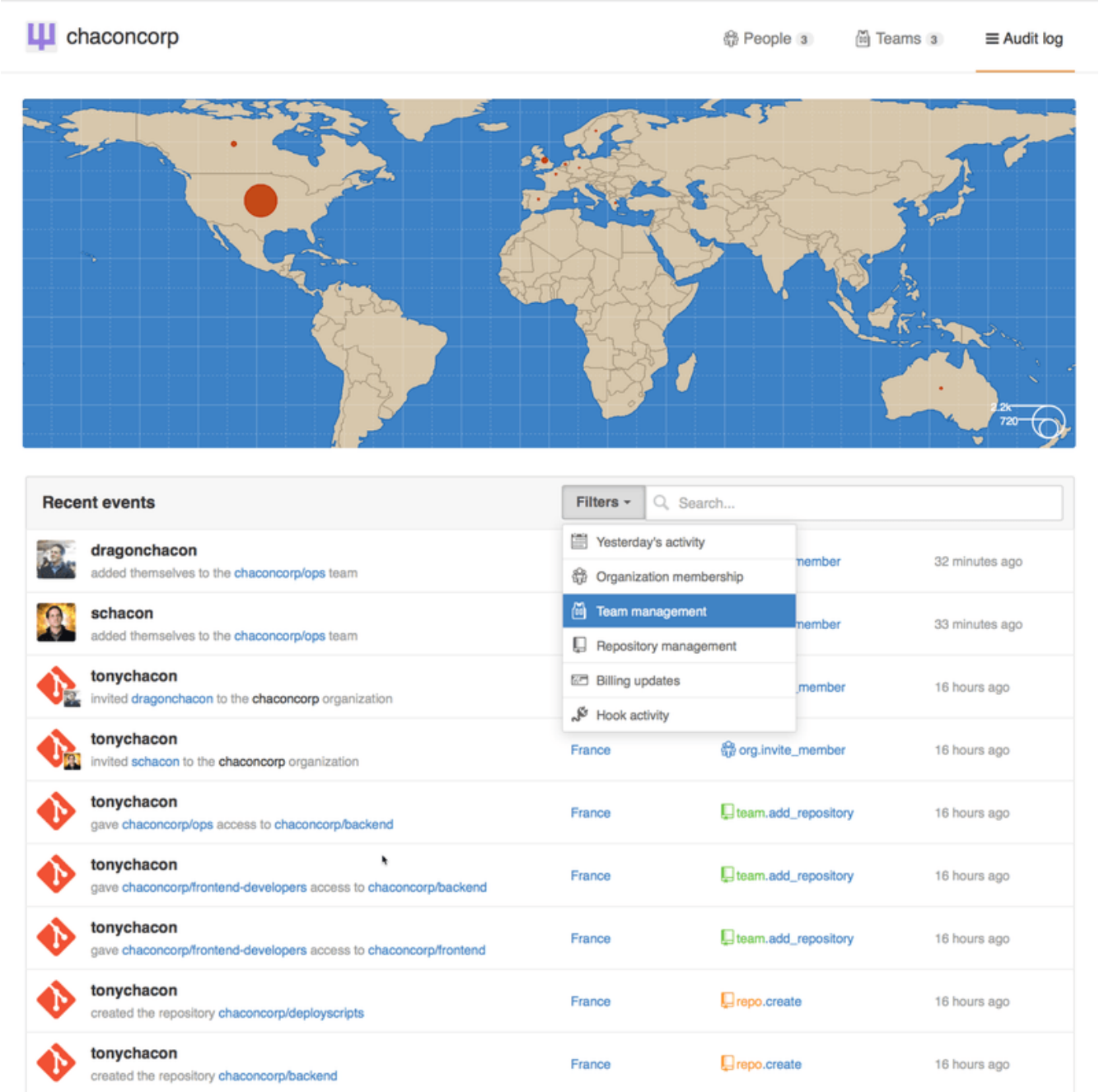


Рисунок 128. Журнал аудита

Вы так же можете отфильтровать события по типам, определённым людям или местам.

Создание сценариев GitHub

Итак, мы рассмотрели все основные функции и рабочие процессы GitHub, но у любой большой группы или проекта могут быть настройки, которые они могут захотеть сделать, или внешние сервисы, которые они могут захотеть интегрировать.

К счастью для нас, GitHub действительно можно взломать во многих отношениях. В этом разделе мы расскажем, как использовать систему хуков GitHub и его API, чтобы заставить GitHub работать так, как мы хотим.

Сервисы и хуки

Раздел Hooks and Services администрирования репозитория GitHub — это самый простой способ взаимодействия GitHub с внешними системами.

Сервисы

Сначала мы рассмотрим сервисы. Интеграцию хуков и сервисов можно найти в разделе «Настройки» («Settings») вашего репозитория, где ранее мы рассматривали возможность добавления соавторов и изменения ветки вашего проекта по умолчанию. На вкладке «Вебхуки и сервисы» («Webhooks and Services») вы увидите что-то вроде Раздел настройки сервисов и хуков.

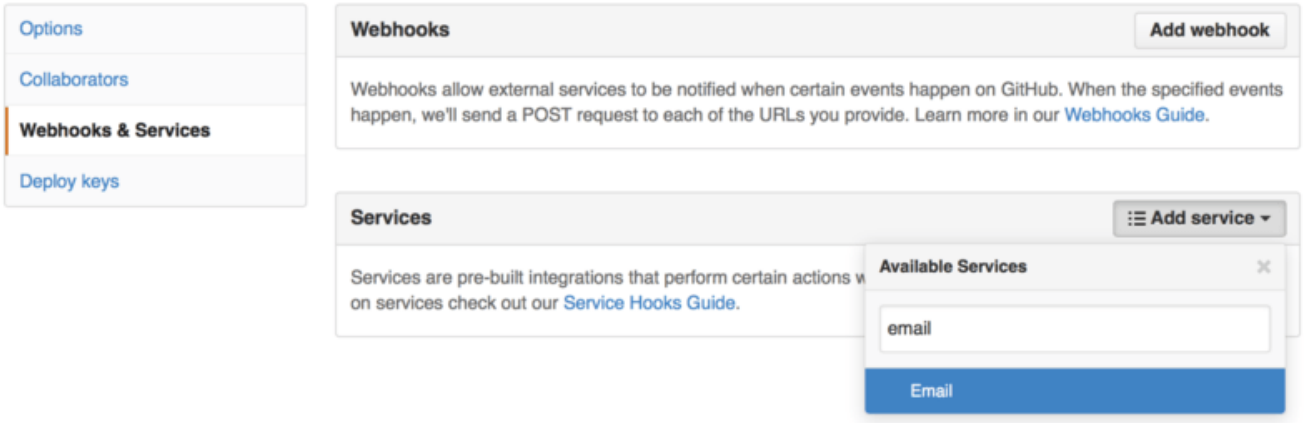


Рисунок 129. Раздел настройки сервисов и хуков

Вы можете выбирать из десятков сервисов, большинство из которых интегрируются в другие коммерческие системы и системы с открытым исходным кодом. Большинство из них предназначены для сервисов непрерывной интеграции, систем отслеживания ошибок и проблем, систем чатов и систем документации. Мы рассмотрим настройку очень простого хука электронной почты. Если вы выберете «email» в раскрывающемся списке «Добавить сервис» («Add Service»), вы получите экран конфигурации, например Конфигурация службы электронной почты.

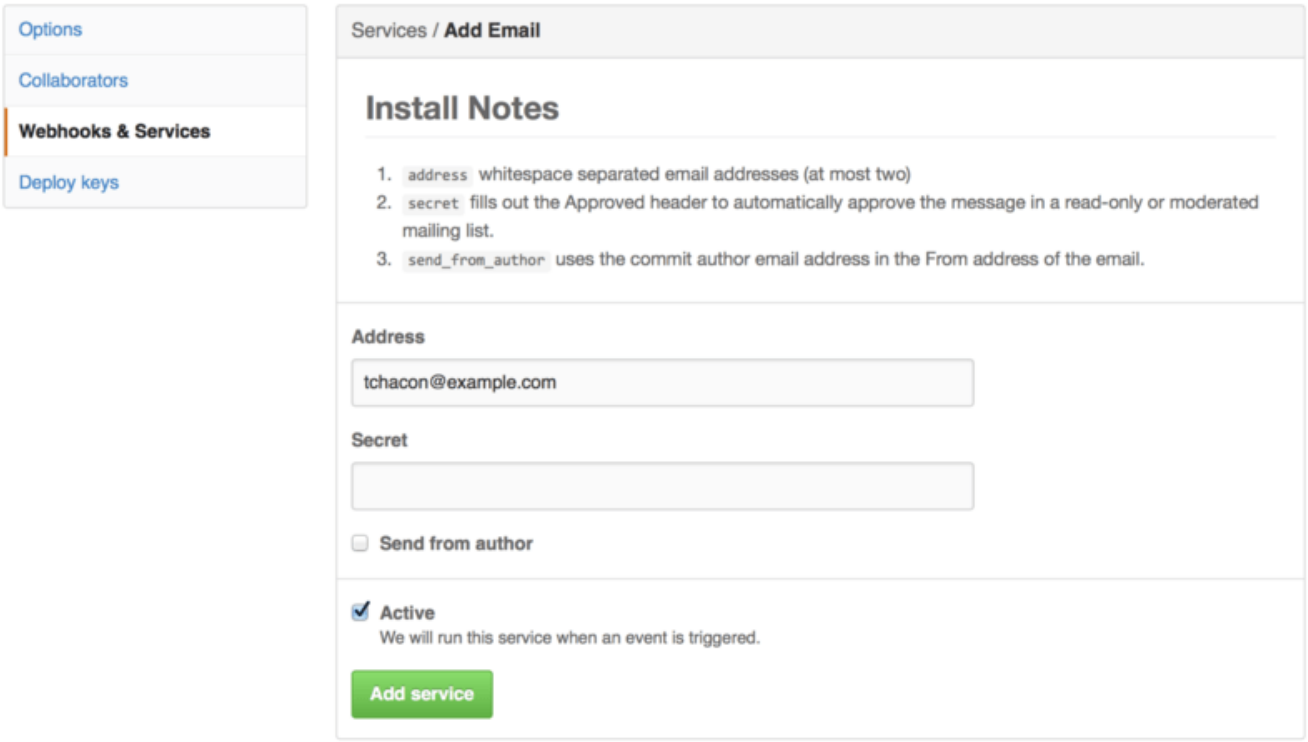


Рисунок 130. Конфигурация службы электронной почты

В этом случае, если мы нажмём кнопку «Добавить сервис» («Add Service»), указанный нами адрес электронной почты будет получать электронное письмо каждый раз, когда кто-то отправляет в репозиторий. Сервисы могут прослушивать множество различных типов событий, но большинство из них прослушивают только push-события, а затем что-то делают с этими данными.

Если вы используете систему, которую хотите интегрировать с GitHub, вам следует проверить здесь, чтобы узнать, доступна ли существующая интеграция сервиса. Например, если вы используете Jenkins для запуска тестов своей кодовой базы, вы можете включить интеграцию встроенного сервиса Jenkins, чтобы запускать тестовый прогон каждый раз, когда кто-то отправляет данные в ваш репозиторий.

Хуки

Если вам нужно что-то более конкретное или вы хотите интегрироваться с сервисом или сайтом, не включённым в этот список, вы можете вместо этого использовать более общую систему хуков. Хуки репозитория GitHub довольно просты. Вы указываете URL-адрес, и GitHub отправит полезные данные HTTP на этот URL-адрес для любого события, которое вы хотите.

Как правило, это работает так: вы можете настроить небольшой веб-сервис для прослушивания полезных данных хука GitHub, а затем что-то делать с данными, когда они будут получены.

Чтобы включить хук, вы нажимаете кнопку «Добавить вебхук» («Add webhook») в Раздел настройки сервисов и хуков. Это приведёт вас на страницу, которая выглядит как Конфигурация вебхука.

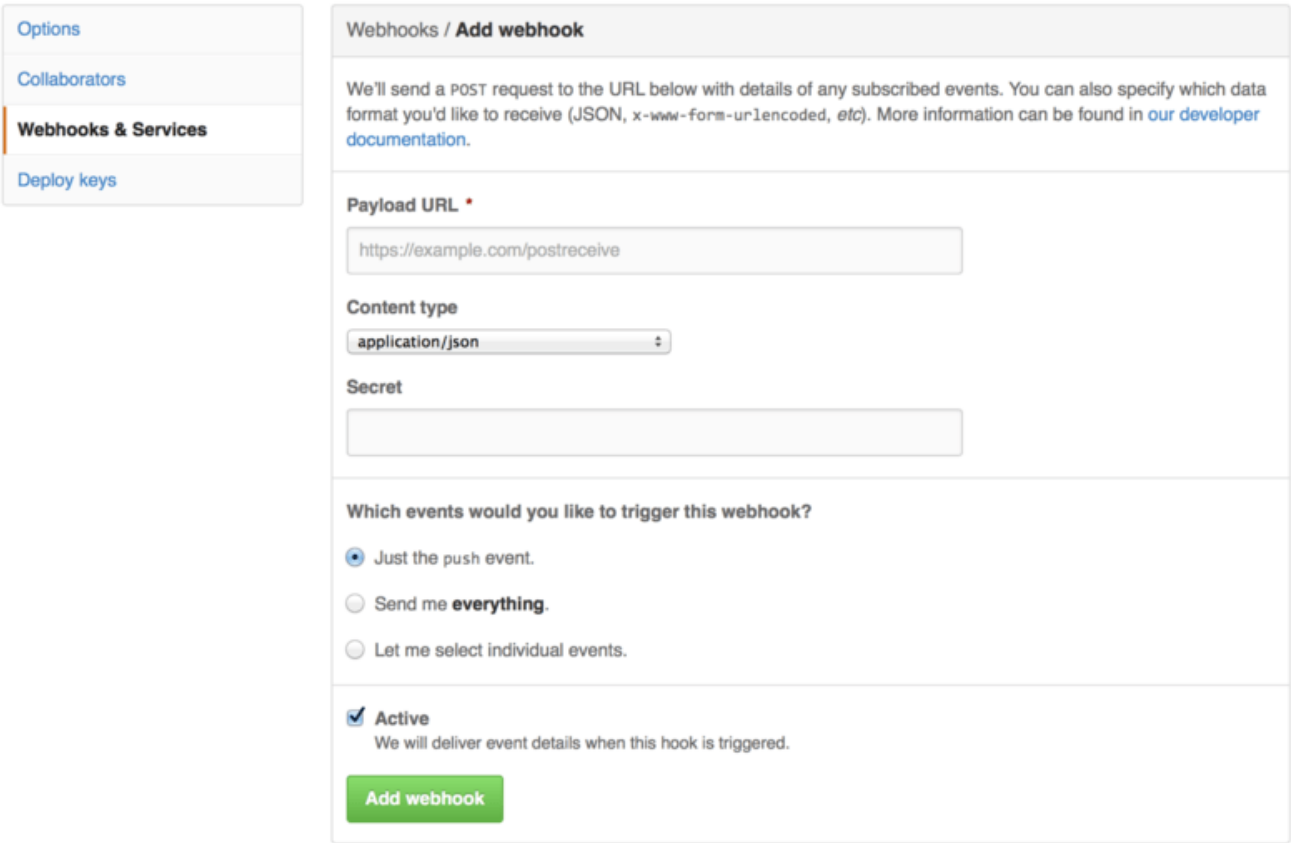


Рисунок 131. Конфигурация вебхука

Конфигурация вебхука довольно проста. В большинстве случаев вы просто вводите URL и секретный ключ и нажимаете «Добавить вебхук» («Add webhook»). Есть несколько вариантов, для каких событий вы хотите, чтобы GitHub отправлял вам полезные данные — по умолчанию вы получаете полезные данные только для события `push`, когда кто-то отправляет новый код в любую ветку вашего репозитория.

Давайте рассмотрим небольшой пример веб-сервиса, который вы можете настроить для обработки вебхука. Мы будем использовать веб-фреймворк Ruby Sinatra, так как он довольно лаконичен, и вы сможете легко увидеть, что мы делаем.

Допустим, мы хотим получать электронное письмо, если конкретный человек отправляет на определённую ветку нашего проекта изменённый определённый файл. Мы могли бы довольно легко сделать это с помощью такого кода:

```
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['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
```

Здесь мы берём полезные данные JSON, которые доставляет нам GitHub, и ищем, кто их отправил, в какую ветку они отправили и какие файлы были затронуты во всех отправленных коммитах. Затем мы проверяем это на соответствие нашим критериям и отправляем электронное письмо, если оно соответствует.

Чтобы разработать и протестировать что-то подобное, у вас есть хорошая консоль разработчика на том же экране, где вы устанавливаете связь. Вы можете увидеть последние несколько доставок, которые GitHub пытался сделать для этого вебхука. Для каждого хука вы можете узнать, когда он был доставлен, если он был успешным, а также тело и заголовки как для запроса, так и для ответа. Это делает невероятно простым тестирование и отладку ваших хуков.

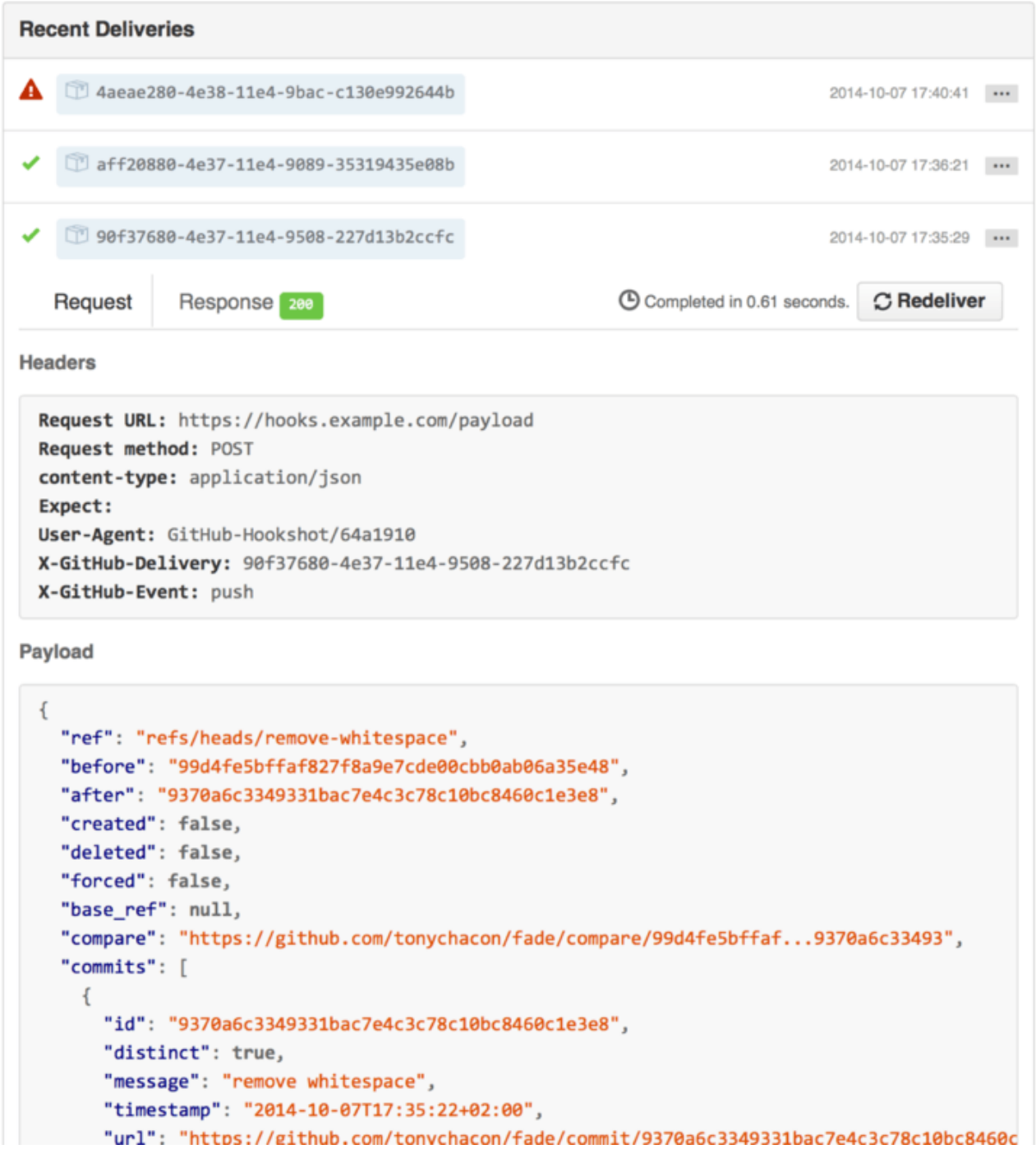


Рисунок 132. Информация об отладке вебхуков

Другая замечательная особенность этого заключается в том, что вы можете повторно доставить любые полезные данные, чтобы легко протестировать свой сервис.

Для получения дополнительной информации о том, как писать вебхуки и обо всех различных типах событий, которые вы можете прослушивать, перейдите к документации GitHub Developer по адресу https://developer.github.com/webhooks/.

GitHub API

Сервисы и хуки дают вам возможность получать push-уведомления о событиях, происходящих в ваших репозиториях, но что, если вам нужна дополнительная информация об этих событиях? Что, если вам нужно автоматизировать что-то вроде добавления

соавторов или маркировки проблем?

Вот где API GitHub пригодится. GitHub имеет множество конечных точек API для автоматического выполнения почти всего, что вы можете делать на веб-сайте. В этом разделе мы узнаем, как пройти аутентификацию и подключиться к API, как прокомментировать проблему и как изменить статус запроса на слияние через API.

Основное использование

Самое простое, что вы можете сделать, — это отправить простой запрос GET на конечную точку, не требующую аутентификации. Это может быть пользовательская или доступная только для чтения информация о проекте с открытым исходным кодом. Например, если мы хотим узнать больше о пользователе с именем «schacon», мы можем запустить что-то вроде этого:

```
$ 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"
}
```

Существует множество подобных конечных точек для получения информации об организациях, проектах, проблемах, коммитах — почти обо всём, что вы можете публично увидеть на GitHub. Вы даже можете использовать API для визуализации произвольного Markdown или найти шаблон `.gitignore`.

```
$ 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
https://www.java.com/en/download/help/error_hotspot.xml
```

```
hs_err_pid*
"
}
```

Комментирование проблемы

Однако, если вы хотите выполнить какое-либо действие на веб-сайте, например, прокомментировать проблему или запрос на слияние, или если вы хотите просмотреть или взаимодействовать с частным контентом, вам необходимо пройти аутентификацию.

Существует несколько способов аутентификации. Вы можете использовать обычную аутентификацию только с вашим именем пользователя и паролем, но, как правило, лучше использовать токен личного доступа. Вы можете сгенерировать его на вкладке «Приложения» («Applications») на странице настроек.

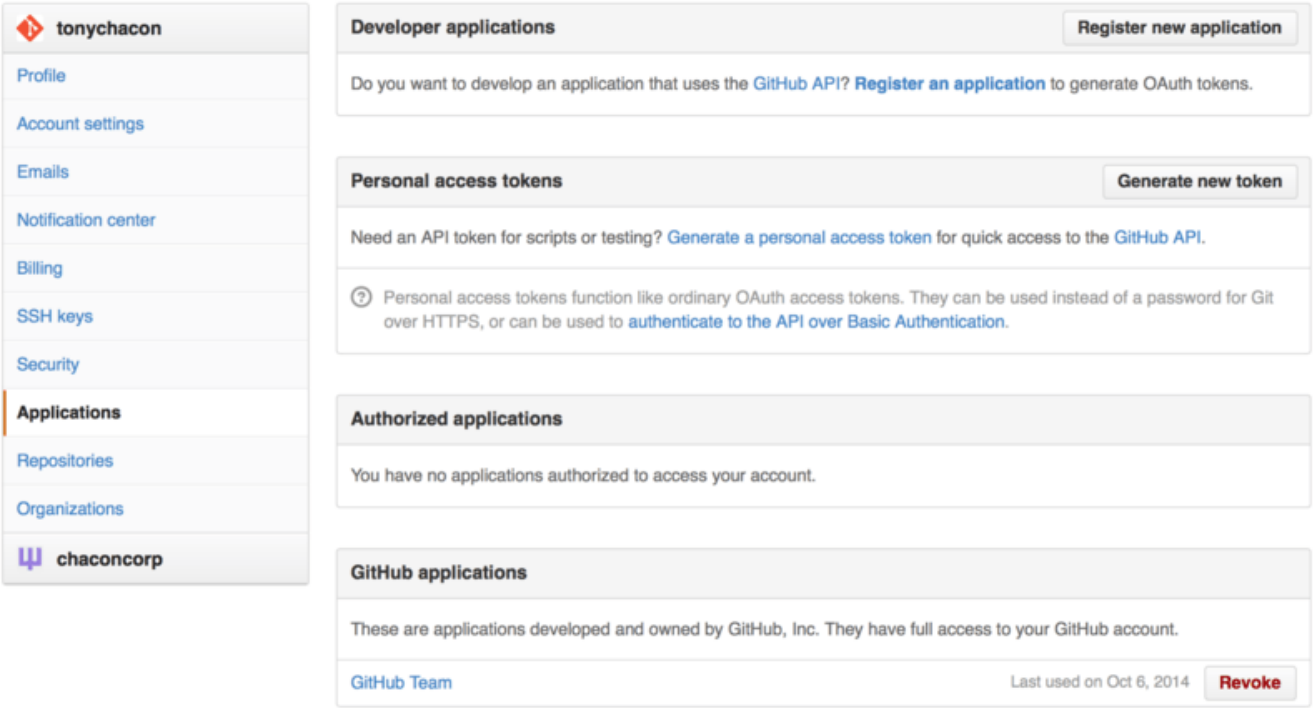


Рисунок 133. Сгенерируйте токен доступа на вкладке «Приложения» («Applications») на странице настроек

Он спросит вас, какие области вы хотите для этого токена и описание. Обязательно используйте хорошее описание, чтобы вам было удобно удалять токен, когда ваш скрипт или приложение больше не используются.

GitHub покажет вам токен только один раз, поэтому обязательно скопируйте его. Теперь вы можете использовать это для аутентификации в своём скрипте вместо использования имени пользователя и пароля. Это хорошо, потому что вы можете ограничить объём того, что вы хотите сделать, и токен может быть отозван.

Это также имеет дополнительное преимущество в виде увеличения лимита скорости. Без аутентификации вы будете ограничены 60 запросами в час. Если вы аутентифицируетесь, вы можете делать до 5000 запросов в час.

Итак, давайте воспользуемся им, чтобы прокомментировать одну из наших проблем. Допустим, мы хотим оставить комментарий к конкретной проблеме № 6 (Issue #6). Для этого нам нужно выполнить HTTP-запрос POST к `repos/<user>/<repo>/issues/<num>/comments` с токеном, который мы только что сгенерировали в качестве заголовка авторизации.

```
$ 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:"
}
```

Теперь, если вы перейдёте к этой проблеме, вы увидите комментарий, который мы только что успешно разместили, например Комментарий, отправленный из GitHub API.

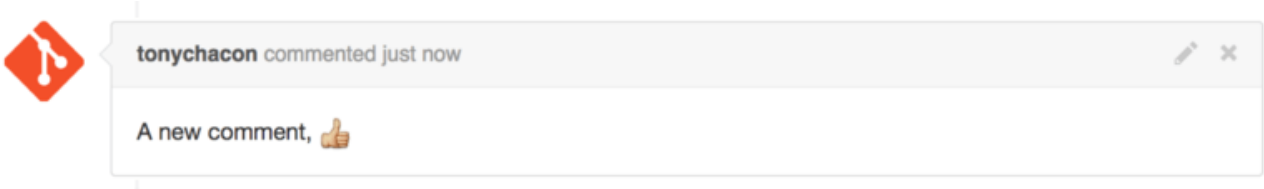


Рисунок 134. Комментарий, отправленный из GitHub API

Вы можете использовать API, чтобы делать практически всё, что вы можете делать на веб-сайте: создавать и устанавливать вехи (milestones), назначать людей для проблем и запросов на слияние, создавать и изменять метки, получать доступ к данным коммитов, создавать новые коммиты и ветки, открывать, закрывать или объединение запросов на слияние, создание и редактирование команд, комментирование строк кода в запросе на слияние, поиск по сайту и так далее.

Изменение статуса запроса на слияние

Есть ещё один последний пример, который мы рассмотрим, так как он действительно полезен, если вы работаете с запросами на слияние. С каждым коммитом может быть связан один или несколько статусов, и существует API для добавления и запроса этого статуса.

Большинство сервисов непрерывной интеграции и тестирования используют этот API для

реагирования на отправку путём тестирования кода, который был отправлен, а затем сообщают, прошёл ли этот коммит все тесты. Вы также можете использовать это, чтобы проверить, правильно ли отформатировано сообщение о коммите, следовал ли отправитель всем вашим рекомендациям по вкладу, был ли коммит действительно подписан — что угодно.

Допустим, вы настроили вебхук в своём репозитории, который обращается к небольшому веб-сервису, который проверяет наличие строки «Signed-off-by» в сообщении коммита.

```
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
```

Надеюсь, следовать этому довольно просто. В этом обработчике вебхука мы просматриваем каждый только что отправленный коммит, ищем строку «Signed-off-by» в сообщении коммита и, наконец, отправляем POST через HTTP в `/repos/<user>/<repo>/statuses/<commit_sha>` Конечная точка API со статусом.

В этом случае вы можете отправить состояние («успех» ('success'), «сбой» ('failure'), «ошибка» ('error')), описание того, что произошло, целевой URL-адрес, по которому пользователь может перейти для получения дополнительной информации, и «контекст» в случае наличия нескольких статусов за один коммит. Например, сервис тестирования может предоставить статус, а сервис проверки, подобная этой, также может предоставить статус — поле «контекст» — это то, как они различаются.

Если кто-то откроет новый запрос на слияние на GitHub и этот хук настроен, вы можете увидеть что-то вроде Статус коммита через API.

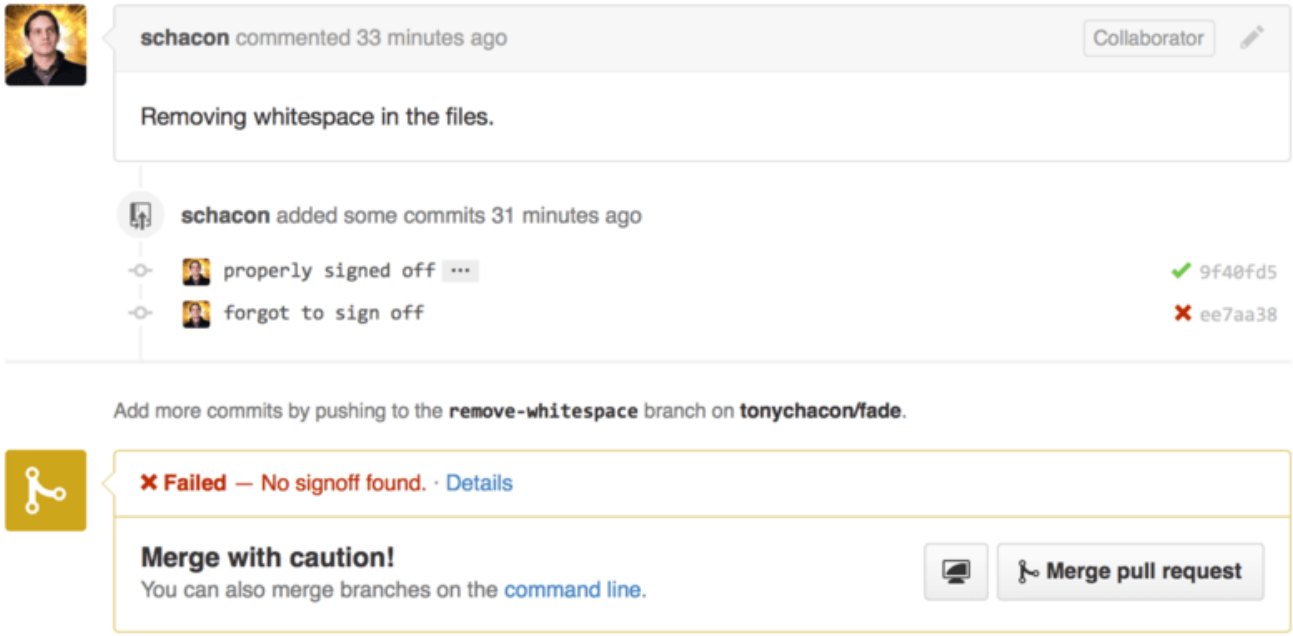


Рисунок 135. Статус коммита через API

Теперь вы можете увидеть маленькую зелёную галочку рядом с коммитом, в сообщении которой есть строка «Signed-off-by», и красным крестиком тот коммит, который автор забыл подписать. Вы также можете видеть, что запрос на слияние принимает статус последнего коммита в ветке и предупреждает вас, если это сбой. Это действительно полезно, если вы используете этот API для результатов тестирования, чтобы случайно не объединить что-то, где последний коммит не прошёл тесты.

Octokit

Хотя в этих примерах мы делали почти всё с помощью `curl` и простых HTTP-запросов, существует несколько библиотек с открытым исходным кодом, которые делают этот API доступным более идиоматичным образом. На момент написания этой статьи поддерживаемые языки включают Go, Objective-C, Ruby и .NET. Посетите https://github.com/octokit для получения дополнительной информации о них, так как они обрабатывают большую часть HTTP для вас.

Надеюсь, эти инструменты помогут вам настроить и модифицировать GitHub, чтобы он лучше работал с вашими конкретными рабочими процессами. Для получения полной документации по всему API, а также руководств по общим задачам посетите https://developer.github.com.

Заключение

Теперь вы полноценный пользователь GitHub. 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 коммиты или их диапазоны. Эти способы не всегда очевидны, но их полезно знать.

Одиночные ревизии

Конечно, вы можете ссылаться на коммит по его SHA-1 хешу, но существуют более удобные для человека способы. В данном разделе описываются различные способы обращения к одному коммиту.

Сокращённый SHA-1

Git достаточно умён, чтобы понять какой коммит имеется в виду по нескольким первым символам его 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 некоторый коммит, вы выполнили команду `git log` и нашли коммит, в которой добавили определённую функциональность:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    Fix refs handling, add gc auto, update 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

    Add some blame and merge stuff
```

Предположим, что в нашем примере это коммит `1c002dd…`. Если вы хотите выполнить для него `git show`, то следующие команды эквиваленты (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 сокращённые 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 Change the version number
085bb3b Remove unnecessary test code
a11bef0 Initial 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. коммитов 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 1.2×10^{24}, или 1 миллион миллиардов миллиардов, что в 1200 раз больше количества песчинок на земле.

Приведём 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.

Если выделить на это несколько тысяч долларов вычислительной мощности, можно будет синтезировать два файла с одним и тем же хешем, что было доказано проектом https://shattered.io/ в феврале 2017 года. Git движется к использованию SHA256 в качестве алгоритма хеширования по умолчанию, который намного более устойчив к атакам с коллизиями и имеет код, помогающий смягчить эту атаку (хотя он не может полностью её устранить).

Ссылки на ветки

Для наиболее простого способа указать коммит требуется существование ветки, указывающей на этот коммит. Тогда вы можете использовать имя ветки в любой команде Git, которая ожидает коммит или значение SHA-1. Например, если вы хотите просмотреть последний коммит в ветке, то следующие команды эквивалентны (предполагается, что ветка `topic1` указывает на коммит `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`. Служебные команды подробно рассмотрены в главе Git изнутри; в основном, 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 делает в фоновом режиме, является ведение журнала ссылок, в котором сохраняется то, куда указывали HEAD и ветки за последние несколько месяцев.

Для просмотра этого журнала используется команда `git reflog`:

```
$ git reflog
734713b HEAD@{0}: commit: Fix refs handling, add gc auto, update tests
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by the 'recursive' strategy.
1c002dd HEAD@{2}: commit: Add 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 коммиты, используя эти данные. Например, чтобы посмотреть, куда ссылался указатель HEAD пять шагов назад, используйте ссылку @{5}, которую можно увидеть в выводимых данных команды reflog:

```
$ git show HEAD@{5}
```

Этот синтаксис используется и в случае, когда требуется посмотреть, в каком состоянии пребывала ветка некоторое время назад. В частности, чтобы увидеть где была ветка `master` вчера, следует написать:

```
$ git show master@{yesterday}
```

Вы увидите, что было на вершине ветки вчера. Такой способ работает только для данных, которые всё ещё содержатся в вашем журнале ссылок, поэтому вы не можете использовать её для коммитов, 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: Fix refs handling, add gc auto, update tests
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    Fix refs handling, add gc auto, update 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.

Воспринимайте reflog Git как историю командной строки

Если у вас есть опыт работы с UNIX или Linux, можете думать о reflog как об истории командной строки Git, которая подчёркивает, что то, что там есть, явно актуально только для вас и вашего «сеанса» и не имеет ничего общего с кем-либо ещё, кто может работать на той же машине.

Экранирование фигурных скобок в PowerShell

При использовании PowerShell фигурные скобки, такие как `{` и `}`, являются специальными символами и должны быть экранированы. Вы можете экранировать их с помощью апострофа `` ` `` или поместить ссылку на коммит в кавычки:

```
$ git show HEAD@{0}     # НЕ будет работать
$ git show HEAD@`{0`}   # OK
$ git show "HEAD@{0}"   # OK
```

Ссылки на предков

Ещё один популярный способ указать коммит — это использовать его родословную. Если вы поместите `^` в конце ссылки, Git поймёт, что нужно использовать родителя этого коммита. Предположим, история вашего проекта выглядит следующим образом:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b Fix refs handling, add gc auto, update tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd Add some blame and merge stuff
|/
* 1c36188 Ignore *.gem
```

```
* 9b29157 Add open3_detach to gemspec file list
```

Для просмотра предыдущего коммита достаточно написать `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'
```

Экранирование карета в Windows

В командной строке Windows (`cmd.exe`) `^` является специальным символом и требует другого обращения. Вы можете либо удвоить его, либо поместить ссылку на коммит в кавычки:

```
$ git show HEAD^     # НЕ будет работать в Windows
$ git show HEAD^^    # OK
$ git show "HEAD^"   # OK
```

Также вы можете указать число после `^` — например, `d921970^2` означает «второй родитель коммита d921970». Такой синтаксис полезен только для коммитов слияния, которые имеют больше одного родителя. Первым родителем является ветка, в которую вы выполняли слияние, а вторым — коммит в ветке, которую вы сливали:

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    Add 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 символ тильда `~`. Он также соответствует ссылке на первого родителя, поэтому `HEAD~` и `HEAD^` эквивалентны. Различия становятся заметными, когда вы указываете число. `HEAD~2` означает «первый родитель первого родителя» или «дедушка» — при этом происходит переход от заданного предка

вглубь указанное число раз. К примеру, для показанной ранее истории, коммитом `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
```

То же 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 коммит слияния) используя запись `HEAD~3^2`, и так далее.

Диапазоны коммитов

Теперь вы умеете указывать отдельные коммиты, давайте посмотрим как указывать диапазоны коммитов. Это в частности полезно для управления вашими ветками — если у вас есть множество веток, вы можете использовать указание диапазонов коммитов для ответа на вопрос «Что было сделано в этой ветке, что я ещё не слил в основную ветку?»

Две точки

Наиболее часто для указания диапазона коммитов используется синтаксис с двумя точками. Таким образом, вы, по сути, просите Git включить в диапазон коммитов только те, которые достижимы из одной, но не достижимы из другой. Для примера предположим, что ваша история выглядит, как представлено на Пример истории для выбора диапазонов коммитов.

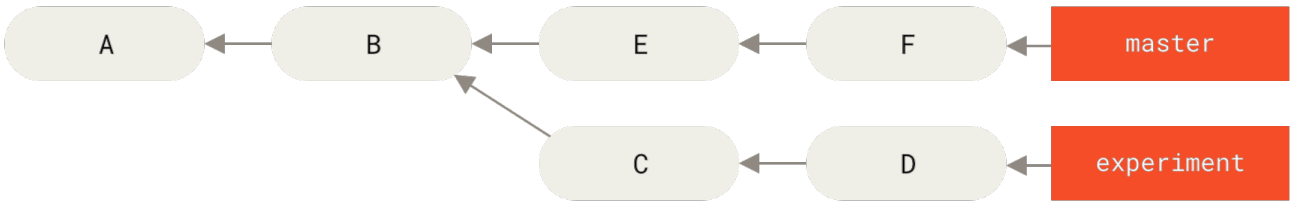


Рисунок 136. Пример истории для выбора диапазонов коммитов

Вы хотите посмотреть что находится в вашей экспериментальной ветке, которая ещё не

была слита в основную. Вы можете попросить Git отобразить в логе только такие коммиты, используя запись `master..experiment` — она означает «все коммиты, 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 коммитов их буквенные обозначения из диаграммы, располагая их в должном порядке:

```
$ git log master..experiment
D
C
```

С другой стороны, если вы хотите наоборот увидеть все коммиты ветки `master`, которых нет в ветке `experiment`, вы можете поменять имена веток в команде. При использовании записи `experiment..master` будут отображены все коммиты 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 неё слить. Другое частое использование такого синтаксиса — просмотр того, что будет отправлено в удалённый репозиторий.

```
$ git log origin/master..HEAD
```

Такая команда покажет вам все коммиты вашей текущей ветки, которые отсутствуют в ветке `master` удалённого репозитория `origin`. Если вы выполните `git push`, находясь на ветке, отслеживающей `origin/master`, то коммиты, отображённые командой `git log origin/master..HEAD`, будут теми коммитами, которые отправятся на сервер. Вы также можете опустить одну из частей в такой записи, Git будет считать её 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.

Множественная выборка

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 коммиты 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 коммиты 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 коммиты, 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 коммиты, доступные хотя бы из одной ссылки, но не из обеих сразу. Вспомните пример истории коммитов в Пример истории для выбора диапазонов коммитов. Если вы хотите узнать какие коммиты 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 коммитов, но в нём содержится информация только об этих четырёх коммитах, традиционно отсортированная по дате коммитов.

В таких случаях с командой `log` часто используют опцию `--left-right`, которая отображает сторону диапазона, с которой был сделан каждый из коммитов. 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 какой коммит или коммиты 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

определённый 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 разделённые изменения и могут быть легко прорецензированы вашими коллегами.

Если вы выполните `git add` с опцией `-i` или `--interactive`, Git перейдёт 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: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
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 введёте `2` или `u` в поле ввода `What now>`, скрипт спросит у вас какие файлы вы хотите добавить в индекс:

```
What now> u
           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 нажмёте Enter, не вводя ничего в поле ввода `Update>>`, Git добавит в индекс всё, чтобы было выбрано ранее:

```
Update>>
updated 2 paths

*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> s
           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 ещё нет. Если вы в этот момент хотите исключить файл TODO из индекса, вы можете использовать опции `3` или `r` (для выполнения revert):

```
*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> r
           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 вашего рабочего каталога Git, вы увидите, что файл `TODO` исключён из индекса:

```
*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> s
```

```
            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 `6` и `d` (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: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> d
            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 этой точке у вас есть множество вариантов дальнейших действий. Если вы введёте `?`, Git отобразит, что именно вы можете сделать:

```
Добавить в индекс эту часть [y,n,a,d,/,j,J,g,e,?]? ?
y - добавить в индекс эту часть
n - не добавлять в индекс эту часть
a - добавить в индекс эту и все оставшиеся в этом файле части
d - не добавлять в индекс эту и все оставшиеся в этом файле части
g - перейти к некоторой части файла (g - показывает список частей и затем выполняет
переход, g<N> - перейти к части N)
/ - найти часть, соответствующую регулярному выражению
j - отложить принятие решения по этой части, перейти к следующей части, решение по
которой не принято
J - отложить принятие решения по этой части, перейти к следующей части
k - отложить принятие решения по этой части, перейти к предыдущей части, решение по
которой не принято
K - отложить принятие решения по этой части, перейти к предыдущей части
s - разбить текущую часть на части меньшего размера
e - вручную отредактировать текущую часть
? - отобразить помощь
```

Обычно вы будете вводить `y` или `n`, если вы хотите индексировать каждую часть по отдельности, но индексация всех частей в некоторых файлах или откладывание решения по индексацию части также может быть полезным. Если вы добавили в индекс одну часть файла, но не добавили другую, состояние вашего рабочего каталога будет подобно приведённому 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 добавлена 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 её, используя команды `git add -p` 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 припрятывания частей файлов с помощью `stash save --patch`. Мы рассмотрим каждую из этих команд более подробно, когда будем изучать более продвинутые варианты их использования.

Припрятывание 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 ещё. 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` берёт изменённое состояние вашего рабочего каталога, то есть изменённые отслеживаемые файлы и проиндексированные изменения, и сохраняет их в хранилище незавершённых изменений, которые вы можете в любое время применить обратно.

Переход на `git stash push`

В конце октября 2017 года в списке рассылки Git проходило обширное обсуждение, по итогам которого команда `git stash save` признана устаревшей в пользу существующей альтернативы `git stash push`. Основная причина этого заключается в том, что в `git stash push` есть возможность сохранить выбранные *спецификации пути*, что не поддерживает `git stash save`.

Команда `git stash save` не исчезнет в ближайшее время, поэтому не беспокойтесь о её внезапной пропаже. Но вы можете начать переход на `push` для использования новой функциональности.

Припрятывание 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 изменённое 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 припрячете эти изменения. Для того, чтобы припрятать изменение в выделенное для этого специальное хранилище, выполните `git stash` или `git stash push`:

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 Create index file"
HEAD is now at 049d078 Create index file
(To restore them type "git stash apply")
```

Теперь вы можете увидеть, что рабочая копия 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 припрятанных изменений, вы можете использовать `git stash list`:

```
$ git stash list
stash@{0}: WIP on master: 049d078 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
stash@{2}: WIP on master: 21d80a5 Add number to log
```

В данном примере, предварительно были припрятаны два изменения, поэтому теперь вам доступны три различных отложенных наработки. Вы можете применить только что припрятанные изменения, используя команду, указанную в выводе исходной команды: `git stash apply`. Если вы хотите применить одно из предыдущих припрятанных 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 припрятанное изменение:

```
$ git stash apply
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:   index.html
    modified:   lib/simplegit.rb
```

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

Как видите, Git восстановил в файлах изменения, которые вы отменили ранее, когда прятали свои наработки. В данном случае при применении отложенных наработок ваш рабочий каталог был без изменений, а вы пытались применить их в той же ветке, в которой вы их и сохранили; но отсутствие изменений в рабочем каталоге и применение их в той же ветке не являются необходимыми условиями для успешного восстановления припрятанных наработок. Вы можете припрятать изменения, находясь в одной ветке, а затем переключиться на другую и попробовать восстановить эти изменения. Также при восстановлении припрятанных наработок в вашем рабочем каталоге могут присутствовать изменённые 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

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
```

Команда `apply` только пытается восстановить припрятанные наработки — при этом они останутся 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 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
stash@{2}: WIP on master: 21d80a5 Add number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Вы также можете выполнить `git stash pop`, чтобы применить припрятанные изменения и тут же удалить их из хранилища.

Необычное припрятывание

У припрятанных изменений есть несколько дополнительных вариантов использования, которые также могут быть полезны. Первый — это использование довольно популярной опции `--keep-index` с командой `git stash`. Она просит Git не только припрятать то, что вы уже добавили в индекс, но одновременно оставить это в индексе.

```
$ 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 распространённый вариант, который вы, возможно, захотите использовать — это припрятать помимо отслеживаемых файлов также и неотслеживаемые. По умолчанию `git stash` будет сохранять только изменённые и проиндексированные *отслеживаемые* файлы. Если вы укажете опцию `--include-untracked` или `-u`, Git также припрячет все неотслеживаемые файлы, которые вы создали. Однако включение этой опции по-прежнему не будет прятать файлы с явным игнорированием; чтобы дополнительно припрятать игнорируемые файлы, используйте `--all` (или просто `-a`).

```
$ 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 режиме спросит вас о том, какие из изменений вы хотите припрятать, а какие оставить в вашем рабочем каталоге.

```
$ 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 file
```

Создание ветки из припрятанных 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 изменён с момента сохранения наработок, то вы получите конфликт слияния и должны будете попытаться разрешить его. Если вам нужен более простой способ снова протестировать припрятанные изменения, вы можете выполнить команду `git stash branch`, которая создаст для вас новую ветку, перейдёт на коммит, на котором вы были, когда прятали свои наработки, применит на нём эти наработки и затем, если они применились успешно, удалит эти припрятанные изменения:

```
$ git stash branch testchanges
M   index.html
M   lib/simplegit.rb
Switched to a new branch 'testchanges'
On branch testchanges
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

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

Это удобное сокращение для того, чтобы легко восстановить припрятанные изменения и поработать над ними в новой ветке.

Очистка рабочего каталога

Наконец, у вас может возникнуть желание не прятать некоторые из изменений или файлов в вашем рабочем каталоге, а просто избавиться от них. Команда `git clean` сделает это для вас.

Одной из распространённых причин для этого может быть удаление мусора, который был сгенерирован при слиянии или внешними утилитами, или удаление артефактов сборки в процессе её очистки.

Вам нужно быть очень аккуратными с этой командой, так как она предназначена для удаления неотслеживаемых файлов из вашего рабочего каталога. Даже если вы передумаете, очень часто нельзя восстановить содержимое таких файлов. Более безопасным вариантом является использование команды `git stash --all` для удаления всего, но с сохранением этого в виде припрятанных изменений.

Предположим, вы хотите удалить мусор и очистить ваш рабочий каталог; вы можете сделать это с помощью `git clean`. Для удаления всех неотслеживаемых файлов в вашем рабочем каталоге, вы можете выполнить команду `git clean -f -d`, которая удалит все файлы и также все каталоги, которые в результате станут пустыми. Параметр `-f` (сокращение от слова force — заставить) означает принудительное удаление, подчёркивая, что вы действительно хотите это сделать, и требуется, если переменная конфигурации Git `clean.requireForce` явным образом не установлена в `false`.

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, которые 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 её 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             5: quit
    6: help
What now>
```

Таким образом, вы можете просмотреть каждый файл индивидуально или указать шаблоны для удаления в интерактивном режиме.

Существует причудливая ситуация, когда вам, возможно, придётся проявить особую настойчивость, попросив Git очистить ваш рабочий каталог. Если вы оказались в рабочем каталоге, в который вы скопировали или клонировали другие репозитории Git (возможно, в виде подмодулей), даже `git clean -fd` откажется удалить эти каталоги. В таких случаях вам нужно добавить второй параметр `-f` для акцентирования.

Подпись

Благодаря шифрованию система Git является безопасной, но полностью она не защищена. На случай, если вы берёте 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 прикреплённую 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
ecorc4iXzQu7tupRihslbNkfvfciMnSDeSvzCpWAHl7h8Wj6hhqePmLm9lAYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1PblGfHR4XAhuOqN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----
```

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

    Change 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 параметр `--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  Change the version number
085bb3b N Scott Chacon  Remove unnecessary test code
a11bef0 N Scott Chacon  Initial 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 сливаемая 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 сливаемой 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 придётся 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.

Команда `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. В качестве первого варианта вы можете использовать любой из параметров `-n` или `--line-number`, чтобы распечатать номера строк, в которых Git нашёл совпадения:

```
$ 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:826:struct tm *gmtime_r(const time_t *timep, struct tm *result)
compat/mingw.h:206:struct tm *gmtime_r(const time_t *timep, struct tm *result);
date.c:482:             if (gmtime_r(&now, &now_tm))
date.c:545:             if (gmtime_r(&time, tm)) {
date.c:758:             /* gmtime_r() in match_digit() may have clobbered it */
git-compat-util.h:1138:struct tm *git_gmtime_r(const time_t *, struct tm *);
git-compat-util.h:1140:#define gmtime_r git_gmtime_r
```

В дополнение к базовому поиску, показанному выше, `git grep` поддерживает множество других интересных параметров.

Например, вместо того, чтобы печатать все совпадения, вы можете попросить `git grep` обобщить выводимые командой данные, показав только те файлы, в которых обнаружены совпадения, вместе с количеством этих совпадений в каждом файле. Для этого потребуется параметр `-c` или `--count`:

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:3
git-compat-util.h:2
```

Если вас интересует *контекст* строки поиска, можно показать метод или функцию, в котором присутствует совпадение с помощью параметра `-p` или `--show-function`:

```
$ git grep -p gmtime_r *.c
date.c=static int match_multi_number(timestamp_t 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)) {
date.c=int parse_date_basic(const char *date, timestamp_t *timestamp, int *offset)
date.c:         /* gmtime_r() in match_digit() may have clobbered it */
```

Здесь вы можете видеть, что `gmtime_r` вызывается из функций `match_multi_number` и `match_digit` в файле `date.c` (третье отображаемое совпадение представляет собой только строку, появившуюся в комментарии).

Вы также можете искать сложные комбинации строк, используя опцию `--and`, которая гарантирует, что будут отображены только строки, имеющие сразу несколько совпадений. Например, давайте поищем любые строки, которые определяют константу, имя которой содержит *любую* из подстрок «LINK» или «BUF_MAX», особенно в более старой версии

кодовой базы Git, представленной тегом v1.8.0 (мы добавим параметры `--break` и `--heading`, которые помогут 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_CREATION_USES_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 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 определённых 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 -S ZLIB_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 изменений строки

Другой, довольно продвинутый, поиск по истории, который бывает чрезвычайно полезным — поиск по истории изменений строки. Просто запустите `git log` с параметром `-L`, и он покажет вам историю 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 определённой строки и в этом случае вы получите похожий результат.

Перезапись истории

Неоднократно при работе с Git, вам может потребоваться по какой-то причине внести исправления в историю коммитов. Одно из преимуществ Git заключается в том, что он позволяет вам отложить принятие решений на самый последний момент. Область индексирования позволяет вам решить, какие файлы попадут в коммит непосредственно перед его выполнением; благодаря команде `git stash` вы можете решить, что не хотите продолжать работу над какими-то изменениями; также можете внести изменения в сделанные коммиты так, чтобы они выглядели как будто они произошли по-другому. В частности, можно изменить порядок коммитов, сообщения или изменённые в коммитах файлы, объединить вместе или разбить на части, полностью удалить коммит — но только до того, как вы поделитесь своими наработками с другими.

В этом разделе вы познакомитесь со способами решения всех этих задач и научитесь перед публикацией данных приводить историю коммитов в нужный вам вид.

Не отправляйте свои наработки, пока вы ими не довольны

Одно из основных правил Git заключается в том, что, так как большую часть работы вы делаете в своём локальном репозитории, то вы вольны переписывать свою историю *локально*. Однако, как только вы отправите свои наработки, то это уже совсем другая история и вам следует рассматривать отправленные изменения как финальные до тех пор, пока у вас не появится весомая причина что-то изменить. Если кратко, то вы должны воздержаться от отправки своих изменений, пока не будете полностью довольны и готовы поделиться ими со всем миром.

Изменение последнего коммита

Изменение вашего последнего коммита, наверное, наиболее частое исправление истории, которое вы будете выполнять. Наиболее часто с вашим последним коммитом вам будет нужно сделать две основные операции: изменить сообщение коммита или изменить только что сделанный снимок, добавив, изменив или удалив файлы.

Если вы хотите изменить только сообщение вашего последнего коммита, это очень просто:

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

Эта команда откроет в вашем текстовом редакторе сообщение вашего последнего коммита, для того, чтобы вы могли его исправить. Когда вы сохраните его и закроете редактор, будет создан новый коммит, содержащий это сообщение, который теперь и будет вашим последним коммитом.

Если вы создали коммит и затем хотите изменить зафиксированный снимок, добавив или изменив файлы (возможно, вы забыли добавить вновь созданный файл, когда совершали изначальный коммит), 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` берёт вашу текущую область подготовленных изменений и делает её снимок для нового коммита.

Вы должны быть осторожными, используя этот приём, так как при этом изменяется SHA-1 коммита. Поэтому, как и с операцией `rebase` — не изменяйте ваш последний коммит, если вы уже отправили его в общий репозиторий.

Изменённый коммит может потребовать изменения сообщения коммита

При изменении коммита существует возможность изменить как его содержимое, так и сообщение коммита. Если в коммит внесены существенные изменения, то почти наверняка следует обновить и его сообщение, чтобы оно более точно отражало содержимое коммита.

С другой стороны, если изменения незначительны (исправление опечаток, добавление в коммит забытого файла), то текущее сообщение вполне можно оставить; чтобы лишний раз не вызывать редактор, просто добавьте изменённые файлы в индекс и выполните команду:

```
$ git commit --amend --no-edit
```

Изменение сообщений нескольких коммитов

Для изменения коммита, расположенного раньше в вашей истории, вам нужно обратиться к более сложным инструментам. В Git отсутствуют инструменты для переписывания истории, но вы можете использовать команду `rebase`, чтобы перебазировать группу коммитов туда же на HEAD, где они были изначально, вместо перемещения их в другое место. С помощью интерактивного режима команды `rebase`, вы можете останавливаться после каждого нужного вам коммита 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 коммиты вы хотите изменить, передав команде коммит, на который нужно выполнить перебазирование.

Например, если вы хотите изменить сообщения последних трёх коммитов, или сообщение какого-то одного коммита этой группы, то передайте как аргумент команде `git rebase -i` родителя последнего коммита, который вы хотите изменить — `HEAD~2^` или `HEAD~3`. Может быть, проще будет запомнить `~3`, так как вы хотите изменить последние три коммита; но не забывайте, что вы, в действительности, указываете четвертый коммит с конца — родителя последнего коммита, который вы хотите изменить:

```
$ git rebase -i HEAD~3
```

Напомним, что это команда перебазирования — каждый коммит, входящий в диапазон

`HEAD~3..HEAD`, будет изменён вне зависимости от того, изменили вы сообщение или нет. Не включайте в такой диапазон коммит, который уже был отправлен 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 коммитов, в нашем случае, например, следующее:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# 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 коммиты 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 Add cat-file
310154e Update README formatting and add blame
f7f3f6d Change my name a bit
```

Обратите внимание на обратный порядок. Команда `rebase` в интерактивном режиме

предоставит вам скрипт, который она будет выполнять. Она начнёт с коммита, который вы указали в командной строке (`HEAD~3`) и повторит изменения, внесённые каждым из коммитов, сверху вниз. Наверху отображается самый старый коммит, а не самый новый, потому что он будет повторен первым.

Вам необходимо изменить скрипт так, чтобы он остановился на коммите, который вы хотите изменить. Для этого измените слово `pick` на слово `edit` напротив каждого из коммитов, после которых скрипт должен остановиться. Например, для изменения сообщения только третьего коммита, измените файл следующим образом:

```
edit f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

Когда вы сохраните сообщение и выйдете из редактора, Git переместит вас к самому раннему коммиту из списка и вернёт вас в командную строку со следующим сообщением:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... Change 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. Выполните:

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

Измените сообщение коммита и выйдите из редактора. Затем выполните:

```
$ git rebase --continue
```

Эта команда автоматически применит два оставшиеся коммита и завершится. Если вы измените `pick` на `edit` в других строках, то можете повторить эти шаги для соответствующих коммитов. Каждый раз Git будет останавливаться, позволяя вам исправить коммит, и продолжит, когда вы закончите.

Упорядочивание коммитов

Вы также можете использовать интерактивное перебазирование для изменения порядка или полного удаления коммитов. Если вы хотите удалить коммит «Add cat-file» и изменить порядок, в котором были внесены два оставшихся, то вы можете изменить скрипт

перебазирования с такого:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

на такой:

```
pick 310154e Update README formatting and add blame
pick f7f3f6d Change my name a bit
```

Когда вы сохраните скрипт и выйдете из редактора, Git переместит вашу ветку на родителя этих коммитов, применит `310154e`, затем `f7f3f6d` и после этого остановится. Вы, фактически, изменили порядок этих коммитов и полностью удалили коммит «Add cat-file».

Объединение коммитов

С помощью интерактивного режима команды `rebase` также можно объединить несколько коммитов в один. Git добавляет полезные инструкции в сообщение скрипта перебазирования:

```
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# 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 текущего и

предыдущего коммитов и предложит вам объединить их сообщения. Таким образом, если вы хотите из этих трёх коммитов сделать один, вы должны изменить скрипт следующим образом:

```
pick f7f3f6d Change my name a bit
squash 310154e Update README formatting and add blame
squash a5f4a0d Add cat-file
```

Когда вы сохраните скрипт и выйдете из редактора, Git применит изменения всех трёх коммитов и затем вернёт вас обратно в редактор, чтобы вы могли объединить сообщения коммитов:

```
# This is a combination of 3 commits.
# The first commit's message is:
Change my name a bit

# This is the 2nd commit message:

Update README formatting and add blame

# This is the 3rd commit message:

Add cat-file
```

После сохранения сообщения, вы получите один коммит, содержащий изменения всех трёх коммитов, существовавших ранее.

Разбиение коммита

Разбиение коммита отменяет его и позволяет затем по частям индексировать и фиксировать изменения, создавая таким образом столько коммитов, сколько вам нужно. Например, предположим, что вы хотите разбить средний коммит на два. Вместо одного коммита «Update README formatting and add blame» вы хотите получить два разных: первый — «Update README formatting», и второй — «Add blame». Вы можете добиться этого, изменив в скрипте `rebase -i` инструкцию для разбиваемого коммита на «edit»:

```
pick f7f3f6d Change my name a bit
edit 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

Затем, когда скрипт вернёт вас в командную строку, вам нужно будет отменить индексацию изменений этого коммита, и создать несколько коммитов на основе этих изменений. Когда вы сохраните скрипт и выйдете из редактора, Git переместится на родителя первого коммита в вашем списке, применит первый коммит (`f7f3f6d`), применит второй (`310154e`), и вернёт вас в консоль. Здесь вы можете отменить коммит с помощью команды `git reset HEAD^`, которая, фактически, отменит этот коммит и удалит из индекса изменённые файлы.

Теперь вы можете добавлять в индекс и фиксировать файлы, пока не создадите требуемые коммиты, а после этого выполнить команду `git rebase --continue`:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'Update README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'Add blame'
$ git rebase --continue
```

Git применит последний коммит (`a5f4a0d`) из скрипта, и ваша история примет следующий вид:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd Add cat-file
9b29157 Add blame
35cfb2b Update README formatting
f7f3f6d Change my name a bit
```

И снова, при этом изменились SHA-1 хеши всех коммитов в вашем списке, поэтому убедитесь, что ни один коммит из этого списка ранее не был отправлен в общий репозиторий. Обратите внимание, что последний коммит в списке (`f7f3f6d`) не изменился. Несмотря на то, что коммит был в списке перебазирования, он был отмечен как «pick» и применён до применения перебазирования, поэтому Git оставил его нетронутым.

Удаление коммита

Если вы хотите избавиться от какого-либо коммита, то удалить его можно во время интерактивного перебазирования `rebase -i`. Напишите слово «drop» перед коммитом, который хотите удалить, или просто удалите его из списка:

```
pick 461cb2a This commit is OK
drop 5aecc10 This commit is broken
```

Из-за того, как Git создаёт объекты коммитов, удаление или изменение коммита влечёт за собой перезапись всех последующих коммитов. Чем дальше вы вернётесь в историю ваших коммитов, тем больше коммитов потребуется переделать. Это может вызвать множество конфликтов слияния, особенно если у вас много последующих коммитов, которые зависят от удалённого.

Если во время подобного перебазирования вы поняли, что это была не очень хорошая идея, то всегда можно остановиться. Просто выполните команду `git rebase --abort` и ваш репозиторий вернётся в то состояние, в котором он был до начала перебазирования.

Если вы завершили перебазирование, а затем решили, что полученный результат это не то, что вам нужно — воспользуйтесь командой `git reflog`, чтобы восстановить предыдущую

версию вашей ветки. Дополнительную информацию по команде `reflog` можно найти в разделе Восстановление данных главы 10.

Дрю Дево создал практическое руководство с упражнениями по использованию `git rebase`. Найти его можно здесь: https://git-rebase.io/

Продвинутый инструмент: filter-branch

Существует ещё один способ переписывания истории, который вы можете использовать при необходимости изменить большое количество коммитов каким-то программируемым способом — например, изменить глобально ваш адрес электронной почты или удалить файл из всех коммитов. Для этого существует команда `filter-branch`, и она может изменять большие периоды вашей истории, поэтому вы, возможно, не должны её использовать кроме тех случаев, когда ваш проект ещё не стал публичным и другие люди ещё не имеют наработок, основанных на коммитах, 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.

Команда `git filter-branch` таит в себе много подводных камней и более не является рекомендуемым способом изменения истории. Вместо этого, рассмотрите возможность использования Python скрипта `git-filter-repo`, который лучше подходит для большинства ситуаций, в которых вы обычно используете `filter-branch`. С документаций и исходным кодом скрипта можно ознакомиться здесь: https://github.com/newren/git-filter-repo.

Удаление файла из каждого коммита

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 каждый коммит 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 коммиты, 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 неё указателя основной ветки. Для выполнения `filter-branch` на всех ваших ветках, вы можете передать команде опцию `--all`.

Установка подкаталога как корневого каталога проекта

Предположим, вы выполнили импорт из другой системы контроля версий и получили в результате подкаталоги, которые не имеют никакого смысла (`trunk`, `tags` и так далее). Если вы хотите сделать подкаталог `trunk` корневым для каждого коммита, 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 вашим новым корневым каталогом проекта будет являться подкаталог `trunk`. Git также автоматически удалит коммиты, которые не затрагивали этот подкаталог.

Глобальное изменение адреса электронной почты

Ещё 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 электронный почты сразу в нескольких коммитах 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 пройдёт по всем коммитам и установит в них ваш новый адрес. Так как коммиты содержат значения SHA-1-хешей их родителей, эта команда изменяет хеш SHA-1 каждого коммита в вашей истории, а не только тех, которые соответствовали адресам электронной почты.

Раскрытие тайн reset

Перед тем, как перейти к более специализированными 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 безнадёжными. 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 трёх различных деревьев. Здесь под «деревом» мы понимаем «набор файлов», а не специальную структуру данных. (В некоторых случаях индекс ведёт 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

Указатель 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 приведён 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

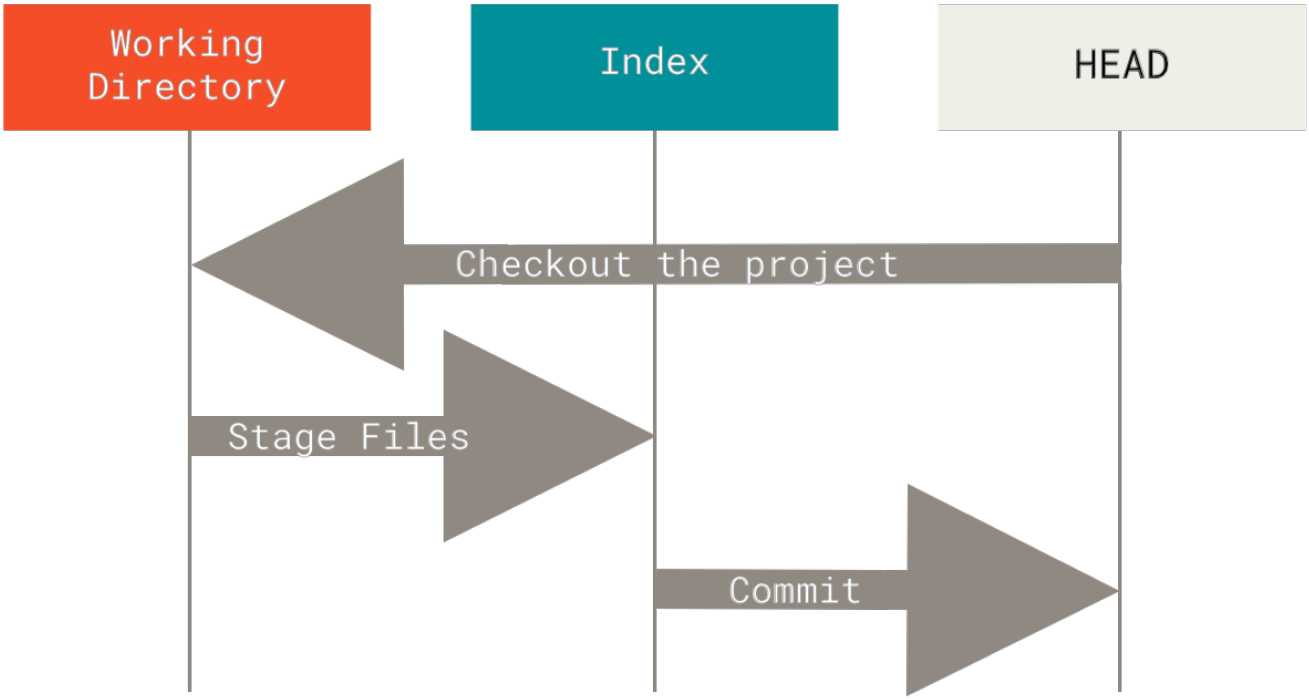
Наконец, у вас есть рабочий каталог. Два других дерева сохраняют своё содержимое эффективным, но неудобным способом внутри каталога `.git`. Рабочий Каталог распаковывает их в настоящие файлы, что упрощает для вас их редактирование. Считайте Рабочий Каталог **песочницей**, где вы можете опробовать изменения перед их коммитом в индекс (область подготовленных изменений) и затем в историю.

```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

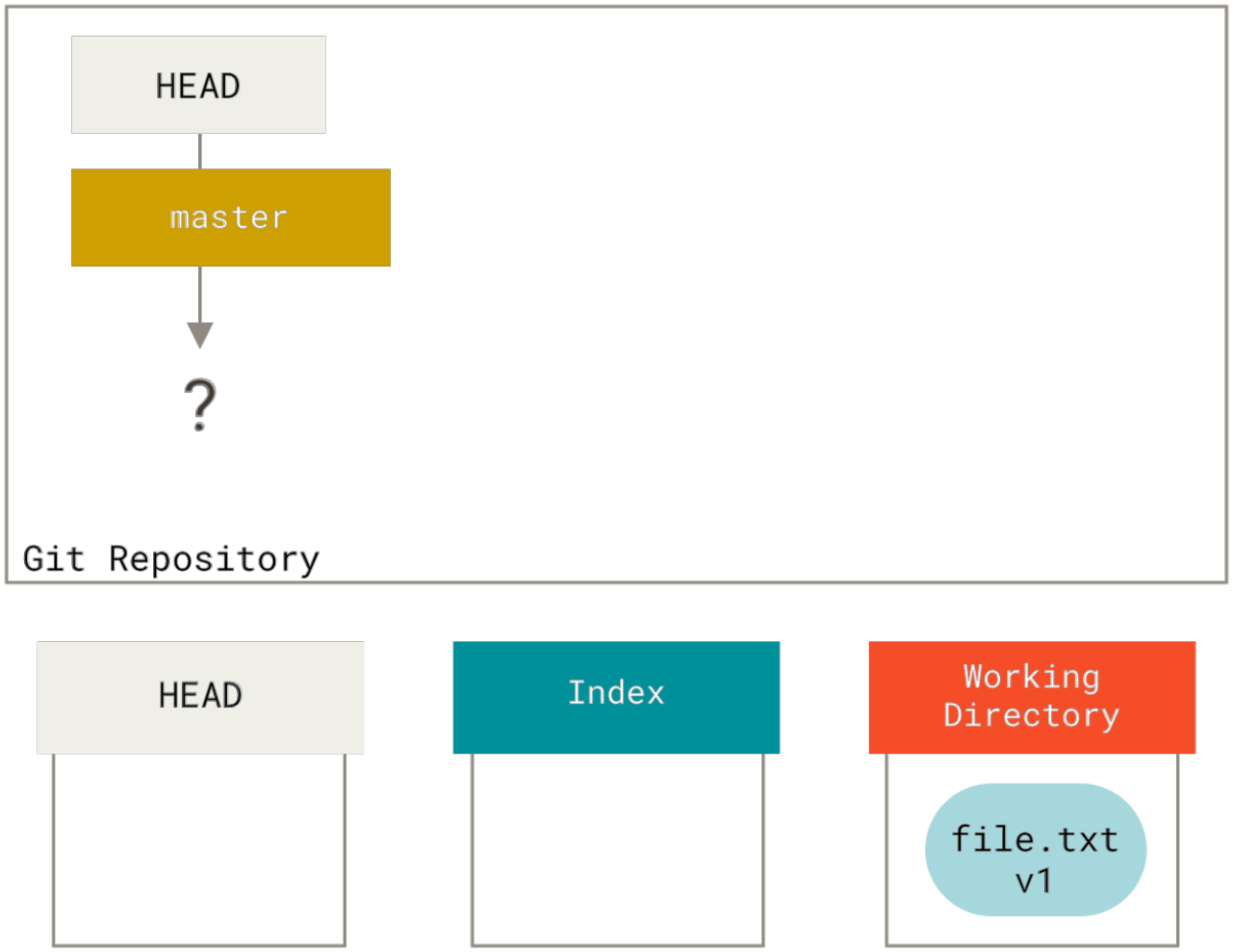
1 directory, 3 files
```

Технологический процесс

Основное предназначение Git — это сохранение снимков последовательно улучшающихся состояний вашего проекта, путём управления этими тремя деревьями.

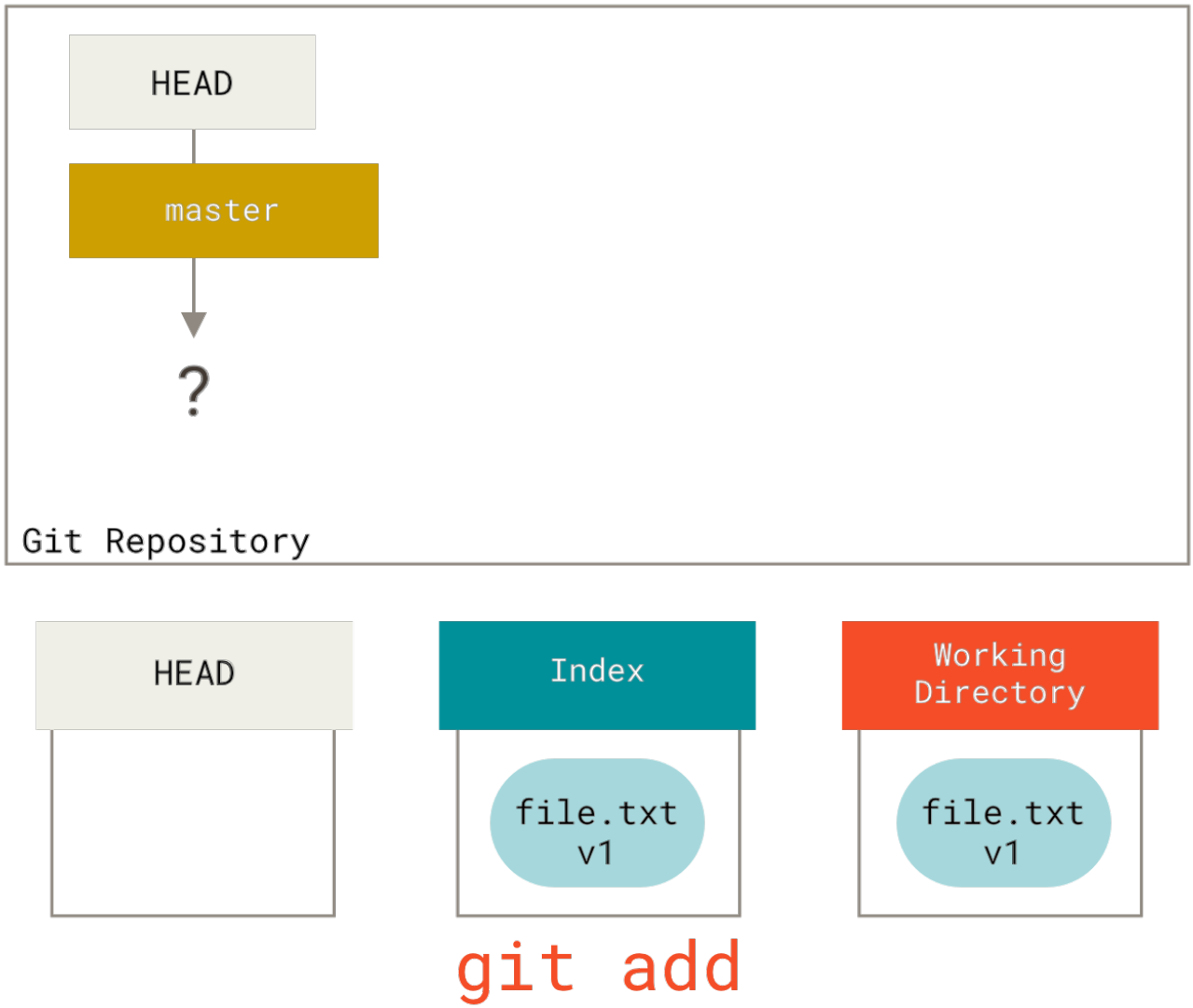


Давайте рассмотрим этот процесс: пусть вы перешли в новый каталог, содержащий один файл. Данную версию этого файла будем называть **v1** и изображать голубым цветом. Выполним команду `git init`, которая создаст Git-репозиторий, у которого ссылка HEAD будет указывать на ещё несуществующую ветку (`master` пока не существует).

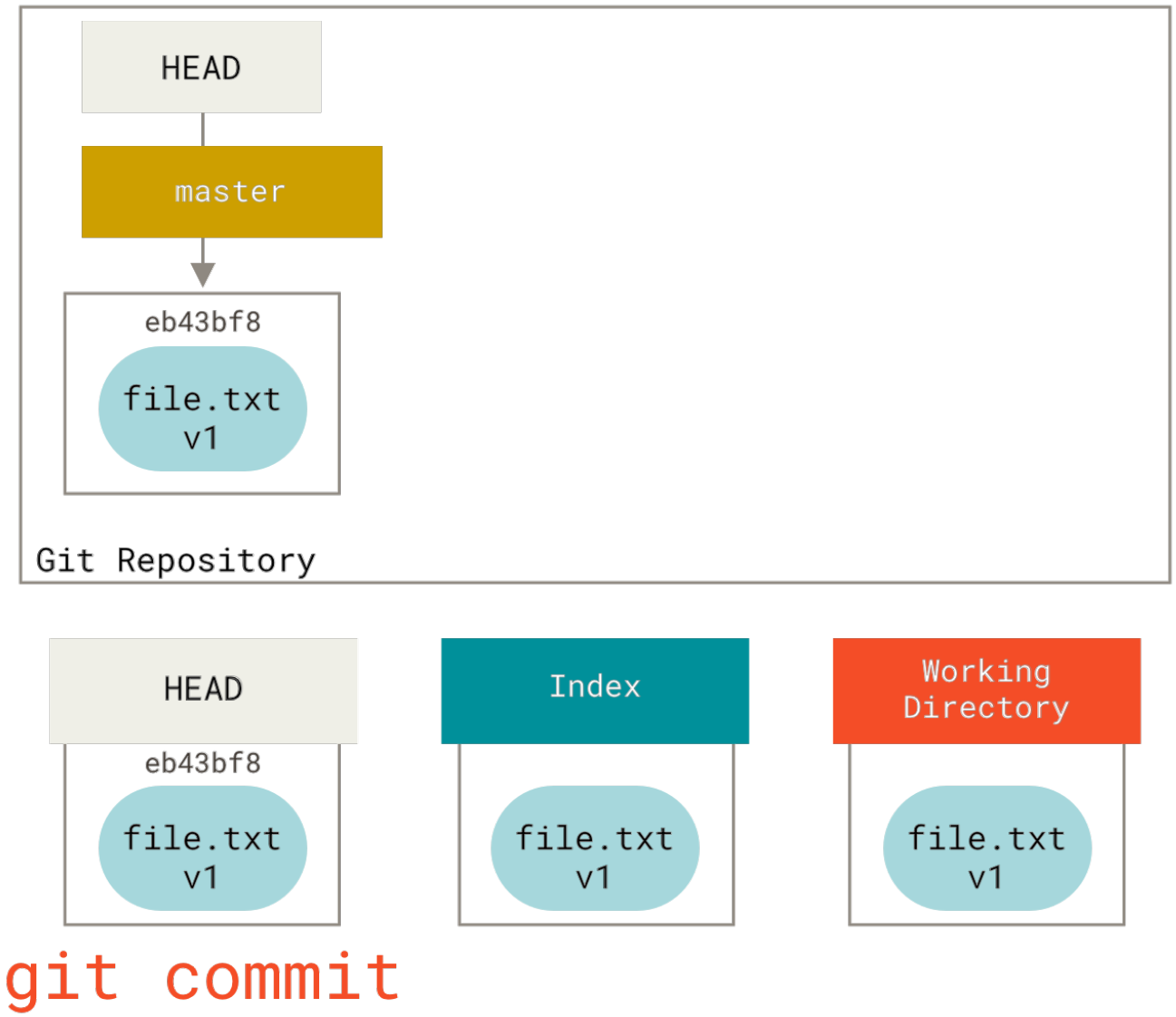


На данном этапе только дерево Рабочего Каталога содержит данные.

Теперь мы хотим закоммитить этот файл, поэтому мы используем `git add` для копирования содержимого Рабочего Каталога в Индекс.

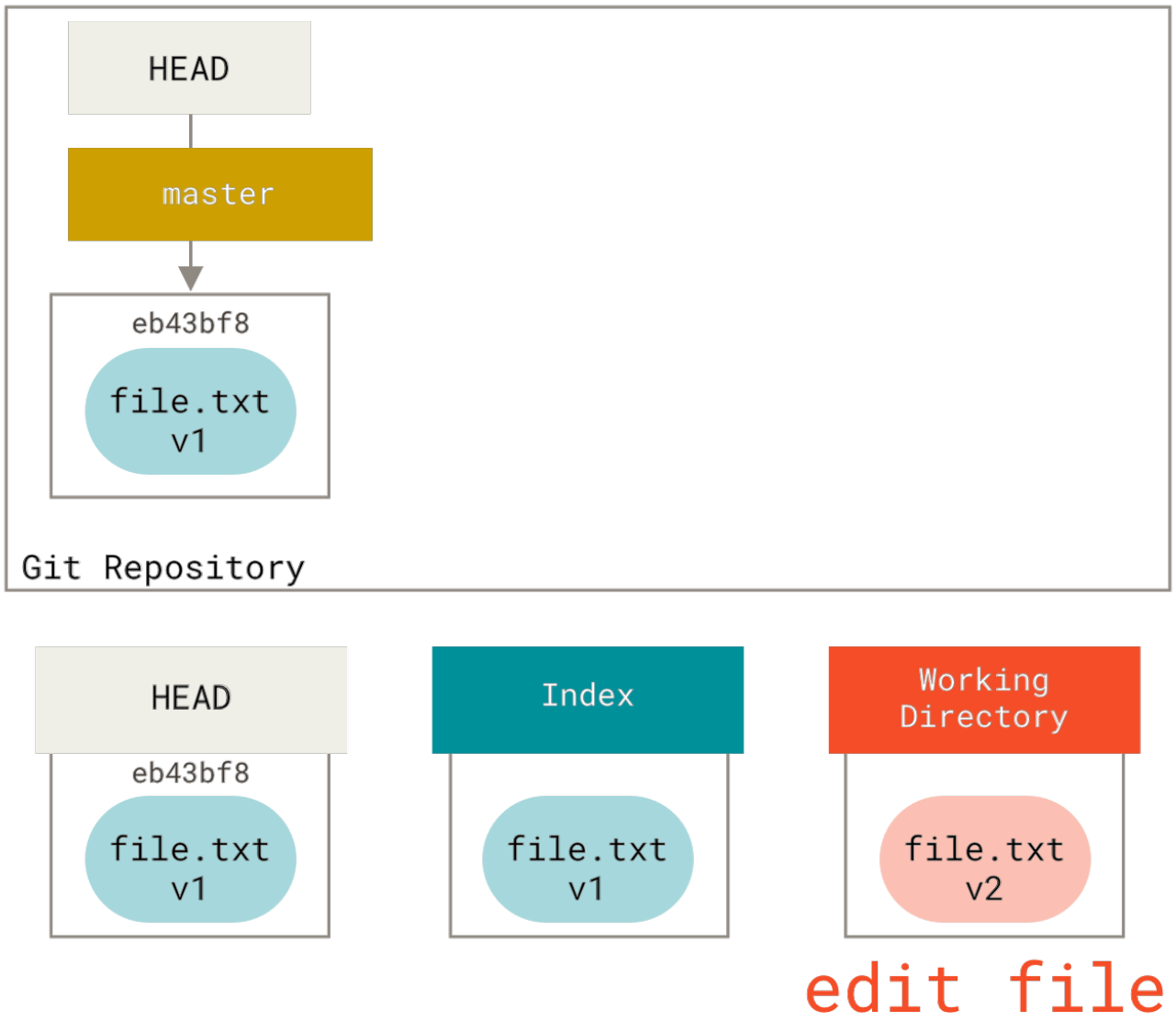


Затем, мы выполняем команду `git commit`, которая сохраняет содержимое Индекса как неизменяемый снимок, создаёт объект коммита, который указывает на этот снимок, и обновляет `master` так, чтобы он тоже указывал на этот коммит.

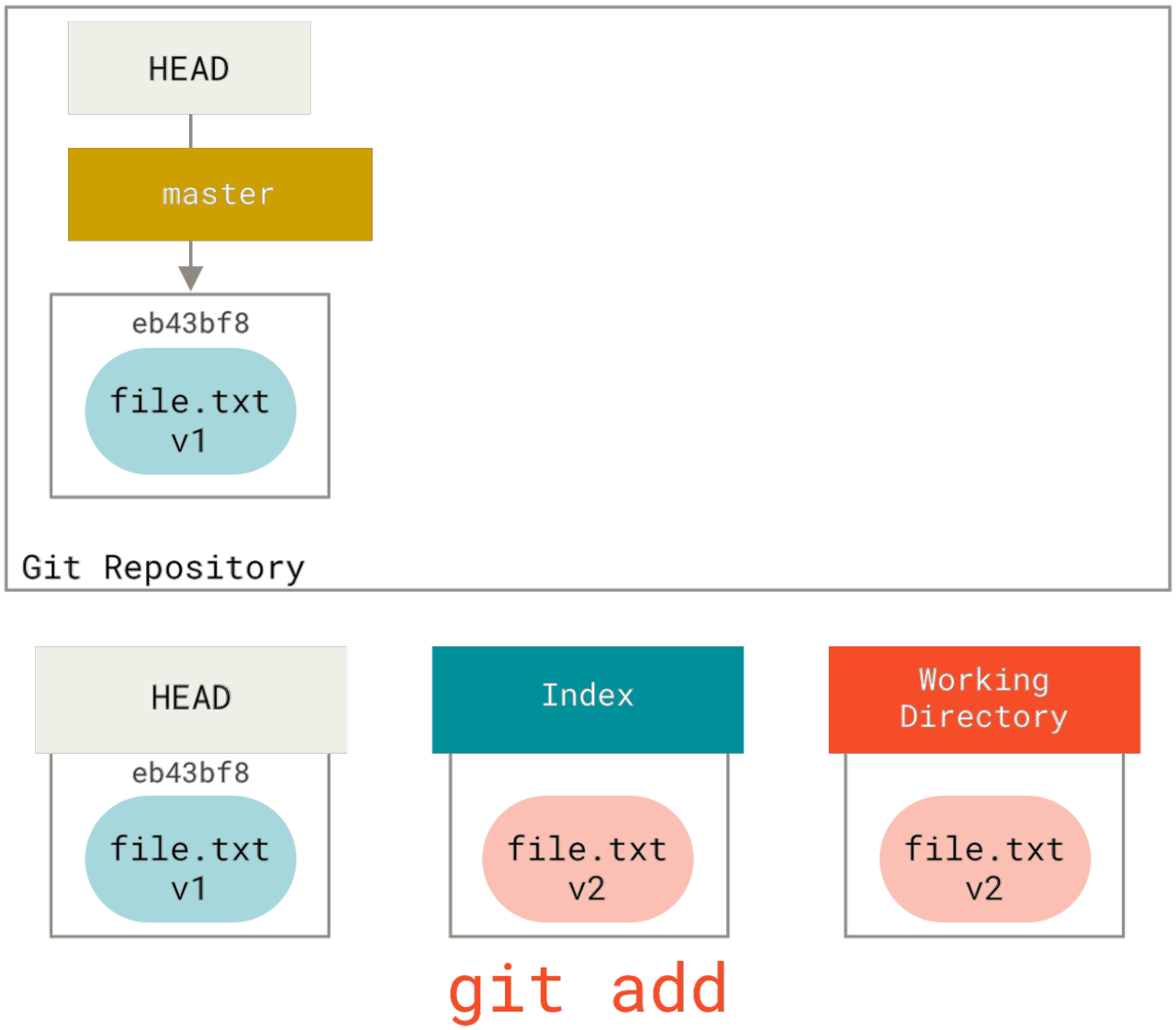


Если сейчас выполнить `git status`, то мы не увидим никаких изменений, так как все три дерева одинаковые.

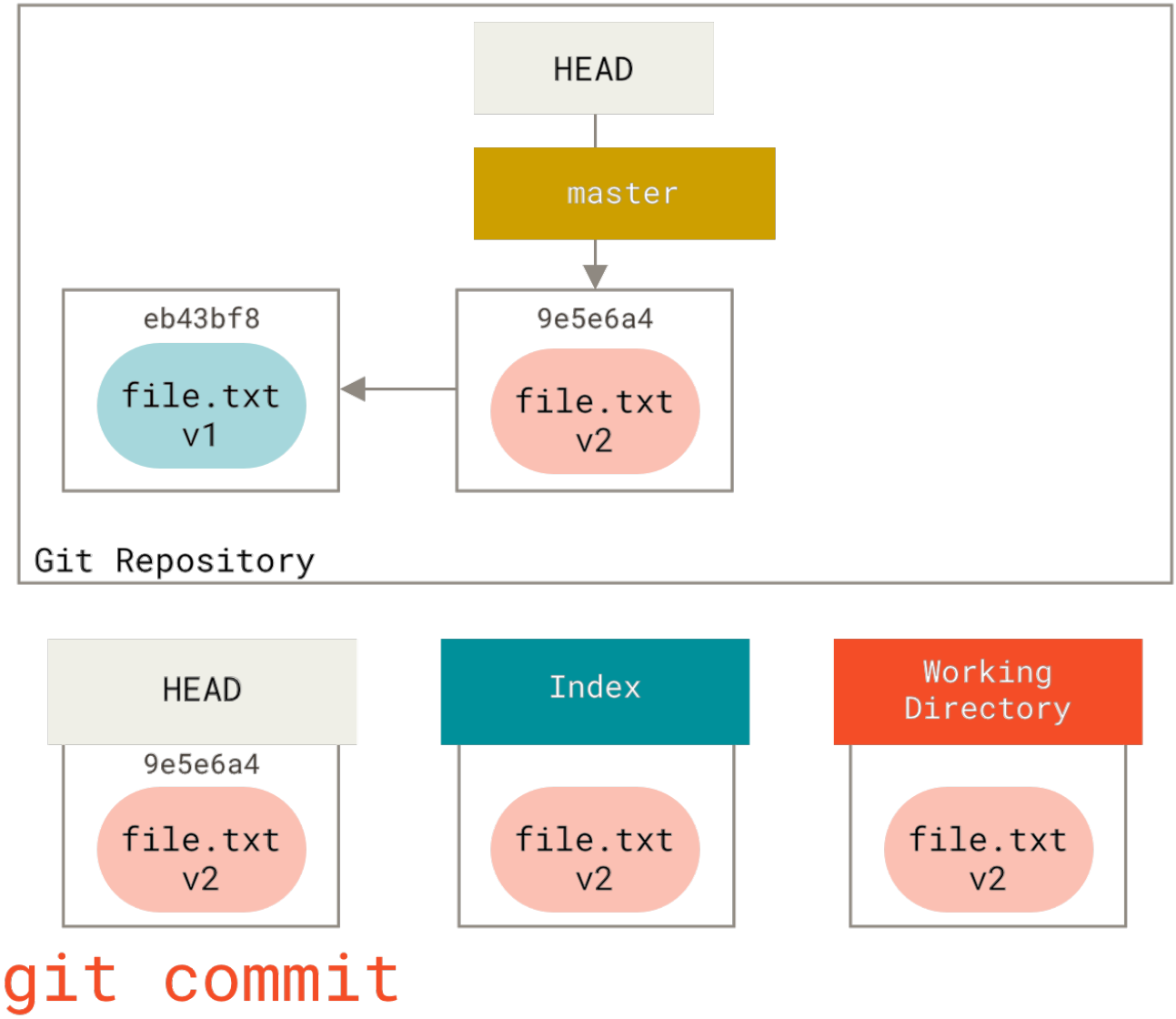
Теперь мы хотим внести изменения в файл и закоммитить его. Мы пройдём через всё ту же процедуру; сначала мы отредактируем файл в нашем рабочем каталоге. Давайте называть эту версию файла **v2** и обозначать красным цветом.



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.



Если сейчас мы выполним `git status`, то увидим, что этот файл выделен зелёным 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.



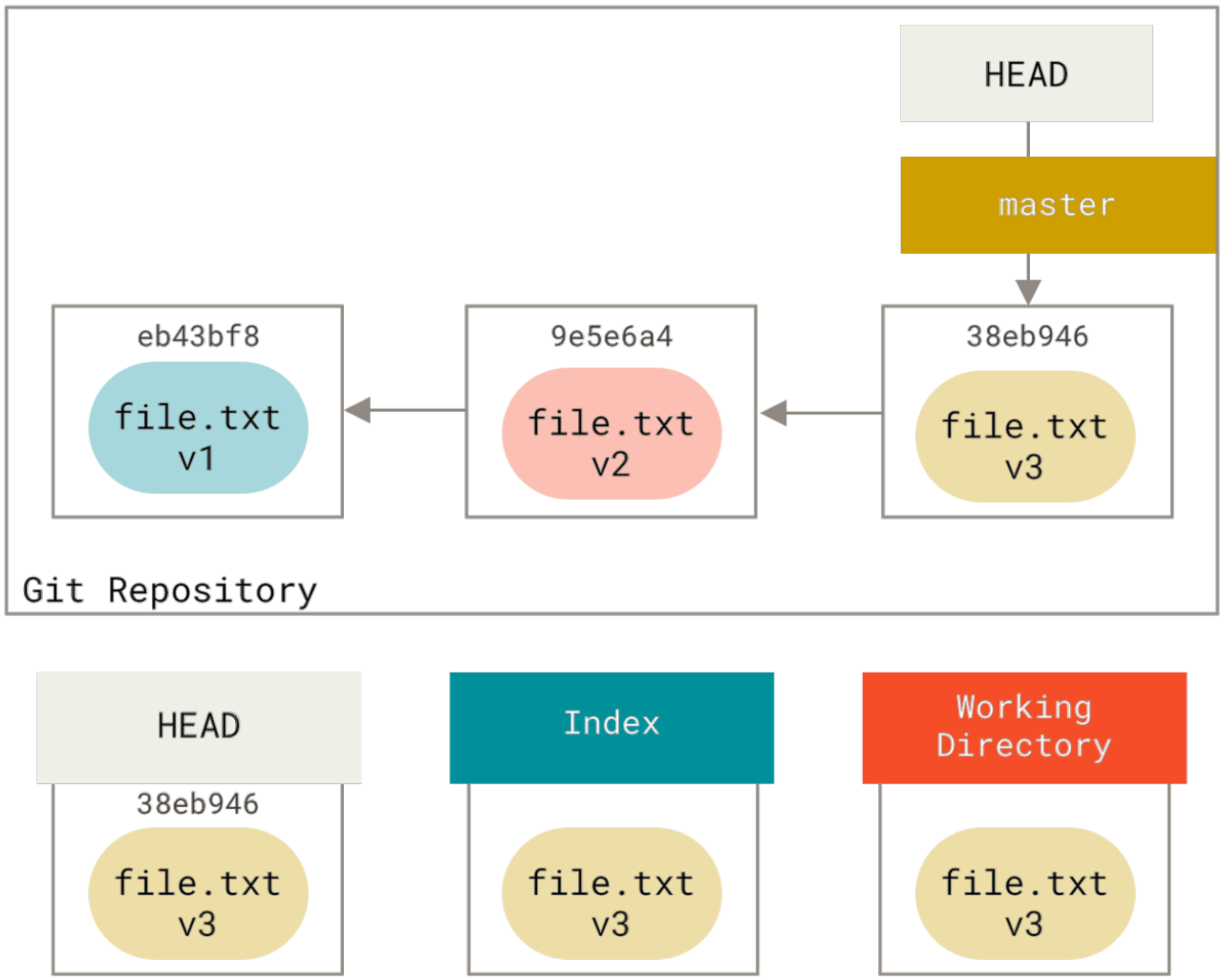
Сейчас команда `git status` не показывает ничего, так как снова все три дерева одинаковые.

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 команды `reset`

Команда `reset` становится более понятной, если рассмотреть её с учётом вышеизложенного.

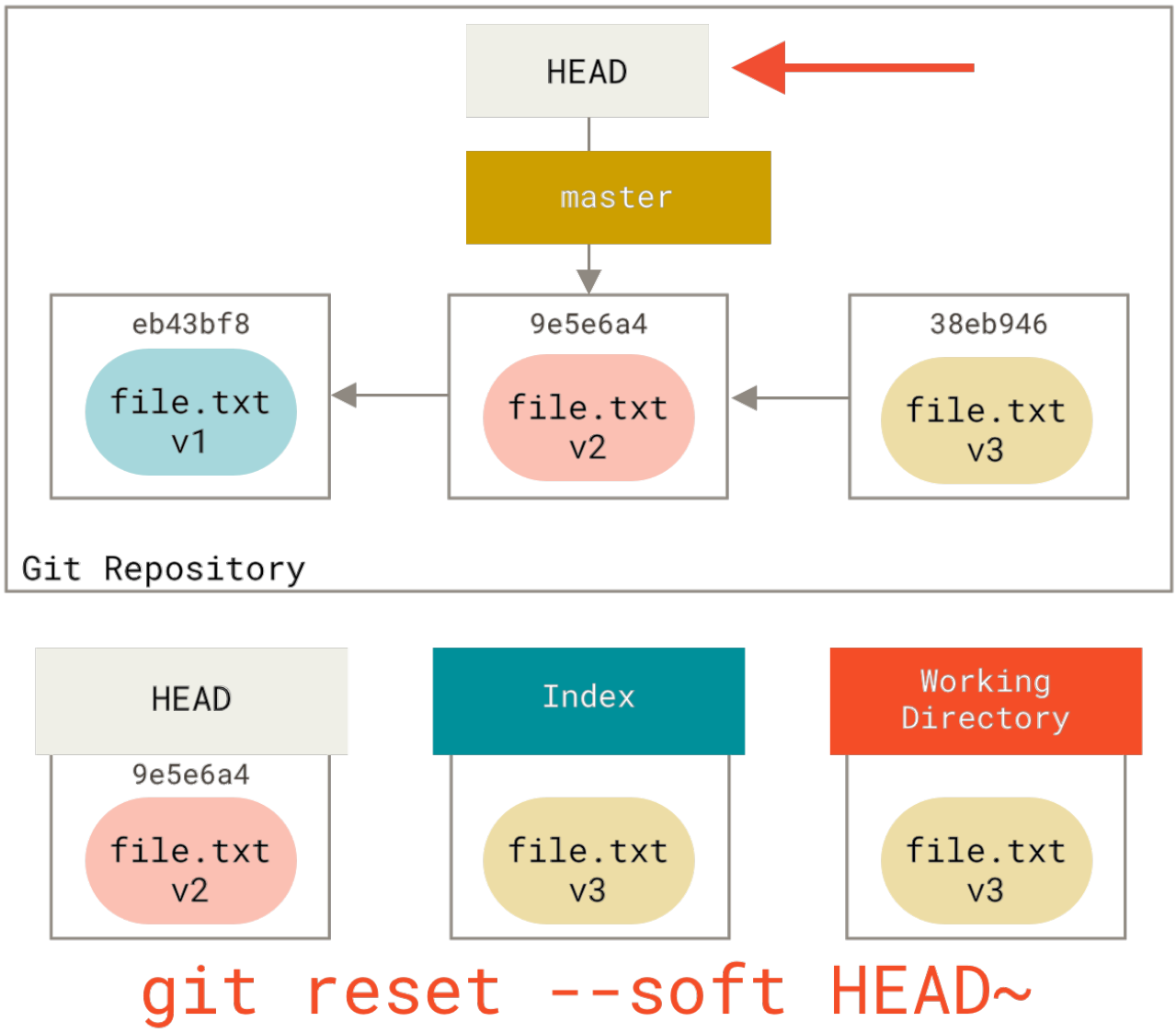
В следующих примерах предположим, что мы снова изменили файл `file.txt` и закоммитили его в третий раз. Так что наша история теперь выглядит так:



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 указателя 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`.



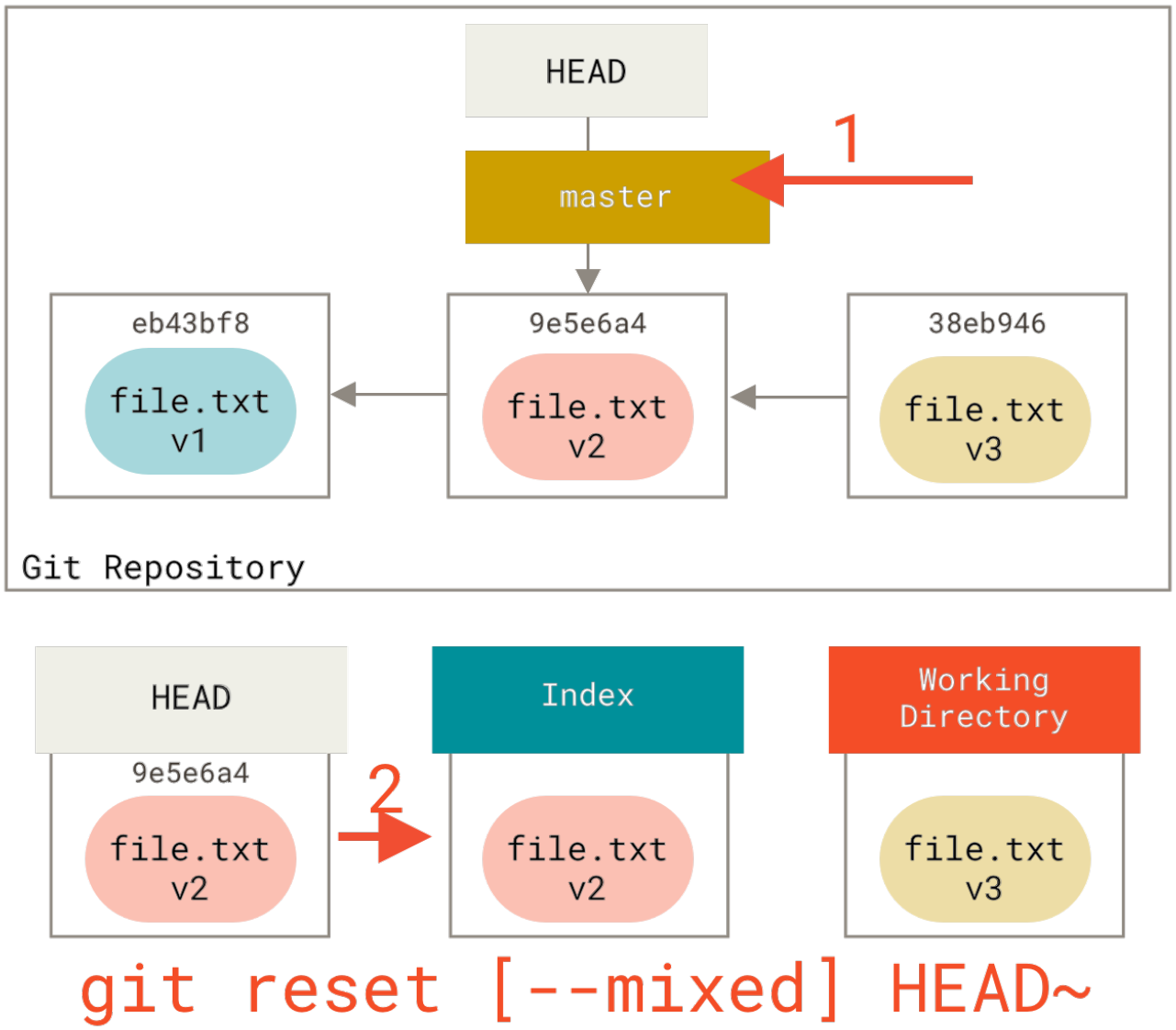
Не важно с какими опциями вы вызвали команду `reset` *с указанием коммита* (`reset` также можно вызывать с указанием пути), она всегда будет пытаться сперва сделать данный шаг. При вызове `reset --soft` на этом выполнение команды и остановится.

Теперь взгляните на диаграмму и постарайтесь разобраться, что случилось: фактически была отменена последняя команда `git commit`. Когда вы выполняете `git commit`, Git создаёт 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 последнего коммита).

Шаг 2: Обновление Индекса (--mixed)

Заметьте, если сейчас вы выполните `git status`, то увидите отмеченные зелёным цветом изменения между Индексом и новым HEAD.

Следующим, что сделает `reset`, будет обновление Индекса содержимым того снимка, на который указывает HEAD.

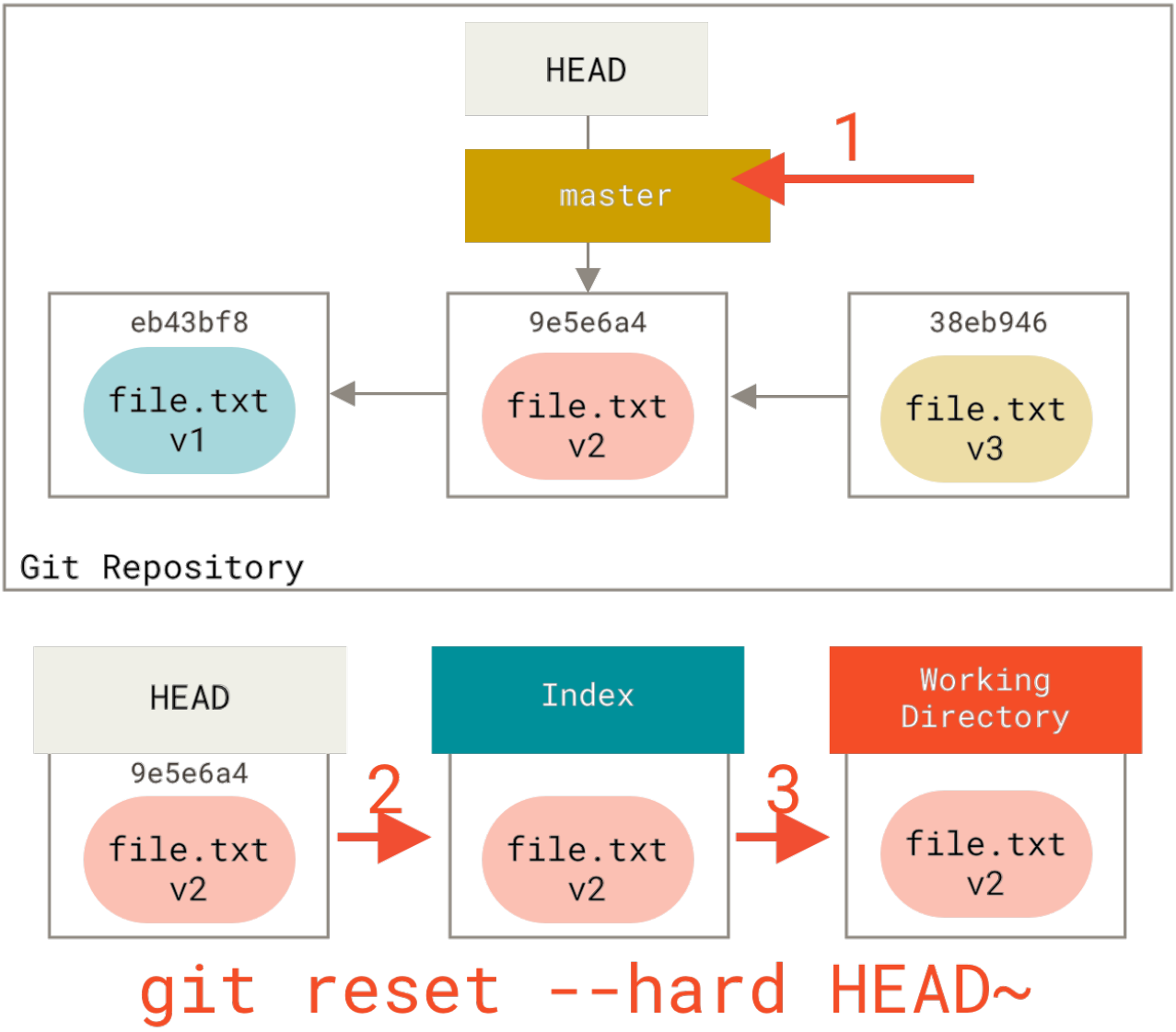


Если вы указали опцию `--mixed`, выполнение `reset` остановится на этом шаге. Такое поведение также используется по умолчанию, поэтому если вы не указали совсем никаких опций (в нашем случае `git reset HEAD~`), выполнение команды также остановится на этом шаге.

Снова взгляните на диаграмму и постарайтесь разобраться, что произошло: отменён 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.



Давайте разберёмся, 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 ещё остаётся в коммите внутри базы данных Git и мы можем вернуть её, просматривая наш `reflog`, но если вы не коммитили эту версию, Git перезапишет файл и её уже нельзя будет восстановить.

Резюме

Команда `reset` в заранее определённом порядке перезаписывает три дерева Git, останавливаясь тогда, когда вы ей скажете:

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.

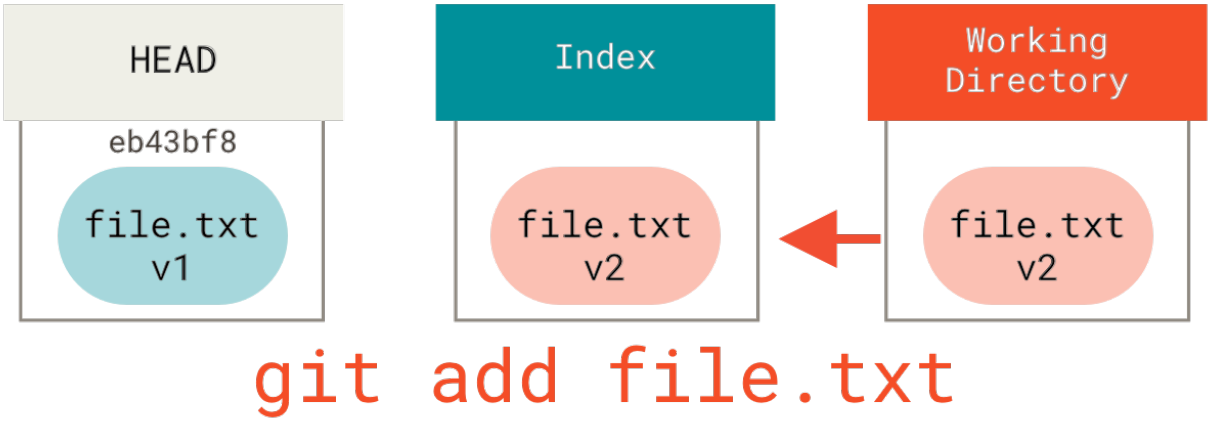
Итак, предположим вы выполнили команду `git reset file.txt`. Эта форма записи (так как вы не указали ни SHA-1 коммита, ни ветку, ни опций `--soft` или `--hard`) является сокращением для `git reset --mixed HEAD file.txt`, которая:

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.

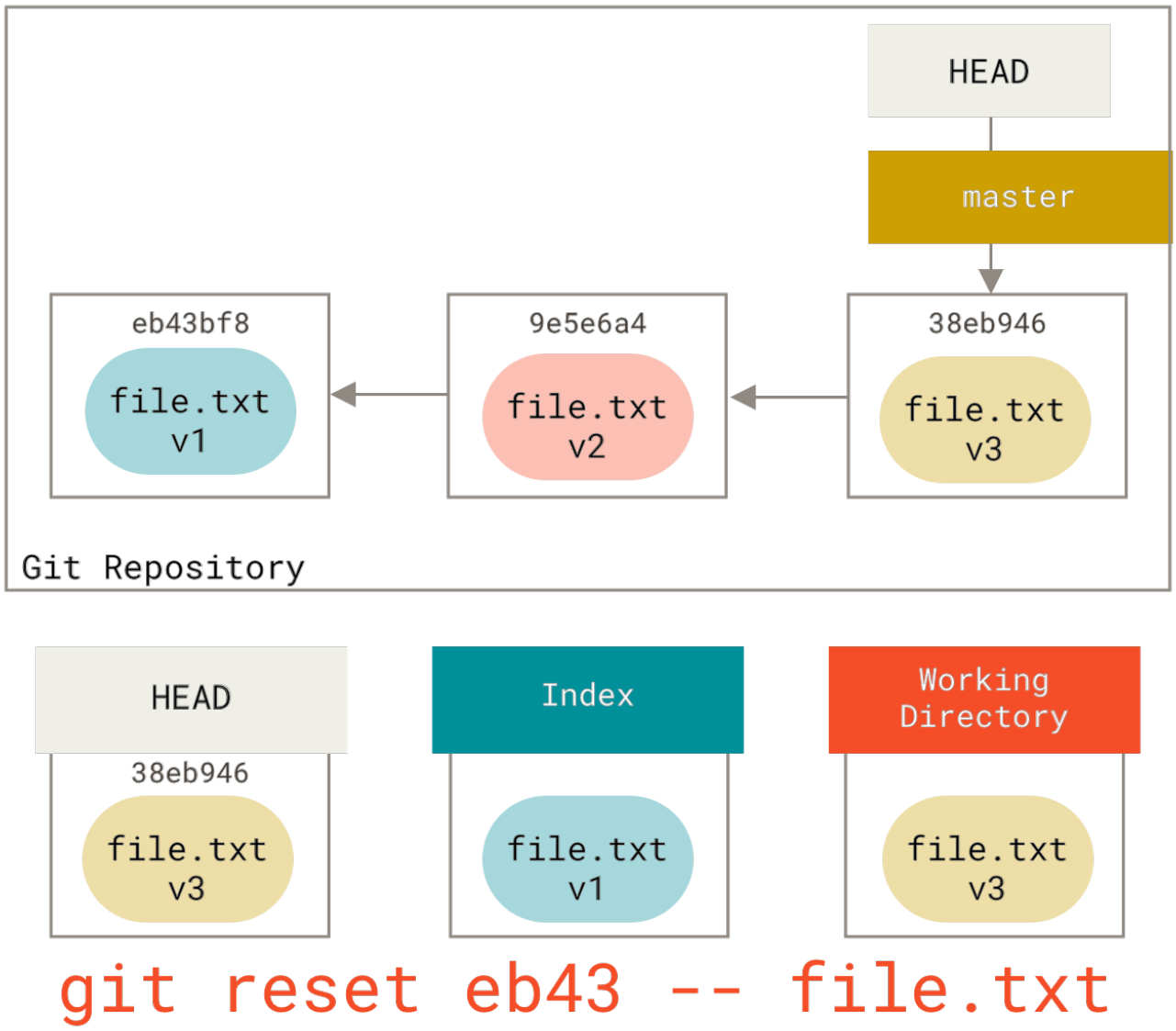
```
git reset file.txt
```

Это создаёт эффект *отмены индексации* файла. Если вы посмотрите на диаграммы этой команды и команды `git add`, то увидите, что их действия прямо противоположные.



Именно поэтому в выводе `git status` предлагается использовать такую команду для отмены индексации файла. (Смотрите подробности в Отмена индексации 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`.



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 возвращался к такой версии.

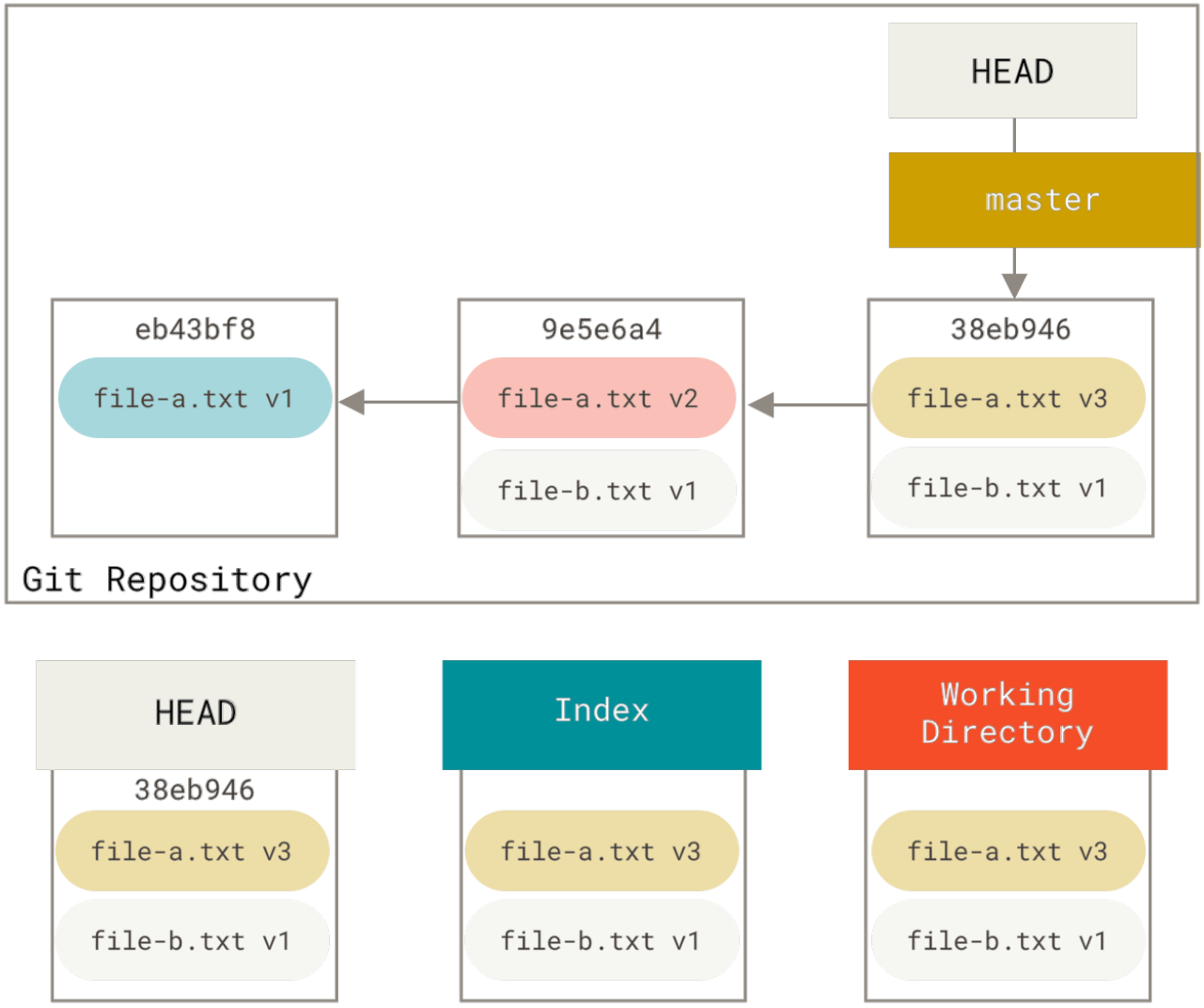
Заметим, что как и команде `git add`, `reset` можно указывать опцию `--patch` для отмены индексации части содержимого. Таким способом вы можете избирательно отменять индексацию или откатывать изменения.

Слияние коммитов

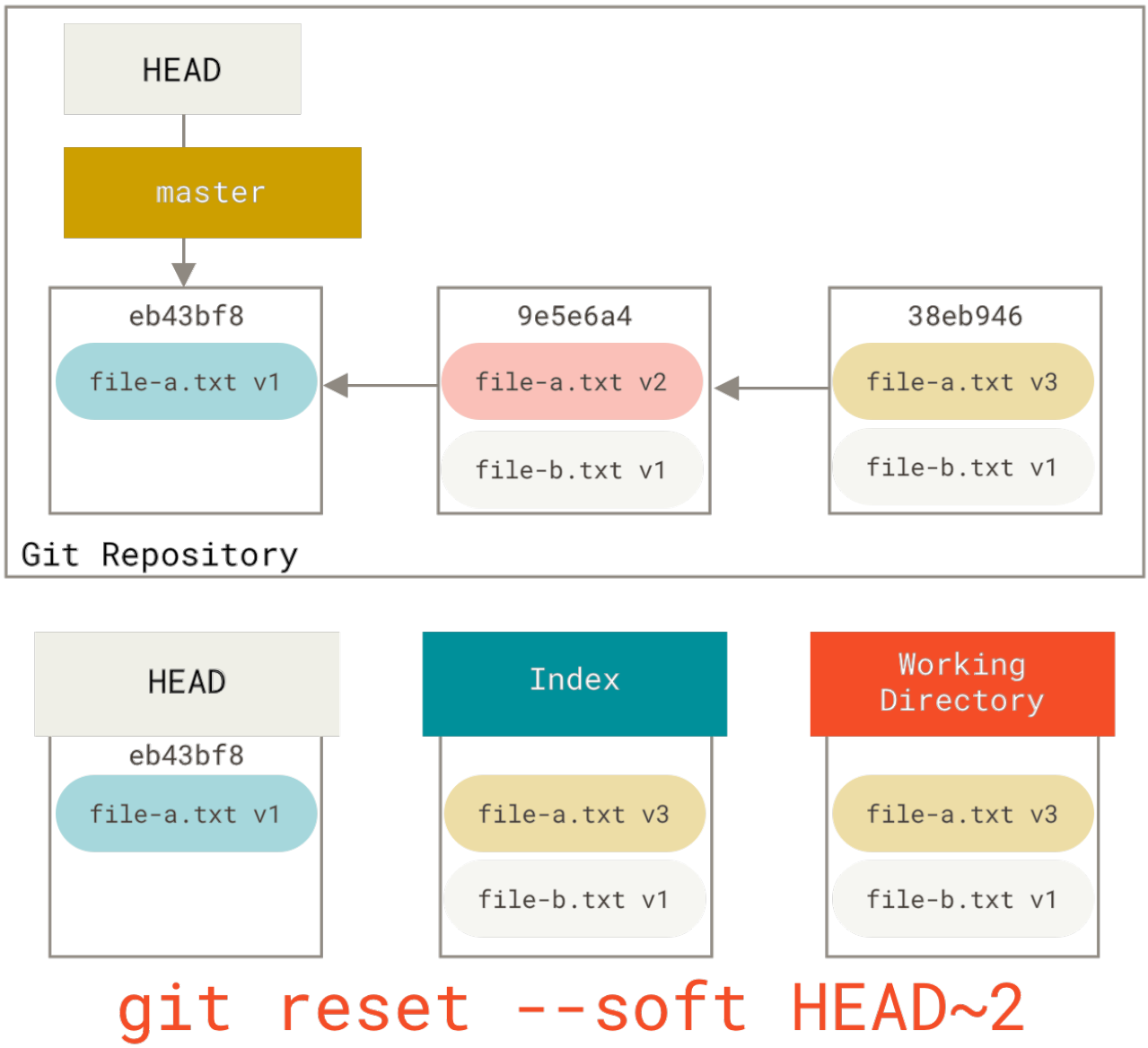
Давайте посмотрим, как, используя вышеизложенное, сделать кое-что интересное — слияние коммитов.

Допустим, у вас есть последовательность коммитов с сообщениями вида «упс.», «В работе» и «позабыл этот файл». Вы можете использовать `reset` для того, чтобы просто и быстро слить их в один. (В разделе Объединение коммитов главы 7 представлен другой способ сделать то же самое, но в данном примере проще воспользоваться `reset`.)

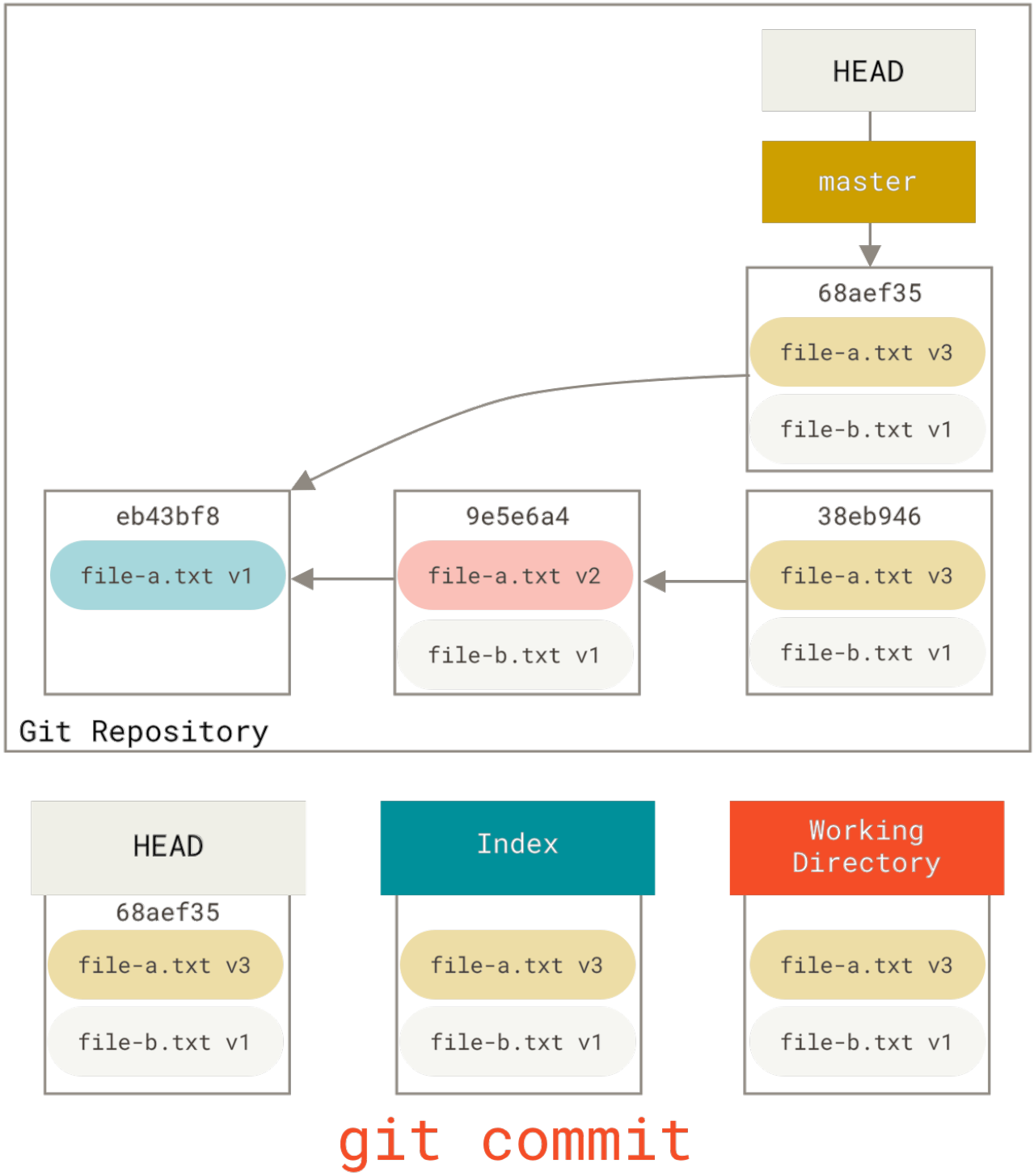
Предположим, у вас есть проект, в котором первый коммит содержит один файл, второй коммит добавляет новый файл и изменяет первый, а третий коммит снова изменяет первый файл. Второй коммит был сделан в процессе работы и вы хотите слить его со следующим.



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):



Затем просто снова выполните `git commit`:



Теперь вы можете видеть, что ваша «достижимая» 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 чём же состоит отличие между `checkout` и `reset`. Как и `reset`, команда `checkout` управляет тремя деревьями Git, и также её поведение зависит от того указали ли вы путь до файла или нет.

Без указания пути

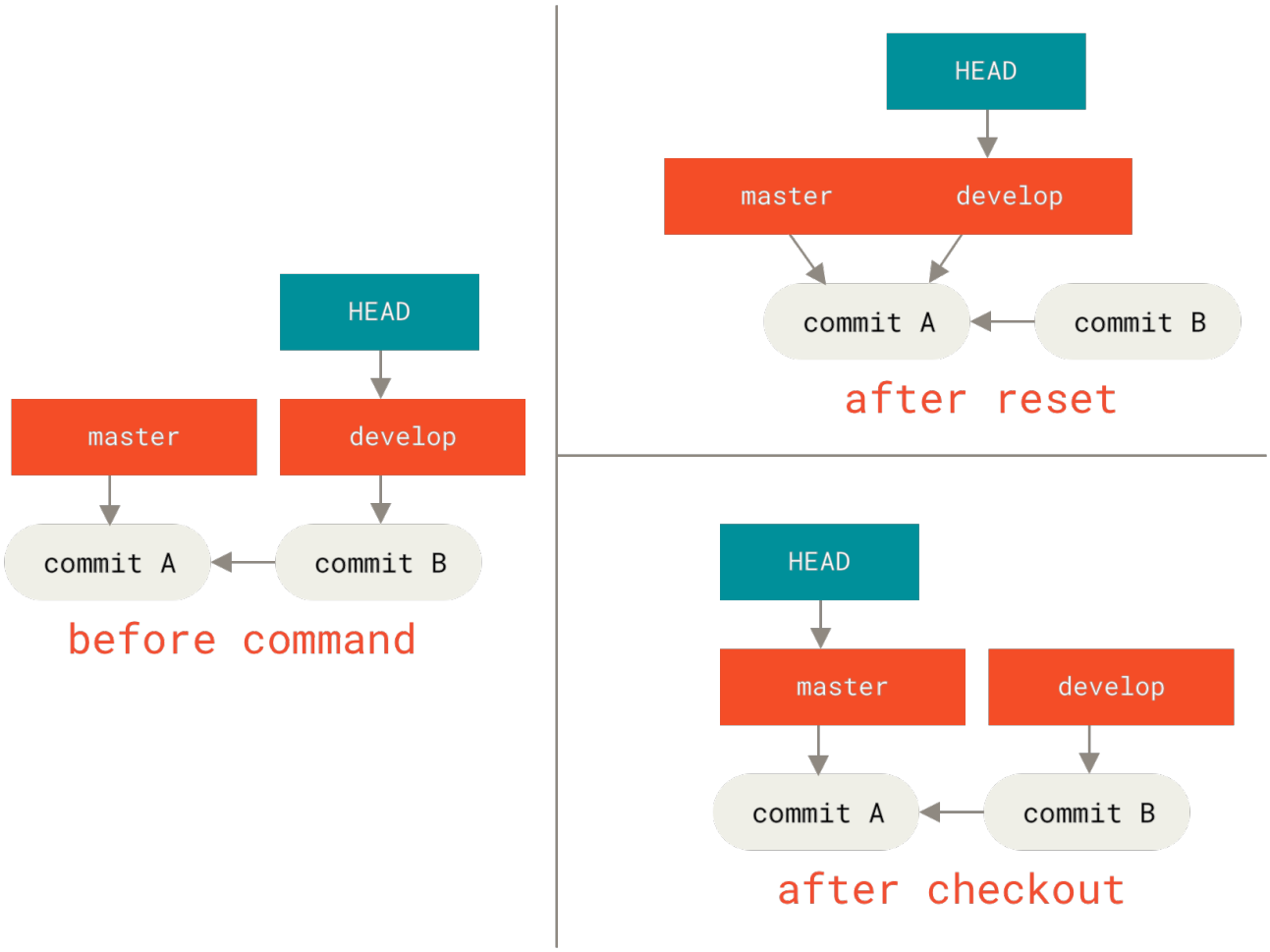
Команда `git checkout [branch]` очень похожа на `git reset --hard [branch]`, в процессе их выполнения все три дерева изменяются так, чтобы выглядеть как `[branch]`. Но между этими командами есть два важных отличия.

Во-первых, в отличие 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* изменяли, были обновлены. С другой стороны, команда `reset --hard` просто заменяет всё целиком, не выполняя проверок.

Второе важное 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 неё). 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.



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 её спокойно использовать. Но, возможно, вы всё ещё 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 рабочего каталога?» — если в нём указано **НЕТ**, то хорошенько подумайте прежде чем выполнить эту команду.

	HEAD	Индекс	Рабочий Каталог	Сохранность рабочего каталога?
На уровне коммитов				
`reset --soft [коммит]`	REF	НЕТ	НЕТ	ДА
`reset [коммит]`	REF	ДА	НЕТ	ДА
`reset --hard [коммит]`	REF	ДА	ДА	**НЕТ**
`checkout [коммит]`	HEAD	ДА	ДА	ДА
На уровне файлов				
`reset (коммит) [путь]`	НЕТ	ДА	НЕТ	ДА
`checkout (коммит) [путь]`	НЕТ	ДА	ДА	**НЕТ**

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 её всё это время в актуальном состоянии, часто разрешая маленькие конфликты, а не доводить дело до одного большого конфликта по завершению всех изменений.

Однако, иногда всё же будут возникать сложные конфликты. В отличие 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 копия была без локальных изменений. Если у вас есть несохранённые наработки, либо припрячьте их, либо сохраните их во временной ветке. Таким образом, вы сможете легко отменить **любые** изменения, которые сделаете в рабочем каталоге. Если при выполнении слияния вы не сохраните сделанные изменения, то некоторые из описанных ниже приёмов 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 'Convert hello.rb to DOS'
[whitespace 3270f76] Convert hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -b
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 'Use Spanish instead of English'
[whitespace 6d338d2] Use Spanish instead of English
 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 'Add comment documenting the function'
[master bec6336] Add comment documenting the function
 1 file changed, 1 insertion(+)
```

Теперь мы попытаемся слить в текущую 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 или не зафиксированные изменения в рабочем каталоге, во всех остальных случаях всё будет хорошо.

Если по каким-то причинам вы обнаружили себя в ужасном состоянии и хотите просто начать всё 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 рабочем каталоге, 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 нём будет удалена и добавлена снова. По умолчанию Git считает все эти строки изменёнными 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-space-change 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 всё сольётся 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 сливаемый файл через программу `dos2unix`. Как мы будем делать это?

Во-первых, мы перейдём в состояние конфликта слияния. Затем нам необходимо получить копии *нашей* версии файла, *их* версии файла (из ветки, которую мы сливаем) 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 вы сливаете («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 нашем рабочем каталоге присутствует содержимое всех трёх 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 -b
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 коммитом изменений вы хотите посмотреть какие в действительности были различия между состояниями, то можете воспользоваться командой `git diff`, сравнивающей содержимое вашего рабочего каталога, которое будет зафиксировано как результат слияния, с любым из трёх 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 -b
* 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 -b
* 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 команды `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 коммитов, что при слиянии приводит к справедливому конфликту.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) Update README
```

```
* 9af9d3b Create README
* 694971d Update phrase to 'hola world'
| * e3eb223 (mundo) Add more tests
| * 7cff591 Create initial testing script
| * c3ffff1 Change text to 'hello mundo'
|/
* b7dcc89 Initial hello world code
```

У нас есть три уникальных коммита, 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 world'
=======
  puts 'hello mundo'
>>>>>>> mundo
end

hello()
```

В обеих сливаемых ветках в этот файл было добавлено содержимое, но в некоторых коммитах изменялись 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 выкачает 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 world'
||||||| base
  puts 'hello world'
=======
  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 коммитом 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 полезным просмотреть историю, чтобы понять почему в двух ветках разработки изменялась одна и та же область кода.

Для получения полного списка всех уникальных коммитов, которые были сделаны в любой из сливаемых веток, мы можем использовать синтаксис «трёх точек», который мы изучили в Три точки.

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
```

```
< f1270f7 Update README
< 9af9d3b Create README
< 694971d Update phrase to 'hola world'
> e3eb223 Add more tests
> 7cff591 Create initial testing script
> c3ffff1 Change text to 'hello mundo'
```

Это список всех шести коммитов, включённых в слияние, с указанием также ветки разработки, в которой находится каждый из коммитов.

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 коммиты, в которых изменялся конфликтующий в данный момент файл.

```
$ git log --oneline --left-right --merge
< 694971d Update phrase to 'hola world'
> c3ffff1 Change text to 'hello mundo'
```

Если вы выполните эту команду с опцией `-p`, 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 ещё 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'
++>>>>>>> mundo
  end
```

```
  hello()
```

Такой формат называется «комбинированным» («Combined Diff»), для каждого различия в нём содержится два раздела с информацией. В первом разделе отображены различия строки (добавлена она или удалена) между «вашей» веткой и содержимым вашего рабочего каталога, а во втором разделе содержится то же самое, но между «их» веткой и рабочим каталогом.

Таким образом, в данном примере вы можете увидеть строки `<<<<<<<` и `>>>>>>>` в файле в вашем рабочем каталоге, 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 копии, строка «hello mundo» была в «их» ветке, но не в рабочей копии, и, наконец, «hola mundo» не была ни в одной из сливаемых веток, но сейчас присутствует в рабочей копии. Это бывает полезно просмотреть перед коммитом разрешения конфликта.

Такую же информацию вы можете получить и после выполнения слияния с помощью команды `git log`, узнав таким образом как был разрешён конфликт. Git выводит информацию в таком формате, если вы выполните `git show` для коммита слияния или вызовете команду `git log -p` с опцией `--cc` (без неё данная команда не показывает изменения для коммитов 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 world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

Отмена слияний

Теперь когда вы знаете как создать коммит слияния, вы можете сделать его 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 большинстве случаев сделать это просто).

Коммит слияния не исключение. Допустим, вы начали работать в тематической ветке, случайно слили её в `master`, и теперь ваша история коммитов выглядит следующим образом:

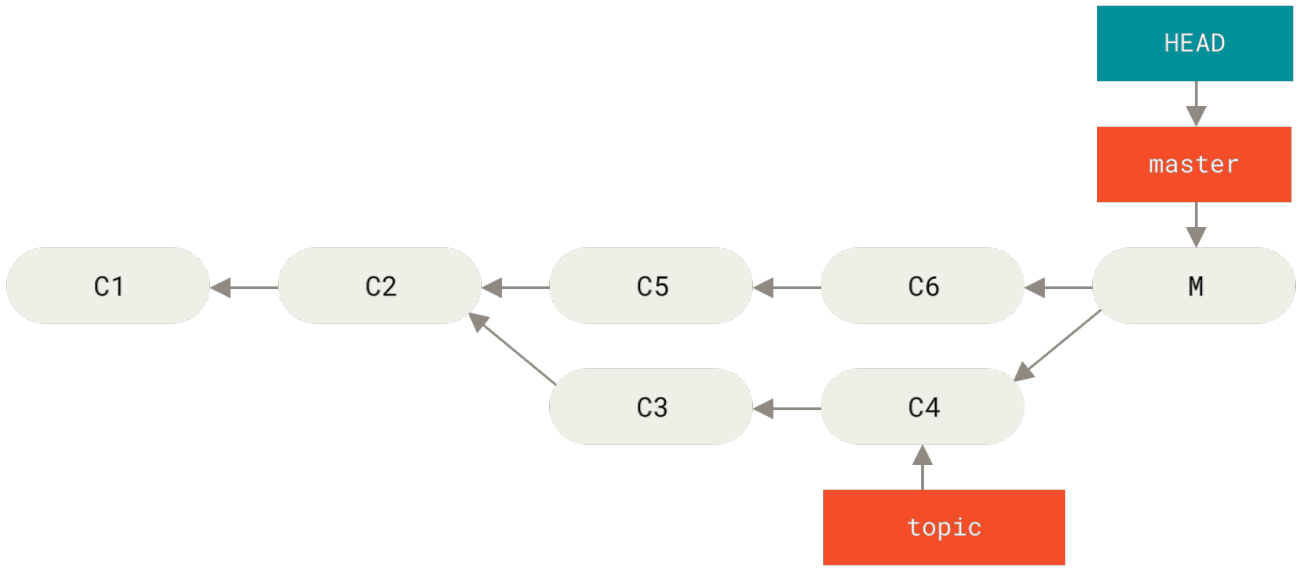


Рисунок 137. Случайный коммит слияния

Есть два подхода к решению этой проблемы, в зависимости от того, какой результат вы хотите получить.

Исправление ссылок

Если нежелаемый коммит 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:

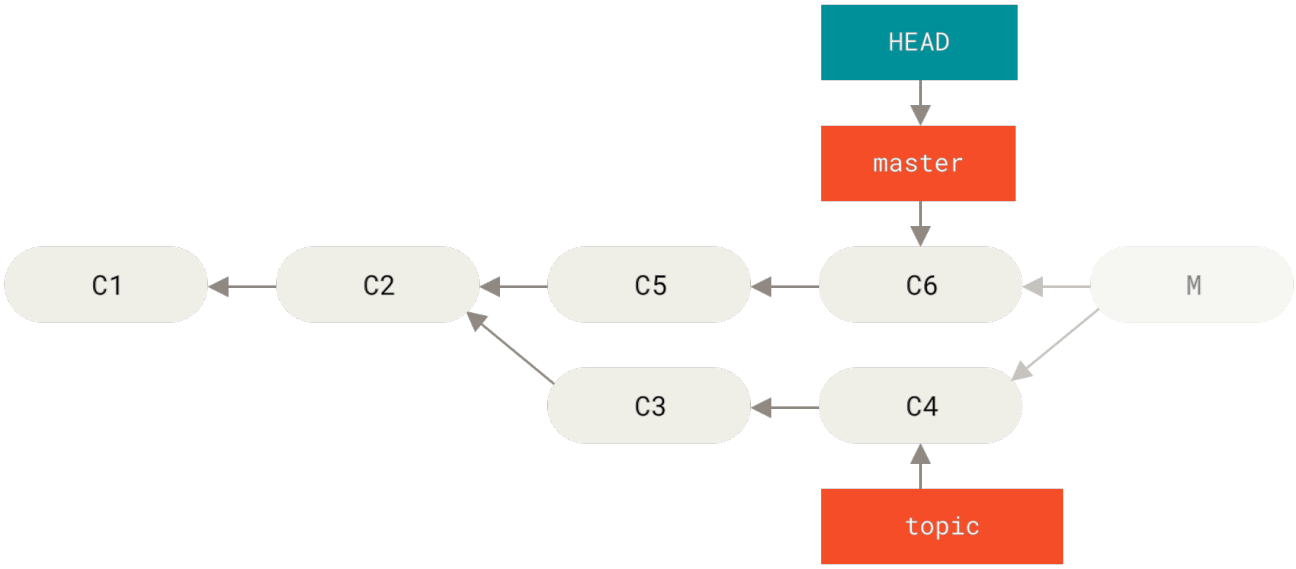


Рисунок 138. История после `git reset --hard HEAD~`

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 коммита слияния (`C6`).
2. Приводит индекс к такому же виду что и HEAD.
3. Приводит рабочий каталог 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 Опасности перемещения, чтобы узнать что именно может произойти; кратко говоря, если у других людей уже есть какие-то из изменяемых вами коммитов, вы должны отказаться от использования `reset`. Этот подход также не будет работать, если после слияния уже был сделан хотя бы один коммит; перемещение ссылки фактически приведёт к потере этих изменений.

Отмена коммита

Если перемещение указателей ветки вам не подходит, Git предоставляет возможность сделать новый коммит, который откатывает все изменения, сделанные в другом. Git называет эту операцию «восстановлением» («revert»), в данном примере вы можете вызвать

её 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`), новый коммит 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, внесённые слиянием родителя #2 (`C4`), и сохранить при этом всё содержимое из родителя #1 (`C6`).

История с коммитом восстановления (отменой коммита слияния) выглядит следующим образом:

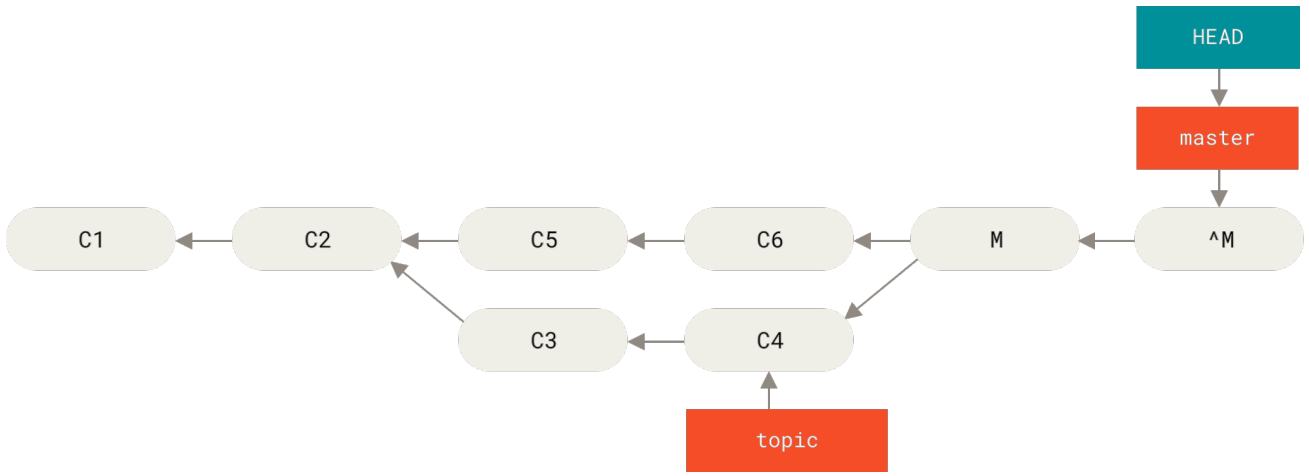


Рисунок 139. История после `git revert -m 1`

Новый коммит `^M` имеет точно такое же содержимое как `C6`, таким образом, начиная с него всё выглядит так, как будто слияние никогда не выполнялось, за тем лишь исключением, что «теперь уже не слитые» коммиты всё также присутствуют в истории `HEAD`. Git придёт в замешательство, если вы вновь попытаетесь слить `topic` в ветку `master`:

```
$ git merge topic
Already up-to-date.
```

В ветке `topic` нет ничего, что ещё 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:

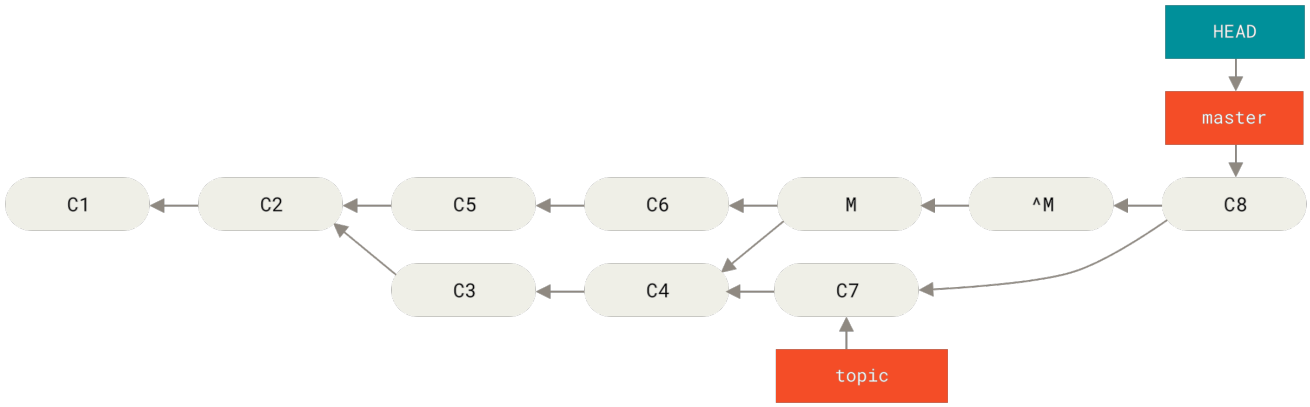


Рисунок 140. История с плохим слиянием

Лучшим решением данной проблемы является откат коммита отмены слияния, так как теперь вы хотите внести изменения, которые были отменены, а **затем** создание нового коммита слияния:

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'""
$ git merge topic
```

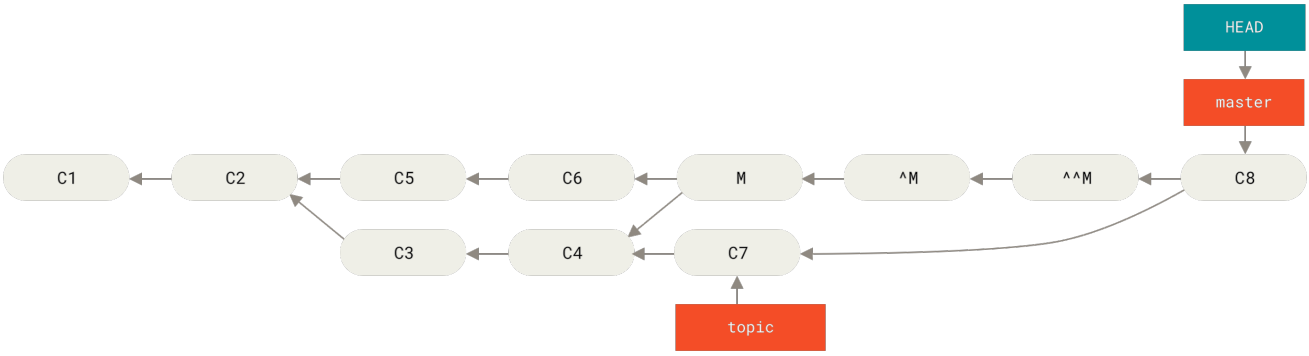


Рисунок 141. История после повторения отменённого слияния

В этом примере, `M` и `^M` отменены. В коммите `^^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 ещё один полезный приём, 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 определённую 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 сольёт, а для конфликтующих он целиком возьмёт ту версию, которую вы указали (это относится и к бинарным файлам).

Если мы вернёмся к примеру «hello world», который использовали раньше, то увидим, что попытка слияния в нашу ветку приведёт 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 то же самое что *опция* «ours» рекурсивной стратегии слияния.

Фактически, эта стратегия выполнит ненастоящее слияние. Она создаст новый коммит слияния, у которого 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 слить обратно в `master`. Тем временем в `master` были сделаны некоторые исправления, которые необходимо перенести также в вашу ветку `release`. Вы можете слить 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 поддеревьев

Идея слияния поддеревьев состоит в том, что у вас есть два проекта и один из проектов отображается в подкаталог другого. Когда вы выполняете слияние поддеревьев, Git в большинстве случаев способен понять, что одно из них является поддеревом другого и выполнить слияние подходящим способом.

Далее мы рассмотрим пример добавления в существующий проект другого проекта и последующее слияние кода второго проекта в подкаталог первого.

Первым делом мы добавим в наш проект приложение Rack. Мы добавим Rack в наш собственный проект, как удалённый репозиторий, а затем выгрузим его в отдельную ветку.

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote --no-tags
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 подкаталог 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 главе Git изнутри, 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_branch` в подкаталог `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 этот подкаталог — 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`.

Для того, чтобы получить изменения и заполнить сообщение коммита используйте параметр `--squash`, 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 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 коммита. Вы также можете поступить наоборот — сделать изменения в подкаталоге `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 поддеревьев даёт нам возможность использовать рабочий процесс в некоторой степени похожий на рабочий процесс с подмодулями, но при этом без использования подмодулей (которые мы рассмотрим в Подмодули). Мы можем держать ветки с другими связанными проектами в нашем репозитории и периодически сливать их как поддеревья в наш проект. С одной стороны это удобно, например, тем, что весь код хранится в одном месте. Однако, при этом есть и некоторые недостатки — поддеревья 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 подкаталога `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 следует выполнить команду `git diff-tree`, указав ветку, с которой производится сравнение:

```
$ git diff-tree -p rack_branch
```

Вот как выглядит процедура сравнения содержимого подкаталога `rack` с содержимым ветки `master` на сервере после последнего скачивания изменений:

```
$ git diff-tree -p rack_remote/master
```

Rerere

Функциональность `git rerere` — частично скрытый компонент Git. Её имя является сокращением для «reuse recorded resolution» («повторно использовать сохранённое решение»). Как следует из названия, 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 `rerere` 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 `rerere` 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 уже сливали и разрешали при этом некоторые конфликты, вместо слияния перебазировать — навряд ли вы захотите разрешать те же конфликты ещё раз.

Другая ситуация возникает, когда вы изредка сливаете несколько веток, относящихся к ещё

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 `rerere`, достаточно изменить настройки следующим образом:

```
$ git config --global rerere.enabled true
```

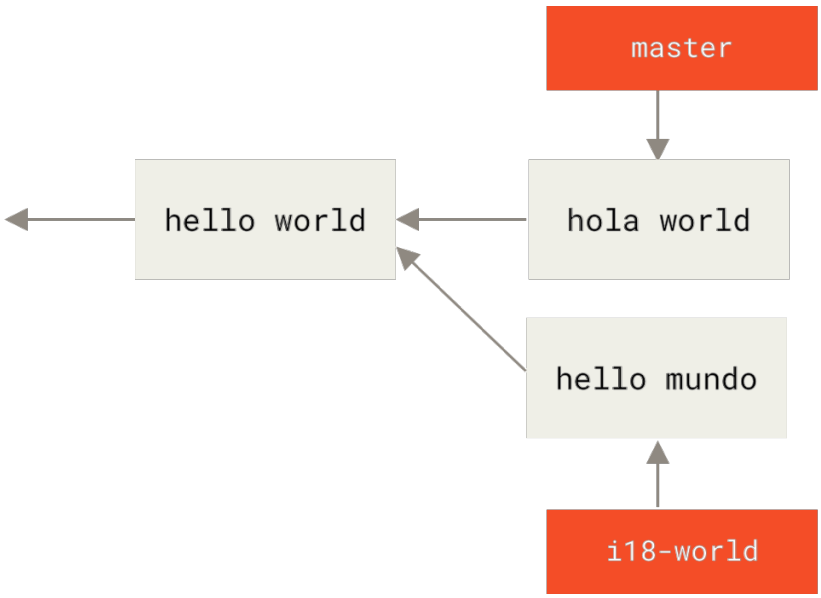
Также вы можете включить её, создав каталог `.git/rr-cache` в нужном репозитории, но включение через настройки понятнее и может быть сделано глобально.

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».



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 preimage for FILE`. Во всём остальном вывод такой же, как при обычном конфликте слияния. В этот момент `rerere` 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 rerere status` вы также можете узнать, для каких файлов `git rerere` сохранил снимки состояния, в котором они были до начала слияния:

```
$ git rerere status
hello.rb
```

А команда `git rerere 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 rerere 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 `rerere`), 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 `rerere diff`, чтобы увидеть, что именно `rerere` запомнит:

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
 #! /usr/bin/env ruby

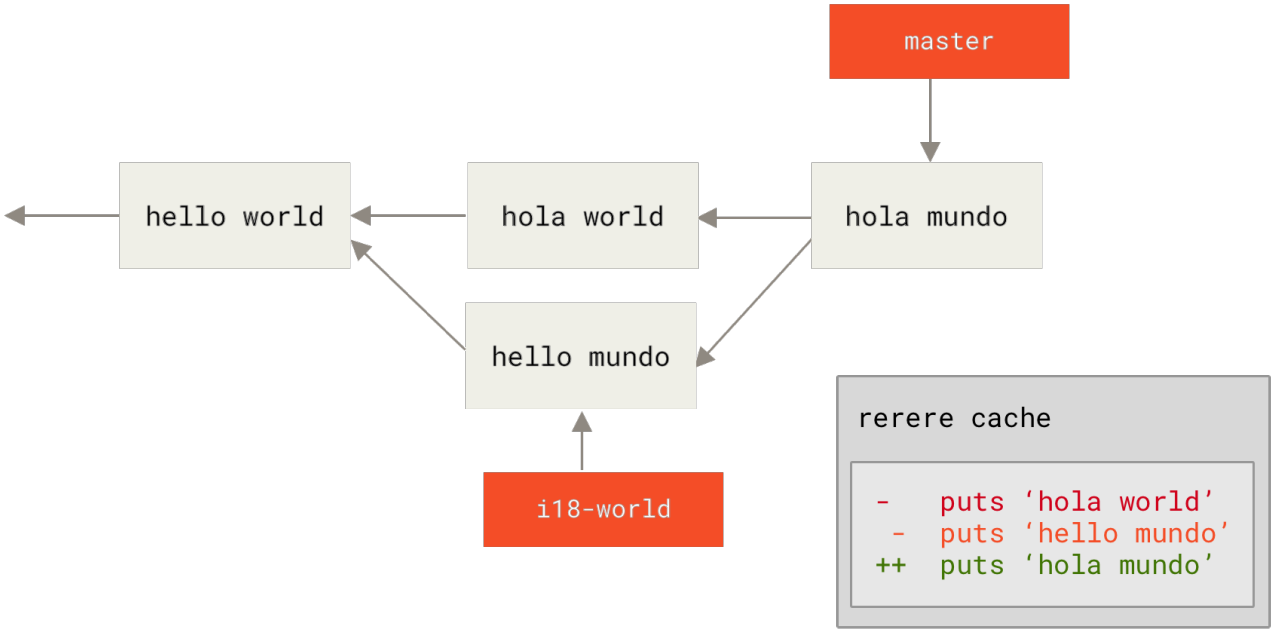
 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
-  puts 'hola world'
->>>>>>>
+  puts 'hola mundo'
 end
```

То есть, когда Git увидит в файле `hello.rb` конфликт, в котором с одной строны стоит «hello mundo» и «hola world» с другой, он разрешит его как «hola mundo».

Теперь мы можем отметить конфликт как разрешённый 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»).



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 разрешён, и в файле отсутствуют маркеры конфликта слияния.

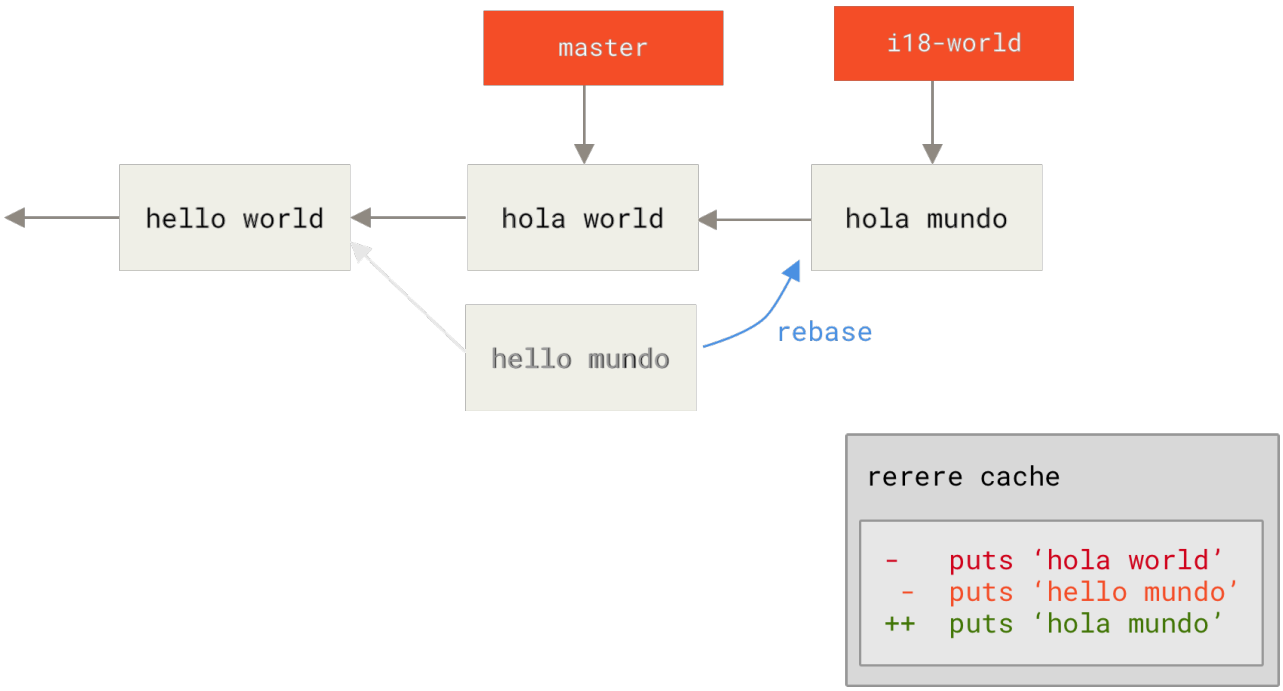
```
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

При этом команда `git diff` покажет вам *как именно* этот конфликт был автоматически повторно разрешён:

```
$ 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
```



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 world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end
```

Мы видели пример этого в Продвинутое слияние. Теперь давайте повторно разрешим конфликт используя `rerere`:

```
$ 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 сохранённый `rerere` 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 `rerere`. 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 обобщённые возможности, но часто они могут помочь вам отловить ошибку или её 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 неё для любого файла можно увидеть, каким коммитом последний раз изменяли 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 `git blame`, чтобы определить, какой коммит и коммиттер отвечал за строки в `Makefile` ядра Linux верхнего уровня и, кроме того, использует параметр `-L` для ограничения вывода аннотации строками с 69 по 82 из этого файла:

```
$ git blame -L 69,82 Makefile
b8b0618cf6fab (Cheng Renquan  2009-05-26 16:03:07 +0800 69) ifeq ("$(origin V)",
"command line")
b8b0618cf6fab (Cheng Renquan  2009-05-26 16:03:07 +0800 70)   KBUILD_VERBOSE = $(V)
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 71) endif
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 72) ifndef KBUILD_VERBOSE
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 73)   KBUILD_VERBOSE = 0
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 74) endif
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 75)
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 76) ifeq ($(KBUILD_VERBOSE),1)
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 77)   quiet =
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 78)   Q =
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 79) else
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 80)   quiet=quiet_
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 81)   Q = @
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 82) endif
```

Обратите внимание, что первое поле — это неполная SHA-1 сумма последнего коммита, который изменял соответствующую строку. Следующими двумя полями являются значения, извлечённые 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 попытается 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 COMMI
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)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 149)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)         if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)                 [refDict setOb
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 отчёты о некоторой ошибке, которая не возникала в вашем разработческом окружении, и вы не можете представить, почему код ведёт себя так. Вы возвращаетесь к вашему коду и выясняете, что можете воспроизвести проблему, но всё ещё 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, расположенном посредине 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
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 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-error.sh
```

Это приведёт к автоматическому выполнению `test-error.sh` на каждый выгруженный коммит до тех пор, пока Git не найдёт первый сломанный коммит. Вы также можете использовать что-то вроде `make` или `make tests`, или что-то ещё, что у вас есть для запуска автоматизированных тестов.

Подмодули

Часто при работе над одним проектом, возникает необходимость использовать в нём 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.

Приведём пример. Предположим, вы разрабатываете веб-сайт и создаёте ленту в формате Atom. Вместо написания собственного генератора Atom, вы решили использовать библиотеку. Скорее всего, вы либо подключите библиотеку как сторонний модуль, такой как модуль CPAN или Ruby Gem, либо скопируете исходный код библиотеки в свой проект. Проблема использования библиотеки состоит в трудности её адаптации под собственные нужды и часто более сложным является её распространение, так как нужно быть уверенным, что каждому клиенту доступна изменённая версия. При включении кода библиотеки в свой проект проблемой будет объединение ваших изменений в ней с новыми изменениями в оригинальном репозитории.

Git решает эту проблему с помощью подмодулей. Подмодули позволяют вам сохранить один Git-репозиторий, как подкаталог другого Git-репозитория. Это даёт вам возможность клонировать в ваш проект другой репозиторий, но коммиты при этом хранить отдельно.

Начало работы с подмодулями

Далее мы рассмотрим процесс разработки простого проекта, разбитого на один главный проект и несколько подпроектов.

Давайте начнём с добавления существующего Git-репозитория как подмодуля репозитория, в котором мы работаем. Для добавления нового подмодуля используйте команду `git submodule add`, указав относительный или абсолютный 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 подмодуль сохраняется в каталог с именем репозитория, в нашем примере — «`DbConnector`». Изменить каталог сохранения подмодуля можно указав путь к нему в конце команды.

Если в данный момент выполнить `git status`, то можно заметить следующее.

```
$ 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 увидите новый файл `.gitmodules`. Это конфигурационный файл, в котором хранится соответствие между URL проекта и локальным подкаталогом, в который вы его выкачали:

```
[submodule "DbConnector"]
    path = DbConnector
    url = https://github.com/chaconinc/DbConnector
```

Если у вас несколько подмодулей, то в этом файле у вас будет несколько записей. Важно заметить, что этот файл добавлен под версионный контроль 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, которые 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 тот URL, к которому у других участников будет доступ. В таком случае для себя вы можете задать локальное значение URL с помощью команды `git config submodule.DbConnector.url PRIVATE_URL`. Если применимо, относительные URL могут помочь.

Следующим пунктом в списке вывода команды `git status` будет запись о каталоге проекта. Команда `git diff` покажет для него кое-что интересное:

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

Хоть `DbConnector` и является подкаталогом вашего рабочего каталога, Git распознает его как подмодуль и не отслеживает его содержимое пока вы не перейдёте в него. Вместо этого, Git видит его как конкретный коммит этого репозитория.

Если вам нужен немного более понятный вывод, передайте команде `git diff` параметр `--submodule`.

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+       path = DbConnector
+       url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000...c3f01dc (new submodule)
```

При создании коммита вы увидите следующее:

```
$ git commit -am 'Add DbConnector module'
[master fb9093c] Add 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 означают, что вы сохраняете коммит как элемент каталога, а не как подкаталог или файл.

Наконец, отправьте эти изменения:

```
$ git push origin master
```

Клонирование проекта с подмодулями

Далее мы рассмотрим клонирование проекта, содержащего подмодули. Когда вы клонируете такой проект, по умолчанию вы получите каталоги, 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
$
```

Каталог `DbConnector` присутствует, но он пустой. Вы должны выполнить две команды: `git submodule init` — для инициализации локального конфигурационного файла, и `git submodule update` — для получения всех данных этого проекта и извлечения соответствующего коммита, указанного в 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 ваш каталог `DbConnector` находится в точно таком же состоянии, как и ранее при выполнении коммита.

Однако, существует немного более простой вариант сделать то же самое. Если вы передадите опцию `--recurse-submodules` команде `git clone`, то она автоматически инициализирует и обновит каждый подмодуль в этом репозитории, включая вложенные подмодули, при условии, что подмодули репозитория имеют свои подмодули.

```
$ git clone --recurse-submodules 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 path
'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.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Если вы уже клонировали проект, но забыли указать `--recurse-submodules`, вы можете объединить шаги `git submodule init` и `git submodule update`, запустив `git submodule update --init`. Также для инициализации, получения и извлечения всех существующих вложенных подмодулей можно использовать безопасную команду `git submodule update --init --recursive`.

Работа над проектом с подмодулями

Теперь, когда у нас есть копия проекта с подмодулями, мы будем работать совместно с нашими коллегами над обоими проектами: основным проектом и проектом подмодуля.

Получение изменений подмодуля из удалённого репозитория

Простейший вариант использования подмодулей в проекте состоит в том, что вы просто получаете сам подпроект и хотите периодически получать обновления, при этом в своей копии проекта ничего не изменяете. Давайте рассмотрим этот простой пример.

Если вы хотите проверить наличие изменений в подмодуле, перейдите в его каталог и выполните `git fetch`, а затем `git merge` для обновления локальной версии отслеживаемой ветки.

```
$ 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 вернётесь в основной проект и выполните `git diff --submodule`, то увидите, что подмодуль обновился, а так же список новых коммитов. Если вы не хотите каждый раз при вызове команды `git diff` указывать опцию `--submodule`, то задайте конфигурационному параметру `diff.submodule` значение «`log`» для использования такого формата вывода по умолчанию.

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

Если сейчас вы создадите коммит, то состояние подмодуля будет зафиксировано с учётом последних изменений, что позволит другим людям их получить.

Если вы не хотите вручную извлекать и сливать изменения в подкаталог, то для вас существует более простой способ сделать то же самое. Если вы выполните `git submodule update --remote`, то Git сам перейдёт в ваши подмодули, получит 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, при желании вы можете задать другую ветку. Например, если вы хотите, чтобы подмодуль DbConnector отслеживал ветку `stable`, то можете указать 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 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'
```

Если опустить `-f .gitmodules`, то команда сделает изменения в локальном файле `.git/config`, но, вероятно, имеет смысл всё же отслеживать эту информацию в репозитории, так чтобы и все остальные участники имели к ней доступ.

Если сейчас выполнить команду `git status`, то Git покажет нам наличие «новых коммитов» 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 установить 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 выполнить `git diff`, то увидите изменённый файл `.gitmodules`, а также наличие нескольких недавно полученных коммитов, которые готовы к включению в проект нашего

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 коммитов в нашем подмодуле. После создания коммита вы тоже можете получить эту информацию, выполнив команду `git log -p`.

```
$ git log -p --submodule
commit 0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae
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
```

При выполнении команды `git submodule update --remote` Git по умолчанию будет пытаться обновить **все** подмодули. Поэтому, при большом количестве подмодулей вы возможно захотите указать название только того подмодуля, который необходимо обновить.

Получение изменений основного проекта из удалённого репозитория

Теперь давайте поставим себя на место нашего коллеги, у которого есть собственная копия репозитория MainProject. Простого запуска команды `git pull` для получения последних изменений уже недостаточно:

```
$ git pull
From https://github.com/chaconinc/MainProject
   fb9093c..0a24cfc  master     -> origin/master
Fetching submodule DbConnector
From https://github.com/chaconinc/DbConnector
   c3f01dc..c87d55d  stable     -> origin/stable
Updating fb9093c..0a24cfc
Fast-forward
 .gitmodules         | 2 +-
 DbConnector         | 2 +-
 2 files changed, 2 insertions(+), 2 deletions(-)

$ 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:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c87d55d...c3f01dc (4):
  < catch non-null terminated lines
  < more robust error handling
  < more efficient db routine
  < better connection routine

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

По умолчанию, команда `git pull` рекурсивно получает изменения для подмодулей, что можно увидеть выше в выводе первой команды. При этом, она не **обновляет** подмодули. Это отображается в выводе команды `git status`, которая показывает, что подмодуль изменён (`modified`) и содержит новые коммиты (`new commits`). Более того, новые коммиты обозначены символом «<», означающим, что эти коммиты добавлены в проект MainProject, но не извлечены в текущем состоянии подмодуля DbConnector. Для завершения обновления следует выполнить команду `git submodule update`:

```
$ git submodule update --init --recursive
Submodule path 'vendor/plugins/demo': checked out
'48679c6302815f6c76f1fe30625d795d9e55fc56'
```

```
$ git status
 On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

На всякий случай, команду `git submodule update` лучше запускать с параметром `--init`, чтобы проинициализировать новые подмодули, которые, возможно, были добавлены в новых коммитах MainProject репозитория, а так же с параметром `--recursive`, чтобы обновить вложенные подмодули при их наличии.

Если вы хотите автоматизировать этот процесс, то добавьте параметр `--recurse-submodules` при запуске команды `git pull` (начиная с версии 2.14). Это приведёт к выполнению команды `git submodule update` сразу после получения и объединения изменений, приводя подмодули в корректное состояние. Более того, если вы хотите, чтобы Git всегда добавлял параметр `--recurse-submodules` к команде `git pull`, можно установить конфигурационный параметр `submodule.recurse` в значение `true` (работает для команды `git pull` начиная с версии 2.15). Если этот параметр конфигурации установлен, Git будет всегда добавлять `--recurse -submodules` ко всем командам, которые его поддерживают (кроме `clone`).

Существует специфическая ситуация, которая может возникнуть при получении изменений основного проекта: в одном из коммитов изменён URL подмодуля в файле `.gitmodules`. Такое может произойти, например, когда проект подмодуля переехал на другой хостинг. В таком случае команды `git pull --recurse-submodules` или `git submodule update` могут завершиться с ошибкой, потому как основной проект ссылается на коммит подмодуля, который не найден в удалённом репозитории этого подмодуля, на который указывает локальная конфигурация в основном репозитории. Чтобы исправить эту ситуацию требуется запуск команды `git submodule sync`:

```
# copy the new URL to your local config
$ git submodule sync --recursive
# update the submodule from the new URL
$ git submodule update --init --recursive
```

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 внесём изменения в подмодуль, зафиксировав и опубликовав все эти изменения в одно и то же время.

Ранее, когда мы выполняли команду `git submodule update` для извлечения изменений из репозитория подмодуля, Git получал изменения и обновлял файлы в поддиректории, но оставлял вложенный репозиторий в состоянии, называемом «отсоединённый HEAD» («`detached HEAD`»). Это значит, что локальная рабочая ветка (такая, например, как `master`),

отслеживающая изменения, отсутствует. Отсутствие отслеживающей ветки означает, что даже если вы зафиксируете изменения в подмодуле, эти изменения, скорее всего, будут потеряны при следующем запуске `git submodule update`. Если хотите, чтобы изменения в подмодуле отслеживались, потребуется выполнить несколько дополнительных шагов.

Для подготовки репозитория подмодуля к такой работе с ним, нужно сделать две вещи. Нужно перейти в каждый подмодуль и извлечь ветку, в которой будете далее работать. Затем необходимо сообщить Git что ему делать, если вы внесли свои изменения, а затем получаете новые изменения из удалённого репозитория командой `git submodule update --remote`. Возможны два варианта — вы можете слить их в вашу локальную версию или попробовать перебазировать ваши локальные наработки поверх новых изменений.

Первым делом, давайте перейдём в каталог нашего подмодуля и переключимся на нужную ветку.

```
$ cd DbConnector/
$ git checkout stable
Switched to branch 'stable'
```

Давайте попробуем обновить подмодуль путём слияния изменений. Для этого просто добавьте опцию `--merge` при вызове команды `update`. Ниже мы видим, что с удалённого сервера получен коммит для подмодуля и слит с текущим состоянием:

```
$ cd ..
$ 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'
```

Перейдя в каталог DbConnector, можно увидеть, что новые изменения уже слиты в нашу локальную ветку `stable`. Теперь давайте посмотрим что происходит, когда мы вносим локальные изменения в библиотеку, а кто-то другой в то же время отправляет другие изменения в удалённый репозиторий.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'Unicode support'
[stable f906e16] Unicode support
```

```
 1 file changed, 1 insertion(+)
```

Теперь мы можем увидеть что происходит при обновлении подмодуля, когда локальные изменения должны включать новые изменения, полученные с удалённого сервера.

```
$ cd ..
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
Applying: Unicode support
Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Если вы забудете указать параметры `--rebase` или `--merge`, то Git просто обновит подмодуль до состояния репозитория на сервере и установит его в состояние отсоединённого HEAD.

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Если такое происходит — не беспокойтесь, вы всегда можете перейти в каталог подмодуля, переключиться на вашу ветку (которая всё ещё будет содержать ваши наработки) и вручную слить ветку `origin/stable` или перебазировать свои изменения относительно неё (или любую другую удалённую ветку).

Если вы не зафиксировали ваши изменения в подмодуле и выполнили его обновление, то это приведёт к проблемам — Git получит изменения из удалённого репозитория, но не перезапишет несохранённые изменения в каталоге 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 ваши локальные изменения в подмодуле конфликтуют с какими-либо изменениями в удалённом репозитории, то Git вам сообщит об этом при обновлении подмодуля.

```
$ 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'
```

Для разрешения конфликта вы можете перейти в каталог подмодуля и сделать это как вы бы делали в обычной ситуации.

Публикация изменений подмодуля

Теперь у нас есть некоторые изменения в каталоге нашего подмодуля. Какие-то из них мы получили при обновлении из удалённого репозитория, а другие были сделаны локально и пока никому не доступны, так как мы их ещё никуда не отправили.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > Update setup script
  > Unicode support
  > Remove unnecessary method
  > Add new option for conn pooling
```

Если мы создадим коммит в основном проекте и отправим его на сервер, не отправив при этом изменения в подмодуле, то другие люди, которые попытаются использовать наши изменения, столкнутся с проблемой, так как у них не будет возможности получить требуемые изменения подмодуля. Эти изменения будут присутствовать только в нашей локальной копии.

Чтобы гарантированно избежать такой проблемы, вы можете попросить Git проверять отправлены ли изменения всех подмодулей до отправки изменений основного репозитория на сервер. Команда `git push` принимает аргумент `--recurse-submodules`, который может принимать значения либо «`check`», либо «`on-demand`». Использование значения «`check`» приведёт к тому, что `push` просто завершится ошибкой, если какой-то из 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 даёт 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 удалённые серверы, чтобы гарантировать доступность изменений другим людям, а затем попытать отправить изменения снова. Если вы хотите использовать поведение “check” при каждом выполнении `git push`, то его можно установить поведением по умолчанию, выполнив команду `git config push.recurseSubmodules check`.

Другой вариант — это использовать значение «`on-demand`», тогда Git попытается сделать всё вышеописанное 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 перешёл в каталог модуля DbConnector и отправил его изменения на сервер. Отправка изменений основного репозитория также завершится неудачей если во время отправки изменений подмодуля возникла ошибка. Установить такое поведение по умолчанию можно с помощью команды `git config push.recurseSubmodules on-demand`.

Слияние изменений подмодуля

Если вы измените ссылку на подмодуль одновременно с кем-то ещё, то вы можете столкнуться с некоторыми проблемами. Такое случается если истории подмодуля разошлись и они зафиксированы в разошедшихся ветках основного проекта, что может потребовать некоторых усилий для исправления.

Если один из коммитов является прямым предком другого (слияние может быть выполнено перемоткой вперёд), то Git просто выберет последний для выполнения слияния, то есть всё отработает хорошо.

Однако, Git не будет пытаться выполнить даже простейшего слияния. Если коммиты 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 «слияние последующих коммитов не найдено» («merge following commits not found»), что несколько обескураживает, но вскоре 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 коммитов разошедшихся веток истории. К счастью, получить эту информации довольно просто. Если выполнить команду `git diff`, то вы получите SHA-1 хеши коммитов обоих веток, которые вы пытаетесь слить.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

В данном примере `eb41d76` является **нашим** коммитом в подмодуле, а `c771610` — коммитом из вышестоящего репозитория. Если мы перейдём в каталог нашего подмодуля, то он должен быть на коммите `eb41d76`, так как операция слияния его не изменяла. Если по каким-то причинам это не так, то вы можете просто создать новую ветку из этого коммита и извлечь её.

Куда более важным является SHA-1 хеш коммита второй ветки истории. Именно его мы должны будем слить и разрешить конфликты. Вы можете попытаться либо просто выполнить слияние, указав непосредственно этот SHA-1 хеш, либо создать из него отдельную ветку и слить её. Мы предлагаем использовать последний вариант, хотя бы только потому, что сообщение коммита слияния получается более читаемым.

Итак, перейдите в каталог нашего подмодуля, создайте ветку «`try-merge`» 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

$ 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 конфликт слияния, поэтому если мы разрешим его и создадим коммит, то можно просто включить результирующий коммит в 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
```

① Во-первых, мы разрешили конфликт.

② Затем мы вернулись в каталог основного проекта.

③ Мы снова проверили SHA-1 хеши.

④ Обновили указатель на подмодуль с учётом разрешённого конфликта.

⑤ Зафиксировали наше слияние, создав новый коммит.

Это может немного запутать, но на самом деле здесь нет ничего сложного.

Интересно, что существует ещё один случай, с которым Git умеет работать. Если подмодуль содержит некий коммит слияния, который в своей истории содержит **оба** первоначальных коммита, то Git предложит его как возможное решение. Git видит, что в проекте подмодуля ветки, содержащие эти два коммита, уже когда-то кем-то сливались и этот коммит слияния, вероятно, именно то, что вам нужно.

Именно поэтому сообщение об ошибке выше было «merge following commits not found» — Git не мог найти такой коммит. Кто бы мог подумать, что Git **пытается** такое сделать?

Если Git нашёл только один приемлемый коммит слияния, 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.
```

Предлагаемая команда обновляет индекс таким же образом, как если бы вы выполнили `git add` (что очищает список конфликтов), а затем создали коммит. Хотя, наверное, вам не стоит так делать. Будет лучше просто перейти в каталог подмодуля, просмотреть изменения, сместить вашу ветку на этот коммит, должным образом всё протестировать и только потом создать коммит в основном репозитории.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forward to a common submodule child'
```

Это приводит к аналогичному результату, но таким способом вы получаете рабочий проверенный код в каталоге подмодуля.

Полезные советы

Существует несколько хитростей, которые могут немного упростить вашу работу с подмодулями.

Команда Foreach

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.

Для примера допустим, что мы хотим начать работу над какой-то новой функциональностью или исправить какую-то ошибку и наша работа затронет сразу несколько подмодулей. Мы можем легко припрятать 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 неё 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'
```

Вы уловили идею. Ещё одна полезная вещь, которую можно сделать с помощью этой команды — создать комплексную дельту изменений основного проекта и всех подмодулей.

```
$ 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_pager_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 её в основном проекте. Это, конечно, упрощённый 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 для большинства из них вы не можете задать значения по умолчанию. Мы рассмотрели настройку псевдонимов Git в разделе Псевдонимы в Git главы 2, однако ниже приведён пример, который вы наверняка захотите использовать, 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, для обновления подмодулей достаточно выполнить команду `git supdate`, а для отправки всех изменений с учётом подмодулей — команду `git spush`.

Проблемы с подмодулями

Однако, использование подмодулей не обходится без проблем.

Переключение веток

Например, переключение веток с подмодулями в них может оказаться довольно запутанным, особенно для версий Git старше 2.13. Если вы создадите новую ветку и добавите в ней подмодуль, а затем переключитесь обратно на ветку без подмодуля, то каталог подмодуля всё равно останется и будет неотслеживаемым:

```
$ git --version
git version 2.12.2

$ 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 'Add crypto library'
[add-crypto 4445836] Add 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 каталог не сложно, но может показаться странным, что он вообще присутствует. Если удалить каталог и снова переключиться на ветку с подмодулем, то потребуется выполнить `submodule update --init`, чтобы в нём появились файлы.

```
$ git clean -ffdx
Removing CryptoLibrary/

$ git checkout add-crypto
Switched to branch 'add-crypto'

$ ls CryptoLibrary/

$ git submodule update --init
Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile    includes    scripts     src
```

Опять же, это не сильно сложно, но может немного сбивать с толку.

В более новых версиях (Git >= 2.13) всё несколько упрощено за счёт поддержки командой `git checkout` параметра `--recurse-submodules`, который отвечает за приведение подмодулей в состояние, соответствующее извлекаемой ветке.

```
$ git --version
git version 2.13.3
```

```
$ 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 'Add crypto library'
[add-crypto 4445836] Add crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout --recurse-submodules master
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'.

nothing to commit, working tree clean
```

Использование параметра `--recurse-submodules` команды `git checkout` помогает при работе с несколькими ветками основного проекта, когда в каждой из них используется разное состояние подмодуля. Действительно, если вы переключитесь между ветками, в которых зафиксированы разные состояния подмодуля, то команда `git status` отметит подмодуль как изменённый («`modified`») и покажет наличие новых коммитов в нём. Это происходит из-за того, что по умолчанию состояние подмодуля не изменяется при переключении веток.

Такое поведение может сильно сбивать с толку, поэтому рекомендуется всегда использовать `git checkout --recurse-submodules` если в вашем проекте есть подмодули. В более старых версиях Git, в которых команда `git checkout` не поддерживает параметр `--recurse-submodules`, для приведения подмодулей в ожидаемое состояние используйте команду `git submodule update --init --recursive` сразу после извлечения ветки.

К счастью, вы можете указать Git (начиная с версии 2.14) всегда использовать параметр `--recurse-submodules` с помощью изменения конфигурации: `git config submodule.recurse true`. Как упоминалось выше, это заставит Git автоматически добавлять параметр `--recurse-submodules` ко всем командам (кроме `git clone`), которые его поддерживают, и рекурсивно обрабатывать подмодули.

Переход от подкаталогов к подмодулям

Другая большая проблема возникает при переходе от использования подкаталогов 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 каком-то каталоге вашего проекта и вы хотите переместить их в подмодуль. Если вы удалите подкаталог, а затем выполните

`submodule add`, то Git ругнётся на вас:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

Первым делом вы должны удалить каталог `CryptoLibrary` из индекса. Затем можно 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, в которой эти файлы всё ещё являются частью основного проекта, 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 всё же можете извлечь ветку принудительно, используя команду `checkout -f`, только убедитесь перед этим в отсутствии несохранённых изменений, так как они могут быть перезаписаны.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Если вернуться обратно на предыдущую ветку, то по какой-то причине каталог `CryptoLibrary` будет пустым, а команда `git submodule update` не сможет это исправить. Чтобы вернуть свои файлы, вам понадобится перейти в каталог подмодуля и выполнить команду `git checkout .`. Можно использовать эту команду вместе с `submodule foreach` для извлечения сразу всех подмодулей.

Важно отметить, что в современных версиях Git подмодули сохраняют все служебные данные в каталоге `.git` основного проекта, поэтому, в отличие от более старых версий Git,

удаление каталога подмодуля не приведёт к потере каких-либо коммитов или веток.

Все эти инструменты делают подмодули довольно простым и эффективным методом работы одновременно над несколькими тесно связанными, но разделёнными проектами.

Создание пакетов

Помимо рассмотренных ранее основных способов передачи данных Git по сети (HTTP, SSH и т. п.), существует ещё 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 всё, 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 клонировать репозиторий где-нибудь ещё, 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 `repo.bundle` и хотите поработать над этим проектом. Вы можете клонировать репозиторий из бинарного файла в каталог, почти также как вы делаете это при использовании URL.

```
$ git clone repo.bundle repo
Cloning into 'repo'...
...
$ cd repo
$ 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, включённую 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 repo
c99cf5b Fourth commit - second repo
7011d3d Third commit - second repo
9a466c5 Second commit
b1ec324 First commit
```

Во-первых, нам нужно определить диапазон коммитов, которые мы хотим включить в пакет. В отличие 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 коммитов, вы можете указать диапазон коммитов несколькими способами. Для того, чтобы получить три коммита из нашей основной ветки, которые отсутствовали в

изначально клонированной ветке, мы можем использовать запись вида `origin/master..master` или `master ^origin/master`. Вы можете проверить её с помощью команды `log`.

```
$ git log --oneline master ^origin/master
71b84da Last commit - second repo
c99cf5b Fourth commit - second repo
7011d3d Third commit - second repo
```

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` вернёт нечто подобное:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 Third commit - second repo
```

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 repo
| * 71b84da (other-master) Last commit - second repo
| * c99cf5b Fourth commit - second repo
| * 7011d3d Third commit - second repo
|/
* 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 придётся по-

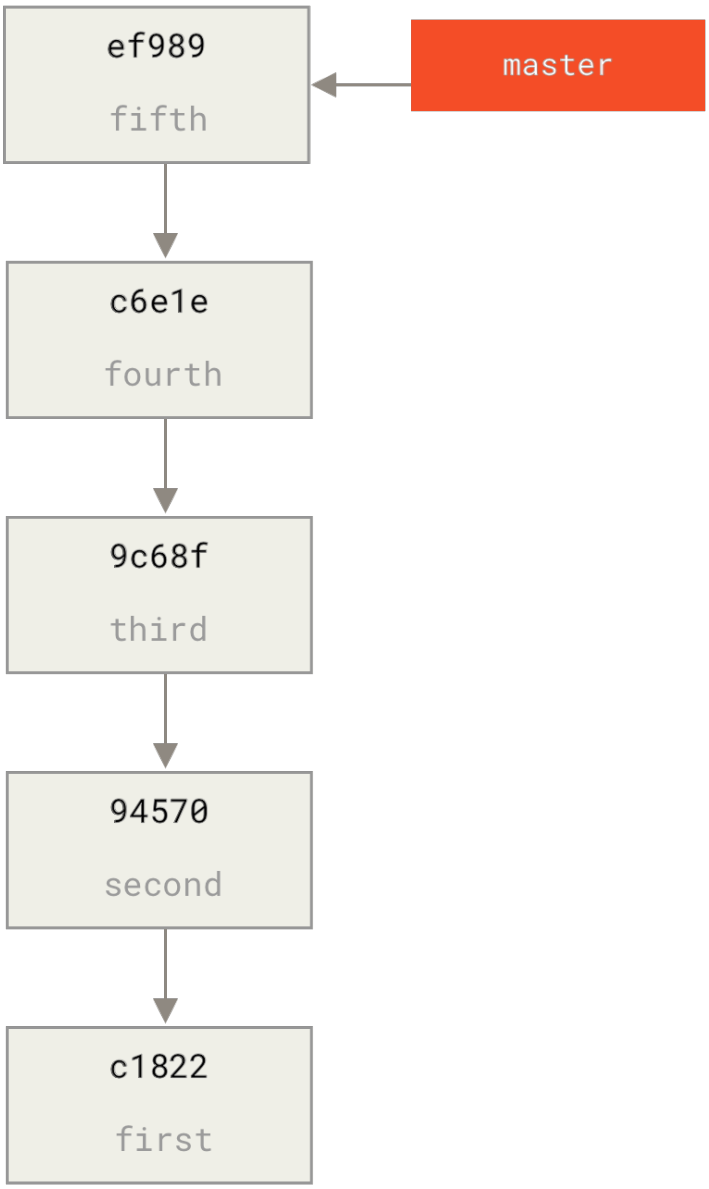
настоящему изменять каждый коммит в новой истории, как это вам бы потребовалось делать в случае обычного объединения историй (так как родословная коммитов влияет на SHA-1).

Давайте испробуем как это работает, возьмём существующий репозиторий и разобьём 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:

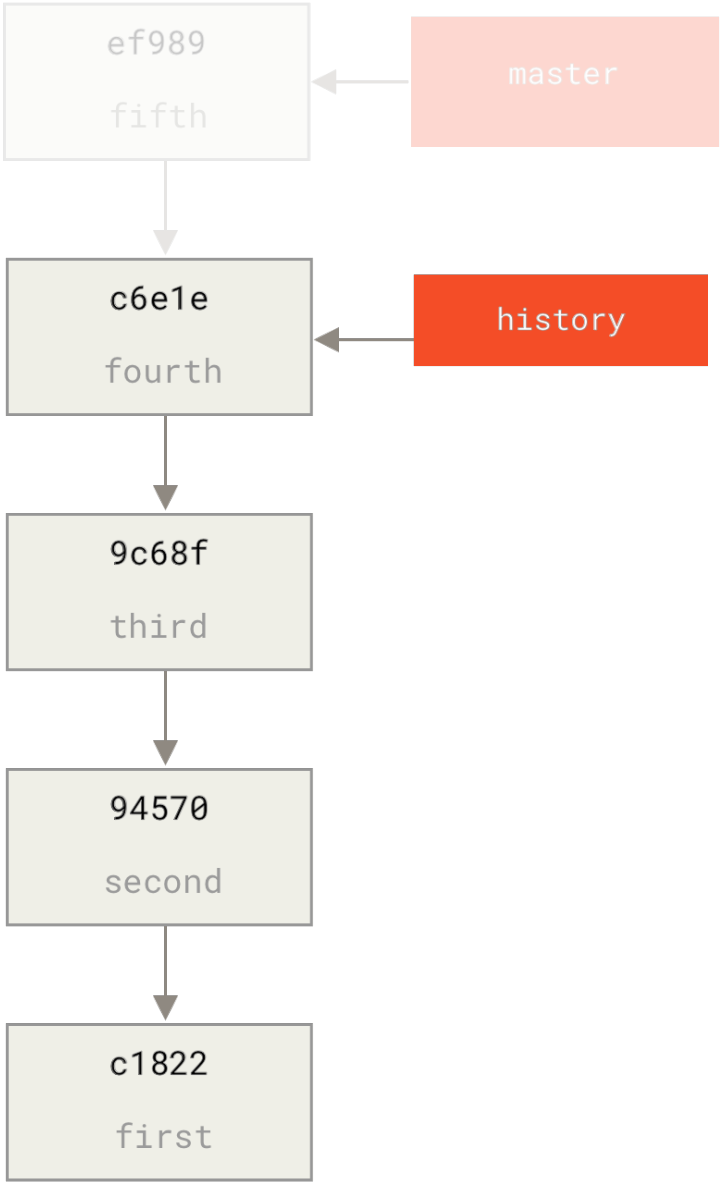
```
$ git log --oneline
ef989d8 Fifth commit
c6e1e95 Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

Мы хотим разбить его на два семейства историй. Одно семейство, которое начинается от первого коммита и заканчивается четвёртым, будет историческим. Второе, состоящее пока только из четвёртого и пятого коммитов — будет семейством со свежей историей.



Создать историческое семейство легко, мы просто создаём ветку с вершиной на нужном коммите и затем отправляем эту ветку как `master` в новый удалённый 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
```



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 займёмся более сложной частью — усечём свежую историю. Нам необходимо перекрытие, так чтобы мы смогли заменить коммит из одной части коммитом из другой, то есть мы будем обрезать историю, оставив четвёртый и пятый коммиты (таким образом четвёртый коммит будет входить в пересечение).

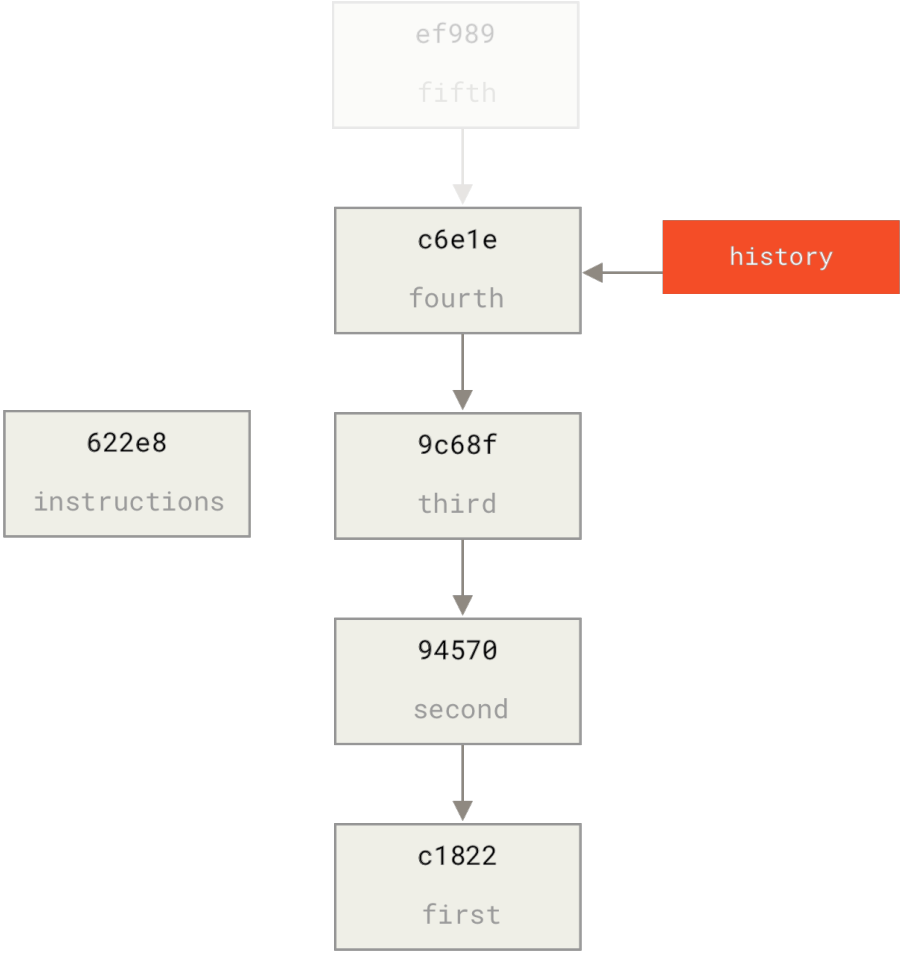
```
$ git log --oneline --decorate
ef989d8 (HEAD, master) Fifth commit
c6e1e95 (history) Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

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 (четвёртый 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 берёт дерево и возвращает SHA-1 объекта, представляющего новый сиротский коммит.

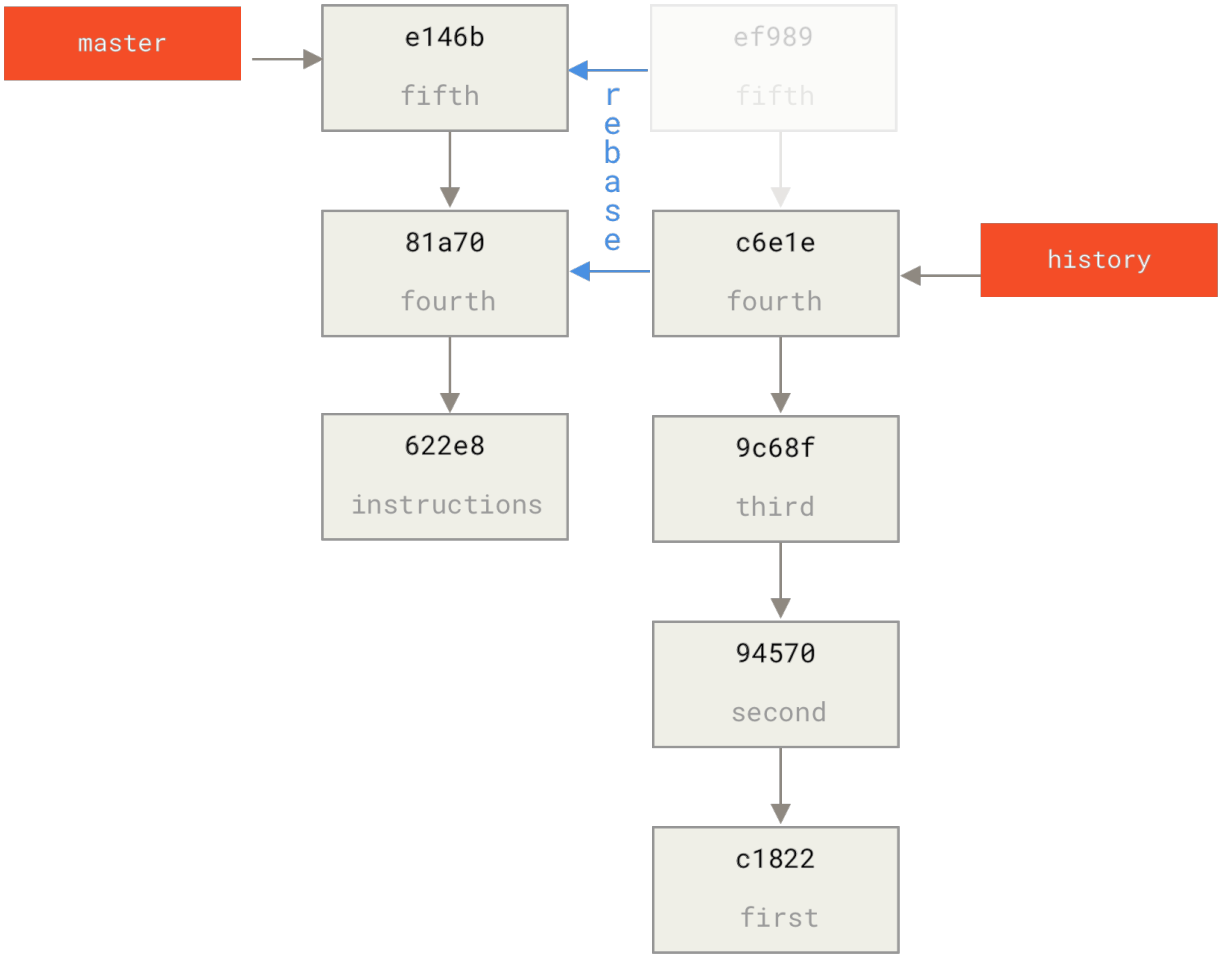
```
$ echo 'Get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

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.



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, который 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
```



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 клонировал проект и хочет получить полную историю. Для получения исторических данных после клонирования усечённого репозитория, ему нужно добавить в список удалённых 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
```

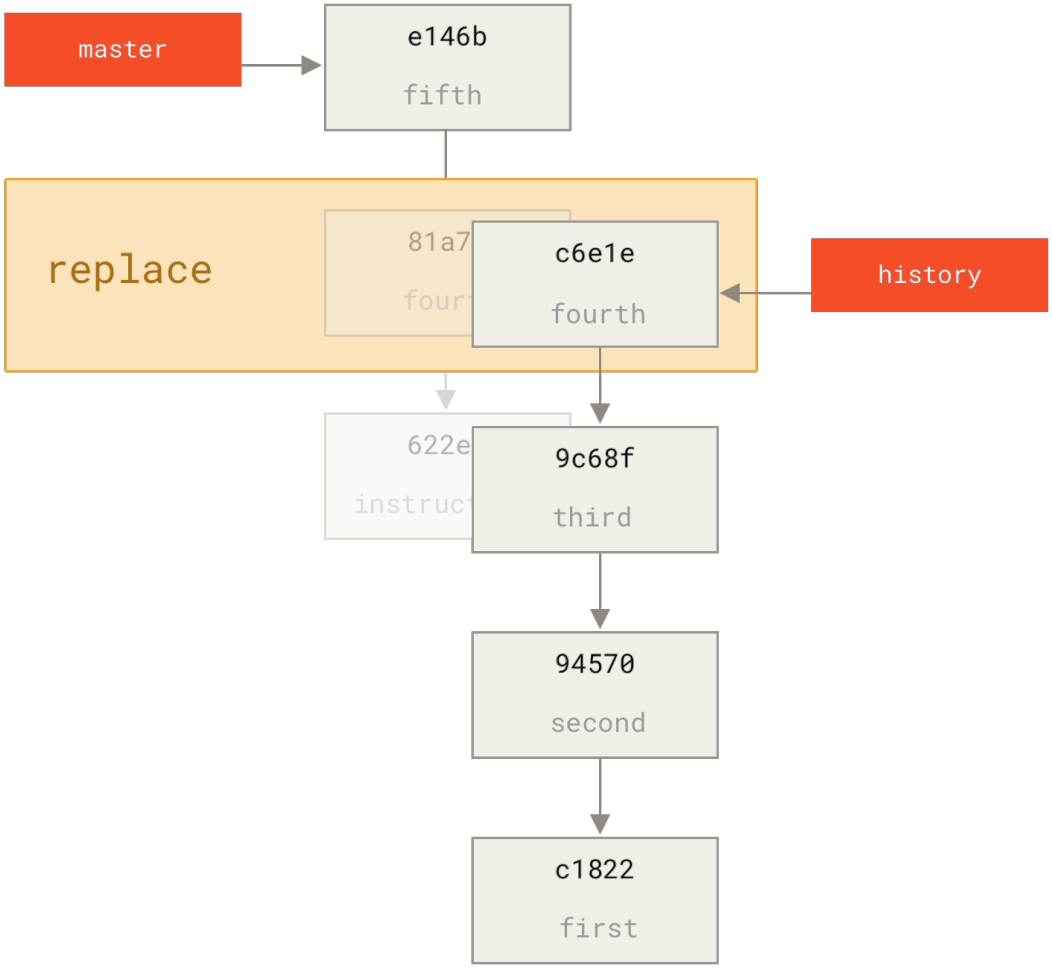
Для объединения этих веток вы можете просто вызывать `git replace`, указав коммит, который вы хотите заменить, и коммит, которым вы хотите заменить первый. Так мы хотим заменить «четвёртый» коммит в основной ветке «четвёртым» коммитом из ветки `project-history/master`:

```
$ 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.



Интересно, что для четвёртого коммита SHA-1 хеш выводится равный `81a708d`, хотя в действительности он содержит данные коммита `c6e1e95`, которым мы его заменили. Даже если вы выполните команду типа `cat-file`, она отобразит заменённые 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 произведённой 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/81a708dd0e167a3f691541c7a6463343bc457040
```

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 `replace` 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 учётных данных

Если для подключения к удалённым серверам вы используете протокол SSH, то вы можете использовать ключ вместо 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. Всё 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 учётными данными, которая может помочь в этом. В Git «из коробки» есть несколько опций:

- По умолчанию Git не кеширует учётные данные совсем. Каждое подключение будет запрашивать у вас логин и пароль.
- В режиме «cache» учётные данные сохраняются в памяти в течение определённого периода времени. Ни один из паролей никогда не сохраняется на диск и все они удаляются из кеша через 15 минут.
- В режиме «store» учётные 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 учётные 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 учётные данные хранятся в защищённом хранилище, привязанному к вашему системному аккаунту. В этом режиме учётные 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, то можете включить использование «Git Credential Manager» во время установки Git для Windows или установить крайнюю версию GCM как отдельный сервис. Он похож на «osxkeychain», описанный выше, но для управления секретной информацией использует Windows Credential Store. Его так же можно использовать в WSL1 и WSL2. Больше имнформации об установке и настройке GCM можно найти на странице проекта на GitHub.

Вы можете 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 учётными данный (по умолчанию используется `~/.git-credentials`). Помощник «cache» принимает опцию `--timeout <seconds>`, которая изменяет промежуток времени, в течение которого демон остаётся запущенным (по умолчанию «900», или 15 минут). Ниже приведён пример как вы можете настроить помощник «store» на использование определённого файла:

```
$ git config --global credential.helper 'store --file ~/.my-credentials'
```

Git позволяет настраивать сразу несколько помощников. При поиске учётных данных для конкретного сервера, Git будет по порядку запрашивать у них учётные данные и остановится при получении первого ответа. При сохранении учётных 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 учётными данными на флэш-диске, но, на случай его отсутствия, вы хотите дополнительно использовать кеширование в оперативной памяти.

```
[credential]
    helper = store --file /mnt/thumbdrive/.git-credentials
    helper = cache --timeout 30000
```

Под капотом

Как же это всё 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 нём сохранены учётные данные для `mygithost`. Ниже приведена рабочая сессия, в которой используется команда «fill», вызываемая Git при попытке найти учётные данные для сервера:

```
$ git credential fill ①
protocol=https ②
host=mygithost
③
protocol=https ④
host=mygithost
username=bob
password=s3cre7
```

```
$ git credential fill ⑤
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

① Это команда, которая начинает взаимодействие.

② После этого Git-credential ожидает данные из стандартного потока ввода. Мы передаём ему то, что знаем: протокол и имя сервера.

③ Пустая строка обозначает, что ввод закончен и система управления учётными данными должна ответить, что ей известно.

④ После этого Git-credential выполняет какую-то работу и выводит обнаруженную информацию.

⑤ Если учётные 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 учётными данными вызывает программы, отделённые от самого Git; какие и как зависит в том числе и от настроек `credential.helper`. Существует несколько вариантов вызова:

Настройки	Поведение
`foo`	Выполняется `git-credential-foo`
`foo -a --opt=bcd`	Выполняется `git-credential-foo -a --opt=bcd`
`/absolute/path/foo -xyz`	Выполняется `/absolute/path/foo -xyz`
`!f() { echo "password=s3cre7"; }; f`	Код после символа `!` выполняется в шелле

Итак, помощники, описанные выше на самом деле называются `git-credential-cache`, `git-credential-store` и т. д. и мы можем настроить их на приём 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 учётных данных в памяти помощника.
- `erase` удаляет учётные 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 введённой информации имеющуюся у него информацию. Вывод обрабатывается как набор операций присваивания; выведенные значения заменят те, что Git знал до этого.

Ниже приведён пример, используемый ранее, но вместо git-credential напрямую вызывается git-credential-store:

```
$ git credential-store --file ~/git.store store ①
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get ②
protocol=https
host=mygithost

username=bob ③
password=s3cre7
```

① Здесь мы просим `git-credential-store` сохранить некоторые учётные данные: логин «bob» и пароль «s3cre7», которые будут использоваться при доступе к `https://mygithost`.

② Теперь мы извлечём эти учётные данные. Мы передаём часть уже известных нам параметров подключения (`https://mygithost`) и пустую строку.

③ `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 учётные данные. Помощники `osxkeychain` и `wincred` используют форматы, лежащие в основе их хранилищ, а `cache` использует его собственный формат хранения во внутренней памяти (который другие процессы прочитать не могут).

Собственное хранилище учётных 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 распространённые варианты использования, но не все. Для примера допустим, что ваша команда имеет некоторые учётные данные, совместно используемые всей командой, например, для развёртывания. Эти данные хранятся в общедоступном каталоге, но вы не хотите копировать их в ваше собственное хранилище учётных 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 учётными 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 подойдёт любой язык, так как Git может использовать всё, что сможет запустить на выполнение. Ниже приведён полный исходный код нашего нового помощника авторизации:

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' ①
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' ②
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'] and user ==
known['username'] 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 цикле считываются данные из стандартного ввода, до тех пор пока не будет прочитана пустая строка. Введённые 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`, поместим его в один из каталогов, содержащихся в списке `PATH`, а так же 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 столкнётесь 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 операций, 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, а так же мы познакомились с некоторыми инструментами Git, которые делают его использование простым и эффективным. В этой главе мы рассмотрим некоторые настройки Git и систему хуков, что позволяет настроить поведение Git. Таким образом, вы сможете заставить Git работать именно так как нужно вам или вашей компании.

Конфигурация Git

В главе Введение кратко упоминалось, что вы можете настроить Git, используя команду `git config`. Первое, что вы делали, это установили своё имя и e-mail адрес:

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

Сейчас вы познакомитесь с несколькими наиболее интересными опциями, которые можно установить для настройки поведения Git.

Кратко: Git использует набор конфигурационных файлов для изменения стандартного поведения, если это необходимо. Вначале, Git ищет настройки в файле `/etc/gitconfig`, который содержит настройки для всех пользователей в системе и всех репозиториев. Если передать опцию `--system` команде `git config`, то операции чтения и записи будут производиться именно с этим файлом.

Следующее место, куда смотрит Git — это файл `~/.gitconfig` (или `~/.config/git/config`), который хранит настройки конкретного пользователя. Вы можете указать Git читать и писать в него, используя опцию `--global`.

Наконец, Git ищет параметры конфигурации в файле настроек в каталоге Git (`.git/config`) текущего репозитория. Эти значения относятся только к текущему репозиторию и доступны при передаче параметра `--local` команде `git config`. (Если уровень настроек не указан явно, то подразумевается локальный.)

Каждый из этих уровней (системный, глобальный, локальный) переопределяет значения предыдущего уровня, например, значения из `.git/config` важнее значений из `/etc/gitconfig`.

> Конфигурация Git это обычные текстовые файлы, поэтому можно вручную установить необходимые значения используя соответствующий синтаксис. Как правило, это проще чем вызывать команду `git config` для каждого параметра.

Базовая конфигурация клиента

Конфигурационные параметры Git разделяются на две категории: настройки клиента и настройки сервера. Большая часть — клиентские, для настройки ваших личных предпочтений в работе. Существует много, *очень много* настроек, но подавляющее

большинство из них применимо только в конкретных случаях; мы рассмотрим только самые основные и самые полезные из них. Для просмотра полного списка настроек, поддерживаемых вашей версией Git, выполните команду:

```
$ man git-config
```

Эта команда выведет список доступных настроек с довольно подробным описанием. Так же, соответствующую документацию можно найти здесь https://git-scm.com/docs/git-config.html.

`core.editor`

По умолчанию, Git использует ваш редактор по умолчанию (`$VISUAL` или `$EDITOR`), если значение не задано — переходит к использованию редактора `vi` при создании и редактировании сообщений коммитов или тегов. Чтобы изменить редактор по умолчанию, воспользуйтесь настройкой `core.editor`:

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

Теперь, вне зависимости от того, какой редактор является основным для вашего окружения, Git будет вызывать `Emacs` для редактирования сообщений.

`commit.template`

Если указать путь к существующему файлу, то он будет использован как сообщение по умолчанию при создании коммита. Смысл создания шаблона сообщения коммита в том, чтобы лишний раз напомнить себе (или другим) о требованиях к формату или стилю оформления сообщения коммита.

Например, предположим что вы создали файл `~/.gitmessage.txt`, который выглядит так:

```
Subject line (try to keep under 50 characters)

Multi-line description of commit,
feel free to be detailed.

[Ticket: X]
```

Обратите внимание, что шаблон напоминает коммитеру о том, чтобы строка заголовка сообщения была короткой (для поддержки однострочного вывода команды `git log --oneline`), что дополнительную информацию в сообщении следует располагать ниже, а так же о том, что было бы неплохо при наличии добавить ссылку на номер задачи или сообщения в системе отслеживания ошибок.

Чтобы заставить Git отображать содержимое этого файла в редакторе каждый раз при выполнении команды `git commit`, следует установить значение параметра `commit.template`:

```
$ git config --global commit.template ~/.gitmessage.txt
```

```
$ git commit
```

Теперь, при создании коммита, в вашем редакторе будет отображаться сообщение изменённого вида:

```
Subject line (try to keep under 50 characters)

Multi-line description of commit,
feel free to be detailed.

[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
```

Если ваша команда придерживается требований к сообщениям коммитов, то создание шаблона такого сообщения и настройка Git на его использование увеличит вероятность соответствия заданным требованиям.

`core.pager`

Данная настройка определяет какая программа будет использована для разбиения текста на страницы при выводе такой информации как `log` и `diff`. Вы можете указать `more` или любую другую (по умолчанию используется `less`), а так же выключить совсем, установив пустое значение:

```
$ git config --global core.pager ''
```

В таком случае, Git будет выводить весь текст полностью, вне зависимости от его длины.

`user.signingkey`

Если вы создаёте подписанные аннотированные теги (как описано в разделе Подпись главы 7), то установка GPG ключа в настройках облегчит вам задачу. Установить ключ можно следующим образом:

```
$ git config --global user.signingkey <gpg-key-id>
```

Теперь, вам не нужно указывать ключ для подписи каждый раз при вызове команды `git`

`tag`:

```
$ git tag -s <tag-name>
```

`core.excludesfile`

В разделе Игнорирование файлов главы 2 сказано, что вы можете указывать шаблоны исключений в файле `.gitignore` вашего проекта, чтобы Git не отслеживал их и не добавлял в индекс при выполнении команды `git add`.

Однако, иногда вам нужно игнорировать определённые файлы во всех ваших репозиториях. Если на вашем компьютере работает Mac OS X, вероятно вы знакомы с файлами `.DS_Store`. Если вы используете Emacs или Vim, то вы знаете про файлы, имена которых заканчиваются на `~` или `.swp`.

Данная настройка позволяет вам определить что-то вроде глобального файла `.gitignore`. Если вы создадите файл `~/.gitignore_global` с содержанием:

```
*~
.*.swp
.DS_Store
```

... и выполните команду `git config --global core.excludesfile ~/.gitignore_global`, то Git больше не потревожит вас на счёт этих файлов.

`help.autocorrect`

Если вы ошибётесь в написании команды, Git покажет вам что-то вроде этого:

```
$ git chekcout master
git: 'chekcout' is not a git command. See 'git --help'.

The most similar command is
    checkout
```

Git старается угадать, что вы имели в виду, но при этом команду не выполняет. Если вы установите `help.autocorrect` в значение 1, то Git будет выполнять эту команду:

```
$ 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...
```

Обратите внимание, что команда выполнилась через «0.1» секунды. `help.autocorrect` — это число, указываемое в десятых долях секунды. Поэтому, если вы установите значение 50, то Git даст вам 5 секунд изменить своё решение перед тем, как выполнить скорректированную

команду.

Цвета в Git

Git полностью поддерживает цветовой вывод в терминале, что позволяет быстро и легко визуально анализировать вывод команд. Существует несколько опций для настройки цветов.

`color.ui`

Git автоматически подсвечивает большую часть своего вывода, но это можно отключить, если вам не нравится такое поведение. Для отключения цветового вывода в терминал, выполните следующую команду:

```
$ git config --global color.ui false
```

Значение по умолчанию — `auto`, при котором цвета используются при непосредственном выводе в терминал, но исключаются при перенаправлении вывода в именованный канал или файл.

Вы так же можете установить значение `always`, что делает вывод одинаковым как в терминал, так и в именованный канал. Скорее всего, вам это не понадобится; в большинстве случаев, при желании использовать цвета в перенаправленном выводе, указывается флаг `--color` команде Git для принудительного использования цветовых кодов. Практически всегда стандартное значение подходит лучше всего.

`color.*`

Если вы хотите явно указать вывод каких команд должен быть подсвечен и как, Git предоставляет соответствующие настройки. Каждая из них может быть установлена в значения `true`, `false` или `always`:

```
color.branch
color.diff
color.interactive
color.status
```

Каждая из них имеет вложенную конфигурацию, которую можно использовать для настройки отдельных частей вывода при желании переопределить их цвет. Например, чтобы установить для метаинформации вывода команды diff синий цвет, чёрный фон и полужирный шрифт, выполните команду:

```
$ git config --global color.diff.meta "blue black bold"
```

Для установки цвета доступны следующие значения: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, или `white`. Для указания атрибутов текста, как `bold` в предыдущем примере, доступны значения: `bold`, `dim`, `ul` (подчёркнутый), `blink` и `reverse` (поменять местами цвет

фона и цвет текста).

Внешние программы слияния и сравнения

Хоть в Git и есть встроенная программа сравнения, которая описывается в этой книге, вы можете установить вместо неё другую. Вы также можете настроить графический инструмент разрешения конфликтов слияния вместо того, чтобы разрешать конфликты вручную. Мы покажем как настроить Perforce Visual Merge Tool (P4Merge) для разрешения конфликтов слияния, так как это прекрасный и бесплатный инструмент.

Если у вас есть желание попробовать P4Merge, то она работает на всех основных платформах, так что у вас должно получиться. В примерах мы будем использовать пути к файлам, которые работают в системах Linux и Mac; для Windows вам следует изменить `/usr/local/bin` на путь к исполняемому файлу у вас в системе.

Для начала скачайте P4Merge. Затем, создайте скрипты обёртки для вызова внешних программ. Мы будем использовать путь к исполняемому файлу в системе Mac; в других системах — это путь к файлу `p4merge`. Создайте скрипт с названием `extMerge` для вызова программы слияния и передачи ей заданных параметров:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

Скрипт вызова программы сравнения проверяет наличие 7 аргументов и передаёт 2 из них в скрипт вызова программы слияния. По умолчанию, Git передаёт следующие аргументы программе сравнения:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Так как вам нужны только `old-file` и `new-file`, следует использовать скрипт, который передаст только необходимые параметры.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Так же следует убедиться, что созданные скрипты могут исполняться:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Теперь можно изменить файл конфигурации для использования ваших инструментов слияния и сравнения. Для этого необходимо изменить ряд настроек: `merge.tool` — чтобы сказать Git какую стратегию использовать, `mergetool.<tool>.cmd` — чтобы сказать Git как

запускать команду, `mergetool.<tool>.trustExitCode` — чтобы сказать Git как интерпретировать код выхода из программы, `diff.external` — чтобы сказать Git какую команду использовать для сравнения. Таким образом, команду конфигурации нужно запустить четыре раза:

```
$ 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
```

или вручную отредактировать файл `~/.gitconfig` добавив соответствующие строки:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

После этого, вы можете запускать команды diff следующим образом:

```
$ git diff 32d1776b1^ 32d1776b1
```

Вместо отображения вывода diff в терминале Git запустит P4Merge, выглядеть это будет примерно так:

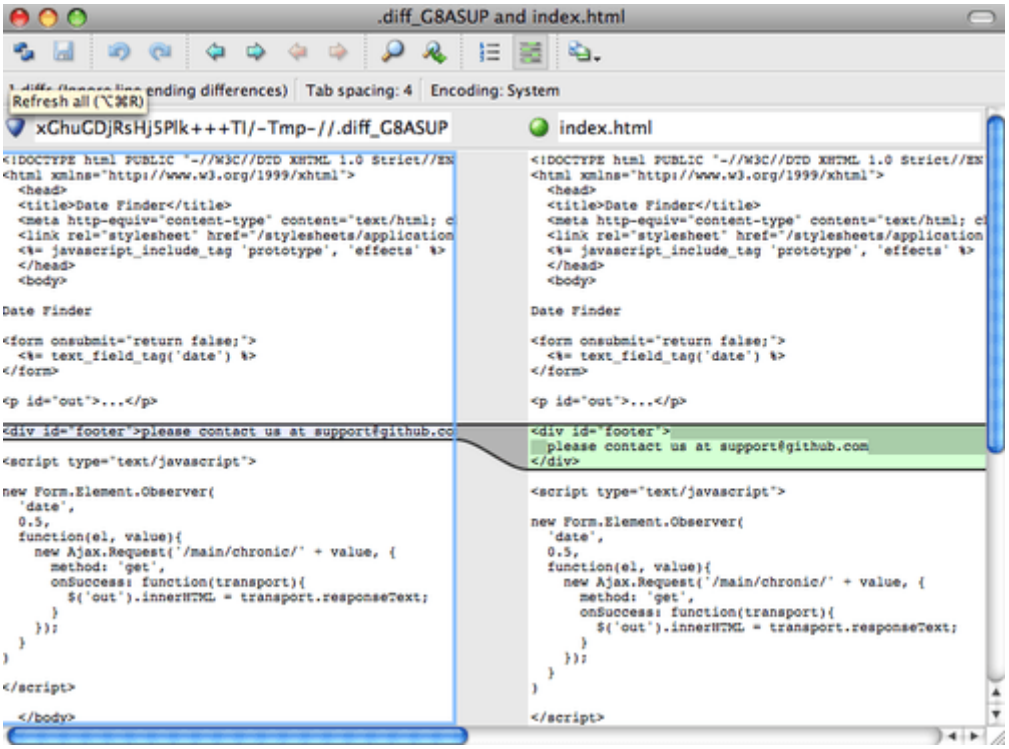


Рисунок 142. P4Merge

Если при слиянии двух веток у вас возникнут конфликты, выполните команду `git mergetool`; она запустит P4Merge чтобы вы могли разрешить конфликты используя графический интерфейс.

Используя скрипт обёртку для вызова внешних программ, вы можете легко изменить вызываемую программу. Например, чтобы начать использовать KDiff3 вместо P4Merge, достаточно изменить файл `extMerge`:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Теперь, Git будет использовать программу KDiff3 для сравнения файлов и разрешения конфликтов слияния.

Git изначально настроен на использование ряда других инструментов для разрешения конфликтов слияния, поэтому вам не нужно дополнительно что-то настраивать. Для просмотра списка поддерживаемых инструментов, выполните команду:

```
$ 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.
```

Если вы хотите использовать KDiff3 только для разрешения конфликтов слияния, но не для сравнения, выполните команду:

```
$ git config --global merge.tool kdiff3
```

Если выполнить эту команду вместо настройки использования файлов `extMerge` и `extDiff`, то Git будет использовать KDiff3 для разрешения конфликтов слияния, а для сравнения — стандартную программу diff.

Форматирование и пробелы

Проблемы форматирования и пробелов являются одними из самых неприятных и незаметных проблем, с которыми сталкивают разработчики при совместной работе, особенно используя разные платформы. Это легко может произойти с патчами или с любой другой совместной работой, так как редакторы молча исправляют несоответствия, и если ваши файлы когда либо касаются систем Windows, то переносы строк могут быть заменены. В Git есть несколько настроек, чтобы справиться с этими проблемами.

`core.autocrlf`

Если вы программируете в Windows и работаете с людьми, которые не используют её (или наоборот), рано или поздно, вы столкнётесь с проблемами переноса строк. Это происходит потому, что Windows при создании файлов использует для обозначения переноса строки два символа «возврат каретки» и «перевод строки», в то время как Mac и Linux используют только один — «перевод строки». Это незначительный, но невероятно раздражающий факт кроссплатформенной работы; большинство редакторов в Windows молча заменяют переносы строк вида LF на CRLF или вставляют оба символа, когда пользователь нажимает клавишу ввод.

Git может автоматически конвертировать переносы строк CRLF в LF при добавлении файла в индекс и наоборот — при извлечении кода. Такое поведение можно включить используя настройку `core.autocrlf`. Если у вас Windows, то установите значение `true` — при извлечении кода LF окончания строк будут преобразовываться в CRLF:

```
$ git config --global core.autocrlf true
```

Если у вас система Linux или Mac, то вам не нужно автоматически конвертировать переносы строк при извлечении файлов; однако, если файл с CRLF окончаниями строк случайно попал в репозиторий, то Git может его исправить. Можно указать Git конвертировать CRLF в LF во время коммита, но не наоборот, установив настройке `core.autocrlf` значение `input`:

```
$ git config --global core.autocrlf input
```

Такая конфигурация позволит вам использовать CRLF переносы строк в Windows, при этом в репозитории и системах Mac и Linux будет использован LF.

Если вы используете Windows и программируете только для Windows, то вы можете отключить описанный функционал задав значение `false`, сохраняя при этом CR символы в

репозитории:

```
$ git config --global core.autocrlf false
```

core.whitespace

Git поставляется настроенным на обнаружение и исправление некоторых проблем с пробелами. Он в состоянии найти шесть основных проблем, обнаружение трёх из них включено по умолчанию, а трёх других — выключено.

Те, что включены по умолчанию — это `blank-at-eol`, что ищет пробелы в конце строки; `blank-at-eof`, что ищет пробелы в конце файла; и `space-before-tab`, что ищет пробелы перед символом табуляции в начале строки.

Те, что выключены по умолчанию — это `indent-with-non-tab`, что ищет строки с пробелами вначале вместо символа табуляции (и контролируется настройкой `tabwidth`); `tab-in-indent`, что ищет символы табуляции в отступах в начале строки; и `cr-at-eol`, которая указывает Git на валидность наличия CR в конце строки.

Указав через запятую значения для настройки `core.whitespace`, можно сказать Git какие из этих опций должны быть включены. Чтобы отключить ненужные проверки, достаточно удалить их из строки значений или поставить знак `-` перед каждой из них. Например, чтобы включить все проверки, кроме `space-before-tab`, выполните команду (при этом `trailing-space` является сокращением и охватывает как `blank-at-eol`, так и `blank-at-eof`):

```
$ git config --global core.whitespace \
    trailing-space,-space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

Или можно указать только часть проверок:

```
$ git config --global core.whitespace \
    -space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

Git будет искать указанные проблемы при выполнении команды `git diff` и пытаться подсветить их, чтобы вы могли исправить их перед коммитом. Так же эти значения будут использоваться во время применения патчей командой `git apply`. При применении патчей, можно явно указать Git информировать вас в случае нахождения проблем с пробелами:

```
$ git apply --whitespace=warn <patch>
```

Так же можно указать Git автоматически исправлять эти проблемы перед применением патча:

```
$ git apply --whitespace=fix <patch>
```

Эти настройки так же применяются при выполнении команды `git rebase`. Если проблемные пробелы попали в коммит, но ещё не отправлены в удалённую ветку, можно выполнить `git rebase --whitespace=fix` для автоматического исправления этих проблем.

Конфигурация сервера

Для серверной части Git не так много настроек, но есть несколько интересных, на которые стоит обратить внимание.

`receive.fsckObjects`

Git способен убедиться, что каждый объект, отправленный командой `push`, валиден и соответствует своему SHA-1-хешу. По умолчанию эта функция отключена; это очень дорогая операция и может привести к существенному замедлению, особенно для больших объёмов отправляемых данных или для больших репозиториев. Вы можете включить проверку целостности объектов для каждой операции отправки, установив значение `receive.fsckObjects` в `true`:

```
$ git config --system receive.fsckObjects true
```

Теперь, Git будет проверять целостность репозитория до принятия новых данных для уверенности, что неисправные или злонамеренные клиенты не смогут отправить повреждённые данные.

`receive.denyNonFastForwards`

Если вы перебазируете коммиты, которые уже отправлены, и попытаетесь отправить их снова или попытаетесь отправить коммит в удалённую ветку, в которой не содержится коммит, на который она указывает, то данные приняты не будут. В принципе, это правильная политика; но в случае перебазирования — вы знаете, что делаете и можете принудительно обновить удалённую ветку используя флаг `-f` для команды `push`.

Для запрета перезаписи истории установите `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

Сделать то же самое можно другим способом — используя хук на стороне сервера, мы рассмотрим его немного позже. Этот подход позволяет более гибко настроить ограничения, например, запретить перезапись истории определённой группе пользователей.

`receive.denyDeletes`

Политику `denyNonFastForwards` можно обойти, удалив ветку и создав новую с таким же именем. Для предотвращения этого, установите `receive.denyDeletes` в значение `true`:

```
$ git config --system receive.denyDeletes true
```

Эта команда запретит удаление веток и тегов всем пользователям. Чтобы удалить ветку, придётся удалить все соответствующие ей файлы на сервере вручную. Куда более интересный способ — это настроить права пользователей, с ним вы познакомитесь в разделе Пример принудительной политики Git.

Атрибуты Git

Некоторые из настроек могут быть применены к каталогу, поэтому Git применяет их только к подкаталогам или набору файлов. Настройки, зависящие от пути, называются атрибутами и могут быть установлены либо в файле `.gitattributes` в любом из каталогов проекта (обычно, в корневом каталоге), либо в файле `.git/info/attributes`, если вы не хотите хранить их в репозитории вместе с вашим проектом.

Используя атрибуты, вы можете настраивать различные стратегии слияния для отдельных файлов или каталогов вашего проекта, указать Git как сравнивать бинарные файлы, настраивать фильтры добавления или извлечения данных из репозитория. В этом разделе вы узнаете о некоторых атрибутах, которые можно установить для заданных путей в вашем проекте и рассмотрите несколько практических примеров.

Бинарные файлы

Интересная возможность атрибутов Git, которая может вам пригодиться, заключается в указании файлов, которые следует считать бинарными (особенно, когда определить это не представляется возможным), и определении инструкций как именно Git должен обрабатывать эти файлы. Например, некоторые текстовые файлы могут генерироваться автоматически и отследить изменения в них невозможно, в то же время, отследить изменения некоторых бинарных файлов вполне возможно. Далее вы увидите как объяснить Git где какой файл.

Идентификация бинарных файлов

Некоторые файлы выглядят как текстовые, но работать с ними нужно как с бинарными данными. Например, проекты Xcode на macOS содержат файл с расширением `.pbxproj`, который по сути является набором данных JSON (текстовое представление данных JavaScript), сохранённым на диск IDE и содержащим различные настройки проекта. Технически — это текстовый файл (потому как все данные в UTF-8), фактически — легковесная база данных; отслеживание изменений в нём бесполезно, а слияние изменений — сделанных двумя людьми — невозможно, поэтому вы вряд ли захотите обрабатывать его как текстовый файл. Файл предназначен для чтения машиной, а не человеком. Вот почему его следует рассматривать как бинарный.

Чтобы Git начал обрабатывать все `pbxproj` файлы как бинарные, добавьте в файл `.gitattributes` следующую строку:

```
*.pbxproj binary
```

Теперь, Git не будет конвертировать или исправлять CRLF в этом файле; не будет пытаться

определить изменения или выводить их на экран при выполнении команд `git show` или `git diff`.

Сравнение бинарных файлов

Возможности атрибутов Git так же можно использовать для эффективного сравнения бинарных файлов. Сделать это можно указав Git каким образом конвертировать бинарные данные в текстовый формат, чтобы затем сравнить их обычным способом.

Этот подход решает одну из самых досадных проблем, известных человечеству: контроль версий документов Microsoft Word. Все уже знают, что Word — самый ужасный редактор из всех, но, как ни странно, продолжают его использовать. Если вы хотите контролировать версии документов Word, то поместите их в Git репозиторий и периодически делайте коммиты; но как это решает проблему? Если вы просто выполните команду `git diff`, то увидите следующее:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

Нельзя просто взять и сравнить два файла не читая их и не сравнивая вручную, так? Это становится возможным при использовании атрибутов Git. Добавьте в файл `.gitattributes` следующую строку:

```
*.docx diff=word
```

Это говорит Git, что нужно использовать фильтр «word» при просмотре изменений файлов, соответствующих шаблону `.docx`. Что такое «word» фильтр? Вам следует его настроить. Нужно сконфигурировать Git на использование программы `docx2txt` для конвертации документов Word в текстовые, которые можно корректно сравнивать.

Для начала, нужно установить программу `docx2txt`; вы можете скачать её здесь https://sourceforge.net/projects/docx2txt. Следуйте инструкциям из файла `INSTALL` для её установки и настройки запуска из командной строки. Затем следует написать скрипт обёртку для конвертации вывода программы в формат, понятный Git. Создайте файл `docx2txt` в любом доступном для запуска месте и добавьте в него следующее содержимое (прим. пер.: применимо для Linux и Mac):

```
#!/bin/bash
docx2txt.pl "$1" -
```

Не забудьте добавить права запуска для созданного файла. Наконец, настройте Git на использование созданного скрипта:

```
$ git config diff.word.textconv docx2txt
```

Теперь Git знает, что при сравнении двух коммитов, имена файлов в которых заканчиваются на `.docx`, он должен обработать эти файлы с помощью фильтра «word», который определён как программа `docx2txt`. Это позволяет автоматически генерировать текстовые версии файлов Word и только потом их сравнивать.

Для примера, первый раздел этой книги был сохранён в формате Word и добавлен в Git репозиторий. Затем был добавлен новый абзац. Вот что покажет команда `git diff`:

```
$ 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
by explaining some background on version control tools, then move on to how to get Git
running on your system and finally how to get it setup to start working with. At the
end of this chapter you should understand why Git is around, why you should use it and
you should be all setup to do so.
 1.1. About Version Control
 What is "version control", and why should you care? Version control is a system that
records changes to a file or set of files over time so that you can recall specific
versions later. For the examples in this book you will use software source code as the
files being version controlled, though in reality you can do this with nearly any type
of file on a computer.
+Testing: 1, 2, 3.
 If you are a graphic or web designer and want to keep every version of an image or
layout (which you would most certainly want to), a Version Control System (VCS) is a
very wise thing to use. It allows you to revert files back to a previous state, revert
the entire project back to a previous state, compare changes over time, see who last
modified something that might be causing a problem, who introduced an issue and when,
and more. Using a VCS also generally means that if you screw things up or lose files,
you can easily recover. In addition, you get all this for very little overhead.
 1.1.1. Local Version Control Systems
 Many people's version-control method of choice is to copy files into another
directory (perhaps a time-stamped directory, if they're clever). This approach is very
common because it is so simple, but it is also incredibly error prone. It is easy to
forget which directory you're in and accidentally write to the wrong file or copy over
files you don't mean to.
```

Git кратко сообщает нам, что была добавлена строка «Testing: 1, 2, 3.». Решение не идеальное — изменения форматирования не отображаются — но это работает.

Проблему сравнения файлов изображений можно решить аналогичным образом. Один из способов реализации заключается в передаче изображения на фильтр для извлечения EXIF информации — метаданных, которые сохраняются для большинства форматов

изображений. Скачав и установив программу `exiftool`, вы сможете использовать её для конвертации изображений в текстовые метаданные, изменение которых будет являться текстовым представлением изменений изображений. Добавьте следующую строку в ваш файл `.gitattributes`:

```
*.png diff=exif
```

Настройте Git на использование этой программы:

```
$ git config diff.exif.textconv exiftool
```

Заменив картинку и выполнив команду `git diff`, вы увидите что-то похожее:

```
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
```

Легко заметить, что размеры изображения и файла изменились.

Разворачивание ключевых слов

Разработчики часто хотят использовать разворачивание ключевых слов в стиле SVN или CVS. Основная проблема в Git — это невозможность изменять файлы с информацией о коммите после его совершения, так как Git сначала вычисляет контрольную сумму. Однако, вы можете добавить текст в файл после извлечения и убрать его перед добавлением файла в коммит. Атрибуты Git позволяют это сделать двумя способами.

Для начала, вы можете автоматически добавлять SHA-1-хеш объекта в поле `$Id$`. Если установить этот атрибут для одного или нескольких файлов, то каждый раз при извлечении ветки Git будет заменять это поле на SHA-1-хеш объекта. Важно заметить, что это SHA-1-хеш не коммита, а самого объекта. Добавьте следующую строку в ваш файл `.gitattributes`:

```
*.txt ident
```

Добавьте ссылку на `$Id$` в тестовый файл:

```
$ echo '$Id$' > test.txt
```

При последующих извлечениях Git будет добавлять SHA-1-хеш объекта:

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Однако, этот результат имеет ограниченное применение. Если вы использовали подстановку ключевых слов в CVS или Subversion, то вы могли включать метку времени, а SHA-1-хеш не так полезен, потому что вы не можете сказать какой из двух хешей старше, а какой новее просто взглянув на них.

Оказывается, вы можете написать свои фильтры для выполнения подстановок в файлах в момент коммита/извлечения. Эти фильтры называются «clean» и «smudge». В файле `.gitattributes` вы можете установить фильтр для конкретных путей, а затем указать скрипты для обработки файлов при извлечении («smudge», смотри Фильтр «smudge» применяется при извлечении) и при индексировании («clean», смотри Фильтр «clean» применяется при индексации). С помощью этих фильтров можно делать всевозможные операции.

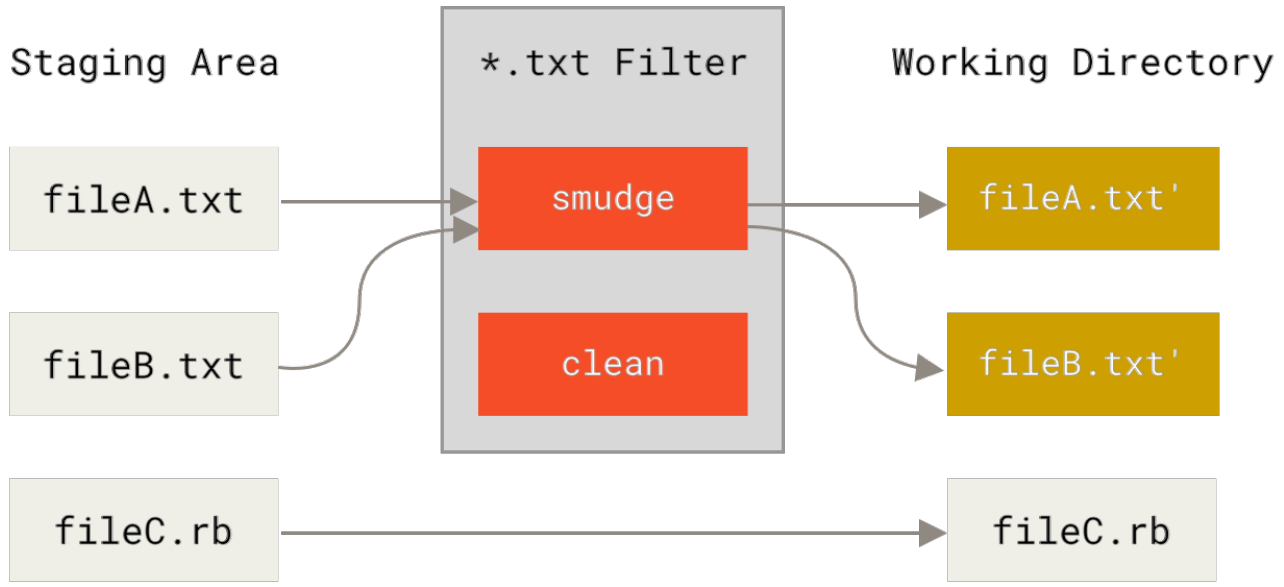


Рисунок 143. Фильтр «smudge» применяется при извлечении

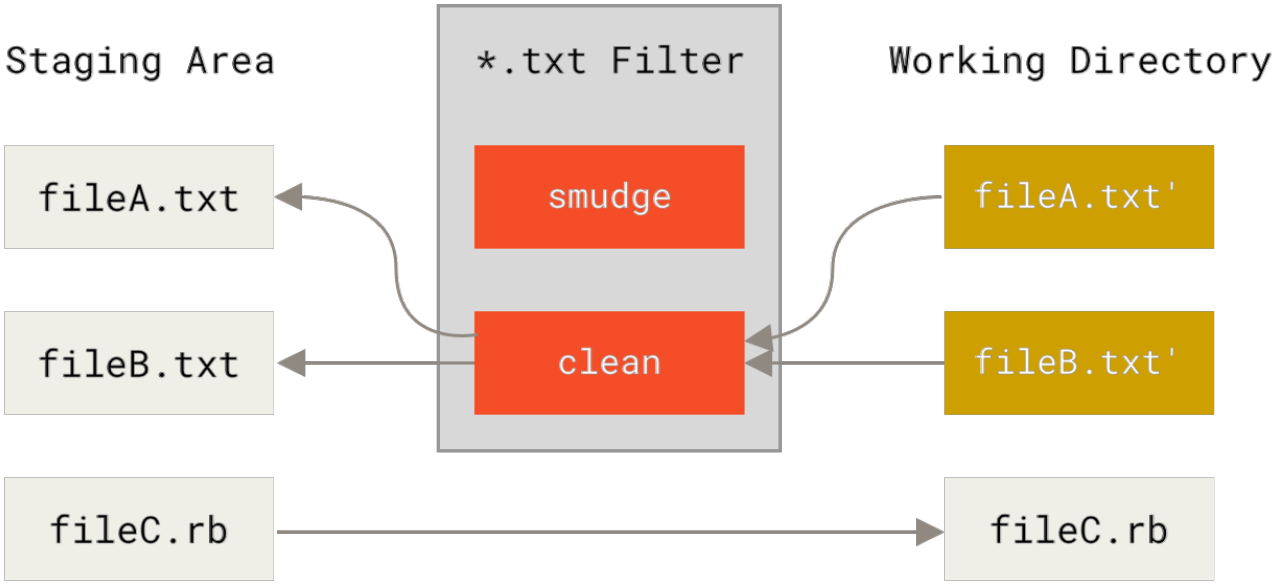


Рисунок 144. Фильтр «clean» применяется при индексации

Исходное сообщение коммита является простым примером как передать весь ваш код на C программе `indent` перед коммитом. Это можно настроить путём указания фильтра «indent» в файле `.gitattributes` для файлов `*.c`.

```
*.c filter=indent
```

Затем скажите Git что должен делать фильтр «indent» на стадиях smudge и clean:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

В таком случае, Git будет обрабатывать файлы программой `indent` все файлы по маске `*.c` перед тем, как добавить их в индекс; и наоборот, обрабатывать эти же файлы программой `cat` при их извлечении. По сути, программа `cat` ничего не делает: она возвращает те же данные, что и получает на вход. Указанная комбинация позволяет эффективно обрабатывать файлы с исходным кодом на C программой `indent` перед коммитом.

Другой интересный пример — это подстановка ключевого слова `$Date$` в стиле системы контроля ревизий. Чтобы правильно это реализовать, вам нужен простой скрипт, который получает имя файла, определяет дату последнего коммита и вставляет её в файл. Ниже приведён небольшой пример такого скрипта на Ruby:

```
#! /usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Всё, что делает скрипт — это получает дату последнего коммита с помощью команды `git`

`log`, заменяет результатом все подстроки `$Date$` и возвращает итоговый результат; вы можете написать аналогичный скрипт на любом языке. Назовите файл со скриптом, например, `expand_date` и сохраните в каталоге с программами. Теперь, нужно настроить Git фильтр (назовите его `dater`) и укажите ему использовать ваш скрипт `expand_date` при извлечении файлов. Вместе с этим, мы будем использовать регулярное выражение Perl для очистки перед коммитом:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\$/"'
```

Указанная Perl команда очищает любое значение в строке, где она видит `$Date$`, чтобы вернуть файл в изначальное состояние. Теперь фильтр готов и вы можете проверить его добавив ключевое слово `$Date$` в файл и настроив Git атрибут, чтобы для вашего файла применялся созданный фильтр:

```
date*.txt filter=dater
```

```
$ echo '# $Date$' > date_test.txt
```

Если добавить в коммит последние изменения, а затем извлечь файл, то вы увидите корректную подстановку ключевого слова:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Test date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Как вы могли заметить, описанный подход предоставляет большие возможности. Однако, вам стоит быть осторожным, так как файл `.gitattributes` включается в коммит и распространяется вместе с проектом, а драйвер (в данном случае `dater`) нет, поэтому он не будет работать везде. Учитывайте это при разработке фильтров оставляя возможность работы без них — так вы сохраните проект в рабочем состоянии.

Экспорт репозитория

Атрибуты Git так же позволяют вам делать некоторые интересные вещи при экспорте вашего проекта.

`export-ignore`

Вы можете указать Git игнорировать определённые файлы и каталоги при создании архива. Если в вашем проекте есть файл или каталог, которые вам нужны, но вы не хотите включать их в архив при экспорте, то можно присвоить им атрибут `export-ignore`.

Например, у вас есть несколько файлов в каталоге `test/` и совершенно нет смысла включать их в архив вашего проекта. В этом случае достаточно добавить следующую строку в файл `.gitattributes`:

```
test/ export-ignore
```

Теперь, при создании архива проекта командой `git archive`, каталог `test/` не будет включен в архив.

`export-subst`

При создании архива так же доступна подстановка по ключевым словам и поддерживается использование форматирования `git log` для файлов, отмеченных атрибутом `export-subst`.

Например, если вы хотите добавить в проект файл с именем `LAST_COMMIT`, в который при экспорте будут автоматически сохраняться метаданные последнего коммита, то измените файлы `.gitattributes` и `LAST_COMMIT` следующим образом:

```
LAST_COMMIT export-subst
```

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

Теперь, при создании архива проекта командой `git archive`, в него будет включён файл со следующим содержанием:

```
$ git archive HEAD | tar xCf ../deployment-testing -
$ cat ../deployment-testing/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

В качестве подстановок можно использовать, например, сообщение коммита и `git notes`, а `git log` может делать простой перенос слов:

```
$ echo '$Format:Last commit: %h by %aN at %cd%n%+w(76,6,9)%B$' > LAST_COMMIT
$ git commit -am 'export-subst uses git log'\''s custom formatter

git archive uses git log'\''s `pretty=format:` processor
directly, and strips the surrounding `$Format:` and `$`
markup from the output.
'
$ git archive @ | tar xfO - LAST_COMMIT
Last commit: 312ccc8 by Jim Hill at Fri May 8 09:14:04 2015 -0700
       export-subst uses git log's custom formatter
```

```
        git archive uses git log's `pretty=format:` processor directly, and
        strips the surrounding `$Format:` and `$` markup from the output.
```

Полученный архив подходит для использования, но как и любой экспортированный архив не годится для дальнейшей разработки.

Стратегии слияния

Используя атрибуты Git можно применять разные стратегии слияния для разных типов файлов вашего проекта. Одна из полезных опций — это указать Git не сливать изменения в определённых файлах в случае конфликта, при этом использовать вашу версию файла.

Это полезно в случае, когда ветка разошлась или у вас специализированная ветка, при этом вы хотите иметь возможность сливать изменения, но игнорировать определённые файлы. Предположим, что у вас есть файл с настройками базы данных `database.xml`, содержимое которого в разных ветках отличается, при этом вы хотите сливать изменения из другой ветки не меняя этот файл. Для этого нужно добавить следующий атрибут:

```
database.xml merge=ours
```

А затем определить фиктивную стратегию слияния `ours`, выполнив команду:

```
$ git config --global merge.ours.driver true
```

Если вы сольёте изменения из другой ветки, то вместо конфликта слияния для файла `database.xml` вы увидите что-то вроде этого:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

В этом случае файл `database.xml` всегда остаётся неизменным.

Хуки в Git

Как и многие другие системы контроля версий, Git предоставляет возможность запуска пользовательских скриптов в случае возникновения определённых событий. Такие действия называются хуками и разделяются на две группы: серверные и клиентские. Если хуки на стороне клиента запускаются такими операциями как слияние или создание коммита, то на стороне сервера они инициируются сетевыми операциями, такими как получение отправленного коммита. Хуки часто используются для широкого круга задач.

Установка хука

Хуки хранятся в подкаталоге `hooks` относительно основного каталога Git. Для большинства

проектов это `.git/hooks`. Когда вы инициализируете новый репозиторий командой `git init`, Git наполняет каталог `hooks` примерами скриптов, большинство из которых готовы к использованию, при этом каждый из них содержит документацию по используемым входным данным. Все примеры представлены в виде шелл скриптов, содержащими код на Perl, но вы можете использовать любой язык для написания скриптов — главное правильно именовать исполняемые файлы. Если вы решите использовать какой-либо из предустановленных скриптов, то достаточно его просто переименовать, убрав суффикс `.sample`.

Для подключения собственного скрипта достаточно задать ему соответствующее имя, поместить в подкаталог `hooks` основного каталога Git и сделать его исполняемым. Далее, мы рассмотрим наиболее часто используемые хуки.

Клиентские Хуки

Для клиента существует множество различных хуков. В этой главе они разделены на хуки уровня коммита, уровня e-mail и прочие.

Необходимо отметить, что клиентские хуки **НЕ** копируются при клонировании репозитория. Если вы намерены использовать такие скрипты для обеспечения соблюдения политики, то вам следует использовать серверные хуки; например Пример принудительной политики Git.

Хуки уровня коммита

Первые четыре хука работают во время создания коммитов.

Первым запускается `pre-commit` хук, до того как вы напечатаете сообщение коммита. Он используется для проверки данных перед созданием коммита и позволяет увидеть если вы что-то забыли, запустить тесты, или выполнить другую необходимую проверку кода. Создание коммита будет отменено если выполнение хука завершится с кодом отличным от нуля. Пропустить выполнение хука можно с помощью `git commit --no-verify`. С помощью этого хука можно проверять стиль кода (запустить `lint` или аналог), проверять наличие пробелов в конце строк (именно это делает стандартный хук) или проверять наличие документации для новых методов.

Хук `prepare-commit-msg` запускается до вызова редактора сообщения коммита, но после создания стандартного сообщения. Это позволяет вам изменить стандартное сообщение коммита до того, как автор коммита увидит его. Хук принимает несколько параметров: путь к файлу, содержащему сообщение коммита, тип коммита и SHA-1-хеш, если текущий коммит является исправлением существующего. Для обычных коммитов этот хук бесполезен, однако находит своё применение для коммитов, где сообщение генерируется автоматически, например, для сообщений на основе шаблонов, коммитов слияния, сжимаемых и исправляемых коммитов. Его можно использовать для программного заполнения шаблона коммита необходимой информацией.

Хук `commit-msg` принимает один параметр — путь к временному файлу, содержащему указанное разработчиком сообщение коммита. Если скрипт завершается с ненулевым

кодом, то Git отменяет создание коммита, поэтому вы можете использовать этот хук для валидации состояния проекта или сообщения коммита до того как он будет создан. В последнем разделе этой главы мы покажем как использовать этот хук для проверки сообщения коммита на соответствие заданному шаблону.

Хук `post-commit` запускается после того, как коммит создан. Он не принимает никаких параметров, но вы можете легко получить информацию о последнем коммите выполнив `git log -1 HEAD`. Обычно, этот скрипт используется для уведомлений или чего-то подобного.

Хуки для рабочего процесса на основе E-mail

Для рабочего процесса на основе e-mail на стороне клиента можно задать три хука. Все они вызываются командой `git am`, поэтому если вы не используете её в своём рабочем процессе, то можете смело перейти к следующему разделу. Если вы получаете по почте патчи, подготовленные командой `git format-patch`, то найдёте здесь немного полезной информации.

В первую очередь запускается хук `applypatch-msg`. Он принимает единственный аргумент: имя временного файла, содержащее предлагаемое сообщение коммита. Git отменит патч если этот скрипт завершится с ненулевым кодом. Этот хук можно использовать для проверки формата сообщения или для его нормализации, если ваш скрипт умеет редактировать сообщение коммита.

Следующим запускается хук `pre-applypatch`. Здесь всё немного запутанно: хук запускается *после* применения патча, но перед созданием коммита, что позволяет проверить состояние кода до создания коммита. В этот момент можно запустить тесты или другим способом проверить состояние проекта. Если что-то пропущено или тесты не пройдены, скрипт должен завершиться с ненулевым кодом, что остановит выполнение команды `git am`, а коммит не будет создан.

Последним запускается хук `post-applypatch`, который вызывается уже после того как коммит создан. Вы можете его использовать для уведомления группы или автора патча о его применении. С помощью этого хука вы не можете прервать процесс применения патча.

Прочие хуки на стороне клиента

Хук `pre-rebase` выполняется при попытке перебазирования и может остановить процесс вернув ненулевой код. Его можно использовать для запрета перебазирования уже отправленных коммитов. Git устанавливается с примером такого скрипта, однако он делает некоторые допущения, которые могут не соответствовать вашему рабочему процессу.

Хук `post-rewrite` запускается командами, которые заменяют коммиты: `git commit --amend` и `git rebase` (но не `git filter-branch`). Его единственный аргумент — команда, которая инициировала перезапись, а список перезаписанных изменений передаётся через `stdin`. Его применение практически аналогично хукам `post-checkout` и `post-merge`.

После успешного выполнения `git checkout` запускается хук `post-checkout`; его можно использовать для настройки рабочего каталога в соответствии с требованиями проекта. Например, перемещение в рабочий каталог больших бинарных файлов, которые не должны отслеживаться, автогенерация документации и тому подобное.

Хук `post-merge` запускается после успешного выполнения команды `merge`. Его можно использовать для восстановления данных в рабочем каталоге, которые Git не может отслеживать, такие как права доступа. Так же этот хук может проверять наличие внешних по отношению к Git файлов, которые вы захотите скопировать при внесении изменений.

Хук `pre-push` выполняется во время работы команды `git push`: после обновления удалённых ссылок, но до непосредственной отправки данных. Он принимает название и путь удалённого репозитория как параметры, а список изменений для отправки через `stdin`. Его можно использовать для валидации набора изменений до их реальной отправки (ненулевой код отменяет отправку изменений).

Время от времени, как часть нормальной работы, Git выполняет сборку мусора вызовом команды `git gc --auto`. Хук `pre-auto-gc` вызывается непосредственно перед выполнением операции сборки мусора и может быть использован для уведомления о её запуске или для её отмены, если сейчас не самое подходящее для этого время.

Хуки на сервере

В дополнение к хукам на стороне клиента, как системный администратор вы можете использовать несколько важных хуков на сервере для вашего проекта, тем самым обеспечив выполнение практически любой политики. Эти скрипты выполняются до и после отправки на сервер. Pre-хуки могут возвращать ненулевой код в любой момент, что отменит передачу и отправит сообщение об ошибке клиенту; таким образом вы можете реализовать сколь угодно сложную политику.

`pre-receive`

Хук `pre-receive` запускается первым при старте получения данных от клиента. Он получает на `stdin` список отправленных изменений и если завершается ненулевым кодом, то ни одно из них принято не будет. Этот хук можно использовать для того, чтобы убедиться что все изменения можно применить методом перемотки вперёд, а так же для проверки прав доступа.

`update`

Хук `update` очень похож на `pre-receive`, за исключением того, что он выполняется для каждой ветки, которую отправитель пытается обновить. Если отправитель пытается отправить изменения в несколько веток, то `pre-receive` хук будет вызван однократно, а `update` выполнен для каждой изменяемой ветки. Вместо чтения из `stdin`, хук принимает три аргумента: название ссылки (ветка), SHA-1-хеш, на который указывала ссылка до отправки, и SHA-1-хеш коммита, отправляемого пользователем. Если скрипт завершается ненулевым кодом, то отклоняются все изменения только для текущей ветки, при этом изменения для других веток всё ещё могут быть применены.

`post-receive`

Хук `post-receive` вызывается после окончания всего процесса и может быть использован для обновления других сервисов или уведомления пользователей. Он принимает на `stdin` те же данные, что и хук `pre-receive`. Использовать его можно, например, для e-mail рассылки, для уведомления сервера непрерывной интеграции или обновления системы управления

задачами — разобрав сообщение коммита, можно определить необходимость создания, изменения или закрытия каких либо задач. Этот хук не может прервать процесс, но клиент остаётся подключённым пока он не завершится, поэтому избегайте выполнения длительных операций.

Если вы пишете сценарий/хук, который другие должны будут прочитать, используйте длинные версии параметров командной строки; через шесть месяцев вы будете нас благодарить.

Пример принудительной политики Git

В этом разделе вы сможете применить полученные знания для создания рабочего процесса Git, при котором будет проверяться формат сообщения коммита и определённым пользователям будет разрешено изменять содержимое заданных каталогов проекта. Вы создадите клиентские скрипты, которые помогут разработчикам понять, когда их изменения будут отклонены, а также серверные скрипты, которые обеспечат выполнение заданных политик.

Скрипты, которые будут приведены ниже, написаны на Ruby; отчасти по причине нашей интеллектуальной инерции, но также и потому, что Ruby легко читать, даже если вы не пишите на нём. К слову, любой язык будет работать — все примеры хуков, распространяемые с Git, написаны на Perl или Bash; с ними вы можете ознакомиться, посмотрев примеры.

Серверный Хук

На стороне сервера вся работа производится в файле `update` из каталога `hooks`. Хук `update` запускается однократно для каждой отправляемой ветки и принимает три параметра:

- Ссылка на ветку, в которую производится отправка
- Текущая ревизия ветки назначения
- Отправляемая ревизия

Так же можно получить имя пользователя, производящего отправку, если действия выполняются по протоколу SSH. Если вы настроили аутентификацию по публичному ключу используя одного пользователя (например, «git»), то вам потребуется использовать дополнительную обёртку командной оболочки, чтобы определить реального пользователя по его публичному ключу и правильно установить переменную окружения `$USER`. Далее предполагается, что переменная `$USER` уже содержит имя подключившегося пользователя, поэтому скрипт `update` начинается со сбора необходимой информации:

```
#!/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]})"
```

Да, здесь используются глобальные переменные. Не судите строго — это самый простой способ демонстрации.

Проверка формата сообщения коммита

Ваша первая задача — сделать так, чтобы каждый коммит соответствовал заданному формату. Предположим, что сообщение каждого коммита должно содержать строку вида «ref: 1234», так как вы хотите связать каждый коммит с соответствующим элементом в вашей системе управления задачами. Для этого вам понадобится проверять каждый получаемый коммит, искать в сообщении заданную подстроку и, в случае её отсутствия в сообщении любого из коммитов, прекращать обработку с ненулевым кодом, что приведёт к отклонению отправки целиком.

Вы можете получить список SHA-1 значений всех отправляемых коммитов передав значения `$newrev` и `$oldrev` низкоуровневой команде Git под названием `git rev-list`. В действительности, это команда `git log`, которая по умолчанию выводит только список значений SHA-1 и ничего больше. Поэтому, для получения списка SHA-1 хешей коммитов, находящихся между двумя заданными, вам следует выполнить, например, следующую команду:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Для каждого SHA-1 хеша из полученного результата можно получить соответствующее сообщение коммита и с помощью регулярного выражения проверить наличие искомой подстроки.

Осталось выяснить как получить сообщение коммита, зная его SHA-1 хеш. Чтобы получить содержимое коммита, следует использовать другую низкоуровневую команду `git cat-file`. Более детально мы рассмотрим эти низкоуровневые команды в главе Git изнутри; а сейчас покажем, что эта команда вам даёт:

```
$ 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

Change the version number
```

Самый простой способ извлечь сообщение коммита — это найти первую пустую строку и взять всё, что идёт после неё. В системах Unix и Mac это можно сделать с помощью программы `sed`:

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
Change the version number
```

Вы можете использовать эту магическую команду для извлечения сообщения отправляемого коммита и прерывать проверку в случае, когда что-то не соответствует. Для прерывания выполнения скрипта и отклонения отправки используйте ненулевой код возврата. Полностью функция выглядит следующим образом:

```
$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
```

Размещение указанного кода в скрипте `update` приведёт к отклонению всех обновлений, в которых содержатся один или несколько коммитов с сообщением, которое не соответствует вашему правилу.

Контроль доступа по списку имён пользователей

Предположим, вы хотите применить механизм контроля доступа на основе списков контроля доступа, позволяющий определённым пользователям вносить изменения в определённые части вашего проекта. К примеру, некоторые пользователи имеют полный доступ, а другие могут изменять только определённые каталоги проекта или отдельные файлы. Для реализации этого, следует записать эти правила в файл `acl`, находящийся в репозитории на сервере. Затем обновить хук `update`, чтобы он использовал эти правила при просмотре списка файлов в отправляемых коммитах для определения наличия прав доступа ко всем этим файлам у отправляющего пользователя.

Первое, что надо сделать — это создать список контроля доступа. Здесь следует использовать формат, который очень похож на CVS и представляет собой список строк, в каждой из которых первое поле имеет значение `avail` или `unavail`, второе поле содержит список пользователей, разделённых запятой, а третье поле — это путь к файлу или каталогу, для которого применяется это правило (пустое значение подразумевает отсутствие

ограничения). В качестве разделителя для этих полей применяется вертикальная черта (|).

В случае, когда у вас есть группа администраторов, несколько технических писателей с доступом к каталогу `doc` и один разработчик, у которого есть доступ только к каталогам `lib` и `tests`, файл со списком контроля доступа будет выглядеть так:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Для начала нужно прочитать эти данные и сформировать структуры для дальнейшего использования. С целью упрощения здесь мы используем только директивы `avail`. Ниже представлен метод, который возвращает ассоциативный массив, в котором ключом является имя пользователя, а значением — массив путей, к которым пользователь имеет доступ на запись.

```
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
```

Для представленного ранее файла списка контроля доступа этот метод вернёт следующую структуру данных:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Теперь, когда вопрос с правами доступа решён, необходимо извлечь список путей, изменения по которым присутствуют в отправляемых коммитах, чтобы убедиться в наличии доступа к ним у отправляющего пользователя.

Список файлов одного коммита можно легко получить, используя опцию `--name-only` команды `git log` (кратко рассматривалось в Главе 2):

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

Если воспользоваться структурой данных, полученной методом `get_acl_access_data`, и проверить соответствие путей из каждого коммита на соответствие ей, то можно определить наличие прав доступа у пользователя на отправку всех коммитов:

```
# 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
```

В результате, вы получаете список отправляемых коммитов командой `git rev-list`. Затем для каждого коммита извлекаете список файлов и проверяете наличие прав доступа у отправляющего пользователя на их изменение.

Теперь ваши пользователи не смогут отправить коммиты с плохо оформленными сообщениями или содержащие изменения в файлах, находящихся за пределами заданных путей.

Тестирование

Если выполнить `chmod u+x .git/hooks/update` для файла, в который вам следует поместить весь приведённый выше код, и попытаетесь отправить плохо оформленный коммит, то получите приблизительно следующее сообщение:

```
$ 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'
```

Здесь стоит обратить внимание на несколько интересных моментов. Первое — это момент начала работы хука.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Как вы помните, эти строки вы выводите в самом начале скрипта `update`. Всё, что ваш скрипт выводит в `stdout`, будет передано клиенту.

Второе, на что следует обратить внимание, это сообщение об ошибке.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

Первая строка — это ваше сообщение, две другие добавляет Git сообщая, что скрипт `update` завершился с ненулевым кодом, что привело к отклонению отправки. Ну и наконец, у вас есть вот это:

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Здесь можно увидеть сообщение об отказе для каждой из веток, которые отклонил ваш хук, при этом будет явно указано, что именно он является причиной отказа.

Более того, если кто-то отредактирует файл, на изменение которого у него нет прав, и попытается отправить содержащий это изменение коммит, то получит аналогичное сообщение. Например, если технический писатель попытается отправить коммит, содержащий изменения в каталоге `lib`, то он увидит следующее:

```
[POLICY] You do not have access to push to lib/test.rb
```

С момента как скрипт `update` существует и исполняем, ваш репозиторий не будет содержать коммиты с сообщением, неудовлетворяющем установленному шаблону, а доступ ваших пользователей будет ограничен.

Клиентские хуки

Недостатком этого подхода является неизбежное нытьё ваших пользователей, к которому приводит отклонение отправки коммитов. Получение отказа в последний момент при отправке тщательно продуманной работы может сильно расстроить и вызвать непонимание; кроме этого, придётся ещё актуализировать историю, что не всегда для слабонервных.

Решением в данной ситуации является предоставление клиентских хуков, которые пользователи могут использовать для получения уведомлений, когда они делают то, что сервер скорее всего отклонит. Таким образом они могут исправить любые проблемы до создания коммита и до того, как исправление проблемы станет гораздо сложнее. Так как хуки не копируются при клонировании репозитория, вам следует распространять их каким-то другим способом, а ваши пользователи должны будут их скопировать в каталог `.git/hooks` и сделать исполняемыми. Вы можете хранить эти хуки внутри проекта или в отдельном проекте — в любом случае Git не установит их автоматически.

Для начала, необходимо проверять сообщение коммита непосредственно перед его созданием, так вы будете уверены, что сервер не отклонит ваши изменения из-за плохо оформленных сообщений. Это реализуется созданием `commit-msg` хука. Если читать файл, переданный в качестве первого аргумента, и сравнивать его содержимое с заданным шаблоном, то можно заставить Git отменять создание коммита в случае отсутствия совпадения:

```
#!/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
```

Если скрипт на месте (`.git/hooks/commit-msg`) и исполняем, а вы создаёте коммит с плохо

оформленным сообщением, то увидите следующее:

```
$ git commit -am 'Test'
[POLICY] Your message is not formatted correctly
```

В этом случае коммит создан не будет. Однако, если ваше сообщение соответствует заданному шаблону, то коммит будет создан:

```
$ git commit -am 'Test [ref: 132]'
[master e05c914] Test [ref: 132]
 1 file changed, 1 insertions(+), 0 deletions(-)
```

Далее, следует убедиться, что внесённые изменения соответствуют вашим правам доступа. Если в каталоге `.git` содержится файл списка контроля доступа, который использовался ранее, то следующий `pre-commit` скрипт поможет вам реализовать такую проверку:

```
#!/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
    if !has_file_access
      puts "[POLICY] You do not have access to push to #{path}"
      exit 1
    end
  end
end

check_directory_perms
```

Этот скрипт практически такой же как и серверный, за исключением двух важных отличий. Во первых, файл списка контроля доступа находится в другом месте, так как скрипт запускается из рабочего каталога, а не из каталога `.git`. Поэтому необходимо изменить путь к файлу с:

```
access = get_acl_access_data('acl')
```

на следующий:

```
access = get_acl_access_data('.git/acl')
```

Второе отличие состоит в способе получения списка изменённых файлов. Если на сервере метод извлекает его из истории коммитов, то в данный момент на стороне клиента коммит ещё не создан, поэтому извлекать этот список необходимо из индекса. Вместо

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

следует использовать:

```
files_modified = `git diff-index --cached --name-only HEAD`
```

Вот и все отличия — в остальном скрипт работает одинаково. Так же предполагается, что локально скрипт будет запускаться от имени того же пользователя, что и на удалённом сервере. Если имя вашего локального пользователя не совпадает с именем пользователя на сервере, то следует задать значение переменной `$user` вручную.

Ещё одна вещь, которую можно здесь сделать, это убедиться, что пользователь не отправляет ветки, которые не могут быть обновлены простым смещением вперёд. Чтобы создать такую ситуацию, вам нужно либо перебазировать уже отправленный коммит, либо попытаться отправить другую локальную ветку в ту же удалённую.

Предположим, что на сервере уже включены опции `receive.denyDeletes` и `receive.denyNonFastForwards` для обеспечения политики, поэтому воспроизвести ситуацию можно только перебазировав отправленные коммиты.

Ниже представлен скрипт `pre-rebase`, который выполняет такую проверку. Он получает список коммитов, которые вы собираетесь перезаписать, и проверяет их наличие в удалённых ветках. Если хотя бы один из этих коммитов будет доступен из какой-либо удалённой ветки, то процесс перебазирования прерывается.

```
#!/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}"
      exit 1
    end
  end
end
```

Скрипт использует синтаксис, который не был рассмотрен в разделе Выбор ревизии главы 7. Получить список коммитов, которые уже были отправлены, можно с помощью команды:

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

Синтаксис `SHA^@` позволяет получить список всех родителей коммита. Вы ищите любой коммит, который доступен относительно последнего коммита на удалённом сервере, но недоступен относительно любого родителя отправляемых SHA-1, что определяет простое смещение вперёд.

Основной недостаток этого подхода в том, что он может быть очень медленным и не всегда необходим — если вы не форсируете отправку опцией `-f`, то сервер отклонит её с соответствующим предупреждением. Тем не менее, это интересная задача, которая теоретически может вам помочь избежать перебазирования, которое в будущем, возможно, придётся исправлять.

Заключение

Мы рассмотрели большинство основных способов настройки клиента и сервера Git с тем, чтобы он был максимально соответствовал вашим рабочим процессам и проектам. 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 только можно вообразить.

Git и другие системы контроля версий

Наш мир несовершенен. Как правило, вы не можете быстро перевести свой проект на использование Git. 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 в качестве клиента для работы с проектом, размещённом в другой системе контроля версий.

В какой-то момент, вы, возможно, захотите перевести свой существующий проект на Git. Во второй части главы вы узнаете о том, как провести миграцию в Git из некоторых специфических систем, а также познакомитесь с методом, который будет работать в нестандартных ситуациях, когда готовых инструментов миграции не существует.

Git как клиент

Git оказывает 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 систему контроля версий. Для этого разработан целый ряд специальных адаптеров, называемых «мостами» («bridges»). Здесь мы рассмотрим те, с которыми вы, скорее всего, столкнётесь при работе над реальными проектами.

Git и Subversion

Весомая часть проектов разработки с открытым исходным кодом, равно как и огромное количество корпоративных проектов, до сих пор используют Subversion (SVN) для управления исходным кодом. Он существует уже более десяти лет и большую часть этого времени был *де-факто* единственной системой контроля версий для проектов с открытым исходным кодом. Он также во многом похож на CVS, своего предка — «крёстного отца» всех современных систем управления версиями.

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 всю функциональность 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 коммиты 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 «DVCS», 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 коммитов. 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 коммитов. 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. `revprops`; самый простой способ сделать это — добавить скрипт `pre-revprop-change`, всегда возвращающий 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Теперь вы можете синхронизировать репозиторий на локальной машине, вызвав `svnsync init`, передав входной и выходной репозитории:

```
$ svnsync init file:///tmp/test-svn \
  http://your-svn-server.example.org/svn/
```

Наконец (SVN вам ещё не надоел?), можно запустить саму синхронизацию. Затем можно будет клонировать 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 клонирует 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/branches/my-calc-branch, 75
Found branch parent: (refs/remotes/origin/my-calc-branch)
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
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 коммитов — но Git вынужден последовательно скачивать SVN ревизии и превращать их в Git коммиты 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 каталогов с транком, ветками и тегами. Если же каталоги 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 каталогов» 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 тегами:

```
$ 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
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda 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, полученный коммит 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 коммитов. Это одна из причин, почему не рекомендуется смешивать Subversion и Git сервер в одном проекте. Если посмотреть на последний коммит, 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 коммиты 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
c80b6127dd04f5fcda218730ddf3a2da4eb39138 M  README.txt
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,
using rebase:
:100644 100644 65536c6e30d263495c17d781962cfff12422693a
b34372b25ccf4945fe5658fa381b075045e7702a M  README.txt
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,
using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18
e757b59a9439312d80d5d43bb65d4a7d0389ed6d M  autogen.sh
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 коммита 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 коммиты наверх.

```
$ git svn rebase
    M   autogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (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 ваш рабочий каталог не должен содержать незафиксированных изменений. Если это не так, вам следует либо «припрятать» (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 любого коммита 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 историй пройдёт успешно, за исключением того момента, что при просмотре истории вы заметите, что коммита из ветки `experiment` не были переписаны один за другим; вместо этого они схлопнулись в один коммит слияния.

Когда кто-нибудь клонирует 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 коммиты.

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, 90
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/opera` 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 `opera`.

Переключение активных веток

Git определяет ветку, в которую он отправит ваши коммиты 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 `opera`, в которой вы можете поработать отдельно, можете выполнить:

```
$ git branch opera remotes/origin/opera
```

Теперь, если вы захотите слить ветку `opera` в `trunk` (`master`), вы сможете сделать это с помощью обычной команды `git merge`. Однако вам потребуется добавить подробное описание к коммиту (через параметр `-m`), иначе при слиянии комментарий будет иметь вид «Merge branch opera» 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 коммитом, имеющим 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 `opera`) после того, как вы вольёте её в trunk.

Команды Subversion

В `git svn` содержится несколько команд для облегчения перехода на Git путём предоставления схожей с Subversion функциональности. Ниже приведены несколько команд, которые дают вам то, что вы имели в Subversion.

Просмотр истории в стиле 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 коммиты, которые были переданы на Subversion сервер. Локальные Git коммиты, которые вы ещё не отправили с помощью `dcommit`, не будут отображаться, равно как и коммиты, 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-Аннотации

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 коммиты, которые вы сделали локально в Git или те, что были отправлены на Subversion сервер с момента последней связи с ним.

Информация о SVN-сервере

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.

Игнорирование того, что игнорирует 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 `pre-receive`, который будет проверять каждое изменение на наличие `git-svn-id` и отклонять коммиты, 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 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 любое 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 https://www.mercurial-scm.org/.

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 работе с Mercurial.

```
$ hg clone http://selenic.com/repo/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/origin/branches/default, refs/hg/origin/bookmarks/master, master) Create a
makefile
* 65bb417 Create a standard 'hello, world' program
```

Наверняка вы обратили внимание, что мы использовали обыкновенный `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, Mercurial рассчитан на то, что каждый клиент хранит полную копию репозитория со всей историей, поэтому приведённая выше команда выполняет полное копирование со всей историей и делает это достаточно быстро.

`git log` показывает два коммита, на последний из которых указывает довольно много ссылок. На самом деле, не все из них реально существуют. Давайте-ка посмотрим, что хранится внутри каталога `.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 каталоге `refs/hg`. Например, `refs/hg/origin/branches/default` — это Git-ссылка, содержащая SHA-1 `ac7955c` — коммит на который ссылается ветка `master`. Таким образом, каталог `refs/hg` — это что-то типа `refs/remotes/origin`, с той разницей, что здесь же отдельно хранятся закладки и ветки Mercurial.

Файл `notes/hg` — отправная точка для выяснения соответствия между хешами коммитов 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 коммита, 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 беспокоится 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/branches/default, refs/hg/origin/bookmarks/master) Create a makefile
* 65bb417 Create a standard 'hello, world' program
```

Наша ветка `master` опережает `origin/master` на два коммита, 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, refs/hg/origin/bookmarks/master) Add some
documentation
| * 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 один коммит и история разошлась. В отличие от остальных систем контроля версий, рассматриваемых в этой главе, Mercurial умеет работать со слияниями, так что мы не будет 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, refs/hg/origin/bookmarks/master) Add some
documentation
* | 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   64f27bcefc35   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, 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 коммита 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 один из коммитов, 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-remote-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 обёрток для взаимодействия с другими системами контроля версий).

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 «принудительной» 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 системой контроля версий.

Git и Bazaar

Ещё одна известная распределённая система контроля версий Bazaar. Bazaar — это бесплатная система с открытым исходным кодом, являющаяся частью проекта GNU Project.

Её поведение сильно отличается от Git. Иногда, чтобы сделать то же самое, что и в Git, следует использовать другое ключевое слово, а некоторые такие же ключевые слова имеют другое значение. В частности, управления ветками сильно отличается и может вызвать путаницу, особенно для кого-нибудь из вселенной Git. Тем не менее, с Bazaar репозиторием возможно работать из Git.

Существует много проектов, которые позволяют использовать Git как клиент Bazaar. Далее, мы будем использовать проект Филипа Контрераса, который можно найти здесь https://github.com/felipec/git-remote-bzr. Для установки достаточно просто скачать файл `git-remote-bzr` и поместить его в один из каталогов вашего `$PATH`:

```
$ wget https://raw.github.com/felipec/git-remote-bzr/master/git-remote-bzr -O
~/bin/git-remote-bzr
$ chmod +x ~/bin/git-remote-bzr
```

Так же вам понадобится установленный Bazaar. И всё!

Создание репозитория Git из репозитория Bazaar

Им просто пользоваться. Чтобы клонировать Bazaar репозиторий достаточно добавить префикс `bzr::`. Так как Git и Bazaar полностью клонируют репозиторий на ваш компьютер, то можно добавить клон Git к локальному клону Bazaar, но так делать не рекомендуется. Гораздо проще связать клон Git с центральным хранилищем — тем же местом, с которым связан клон Bazaar.

Предположим, что вы работали с удалённым репозиторием, находящимся по адресу `bzr+ssh://developer@mybazaarserver:myproject`. Чтобы его клонировать, нужно выполнить следующие команды:

```
$ git clone bzr::bzr+ssh://developer@mybazaarserver:myproject myProject-Git
```

```
$ cd myProject-Git
```

На текущий момент, созданный Git репозиторий использует дисковое пространство не оптимально. Поэтому вы должны очистить и сжать его, особенно если репозиторий большого размера:

```
$ git gc --aggressive
```

Ветки в Bazaar

Bazaar позволяет клонировать только ветки, при этом репозиторий может содержать их несколько, а `git-remote-bzr` может клонировать все. Например, чтобы клонировать ветку выполните:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs/trunk emacs-trunk
```

Чтобы клонировать весь репозиторий, выполните команду:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs emacs
```

Последняя команда клонирует все ветки репозитория emacs; тем не менее, конфигурацией допускается указывать только некоторые из них:

```
$ git config remote-bzr.branches 'trunk, xwindow'
```

Некоторые удалённые репозитории не позволяют просматривать список веток, поэтому их перечисление в конфигурации для команды клонирования может оказаться проще в использовании:

```
$ git init emacs
$ git remote add origin bzr::bzr://bzr.savannah.gnu.org/emacs
$ git config remote-bzr.branches 'trunk, xwindow'
$ git fetch
```

Игнорируем то, что игнорируется в .bzrignore

При работе с проектом под управлением Bazaar вы не должны создавать файл `.gitignore`, потому что можете случайно добавить его в отслеживаемые, чем могут возмутиться другие пользователи, работающие с Bazaar. Решением может выступать создание файла `.git/info/exclude`, который может быть как символической ссылкой, так и обычным файлом. Позже мы рассмотрим пример решения этой проблемы.

Bazaar использует ту же модель игнорирования файлов что и Git, за исключением двух особенностей, не имеющих эквивалента в Git. Полное описание можно найти в

документации. Эти два отличия следующие:

1. `!!` позволяет игнорировать определённые шаблоны файлов, даже если они указаны со знаком `!`
2. `RE:` в начале строки позволяет указать регулярное выражение Python. Git допускает только шаблоны оболочки.

Следовательно, возможны две ситуации:

1. Если файл `.bzrignore` не содержит специфических префиксов, то можно просто создать символическую ссылку на него: `ln -s .bzrignore .git/info/exclude`
2. Иначе, нужно создать файл `.git/info/exclude` и адаптировать его в соответствии с `.bzrignore` так, чтобы игнорировались те же файлы.

Вне зависимости от ситуации, вам нужно следить за изменениями в `.bzrignore`, чтобы файл `.git/info/exclude` всегда соответствовал `.bzrignore`. На самом деле, если в файл `.bzrignore` будут добавлены изменения в виде одной или нескольких строк с `!!` или `RE:` вначале, то Git просто не сможет их интерпретировать и вам понадобиться изменить файл `.git/info/exclude` так, чтобы игнорировались те же файлы. Более того, если файл `.git/info/exclude` был символической ссылкой, то сначала нужно его удалить, скопировать `.bzrignore` в `.git/info/exclude` и адаптировать последний. Однако, будьте осторожны с его созданием, потому что в Git невозможно повторно включить файл в индекс, если исключен родительский каталог этого файла.

Получение изменений с удалённого репозитория

Используя обычные команды Git можно получить изменения с удалённого репозитория. Предположим, что изменения находятся в ветке `master`, вы сливаете или перебазируете свою работу относительно ветки `origin/master`:

```
$ git pull --rebase origin
```

Отправка в удалённый репозиторий

Поскольку Bazaar так же имеет концепцию коммитов слияния, то проблем не возникнет при отправке такого коммита. Таким образом, вы можете работать в ветке, сливать изменения в `master` и отправлять их. Вы можете создавать ветки, делать коммиты и тестировать изменения как обычно. Наконец, вы отправляете проделанную работу в репозиторий Bazaar:

```
$ git push origin master
```

Предупреждение

Существуют ограничения на выполнение операций с удалённым репозиторием. В частности, следующие команды не работают:

- git push origin :branch-to-delete (Bazaar не понимает удаление ссылок таким способом.)
- git push origin old:new (будет отправлена 'old')
- git push --dry-run origin branch (push будет выполнен)

Заключение

Поскольку модели Git и Bazaar схожи, то не так много усилий требуется для их совместной работы. Всё будет в порядке пока вы следите за ограничениями и знаете, что удалённый репозиторий изначально не Git.

Git и Perforce

Perforce — очень распространённая система контроля версий в корпоративной среде. Она появилась в 1995 году, что делает её самой старой системой контроля версий, 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 поддеревья 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.

Установка

Для примера мы воспользуемся простейшим способом настройки Git Fusion — подготовленным образом для виртуальной машины с предустановленным Perforce демоном и собственно Git Fusion. 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:

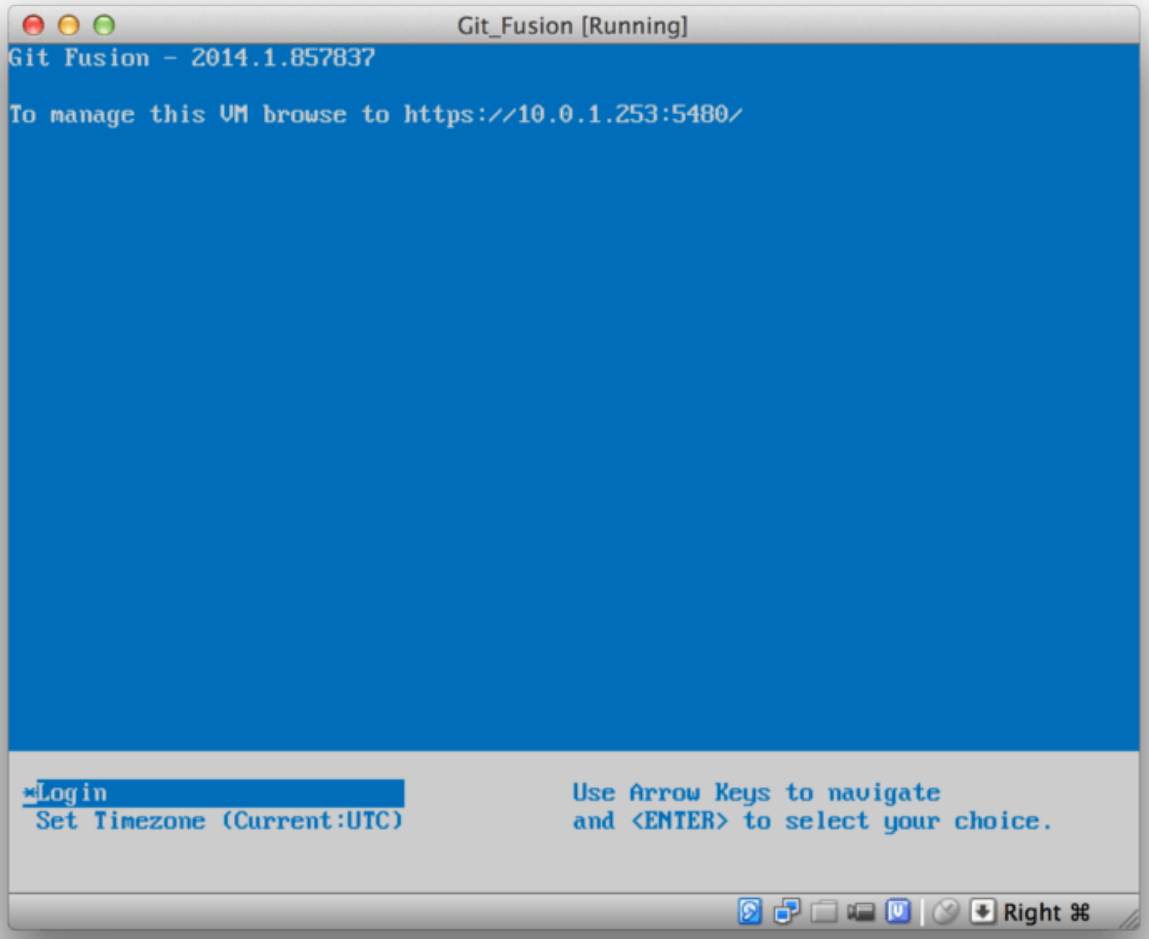


Рисунок 145. Экран виртуальной машины Git Fusion

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.

Настройка Fusion

После установки Git Fusion вы, возможно, захотите настроить его. Это относительно несложно сделать, используя ваш любимый Perforce клиент; просто отобразите каталог `//.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
```

Каталог `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 обыкновенный 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 [@repo], переопределяющую 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 каталогов, 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 коммит, 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 коммитов 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 коммитов Perforce используется информация из первой совпавшей строки.

Последние две строки скрывают настоящие имена Боба и Джо в созданных Git коммитах. 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.

Рабочий процесс

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://john@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 -- the Intel one.
| * 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 коммиты. 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
Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

Теперь у нас два новых коммита. 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 коммит с SHA-1 `6afeb15` был создан Perforce клиентом. Он выглядит так же, как и любой другой коммит, и это именно то, для чего создан Git Fusion. Давайте посмотрим, как Perforce обработает коммит слияния:

```
$ 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`:

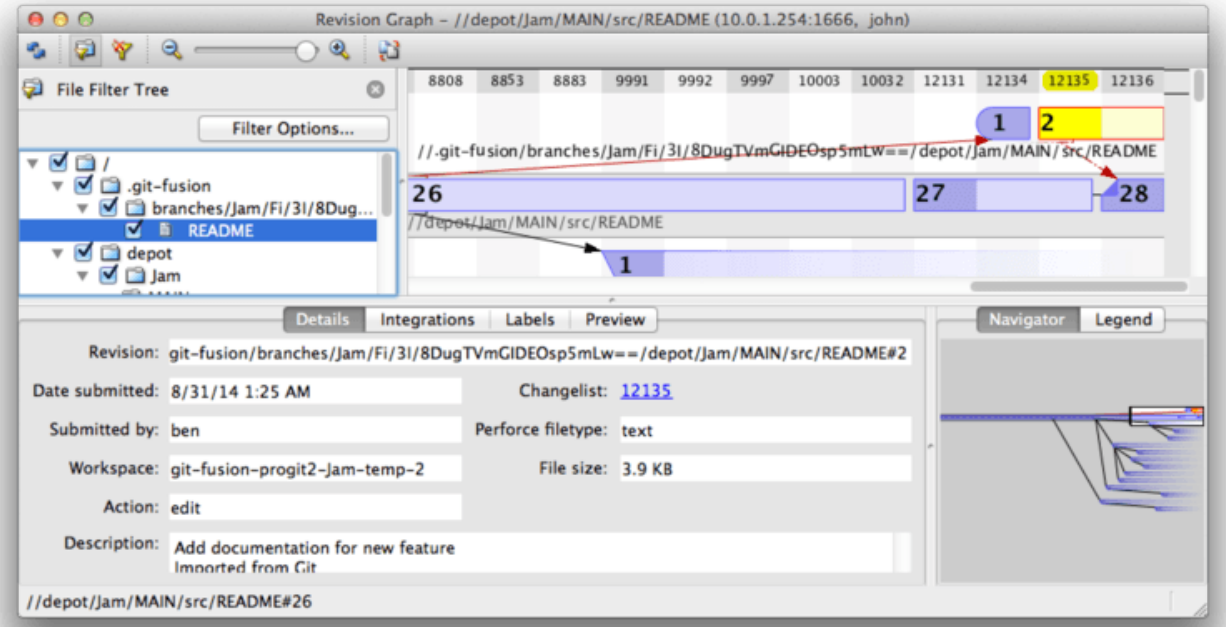


Рисунок 146. Граф ревизий Perforce после отправки данных из Git

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 каталогов слева вверху показывает этот файл в разных ветках. Справа вверху мы видим граф зависимости разных ревизий файла, справа внизу этот же граф показан целиком для быстрого ориентирования. Оставшаяся часть окна отображает детали выбранной ревизии (в нашем случае это ревизия `2`).

Граф выглядит в точности как в Git. У Perforce не было именованной ветки для сохранения коммитов `1` и `2`, так что он создал «анонимную» ветку в каталоге `.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.

Заключение по 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 всей функциональности 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.

Настройка

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
```

Начало работы

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 state at revision #head
```

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.

Рабочий процесс

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 at
revision #head
```

Мы подготовили два коммита 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 коммиты 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 коммитов 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   2014-
08-31 18:26:05.000000000 0000
```

```
@@ -60,7 +60,7 @@
 </td>
 <td valign=top>
 Source and documentation for
-<a href="http://www.perforce.com/jam/jam.html">
+<a href="jam.html">
 Jam/MR</a>,
 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 коммита, но в некоторых случаях вы захотите изменить его. Например, если коммит в Git был создан человеком, у которого нет Perforce аккаунта, вы всё равно захотите сделать автором коммита его, а не себя.

`git-p4` вставил сообщение из коммита Git в содержимое набора изменений Perforce, так что всё что нам остаётся сделать — это дважды сохранить и закрыть редактор (по одному разу на каждый коммит). В результате мы получим такой вывод:

```
$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
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 каждый коммит в Git превращается в отдельный набор изменений Perforce; если вы хотите слепить их воедино, вы можете сделать это с помощью интерактивного переноса коммитов до выполнения `git p4 submit`. Ещё один важный момент: SHA-1 хеши коммитов, 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 коммит слияния? 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 коммит `775a46f`. В Git мы имеем два дополнительные коммита, затем слияние с состоянием Perforce, затем ещё один коммит. 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-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-
mbp_8487/depot/www/live/
```

```
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 коммита 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.

Ветвление

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 поддерева, 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 смышлёным, 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 коммитов: если у вас есть удалённый репозиторий, который используют другие люди, не публикуйте в нём коммиты, 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.

Переход на Git

Если у вас уже есть кодовая база в другой системе контроля версий, но вы решили начать использовать Git, вам необходимо перенести проект тем или иным способом. 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 основных профессионально используемых систем контроля версий, так как они используются большинством разработчиков, желающих переключиться на использование Git, а так же 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 клонировать Subversion репозиторий. После этого вы можете прекратить использовать Subversion и перейти на Git. Сразу же после клонирования вам будет доступна вся история репозитория, хотя сам процесс получения копии может затянуться.

Вдобавок 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 --quiet | grep author | sort -u | \
  perl -pe 's/.*>(.*?)<.*/$1 = /'
```

Эта команда выводит историю коммитов в формате XML, затем оставляет только строки с информацией об авторе, удаляет дубликаты и обрезает XML-теги. Естественно, она сработает только на компьютерах с установленными `grep`, `sort` и `perl`. Перенаправив вывод этой команды в файл `users.txt`, вам останется только дописать в каждой строке соответствующих авторов для Git.

> Если вы пытаетесь выполнить это на компьютере с Windows, то у вас может возникнуть ряд проблем. Однако, Microsoft предоставила несколько полезных советов по миграции https://docs.microsoft.com/en-us/azure/devops/repos/git/perform-migration-from-svn-to-git.

Для точного сопоставления авторов коммитов передайте файл `users.txt` команде `git svn`. Добавив флаг `--no-metadata` в команды `clone` или `init`, можно указать `git svn` исключить импорт метаданных, которые импортируются по умолчанию. Как часть метаданных, `git-svn-id` включается в каждое сообщение коммита, генерируемое Git при импорте, что может привести к необоснованному увеличению истории и сделать её более запутанной.

> Метаданные следует сохранять, если вы планируете отправлять коммиты из Git обратно в SVN репозиторий. Если полной синхронизации не требуется, то спокойно добавляйте параметр `--no-metadata`.

В результате, команда `import` примет вид:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
      --authors-file=users.txt --no-metadata --prefix "" -s my_project
$ cd my_project
```

Теперь у вас будет красивая копия репозитория Subversion в каталоге `my_project`. Вместо коммитов 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@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Теперь не только поле `Author` выглядит лучше, но и `git-svn-id` не мозолит глаза.

Также вам следует немного почистить репозиторий сразу после импорта. Во-первых, следует удалить ненужные ссылки, устанавливаемые `git svn`. Для начала, переместим теги, потому как в действительности это теги, а не странные удалённые ветки; затем все удалённые ветки сделаем локальными.

Чтобы переместить теги, выполните следующую команду:

```
$ for t in $(git for-each-ref --format='%(refname:short)' refs/remotes/tags); do git
tag ${t/tags\//} $t && git branch -D -r $t; done
```

Эта команда берёт ссылки на удалённые ветки, которые располагаются в `refs/remotes/tags/`, и делает их настоящими (легковесными) тегами.

Затем, переместим оставшиеся ссылки из `refs/remotes`, чтобы сделать из них локальные ветки:

```
$ for b in $(git for-each-ref --format='%(refname:short)' refs/remotes); do git branch
$b refs/remotes/$b && git branch -D -r $b; done
```

Возможно, для одной ветки Subversion будут созданы дополнительные ветки с суффиксом `@xxx` (где xxx — это число). Это связано с особенностью Subversion, которая называется «peg-revisions», для которой Git не имеет синтаксического аналога. Поэтому, `git svn` просто добавляет номер версии svn в название ветки, точно так же как вы бы это сделали в svn при добавлении peg-revision для ветки. Если эти ревизии вам больше не нужны, то просто удалите их используя команду:

```
$ for p in $(git for-each-ref --format='%(refname:short)' | grep @); do git branch -D
$p; done
```

Теперь все ветки стали настоящими Git ветками, а теги — настоящими Git тегами.

К сожалению, `git svn` создаёт дополнительную ветку с названием `trunk`, которая соответствует ветке по умолчанию в Subversion и аналогична ветке `master`. Так как `master` больше подходит для Git, удалим лишнюю ветку:

```
$ git branch -d trunk
```

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 теги, выполите следующие команды:

```
$ git push origin --all
$ git push origin --tags
```

Наконец, все ваши ветки и теги 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 https://github.com/frej/fast-export.git
```

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 однострочный скрипт на `bash`, который генерирует 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, преобразовав каждую строку в правило: `"<input>"="<output>"`, где `<input>` преобразуется в `<output>`. Строки `<input>` и `<output>` могут содержать экранированные последовательности, поддерживаемые кодировкой python `string_escape`. Если файл сопоставлений авторов коммитов не содержит соответствующего `<input>`, то значение будет передано Git без модификации. Если же все имена выглядят хорошо, этот файл и вовсе не потребуется. В нашем примере мы хотим

чтобы файл выглядел так:

```
"bob"="Bob Jones <bob@company.com>"
"bob@localhost"="Bob Jones <bob@company.com>"
"bob <bob@company.com>"="Bob Jones <bob@company.com>"
"bob jones <bob <AT> company <DOT> com>"="Bob Jones <bob@company.com>"
```

Аналогичные файлы применяются для переименования веток и тегов, когда сохранённое в Mercurial название недопустимо в Git.

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 займёт 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
files
master: Exporting simple delta revision 22207/22208 with 0/2/0 added/changed/removed
files
master: Exporting thorough delta revision 22208/22208 with 3/213/0
added/changed/removed files
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
attempts)
      trees  :        52320 (      2851 duplicates      47467 deltas of      47599
```

```
attempts)
      commits:        22208 (         0 duplicates          0 deltas of          0
attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0
attempts)
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 теги были преобразованы в теги 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
```

Bazaar

Bazaar — это распределённая система контроля версий очень похожая на Git, поэтому репозиторий Bazaar достаточно легко сконвертировать в репозиторий Git. Для этого вам необходимо подключить плагин `bzr-fastimport`.

Установка плагина bzr-fastimport

Для UNIX подобных систем и Windows процедура установки плагина отличается. В первом случае, самый простой способ это установить пакет `bzr-fastimport`, вместе с которым будут установлены все необходимые зависимости.

Например, для Debian и подобных, следует выполнить:

```
$ sudo apt-get install bzr-fastimport
```

Для RHEL выполните следующую команду:

```
$ sudo yum install bzr-fastimport
```

Для Fedora, начиная с версии 22, новый менеджер пакетов dnf:

```
$ sudo dnf install bzr-fastimport
```

Если пакет отсутствует в репозитории для вашего дистрибутива, то вы можете установить его как плагин, используя следующие команды:

```
$ mkdir --parents ~/.bazaar/plugins     # создаст необходимые каталоги для плагинов
$ cd ~/.bazaar/plugins
$ bzr branch lp:bzr-fastimport fastimport   # импортирует плагин fastimport
$ cd fastimport
$ sudo python setup.py install --record=files.txt   # установит плагин
```

Чтобы плагин заработал, вам понадобится модуль Python `fastimport`. Проверить наличие и установить его можно следующими командами:

```
$ python -c "import fastimport"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named fastimport
$ pip install fastimport
```

Если модуль недоступен, то его можно скачать по адресу https://pypi.python.org/pypi/fastimport/.

Во втором случае (в Windows), `bzr-fastimport` устанавливается автоматически при стандартной установке (все галочки отмечены). В таком случае дальнейших действий не требуется.

Процесс импорта Bazaar репозитория отличается в зависимости от того одна ветка в вашем репозитории или несколько.

Проект с одной веткой

Войдите в каталог, содержащий ваш Bazaar репозиторий и проинициализируйте Git репозиторий:

```
$ cd /path/to/the/bzr/repository
$ git init
```

Теперь, просто экспортируйте свой Bazaar репозиторий и сконвертируйте его в Git

репозиторий используя следующую команду:

```
$ bzr fast-export --plain . | git fast-import
```

В зависимости от размера проекта, Git репозиторий будет готов через несколько секунд или минут.

Проект с основной и рабочей ветками

Вы так же можете импортировать Bazaar репозиторий с несколькими ветками. Предположим, что в вашем репозитории две ветки: одна является основной веткой проекта (myProject.trunk), другая — рабочей (myProject.work).

```
$ ls
myProject.trunk myProject.work
```

Проинициализируйте Git репозиторий и перейдите в его каталог:

```
$ git init git-repo
$ cd git-repo
```

Импортируйте в Git основную ветку с помощью команды:

```
$ bzr fast-export --export-marks=../marks.bzr ../myProject.trunk | \
git fast-import --export-marks=../marks.git
```

Импортируйте в Git рабочую ветку с помощью команды:

```
$ bzr fast-export --marks=../marks.bzr --git-branch=work ../myProject.work | \
git fast-import --import-marks=../marks.git --export-marks=../marks.git
```

Теперь, команда `git branch` покажет вам две ветки: `master` и `work`. Проверьте логи, чтобы убедиться в отсутствии ошибок, после этого можно удалить файлы `marks.bzr` и `marks.git`.

Синхронизация индекса

Вне зависимости от количества веток и выбранного метода импорта, индекс не синхронизируется с `HEAD`, а при импорте нескольких веток — так же не синхронизируется рабочий каталог. Эту ситуацию можно легко исправить следующей командой:

```
$ git reset --hard HEAD
```

Игнорирование файлов из .bzrignore

Теперь давайте посмотрим на файлы, которые следует игнорировать. Первое, что нужно сделать — это переименовать `.bzrignore` в `.gitignore`. Если файл `.bzrignore` содержит одну или несколько строк начинающихся с `!!` или `RE:`, нужно их изменить и, возможно, создать несколько файлов `.gitignore`, чтобы заставить Git игнорировать точно те же файлы, которые игнорируются Bazaar.

Наконец, создайте коммит со всеми изменениями, внесёнными во время миграции:

```
$ git mv .bzrignore .gitignore
$ # modify .gitignore if needed
$ git commit -am 'Migration from Bazaar to Git'
```

Отправка репозитория на сервер

Вот и всё! Теперь вы можете отправить репозиторий на сервер в его новый дом:

```
$ git remote add origin git@my-git-server:mygitrepository.git
$ git push origin --all
$ git push origin --tags
```

Ваш Git репозиторий готов к использованию.

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, 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 каталог, в который 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 перейдёте в каталог `p4import` и выполните `git log`, вы увидите результат:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

    [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 (cummulative -> cumulative).

    [git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

`git-p4` оставил идентификаторы в сообщениях всех коммитов. 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 приступать 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 коммитов 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 </UL> to </OL>.

commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).
```

Теперь ваш репозиторий готов к отправке на Git сервер.

Импорт произвольного репозитория

Если вы пользуетесь какой-либо другой системой контроля версий, не перечисленной выше, вам следует поискать инструмент для импорта в Сети — качественные решения доступны для CVS, Clear Case, Visual Source Safe и даже каталогов с архивами. Если всё же существующие решения вам не подошли, вы пользуетесь менее известной системой контроля версий или вам нужно больше контроля над процессом импорта — используйте `git fast-import`. Эта команда читает простые инструкции из потока ввода и записывает данные в Git. Создать Git-объекты таким путём намного проще, чем через низкоуровневые Git-команды или пытаясь воссоздать их вручную (обратитесь к главе Git изнутри за деталями). Таким образом, вы можете написать небольшой скрипт, считывающий нужную информацию из вашего хранилища и выводящий инструкции в стандартный поток вывода. Затем вы можете запустить эту программу и передать её вывод прямиком в `git fast-import`.

Для демонстрации, мы с вами напишем простой скрипт для импорта. Предположим, вы работаете в каталоге `current` и периодически создаёте резервные копии в каталогах вида `back_YYYY_MM_DD`, и хотите перенести данные в Git. Структура каталогов выглядит следующим образом:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

Чтобы успешно импортировать репозиторий, давайте вспомним, как Git хранит данные. Как вы наверное помните, Git по сути представляет собой связанный список ревизий, каждая из которых указывает на слепок состояния. Всё что от вас требуется, это указать

`fast-import'у на данные для создания слепков и порядок их применения. Итак, мы пробежимся по всем слепкам, создадим коммит для каждого из них и свяжем каждый новый коммит с предыдущим.

Как и в разделе Пример принудительной политики Git главы 8, мы проделаем это на Ruby, потому что это тот язык, с которым мы обычно работаем, и его легко читать. Вы можете использовать любой другой язык — всё что требуется, это вывести нужную информацию в стандартный поток вывода.

Если вы работаете на Windows, будьте особо осторожными с переводами строк: `fast-import` ожидает лишь символы перевода строки (`LF`), но не возврат каретки + перевод строки (`CRLF`), как принято в Windows.

Для начала перейдём в исходный каталог и определим подкаталоги, содержащие состояния проекта в разные моменты времени, которые будут использованы для построения соответствующих коммитов. Вы поочерёдно посетите каждую из них и выполните команды, необходимые для экспорта. Примерно так:

```
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
```

Вы выполняете функцию `print_export` внутри каждого каталога. Она принимает на вход текущий каталог и результат предыдущего вызова и помечает текущий каталог, возвращая данные для последующих вызовов, таким образом связывая коммиты. Метки используются для связи коммитов вместе. Итак, первым делом нужно сгенерировать метку по каталогу:

```
mark = convert_dir_to_mark(dir)
```

Создадим массив каталогов и используем индекс каталога в нём как метку; это удобно, ведь метка должна быть целым числом. Мы написали такой код:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
```

```
  ($marks.index(dir) + 1).to_s
end
```

Теперь, когда у нас есть целочисленная метка для коммита, нужна дата. У нас она хранится в имени каталога, придётся достать её оттуда. Следующая строка в `print_export`:

```
date = convert_dir_to_date(dir)
```

где `convert_dir_to_date` определяется как

```
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
```

Этот код вернёт целочисленное представление даты для каждого каталога. И последний кусочек мозаики: автор изменений. Это значение жёстко задано в глобальной переменной:

```
$author = 'John Doe <john@example.com>'
```

Теперь всё готово для вывода нужной `fast-import'у информации. Нужно указать, что создаётся коммит на определённой ветке, затем вывести сгенерированную метку, автора и время изменений и ссылку на предыдущий коммит, если такой имеется. Код выглядит следующим образом:

```
# 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
```

Для простоты, мы определили часовой пояс как -0700 прямо в выходной строке. Часовой пояс задаётся как смещение от UTC. Сообщение коммита задаётся следующим образом:

```
data (size)\n(contents)
```

Первым идёт слово `data`, затем длина сообщения, новая строка и, наконец, само сообщение. Похожим образом задаётся и содержимое коммитов, поэтому создадим метод-помощник:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Осталось лишь задать содержимое каждого коммита. Это довольно просто, потому что все данные хранятся в отдельных каталогах — достаточно напечатать команду `deleteall`, а следом за ней содержимое всех файлов каталога. После этого Git запишет слепки:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Замечание: многие системы работают с дельтами (разницами от одного состояния к последующему); `fast-import` имеет команды для задания изменений: какие файлы были добавлены, удалены или изменены. Вы можете вычислять разницу между состояниями и передавать её в `fast-import`, но это довольно сложно, гораздо проще передавать Git все данные. За полным описанием принимаемых форматов обратитесь к руководству `fast-import`.

Формат для указания нового содержимого или изменений следующий:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Здесь `644` — это права доступа к файлу. Если файл должен быть исполняемым, вам нужно определить это и передать `755`. Слово `inline` говорит о том, что вы выведете содержимое файла после этой строки. Таким образом, метод `inline_data` может выглядеть так:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

Мы используем определённый ранее метод `export_data` потому что форматы содержимого коммитов и их сообщений одинаковы.

И последнее что нужно сделать — это вернуть метку для последующих вызовов:

```
return mark
```

Если вы используете ОС Windows есть ещё кое-что. Как мы упоминали ранее, Windows использует `CRLF` для новых строк, в то время как `git fast-import` ожидает только `LF`. Чтобы исправить этот недостаток Windows и осчастливить `git fast-import`, просто прикажите Ruby использовать `LF` вместо `CRLF`:

```
$stdout.binmode
```

Вот и всё. Ниже приведён весь скрипт целиком:

```
#!/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
```

Если вы выполните этот скрипт, он выведет примерно следующее:

```
$ 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
(...)
```

Чтобы импортировать репозиторий перенаправьте этот вывод в команду `git fast-import`, запущенную в каталоге с целевым Git-репозиторием. Вы можете создать новый каталог, выполнить в нём `git init`, а затем запустить свой скрипт:

```
$ 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          5
attempts)
      trees  :            4 (         1 duplicates          0 deltas of          4
attempts)
      commits:            4 (         1 duplicates          0 deltas of          0
attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0
attempts)
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
---------------------------------------------------------------------
```

Как вы можете видеть, после успешного завершения `fast-import` выводит некоторую статистику о проделанной работе. В этом случае, вы импортировали 13 объектов в 4-х коммитах одной ветки. Теперь можете выполнить `git log` просмотреть созданную историю коммитов:

```
$ 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
```

Вот он: ваш новый классный Git репозиторий! Обратите внимание, ваш рабочий каталог пуст, активная ветка не выбрана. Переключимся на ветку `master`:

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

Функциональность `fast-import` гораздо шире описанного: он поддерживает права доступа к файлам, двоичные файлы, множественные ветки и их слияния, метки, индикатор прогресса и ещё кучу вещей. Несколько примеров более сложных сценариев использования `fast-import` можно найти в каталоге `contrib/fast-import` исходного кода Git.

Заключение

После всего вышесказанного вы должны чувствовать себя уверенно, используя Git как клиент для других систем контроля версий или импортируя практически любой существующий репозиторий в Git без потери данных. Следующая глава раскроет перед вами внутреннюю механику Git, так что вы будете способны контролировать каждый байт данных, если это потребуется.

Git изнутри

Вы могли прочитать почти всю книгу перед тем, как приступить к этой главе, а могли только часть — в любом случае в данной главе рассматриваются внутренние процессы Git и особенности его реализации. На наш взгляд, изучение этого материала является основой понимания того, насколько Git полезный и мощный инструмент, однако, некоторые утверждают, что изложение этого материала может сбить новичков с толку и оказаться для них неоправданно сложным. Именно поэтому глава отнесена в самый конец книги, 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 упрощён до уровня аналогов; тем не менее, 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 возможно вам придётся столкнуться.

Сантехника и Фарфор

В этой книге было описано, как пользоваться Git, применяя примерно три десятка команд, таких как `checkout`, `branch`, `remote` и так далее. Но так как сначала Git был скорее инструментарием для создания системы контроля версий, чем самой VCS, удобной для пользователей, то в нём полно команд, выполняющих низкоуровневые операции, которые спроектированы так, чтобы их можно было использовать объединив в цепочку в стиле UNIX, а также использовать в скриптах. Эти команды, как правило, называют служебными («plumbing» — трубопровод), а ориентированные на пользователя называют пользовательскими («porcelain» — фарфор).

Первые девять глав книги были посвящены в основном пользовательским командам. В данной главе рассматриваются именно низкоуровневые служебные команды, дающие контроль над внутренними процессами Git 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 скриптов.

Когда вы выполняете `git init` в новом или существовавшем ранее каталоге, Git создаёт подкаталог `.git`, в котором располагается почти всё, чем он манипулирует. Если требуется выполнить резервное копирование или клонирование репозитория, достаточно скопировать лишь этот каталог, чтобы получить почти всё необходимое. Данная глава

почти полностью посвящена его содержимому. Вот так выглядит только что созданный каталог `.git`:

```
$ ls -F1
config
description
HEAD
hooks/
info/
objects/
refs/
```

В зависимости от используемой версии Git, здесь могут присутствовать и другие файлы, но по умолчанию команда `git init` создаёт именно такое содержимое в каталоге `.git`. Файл `description` используется только программой GitWeb, не обращайте на него внимание. Файл `config` содержит специфичные для этого репозитория конфигурационные параметры, а в каталоге `info` расположен файл с глобальными настройкам игнорирования файлов — он позволяет исключить файлы, которые вы не хотите помещать в `.gitignore`. В каталоге `hooks` располагаются клиентские и серверные хуки, подробно рассмотренные в разделе Хуки в Git главы 8.

Итак, осталось четыре важных элемента: файлы `HEAD` и `index` (ещё не созданный) и каталоги `objects` и `refs`. Это ключевые элементы Git. В каталоге `objects` находится база данных объектов Git; в `refs` — ссылки на объекты коммитов в этой базе (ветки, теги и другие); файл `HEAD` указывает на текущую ветку, а в файле `index` хранится содержимое индекса. Далее мы детально рассмотрим эти элементы, 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.

В качестве примера, воспользуемся служебной командой `git hash-object`, которая берёт некоторые данные, сохраняет их в виде объекта в каталоге `.git/objects` (*база данных объектов*) и возвращает уникальный ключ, который является ссылкой на созданный объект.

Для начала создадим новый Git-репозиторий и убедимся, что каталог `objects` пуст:

```
$ 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 каталог `objects` и создал в нём пустые подкаталоги `pack` и `info`. Теперь с помощью `git hash-object` создадим объект и вручную добавим его в базу Git:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

В простейшем случае `git hash-object` берёт переданный контент и возвращает уникальный ключ, который будет использоваться для хранения данных в базе Git. Параметр `-w` указывает команде `git hash-object` не просто вернуть ключ, а ещё и сохранить объект в базе данных. Последний параметр `--stdin` указывает, что `git hash-object` должна использовать данные, переданные на стандартный потока ввода; в противном случае команда ожидает путь к файлу в качестве аргумента.

Результат выполнения команды — 40-символьная контрольная сумма. Это SHA-1 хеш — контрольная сумма содержимого и заголовка, который будет рассмотрен позднее. Теперь можно посмотреть как Git хранит ваши данные:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Мы видим новый файл в каталоге `objects`. Это и есть начальное внутреннее представление данных в Git — один файл на единицу хранения с именем, являющимся контрольной суммой содержимого и заголовка. Первые два символа SHA-1 определяют подкаталог файла внутри `objects`, остальные 38 — его имя.

Извлечь содержимое объекта можно при помощи команды `cat-file`. Она подобна швейцарскому ножу для анализа объектов Git. Ключ `-p` указывает команде `cat-file` автоматически определять тип объекта и выводить результат в соответствующем 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 проверсионировать 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 имён 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 каталогу на файловой системе, а блоб более или менее соответствует индексу узла (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 подкаталог `lib` — не блоб, а указатель на другое дерево:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

> Вы можете столкнуться с различными ошибками при использовании синтаксиса `master^{tree}` в зависимости от того, какую оболочку используете.
>
> В Windows CMD символ `^` используется для экранирования, поэтому для исключения ошибок следует использовать двойной символ: `git cat-file -p master^^{tree}`. В PowerShell параметры, использующие символы {}, должны быть заключены в кавычки: `git cat-file -p 'master^{tree}'`.
>
> В ZSH символ `^` используется для подстановки, поэтому выражение следует помещать в кавычки: `git cat-file -p "master^{tree}"`.

Концептуально, данные хранятся в Git примерно так:

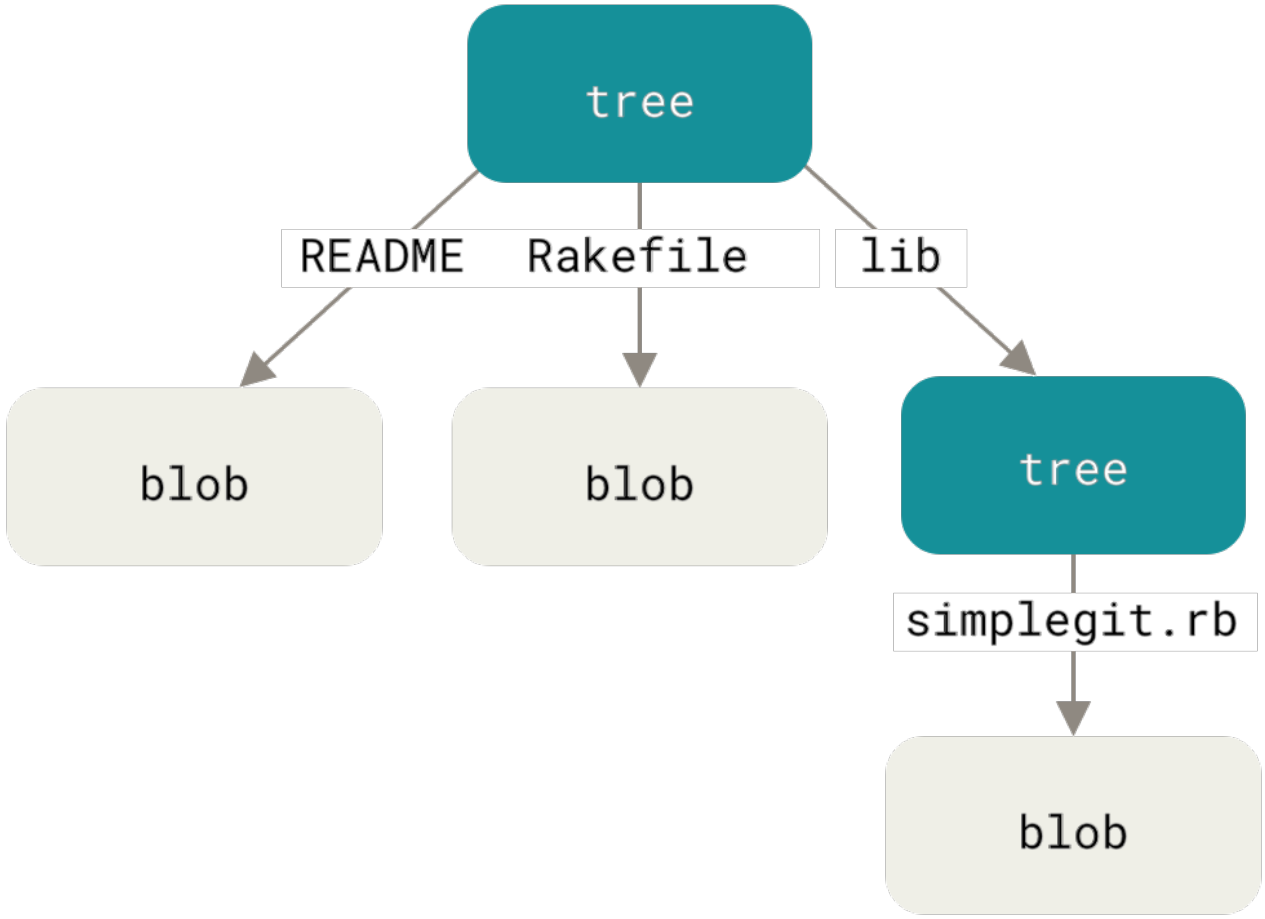


Рисунок 147. Упрощённая модель данных Git

Можно создать дерево самому. Обычно, Git создаёт дерево путём создания набора объектов из состояния области подготовленных файлов или индекса. Поэтому для создания дерева необходимо проиндексировать какие-нибудь файлы. Для создания индекса из одной записи — первой версии файла `test.txt` — воспользуемся низкоуровневой командой `git update-index`. Данная команда может искусственно добавить более раннюю версию `test.txt` в новый индекс. Необходимо передать опции `--add`, так как файл ещё не существует в индексе (да и самого индекса ещё нет), и `--cacheinfo`, так как добавляемого файла нет в рабочем каталоге, 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` — символическая ссылка. Права доступа в Git сделаны по аналогии с правами доступа в UNIX, но они гораздо менее гибки: указанные три режима — единственные доступные для файлов (блобов) в Git (хотя существуют и другие режимы, используемые для каталогов и подмодулей).

Теперь можно воспользоваться командой `git write-tree` для сохранения индекса в объект дерева. Здесь опция `-w` не требуется — команда автоматически создаст дерево из индекса, если такого дерева ещё не существует:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

Используя ту же команду `git cat-file`, можно проверить, что созданный объект действительно является деревом:

```
$ 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 --add --cacheinfo 100644 \
  1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
$ git update-index --add new.txt
```

Теперь в области подготовленных файлов 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 подкаталог текущего. Добавлять деревья в область подготовленных файлов можно с помощью команды `git read-tree`. В нашем случае, чтобы включить 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 добавили только что сохранённое дерево в рабочий каталог, вы бы

увидели два файла в его корне и подкаталог `bak` с первой версией файла `test.txt`. В таком случае хранимые структуры данных можно представить следующим образом:

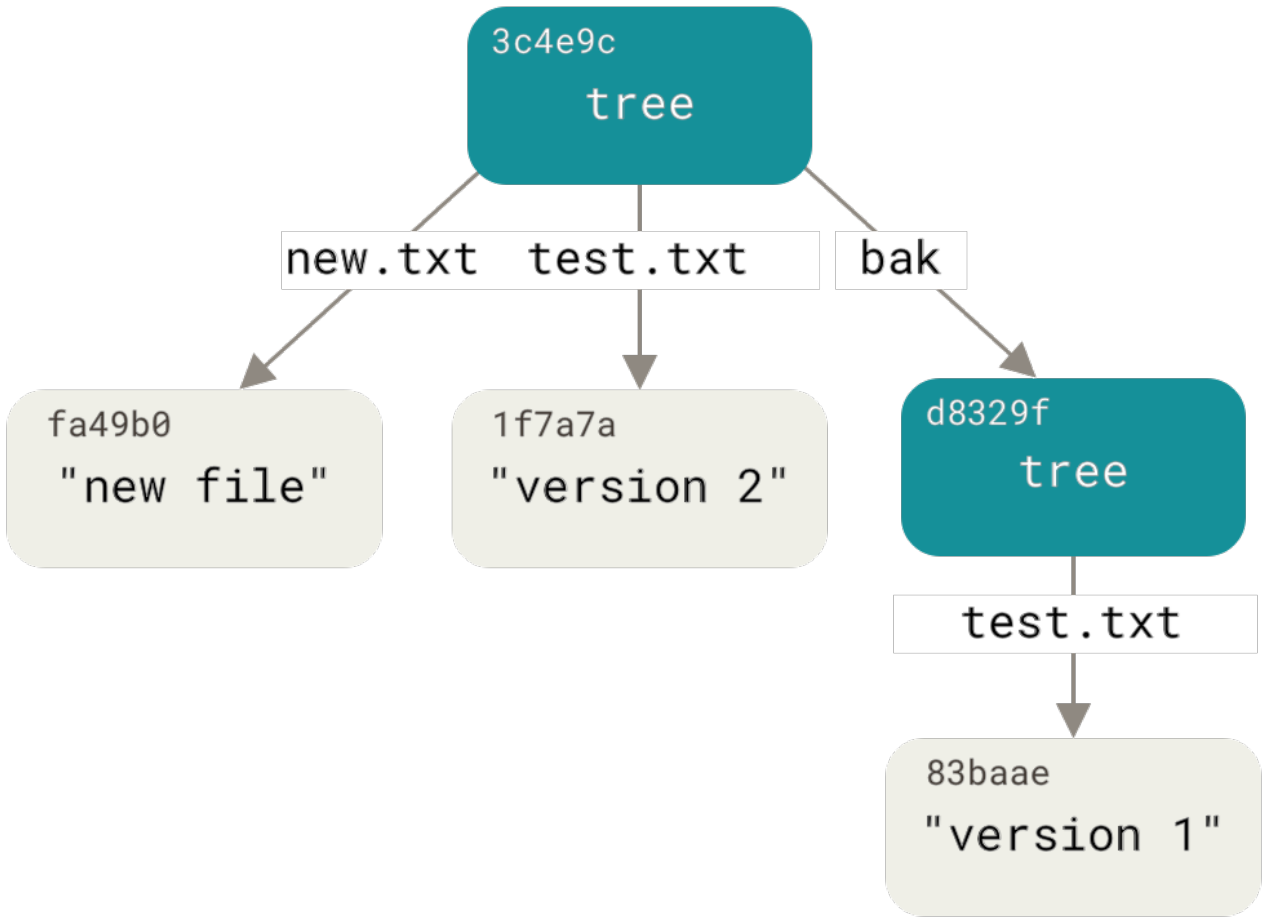


Рисунок 148. Структура данных Git для текущего состояния

Объекты коммитов

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 объекте коммита.

Для создания коммита необходимо вызвать команду `commit-tree` и задать SHA-1 нужного дерева и, если необходимо, родительские коммиты. Начнём с создания коммита для самого первого дерева:

```
$ echo 'First commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Полученный вами хеш будет отличаться, так как отличается дата создания и информация об авторе. Далее в этой главе используйте собственные хеши коммитов и тегов. Просмотреть созданный объект коммита 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 объекта коммита прост: в нём указано дерево верхнего уровня, соответствующее состоянию проекта на некоторый момент; родительские коммиты, если существуют (в примере выше объект коммита не имеет родителей); имена автора и коммиттера (берутся из полей конфигурации `user.name` и `user.email`) с указанием временной метки; пустая строка и сообщение коммита.

Далее, создадим ещё два объекта 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 созданных объектов коммитов указывает на одно из созданных ранее деревьев состояния 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 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, создаёт деревья и фиксирует изменения в объекте коммита, ссылающемся на дерево верхнего уровня и предшествующие коммиты. Эти три основных вида объектов Git — блоб, дерево и коммит — сохраняются в виде отдельных файлов в каталоге `.git/objects`. Вот как сейчас выглядит список объектов в этом каталоге, в комментарии указано чему соответствует каждый из них:

```
$ 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, представленный на рисунке:

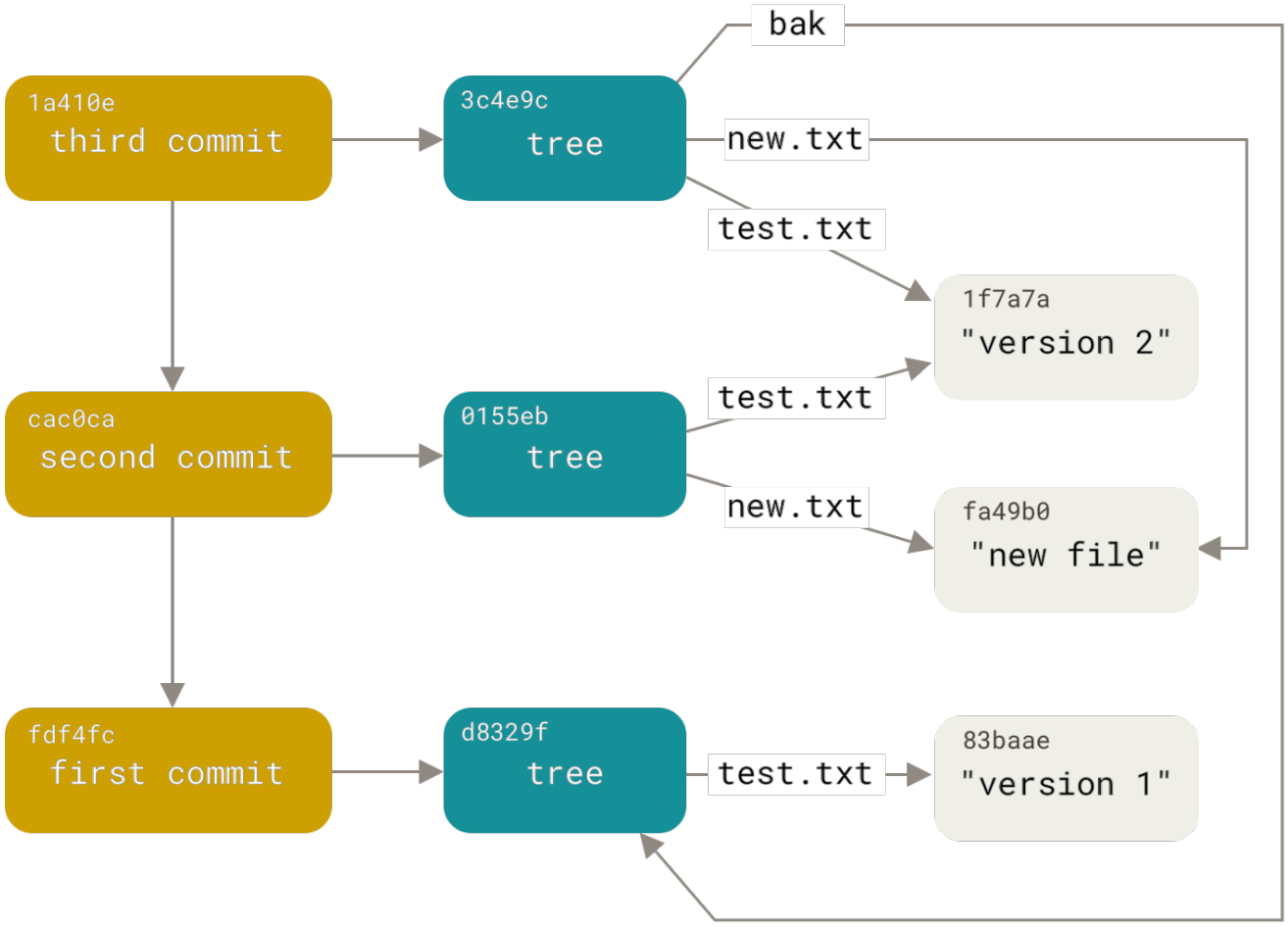


Рисунок 149. Все объекты в каталоге Git

Хранение объектов

Ранее мы упоминали, что вместе с содержимым объекта сохраняется дополнительный заголовок. Давайте посмотрим, как Git хранит объекты на диске. Мы рассмотрим как происходит сохранение блоб объекта — в данном случае это будет строка «what is up, doc?» — в интерактивном режиме 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 в байтах и в конце нулевой байт:

```
>> header = "blob #{content.bytesize}\0"
=> "blob 16\u0000"
```

Git объединяет заголовок и оригинальный контент, а затем вычисляет SHA-1 сумму от полученного результата. В Ruby значение SHA-1 для строки можно получить, подключив

соответствующую библиотеку командой `require` и затем вызвав `Digest::SHA1.hexdigest()`:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Давайте сравним полученный результат с выводом команды `git hash-object`. Здесь используется `echo -n` для предотвращения автоматического добавления переноса строки.

```
$ echo -n "what is up, doc?" | git hash-object --stdin
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\x9CK\xCA\xC9OR04c(\xCFH,Q\xC8,V(-\xD0QH\xC9O\xB6\a\x00_\x1C\a\x9D"
```

После этого сохраним сжатую строку в объект на диске. Определим путь к файлу, который будет записан (первые два символа хеша используются в качестве названия каталога, оставшиеся 38 — в качестве имени файла в ней). В Ruby для безопасного создания нескольких вложенных каталогов 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
```

Теперь проверим содержимое объекта с помощью `git cat-file`:

```
---
$ git cat-file -p bd9dbf5aae1a3862dd1526723246b20206e5fc37
what is up, doc?
```

```
---
```

Вот и всё, мы создали корректный блоб объект для Git.

Все другие объекты создаются аналогичным образом, меняется лишь запись о типе в заголовке: «blob», «commit» либо «tree». Стоит добавить, что блоб может иметь практически любое содержимое, однако содержимое объектов деревьев и коммитов записывается в очень строгом формате.

Ссылки в Git

Если вас интересует история репозитория начиная с определённого коммита, например `1a410e`, то для её отображения вы можете воспользоваться командой `git log 1a410e`, однако при этом вам всё ещё необходимо помнить хеш коммита `1a410e`, который является начальной точкой истории. Было бы неплохо, если бы существовал файл, в который можно было бы сохранить значение SHA-1 под простым именем, а затем использовать это имя вместо хеша SHA-1.

В Git такие файлы называются ссылками («references» или, сокращённо, «refs») и расположены в каталоге `.git/refs`. В нашем проекте этот каталог пока пуст, но в нём 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, технически, достаточно выполнить примерно следующее:

```
$ echo 1a410efbd13591db07496601ebc7a059dd55cfe9 > .git/refs/heads/master
```

Теперь в командах Git вместо SHA-1 можно использовать только что созданную ссылку:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Тем не менее, редактировать файлы ссылок вручную не рекомендуется, вместо этого Git предоставляет более безопасную команду `update-ref` на случай, если вам потребуется изменить ссылку:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Вот что такое, по сути, ветка в Git — простой указатель или ссылка на последний коммит в цепочке. 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
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Теперь база данных Git схематично выглядит так, как показано на рисунке:

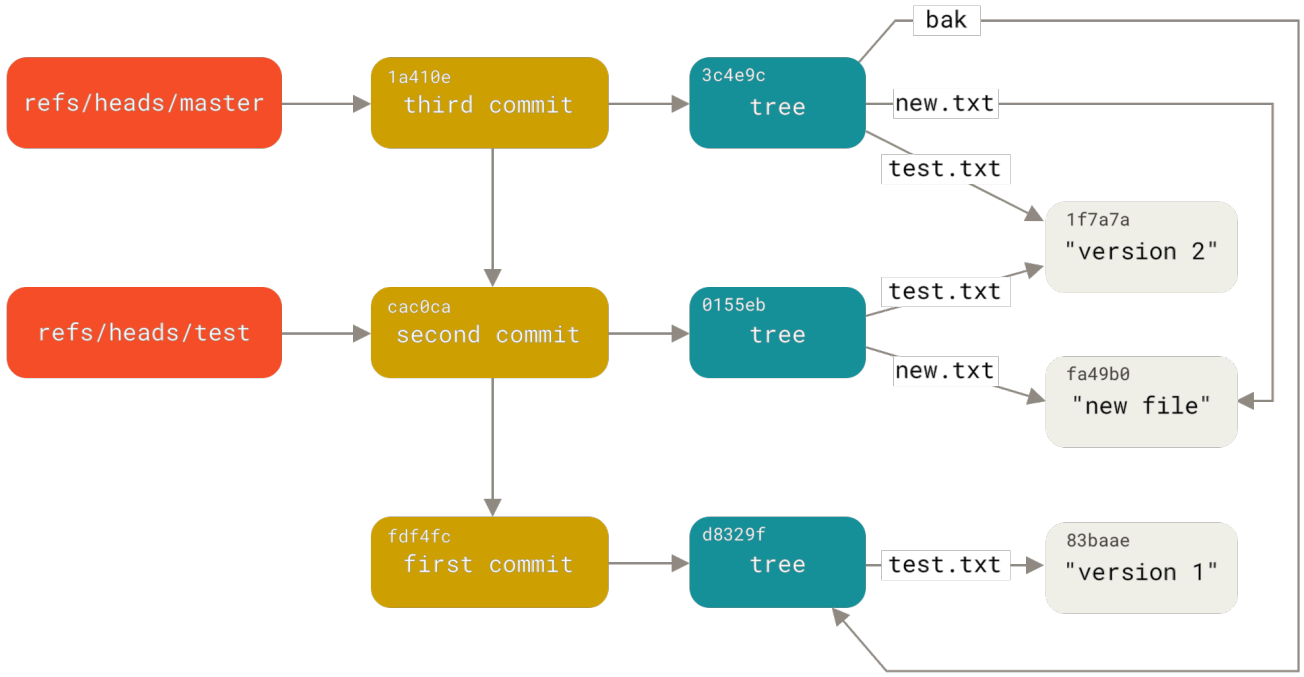


Рисунок 150. Объекты в каталоге .git, а также указатели на вершины веток

При выполнении команды `git branch <branch>`, в действительности Git запускает команду `update-ref`, которая добавляет SHA-1 хеш последнего коммита текущей ветки в файл с именем указанной ветки.

HEAD

Как же Git получает хеш последнего коммита при выполнении `git branch <имя ветки>`? Ответ кроется в файле HEAD.

Файл HEAD — это символическая ссылка на текущую ветку. Символическая ссылка

отличается от обычной тем, что она содержит не сам хеш SHA-1, а указатель на другую ссылку.

В некоторых случаях файл HEAD может содержать SHA-1 хеш какого-либо объекта. Это происходит при извлечении тега, коммита или удалённой ветки, что приводит репозиторий в состояние "detached HEAD".

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/
```

Теги

Мы рассмотрели три основных типа объектов Git, но есть ещё один. Объект тега очень похож на объект коммита: он содержит имя своего автора, дату, сообщение и указатель. Разница же в том, что объект тега указывает на коммит, а не на дерево. Он похож на ветку, которая никогда не перемещается: он всегда указывает на один и тот же коммит, просто давая ему понятное имя.

Как мы знаем из главы Основы Git, теги бывают двух типов: аннотированные и легковесные. Легковесный тег можно создать следующей командой:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Вот и всё, легковесный тег — это ветка, которая никогда не перемещается. Аннотированный тег имеет более сложную структуру. При создании аннотированного тега Git создаёт специальный объект и указывающую на него ссылку, а не просто указатель на коммит. Мы можем увидеть это, создав аннотированный тег, используя опцию `-a`:

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'Test tag'
```

Вот значение SHA-1 созданного объекта:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Теперь выполним `git cat-file -p` 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, что 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 сопровождающий проекта 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 тег, указывающий не на коммит: самый первый тег 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 каталоге `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 (ссылок в `refs/heads`) тем, что они считаются неизменяемыми. Это означает, что вы можете переключиться на любую из таких веток с помощью `git checkout`, но Git не установит HEAD на неё, а значит вы не сможете фиксировать свои изменения в ней с помощью `git commit`. Git воспринимает удалённые ветки как закладки на последние известные состояния веток на удалённых серверах.

Pack-файлы

Если вы следовали всем инструкциям из примеров предыдущего раздела, то теперь ваш тестовый репозиторий должен содержать 11 объектов: 4 блоба, 3 дерева, 3 коммита и один тег:

```
$ 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/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.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
```

Git использует zlib для сжатия содержимого этих файлов; к тому же у нас не так 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 checkout master
$ git add repo.rb
$ git commit -m 'Create repo.rb'
[master 484a592] Create 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 посмотреть на полученное дерево, мы увидим значение SHA-1 блоба для файла `repo.rb`:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.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' >> repo.rb
$ git commit -am 'Modify repo.rb a bit'
[master 2431da6] Modify repo.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      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Теперь файлу `repo.rb` соответствует совершенно другой блоб; это означает, что после добавления всего одной единственной строки в конец 400-строчного файла, Git сохранит новый контент в отдельный объект:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

Итак, мы имеем два практически одинаковых объекта, занимающих по 22 Кб на диске (в сжатом виде — приблизительно 7 Кб каждый). 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 для сохранения места на диске и повышения эффективности. Это происходит, когда «рыхлых» объектов становится слишком много, а также при ручном вызове `git gc` или 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 заглянуть в каталог с объектами, то можно обнаружить, что большинство объектов исчезло, 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, содержащие строки «what is up, doc?» и «test content». В силу того, что ни в одном коммите данные файлы не присутствуют, они считаются «висячими» и в pack-файл не включаются.

Остальные файлы — это pack-файл и его индекс. Pack-файл — это файл, в котором теперь находится содержимое всех удалённых объектов. Индекс — это файл, в котором записаны смещения для быстрого доступа к содержимому прежних объектов. Упаковка данных положительно повлияла на общий размер файлов: если до вызова команды `gc` в сжатом виде они занимали примерно 15 Кб, то pack-файл занимает всего 7 Кб. За счёт упаковки объектов мы только что освободили как минимум половину 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-
```

```
978e03944f5c581011e6998cd0e9e30000905586.idx
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 `repo.rb`, ссылается на блоб `b042a`, который хранит вторую его версию. Третья колонка в выводе — это размер содержимого объекта в pack-файле; 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 вручную используя `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 простым смещением.

По умолчанию, после выполнения `git remote add origin`, 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`, историю данной ветки можно получить, выполнив любую из следующих команд:

```
$ 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 конкретное соответствие веток 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 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

простым смещением вперёд. 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` из репозитория `origin`, добавьте следующие 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
```

Начиная с версии Git 2.6.0 можно указывать шаблоны спецификаций, соответствующие нескольким веткам:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Для достижения аналогичного результата можно так же использовать пространства имён (или каталоги). 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
```

Аналогично, это приведёт к тому, что при вызове `git push origin` локальная ветка `master` будет по умолчанию отправляться в удалённую ветку `qa/master`.

Вы не можете использовать спецификации ссылок, чтобы получать данные из одного репозитория, а отправлять в другой. Для реализации такого поведения обратитесь к разделу Поддержание GitHub репозитория в актуальном состоянии главы 6.

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.

Начиная с версии Git v1.7.0, можно использовать следующий синтаксис:

```
$ git push origin --delete topic
```

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 наличия на сервере структуры каталогов аналогичной Git репозиторию.

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`, поэтому 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

Change 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 коммита. Теперь получим 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 возможность для проектов-форков, позволяющая устранить дублирование объектов на диске. Так как в данном случае альтернатив нет, объект должен быть упакован в pack-файле. Чтобы посмотреть доступные на сервере pack-файлы, нужно скачать файл `objects/info/packs`, содержащий их список (также генерируется 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)
```

Так как в индексе содержится список SHA-1 хешей объектов и соответствующих им смещений объектов внутри pack-файла, то можно проверить наличие объекта в этом pack-файле. 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 только что скачанного pack-файла, так что снова обращаться к серверу не надо. Git извлекает рабочую копию ветки `master`, так как на неё указывает ссылка `HEAD`, которая была скачана 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 наличия на сервере специального процесса, знающего о структуре Git репозитория, умеющего выяснять, 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'"
00a5ca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status \
    delete-refs side-band-64k quiet ofs-delta \
    agent=git/2:2.1.1+github-607-gfba4028 delete-refs
0000
```

Команда `git-receive-pack` тут же посылает 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 идентификатор клиента).

Данные передаются пакетами. Каждый пакет начинается с 4-байтового шестнадцатеричного значения, определяющего его размер (включая эти 4 байта). Пакеты обычно содержат одну строку данных и завершающий символ переноса строки. Первый пакет начинается с 00a5, что в десятичной системе равно 165 и означает, что размер пакета составляет 165 байт. Следующий пакет начинается с 0000, что говорит об окончании передачи списка ссылок сервером.

Теперь, когда `send-pack` выяснил состояние сервера, он определяет коммиты, которые есть локально, но отсутствуют на сервере. Эту информацию процесс `send-pack` передаёт процессу `receive-pack` по каждой ссылке, которая подлежит отправке. Например, если мы обновляем ветку `master` и добавляем ветку `experiment`, ответ `send-pack` будет выглядеть следующим образом:

```
0076ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6
\
```

```
    refs/heads/master report-status
006c0000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d
\
    refs/heads/experiment
0000
```

Для каждой обновляемой ссылки Git посылает по строке, содержащей собственную длину, старый хеш, новый хеш и имя ссылки. В первой строке также посылаются возможности клиента. Хеш, 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 посылает pack-файл с объектами, которых нет на сервере. Наконец, сервер передаёт статус операции — успех или ошибка:

```
000eunpack 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.

Имейте в виду, что HTTP протокол может дополнительно кодировать данные внутри каждого пакета.

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 HEAD\0multi_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
003fe2409a098dc3e53539a9028a94b6224db9d6a6b6 refs/heads/master
0000
```

Это очень похоже на ответ `receive-pack`, но только возможности другие. Вдобавок `upload-pack` отсылает обратно ссылку HEAD (`symref=HEAD:refs/heads/master`), чтобы клиент понимал, на какую ветку переключиться, если выполняется клонирование.

На данном этапе процесс `fetch-pack` смотрит на имеющиеся в наличии объекты, а для недостающих объектов отвечает словом «want» с указанием SHA-1 необходимого объекта. Для каждого из имеющихся объектов процесс отправляет слово «have» с указанием SHA-1 объекта. В конце списка он пишет «done», что указывает процессу `upload-pack` начать отправлять pack-файл с необходимыми данными:

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

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
00e7ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_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
003fca82a6dff817ec66f44342007202690a93763949 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 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

Используется тот же формат, что и ранее. В ответ сервер посылает статус операции и сгенерированный pack-файл.

Заключение

В этом разделе мы вкратце рассмотрели протоколы передачи данных. Протоколы обмена данных в Git включают в себя множество возможностей, таких как `multi_ack` или `side-band`, но их рассмотрение 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.

Обслуживание репозитория 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.

Обслуживание репозитория

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 недостижимые объекты, хранящиеся дольше нескольких месяцев.

Сборку мусора можно запустить вручную следующим образом:

```
$ 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 каталога `refs`. Git перенесёт их в файл `.git/packed-refs` в угоду эффективности:

```
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d 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 каталоге `refs`, а к файлу `packed-refs` обратится только в случае отсутствия оного. Так что, если вы не можете найти ссылку в каталоге `refs`, скорее всего она упакована в файле `packed-refs`.

Обратите внимание, последняя строка файла начинается с `^`. Это означает, что предыдущая строка является аннотированным тегом, а текущая строка — это коммит, на который указывает аннотированный тег.

Восстановление данных

В какой-то момент при работе с Git вы можете нечаянно потерять коммит. 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 коммитов, которые затем понадобились. Как же в таком случае вернуть свои коммиты обратно?

Ниже приведён пример, в котором мы сбрасываем ветку `master` с потерей данных до более раннего состояния, а затем восстанавливаем потерянные коммиты. Для начала, давайте посмотрим, как сейчас выглядит история изменений:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo.rb a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Теперь сбросим ветку `master` на третий коммит:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef Third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Итак, теперь два последних коммита по-настоящему потеряны — они не достижимы ни из одной ветки. Необходимо найти SHA-1 хеш последнего коммита и создать ветку, указывающую на него. 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 коммитов 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: Modify repo.rb a bit
484a592 HEAD@{2}: commit: Create repo.rb
```

Здесь мы видим два коммита, на которые когда-то указывал HEAD, однако информации не так уж и много. Для получения информации в более удобном виде, можно воспользоваться командой `git log -g`, которая выведет лог записей из reflog в привычном формате:

```
$ 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

       Modify repo.rb a bit
```

Похоже, что последний коммит — это и есть тот, который мы потеряли; его можно восстановить, создав новую ветку, указывающую на него. Например, создадим новую ветку с именем `recover-branch`, указывающую на этот коммит (`ab1afef`):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo.rb a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Отлично — теперь у нас есть ветка `recover-branch`, указывающая туда, куда ранее указывала ветка `master`, тем самым делая потерянные коммиты вновь доступными. Теперь предположим, что потерянные изменения отсутствуют в reflog — для симуляции такого состояния удалим восстановленную ветку и очистим reflog.

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

В этом случае два первых коммита недоступны ниоткуда. Так как данные reflog хранятся в каталоге `.git/logs/`, которую мы только что удалили, то теперь у нас нет reflog. Как теперь восстановить коммиты? 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 нашем случае потерянный коммит указан после слов «dangling commit» («висячий коммит»). Его можно восстановить аналогичным образом, создав новую ветку, указывающую на этот SHA-1.

Удаление объектов

Git — замечательный инструмент с кучей классных возможностей, но некоторые из них способны стать источником проблем; 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 следующем коммите. 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 переписывают историю изменений. Каждый коммит, начиная с самого раннего, из которого нужно удалить большой файл, будет переписан. Если сделать это непосредственно после импорта, пока никто ещё не работал с репозиторием, то всё окей, иначе придётся сообщить всем участникам о необходимости перебазирования их правок относительно ваших новых коммитов.

Для примера добавим большой файл в тестовый репозиторий, удалим его в следующем коммите, а потом найдём и удалим его полностью из базы. Сначала 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 избавиться от этого файла:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'Oops - remove large tarball'
[master dadf725] Oops - remove large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Теперь запустим сборщик мусора и посмотрим, сколько места сейчас используется:

```
$ 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 последним коммитом использовалось около 2 КБ — очевидно, удаление файла не удалило его из истории. Всякий раз, когда кто-либо захочет клонировать этот репозиторий, ему придётся скачивать все 5 МБ для того, чтобы заполучить этот крошечный проект, просто потому, что однажды вы имели неосторожность добавить большой файл. Давайте же исправим это!

Для начала найдём проблемный файл. В данном случае, мы уже знаем, 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 внизу списка, его размер — 5 МБ. Для того чтобы узнать, что это за файл, воспользуемся командой `rev-list`, которая уже упоминалась в разделе Проверка формата сообщения коммита главы 8. Если передать ей ключ `--objects`, она выдаст хеши всех коммитов, 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 коммиты, которые изменяли данный файл:

```
$ git log --oneline --branches -- git.tgz
dadf725 Oops - remove large tarball
```

```
7b30847 Add git tarball
```

Для полного удаления этого файла из истории необходимо переписать все коммиты, начиная с `7b30847`. Воспользуемся командой `filter-branch`, о которой мы писали в разделе Перезапись истории главы 7:

```
$ 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 разделе Перезапись истории главы 7, 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 коммита `7b30847`, потому что мы знаем, что именно в нём впервые появилась проблема. По умолчанию перезапись начинается с самого первого коммита, 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 затем переупаковать базу. Необходимо избавиться от всех возможных ссылок на старые коммиты 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 окружения

Git всегда запускается в оболочке `bash` и использует ряд переменных окружения, определяющих его поведение. Полезно знать, какие из них и как использовать, чтобы настроить поведение Git так как нужно вам. Мы не будем приводить исчерпывающий список переменных окружения, используемых Git, а рассмотрим самые полезные из них.

Глобальное поведение

Поведение Git как компьютерной программы определяется переменными окружения.

`GIT_EXEC_PATH` определяет где Git будет искать свои подпрограммы (такие как `git-commit`, `git-diff` и другие). Текущие настройки можно узнать командой `git --exec-path`.

`HOME` обычно не рассматривается в качестве изменяемого параметра (чересчур много вещей от неё зависят), но именно тут Git ищет глобальный файл конфигурации. Если вам нужна по-настоящему портируемая 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` отключает использование файла общесистемной конфигурации, если задана. 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 постраничного вывода в командной строке. Если не задана, в качестве запасного варианта используется `PAGER`.

`GIT_EDITOR` редактор, который Git запустит, когда пользователю понадобится отредактировать какой-нибудь текст (например, сообщение коммита). Если не задана — используется `EDITOR`.

Расположение репозитория

Git использует некоторые переменные окружения, чтобы определить как взаимодействовать с текущим репозиторием.

`GIT_DIR` — это месторасположение каталога `.git`. Если эта переменная не задана, Git будет переходить вверх по дереву каталогов, пока не достигнет `~` (домашнего каталога пользователя) или `/` (корневого каталога), проверяя на каждом шагу наличие каталога `.git`.

`GIT_CEILING_DIRECTORIES` управляет процессом поиска каталога `.git`. Если вы работаете с медленной файловой системой (типа ленточного накопителя или сетевого каталога), вы можете запретить Git доступ к `.git` без надобности, например, для построения строки приветствия.

`GIT_WORK_TREE` — это путь к корневому рабочему каталогу для не-серверного репозитория (с непустым рабочим каталогом). Если задана `GIT_DIR` или указан параметр `--git-dir`, но ничего из `--work-tree`, `GIT_WORK_TREE` или `core.worktree` не задано, то текущий каталог будет считаться корневым.

`GIT_INDEX_FILE` — это путь к файлу индекса (только для репозиториев с непустым рабочим каталогом).

`GIT_OBJECT_DIRECTORY` может быть использована для указания каталога с объектами вместо `.git/objects`.

`GIT_ALTERNATE_OBJECT_DIRECTORIES` — это список разделённых двоеточием каталогов (типа `/dir/one:/dir/two:…`), в которых Git будет пытаться найти объекты, которых нет в `GIT_OBJECT_DIRECTORY`. Это может быть полезно, если у вас много проектов с большими файлами с абсолютно одинаковым содержимым, что позволит не хранить много дубликатов.

Пути к файлам

Эти переменные определяют 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 как шаблон (поведение по умолчанию); 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 работать не будут.

`GIT_ICASE_PATHSPECS` делает указание любого пути регистронезависимым.

Фиксация изменений

Окончательное создание объекта коммита обычно производится командой `git-commit-tree`, которая использует приведённые ниже переменные окружения в качестве основного источника информации и прибегает к использованию файлов конфигурации только если эти переменные не заданы.

`GIT_AUTHOR_NAME` используется для указания имени автора коммита.

`GIT_AUTHOR_EMAIL` задаёт адрес электронной почты автора коммита.

`GIT_AUTHOR_DATE` метка времени на момент создания коммита.

`GIT_COMMITTER_NAME` используется для указания имени, применившего коммит.

`GIT_COMMITTER_EMAIL` задаёт адрес электронной почты, применившего коммит.

`GIT_COMMITTER_DATE` метка времени на момент применения коммита.

`EMAIL` используется, как запасное значение, если конфигурационный параметр `user.email` не задан. Если же и *эта переменная* не задана, Git будет использовать имя пользователя в системе и имя хоста.

Работа с сетью

Git использует библиотеку `curl` для работы с сетью через HTTP, поэтому `GIT_CURL_VERBOSE` указывает Git выводить все сообщения, генерируемые этой библиотекой. Это аналогично использованию `curl -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 переопределяют значения конфигурационных параметров `http.lowSpeedLimit` и `http.lowSpeedTime`.

`GIT_HTTP_USER_AGENT` задаёт заголовок `User-Agent` при работе через HTTP. По умолчанию используется значение вида `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 детализации вывода при рекурсивном слиянии. Возможные значения перечислены ниже:

- 0 не выводить ничего, кроме возможного единственного сообщения об ошибке.
- 1 выводить только конфликты.
- 2 также выводить изменения файлов.
- 3 показывать пропущенные файлы, в которых нет изменений.
- 4 выводить все пути в том же порядке как они обрабатываются.
- 5 и выше выводят подробную отладочную информацию.

По умолчанию значение переменной равно 2.

Отладка

Хотите знать что *на самом деле* делает Git? Git ведёт достаточно подробный логи выполняемых действий и всё что вам нужно — включить их. 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 подпадающих под какую-либо определённую категорию. Это включает в себя раскрытие 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'
'--pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.879885 git.c:349               trace: built-in: git 'log' '--graph' '--
pretty=oneline' '--abbrev-commit' '--decorate' '--all'
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-
pack
20:15:14.867071 pkt-line.c:46           packet:          git< 0000
20:15:14.867079 pkt-line.c:46           packet:          git<
97b8860c071898d9e162678ea1035a8ced2f8b1f HEAD\0multi_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/2.0.4
20:15:14.867088 pkt-line.c:46           packet:          git<
0f20ae29889d61f2e93ae00fd34f1cdb53285702 refs/heads/ab/add-interactive-show-diff-func-
name
20:15:14.867094 pkt-line.c:46           packet:          git<
36dc827bc9d17f80ed4f326de21247a5d1341fbc refs/heads/ah/doc-gitk-config
# […]
```

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: 'git'
'pack-refs' '--all' '--prune'
20:18:19.845585 trace.c:414             performance: 0.343020000 s: git command: 'git'
'reflog' 'expire' '--all'
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: 'git'
'pack-objects' '--keep-true-parents' '--honor-pack-keep' '--non-empty' '--all' '--
reflog' '--unpack-unreachable=2.weeks.ago' '--local' '--delta-base-offset'
'.git/objects/pack/.tmp-49190-pack'
20:18:23.584728 trace.c:414             performance: 0.000910000 s: git command: 'git'
'prune-packed'
20:18:23.605218 trace.c:414             performance: 0.017972000 s: git command: 'git'
'update-server-info'
20:18:23.606342 trace.c:414             performance: 3.756312000 s: git command: 'git'
'repack' '-d' '-l' '-A' '--unpack-unreachable=2.weeks.ago'
Checking connectivity: 170994, done.
20:18:25.225424 trace.c:414             performance: 1.616423000 s: git command: 'git'
'prune' '--expire' '2.weeks.ago'
20:18:25.232403 trace.c:414             performance: 0.001051000 s: git command: 'git'
'rerere' 'gc'
20:18:25.233159 trace.c:414             performance: 6.112217000 s: git command: 'git'
'gc'
```

`GIT_TRACE_SETUP` задаёт журналирование информации о репозитории и окружении, с которым взаимодействует 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` — если значение задано, указанная программа будет использоваться вместо `ssh` при попытке Git подключиться по SSH протоколу. Формат вызова этой программы такой: `$GIT_SSH [имя пользователя@]хост [-p <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 путь к нему в `GIT_SSH`. Возможно, проще будет использовать `~/.ssh/config`.

`GIT_ASKPASS` заменяет значение конфигурационного параметра `core.askpass`. Git вызывает эту программу каждый раз, когда требуется запросить у пользователя пароль. Строка с текстом запроса передаётся этой программе как параметр командной строки, а введённое пользователем значение она должна передать в стандартный поток вывода `stdout`. (Читайте подробнее в разделе Хранилище учётных данных главы 7.)

`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 в нём же, разделяя лишь ссылки по пространствам имён.

`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. Если ничего не задано, по умолчанию используемое значение 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 более низком уровне поможет вам лучше понять его работу 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.

Приложение A: 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 людьми. 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 ним.

Графические интерфейсы

Родная среда обитания Git — это терминал. Новые функции изначально 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 той функциональностью, которую их автор считает наиболее востребованной или эффективной. Учитывая это, ни один из представленных ниже инструментов не может быть «лучше» остальных: они просто предназначены для решения разных задач. Также стоит помнить, что всё, что можно сделать с помощью графического интерфейса, может быть выполнено и из консоли; командная строка по прежнему является местом, где у вас больше всего возможностей и контроля над вашими репозиториями.

`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 grep`. Это тот инструмент, который вы будете использовать для поиска событий или визуализации истории.

Проще всего вызвать Gitk из командной строки: Просто перейдите в каталог с репозиторием и наберите:

```
$ gitk [git log options]
```

Gitk принимает много различных опций, большинство из которых транслируются в используемый `git log`. Возможно, наиболее полезная опция — `--all`, которая указывает Gitk выводить коммиты, доступные из *любой* ссылки, а не только HEAD. Интерфейс Gitk выглядит так:

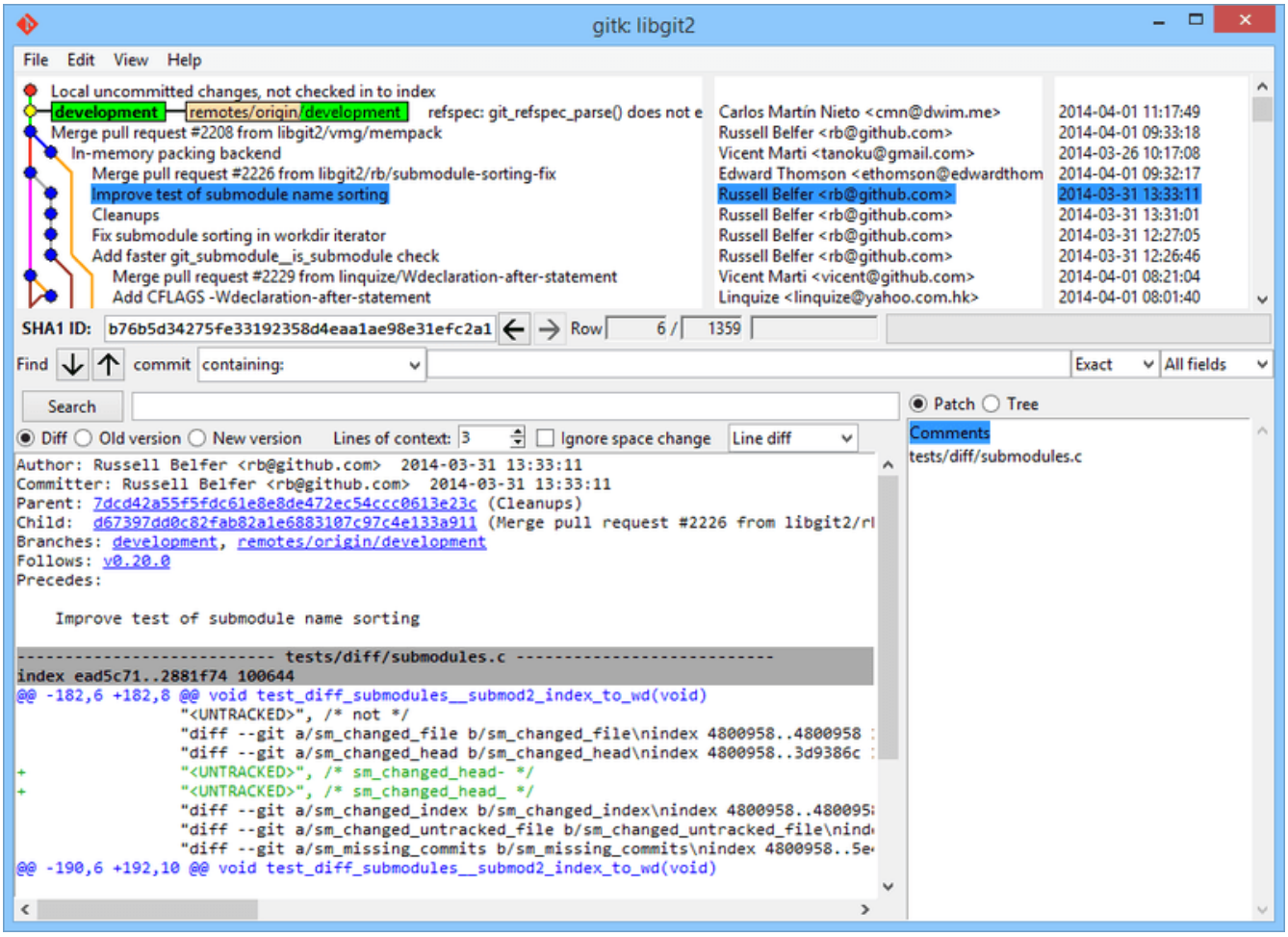


Рисунок 151. `gitk`*- инструмент для просмотра истории*

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. В нижней части расположены элементы интерфейса для просмотра выделенного коммита: слева показаны изменения и комментарий, а справа — общая информация по изменённым файлам. В центре 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 выглядит так:

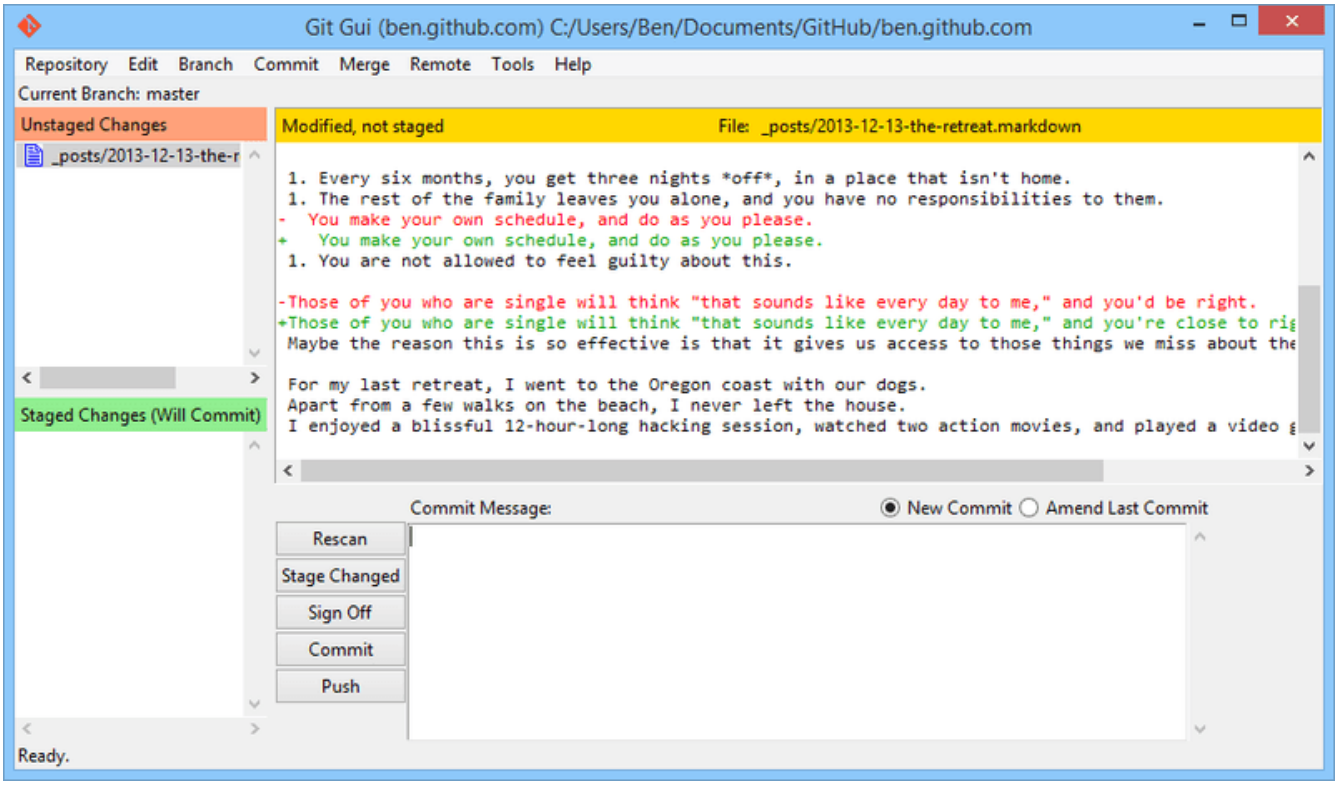


Рисунок 152. `git gui` *— инструмент редактирования коммитов*

Слева находится область редактирования Git индекса: изменения в рабочем каталоге наверху, добавленные в индекс изменения — снизу. Вы можете перемещать файлы целиком между двумя состояниями, кликая на иконки, или же вы можете просмотреть изменения в конкретном файле, кликнув по нему.

Справа вверху расположена область просмотра изменений выделенного файла. Можно добавлять отдельные 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», что приведёт к обновлению индекса содержимым предыдущего коммита. После этого вы можете как обычно добавлять или удалять файлы из индекса, изменить сообщение коммита и, нажав кнопку «Commit» создать новый коммит, заменив им предыдущий.

`gitk` и `git-gui` — это примеры инструментов, ориентированных на задачи. Каждый из них наиболее подходит для решения определённой задачи (просмотр истории или создание коммитов соответственно) и не поддерживает дополнительные функции Git, ненужные для её решения.

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 всей функциональности Git, они концентрируются на небольшом наборе фич, работающих вместе для достижения цели. Выглядят они примерно так:

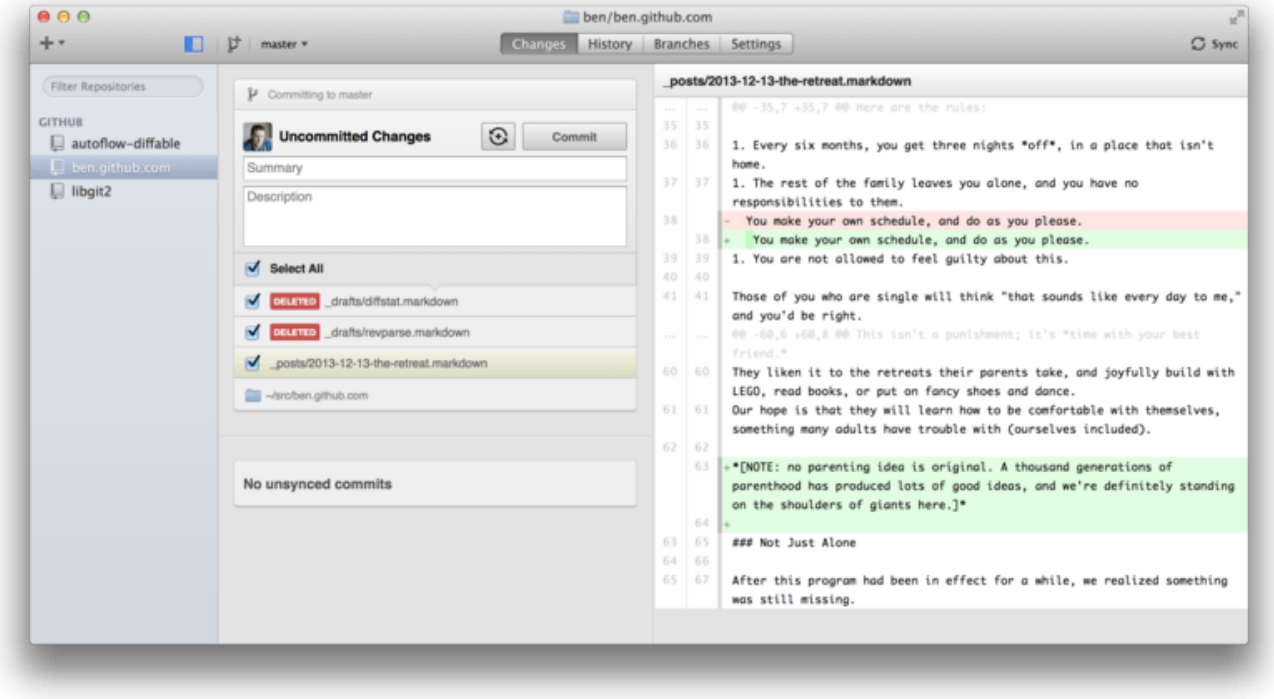


Рисунок 153. GitHub для Mac

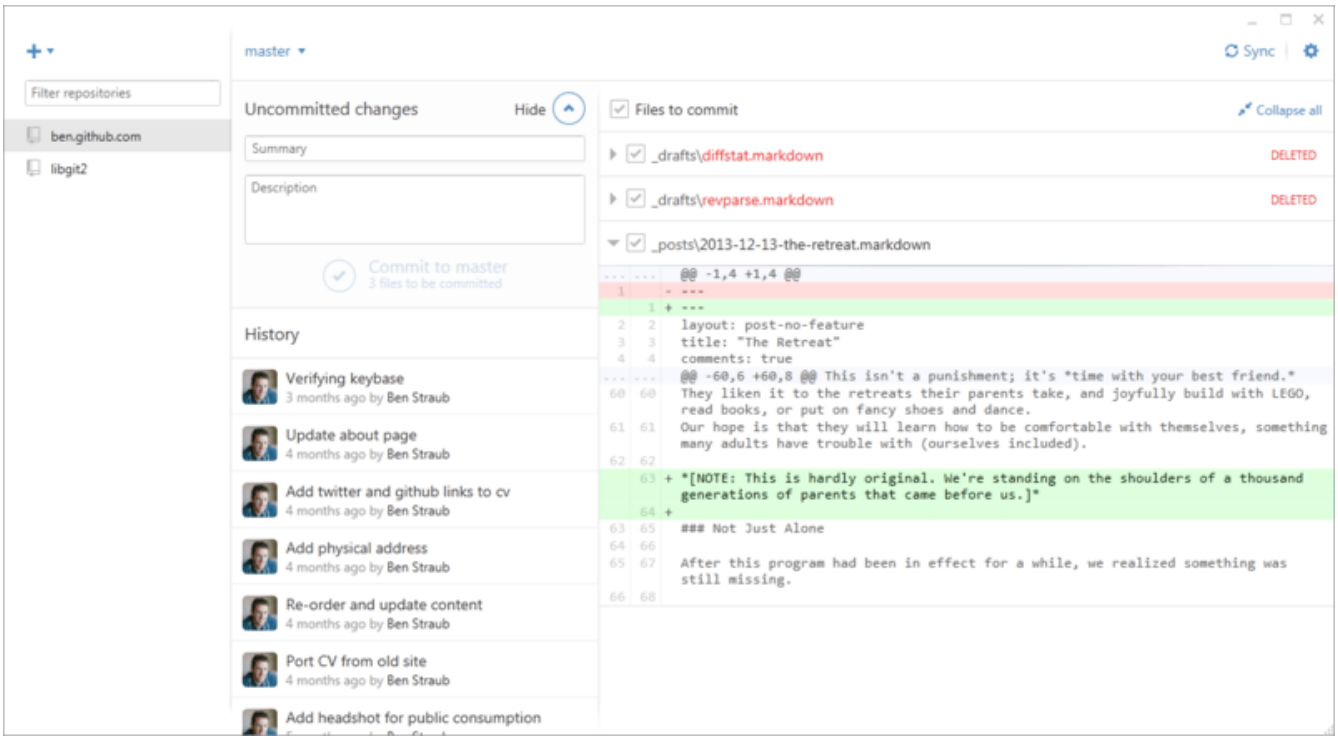


Рисунок 154. GitHub для Windows

Они спроектированы по одному шаблону, поэтому мы будет 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 (клонировав 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 рабочем каталоге, 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 установят 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 — 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 клонировать любой в один клик. Если же у вас уже есть клонированный 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 разделе Рабочий процесс с использованием GitHub главы 6, но вкратце, важны два момента: (а) вы коммитите в отдельные ветки и (б) вы регулярно забираете изменения с удалённого репозитория.

Управление ветками слегка различается на Mac и Windows. В Mac версии для создания ветки есть кнопка вверху окна:

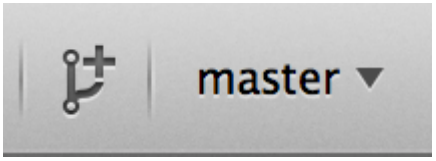


Рисунок 155. Кнопка создания ветки на Mac

На Windows создание ветки происходит путём ввода её имени в переключатель веток:

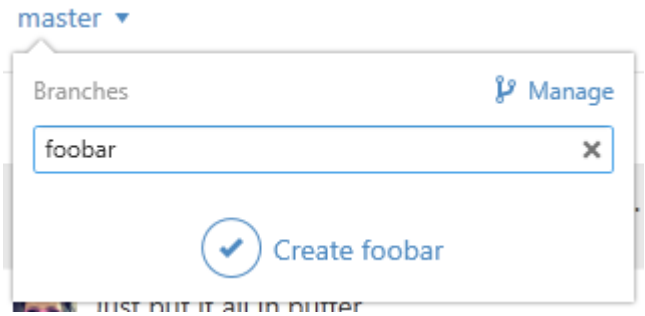


Рисунок 156. Создание ветки в Windows

После создания ветки добавление коммитов в неё довольно тривиально. Измените что-нибудь в рабочем каталоге 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 или □-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 всю функциональность Git. На официальном сайте Git поддерживается в актуальном состоянии список наиболее популярных оболочек: https://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.

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 этой функциональностью, откройте проект, который управляется 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:

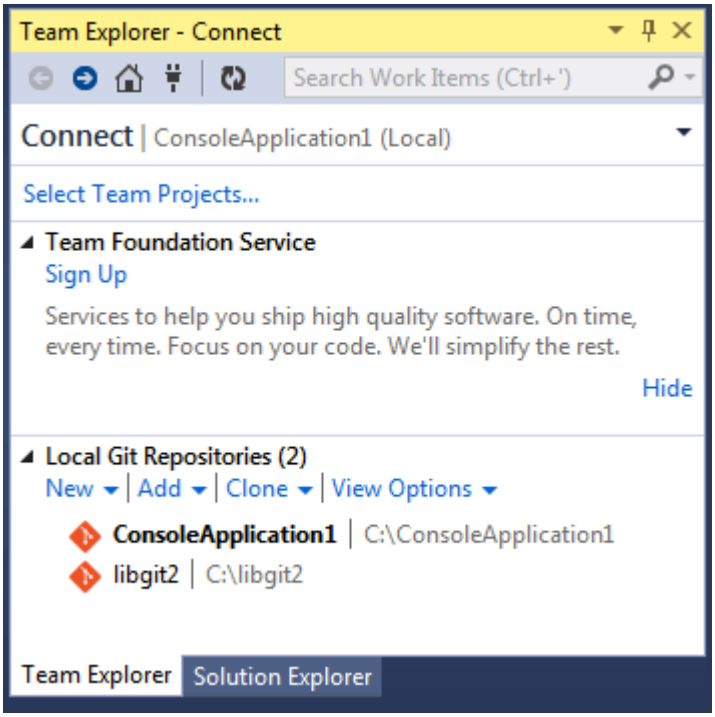


Рисунок 157. Подключение к Git-репозиторию из окна Team Explorer (Командный обозреватель)

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 рабочему каталогу. Двойной клик по одному из локальных Git-репозиториев откроет главную страницу репозитория, которая выглядит примерно так «Домашняя» страница Git-репозитория в Visual Studio.

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 коммиты») и «Branches» («Ветки»).

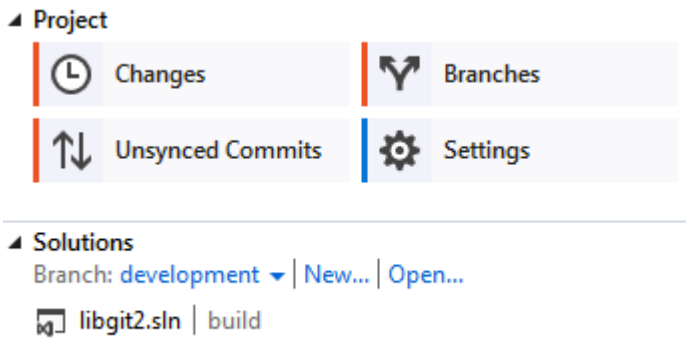


Рисунок 158. «Домашняя» страница Git-репозитория в Visual Studio

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 удалённых команд и множество других возможностей. Подробнее об использовании Git в Visual Studio: https://docs.microsoft.com/en-us/azure/devops/repos/git/command-prompt?view=azure-devops.

Git в Visual Studio Code

Visual Studio Code имеет встроенную поддержку Git. Вам потребуется установить Git версии не ниже чем 2.0.0.

Основные особенности:

- Просмотр изменений редактируемого файла
- Панель состояния Git (слева внизу), на которой отображается текущая ветка, индикатор ошибок, входящие и исходящие коммиты.
- В редакторе можно делать основные Git операции:
  - Инициализация репозитория.
  - Клонирование репозитория.
  - Создание веток и тегов.
  - Индексация изменений и создание коммитов.
  - Push/pull/sync с удалённой веткой.
  - Разрешение конфликтов слияния.
  - Просмотр изменений.
- С помощью плагина можно работать с запросами слияния на GitHub: https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-pull-request-github.

Официальная документация доступна здесь: https://code.visualstudio.com/Docs/editor/versioncontrol.

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`).

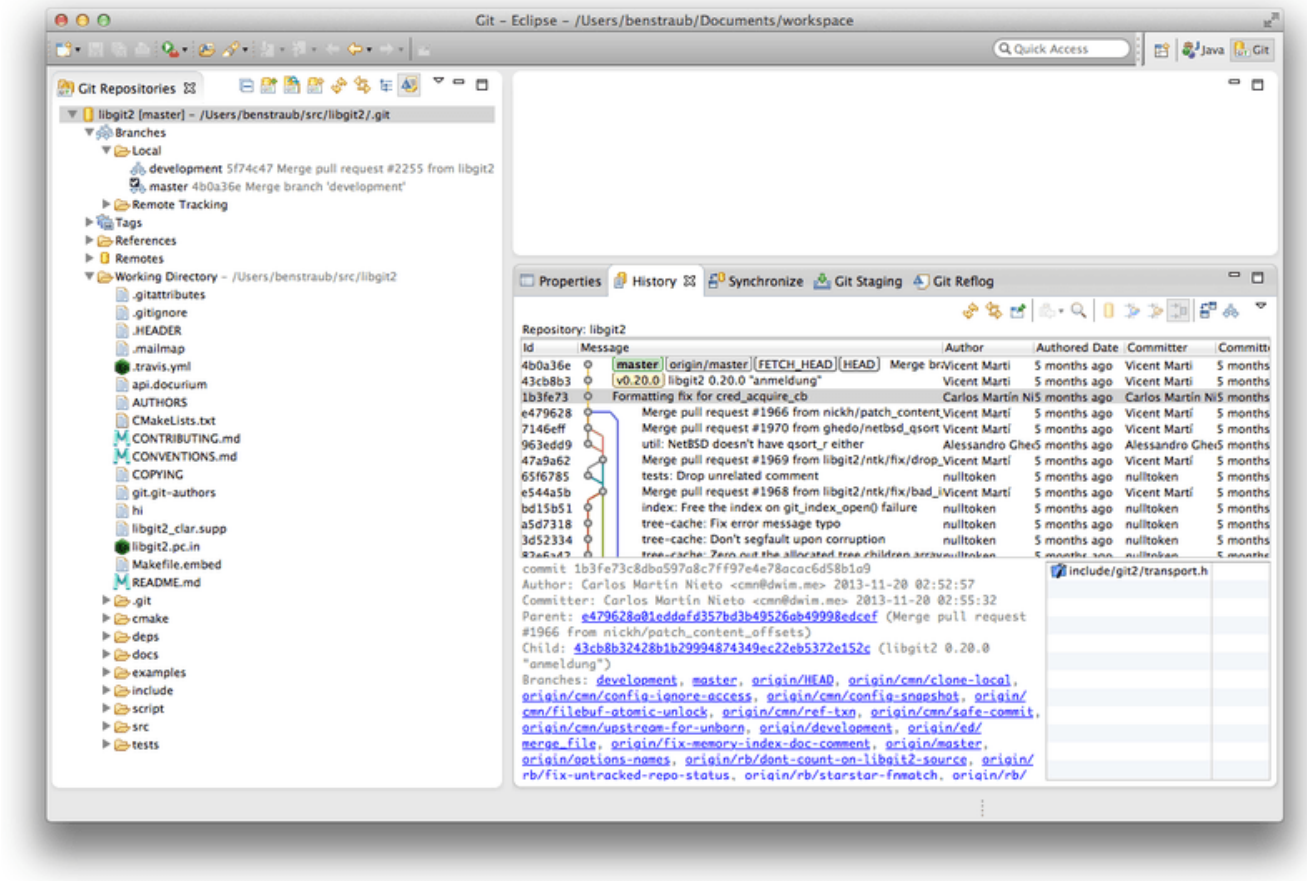


Рисунок 159. EGit в Eclipse

EGit поставляется с неплохой документацией, доступной меню через `Help` > `Help Contents` в разделе `EGit Documentation`.

Git в IntelliJ / PyCharm / WebStorm / PhpStorm / RubyMine

JetBrains IDE (такие как IntelliJ IDEA, PyCharm, WebStorm, PhpStorm, RubyMine и другие) поставляются с соответствующим плагином для интеграции с Git. Он предоставляет собственный интерфейс для работы с запросами слияния Git и GitHub.

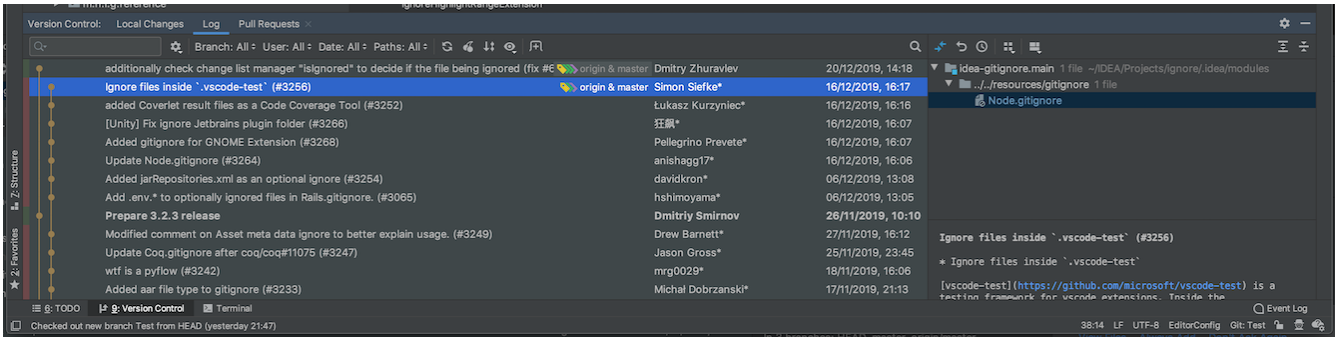


Рисунок 160. Окно версионного контроля в JetBrains IDE

Интеграция обеспечивается за счёт использования консольного Git клиента и требует его

наличия в системе. Официальная документация доступна по ссылке https://www.jetbrains.com/help/idea/using-git-integration.html.

Git в Sublime Text

Начиная с версии 3.2 в редактор Sublime Text встроена поддержка Git.

Основные особенности:

- На боковой панели git статус файлов и каталогов помечается бэйджем/иконкой.
- Файлы и каталоги, указанные в `.gitignore` не отображаются на боковой панели.
- В строке состояния отображается текущая ветка и количество внесённых изменений.
- Изменённые строки помечаются маркерами в канавке с нумерацией.
- Можно использовать некоторые функции git клиента Sublime Merge непосредственно из Sublime Text. (Это требует установки Sublime Merge: https://www.sublimemerge.com/)

Официальная документация для Sublime Text доступна здесь: https://www.sublimetext.com/docs/3/git_integration.html

Git в Bash

Если вы используете Bash, то можете задействовать некоторые из его фишек для облегчения работы с Git. К слову, Git поставляется с плагинами для нескольких командных оболочек, но они выключены по умолчанию.

Для начала, скачайте файл `contrib/completion/git-completion.bash` из репозитория с исходным кодом Git. Поместите его в укромное место — например, в ваш домашний каталог — и добавьте следующие строки в `.bashrc`:

```
. ~/git-completion.bash
```

Как только закончите с этим, перейдите в каталог 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, расположенном в текущем каталоге. Вы можете выводить сколь угодно сложную информацию, но обычно достаточно названия текущей ветки и статуса рабочего каталога. 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 текущий рабочий каталог, `\$` — 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 перейдёте в каталог с Git репозиторием:

Рисунок 161. Кастомизированная строка приветствия `bash`

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. Чтобы начать им пользоваться, просто добавьте строку `autoload -Uz compinit && compinit` в ваш `.zshrc` файл. Интерфейс Zsh более функциональный чем 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 из предложенных, перемещаясь по ним нажатием клавиши `Tab`. Это работает не только для команд Git, но и для их аргументов, названий объектов внутри репозитория (например, ссылки и удалённые репозитории), а также для имён файлов и прочего.

В состав Zsh входит фреймворк `vcs_info`, предназначенный для извлечения информации из систем контроля версий. Чтобы отобразить имя текущей ветки в правой строке приветствия, добавьте следующие строки в файл `~/.zshrc`:

```
autoload -Uz vcs_info
precmd_vcs_info() { vcs_info }
precmd_functions+=( precmd_vcs_info )
setopt prompt_subst
RPROMPT=\$vcs_info_msg_0_
# PROMPT=\$vcs_info_msg_0_'%# '
zstyle ':vcs_info:git:*' formats '%b'
```

В результате вы будете видеть имя текущей ветки в правой части окна терминала каждый раз, как перейдёте внутрь Git репозитория. (Для отображения названия ветки слева используйте `PROMPT` вместо `RPROMPT`) Результат выглядит примерно так:

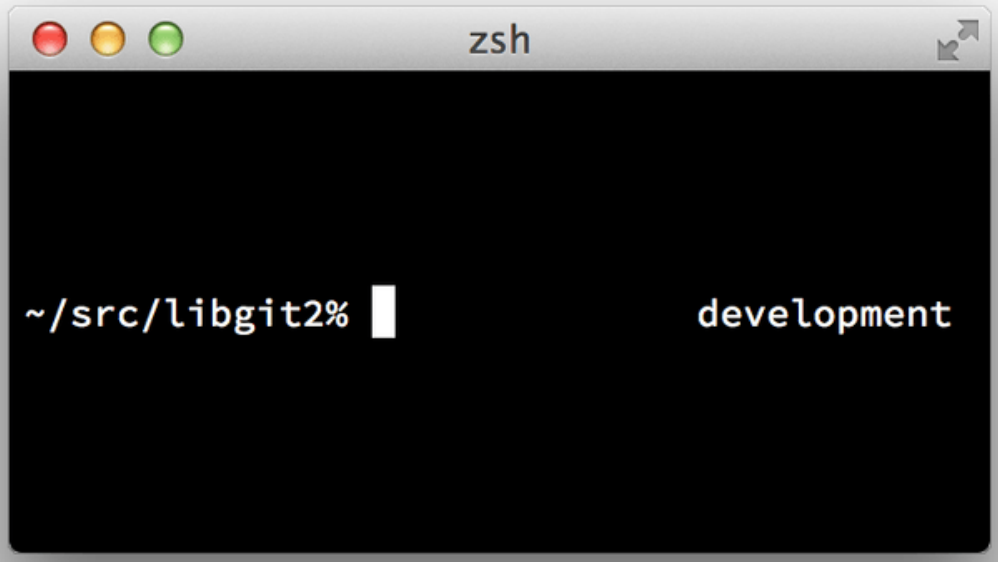


Рисунок 162. Кастомизированная строка приветствия в `zsh`

Дополнительную информацию о `vcs_info` можно найти в документации `zshcontrib(1)` или онлайн http://zsh.sourceforge.net/Doc/Release/User-Contributions.html#Version-Control-Information.

Возможно, вы предпочтёте использовать поставляемый вместе с Git скрипт настройки `git-prompt.sh`; детали использования приведены в комментариях к файлу https://github.com/git/git/blob/master/contrib/completion/git-prompt.sh. Скрипт `git-prompt.sh` совместим с обеими оболочками Bash и 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. Пример темы oh-my-zsh — лишь один из многих вариантов применения.

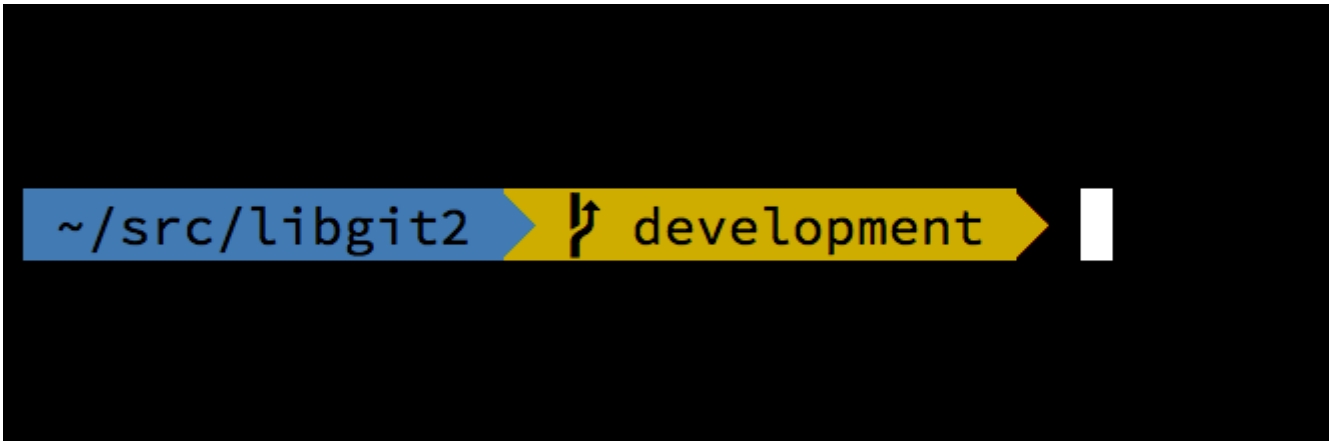


Рисунок 163. Пример темы oh-my-zsh

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. Это также применимо, если вы используете PowerShell Core на Linux или macOS. 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:

Рисунок 164. PowerShell с Posh-Git

Установка

Предустановки (только для Windows)

Для запуска PowerShell скриптов, вам необходимо установить значение локальной политики `ExecutionPolicy` в значение `RemoteSigned` (вообще-то, в любое значение, кроме `Undefined` или `Restricted`). Если вы установите значение в `AllSigned` вместо `RemoteSigned`, то для запуска локальных скриптов (владельцем которых являетесь вы) они должны иметь цифровую подпись. При использовании `RemoteSigned` должны иметь цифровую подпись

только те скрипты, у которых `ZoneIdentifier` установлен в значение `Internet` (скачены по сети). Если вы администратор и хотите установить значение политики для всех пользователей, добавьте флаг `-Scope LocalMachine`. Если вы обычный пользователь и не имеете прав администратора, то используйте флаг `-Scope CurrentUser` для применения политики только для текущего пользователя.

Подробнее о PowerShell Scopes: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_scopes

Подробнее о PowerShell ExecutionPolicy: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy

```
> Set-ExecutionPolicy -Scope LocalMachine -ExecutionPolicy RemoteSigned -Force
```

Галерея PowerShell

Если вы используете PowerShell 5 или PowerShell 4 с установленным PackageManagement, то Posh-Git можно установить с помощью пакетного менеджера.

Дополнительная информация о галерее PowerShell: https://docs.microsoft.com/en-us/powershell/scripting/gallery/overview

```
> Install-Module posh-git -Scope CurrentUser -Force
> Install-Module posh-git -Scope CurrentUser -AllowPrerelease -Force # Последняя бета версия с поддержкой PowerShell Core
```

Если вы хотите установить Posh-Git для всех пользователей в системе, то в команде выше используйте флаг `-Scope AllUsers`, а её выполнение следует производить с повышенными привилегиями. Если вторая команда завершится ошибкой типа `Module 'PowerShellGet' was not installed by using Install-Module`, то необходимо выполнить следующую команду:

```
> Install-Module PowerShellGet -Force -SkipPublisherCheck
```

А затем повторить установку. Это происходит потому, что поставляемые вместе с Windows PowerShell модули подписаны другим сертификатом.

Модификация приглашения ввода в PowerShell

Для добавления git информации в приглашение ввода необходимо импортировать модуль `Posh-Git`. Для автоматического импорта модуля Posh-Git при каждом запуске PowerShell, выполните команду `Add-PoshGitToProfile`, которая добавит инструкции импорта в ваш `$profile` скрипт. Этот скрипт выполняется каждый раз как вы открываете консоль PowerShell. Имейте в виду, что существует несколько `$profile` скриптов: например, один для консоли, а другой для ISE.

```
> Import-Module posh-git
```

```
> Add-PoshGitToProfile -AllHosts
```

Установка из исходников

Скачайте релиз Posh-Git с https://github.com/dahlbyk/posh-git/releases и распакуйте. Затем импортируйте модуль используя полный путь к файлу `posh-git.psd1`:

```
> Import-Module <путь-к-распакованному-каталогу>\src\posh-git.psd1
> Add-PoshGitToProfile -AllHosts
```

В файл `profile.ps1` будет добавлена соответствующая строка, за счёт которой Posh-Git будет подключаться каждый раз при запуске PowerShell.

Описание сводной информации о статусе Git, отображаемой в приглашении ввода, приведено здесь: https://github.com/dahlbyk/posh-git/blob/master/README.md#git-status-summary-information. Дополнительные сведения по настройке приглашения ввода приведены здесь: https://github.com/dahlbyk/posh-git/blob/master/README.md#customization-variables.

Заключение

Теперь вы знаете, как использовать мощь Git внутри инструментов, используемых вами каждый день и как получить доступ к репозиториям из ваших собственных программ.

Приложение B: Встраивание Git в ваши приложения

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, то у вас есть два основных варианта: запустить шелл и выполнять Git команды в нём или добавить библиотеку Git и использовать её. Ниже мы рассмотрим интеграцию командной строки и несколько наиболее популярных встраиваемых библиотек Git.

Git из командной строки

Первый вариант встраивания Git — 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, ориентирующаяся на предоставление отличного API другим программам. Вы можете найти её на https://libgit2.org.

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 рабочему каталогу репозитория или к каталогу `.git`. Кроме этого, существуют методы `git_repository_open_ext`, который принимает набор параметров для поиска репозитория, `git_clone` и сопутствующие — для создания локальной копии удалённого репозитория и `git_repository_init` — для создания нового репозитория с нуля.

Следующий блок кода использует синтаксис `rev-parse` (см. Ссылки на ветки), чтобы получить коммит, на который указывает HEAD. Возвращаемый тип является указателем на структуру `git_object`, которая представляет любой объект, хранящийся во внутренней базе данных Git. `git_object` является «родительским» для некоторых других типов; внутренняя структура всех этих типов одинаковая и соответствует `git_object`, 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 тип к `git_commit`.

Следующий блока кода показывает как получить доступ к свойствам 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, то она, скорее всего, 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 — несколько сложновато.

Последний пункт намекает на маловероятность использования C 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 тот же пример, только написанный с использованием Ruby и обёртки над Libgit2 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 освобождать ресурсы, потому что в Ruby есть встроенный сборщик мусора. Давайте посмотрим на более сложный пример — создание коммита с нуля:

```
blob_id = repo.write("Blob contents", :blob) ①

index = repo.index
index.read_tree(repo.head.target.tree)
index.add(:path => 'newfile.txt', :oid => blob_id) ②

sig = {
    :email => "bob@example.com",
    :name => "Bob User",
    :time => Time.now,
}

commit_id = Rugged::Commit.create(repo,
    :tree => index.write_tree(repo), ③
    :author => sig,
    :committer => sig, ④
    :message => "Add newfile.txt", ⑤
    :parents => repo.empty? ? [] : [ repo.head.target ].compact, ⑥
    :update_ref => 'HEAD', ⑦
)
commit = repo.lookup(commit_id) ⑧
```

① Создание нового blob, включающего содержимое нового файла.

② Заполнение индекса содержимым дерева HEAD и добавление нового файла `newfile.txt`.

③ Создание нового дерева в базе данных объектов и использование его для нового коммита.

④ Мы используем одну и ту же подпись для автора и коммитера.

⑤ Сообщение коммита.

⑥ При создании коммита нужно указать его предков. Для этих целей мы используем HEAD

как единственного родителя.

⑦ Rugged (как и Libgit2) дополнительно могут обновить HEAD при создании коммита.

⑧ Используя полученное значение SHA-1 хеша нового коммита, можно получить объект 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

Libgit2 обладает рядом возможностей, выходящих за рамки стандартного Git. Одна из таких возможностей — расширяемость: Libgit2 позволяет использовать нестандартные интерфейсы для ряда операций, таким образом вы можете хранить объекты по-иному, нежели это делает стандартный Git. 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 для базы данных 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, 1); ③

git_repository *repo;
error = git_repository_open(&repo, "some-path");
error = git_repository_set_odb(repo, odb); ④
```

(Заметьте, ошибки перехватываются, но не обрабатываются. Мы надеемся, ваш код лучше нашего.)

① Инициализация интерфейса для пустой базы данных объектов, который будет использоваться как контейнер для внутренних интерфейсов, которые будут выполнять работу.

② Инициализация произвольного внутреннего интерфейса базы данных объектов.

③ Добавление внутреннего интерфейса к внешнему.

④ Открытие репозитория и настройка на использование собственной базы для поиска объектов.

Что же скрыто внутри `git_odb_backend_mine`? Это ваша собственная реализация базы данных объектов, где вы можете делать что угодно, лишь бы поля структуры `git_odb_backend` были

заполнены верно. Например, внутри *может* быть следующий код:

```
typedef struct {
    git_odb_backend parent;

    // Дополнительные 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: в `my_backend_struct` первое поле должно быть структурой `git_odb_backend`, что обеспечит расположение полей в памяти в формате, ожидаемом Libgit2. Оставшиеся поля можно располагать произвольно, а 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 -1`).

LibGit2Sharp

Если вы пишете под платформы .NET / Mono, LibGit2Sharp (https://github.com/libgit2/

libgit2sharp) — то, что вы искали. 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 https://www.pygit2.org. И наш пример будет выглядеть так:

```
pygit2.Repository("/path/to/repo") # открыть репозиторий
    .head                          # получить текущую ветку
    .peel(pygit2.Commit)           # получить коммит
    .message                       # прочитать сообщение
```

Дополнительные материалы

Конечно же, полное описание возможностей Libgit2 выходит далеко за пределы этой книги. Если вы хотите подробнее ознакомиться с Libgit2, можете начать с документации к API https://libgit2.github.com/libgit2 и набора руководств https://libgit2.github.com/docs. Для привязок к другим языкам, загляните в README и исходники тестов, 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 функциональность Git, написана на чистом Java и широко используется Java сообществом. Проект JGit находится под опекой

Eclipse и расположен по адресу https://projects.eclipse.org/projects/technology.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 https://projects.eclipse.org/projects/technology.jgit/downloads. 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 https://www.eclipse.org/jgit/download. Использовать их в своём проекте можно следующим способом:

```
javac -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App.java
java -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar 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; каталог должен существовать
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
    new File("/tmp/new_repo/.git"));
newlyCreatedRepo.create();

// Открыть существующий репозиторий
Repository existingRepo = new FileRepositoryBuilder()
```

```
    .setGitDir(new File("my_repo/.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 (`.readEnvironment()`), начать поиск с произвольного места в рабочем каталоге (`.setWorkTree(…).findGitDir()`), или просто открыть каталог `.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 синтаксиса rev-parse
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-хеш объекта, который, возможно, сохранён внутри базы данных объектов Git. Следующая строка похожа на предыдущую, но используется синтаксис rev-parse (см. детали в разделе Ссылки на ветки главы 7); 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 «текучих» методов (builder-style / fluent interface). Давайте взглянем на пример — результат выполнения этого кода напоминает `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, а выполнение происходит при вызове `.call()`. В этом примере мы запрашиваем с удалённого репозитория `origin` список тегов, исключая ветки. 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.

Дополнительные материалы

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 https://help.eclipse.org/latest/topic/org.eclipse.egit.doc/help/JGit/User_Guide/User-Guide.html. Это обыкновенный 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.

go-git

Для интеграции Git в сервисы, написанные на языке Golang, существует библиотека на чистом Go. Она не имеет собственных зависимостей, поэтому не подвержена ошибкам ручного управления памятью. Так же эта библиотека прозрачна для стандартных Golang утилит анализа производительности, таких как профайлеры потребления ЦПУ и памяти, детектор гонки и других.

go-git ориентирован на расширяемость, совместимость и поддерживает большинство подключаемых API, которые описаны здесь https://github.com/go-git/go-git/blob/master/COMPATIBILITY.md.

Вот простой пример использования Go API:

```
import "github.com/go-git/go-git/v5"

r, err := git.PlainClone("/tmp/foo", false, &git.CloneOptions{
    URL:      "https://github.com/go-git/go-git",
```

```
    Progress: os.Stdout,
})
```

Как только у вас есть экземпляр `Repository`, вы можете получить доступ к информации и изменять её:

```
// получаем ветку по указателю HEAD
ref, err := r.Head()

// получаем объект коммита по указателю ref
commit, err := r.CommitObject(ref.Hash())

// получаем историю коммита
history, err := commit.History()

// проходим по коммитам и выводим каждый из них
for _, c := range history {
    fmt.Println(c)
}
```

Расширенная функциональность

go-git обладает некоторыми дополнительными функциями, одна из которых — это подключаемое хранилище, что близко по смыслу с бэкендами Libgit2. Реализация по умолчанию — хранилище в памяти, которое очень быстро работает.

```
r, err := git.Clone(memory.NewStorage(), nil, &git.CloneOptions{
    URL: "https://github.com/go-git/go-git",
})
```

Подключаемое хранилище предоставляет много интересных возможностей. Например, https://github.com/go-git/go-git/tree/master/_examples/storage позволяет вам сохранять ссылки, объекты и конфигурацию в базе данных Aerospike.

Другая особенность — гибкая абстракция файловой системы. Используя https://pkg.go.dev/github.com/go-git/go-billy/v5?tab=doc#Filesystem легко сохранять все файлы разными способами: упаковав их все в один архив хранить на диске или держать в памяти.

Ещё одна продвинутая возможность — это тонко настраиваемый HTTP клиент, как например вот этот https://github.com/go-git/go-git/blob/master/_examples/custom_http/main.go.

```
customClient := &http.Client{
    Transport: &http.Transport{ // принимать любой сертификат (полезно при
тестировании)
        TLSClientConfig: &tls.Config{InsecureSkipVerify: true},
    },
    Timeout: 15 * time.Second,  // таймаут в 15 секунд
```

```
        CheckRedirect: func(req *http.Request, via []*http.Request) error {
        return http.ErrUseLastResponse // не следовать перенаправлениям
    },
}

// Перегружаем http(s) протокол по умолчанию для использования собственного клиента
client.InstallProtocol("https", githttp.NewClient(customClient))

// Клонируем репозиторий используя новый клиент, если протокол https://
r, err := git.Clone(memory.NewStorage(), nil, &git.CloneOptions{URL: url})
```

Дополнительные материалы

Полный разбор возможностей go-git выходит за рамки этой книги. Если вы хотите получить больше информации о go-git, воспользуйтесь документацией к API https://pkg.go.dev/github.com/go-git/go-git/v5 и примерами использования https://github.com/go-git/go-git/tree/master/_examples.

Dulwich

Так же существует реализация Git на чистом Python — Dulwich. Проект размещается здесь hhttps://github.com/jelmer/dulwich[^]. Целью проекта является предоставление интерфейса к Git репозиториям (как локальным, так и удалённым) используя чистый Python, а не вызывая Git. В нём используются дополнительные расширения на C, которые существенно увеличивают производительность.

Dulwich следует дизайну Git и разделяет API на два уровня: сантехника и фарфор.

Например, так выглядит использование низкоуровневого API для получения сообщения последнего коммита:

```
from dulwich.repo import Repo
r = Repo('.')
r.head()
# '57fbe010446356833a6ad1600059d80b1e731e15'

c = r[r.head()]
c
# <Commit 015fc1267258458901a94d228e39f0a378370466>

c.message
# 'Add note about encoding.\n'
```

Чтобы вывести лог коммита используя высокоуровневый фарфоровый API, можно использовать:

```
from dulwich import porcelain
```

```
porcelain.log('.', max_entries=1)

#commit: 57fbe010446356833a6ad1600059d80b1e731e15
#Author: Jelmer Vernooij <jelmer@jelmer.uk>
#Date:   Sat Apr 29 2017 23:57:34 +0000
```

Дополнительные материалы

Документацию к API, руководство и множество примеров решения специфичных задач с помощью Dulwich можно найти на странице проекта https://github.com/jelmer/dulwich.

Приложение C: 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.

Можно использовать аббревиатуры для опций. Например, можно использовать команду `git commit --a` вместо `git commit --amend`. Сокращение применимо только для тех опций, первая буква в имени которых не является начальной для других. При написании скриптов рекомендуется использовать полное название.

Настройка и конфигурация

Две довольно распространённые команды, используемые как сразу после установки Git, так и в повседневной практике для настройки и получения помощи — это `config` и `help`.

git config

Сотни вещей в Git 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 разделе Первоначальная настройка Git главы 1 мы использовали эту команду для указания имени, адреса электронной почты и редактора ещё до того, как начать использовать Git.

В разделе Псевдонимы в Git главы 2 мы показали, как можно использовать её для создания сокращённых вариантов команд с длинными списками опций, чтобы не печатать их все каждый раз.

В разделе Перебазирование главы 3 мы использовали `config` чтобы задать поведение `--rebase` по умолчанию для команды `git pull`.

В разделе Хранилище учётных данных главы 7 мы использовали её для задания хранилища ваших HTTP-паролей.

В разделе Разворачивание ключевых слов главы 7 мы показали как настроить фильтры содержимого для данных, перемещаемых между индексом и рабочей копией.

И наконец, этой команде посвящён практически весь раздел Конфигурация Git главы 8.

Команды git config core.editor

Согласно инструкциям, приведённым в разделе Выбор редактора главы 1, большинство редакторов может быть установлено следующим образом:

Таблица 4. Исчерпывающий список команд по настройке `core.editor`

Редактор	Команда
Atom	`git config --global core.editor "atom --wait"`
BBEdit (Mac, with command line tools)	`git config --global core.editor "bbedit -w"`
Emacs	`git config --global core.editor emacs`
Gedit (Linux)	`git config --global core.editor "gedit --wait --new-window"`
Gvim (Windows 64-bit)	`git config --global core.editor "'C:\Program Files\Vim\vim72\gvim.exe' --nofork '%*'"` (смотри примечание ниже)
Kate (Linux)	`git config --global core.editor "kate --block"`
nano	`git config --global core.editor "nano -w"`
Notepad (Windows 64-bit)	`git config core.editor notepad`
Notepad++ (Windows 64-bit)	`git config --global core.editor "'C:\Program Files\Notepad\notepad.exe' -multiInst -notabbar -nosession -noPlugin"` (смотри примечание ниже)
Scratch (Linux)	`git config --global core.editor "scratch-text-editor"`
Sublime Text (macOS)	`git config --global core.editor "/Applications/Sublime\ Text.app/Contents/SharedSupport/bin/subl --new-window --wait"`
Sublime Text (Windows 64-bit)	`git config --global core.editor "'C:\Program Files\Sublime Text 3\sublime_text.exe' -w"` (смотри примечание ниже)
TextEdit (macOS)	`git config --global --add core.editor "open -W -n"`
Textmate	`git config --global core.editor "mate -w"`
Textpad (Windows 64-bit)	`git config --global core.editor "'C:\Program Files\TextPad 5\TextPad.exe' -m"` (смотри примечание ниже)
Vim	`git config --global core.editor "vim --nofork"`
Visual Studio Code	`git config --global core.editor "code --wait"`
VSCodium (Free/Libre Open Source Software Binaries of VSCode)	`git config --global core.editor "codium --wait"`
WordPad	`git config --global core.editor "'C:\Program Files\Windows NT\Accessories\wordpad.exe'"`
Xi	`git config --global core.editor "xi --wait"`

Если у вас установлена 32 битная версия редактора в 64 битной системе, то путь к ней будет содержать `C:\Program Files (x86)\`, а не `C:\Program Files\` как указано в таблице выше.

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 разделе Как получить помощь? главы 1 и показали как её использовать, чтобы найти больше информации о команде `git shell` в разделе Настраиваем сервер главы 4.

Клонирование и создание репозиториев

Существует два способа создать Git репозиторий. Первый — клонировать его из существующего репозитория (например, по сети); второй — создать репозиторий в существующем каталоге.

git init

Чтобы превратить обычный каталог в Git репозиторий и начать версионировать файлы в нём, просто запустите `git init`.

Впервые мы продемонстрировали эту команду в разделе Создание Git-репозитория главы 2 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 разделе Удалённые ветки главы 3.

Мы использовали эту команду для создания чистого репозитория для работы на стороне сервера в разделе Размещение голого репозитория на сервере главы 4.

Ну и наконец мы немного покопались во внутренностях этой команды в разделе Сантехника и Фарфор главы 10.

git clone

На самом деле `git clone` работает как обёртка над некоторыми другими командами. Она создаёт новый каталог, 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, извлекает последний коммит в ваш рабочий каталог, используя `git checkout`.

Команда `git clone` используется в десятке различных мест в этой книге, но мы перечислим наиболее интересные упоминания.

Первоначальное знакомство происходит в разделе Клонирование существующего репозитория главы 2, где мы даём немного объяснений и приводим несколько примеров.

В разделе Установка Git на сервер главы 4 мы рассмотрели как использовать опцию `--bare`, чтобы создать копию Git репозитория без рабочей копии.

В разделе Создание пакетов главы 7 мы использовали `git clone` для распаковки упакованного с помощью `git bundle` репозитория.

Наконец, в разделе Клонирование проекта с подмодулями главы 7 мы научились использовать опцию `--recursive` чтобы упростить клонирование репозитория с подмодулями.

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 рабочего каталога 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 разделе Отслеживание новых файлов главы 2.

О том как использовать `git add` для разрешения конфликтов слияния написано в разделе Основные конфликты слияния главы 3.

В разделе Интерактивное индексирование главы 7 показано как использовать `git add` для добавления в индекс лишь отдельных частей изменённого файла.

В разделе 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 рабочем каталоге 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 разделе Определение состояния файлов главы 2, 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 копией 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 разделе Просмотр индексированных и неиндексированных изменений главы 2, 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 разделе Правила создания коммитов главы 5.

Мы показали вам как эффективно сравнивать ветки используя синтаксис `git diff A…B` в разделе Определение применяемых изменений главы 5.

В разделе Продвинутое слияние главы 7 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 подмодулях показано в разделе Начало работы с подмодулями главы 7.

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 разделе Просмотр индексированных и неиндексированных изменений главы 2.

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 разделе Коммит изменений главы 2. 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 разделе Операции отмены главы 2 мы рассказали об опции `--amend`, используемой для изменения последнего совершённого коммита.

В разделе О ветвлении в двух словах главы 3 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 разделе Подпись коммитов главы 7.

И наконец мы заглянули внутрь команды `git commit` в разделе Объекты коммитов главы 10 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 рабочем каталоге 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 разделе Отмена индексации файла главы 2, где эта команда использовалась для удаления файла из индекса, добавленного туда с помощью `git add`.

В разделе Раскрытие тайн reset, полностью посвящённой этой команде, мы разобрались в деталях её использования.

Мы использовали `git reset --hard` чтобы отменить слияние в разделе Прерывание слияния главы 7, 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 копии. 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 разделе Удаление файлов главы 2, показали как удалять файлы из рабочего каталога и индекса и только из индекса, используя флаг `--cached`.

Ещё один вариант использования `git rm` приведён в разделе Удаление объектов главы 10, 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 эту команду в разделе Перемещение файлов главы 2.

git clean

Команда `git clean` используется для удаления мусора из рабочего каталога. Это могут быть результаты сборки проекта или файлы конфликтов слияний.

Мы рассмотрели множество опций и сценариев использования этой команды в разделе Очистка рабочего каталога главы 7.

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 Ветвление в Git посвящена этой команде, она используется повсеместно в этой главе. Впервые команда `branch` была представлена в разделе Создание новой ветки главы 3, а большинство таких её возможностей как перечисление и удаление веток были разобраны в разделе Управление ветками главы 3.

В разделе Отслеживание веток главы 3 мы показали как использовать сочетание `git branch -u` для отслеживания веток.

Наконец, мы разобрались что происходит за кулисами этой команды в разделе Ссылки в Git главы 10.

git checkout

Команда `git checkout` используется для переключения веток и выгрузки их содержимого в рабочий каталог.

Мы познакомились с этой командой в разделе Переключение веток главы 3 вместе с `git branch`.

В разделе Отслеживание веток главы 3 мы узнали как использовать флаг `--track` для отслеживания веток.

В разделе Использование команды `checkout` в конфликтах главы 7 мы использовали эту команду с опцией `--conflict=diff3` для разрешения конфликтов заново, в случае если предыдущее решение не подходило по некоторым причинам.

Мы рассмотрели детали взаимосвязи этой команды и `git reset` в разделе Раскрытие тайн reset главы 7.

Мы исследовали внутренние механизмы этой команды в разделе HEAD главы 10.

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 разделе Основы ветвления главы 3. 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 раздела Форк публичного проекта.

В разделе Продвинутое слияние главы 7 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 разделе Подпись коммитов главы 7.

Ну и наконец в разделе Слияние поддеревьев главы 7 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 разделе Основные конфликты слияния главы 3 и рассказали как настроить свою программу слияния в разделе Внешние программы слияния и сравнения главы 8.

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 разделе Просмотр истории коммитов главы 2. Там мы видели использование опций `-p` 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 разделе Создание новой ветки главы 3 мы использовали опцию `--decorate` чтобы отобразить указатели веток на истории коммитов, а также `--graph` чтобы просматривать историю в виде дерева.

В разделах Небольшая команда главы 5 и Диапазоны коммитов главы 7 мы познакомили вас с синтаксисом `branchA..branchB`, позволяющем команде `git log` показывать только коммиты, присутствующие в одной ветке, но отсутствующие в другой. Мы довольно подробно рассматриваем этот вопрос в разделе Диапазоны коммитов.

В разделах История при слиянии и Три точки главы 7 мы рассмотрели синтаксис `branchA…branchB` и опцию `--left-right` позволяющие увидеть, что находится в одной или в другой ветке, но не в них обеих сразу. Также в разделе История при слиянии мы рассмотрели опцию `--merge`, которая может быть полезной при разрешении конфликтов, а также `--cc` для просмотра конфликтов слияния в истории проекта.

В разделе RefLog-сокращения главы 7 мы использовали опцию `-g` для вывода `git reflog`, используя `git log`.

В разделе Поиск главы 7 мы рассмотрели использование опций `-S` и `-L` для поиска событий в истории проекта, например, истории развития какой-либо фичи.

В разделе Подпись коммитов главы 7 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 незафиксированных изменений с целью очистки рабочего каталога без необходимости фиксировать незавершённую работу в текущей ветке.

Эта команда практически целиком раскрыта в разделе Припрятывание и очистка главы 7.

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 разделе Работа с тегами главы 2 и использовали на практике в разделе Помечайте свои релизы главы 5.

Мы научились создавать подписанные с помощью GPG метки, используя флаг `-s`, и проверять их, используя флаг `-v`, в разделе Подпись главы 7.

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 разделе Получение изменений из удалённого репозитория — Fetch и Pull главы 2 и продолжили знакомство в разделе Удалённые ветки главы 3.

Мы использовали эту команду в нескольких примерах из раздела Участие в проекте.

Мы использовали её для скачивания запросов на слияние (pull request) из других репозиториев в разделе Ссылки на запрос слияния главы 6, также мы рассмотрели использование `git fetch` для работы с упакованными репозиториями в разделе Создание пакетов главы 7.

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 разделе Получение изменений из удалённого репозитория — Fetch и Pull главы 2 и показали как узнать, какие изменения будут приняты в случае применения в разделе Просмотр удалённого репозитория главы 2.

Мы также увидели как она может оказаться полезной для разрешения сложностей при перемещении веток в разделе Меняя базу, меняй основание главы 3.

Мы показали как можно использовать только URL удалённого репозитория без сохранения его в списке удалённых репозиториев в разделе Извлечение удалённых веток главы 5.

И наконец мы показали как проверять криптографические подписи полученных коммитов, используя опцию `--verify-signatures` в разделе Подпись коммитов главы 7.

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 разделе Отправка изменений в удалённый репозиторий (Push) главы 2. Там мы рассмотрели основы обновления веток в удалённом репозитории. В разделе Отправка изменений главы 3 мы подробнее познакомились с этой командой, а в разделе Отслеживание веток главы 3 мы узнали как настроить отслеживание веток для автоматической передачи на удалённый репозиторий. В разделе Удаление веток на удалённом сервере главы 3 мы использовали флаг `--delete` для удаления веток на сервере, используя `git push`.

На протяжении раздела Участие в проекте мы показали несколько примеров

использования `git push` для совместной работы в нескольких удалённых репозиториях одновременно.

В разделе Публикация изменений подмодуля главы 7 мы использовали опцию `--recurse-submodules` чтобы удостовериться, что все подмодули будут опубликованы перед отправкой проекта на сервер, что может быть реально полезным при работе с репозиториями, содержащими подмодули.

В разделе Прочие хуки на стороне клиента главы 8 мы поговорили о триггере `pre-push`, который может быть выполнен перед отправкой данных, чтобы проверить возможность этой отправки.

Наконец, в разделе Спецификации ссылок для отправки данных на сервер главы 10 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 разделе Работа с удалёнными репозиториями главы 2, 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 тарбола (`tar.gz` файла) всего проекта для передачи по сети в разделе Подготовка релиза главы 5.

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 `submodule` есть несколько под-команд — `add`, `update`, `sync` и др. — для управления такими репозиториями.

Эта команда упомянута и полностью раскрыта в разделе Подмодули главы 7.

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 разделе Аннотированные теги главы 2.

В разделе Выбор ревизии главы 7 мы использовали её для показа коммитов, подпадающих под различные селекторы диапазонов.

Ещё одна интересная вещь, которую мы проделывали с помощью `git show` в разделе Ручное слияние файлов главы 7 — 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 разделе Краткая история (Shortlog) главы 5.

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 Генерация номера сборки и Подготовка релиза 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 разделе Бинарный поиск главы 7, 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 разделе Аннотация файла главы 7.

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 Команда `git grep` главы 7 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` берёт изменения, вносимые одним коммитом, и пытается повторно применить их в виде нового коммита в текущей ветке. Эта возможность полезна в ситуации, когда нужно забрать парочку коммитов из другой ветки, а не сливать ветку целиком со всеми внесёнными в неё изменениями.

Этот процесс описан и показан в разделе Схема с перебазированием и отбором главы 5.

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 разделе Перебазирование главы 3, включая рассмотрение потенциальных проблем переноса опубликованных веток при совместной работе.

Мы использовали эту команду на практике для разбиения истории на два репозитория в разделе Замена главы 7, наряду с использованием флага `--onto`.

В разделе Rerere главы 7 мы рассмотрели случай возникновения конфликта во время переноса коммитов.

Также мы познакомились с интерактивным вариантом `git rebase`, включающемся с помощью опции `-i`, в разделе Изменение сообщений нескольких коммитов главы 7.

git revert

Команда `git revert` — полная противоположность `git cherry-pick`. Она создаёт новый коммит, который вносит изменения, противоположные указанному коммиту, по существу отменяя его.

Мы использовали её в разделе Отмена коммита главы 7 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 почты.

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 разделе Применение патчей, полученных по почте главы 5 и описали случаи, когда вы возможно захотите ею воспользоваться.

git am

Команда `git am` используется для применения патчей из 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 разделе Применение патча командой `am` главы 5, 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 разделе Хуки для рабочего процесса на основе E-mail главы 8.

Также мы использовали `git am` для применения сформированного из GitHub-запроса на слияние patch-файла в разделе Email уведомления главы 6.

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 разделе Публичный проект посредством E-Mail главы 5.

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 разделе Публичный проект посредством E-Mail главы 5.

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 разделе Форк публичного проекта главы 5.

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 разделе Git и Subversion главы 9.

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.

Мы детально рассмотрели эту команду в разделе Импорт произвольного репозитория главы 9.

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 разделе Обслуживание репозитория главы 10.

git fsck

Команда `git fsck` используется для проверки внутренней базы данных на предмет наличия ошибок и несоответствий.

Мы лишь однажды использовали её в разделе Восстановление данных главы 10 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 разделе RefLog-сокращения главы 7, 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 разделе Восстановление данных главы 10.

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 разделе Удаление файла из каждого коммита главы 7 мы объяснили механизм работы этой команды и рассказали про использование опций `--commit-filter`, `--subdirectory-filter` и `--tree-filter`.

Низкоуровневые команды

Также в этой книге встречались некоторые низкоуровневые («сантехнические») команды.

Первая из них — это `ls-remote`, с которой мы столкнулись в разделе Ссылки на запрос слияния главы 6 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 разделе Ссылки на ветки главы 7, используемой для превращения практически произвольно отформатированных строк в SHA-1 указатели.

Так или иначе, большинство низкоуровневых команд собрано в главе Git изнутри, которая на них и сосредоточена. Мы старались избегать этих команд в других местах в этой книге.

Индекс

@

$EDITOR, 359
$VISUAL
    см. $EDITOR, 359
.gitignore, 361
.NET, 507
@{upstream}, 98
@{u}, 98

A

Apache, 124
Apple, 508
autocorrect, 361

B

bash, 497
Bazaar, 433
BitKeeper, 14

C

C, 503
C#, 507
Cocoa, 508
crlf, 366
CVS, 11

D

difftool, 363
Dulwich, 514

E

Eclipse, 495
email, 158
    применение патчей, 160

G

Git как клиент, 391
git-svn, 391
Git-серверы, 110
    GitLab, 127
    GitWeb, 125
    HTTP, 123
    SSH, 117
    протокол Git, 122
GitHub, 176
    API, 225
    Flow, 183
    запросы на слияние, 186
    организации, 217
    учётные записи, 176
GitHub для Mac, 490
GitHub для Windows, 490
gitk, 488
GitLab, 127
GitWeb, 125
Go, 512
go-git, 512
GPG, 360
GUI, 488

I

IRC, 25

J

Java, 508
JGit, 508

L

libgit2, 503
Linux, 14
    установка, 19

M

Mac
    установка, 19
master, 67
Mercurial, 402, 431
mergetool, 363
Mono, 507

O

Objective-C, 508
origin, 90

P

pager, 360
Perforce, 11, 14, 413, 436
    Git Fusion, 413
Posh-Git, 500
PowerShell, 500

Python, 508, 514

R

rerere, 172
Ruby, 504

S

SHA-1, 16
SSH-ключи, 118
    с GitHub, 177
Subversion, 11, 14, 133, 391, 428

V

Visual Studio, 493

W

Windows
    установка, 20

X

Xcode, 19

Z

zsh, 498

А

автодополнение по tab
    bash, 497
    PowerShell, 500
    zsh, 498
архивирование, 375
атрибуты, 369

Б

бинарные файлы, 369

В

Взаимодействие с другими VCS
    Mercurial, 402
    Perforce, 413
    Subversion, 391
ветки, 66
    upstream, 97
    долгоживущие, 86
    отслеживание, 97
    переключение, 69
    простая рабочая схема, 73
    слияние, 78
    создание, 68
    сравнение, 164
    тематические, 87, 160
    удаление внешних, 99
    удалённые, 89, 164
    управление, 82

Г

Графические инструменты, 488

И

Импорт
    из Bazaar, 433
    из Mercurial, 431
    из Perforce, 436
    из Subversion, 428
    из других, 438
игнорирование, 361
игнорирование файлов, 33
интеграция наработок, 166
исключения, 447

К

командные оболочки
    bash, 497
    PowerShell, 500
    zsh, 498
команды git
    add, 31, 31, 32
    am, 161
    apply, 160
    archive, 174
    branch, 68, 82
    checkout, 69
    cherry-pick, 170
    clone, 28
        bare, 116
    commit, 38, 66
    config, 22, 24, 38, 64, 158, 358
    credential, 352
    describe, 173
    diff, 35
        check, 136
    fast-import, 438
    fetch, 55
    fetch-pack, 472
    filter-branch, 437
    format-patch, 157

gitk, 488
gui, 488
help, 25, 122
http-backend, 123
init, 28, 31
bare, 116, 120
instaweb, 126
log, 42
merge, 76
squash, 156
mergetool, 81
p4, 421, 437
pull, 56
push, 56, 62, 95
rebase, 100
receive-pack, 471
remote, 54, 55, 57, 58
request-pull, 153
rerere, 172
send-pack, 471
shortlog, 174
show, 61
show-ref, 394
status, 30, 38
svn, 391
tag, 59, 60, 61
upload-pack, 472
демон, 122
контроль версий, 10
локальный, 10
распределённый, 12
централизованный, 11

Л

Линус Торвальдс, 14

Н

нумерация сборок, 173

О

область индексации
пропускание, 39
отправка, 95

П

перебазирование, 99
опасности, 104
против слияния, 108
переход на Git, 428
подготовка релиза, 174
получение, 98
пример политики, 381
пробел, 366
протоколы
Git, 115
SSH, 114
локальный, 110
простой HTTP, 112
умный HTTP, 112
псевдонимы, 64

Р

рабочие процессы, 132
диктатор и помощники, 134
диспетчер интеграции, 133
множественное слияние, 169
перебазирование и отбор, 170
слияние, 166
централизованная работа, 132
разветвление, 134, 182
разворачивание ключевых слов, 372
распределённый Git, 132
редактор
по умолчанию, 38

С

символы переноса строк, 366
слияние, 78
конфликты, 79
против перебазирования, 108
стратегии, 377
содействие, 136
большой открытый проект, 156
команда с руководителем, 146
небольшая команда, 138
открытый проект, 152
сопровождение проекта, 160

Т

теги, 58, 172
аннотированные, 60
легковесные, 60
подписывание, 172

У

указатели

удалённые, 89
учётные данные, 352

Ф

файлы
перемещение, 41
удаление, 40
фильтрация журнала, 48
форматирование журнала, 45

Х

хуки, 377
post-update, 113

Ц

цвет, 362

Ш

шаблоны коммитов, 359