



БИБЛИОТЕКА ПРОГРАММИСТА

С. Чакон, Б. Штрауб

Git для профессионального программиста

Подробное описание самой
популярной системы
контроля версий.

 ПИТЕР®

Scott Chacon
Ben Straub

Pro Git

Second Edition

Apress®



БИБЛИОТЕКА ПРОГРАММИСТА

С. Чакон, Б. Штрауб

Git для профессионального программиста



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Киев · Екатеринбург · Самара · Минск

2016

ББК 32.973.2-018
УДК 004.3
Д42

Чакон С., Штрауб Б.

Ч-16 Git для профессионального программиста. — СПб.: Питер, 2016. — 496 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-01763-3

Эта книга представляет собой обновленное руководство по использованию Git в современных условиях. С тех пор как проект Git — распределенная система управления версиями — был создан Линусом Торвальдсом, прошло много лет, и система Git превратилась в доминирующую систему контроля версий, как для коммерческих целей, так и для проектов с открытым исходным кодом. Эффективный и хорошо реализованный контроль версий необходим для любого успешного веб-проекта. Постепенно эту систему приняли на вооружение практически все сообщества разработчиков ПО с открытым исходным кодом. Появление огромного числа графических интерфейсов для всех платформ и поддержка IDE позволили внедрить Git в операционные системы семейства Windows. Второе издание книги было обновлено для Git-версии 2.0 и уделяет большое внимание GitHub.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018

УДК 004.3

Права на издание получены по соглашению с Apress.

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

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1484200773 англ.
978-5-496-01763-3

© Apress

© Перевод на русский язык ООО Издательство «Питер», 2016

© Издание на русском языке, оформление ООО Издательство «Питер», 2016

© Серия «Библиотека программиста», 2016

Краткое содержание

Предисловие от Скотта Чакона	16
Предисловие от Бена Страуба	18
1. Начало работы	19
2. Основы Git	34
3. Ветвления в Git	67
4. Git на сервере.....	105
5. Распределенная система Git.....	129
6. GitHub	169
7. Git-инструментарий.....	221
8. Настройка системы Git	328
9. Git и другие системы контроля версий	361
10. Git изнутри	418
Приложение А. Git в других средах.....	458
Приложение Б. Встраивание Git в приложения	471
Приложение В. Git-команды	478
Об авторах.....	495

Содержание

Предисловие от Скотта Чакона	16
Предисловие от Бена Страуба	18
1. Начало работы	19
Управление версиями	19
Локальные системы контроля версий	20
Централизованные системы контроля версий	20
Распределенные системы контроля версий	21
Краткая история Git	23
Основы Git	23
Снимки состояний, а не изменений	24
Локальность операций	25
Целостность Git	26
Git, как правило, только добавляет данные	26
Три состояния	26
Командная строка	28
Установка Git	28
Установка в Linux	29
Установка в Mac	29
Установка в Windows	30
Первая настройка Git	31
Ваш идентификатор	31

Выбор редактора.....	32
Проверка настроек.....	32
Получение справочной информации.....	33
Заключение.....	33

2. Основы Git 34

Создание репозитория в Git.....	34
Инициализация репозитория в существующей папке.....	34
Клонирование существующего репозитория.....	35
Запись изменений в репозиторий.....	36
Проверка состояния файлов.....	37
Слежение за новыми файлами.....	37
Индексация измененных файлов.....	38
Краткий отчет о состоянии.....	39
Игнорирование файлов.....	40
Просмотр индексируемых и неиндексируемых изменений.....	41
Фиксация изменений.....	43
Пропуск области индексирования.....	45
Удаление файлов.....	45
Перемещение файлов.....	47
Просмотр истории версий.....	47
Ограничение вывода команды log.....	52
Отмена изменений.....	54
Отмена индексирования.....	55
Отмена внесенных в файл изменений.....	56
Удаленные репозитории.....	57
Отображение удаленных репозиториях.....	57
Добавление удаленных репозиториях.....	58
Извлечение данных из удаленных репозиториях.....	59
Отправка данных в удаленный репозиторий.....	59
Просмотр удаленных репозиториях.....	60
Удаление и переименование удаленных репозиториях.....	61
Теги.....	61
Вывод списка тегов.....	62
Создание тегов.....	62
Теги с комментариями.....	62
Легковесные теги.....	63
Расстановка тегов постфактум.....	63
Обмен тегами.....	64
Псевдонимы в Git.....	65
Заключение.....	66

3. Ветвления в Git	67
Суть ветвления.....	67
Создание новой ветки	70
Смена веток	71
Основы ветвления и слияния.....	74
Основы ветвления.....	74
Основы слияния	78
Конфликты при слиянии.....	80
Управление ветками.....	82
Приемы работы с ветками	83
Долгоживущие ветки.....	84
Тематические ветки	85
Удаленные ветки	87
Отправка данных.....	91
Слежение за ветками	92
Получение данных с последующим слиянием	94
Ликвидация веток с удаленного сервера	94
Перемещение данных	94
Основы перемещения данных.....	94
Более интересные варианты перемещений.....	97
Риски, связанные с перемещением	99
Перемещение после перемещения	102
Сравнение перемещения и слияния.....	103
Заключение	104
4. Git на сервере.....	105
Протоколы.....	106
Локальный протокол	106
Протоколы семейства HTTP	107
Протокол SSH.....	110
Протокол Git.....	111
Настройка Git на сервере.....	111
Размещение на сервере голого репозитория	112
Простые настройки.....	113
Создание открытого ключа SSH	114
Настройка сервера	115
Git-демон.....	117
Интеллектуальный протокол HTTP.....	119
Интерфейс GitWeb	120
Приложение GitLab	122

Установка.....	122
Администрирование	123
Пользователи	124
Группы	124
Проекты	125
Хуки	126
Базовое применение	126
Совместная работа.....	126
Сторонний хостинг	127
Заключение.....	128

5. Распределенная система Git 129

Распределенные рабочие процессы.....	129
Централизованная работа	130
Диспетчер интеграции	131
Диктатор и помощники.....	132
Заключение.....	133
Содействие проекту.....	133
Рекомендации по созданию коммитов	134
Работа в маленькой группе	136
Маленькая группа с руководителем.....	142
Открытый проект, ветвление.....	146
Открытый проект, электронная почта	150
Заключение.....	153
Сопровождение проекта	153
Работа с тематическими ветками.....	153
Исправления, присланные по почте	154
Просмотр вносимых изменений	158
Интеграция чужих наработок	159
Схема с большим количеством слияний.....	162
Схема с перемещением и отбором.....	163
Программный компонент rerere	165
Идентификация устойчивых версий.....	165
Генерация номера сборки.....	166
Подготовка устойчивой версии.....	167
Команда shortlog	167
Заключение.....	168

6. GitHub 169

Настройка и конфигурирование учетной записи	170
Доступ по протоколу SSH	170
Аватар.....	172

Адреса электронной почты	173
Аутентификация по двум признакам.....	173
Содействие проекту.....	174
Ветвления проектов	175
Схема работы с GitHub	175
Запрос на включение	176
Стадии обработки запроса на включение	180
Более сложные запросы на включение.....	183
Язык разметки Markdown.....	188
GitHub-версия языка Markdown.....	188
Сопровождение проекта	193
Создание нового репозитория	193
Добавление соавторов	194
Управление запросами на включение.....	195
Упоминания и уведомления	201
Специальные файлы	204
Администрирование проекта	206
Управление организацией	207
Основные сведения об организации	207
Группы	208
Журнал регистрации	210
GitHub-сценарии.....	210
Хуки	211
API для GitHub.....	214
От пользователя Octokit	220
Заключение.....	220

7. Git-инструментарий 221

Выбор версии	221
Одна версия.....	221
Сокращения журнала ссылок.....	224
Диапазоны коммитов.....	226
Интерактивное индексирование	229
Индексирование файлов и его отмена.....	229
Индексирование изменений.....	231
Скрытие и очистка.....	232
Скрытие вашей работы.....	233
Более сложные варианты скрытия.....	235
Отмена скрытых изменений.....	236
Создание ветки из скрытого фрагмента.....	236
Очистка рабочей папки	237

Подпись.....	238
Знакомство с GPG	239
Подпись тегов	239
Проверка тегов	240
Подпись коммитов.....	240
Подпись должна быть у всех	242
Поиск	242
Команда git grep.....	242
Поиск в Git-журнале.....	244
Поиск по строкам кода	244
Перезапись истории	245
Редактирование последнего коммита	246
Редактирование нескольких сообщений фиксации	246
Изменение порядка следования коммитов.....	248
Объединение коммитов	249
Разбиение коммита	250
Последнее средство: команда filter-branch.....	251
Команда reset	252
Три дерева	253
Рабочий процесс	254
Роль команды reset.....	259
Команда reset с указанием пути	263
Объединение коммитов	265
Сравнение с командой checkout.....	267
Заключение.....	269
Более сложные варианты слияния.....	269
Конфликты слияния	270
Прерывание слияния.....	272
Игнорирование пробелов	272
Слияние файлов вручную.....	273
Применение команды checkout	276
Протоколирование слияния.....	277
Комбинированный формат	278
Отмена результатов слияния.....	280
Другие типы слияния	284
Команда rerere	287
Отладка с помощью Git.....	292
Примечания к файлам	293
Двоичный поиск	294
Подмодули	296
Начало работы	296
Клонирование проекта с подмодулями	298

Работа над проектом с подмодулями	300
Публикация результатов редактирования подмодуля	305
Слияние результатов редактирования подмодуля	306
Полезные советы.....	309
Пакеты	313
Замена	316
Хранение учетных данных	322
Взгляд изнутри.....	323
Нестандартный вариант хранения учетных данных	325
Заключение.....	327
8. Настройка системы Git	328
Конфигурирование системы Git	328
Основные настройки на стороне клиента	329
Цвета в Git	331
Внешние инструменты для слияния и индикации изменений	332
Форматирование и пробелы	336
Настройка сервера	338
Git-атрибуты	339
Бинарные файлы.....	339
Развертывание ключа	342
Экспорт репозитория.....	345
Стратегии слияния	346
Git-хуки	347
Установка хука	347
Хуки на стороне клиента	347
Хуки для работы с коммитами	347
Хуки для работы с электронной почтой	348
Другие клиентские хуки	349
Хуки на стороне сервера	350
Пример принудительного внедрения политики.....	351
Хук на стороне сервера	351
Формат сообщения фиксации	352
Система контроля доступа пользователей	353
Тестирование	356
Хуки на стороне клиента	357
Заключение.....	360
9. Git и другие системы контроля версий	361
Git в качестве клиента.....	361
Git и Subversion	362
Git и Mercurial.....	372

Git и Perforce	380
Git и TFS.....	394
Переход на Git.....	404
Subversion	404
Mercurial.....	406
Perforce	408
TFS.....	410
Другие варианты импорта	411
Заключение.....	417

10. Git изнутри 418

Канализация и фарфор	419
Объекты в Git	420
Объекты-деревья	421
Объекты-коммиты	424
Хранение объектов.....	427
Ссылки в Git	428
Указатель HEAD.....	429
Теги	430
Удаленные ветки.....	431
Раск-файлы	432
Спецификация ссылок	435
Спецификация ссылок для отправки данных на сервер	437
Ликвидация ссылок	437
Протоколы передачи данных.....	438
Простой протокол	438
Интеллектуальный протокол	440
Обслуживание репозитория и восстановление данных	444
Обслуживание репозитория	444
Восстановление данных	445
Удаление объектов.....	447
Переменные среды	451
Глобальное поведение	451
Расположение репозитория.....	451
Пути доступа.....	452
Фиксация изменений	453
Работа в сети	453
Определение изменений и слияние	454
Отладка.....	454
Разное.....	456
Заключение.....	457

Приложение А. Git в других средах.....	458
Графические интерфейсы.....	458
Утилиты gitk и git-gui.....	459
GitHub-клиенты для Mac и Windows.....	461
Подводя итоги.....	464
Другие GUI.....	464
Git в Visual Studio.....	465
Git в Eclipse.....	466
Git в Bash.....	466
Git в Zsh.....	468
Git в Powershell.....	469
Заключение.....	470
Приложение Б. Встраивание Git в приложения	471
Командная строка.....	471
Libgit2.....	472
Нетривиальная функциональность	474
Другие привязки	476
Приложение В. Git-команды	478
Настройка и конфигурирование.....	478
Копирование и создание проектов.....	479
Фиксация состояния	480
Ветвления и слияния.....	483
Совместная работа и обновление проектов	486
Проверка и сравнение.....	489
Отладка.....	490
Исправления	490
Электронная почта	491
Внешние системы	492
Администрирование.....	493
Служебные команды.....	494
Об авторах.....	495

Эту книгу я посвящаю своим девочкам: жене Джессике, которая поддерживала меня все эти годы, и дочери Джозефине, которая будет поддерживать меня, когда я состарюсь и перестану понимать, что вокруг происходит.

Скотт

Моей жене Бекки, без которой это приключение могло закончиться, даже не начавшись.

Бен

Предисловие от Скотта Чакона

Перед вами второе издание книги. Первая версия увидела свет четыре года назад. Многое изменилось с того времени, хотя наиболее важные вещи остались неизменными. Большинство ключевых команд и концепций до сих пор применимы, ведь Git-разработчики прилагают массу усилий для поддержания обратной совместимости, но появились и значительные нововведения и изменения в окружающую систему Git сообществе. Второе издание книги предусматривает рассказ об этих изменениях и обновлениях, что поможет новым пользователям быстрее войти в курс дела.

На момент написания первой книги система Git была относительно сложной и по сути представляла собой инструмент, ориентированный на опытного разработчика. В некоторых сообществах она начала набирать популярность, но до повсеместного ее использования, которое мы наблюдаем в наши дни, было далеко. Тем не менее, постепенно эту систему приняли на вооружение практически все сообщества разработчиков ПО с открытым исходным кодом. Появление огромного числа графических интерфейсов для всех платформ и поддержка IDE позволили внедрить Git в операционные системы семейства Windows. В первом издании книги об этом не было и речи. Одной из основных целей нового издания является рассмотрение всех этих новшеств.

Резко возросло и количество разработчиков ПО с открытым исходным кодом, которые пользуются системой Git. Почти пять лет назад начало работы над первой версией книги (а процесс написания занял некоторое время) совпало с началом моей работы в малоизвестной компании, занимающейся созданием сайта для хостинга Git. Этот сайт назывался GitHub. К моменту публикации книги сайтом пользовалась

от силы тысяча человек, а работали над ним только мы вчетвером. На момент же написания этого предисловия сайт GitHub объявил о размещении 10-миллионного проекта. Число учетных записей зарегистрированных на нем разработчиков достигло почти 5 миллионов, а количество сотрудников превысило 230. Нравится вам это или нет, но проект GitHub повлиял на сообщество разработчиков ПО с открытым исходным кодом в такой степени, какую я не мог даже вообразить, когда начинал трудиться над первой книгой.

В исходной версии книги я посвятил сайту GitHub небольшой раздел, описав его как место для хостинга системы Git. Мне не очень нравилось, что, по сути, я пишу про общественный ресурс, попутно рассказывая о своей роли в его создании. Мне до сих пор не нравится этот конфликт интересов, но важность проекта GitHub в Git-сообществе давно уже неоспорима. Поэтому теперь я решил не ограничиваться примером хостинга системы Git, а выделить целую главу под детальное описание проекта GitHub и способов эффективного его применения. Если вы собираетесь изучать Git, умение работать с сайтом GitHub поможет вам влиться в огромное сообщество, полезное вне зависимости от того, какой хостинг вы используете для хранения своего кода.

Другим значительным изменением с момента предыдущей публикации стала разработка и растущее использование протокола HTTP для сетевых Git-операций. В большинстве приведенных в книге примеров вместо протокола SSH фигурирует более простой протокол HTTP.

Удивительно наблюдать за тем, как за последние несколько лет система Git развилась из практически неизвестной в доминирующую систему контроля версий, причем как для коммерческих целей, так и для проектов с открытым исходным кодом.

Надеюсь, вы получите удовольствие от чтения новой версии книги.

Предисловие от Бена Страуба

Мой интерес к системе Git вызвало именно первое издание этой книги. Я узнал способ создания программного обеспечения, более естественный, чем все, что я встречал раньше. К этому моменту я был разработчиком уже несколько лет, но это событие помогло мне свернуть на куда более интересную дорогу.

Теперь, годы спустя, я являюсь соавтором одной из основных реализаций Git, я работал и работаю в крупнейшей компании хостинга системы Git, обучая людей по всему миру пользоваться ею. Когда Скотт спросил, не хочу ли я принять участие в подготовке второго издания книги, я, не раздумывая, согласился.

Это большая честь для меня, и в процессе работы я получил огромное удовольствие. Надеюсь, эта книга поможет вам в той же степени, в которой она помогла мне.

1

Начало работы

Эта глава поможет вам начать работу с системой Git. Первым делом мы предоставим общие сведения об инструментах контроля версий, затем поговорим о том, как запустить Git в вашей операционной системе, и закончим описанием процесса настройки. К концу этой главы вы поймете, зачем нужна система Git, почему ею следует пользоваться, и будете готовы приступить к работе.

Управление версиями

Что такое «управление версиями» и почему оно должно нас интересовать? Система управления версиями, или контроля версий, записывает историю изменения файла или набора файлов, чтобы в будущем была возможность вернуться к конкретной версии. В книге процесс управления версиями будет рассматриваться на фрагментах кода, в реальности же эти операции можно проделывать с файлами практически всех типов.

Графическим дизайнерам и веб-дизайнерам, которым нужно хранить все версии изображений или компоновок, можно порекомендовать систему контроля версий (Version Control System, VCS). Она позволяет возвращать в предыдущее состояние как отдельные файлы, так и целый проект, сравнивать сделанные изменения, смотреть, кто и когда последним редактировал файл, являющийся источником проблемы, и многое другое. Наличие VCS в общем случае означает, что при сбое или потере файлов вы можете легко вернуться в рабочее состояние. И это вам практически ничего не будет стоить.

Локальные системы контроля версий

Многие люди в качестве метода контроля версий выбирают простое копирование файлов в другую папку (в лучшем случае такие папки помечаются метками времени). Этот подход крайне популярен в силу своей простоты, но при этом он потрясающе ненадежен. Легко забыть, в какой папке вы работаете, и случайно сделать запись не в тот файл или скопировать вовсе не те файлы, которые вы хотели.

Для урегулирования этого вопроса программисты давно изобрели локальные системы контроля версий, снабженные простой базой данных, хранящей всю информацию об изменениях файлов (рис. 1.1).

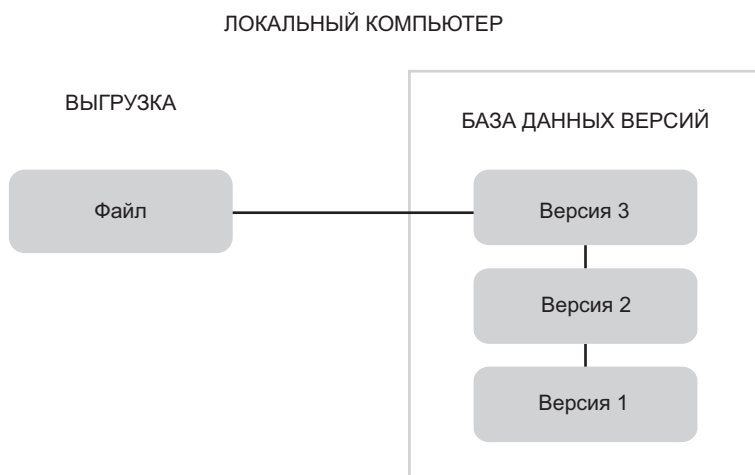


Рис. 1.1. Локальное управление версиями

Одной из наиболее популярных локальных систем контроля версий была система контроля редакций (Revision Control System, RCS), которая до сих пор работает на многих компьютерах. Даже в популярной операционной системе Mac OS X разработчикам доступна команда `rcs`. RCS сохраняет на диске в специальном формате набор всех внесенных изменений (то есть разницу между файлами). В будущем это позволяет воссоздать любой файл в любой момент времени, добавив к нему все изменения.

Централизованные системы контроля версий

Следующая большая проблема — необходимость сотрудничать с разработчиками других систем. Для ее решения были разработаны централизованные системы контроля версий (Centralized Version Control System, CVCS). Они, как и системы CVS, Subversion и Perforce, обладают единым сервером, содержащим все версии

файлов, и набором клиентов, выгружающих файлы с сервера (рис. 1.2). На протяжении многих лет это было стандартом управления версиями.

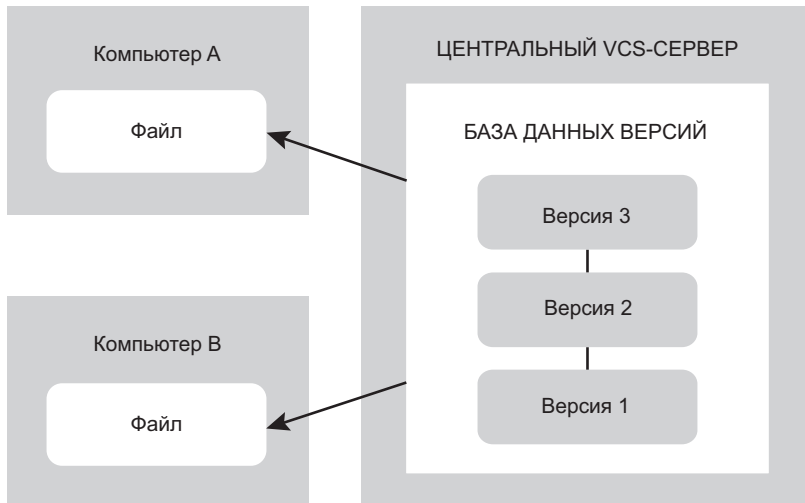


Рис. 1.2. Централизованное управление версиями

Такая схема имеет много преимуществ, особенно перед локальными системами контроля версий. Например, каждый человек, работающий над проектом, до определенной степени знает, чем занимаются его коллеги. Администраторы могут детально контролировать права допуска прочих сотрудников; администрировать CVCS намного проще, чем управлять локальными базами данных на каждом клиенте.

Однако есть у этой схемы и серьезные недостатки. Самым очевидным является единая точка отказа, представленная центральным сервером. Отключение этого сервера на час означает, что в течение часа любые взаимодействия невозможны. То есть вы не сможете сохранять вносимые изменения. При повреждении жесткого диска центральной базы данных и отсутствии нужных резервных копий теряется вся информация — вся история разработки проекта за исключением единичных снимков состояния, которые могут остаться на локальных компьютерах пользователей. Впрочем, та же самая проблема характерна и для локальных систем контроля версий — храня историю разработки проекта в одном месте, вы рискуете потерять все.

Распределенные системы контроля версий

Именно здесь на первый план выходят распределенные системы контроля версий (Distributed Version Control System, DVCS). В DVCS (к примеру, Git, Mercurial, Bazaar или Darcs) клиенты не просто выгружают последние снимки файлов, они создают полную зеркальную копию репозитория. Соответственно в случае выхода

из строя одного из серверов его работоспособность можно восстановить, скопировав один из клиентских репозиторий (рис. 1.3). Каждая такая выгрузка сопровождается полным резервным копированием всех данных.

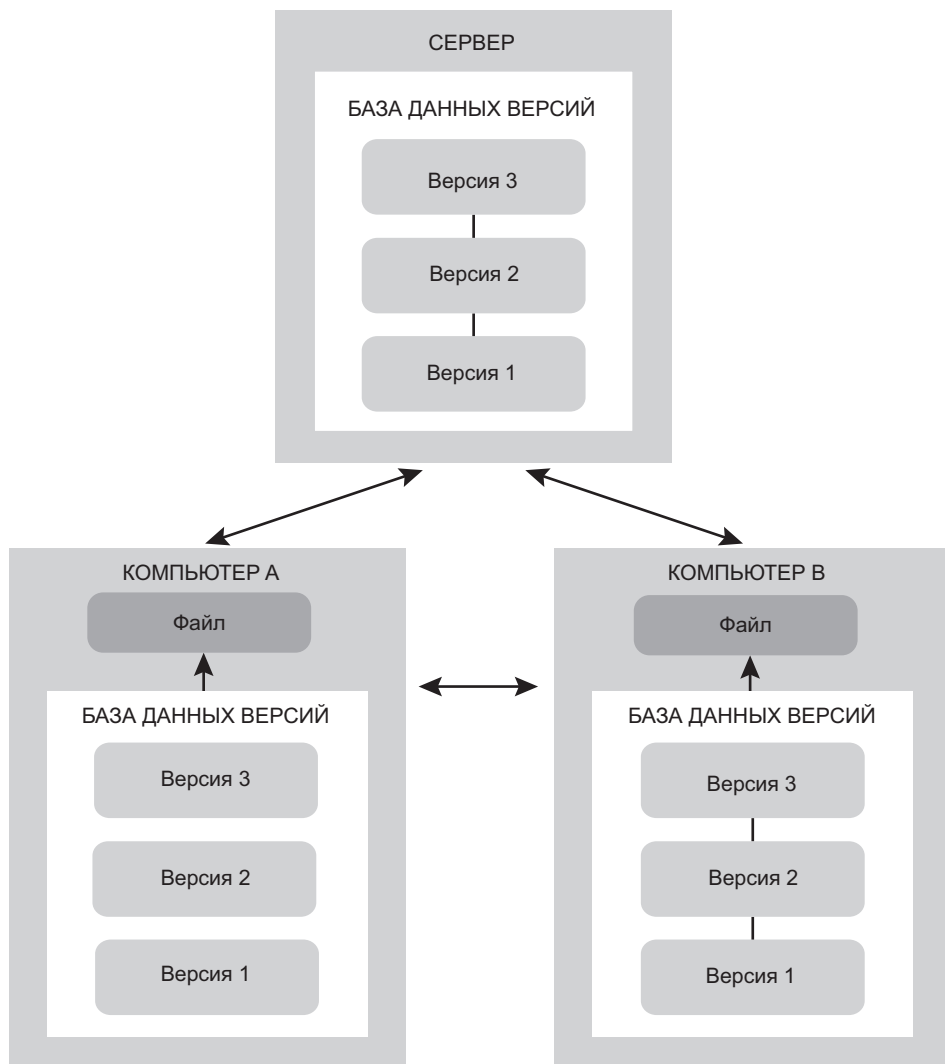


Рис. 1.3. Распределенное управление версиями

Более того, многие из этих систем обладают несколькими удаленными репозиториями, что позволяет разным рабочим группам сотрудничать в рамках одного проекта. Допустима настройка разных типов рабочих процессов, невозможная в централизованных системах, например в иерархических моделях.

Краткая история Git

Подобно множеству других замечательных программных продуктов, система Git начиналась с небольшого созидательного разрушения и пылкой полемики.

Ядро Linux представляет собой крайне масштабный проект ПО с открытым исходным кодом. В истории поддержки ядра Linux изменения программ долгое время передавались в виде исправлений (patches) и архивированных файлов. В 2002 году для проекта Linux стали использовать собственную систему DVCS, которая называлась BitKeeper.

В 2005 году отношения между сообществом, разрабатывавшим ядро Linux, и коммерческой фирмой, создавшей BitKeeper, были разорваны и бесплатное использование этой системы контроля версий стало невозможным, что побудило сообщество разработчиков Linux (и в частности создателя этой операционной системы Линуса Торвальдса) начать работу над собственным инструментом, взяв за основу некоторые идеи BitKeeper. Вот цели, которые ставились для новой системы:

- ☐ быстроедействие;
- ☐ простое проектное решение;
- ☐ мощная поддержка нелинейной разработки (тысячи параллельных ветвей);
- ☐ полностью распределенная система;
- ☐ возможность эффективной (в плане быстрогодействия и объема данных) работы с большими проектами, такими как ядро Linux.

С момента своего появления в 2005 году система Git развивалась и совершенствовалась в попытках добиться простоты использования при сохранении изначальных характеристик. Она работает необыкновенно быстро, крайне эффективна для больших проектов и обладает потрясающей ветвящейся системой нелинейной разработки (см. главу 3).

Основы Git

Так что же в общих чертах представляет собой Git? Крайне важно усвоить информацию этого раздела, так как если вы поймете, что такое система Git и как выглядят основные принципы ее работы, вам будет намного проще эффективно ее использовать. В ходе знакомства с Git постарайтесь забыть обо всем, что вы можете знать о других VCS, таких как Subversion и Perforce; это позволит при работе с данным инструментом избежать путаницы в нюансах. Система Git хранит и воспринимает информацию не так, как они, несмотря на сходный пользовательский интерфейс. Именно понимание этих различий позволит вам избежать ошибок.

Снимки состояний, а не изменений

Главным отличием Git от любой другой системы контроля версий (в том числе Subversion и ей подобных) является восприятие данных. Большинство систем хранит информацию в виде списка изменений, связанных с файлами. Эти системы (CVS, Subversion, Perforce, Bazaar и т. п.) рассматривают хранимые данные как набор файлов и изменений, которые вносились в эти файлы в течение их жизни (рис. 1.4).

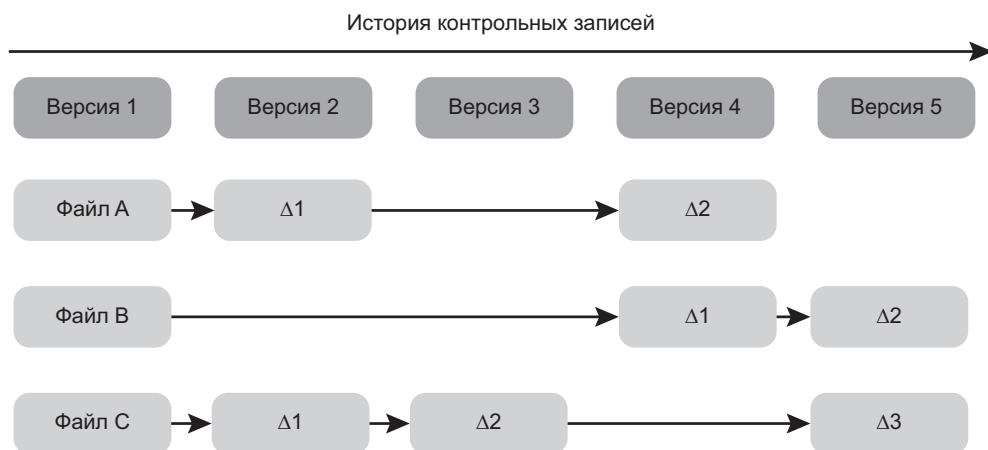


Рис. 1.4. Хранение данных в виде изменений, вносимых в базовую версию каждого файла

Система Git не воспринимает и не хранит файлы подобным образом. В ее восприятии данные представляют собой набор снимков состояния миниатюрной файловой системы. Каждый раз, когда вы создаете новую версию или сохраняете состояние проекта в Git, по сути, делается снимок всех файлов в конкретный момент времени и сохраняется ссылка на этот снимок. Для повышения продуктивности вместо файлов, которые не претерпели изменений, сохраняется всего лишь ссылка на их ранее сохраненные версии (рис. 1.5). Git воспринимает данные скорее как *поток снимков состояния* (stream of snapshots).

Это важное отличие Git почти от всех остальных VCS. И именно из-за него в Git приходится пересматривать практически все аспекты управления версиями, которые остальные системы скопировали у своих предшественниц. В результате Git больше напоминает не простую систему контроля версий, а миниатюрную файловую систему с удивительно мощным инструментарием. Выгоды, которые дает подобное представление данных, мы обсудим в главе 3, когда будем рассматривать ветвления.

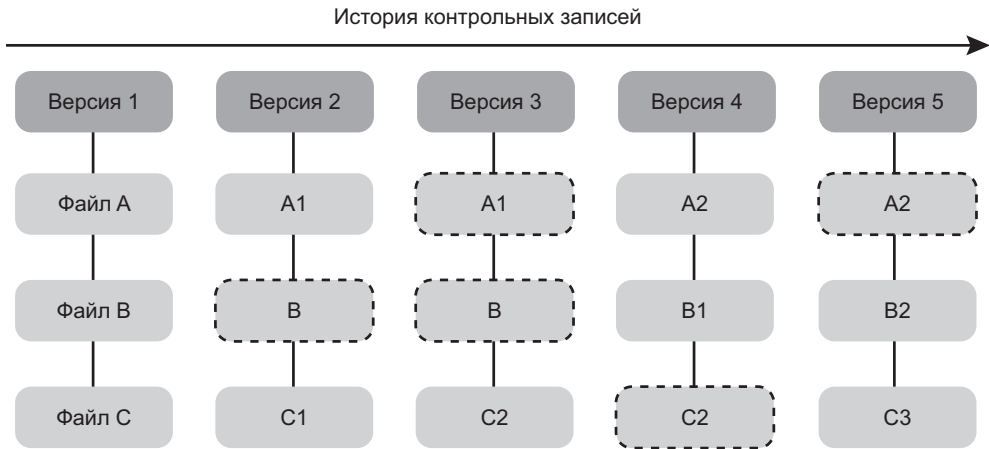


Рис. 1.5. Хранение данных в виде снимков состояния проекта

Локальность операций

Для осуществления практически всех операций системе Git требуются только локальные файлы и ресурсы — в общем случае информация с других компьютеров сети не нужна. Если вы привыкли к CVCS, где для большинства операций характерны задержки из-за передачи данных по сети, этот аспект работы Git наводит на мысль, что боги скорости наделили эту систему нереальной силой. Когда вся история проекта хранится на локальном диске, кажется, что большинство операций выполняется почти мгновенно.

Например, для просмотра истории проекта системе Git нет нужды обращаться к серверу, получать там историю и выводить ее на экран — система просто читает все непосредственно из локальной базы данных. То есть вы видите историю проекта практически сразу же. Если вы хотите посмотреть, чем текущая версия файла отличается от версии месячной давности, Git ищет старый файл и вычисляет внесенные в него правки, вместо того чтобы просить об этой операции удаленный сервер или считывать с этого сервера старую версию файла для локального сравнения.

Это также означает, что практически все операции могут проводиться в автономном режиме и без использования виртуальной частной сети (Virtual Private Network, VPN). Если желание поработать появится в самолете или в поезде, вы можете успешно фиксировать изменения, пока не найдете подключение к сети для их выгрузки. Если дома вы не можете добиться корректной работы VPN-клиента, вам это не мешает. Во многих других системах подобное либо невозможно, либо достигается сложным путем и требует больших усилий. Например, в Perforce при

отсутствии подключения к серверу многие действия недоступны; а в Subversion и в CVS можно редактировать файлы, но нельзя фиксировать внесенные изменения в базе данных (так как она недоступна). Может показаться, что это мелочи, но вы удивитесь, обнаружив, насколько большое значение они имеют.

Целостность Git

В системе Git для всех данных перед сохранением вычисляется контрольная сумма, по которой они впоследствии ищутся. То есть сохранить содержимое файла или папки таким образом, чтобы система Git об этом не узнала, невозможно. Эта функциональность встроена в Git на самом низком уровне и является неотъемлемым принципом ее работы. Невозможно потерять информацию или повредить файл скрытно от Git.

Механизм, которым пользуется Git для вычисления контрольных сумм, называется хешем SHA-1. Это строка из 40 символов, включающая в себя числа в шестнадцатеричной системе (0–9 и a–f) и вычисляемая на основе содержимого файла или структуры папки в Git. Хеш SHA-1 выглядит примерно так:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Вы будете постоянно наталкиваться на эти хеш-значения, так как Git использует их повсеместно. По сути, Git сохраняет данные в базе не по именам файлов, а по хешу их содержимого.

Git, как правило, только добавляет данные

Практически все операции в Git приводят к добавлению данных в базу. Систему сложно заставить выполнить неотменяемое действие или каким-то образом стереть данные. Как и при работе с любой VCS, незафиксированные данные можно потерять или повредить; но как только снимок состояния зафиксирован, потерять его становится крайне сложно, особенно если вы регулярно копируете базу в другой репозиторий.

Это делает работу с Git крайне комфортной, так как можно экспериментировать, не опасаясь серьезно навредить проекту.

Три состояния

А сейчас внимание! Это основное, что требуется запомнить, чтобы дальнейшее освоение Git прошло без проблем. Файлы в Git могут находиться в трех основных состояниях: зафиксированном, модифицированном и индексируемом.

Зафиксированное (committed) состояние означает, что данные надежно сохранены в локальной базе. *Модифицированное* (modified) состояние означает, что изменения уже внесены в файл, но пока не зафиксированы в базе данных. *Индексированное* (staged) состояние означает, что вы поместили текущую версию модифицированного файла как предназначенную для следующей фиксации.

В результате Git-проект разбивается на три основные области: папка Git, рабочая папка и область индексирования (рис. 1.6).

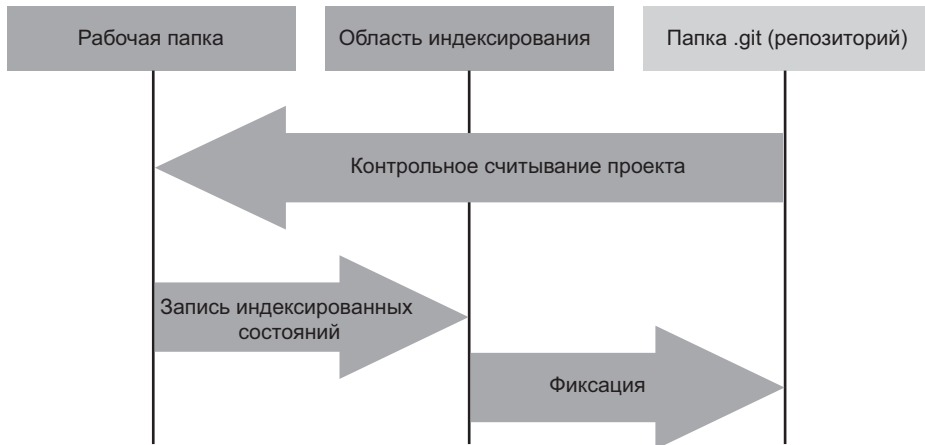


Рис. 1.6. Рабочая папка, область индексирования и папка Git

Папка Git — это место, где Git хранит метаданные и объектную базу данных проекта. Это наиболее важная часть Git, которая копируется при дублировании репозитория (хранилища) с другого компьютера.

Рабочая папка — это место, куда выполняется выгрузка одной из версий проекта. Эти файлы извлекаются из сжатой базы данных в папке Git и помещаются на жесткий диск вашего компьютера, готовые к использованию или редактированию.

Область индексирования — это файл, обычно находящийся в папке Git и хранящий информацию о том, что именно войдет в следующую операцию фиксации. Иногда ее еще называют промежуточной областью.

Базовый рабочий процесс в Git выглядит так:

1. Вы редактируете файлы в рабочей папке.
2. Вы индексируете файлы, добавляя их снимки в область индексирования.
3. Вы выполняете фиксацию, беря файлы из области индексирования и сохраняя снимки в папке Git.

При наличии конкретной версии файла в папке Git файл считается зафиксированным. Если после внесения изменений в файл он был перемещен в область индексирования, он называется проиндексированным. А отредактированный после выгрузки, но не проиндексированный файл называется модифицированным. В главе 2 мы подробно рассмотрим все эти состояния. Заодно вы научитесь пользоваться преимуществами, которые они предоставляют, и пропускать этап индексирования.

Командная строка

Использовать Git можно разными способами. Для этого существуют как инструменты командной строки, так и многочисленные графические пользовательские интерфейсы с различными возможностями. В этой книге рассматривается только работа с Git из командной строки. Главным образом потому, что это единственный способ выполнения всех Git-команд, — в большинстве GUI для простоты реализованы только некоторые варианты функциональности Git. Кроме того, человек, умеющий работать с Git из командной строки, скорее всего, сможет разобраться, что делать с GUI-версией, а вот обратное неверно. И если выбор графического клиента зависит от личных предпочтений, то инструменты командной строки доступны всем без исключения пользователям.

Предполагается, что вы знаете, как открыть терминал в Mac и приглашение на ввод командной строки или оболочку Powershell в Windows. Если вы не понимаете, о чем идет речь, нужно познакомиться с данными инструментами, чтобы следовать приводимым в книге примерам и описаниям.

Установка Git

Перед тем как приступить к работе с Git, эту систему следует установить на компьютер. Если она у вас уже установлена, имеет смысл обновить ее до последней версии. Вы можете воспользоваться менеджером пакетов или другим средством установки или загрузить исходный код и скомпилировать его самостоятельно.

ПРИМЕЧАНИЕ

При написании этой книги использовалась система Git версии 2.0.0. Хотя большинство демонстрируемых команд должно работать даже в самых старых версиях Git, некоторые из них у пользователей устаревших систем могут не работать или работать не так, как описывается. Но так как в Git отлично реализован принцип обратной совместимости, любая версия выше, чем 2.0, должна работать требуемым образом.

Установка в Linux

Если вы хотите установить Git в операционной системе Linux при помощи установщика бинарных пакетов, в общем случае можно воспользоваться инструментом управления пакетами, входящим в имеющийся у вас дистрибутив. Например, в дистрибутиве Fedora применяется утилита `yum`:

```
$ yum install git
```

Если вы пользуетесь дистрибутивом семейства Debian, например Ubuntu, попробуйте утилиту `apt-get`:

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

Инструкции по установке в других операционных системах семейства Unix вы найдете на сайте Git (<http://git-scm.com/download/linux>).

Установка в Mac

Существует несколько способов установки Git на компьютерах Mac. Проще всего, наверное, установить инструменты командной строки Xcode. В операционной системе Mavericks (10.9) и выше достаточно просто попытаться запустить `git` через терминал. Если система еще не установлена, вам будет предложено ее установить.

Те, кому требуется самая последняя версия, могут воспользоваться установщиком бинарных пакетов. Установщик Git для OSX доступен на сайте <http://git-scm.com/download/mac> (рис. 1.7).



Рис. 1.7. Установщик Git OSX

Еще можно установить Git как часть сервиса GitHub для Mac. Инструмент GUI Git дает возможность добавлять инструменты командной строки. Загрузить его можно с сайта <http://mac.github.com>.

Установка в Windows

Установка Git в операционной системе Windows также осуществляется разными способами. Официальный вариант системы доступен на сайте Git. Достаточно зайти на страницу <http://git-scm.com/download/win>, и загрузка начнется автоматически. Обратите внимание, что этот проект называется Git для Windows (или msysGit) и отличается от Git; дополнительная информация находится на сайте <http://msysgit.github.io/>.

Другой простой способ — это установка GitHub для Windows. Пакет установки включает в себя версию как с командной строкой, так и с GUI. Он хорошо работает с оболочкой Powershell и обеспечивает надежное кэширование учетных данных и работоспособные настройки CRLF. Подробно мы рассмотрим эти вещи чуть позже, пока же достаточно сказать, что они вам потребуются. Загрузить эту версию можно с сайта <http://windows.github.com>.

Некоторые пользователи предпочитают установку Git из исходного кода, потому что это позволяет получить самую новую версию. Немного отстают установщики бинарных пакетов, хотя в последние годы развитие Git понемногу стирает эту разницу.

Для установки Git из исходного кода вам потребуются следующие библиотеки, от которых зависит эта система: `curl`, `zlib`, `openssl`, `expat` и `libiconv`. В системах с инструментом `yum` (таких как Fedora) или `apt-get` (как системы семейства Debian) для установки всех зависимостей можно воспользоваться следующими командами:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel
$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

При наличии всех нужных зависимостей архив `tar` с последней версией можно найти в нескольких местах. Например, загрузить с сайта Kernel.org (<https://www.kernel.org/pub/software/scm/git>) или с зеркала сайта GitHub (<https://github.com/git/git/releases>). Как правило, последнюю версию проще найти на странице GitHub, но и на сайте `kernel.org` есть подписи версий, позволяющие проверить, что именно вы загружаете.

Затем остаются процедуры компиляции и установки:

```
$ tar -zxf git-1.9.1.tar.gz
$ cd git-1.9.1
$ make configure
$ ./configure --prefix=/usr
```

```
$ make all doc info
$ sudo make install install-doc install-html install-info
```

После этого обновления можно загружать через саму систему Git:

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

Первая настройка Git

Теперь, когда на вашей машине установлена система Git, нужно настроить ее окружение. На каждом компьютере эта процедура проводится только один раз; при обновлениях все настройки сохраняются. Впрочем, вы можете изменить их в любой момент, снова воспользовавшись указанными командами.

Система Git поставляется с инструментом `git config`, позволяющим получать и устанавливать переменные конфигурации, которые задают все аспекты внешнего вида и работы Git. Эти переменные хранятся в разных местах.

- ❑ Файл `/etc/gitconfig` содержит значения, действующие для всех пользователей системы и всех их репозиториях. Указав параметр `--system` при запуске `git config`, вы добьетесь чтения и записи для этого конкретного файла.
- ❑ Файл `~/.gitconfig` или `~/.config/git/config` связан с конкретным пользователем. Чтение и запись для этого файла инициируются передачей параметра `--global`.
- ❑ Параметры конфигурационного файла в папке Git (то есть `.git/config`) репозитория, с которым вы работаете в данный момент, действуют только на конкретный репозиторий.

Настройки каждого следующего уровня переопределяют настройки предыдущего, то есть конфигурация из `.git/config` перекроет конфигурацию из `/etc/gitconfig`.

В системах семейства Windows Git ищет файл `.gitconfig` в папке `$HOME` (в большинстве случаев она вложена в папку `C:\Users\%USER%`). Кроме того, ищется файл `/etc/gitconfig`, но уже относительно корневого каталога `MSys`, расположение которого вы сами выбираете при установке Git.

Ваш идентификатор

При установке Git первым делом следует указать имя пользователя и адрес электронной почты. Это важно, так как данную информацию Git будет включать в каждую фиксируемую вами версию, и она обязательно включается во все создаваемые вами коммиты (зафиксированные данные):

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

Передача параметра `--global` позволяет сделать эти настройки всего один раз, так как в этом случае Git будет использовать данную информацию для всех ваших действий в системе. Если для конкретного проекта требуется указать другое имя или адрес электронной почты, войдите в папку с проектом и выполните эту команду без параметра `--global`.

Многие GUI-инструменты помогают выполнять эти действия при своем первом запуске.

Выбор редактора

Итак, ваши личные данные указаны, теперь пришло время выбрать текстовый редактор, который будет по умолчанию использоваться, когда вам требуется напечатать сообщение в Git. Без этой настройки Git задействует стандартный редактор операционной системы — обычно это Vim. Если вы предпочитаете другой текстовый редактор, например Emacs, сделайте следующее:

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

ПРИМЕЧАНИЕ

Программы Vim и Emacs представляют собой популярные текстовые редакторы, часто используемые разработчиками операционных систем семейства Unix, таких как Linux и Mac. Если вы никогда ими не пользовались или работаете в операционной системе Windows, вам может потребоваться руководство по настройке вашего любимого текстового редактора в Git.

Проверка настроек

Проверить выбранные настройки позволяет команда `git config --list`, выводящая список всех обнаруженных в текущий момент параметров:

```
$ 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 могут появляться более одного раза, так как Git считывает один и тот же ключ из разных файлов (например, `/etc/gitconfig` и `~/.gitconfig`). В этом случае система использует последнее значение для каждого обнаруженного ею уникального ключа.

Кроме того, можно проверить значение конкретного ключа, воспользовавшись командой `git config <ключ>`:

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

Получение справочной информации

Существует три способа доступа к странице со справочной информацией по любой Git-команде:

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

К примеру, справка по команде `config` открывается так:

```
$ git help config
```

Прелесть этих команд состоит в том, что пользоваться ими можно даже без подключения к сети. Если справочной информации и этой книги недостаточно и вам требуется персональная помощь, воспользуйтесь каналами `#git` или `#github` IRC-сервера Freenode (irc.freenode.net). Обычно там общаются сотни людей, хорошо разбирающихся в Git и готовых прийти на помощь.

Заключение

К этому моменту вы должны в общих чертах представлять, что такое система Git и чем она отличается от централизованных систем управления версиями, которыми вы могли пользоваться ранее. Кроме того, у вас должна быть установлена рабочая версия Git с вашими личными настройками. Теперь все готово для изучения основ Git.

2 Основы Git

Если ваша цель — приступить к работе с Git, прочитав всего одну главу, то эта глава перед вами. Именно здесь вы найдете описания базовых команд, необходимых для решения подавляющего большинства возникающих в процессе использования Git задач. Прочитав эту главу, вы научитесь настраивать и инициализировать репозитории, начинать и прекращать слежение за файлами, индексировать и фиксировать изменения. Заодно мы покажем вам, как заставить Git игнорировать отдельные файлы и их группы, как быстро и просто отменить ошибочные изменения, как посмотреть историю проекта и изменения, вносившиеся в каждую версию, как отправить файл в удаленный репозиторий и извлечь его оттуда.

Создание репозитория в Git

Есть два подхода к созданию Git-проекта. Можно взять существующий проект или папку и импортировать в Git. А можно клонировать уже существующий репозиторий с другого сервера.

Инициализация репозитория в существующей папке

Чтобы начать слежение за существующим проектом, перейдите в папку этого проекта и введите команду

```
$ git init
```


В результате в существующей папке появится еще одна папка с именем `.git` и всеми нужными вам файлами репозитория — это будет основа вашего Git-репозитория. Пока контроль ваших файлов отсутствует. (Подробное описание файлов, входящих в папку `.git`, вы найдете в главе 10.)

Чтобы начать управление версиями существующих файлов (в противовес пустому каталогу), укажите файлы, за которыми должна следить система, и выполните первую фиксацию изменений. Для этого потребуется несколько команд `git add`, добавляющих файлы, за которыми вы хотите следить, а затем команда `git commit`:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'первоначальная версия проекта'
```

Чуть позже мы подробно рассмотрим, что делают эти команды. А пока у вас есть репозиторий Git с добавленными туда файлами и первой зафиксированной версией.

Клонирование существующего репозитория

Получение копии существующего репозитория, например проекта, в котором вы хотите принять участие, выполняется командой `git clone`. Те, кто знаком с другими VCS, например с Subversion, могли заметить, что в данном случае используется команда клонирования (`clone`), а не выгрузки (`checkout`). Это крайне важное отличие. Вы получаете не просто рабочую копию, а полную копию практически всех данных с сервера. Команда `git clone` по умолчанию забирает все версии всех файлов за всю историю проекта. Это означает, что при повреждении серверного диска практически любой клон на любом из клиентов может использоваться для возвращения сервера в состояние, в котором он пребывал до момента клонирования (при этом может быть утрачена часть хуков со стороны сервера, но все данные, относящиеся к версиям, будут на месте). Подробно эта тема рассматривается в главе 4.

Клонирование репозитория осуществляется командой `git clone [url]`. К примеру, вот как клонируется подключаемая Git-библиотека `libgit2`:

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

Команда создает папку с именем `libgit2`, инициализирует в ней папку `.git`, считывает из репозитория все данные и выгружает рабочую копию последней версии. В папке `libgit2` вы найдете все файлы проекта, подготовленные к работе. Репозиторий можно клонировать и в другое место, достаточно указать имя нужной папки в следующем параметре командной строки:

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

Эта команда делает то же самое, что и предыдущая, но все файлы оказываются в папке `mylibgit`.

Для работы Git применяются разные транспортные протоколы. В предыдущем примере использовался протокол `https://`, но можно также встретить вариант

`git://` или `user@server:path/to/repo.git`. В последнем случае применяется протокол SSH. Все доступные варианты серверных конфигураций, обеспечивающих доступ к Git-репозиторию, со всеми их достоинствами и недостатками, рассматриваются в главе 4.

Запись изменений в репозиторий

Итак, у вас есть настоящий Git-репозиторий и некая выгрузка, то есть рабочие копии файлов нашего проекта. Теперь в файлы можно вносить изменения и фиксировать их, как только проект достигнет состояния, которое вы хотели бы сохранить.

Помните, что каждый файл в рабочей папке может пребывать в одном из двух состояний: отслеживаемом и неотслеживаемом. Первый случай — это файлы, входящие в последний снимок системы; они могут быть неизменными, измененными и подготовленными к фиксации. Второй случай — это все остальные файлы рабочей папки, не вошедшие в последний снимок системы и не проиндексированные для последующей фиксации. После первого клонирования репозитория все файлы оказываются отслеживаемыми и неизменными, потому что вы просто выгрузили их и пока не отредактировали.

Отредактированный файл Git рассматривает как измененный, ведь его состояние отличается от последнего зафиксированного. Вы индексируете эти измененные файлы, фиксируете все проиндексированные изменения, после чего цикл повторяется (рис. 2.1).

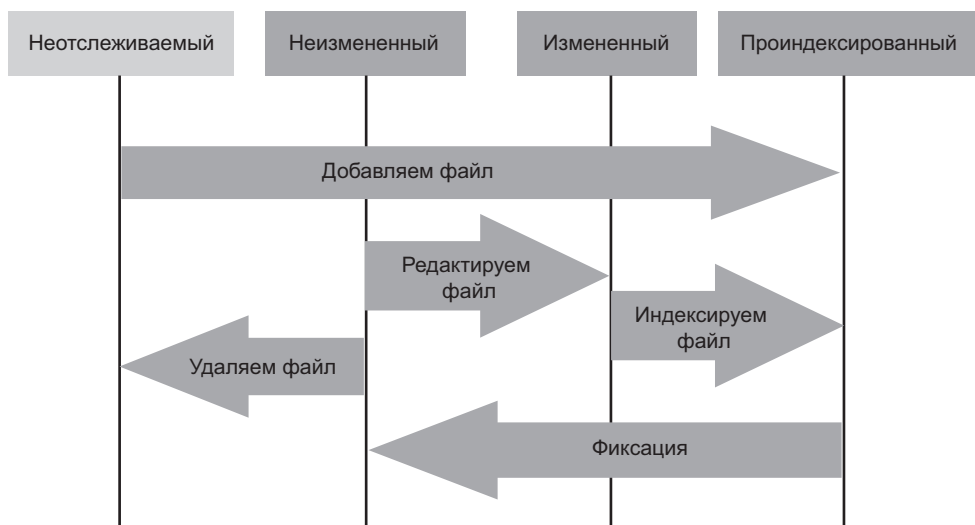


Рис. 2.1. Жизненный цикл состояния файлов

Проверка состояния файлов

Основным инструментом определения состояния файлов является команда `git status`. Непосредственно после клонирования она дает примерно такой результат:

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

Это означает, что в рабочей папке отсутствуют отслеживаемые и измененные файлы. Кроме того, система Git не обнаружила неотслеживаемых файлов, в противном случае они были бы перечислены в выводимых командой данных. Наконец, команда сообщает имя ветки, на которой вы в данный момент находитесь, и информирует о совпадении состояний этой ветки и аналогичной ветки на сервере. Пока мы будем работать только с веткой `master`, предлагаемой по умолчанию, тем более что на данном этапе эта информация не имеет особого значения. Ветки и ссылки подробно обсуждаются в главе 3.

Предположим, вы добавили в проект простой файл `README`. Если до этого момента он не существовал, команда `git status` покажет данный неотслеживаемый файл примерно так:

```
$ echo 'My Project' > README
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

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

Понять, что новый файл `README` является неотслеживаемым, можно по заголовку `Untracked files` в выводимых командой `status` данных. Указанное состояние, по сути, означает, что Git видит файл, отсутствующий в предыдущем снимке состояния (в коммите); без явного указания с вашей стороны Git не будет добавлять этот файл в число коммитов. Подобный подход гарантирует, что в репозитории не сможет случайно появиться сгенерированный бинарный файл или другие файлы, которые вы не собирались туда добавлять. Однако файл `README` туда добавить нужно, поэтому давайте сделаем его отслеживаемым.

Слежение за новыми файлами

Чтобы начать слежение за новым файлом, воспользуйтесь командой `git add`. К примеру, для файла `README` она будет выглядеть так:

```
$ git add README
```

Теперь команда **status** покажет, что этот файл является отслеживаемым и проиндексированным:

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

    new file:   README
```

Все проиндексированные файлы перечисляются под заголовком **Changes to be committed**. Если в этот момент произвести фиксацию, версия файла, существовавшая на момент выполнения команды **git add**, попадет в историю снимков состояния. Надеюсь, вы помните, что за командой **git init** следовала команда **git add** (имена файлов), что заставило систему начать слежение за файлами в папке. Команда **git add** работает с маршрутом доступа к файлу или к папке; если указан путь к папке, команда рекурсивно добавляет все находящиеся в ней файлы.

Индексация измененных файлов

Давайте внесем изменения в файл, находящийся под наблюдением. Если отредактировать ранее отслеживаемый файл **benchmarks.rb** и воспользоваться командой **git status**, результат будет примерно таким:

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

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   benchmarks.rb
```

Файл **benchmarks.rb** появляется под заголовком **Changed but not staged for commit** — это означает, что отслеживаемый файл из рабочей папки был изменен, но пока не проиндексирован. Индексирование выполняется уже знакомой вам командой **git add**. Это многоцелевая команда, позволяющая начать слежение за новыми файлами, произвести индексирование файлов, а также пометить файлы с конфликтом слияния как разрешенные. Возможно, целесообразнее воспринимать эту команду как «добавление содержимого к следующему коммиту», а не как «добавление файла к проекту». Итак, воспользуемся командой **git add** для индексирования файла **benchmarks.rb**, а затем запустим команду **git status**:

```
$ git add benchmarks.rb
$ git status
On branch master
```

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   README
    modified:   benchmarks.rb
```

Теперь оба файла проиндексированы и войдут в следующий коммит. Но предположим, что вы вспомнили про небольшое изменение, которое следует внести в файл `benchmarks.rb` до его фиксации. Вы снова открываете файл, редактируете его, после чего все готово к фиксации. Но сначала еще раз воспользуемся командой `git status`:

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

    new file:   README
    modified:   benchmarks.rb
```

```
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:   benchmarks.rb
```

Удивительное дело. Теперь утверждается, что файл `benchmarks.rb` *одновременно* является и проиндексированным, и непроиндексированным. Как такое может быть? Оказывается, Git индексирует файл в том состоянии, в котором он пребывал на момент выполнения команды `git add`. Если сейчас зафиксировать изменения, в коммит войдет версия файла `benchmarks.rb`, появившаяся после последнего запуска команды `git add`, а не версия, находившаяся в рабочей папке при запуске команды `git commit`. Редактирование файла после выполнения команды `git add` требует повторного запуска этой команды для индексирования самой последней версии файла:

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

    new file:   README
    modified:   benchmarks.rb
```

Краткий отчет о состоянии

Команда `git status` дает исчерпывающий, хотя и многословный результат. Но в Git существует флаг, позволяющий получить сведения в более компактной форме. Запустив команду `git status -s` или `git status --short`, вы получите упрощенный вариант вывода.

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

Рядом с именами новых неотслеживаемых файлов стоит знак `??`, новые файлы, добавленные в область предварительной подготовки, помечены буквой `A`, а модифицированные файлы — буквой `M` и т. п. Пометки выводятся в два столбца: в первом указывается, индексируется ли файл, а во втором — вносились ли в него изменения. Скажем, в приведенном примере файл `README` модифицирован в рабочей папке, но пока не проиндексирован, в то время как файл `lib/simplegit.rb` модифицирован и проиндексирован одновременно. Файл `Rakefile` был модифицирован, проиндексирован и снова модифицирован, то есть в нем присутствуют как индексированные, так и неиндексированные данные.

Игнорирование файлов

Бывает так, что некоторый класс файлов вы не хотите ни автоматически добавлять в репозиторий, ни видеть в списке неотслеживаемых. В эту категорию, как правило, попадают автоматически генерируемые файлы, например журналы регистрации или файлы, генерируемые системой сборки. В подобных случаях создается файл `.gitignore` со списком соответствующих паттернов. Вот пример такого файла:

```
$ cat .gitignore
*.o
*~
```

Первая строка заставляет Git игнорировать любые файлы, заканчивающиеся на «`.o`» или «`.a`», — объектные и архивные файлы, которые могут появляться в процессе сборки кода. Вторая строка предписывает игнорировать файлы, заканчивающиеся на тильду (`~`). Этим значком многие текстовые редакторы, в том числе Emacs, обозначают временные файлы. В этот список можно включить также папки `log`, `tmp` или `pid`, автоматически генерируемую документацию и т. п. Работу всегда имеет смысл начинать с настройки файла `.gitignore`, так как это защищает от случайного добавления в репозиторий файлов, которые там не нужны.

Вот правила для паттернов, которые можно вставлять в файл `.gitignore`.

- ❑ Игнорируются пустые строки и строки, начинающиеся с символа `#`.
- ❑ Работают стандартные глобальные паттерны.
- ❑ Паттерны можно заканчивать слешем (`/`), указывая папку.
- ❑ Можно инвертировать паттерн, начав его с восклицательного знака (`!`).

Глобальные паттерны напоминают упрощенные регулярные выражения, используемые интерпретаторами команд. Звездочка (*) замещает произвольное число символов, запись [abc] соответствует любому символу из указанных в скобках (в данном случае это символ a, b или c), знак вопроса (?) замещает собой один символ, а разделенные дефисом символы в квадратных скобках ([0-9]) совпадают с любым символом из указанного диапазона (в данном случае от 0 до 9). Кроме того, две звездочки применяются для обозначения вложенных папок; запись a/**/z может означать a/z, a/b/z, a/b/c/z и т. д.

Вот еще один пример файла .gitignore:

```
# комментарий - это игнорируется
*.a      # пропускать файлы, заканчивающиеся на .a
!lib.a   # но отслеживать файлы lib.a, несмотря на пропуск файлов на .a
/TODO    # игнорировать только корневой файл TODO, а не файлы вида subdir/TODO
build/    # игнорировать все файлы в папке build/
doc/*.txt # игнорировать doc/notes.txt, но не doc/server/arch.txt
```

СОВЕТ

На сайте GitHub вы найдете исчерпывающий список хороших примеров файла .gitignore для десятков проектов и языков. Он находится на странице <https://github.com/github/gitignore>. Им можно воспользоваться как отправной точкой для ваших собственных проектов.

Просмотр индексированных и неиндексированных изменений

Если команда **git status** дает недостаточно подробный, с вашей точки зрения, результат, например если вы хотите не только получить список отредактированных файлов, но и узнать, что именно изменилось, воспользуйтесь командой **git diff**. Подробно она рассматривается чуть позже, но чаще всего вы будете пользоваться ею для ответа на два вопроса: что вы отредактировали (но пока не проиндексировали) и что из проиндексированного готово к фиксации? Команда **git status** отвечает на эти вопросы в общей форме, перечисляя имена файлов, а вот команда **git diff** показывает добавленные и удаленные строки — то есть все вставки в программу.

Предположим, вы еще раз отредактировали и проиндексировали файл README, а затем подвергли редактированию без последующего индексирования файл benchmarks.rb. Команда **git status** в этом случае даст примерно вот такой результат:

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

```
new file: README
```

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

```
modified: benchmarks.rb
```

Команда **git diff** без дополнительных параметров позволяет посмотреть, что было изменено, но пока не проиндексировано:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
    @commit.parents[0].parents[0].parents[0]
  end

+   run_code(x, 'commits 1') do
+     git.commits.size
+   end
+
+   run_code(x, 'commits 2') do
+     log = git.commits('master', 15)
+     log.size
```

Эта команда сравнивает содержимое в рабочей папке и в области индексирования. Она выводит список изменений, которые вы внесли, но пока не проиндексировали.

Чтобы посмотреть, что из проиндексированного войдет в следующий коммит, воспользуйтесь командой **git diff --staged**. Эта команда сравнивает индексированные изменения с содержимым последней зафиксированной версии:

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1,4 @@
+My Project
+
+ This is my project and it is amazing.
```

Важно помнить, что сама по себе команда **git diff** показывает не все изменения, сделанные с момента последней фиксации состояния, а только те, которые еще не проиндексированы. То есть если вы проиндексировали все сделанные изменения, команда **git diff** ничего вам не вернет.

В то же время если вы проиндексировали файл `benchmarks.rb`, а затем отредактировали его, команда `git diff` покажет как проиндексированные, так и непроиндексированные изменения в этом файле:

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

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:   benchmarks.rb
```

Теперь можно воспользоваться командой `git diff` и узнать, что осталось без индексации:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
  main()
  ##pp Grit::GitRuby.cache_client.stats
  +# test line
```

А команда `git diff --cached` покажет проиндексированные изменения:

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
    @commit.parents[0].parents[0].parents[0]
  end
+  run_code(x, 'commits 1') do
+    git.commits.size
+  end
+
  run_code(x, 'commits 2') do
    log = git.commits('master', 15)
    log.size
```

Фиксация изменений

Теперь, когда область индексирования настроена нужным вам образом, можно зафиксировать внесенные туда изменения. Помните, что все, оставленное неиндексированным, в том числе любые созданные или измененные файлы, для которых

после редактирования не была выполнена команда `git add`, в текущий коммит не войдет. Все эти файлы останутся на вашем диске как измененные. Впрочем, в рассматриваемом сейчас примере мы будем считать, что последний запуск команды `git status` показал все файлы как проиндексированные и все готово к фиксации изменений. Проще всего осуществить фиксацию командой `git commit`:

```
$ git commit
```

Эта команда открывает выбранный вами текстовый редактор. (Редактор устанавливается переменной окружения оболочки `$EDITOR` — обычно это `vim` или `emacs`, хотя вы можете выбрать и другой вариант, воспользовавшись упомянутой в главе 1 командой `git config --global core.editor`.)

В редакторе вы увидите следующий текст (в данном случае для примера взят редактор 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
# Changes to be committed:
#   new file:   README
#   modified:   benchmarks.rb
#
~
~
~
«.git/COMMIT_EDITMSG» 9L, 283C
```

Как видите, по умолчанию сообщение фиксации представляет собой превращенный в комментарий результат работы команды `git status` с пустой строкой сверху. Этот комментарий можно удалить, напечатав вместо него свой вариант, или оставить в качестве напоминания о том, что именно вы фиксируете. Если вам требуется более подробная информация об изменениях, можно добавить к команде `git commit` параметр `-v`. В результате в редакторе появится список изменений, включаемых в новые фиксируемые данные. При выходе из редактора Git создает коммит с указанным сообщением (удаляя комментарии и список изменений).

Сообщение фиксации можно задать и в команде `commit`, поставив перед ним флаг `-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 только что создали первую версию изменений! Надеемся, вы обратили внимание, что версия предоставила вам ряд данных о себе: зафиксированная вами ветка (**master**), контрольная сумма SHA-1 версии (**463dc4f**), количество подвергшихся изменениям файлов и статистика добавленных и удаленных строк.

Напоминаем, что в коммит записывается снимок состояния, указанного в области индексирования. Все, что вы не проиндексировали, до сих пор фигурирует в измененном состоянии; для добавления этой информации в репозиторий нужно еще раз провести фиксацию изменений. При каждой фиксации вы записываете снимок своего проекта, к которому можно будет вернуться в любой момент или использовать его для сравнения с текущим состоянием.

Пропуск области индексирования

Хотя область индексирования помогает получить именно ту версию состояния, которая вам требуется, иногда она чрезмерно усложняет рабочий процесс. Впрочем, в Git есть простой способ обойтись без этой области. Достаточно передать команде **git commit** параметр **-a**, и система Git начнет автоматически индексировать все отслеживаемые файлы перед их фиксацией, позволяя обойтись без команды **git add**:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   benchmarks.rb

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

Как видите, в данном случае перед фиксацией новой версии файла **benchmarks.rb** запускать команду **git add** не нужно.

Удаление файлов

Чтобы система Git перестала работать с файлом, его нужно удалить из числа отслеживаемых (точнее, убрать из области индексирования) и зафиксировать данное изменение. Это делает команда **git rm**, которая заодно удаляет указанный файл из рабочей папки, благодаря чему он исчезает из списка неотслеживаемых.

После простого удаления файла из рабочей папки он появляется в разделе **Changed but not updated** (измененные, но необновленные, то есть непроиндексированные) выводимых командой **git status** данных:

```
$ rm grit.gemspec
$ git status
```

```
On branch 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: grit.gemspec
```

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

Следующий вызов команды `git rm` индексирует удаление файла:

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

```
deleted: grit.gemspec
```

При очередной фиксации изменений файл пропадает и перестает отслеживаться. Если же вы отредактировали файл и уже добавили его в область индексирования, следует воспользоваться параметром `-f`, инициирующим принудительное удаление. Такой подход позволяет избежать случайного удаления данных, которые пока не были записаны в снимок состояния и поэтому не допускают восстановления средствами Git.

Иногда требуется оставить файл в рабочей папке, удалив его из области индексирования. Другими словами, нужно, чтобы файл остался на своем месте, но система Git перестала за ним следить. Эта возможность особенно полезна, когда вы забыли добавить в файл `.gitignore`, к примеру, огромный журнал регистрации или набор скомпилированных файлов `.a` и по ошибке проиндексировали такой файл или журнал. В этом случае на помощь приходит параметр `--cached`:

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

Команде `git rm` можно передавать в качестве аргументов файлы, папки и глобальные паттерны. То есть можно написать:

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

Обратите внимание на обратный слеш (`\`) перед символом `*`. Он необходим потому, что Git добавляет собственные расширения к расширениям имен файлов оболочки. В такой форме команда удаляет все файлы с расширением `.log` из папки `log/`.

Или вы можете написать:

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

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

Перемещение файлов

В отличие от многих других VCS, Git в явном виде не отслеживает перемещения файлов. После переименования файла у системы Git не сохраняется никаких свидетельствующих об этом событии метаданных. Но Git достаточно искусно позволяет отслеживать сделанные перемещения. Подробно этот процесс рассмотрен чуть позднее.

При этом, как ни странно, в Git есть команда `mv`. Для переименования файла можно написать:

```
$ git mv file_from file_to
```

И это отлично сработает. Если после этого посмотреть на статус файла, выяснится, что Git считает его переименованным:

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

    renamed: README.md -> README
```

При этом происшедшее эквивалентно выполнению следующих команд:

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

О том, что произошло переименование, Git узнает косвенным образом, поэтому не имеет значения, пойдете вы указанным путем или воспользуетесь командой `mv`. Во втором случае вы ограничитесь одной командой вместо трех — так что это просто вспомогательная функция. Важнее всего то, что для переименования можно использовать любой удобный вам инструмент, а команду `add/rm` достаточно выполнить позднее, перед фиксацией изменений.

Просмотр истории версий

После сохранения нескольких версий файлов или клонирования уже имеющего содержимое репозитория вы, скорее всего, захотите взглянуть на то, что было сделано ранее. Базовым и самым мощным инструментом в данном случае является команда `git log`.

Мы рассмотрим работу с ней на примере простого проекта `simplegit`. Для его получения воспользуйтесь командой:

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

Примененная к этому проекту команда `git log` даст следующий результат:

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

    changed the version number

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

    removed unnecessary test

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

    first commit
```

По умолчанию при отсутствии параметров команда `git log` выводит в обратном хронологическом порядке список сохраненных в данный репозиторий версий. То есть первыми показываются самые свежие коммиты. Как видите, рядом с каждым коммитом указывается его контрольная сумма SHA-1, имя и электронная почта автора, дата создания и сообщение о фиксации.

У команды `git log` существует великое множество параметров, позволяющих вывести именно ту информацию, которая вам требуется. Рассмотрим несколько самых популярных.

Одним из самых полезных является параметр `-p`, показывающий разницу, внесенную каждым коммитом. А дополнительный параметр `-2` ограничивает выводимый результат последними двумя записями:

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

    changed the verison number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
spec = Gem::Specification.new do |s|
  s.platform = Gem::Platform::RUBY
  s.name = "simplegit"
-  s.version = "0.1.0"
+  s.version = "0.1.1"
  s.author = "Scott Chacon"
  s.email = «schacon@gee-mail.com»
```

```
s.summary = "A simple gem for using Git in Ruby code."
```

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

```
removed unnecessary test
```

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

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

Данный параметр позволяет вывести на экран ту же информацию, но с добавлением внесенных изменений, отображаемых непосредственно после каждого коммита. Это удобно, когда требуется сделать обзор кода или быстро посмотреть, что происходит после серии коммитов, добавленных коллегой. Еще с командой **git log** могут использоваться группы суммирующих параметров. К примеру, для получения краткой статистики по каждой версии применяется параметр **--stat**:

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

```
changed the verison number
```

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

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

```
removed unnecessary test
```

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

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

```
first commit
```

```

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

```

Как видите, параметр `--stat` позволяет вывести под записью о каждой версии список измененных файлов, их количество, а также количество добавленных в них и удаленных из них строк. А в конце выводится сводная информация.

Еще одним крайне полезным параметром является `--pretty`. Он меняет формат вывода информации. Есть несколько предустановленных вариантов. Параметр `oneline` выводит каждый коммит в одну строку, что весьма удобно при просмотре большого числа коммитов. Параметры `short`, `full` и `fuller`, практически не меняя формат вывода, определяют его детализацию:

```

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

```

Наиболее интересен параметр `format`, позволяющий выводить записи журнала в выбранном вами формате. Это особенно полезно при генерации данных для машинного анализа, ведь формат задается в явном виде, и вы можете быть уверены, что при обновлениях Git он не изменится:

```

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

```

Самые распространенные параметры форматирования перечислены в табл. 2.1.

Таблица 2.1. Полезные параметры команды `git log --pretty=format`

Параметр	Описание выводимых данных
%H	Хеш-код коммита
%h	Сокращенный хеш-код коммита
%T	Хеш-код дерева
%t	Сокращенный хеш-код дерева
%P	Хеш-код родительских коммитов
%p	Сокращенный хеш-код родительских коммитов
%an	Имя автора

Параметр	Описание выводимых данных
%ae	Электронная почта автора
%ad	Дата создания оригинала (формат учитывает <code>-date= option</code>)
%ar	Дата создания оригинала, в относительной форме
%cn	Имя создателя версии
%ce	Электронная почта создателя версии
%cd	Дата создания версии
%cr	Дата создания версии в относительном формате
%s	Комментарий

Возможно, у вас появился вопрос, в чем разница между *автором* (author) и *создателем версии* (committer). Автор — это человек, создавший файл, в то время как создателем версии называется тот, кто внес в авторскую версию правки и сохранил их. Так что если вы послали в проект некое исправление, а один из основных разработчиков применил его, вы оба останетесь в истории: вы — как автор, а разработчик — как создатель версии. Более подробно это отличие рассмотрено в главе 5.

Параметры **oneline** и **format** особенно полезны в сочетании с другим параметром команды **log**, который называется **--graph**. Он добавляет небольшой ASCII-граф с историей ветвлений и слияний:

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

В следующей главе, когда мы начнем рассматривать ветвления и слияния, вывод этой команды будет выглядеть куда интереснее.

Мы рассмотрели только самые простые параметры форматирования вывода команды **git log**, а их существует великое множество. В табл. 2.2 представлены как уже знакомые вам варианты, так и другие полезные параметры с описанием их влияния на вывод команды **log**.

Таблица 2.2. Распространенные параметры команды `git log`

Параметр	Описание
<code>-p</code>	Показывает изменения, внесенные в каждую версию
<code>--stat</code>	Показывает статистику измененных файлов в каждом коммите
<code>--shortstat</code>	Показывает только строку с изменениями/вставками/удалениями от команды <code>--stat</code>
<code>--name-only</code>	Показывает список измененных файлов после информации о коммите
<code>--name-status</code>	Показывает список измененных файлов с информацией о добавлении/изменении/удалении
<code>--abbrev-commit</code>	Показывает только первые несколько символов контрольной суммы SHA-1 вместо всех 40
<code>--relative-date</code>	Показывает дату не в полном, а в относительном формате (например, «2 недели назад»)
<code>--graph</code>	Показывает ASCII-граф истории ветвлений и слияний вместе с выводом команды <code>log</code>
<code>--pretty</code>	Показывает коммиты в альтернативном формате. Возможны параметры <code>oneline</code> , <code>short</code> , <code>full</code> , <code>fuller</code> и <code>format</code> (с указанием вашей версии формата)

Ограничение вывода команды `log`

Команда `git log` обладает не только параметрами форматирования вывода, но и ограничивающими параметрами, дающими возможность выводить на экран только часть коммитов. Один из этих параметров вы уже встречали — это был параметр `-2`, благодаря которому в вывод команды попали только два последних коммита. По сути, вы можете указать параметр `-n`, где `n` — это количество выводимых коммитов. На практике этот параметр применяется не очень часто, так как по умолчанию Git пропускает результат работы команды через сценарий постраничного вывода и выводит на экран только первую страницу.

Более полезны параметры, устанавливающие ограничения по времени, такие как `--since` и `--until`. Вот пример команды, которая выводит список зафиксированных за последние две недели версий:

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

Эта команда работает с разными форматами: можно указать как точную дату, например «2008-01-15», так и относительную, то есть «2 года, 1 день и 3 минуты назад».

Еще существует возможность фильтрации списка коммитов по какому-либо критерию. Параметр `--author` позволяет увидеть версии, созданные определенным автором, а параметр `--grep` дает возможность искать в сообщениях фиксации ключевые слова. Для одновременного поиска по этим критериям следует добавлять параметр `--all-match`, в противном случае в результаты попадет все то, что удовлетворяет либо первому, либо второму критерию.

Еще одним крайне полезным фильтром является параметр `-S`, принимающий в качестве аргумента строку. С этим параметром команда выдает только те коммиты, изменения в которых включают добавление или удаление этой строки. К примеру, для поиска последней версии, в которую добавлялась или из которой удалялась ссылка на некую функцию, можно написать:

```
$ git log --Sfunction_name
```

Последним крайне полезным параметром команды `git log`, исполняющим роль фильтра, является путь. Указание имени определенной папки или файла ограничивает вывод команды только версиями, в которых в данные файлы вносились изменения. Этот параметр всегда фигурирует последним и обычно предваряется двойным дефисом (`--`), отделяющим путь от остальных параметров.

В табл. 2.3 перечислены эти и некоторые другие часто употребляемые параметры.

Таблица 2.3. Параметры, ограничивающие вывод команды `git log`

Параметр	Описание
<code>-(n)</code>	Показывает только последние <code>n</code> коммитов
<code>--since, --after</code>	Показывает только коммиты, внесенные после указанной даты
<code>--until, --before</code>	Показывает только коммиты, внесенные до указанной даты
<code>--author</code>	Показывает только коммиты определенного автора
<code>--committer</code>	Показывает только коммиты, внесенные определенным участником
<code>--grep</code>	Показывает только коммиты с сообщением фиксации, содержащим указанную строку
<code>-S</code>	Показывает только коммиты, в которых добавленный или удаленный код совпадает с указанной строкой

Вот каким образом можно посмотреть в истории исходного Git-кода коммиты пользователя Junio Hamano, вносящие изменения в тестовые файлы и не подвергавшиеся слиянию в течение октября 2008 года:

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

Из практически 40 000 коммитов в истории исходного Git-кода эта команда выводит всего 6, подпадающих под указанный критерий.

Отмена изменений

Необходимость отмены внесенных изменений может возникнуть на любой стадии проекта. В этом разделе мы рассмотрим несколько базовых инструментов, позволяющих это сделать. Однако здесь следует быть крайне осторожным, ведь после отмены далеко не всегда существует возможность вернуться в предшествующее состояние. Это одна из немногих ситуаций при работе с Git, когда неверные действия могут привести к потере результатов вашего труда.

Необходимость отмены чаще всего возникает при слишком ранней фиксации изменений, когда вы забыли добавить в коммит какие-то файлы или ошиблись с сообщением фиксации. Для повторного сохранения версии в такой ситуации можно воспользоваться параметром `--amend`:

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

Эта команда берет область индексирования и включает в коммит всю обнаруженную там информацию. Если после последней фиксации версии вы не внесли никаких изменений (к примеру, эта команда была запущена непосредственно после предыдущего коммита), снимок состояния останется таким же, как и был, изменится только сообщение фиксации.

Откроется тот же самый редактор сообщения фиксации, но с уже введенной туда версией сообщения к предыдущему коммиту. Вы можете обычным образом отредактировать это сообщение, но предшествующий коммит в результате будет переписан.

Скажем, если после фиксации очередной версии вы обнаружили, что забыли проиндексировать изменения в одном из файлов, который планировалось включить в новый коммит, можно сделать так:

```
$ git commit -m 'изначальный коммит'
$ git add forgotten_file
$ git commit --amend
```

В итоге у вас останется единственный коммит — второй коммит заменит результат первого.

Отмена индексирования

Следующие два раздела демонстрируют приемы управления изменениями в области индексирования и рабочей папке. Что замечательно, команда, задающая состояние этих двух областей, напоминает, каким образом можно отменить внесенные изменения. Предположим, вы отредактировали два файла и хотели бы зафиксировать их в двух разных коммитах, но случайно набрали команду `git add *`, что привело к их одновременному индексированию. Как отменить индексирование одного из них? Это покажет команда `git status`:

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

    renamed:   README.md -> README
    modified:  benchmarks.rb
```

Сразу под текстом `Changes to be committed` написано, что для отмены индексирования следует воспользоваться командой `git reset HEAD <имя файла>`. Последуем этому совету и отменим индексирование файла `benchmarks.rb`:

```
$ git reset HEAD benchmarks.rb
Unstaged changes after reset:
M benchmarks.rb
$ 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: benchmarks.rb
```

Эта команда выглядит несколько странно, но все же работает. Теперь файл `benchmarks.rb` фигурирует как измененный, но непроиндексированный.

Вот и все, что вам на данный момент требуется знать о команде `git reset`. Более подробно она рассматривается далее, где вы познакомитесь с дополнительными вариантами ее применения, позволяющими делать очень интересные вещи.

ПРИМЕЧАНИЕ

С параметром `--hard` команда `git reset` **может привести** к опасным последствиям, так как в этом случае затрагиваются файлы в рабочей папке. Без этого параметра команда `git reset` совершенно безопасна, так как затрагивает только область индексирования.

Отмена внесенных в файл изменений

Что делать, если вы внезапно поняли, что не хотите сохранять внесенные в файл `benchmarks.rb` изменения? Как быстро отменить их и вернуть файл в состояние, в котором он находился до последней фиксации (или до первоначального клонирования или другого действия, после которого файл попал в рабочий каталог)? Нам на помощь снова приходит команда `git status`. В последнем выводе неиндексированная область выглядела 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:   benchmarks.rb
```

Здесь достаточно четко сказано, как отменить сделанные изменения. Последуем данному совету:

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

    renamed:   README.md -> README
```

Как видите, изменения были отменены.

ВНИМАНИЕ

Важно понимать степень опасности команды `git checkout -- [файл]`. Все внесенные в файл изменения пропадают — вы просто копируете в этот файл содержимое другого файла. Никогда не пользуйтесь этой командой, если не уверены, что данный файл вам больше не понадобится.

Ветвление и скрытие — операции, которые позволят сохранить внесенные в файл изменения, но временно убрать его с глаз, — рассмотрены в главе 3. Как правило, они являются оптимальным решением данной задачи.

Помните, что в Git все, что *входит в коммиты*, почти всегда поддается восстановлению, даже коммиты из удаленных веток или переписанные с помощью параметра

--amend (см. раздел, посвященный восстановлению данных). Но все, что не было зафиксировано, при потере, скорее всего, пропадет навсегда.

Удаленные репозитории

Для совместной работы над проектами Git требуются навыки управления удаленными репозиториями. Удаленные репозитории представляют собой версии проекта, хранимые в Интернете или где-то в сети. Их может быть несколько, и каждый в общем случае доступен вам только для чтения или же для чтения и записи. Вы должны уметь отправлять данные в удаленный репозиторий и извлекать их оттуда каждый раз, когда требуется обменяться результатами работы. Кроме того, вы должны знать, как добавить удаленный репозиторий, как удалить репозиторий, который больше не используется, как управлять различными удаленными ветками и делать их отслеживаемыми и неотслеживаемыми, и многое другое. В данном разделе мы рассмотрим некоторые из этих приемов.

Отображение удаленных репозитория

Просмотр уже настроенных удаленных серверов осуществляется командой **git remote**. Она дает список коротких имен для всех указанных вами областей удаленной работы. Если репозиторий был клонирован, вы должны увидеть по крайней мере источник, то есть имя, которое Git по умолчанию присваивает копируемому серверу:

```
$ 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
```

Параметр **-v** позволяет увидеть URL-адреса, которые Git хранит для сокращенного имени, используемого при чтении из данного удаленного репозитория и при записи в него:

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

Если репозитория несколько, они выводятся списком. Скажем, репозиторий с несколькими удаленными копиями для совместной работы с коллегами может выглядеть следующим образом.

```
$ 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 легко можем скачать себе изменения, внесенные любым из этих пользователей. При этом у нас может быть допуск на запись в один или несколько репозиториях из списка, хотя представленный код данную деталь не показывает.

Обратите внимание, что эти удаленные репозитории пользуются разными протоколами; впрочем, детально эта тема рассматривается в главе 4.

Добавление удаленных репозиторий

Ранее уже не раз упоминался и демонстрировался процесс добавления удаленных репозиторий, но теперь мы рассмотрим его в явном виде. Чтобы добавить такой репозиторий под коротким именем, которое упростит дальнейшие обращения к нему, используйте команду `git remote add [сокращенное имя] [url]`. Вот как добавить репозиторий коллеги Пола:

```
$ 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 полного URL-адреса в командную строку можно вводить имя `pb`. К примеру, для скачивания всей информации, которая есть у коллеги, но отсутствует у вас, используйте 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** этого коллеги теперь доступна вам локально как **pb/master**. Вы можете выполнить ее слияние с одной из ваших веток или перейти в нее, если требуется просто проверить ее содержимое. (Ветви и работа с ними подробно рассматриваются в главе 3.)

Извлечение данных из удаленных репозитория

Как вы уже видели, извлечение данных из удаленных проектов выполняется такой командой:

```
$ git fetch [имя удаленного репозитория]
```

Эта команда связывается с удаленным проектом и извлекает оттуда все пока отсутствующие у вас данные. После этого у вас должны появиться ссылки на все ветки удаленного проекта, которые можно подвергнуть слиянию или просмотреть.

При клонировании данная команда автоматически добавляет удаленный репозиторий под именем «**origin**». Соответственно команда **git fetch origin** извлекает все, что появилось на этом сервере после его клонирования (или после момента последнего извлечения информации). Важно понимать, что команда **git fetch** помещает все данные в ваш локальный репозиторий, — она не выполняет автоматическое слияние с ветками, с которыми вы работаете в данный момент, и вообще никак не затрагивает эти ветки. Слияние вы выполните вручную, как только в этом возникнет необходимость.

Если же у вас есть ветка, настроенная на слежение за какой-то удаленной веткой (подробно эта операция рассматривается в главе 3), команда **git pull** будет автоматически извлекать информацию из удаленной ветки и выполнять слияние с текущей веткой. В некоторых случаях такой порядок вещей оказывается проще и удобнее; кроме того, по умолчанию команда **git clone** автоматически настраивает вашу локальную ветку **master** на слежение за удаленной веткой **master** (она может иметь и другое имя) на сервере, с которого вы выполняли клонирование. В общем случае команда **git pull** извлекает данные с сервера, который вы клонировали, и автоматически пытается слить их с вашим текущим рабочим кодом.

Отправка данных в удаленный репозиторий

Чтобы поделиться результатами своего труда, их нужно отправить в репозиторий. Это делается простой командой **git push [имя удаленного сервера] [ветка]**. Для отправки ветки **master** на сервер **origin** (еще раз напоминаем, что в процессе клонирования эти имена присваиваются автоматически) следует написать:

```
$ git push origin master
```

Команда работает только при условии, что клонирование осуществлялось с сервера, где у вас есть доступ на запись, и за это время никто не отправлял туда свои данные. Если вы выполнили клонирование одновременно с другим пользователем и он уже отправил результаты своей работы на сервер, ваша попытка отправки данных окончится неудачей. Вам сначала нужно скачать все добавленное этим пользователем и встроить это в свои данные, и только после этого появится возможность воспользоваться командой `push`. Более подробно процесс отправки данных на удаленные серверы рассматривается в главе 3.

Просмотр удаленных репозиторий

Для получения дополнительной информации о конкретном удаленном репозитории применяется команда `git remote show [имя удаленного сервера]`. Вставив в команду имя `origin`, вы получите примерно такой 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)
```

Выводятся URL-адрес удаленного репозитория и информация об отслеживаемых ветках. Команда услужливо сообщает, что если, находясь в ветке `master`, вы запустите команду `git pull`, ветка `master` с удаленного сервера будет автоматически слита с вашей сразу же после скачивания всех необходимых данных. Кроме того, она выводит на экран список всех скачанных ею ссылок.

Это был простой пример, не имеющий практического смысла. Но при более интенсивной работе с Git может возникнуть ситуация, когда команда `git remote show` выведет на экран большее количество информации:

```
$ 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)

```

Здесь сообщается, какая именно ветка будет автоматически отправлена на сервер при выполнении команды **git push**. Также в выводе команды отображены пока отсутствующие у вас ветки с удаленного сервера, ветки, которые есть у вас, но удалены с сервера, и наборы веток, которые автоматически подвергаются слиянию при выполнении команды **git pull**.

Удаление и переименование удаленных репозитория

Переименование ссылок осуществляется командой **git remote rename**, меняющей сокращенные имена удаленных репозитория. К примеру, вот как выглядит присвоение репозиторию **pb** имени **paul**:

```

$ git remote rename pb paul
$ git remote
origin
paul

```

Имеет смысл упомянуть, что эта команда умеет менять и имена удаленных веток. Теперь к ветке **pb/master** нужно обращаться по имени **paul/master**.

Если по какой-то причине вы хотите удалить ссылку на удаленный репозиторий (например, вы поменяли сервер, больше не используете конкретное зеркало или участник проекта перестал вносить в него вклад), используйте команду **git remote rm**:

```

$ git remote rm paul
$ git remote
origin

```

Теги

Подобно большинству VCS, Git позволяет пометить определенные моменты истории. Как правило, эту функциональность задействуют для пометки выходящих версий (v1.0 и т. п.). В этом разделе показано, как посмотреть все доступные теги, как создавать новые, какие типы тегов существуют.

Вывод списка тегов

Для просмотра списка доступных тегов применяется очевидная команда `git tag`:

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

Теги выводятся в алфавитном порядке; порядок их вывода никакого значения не имеет.

Кроме того, искать теги можно по шаблону. Например, пусть репозиторий Git содержит более 500 тегов. Предположим, вас интересуют только выпуски 1.8.5:

```
$ 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
```

Создание тегов

В Git используются два основных типа тегов: легковесные и снабженные комментарием.

Легковесный тег (lightweight tag) во многом напоминает не меняющуюся ветку — это просто указатель на конкретный коммит. А вот теги с комментариями (annotated tags) хранятся в базе данных Git как полноценные объекты. Они обладают контрольной суммой; содержат имя человека, поставившего тег, адрес его электронной почты и дату создания; снабжены комментарием; могут быть подписаны и проверены в программе GNU Privacy Guard (GPG). Обычно рекомендуется создавать именно теги с комментариями, чтобы у вас была вся эта информация, но если нужно сделать временный тег или по какой-то причине вы не хотите хранить все эти сведения, можно обойтись и легковесными тегами.

Теги с комментариями

Создать в Git тег, снабженный комментарием, очень легко. Проще всего добавить параметр `-a` к команде `tag`:

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

Параметр `-m` задает сообщение, которое будет храниться вместе с тегом. Если вы не укажете это сообщение, Git запустит редактор, чтобы вы смогли его ввести.

Для просмотра данных тега вместе с помеченным им коммитом служит команда `git show`:

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date: Sat May 3 20:19:12 2014 -0700

my version 1.4

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

    changed the verison number
```

Перед информацией о коммите выводятся данные человека, поставившего тег, дата пометки коммита и текст комментария.

Легковесные теги

Другим средством пометки коммитов являются легковесные теги. По сути, это сохраненная в файле контрольная сумма коммита — больше никакой информации они не содержат. Для создания легковесного тега достаточно опустить параметры `-a`, `-s` и `-m`:

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

На этот раз примененная к тегу команда `git show` не выводит никакой дополнительной информации. На экране появляется только помеченный коммит:

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

    changed the verison number
```

Расстановка тегов постфактум

Тегами можно помечать и сделанные ранее коммиты. Предположим, что история зафиксированных состояний выглядит так:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fc9c added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Также предположим, что вы забыли снабдить тегом версию проекта v1.2, соответствующую коммиту, упомянутому как «updated rakefile». Это можно сделать и позже, указав в конце команды контрольную сумму коммита (или ее часть):

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

Теперь проверим наличие тега:

```
$ 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

    updated rakefile

...
```

Обмен тегами

По умолчанию команда `git push` не отправляет теги на удаленные серверы. Созданные вами теги нужно отправлять на сервер общего доступа отдельно. Этот процесс напоминает совместное использование удаленных веток — вы делаете это командой `git push origin [имя тега]`.

```
$ 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
```

Когда требуется переслать сразу много тегов, добавляйте к команде `git push` параметр `--tags`. В результате на удаленный сервер будут отправлены все теги, которых там пока 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 пользователь, который клонирует ваш репозиторий или скачает из него данные, получит и все ваши теги.

Псевдонимы в Git

Перед тем как закончить эту главу, посвященную основам Git, дадим один совет, который сделает вашу работу с системой Git проще, удобнее и привычнее. Речь идет о псевдонимах (aliases).

При неполном вводе команды Git не пытается догадаться, что за команду вы имели в виду. Но если мысль о вводе длинных команд вас не привлекает, команда `git config` позволяет легко создать псевдоним для любой из них. Вот пара примеров ее применения:

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

Теперь вместо команды `git commit` достаточно будет ввести `git ci`. По мере освоения Git вам, скорее всего, придется часто пользоваться и другими командами; не стесняйтесь создавать для них псевдонимы.

Кроме того, эта техника позволяет создавать команды, которых, с вашей точки зрения, не хватает. К примеру, чтобы устранить функциональную проблему, с которой мы столкнулись при отмене индексирования файла, можно добавить псевдоним `unstage`:

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

После этого следующие команды станут эквивалентными:

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

В таком виде все выглядит понятнее. Также пользователи частенько добавляют команду, выводящую последние коммиты:

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

Теперь для просмотра последнего коммита достаточно написать:

```
$ git last
commit 66938dae3329c7aeb598c2246a8e6af90d04646
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 новые команды созданным вами псевдонимом. А что делать в случае, когда нужно запустить внешнюю команду, а не команду, встроенную в Git? Такую команду следует начинать с символа `!`. Этот прием часто применяется при написании собственных инструментов для работы с Git-репозиторием. Вот пример создания псевдонима для команды `git visual`, служащей для запуска `gitk`:

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

Заключение

Итак, вы научились выполнять все базовые локальные операции в Git: создавать или клонировать репозитории, вносить изменения, индексировать и фиксировать их, а также просматривать историю всех сделанных в репозитории изменений. Пришла пора познакомиться с самым важным преимуществом Git — моделью ветвления.

3

Ветвления в Git

Почему каждая VCS в той или иной форме поддерживает ветвление? Термин *ветвление* (branching) означает отклонение от основной линии разработки, после которого работа перестает затрагивать эту самую основную линию. Во многих VCS это крайне сложный процесс, зачастую требующий создания новой копии папки с исходным кодом, что в случае больших проектов занимает изрядное время.

Некоторые называют модель ветвлений в Git основным преимуществом этой системы, однозначно выделяющим его из ряда прочих VCS. В чем же состоит основное отличие? Способ создания новых веток в Git невероятно упрощен, что делает эту процедуру практически мгновенной, и так же быстро позволяет переходить от одной ветки к другой и обратно. В отличие от других VCS в Git поддерживается рабочий процесс с частыми ветвлениями и слияниями, порой осуществляемыми несколько раз в день. Понимание этой конструктивной особенности и умение ею пользоваться дает вам в руки уникальный и мощный инструмент, полностью меняющий способы разработки.

Суть ветвления

Чтобы понять, каким способом в Git выполняется ветвление, нужно вернуться немного назад и вспомнить, как именно Git сохраняет данные. В главе 1 мы говорили о том, что Git хранит не последовательности изменений, а наборы состояний.

При фиксации очередного состояния Git сохраняет объект-коммит, содержащий указатель на снимок индексированного содержимого. Еще этот объект содержит имя и адрес электронной почты автора, введенный вами комментарий и указатели

на коммит или коммиты, которые являются прямыми предками текущего. Начальный коммит предков не имеет, у обычного коммита один предок, а у коммитов, полученных слиянием двух и более веток, целый набор предков.

Для наглядности предположим, что у вас есть папка с тремя файлами, которые требуется проиндексировать и зафиксировать. При индексировании файлов для каждого из них вычисляется контрольная сумма (хеш SHA-1, упоминавшийся в главе 1), их версии сохраняются в Git-репозиторий (Git трактует их как массивы двоичных данных), а контрольные суммы добавляются в область индексирования:

```
$ git add README test.rb LICENSE
$ git commit -m 'начальный коммит моего проекта'
```

При создании коммита командой `git commit` Git вычисляет контрольную сумму каждой выложенной папки (в нашем случае все ограничивается корневым каталогом проекта) и сохраняет эти объекты-деревья в Git-репозиторий. Затем Git создает объект-коммит с метаданными и указателем на корень дерева проекта. Все это дает возможность при необходимости воссоздать снимок состояния.

В настоящее время ваш Git-репозиторий содержит пять объектов: один массив двоичных данных (blob) с содержимым трех файлов, одно дерево (tree), отображающее содержимое папки и указывающее, какие имена файлов сохранены как массивы двоичных данных, и один коммит с указателем на корень дерева и все свои метаданные (рис. 3.1).

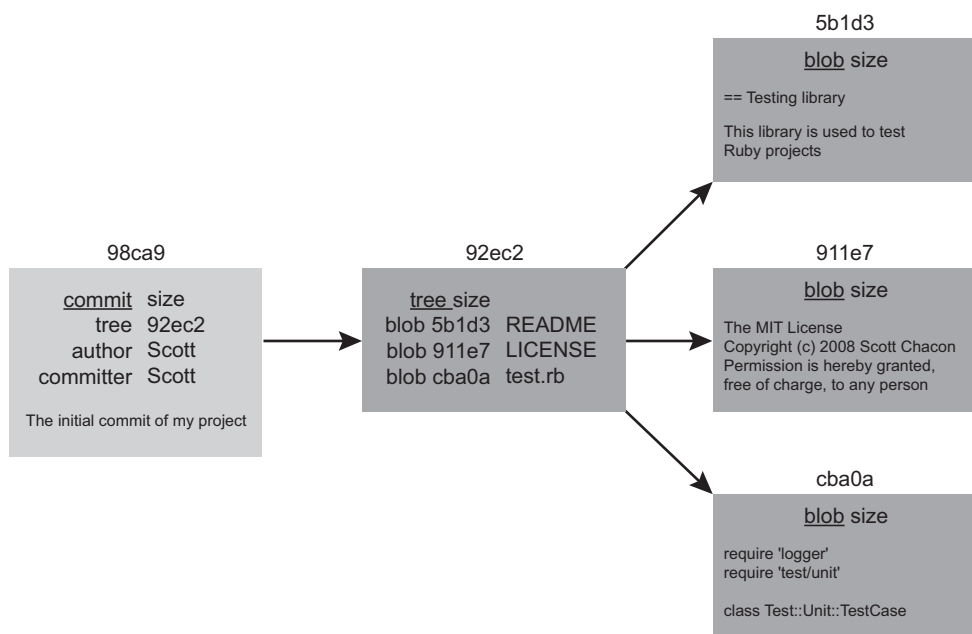


Рис. 3.1. Коммит и его дерево

Если после внесения изменений снова произвести фиксацию, новый коммит будет содержать указатель на коммит, сделанный непосредственно перед ним (рис. 3.2).

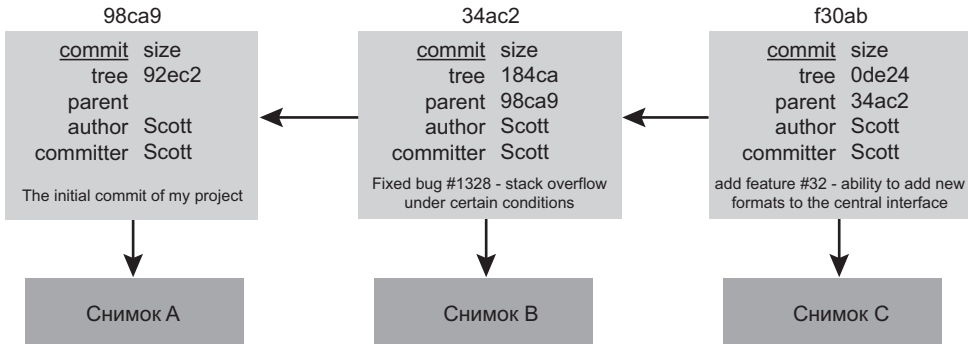


Рис. 3.2. Коммиты и их предки

Ветка в Git представляет собой всего лишь легкий, подвижный указатель на один из этих коммитов. По умолчанию в Git ей присваивается имя **master**. Как только вы начинаете создавать коммиты, появляется ветка **master**, указывающая на последнее зафиксированное вами состояние. При каждой следующей фиксации этот указатель автоматически смещается вперед (рис. 3.3).

ПРИМЕЧАНИЕ

Ветка **master** в Git ничем не отличается от остальных. Но она присутствует практически в каждом репозитории, потому что создается командой `git init` по умолчанию, а большинство пользователей не утруждают себя переименованием.

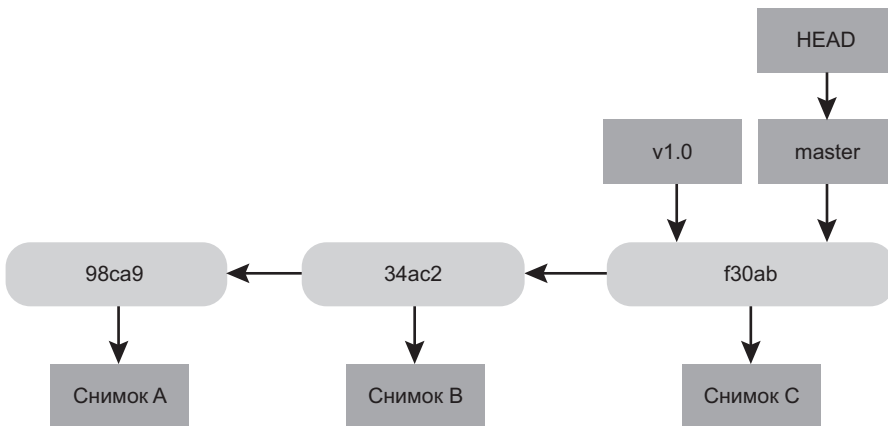


Рис. 3.3. Ветка и история ее коммитов

Создание новой ветки

Что происходит при создании новой ветки? Появляется новый указатель, который можно перемещать. Предположим, вы создаете ветку **testing**. Эта операция выполняется командой **git branch**:

```
$ git branch testing
```

Появится новый указатель на ваш текущий коммит (рис. 3.4).

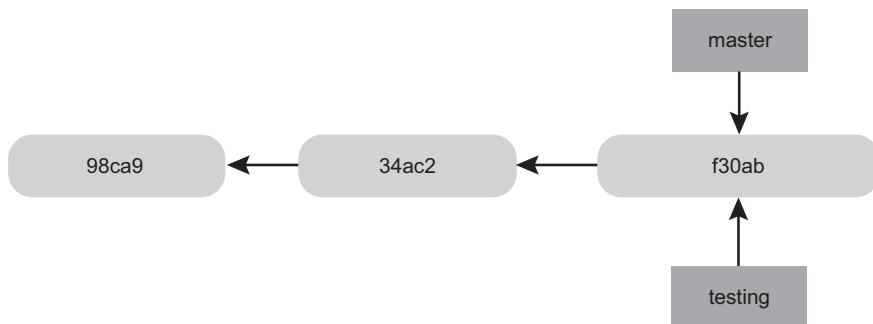


Рис. 3.4. Две ветки, указывающие на один и тот же набор коммитов

Откуда Git узнает, какой именно коммит является текущим? Для этого он хранит специальный указатель **HEAD**. Имейте в виду, что в данном случае смысл указателя **HEAD** не такой, как в других привычных вам VCS, например Subversion или CVS. В Git он указывает на локальную ветку, в которой вы находитесь в данный момент (рис. 3.5). Пока это все еще ветка **master**. Команда **git branch** всего лишь создала новую ветку, но не перевела вас в нее.

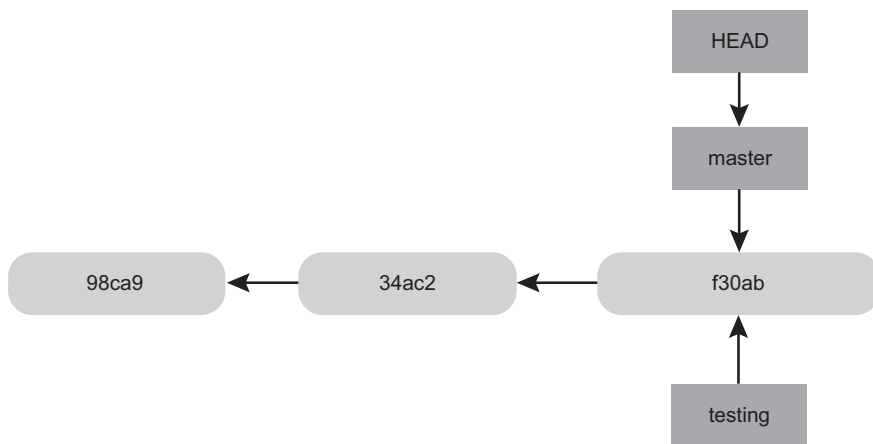


Рис. 3.5. Указатель HEAD нацелен на текущую ветку

Узнать, куда именно нацелены указатели веток, позволяет команда `git log` с параметром `--decorate`.

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

Легко увидеть, что ветки `master` и `testing` находятся непосредственно рядом с коммитом `f30ab`.

Смена веток

Переход на существующую ветку реализует команда `git checkout`. Перейдем на ветку `testing`:

```
$ git checkout testing
```

Теперь указатель `HEAD` нацелен на ветку `testing` (рис. 3.6).

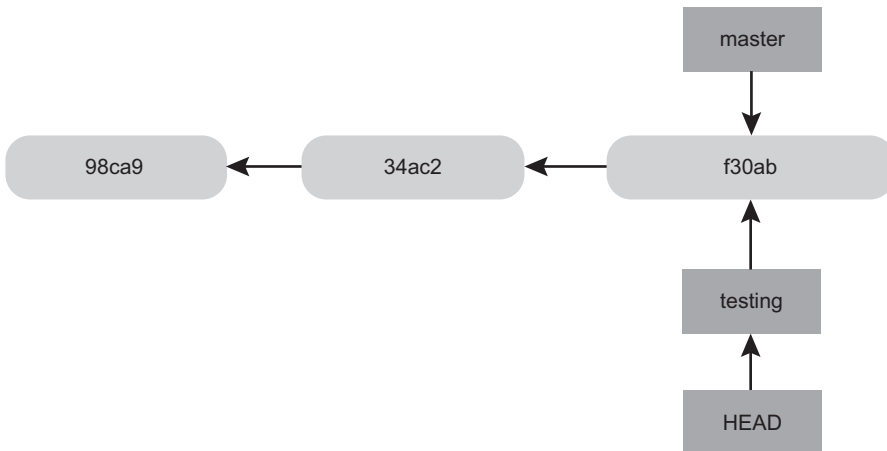


Рис. 3.6. Указатель `HEAD` нацелен на текущую ветку

Какое все это имеет значение? Давайте еще раз зафиксируем состояние:

```
$ vim test.rb
$ git commit -a -m 'внесено изменение'
```

Оказывается, ваша ветка `testing` сместилась вперед, в то время как ветка `master` осталась связанной с коммитом, текущим на момент перехода на другую ветку, при помощи команды `git checkout` (рис. 3.7).

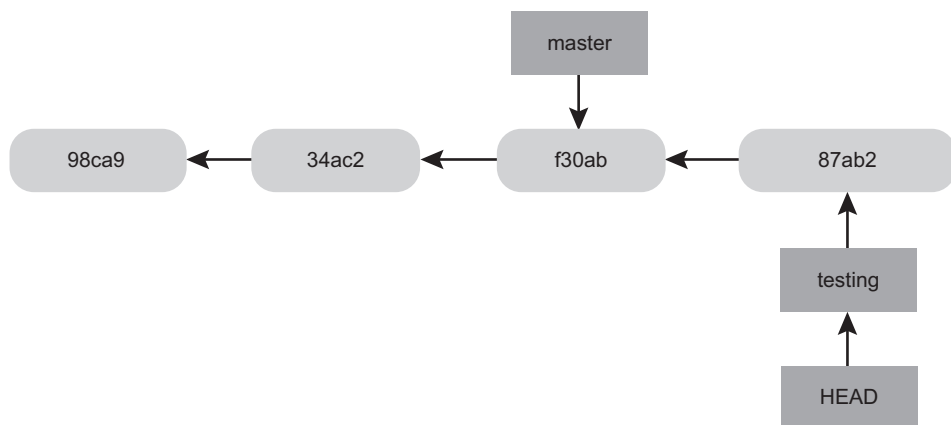


Рис. 3.7. Ветка с указателем HEAD сместилась вперед после нового коммита

Вернемся на ветку `master` (рис. 3.8):

```
$ git checkout master
```

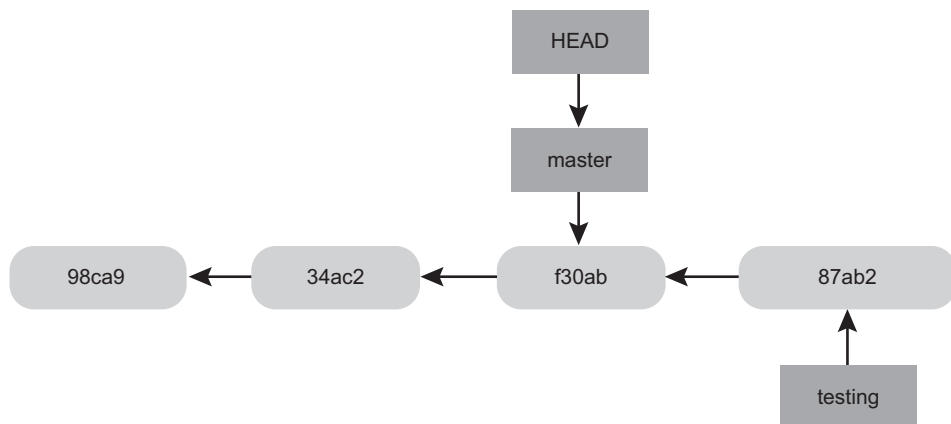


Рис. 3.8. Смещение указателя HEAD при переключении

Эта команда выполняет два действия. Она возвращает указатель `HEAD` ветке `master` и приводит файлы в рабочей папке в состояние, зафиксированное в этой ветке. Это означает, что все изменения, вносимые после этого момента, будут ответвляться от старой версии проекта. По сути, так вы отменяете результаты работы, сделанной в ветке `testing`, и идете дальше в другом направлении.

Внесем несколько изменений и зафиксируем их:

```
$ vim test.rb
$ git commit -a -m 'внесены дополнительные изменения'
```

СМЕНА ВЕТОК МЕНЯЕТ ФАЙЛЫ В РАБОЧЕЙ ПАПКЕ

Важно понимать, что переход на другую ветку сопровождается сменой файлов в рабочей папке. При возвращении в более старую ветку содержимое рабочей папки приобретет тот вид, который оно имело перед последним коммитом в этой ветке. Если система Git не сможет вернуть файлы в это состояние, она просто не даст вам выполнить переключение.

История проекта разветвилась. Вы создали ветку, перешли в нее, поработали немного, вернулись в основную ветку и снова произвели некие действия. Внесенные в этих двух случаях изменения оказались принадлежащими разным веткам: вы можете переходить из одной ветки в другую, а при необходимости объединять их (рис. 3.9). И все это при помощи простых команд `branch`, `checkout` и `commit`.

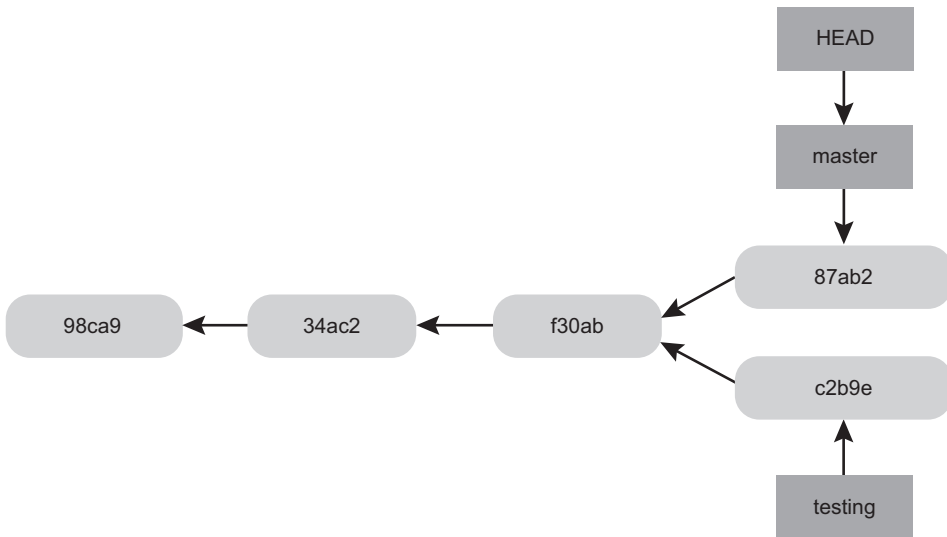


Рис. 3.9. Расходящаяся история

Суть происходящего демонстрирует и команда `git log`. В форме `git log --oneline --decorate --graph --all` она выводит историю коммитов, показывая места расположения указателей и точки расхождения.

```

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

Так как ветка в Git представляет собой обычный файл с 40 символами контрольной суммы SHA-1 коммита, на который она указывает, операции создания и удаления

веток практически не требуют ресурсов. Создать новую ветку так же просто, как записать в файл 41 байт (40 символов и знак переноса строки).

Это разительно отличается от механизма ветвления в большинстве VCS, требующего копирования всех файлов проекта в другую папку. В зависимости от размера проекта эта операция может занять от нескольких секунд до нескольких минут, в то время как в Git это моментальный процесс. Кроме того, благодаря записи предков при фиксации состояний поиск базовой версии для слияния автоматически выполняется за нас и в общем случае элементарно реализуется на практике. Именно эти конструктивные особенности поощряют разработчиков создавать ветки и активно с ними работать.

Сейчас и вы увидите, как это все действует.

Основы ветвления и слияния

Простые случаи ветвления и слияния мы рассмотрим на схемах, которые могут пригодиться вам при решении реальных задач. Последовательность действий будет следующей:

1. Выполняем некие действия на сайте.
2. Создаем ветку для новой истории, над которой тоже нужно работать.
3. В этой ветке тоже производим некие действия.

А теперь предположим, что нам позвонили и сообщили о важной проблеме, требующей срочного решения. Поступаем следующим образом:

4. Переключаемся в производственную ветку.
5. Создаем ветку для решения проблемы.
6. После тестирования выполняем слияние побочной ветки и отправляем ее в разработку.
7. Возвращаемся к первоначальной задаче и продолжаем работу.

Основы ветвления

В качестве исходных условий представим, что у вас уже есть пара коммитов проекта (рис. 3.10).

Вы решили приступить к работе над проблемой 53, фигурирующей в системе отслеживания ошибок вашей фирмы. Создать ветку и сразу перейти туда позволяет команда `git checkout` с параметром `-b` (рис. 3.11):

```
$ git checkout -b iss53
Переход на новую ветку "iss53"
```

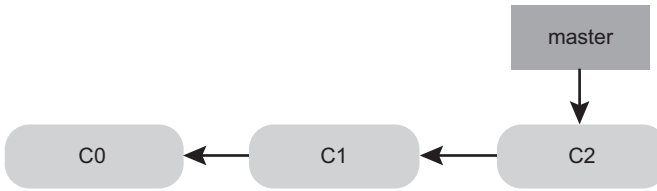



Рис. 3.10. Простая история коммитов

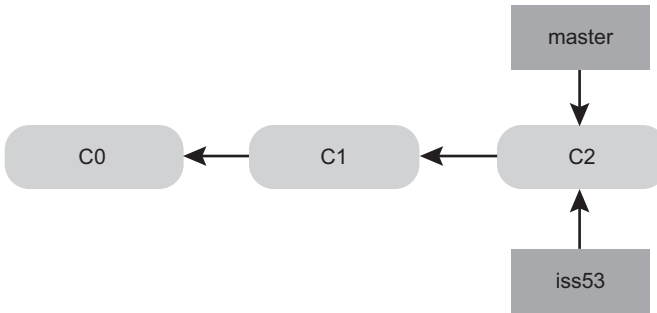


Рис. 3.11. Создание указателя на новую ветку

Это сокращенная запись для команд:

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

Вы работаете над сайтом и несколько раз фиксируете состояние. При этом, как показано на рис. 3.12, ветка **iss53** смещается вперед, так как вы в ней находитесь (то есть именно на нее нацелен указатель **HEAD**):

```
$ vim index.html
$ git commit -a -m 'добавлен новый нижний колонтитул [проблема 53]'
```

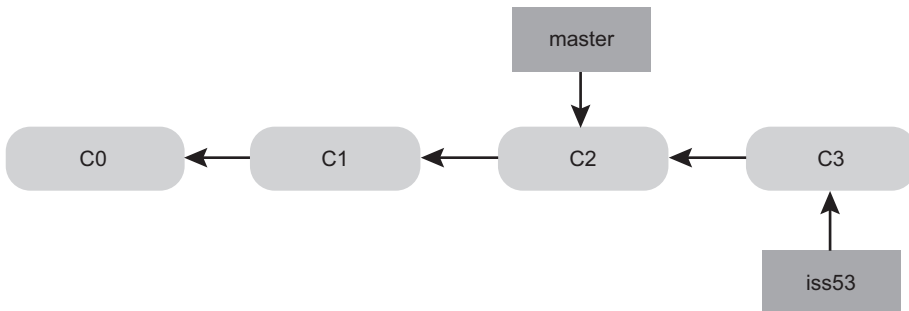


Рис. 3.12. Ветка **iss53** в процессе работы сместилась вперед

Именно на этом этапе возникает звонок с сообщением о проблеме. Благодаря Git вам не нужно выполнять развертывание исправлений вместе с внесенными в ветку `iss53` изменениями или ценой массы усилий отменять эти изменения перед тем, как приступить к устранению проблемы. Достаточно вернуться в ветку `master`.

Однако при наличии в рабочей папке или в области индексирования незафиксированных изменений, конфликтующих с веткой, в которую вы хотите перейти, система Git не позволит выполнить переключение. Перед переходом из ветки в ветку лучше всего иметь чистое рабочее состояние. Это ограничение можно обойти (например, скрыв или переписав коммит), но об этом мы поговорим чуть позже. На данный же момент предположим, что все изменения зафиксированы и ничто не мешает вам вернуться в ветку `master`:

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

Теперь рабочая папка проекта вернулась в то состояние, в котором она пребывала перед началом работы над проблемой 53, и можно сконцентрироваться на решении новой задачи (на рисунках она будет обозначаться как `hotfix`). Важно помнить, что при смене веток Git возвращает рабочую папку в то состояние, которое она имела на момент последнего коммита этой ветки. Добавление, удаление и изменение файлов при этом происходит автоматически.

Итак, у вас есть ошибка, которую нужно исправить. Создадим ветку `hotfix`, с которой мы будем работать, пока не решим поставленную задачу (рис. 3.13):

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

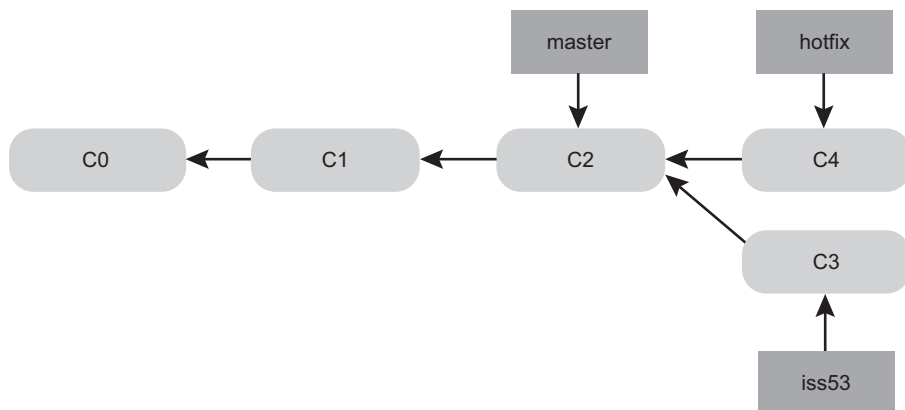


Рис. 3.13. Ветка `hotfix` базируется на ветке `master`

После того как проблема будет решена, вы сможете произвести тестирование, убедиться, что найденное решение работает, и объединить ветку **hotfix** с веткой **master**, чтобы внедрить внесенные изменения в готовый код. Эта операция выполняется командой **git merge**:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
index.html | 2 ++
1 file changed, 2 insertions(+)
```

Обратите внимание на словосочетание «fast-forward» в описании результатов слияния. Так как подвергшаяся слиянию ветка указывала на коммит, являющийся предком текущего коммита, система Git просто сдвинула указатель вперед. Другими словами, при попытке выполнить слияние с коммитом, которого можно достичь, просматривая историю первого коммита, Git просто перемещает указатель вперед, ведь расходящиеся изменения, требующие объединения друг с другом, в данном случае отсутствуют — данный тип слияния называют *перемоткой* (fast-forward).

Теперь ваши изменения зафиксированы, на что указывает ветка **master**, и их можно внедрить в рабочий код (рис. 3.14).

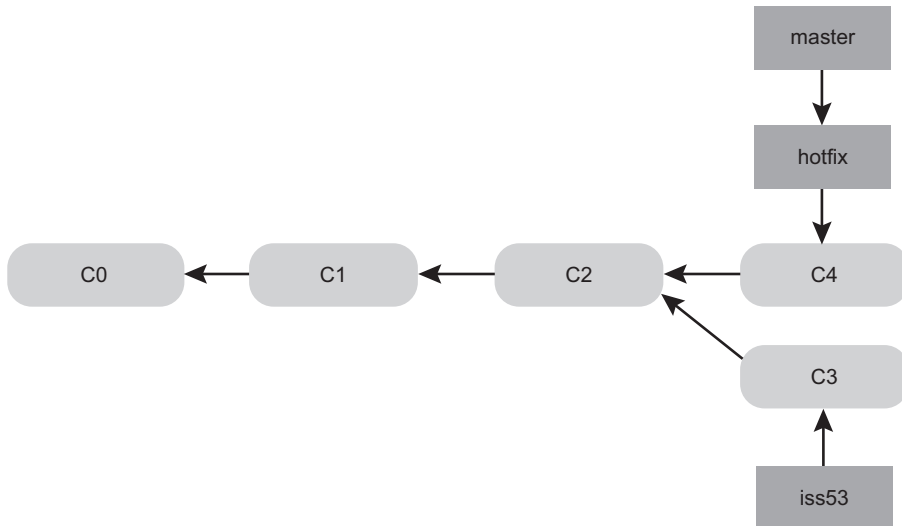


Рис. 3.14. После процедуры слияния ветки **master** и **hotfix** указывают на один коммит

После внедрения важного исправления можно вернуться к прерванной работе. Но сначала следует удалить ветку **hotfix**, так как она нам больше не понадобится, — на этот коммит уже указывает ветка **master**. Для этого достаточно добавить параметр **-d** к команде **git branch**:

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

Теперь можно вернуться в ветку, где ведется работа над проблемой 53, и продолжить ее решение (рис. 3.15):

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

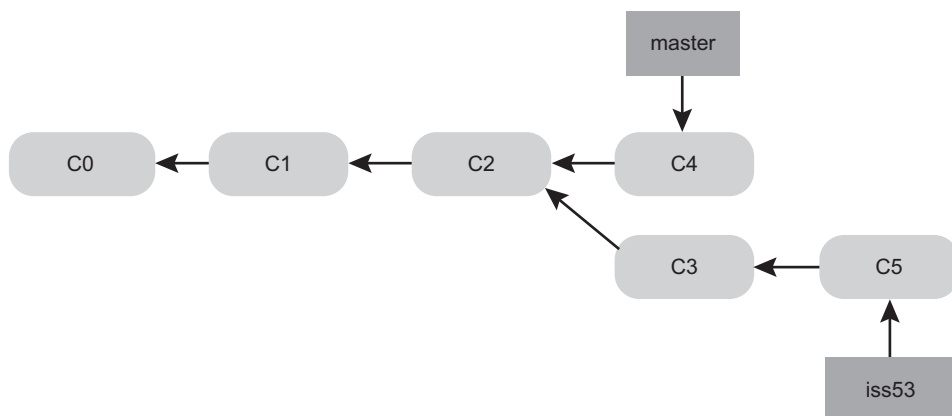


Рис. 3.15. Продолжение работы с веткой iss53

Следует заметить, что изменения, внесенные в ветку **hotfix**, не касаются файлов из ветки **iss53**. Ситуацию можно изменить, слив ветку **master** с веткой **iss53**, — это делает команда **git merge master**. Или отложите этот процесс до момента, пока решите включить содержимое ветки **iss53** в ветку **master**.

ОСНОВЫ СЛИЯНИЯ

Предположим, работа над проблемой 53 завершена и вы готовы вставить сделанные изменения в ветку **master**. Для этого следует выполнить слияние с веткой **iss53** уже знакомым вам способом — как мы демонстрировали слияние с веткой **hotfix**. Достаточно перейти в ветку, с которой будет осуществляться слияние, и воспользоваться командой **git merge**:

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
 README | 1 +
 1 file changed, 1 insertion(+)
```

Но в данном случае наблюдаются отличия от ситуации со слиянием ветки `hotfix`. Дело в том, что в некоторой точке история разработки разделилась. Коммит, соответствующий текущей ветке, не является прямым предком для ветки, с которой осуществляется слияние, поэтому система Git должна проделать кое-какую работу. В рассматриваемом случае Git выполнит простое трехэтапное слияние, используя два состояния, на которые указывают вершины веток, и их общего предка (рис. 3.16).

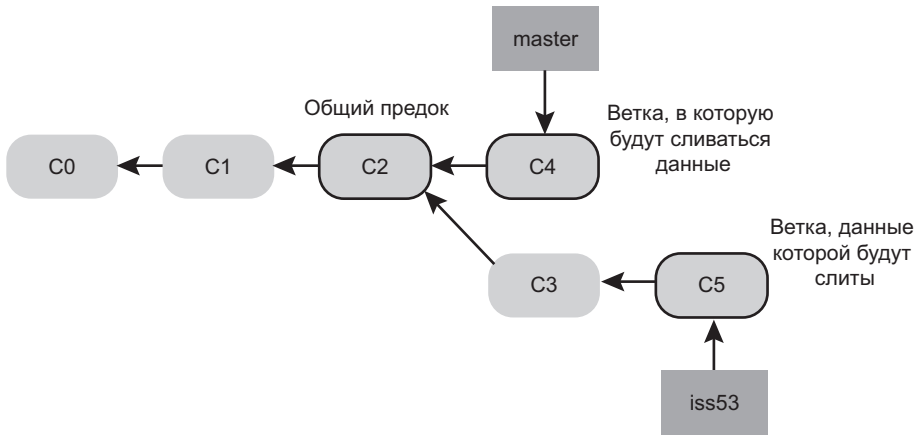


Рис. 3.16. Три снимка состояния, используемые при типичном слиянии

Вместо того чтобы просто сместить указатель ветки вперед, Git формирует новое состояние, возникающее из трехэтапного слияния, и автоматически создает коммит для этого состояния. Его еще называют коммитом слияния (`merge commit`), и его особенность состоит в наличии нескольких предков (рис. 3.17).

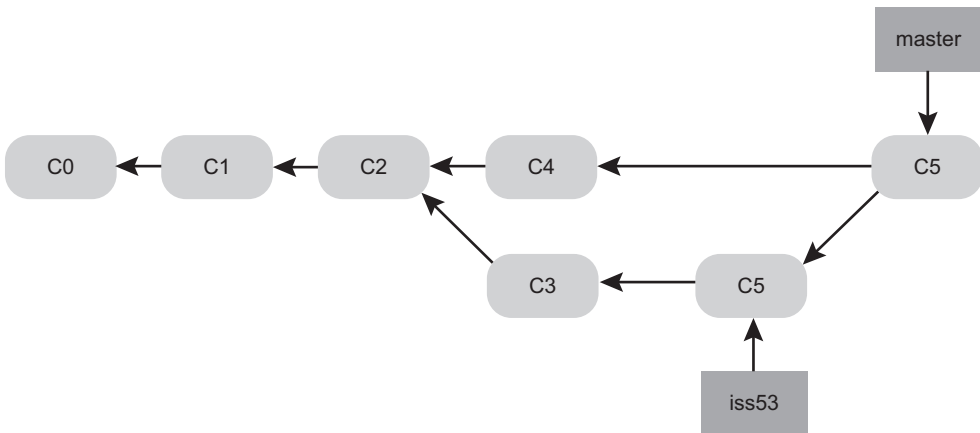


Рис. 3.17. Коммит слияния

Стоит отметить, что Git определяет наиболее подходящего общего предка для слияния веток; это отличается от более старых инструментов, таких как CVS или Subversion (до версии 1.5), где разработчик сам должен выбирать основу для слияния. Именно это делает процедуру слияния в Git такой простой.

Теперь, когда слияние выполнено, ветка `iss53` не нужна. Можно закрыть вопрос в системе обработки заявок и удалить ветку:

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

Конфликты при слиянии

Процесс слияния далеко не всегда проходит гладко. Если в двух ветках, которые вы собираетесь слить, вы внесли разные изменения в один и тот же файл, Git не сможет просто взять и объединить их. Если бы при решении проблемы 53 вы отредактировали ту же самую часть файла, что и в ветке `hotfix`, возник бы конфликт 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.
```

В этом случае Git не может автоматически создать коммит слияния. Система приостанавливает процесс до момента разрешения конфликта. Посмотреть, какие файлы не прошли слияние после возникновения конфликта, позволяет команда `git status`:

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

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

    both modified:   index.html

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

Все, что относится к области конфликта слияния, помечено как неслитое (`unmerged`). Система Git добавляет к проблемным файлам стандартные метки, позволяющие открывать эти файлы вручную и разрешать конфликты. Ваш файл содержит раздел, который выглядит примерно вот 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
```

Версия с указателем **HEAD** (из вашей ветки **master**, так как именно в нее вы перешли перед выполнением команды **merge**) располагается в верхней части блока (то есть выше набора символов **=====**), а версия из ветки **iss53** показана в нижней части. Для разрешения конфликта следует или выбрать одну из версий, или каким-то образом объединить их. К примеру, можно заменить весь блок вот этим:

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

В это решение вошло небольшое количество информации из каждой части, а строки **<<<<<<**, **=====** и **>>>>>>** удалены. Разобравшись с каждым таким разделом в каждом из проблемных файлов, выполните для каждого из этих файлов команду **git add**. Индексируя файл, вы помечаете его как неконфликтующий. Для разрешения конфликтов можно вызвать командой **git mergetool** графический интерфейс:

```
$ 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 diffuze 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 предпочитаете инструмент слияния, отличный от заданного по умолчанию (в данном случае система Git запустила утилиту **opendiff** на Mac), список доступных инструментов выводится после строки **'git mergetool' will now attempt to use one of the following tools**, — достаточно ввести название нужного вам инструмента.

ПРИМЕЧАНИЕ

Более специализированные инструменты, предназначенные для разрешения запутанных конфликтов слияния, рассмотрены чуть позже.

После завершения работы с инструментом слияния Git спросит, удалось ли разрешить конфликты. В случае положительного ответа файл индексируется, что означает разрешение конфликта. Убедиться в том, что это действительно так, позволяет команда **git status**:

```
$ 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 вы довольны полученным результатом и удостоверились в том, что все ранее конфликтовавшие файлы проиндексированы, остается завершить слияние командой **git commit**. В этом случае сообщение фиксации по умолчанию выглядит так:

```
Merge branch 'iss53'

Conflicts:
  index.html
#
# Кажется, вы можете зафиксировать слияние.
# Если это не так, удалите файл
#   .git/MERGE_HEAD
# и сделайте еще одну попытку.
# Введите сообщение фиксации для ваших изменений. Строки, в начале
# которых '#' игнорируются, а пустое сообщение отменяет коммит.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#   modified:   index.html
#
```

Это сообщение можно дополнить информацией о том, как вы разрешили конфликт, если считаете, что это может пригодиться другим пользователям. Если подоплека ваших действий не очевидна, имеет смысл написать, почему вы поступили именно так, а не иначе.

Управление ветками

Теперь, когда вы знаете, как создаются, сливаются и удаляются ветки, рассмотрим другие инструменты, которые пригодятся при постоянной работе с ветками.

Команда **git branch** позволяет не только создавать и удалять ветки. Запущенная без аргументов, она выводит на экран список имеющихся веток:

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


Обратите внимание на символ `*` перед веткой `master`: он указывает, что именно в этой ветке вы сейчас находитесь (то есть именно на нее нацелен указатель `HEAD`). Если сейчас выполнить коммит, ветка `master` дополнится новыми изменениями. Последний коммит, выполненный в каждой ветке, демонстрирует команда `git branch -v`:

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

Полезные параметры `--merged` и `--no-merged` оставляют в этом списке только те ветки, которые вы слили или не слили с текущей веткой. Скажем, для просмотра веток, объединенных с текущей веткой, следует написать `git branch --merged`:

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

В списке вы видите ветку `iss53`, с которой ранее было осуществлено слияние. Ветки, перед именами которых отсутствует символ `*`, в общем случае удаляются командой `git branch -d`; все данные оттуда уже включены в другую ветку, так что вы ничего при этом не теряете.

Просмотр списка веток, данные которых еще не слиты в другие ветки, осуществляется командой `git branch --no-merged`:

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

Как видите, отображается другая ветка. Так как она содержит данные, пока не подвергшиеся слиянию, удалить ее командой `git branch -d` не получится:

```
$ 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 все равно хотите удалить эту ветку, используйте параметр `-D`, как указано в подсказке.

Приемы работы с ветками

Итак, с основами ветвления и слияния вы познакомились. Возникает вопрос: что еще можно делать с ветками? В этом разделе мы разберем некоторые стандартные рабочие приемы, ставшие возможными благодаря облегченной процедуре ветвления. Возможно, что-то из этого вы сможете включить в собственный цикл разработки.

Долгоживущие ветки

Так как в Git применяется простое трехэтапное слияние, ничто не мешает многократно объединять ветки в течение длительного времени. То есть у вас может быть несколько постоянно открытых веток, применяемых для разных этапов цикла разработки; содержимое некоторых из них будет регулярно сливаться в другие ветки.

Многие разработчики, использующие Git, придерживаются именно такого подхода, оставляя полностью стабильный код только в ветке **master**. При этом существует и параллельная ветка с именем **develop** или **next**, служащая для работы и тестирования стабильности. После достижения стабильного результата ее содержимое сливается в ветку **master**. Она используется для объединения завершенных задач из тематических веток (временных веток наподобие **iss53**), чтобы гарантировать, что эти задачи проходят тестирование и не вносят ошибок.

По сути, мы рассматриваем указатели, перемещающиеся по линии фиксируемых нами изменений. Стабильные ветки находятся в нижнем конце истории коммитов, а самые свежие наработки — ближе к ее верхней части (рис. 3.18).

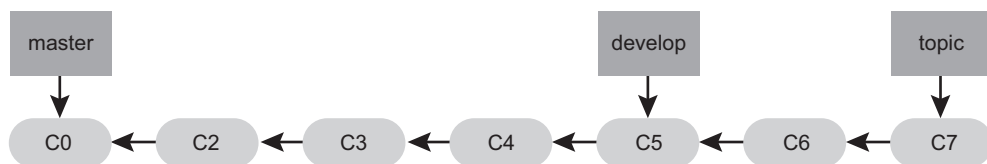


Рис. 3.18. Линейное представление повышения стабильности веток

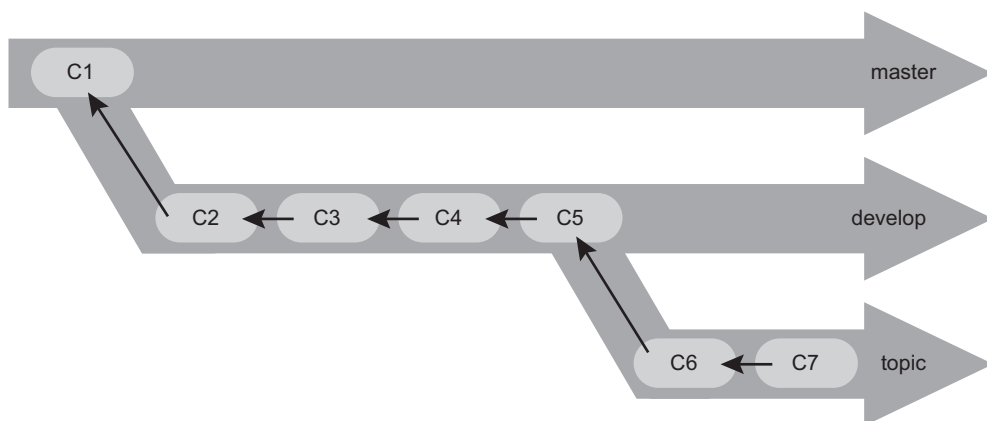


Рис. 3.19. Представление диаграммы стабильности веток в виде многоуровневого накопителя

В общем случае можно представить набор рабочих накопителей, в котором наборы коммитов перемещаются на более стабильный уровень только после полного тестирования (рис. 3.19).

Число уровней стабильности можно увеличить. В крупных проектах зачастую появляется ветка **proposed** или **pu** (сокращение от **proposed updates** — предложенные изменения), объединяющая ветки с содержимым, которое невозможно включить в ветку **next** или **master**. Фактически каждая ветка представляет собственный уровень стабильности; как только он повышается, содержимое сливается в ветку, расположенную выше. Разумеется, можно и 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 (*topic branch*) называется временная ветка, создаваемая и используемая для работы над конкретной функциональной возможностью или решения сопутствующих задач. Скорее всего, при работе с другими VCS вы никогда ничего подобного не делали, так как там создание и слияние веток — затратные операции. Но в Git принято 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**. Для каждой из них было выполнено несколько коммитов, после чего сразу же после слияния с основной веткой они были удалены. Такая техника позволяет быстро и радикально осуществлять переключения контекста. Работа разделена по уровням, и все изменения в конкретной ветке относятся к определенной теме, а значит, во время просмотра кода проще понять, что и где было сделано. Ветку с внесенными в нее изменениями можно хранить минуты, дни или даже месяцы, и выполнять ее слияние, только когда это действительно требуется, независимо от порядка создания веток в рамках проекта и порядка работы с ними.

Предположим, мы работаем в ветке **master**, ответвляемся для решения попутной проблемы (**iss91**), некоторое время занимаемся ею, затем создаем ветку, чтобы попробовать решить эту задачу другим способом (**iss91v2**), возвращаемся в ветку **master**, выполняем там некие действия и создаем новую ветку для действий, в результате которых не уверены (ветка **dumbidea**). Результирующая история коммитов будет выглядеть примерно так, как показано на рис. 3.20.

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 соответствовать рис. 3.21.

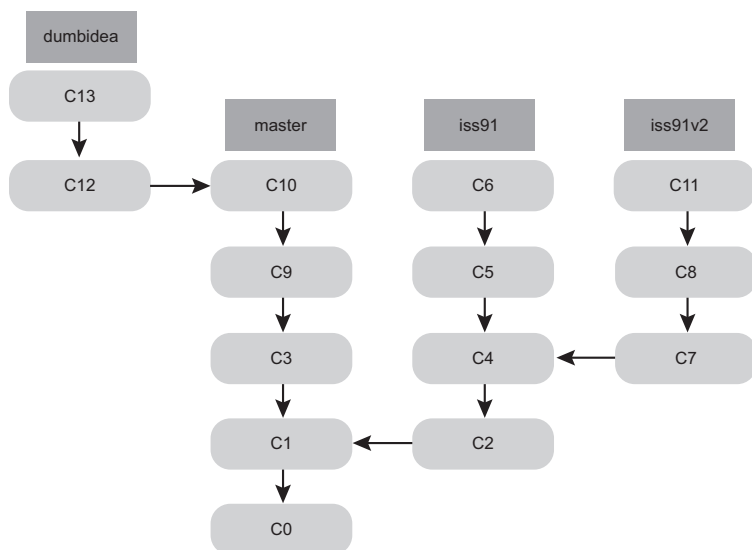


Рис. 3.20. Набор тематических веток

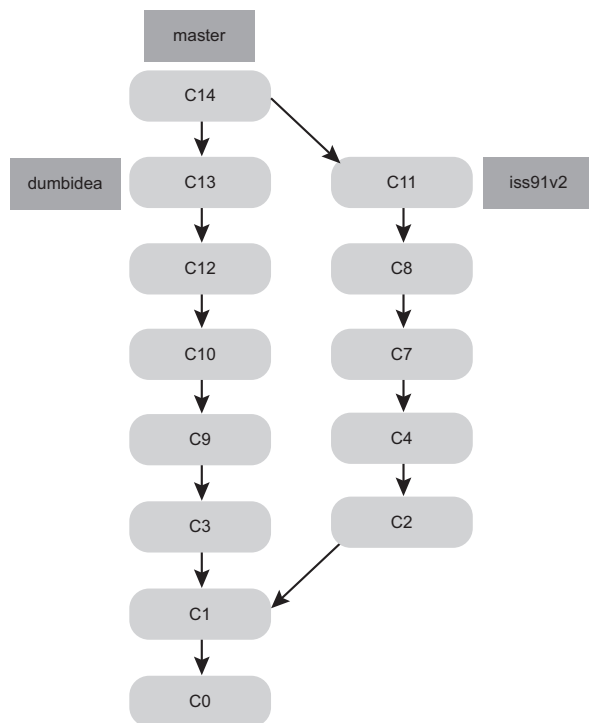


Рис. 3.21. История после слияния веток **dumbidea** и **iss91v2**

Более подробно допустимые варианты рабочих схем для проектов рассматриваются в главе 5, 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 репозитории Git, связь с сервером не требуется.

Удаленные ветки

Удаленные ветки (remote branches) представляют собой ссылки на состояния веток в удаленных репозиториях. Перемещать их локально вы не можете; они смещаются автоматически при каждом подключении по сети. Удаленные ветки работают как закладки, напоминающие, где в удаленных репозиториях находились соответствующие ветки во время вашего последнего подключения к ним.

Они имеют форму (имя удаленного репозитория)/(ветка). К примеру, чтобы увидеть, как выглядела ветка `master` на сервере `origin` во время последнего взаимодействия с ним, проверьте ветку `origin/master`. Если вы выполняли совместную работу с коллегой и он отправил на сервер ветку `iss53`, у вас может присутствовать собственная локальная копия этой ветки; но ветка на сервере будет указывать на коммит по адресу `origin/iss53`.

Проясним ситуацию на примере. Предположим, у вас есть Git-сервер с адресом `git.ourcompany.com`. При клонировании с этого адреса Git автоматически присваивает копии имя `origin`, извлекает все данные, создает указатель на местоположение ветки `master` и присваивает ему локальное имя `origin/master`. Одновременно вам предоставляется собственная локальная ветка `master`, берущая начало в том же самом месте, что и одноименная ветка копии, что дает вам отправную точку для начала работы (рис. 3.22).

ИМЯ ORIGIN

Имя ветки «`origin`», как и имя «`master`», в Git не несет особого значения. Имя `master` широко распространено потому, что оно по умолчанию присваивается ветке, порождаемой командой `git init`, а имя `origin` по умолчанию присваивается удаленной ветке, порождаемой командой `git clone`. Если написать `git clone -o booyah`, по умолчанию вы будете работать с удаленной веткой `booyah/master`.

Если во время вашей работы на локальной ветке `master` кто-то отправит на сервер `git.ourcompany.com` результаты своего труда и обновит удаленную ветку `master`, истории веток будут развиваться по-разному (рис. 3.23). Кроме того, до момента связи с сервером `origin` указатель `origin/master` у вас двигаться не будет.

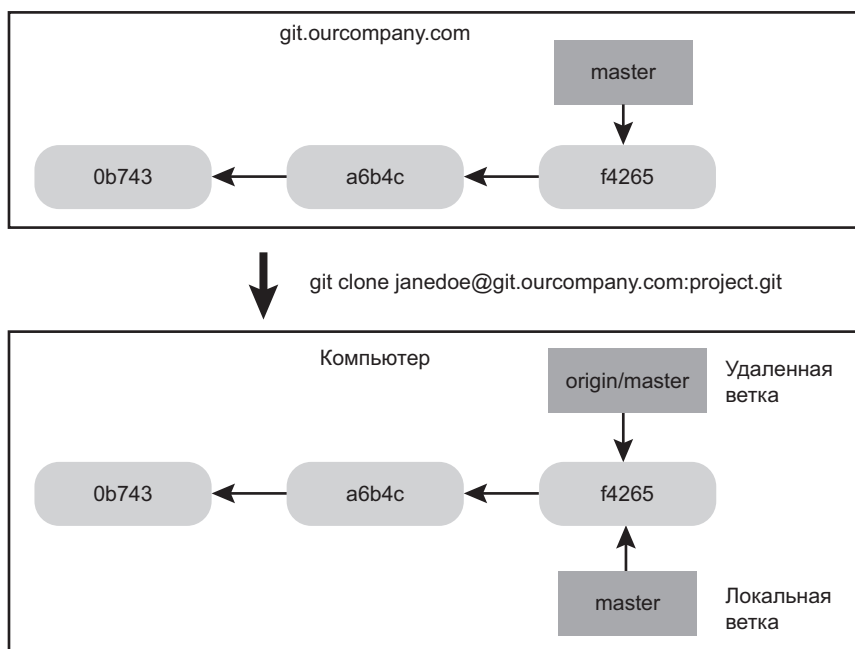


Рис. 3.22. Удаленный и локальный репозитории после клонирования

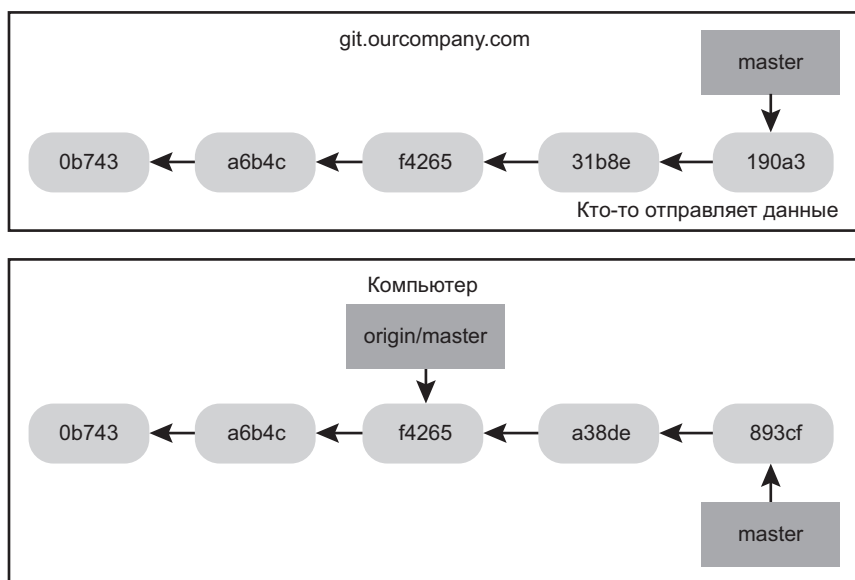


Рис. 3.23. Локальная и удаленная версии ветки могут различаться

Синхронизация работы осуществляется командой `git fetch origin`. Она ищет сервер с именем «origin» (в данном случае это сервер `git.ourcompany.com`), извлекает оттуда все пока отсутствующие у вас данные, обновляет вашу локальную базу данных и сдвигает указатель `origin/master` на новую, более актуальную позицию (рис. 3.24).

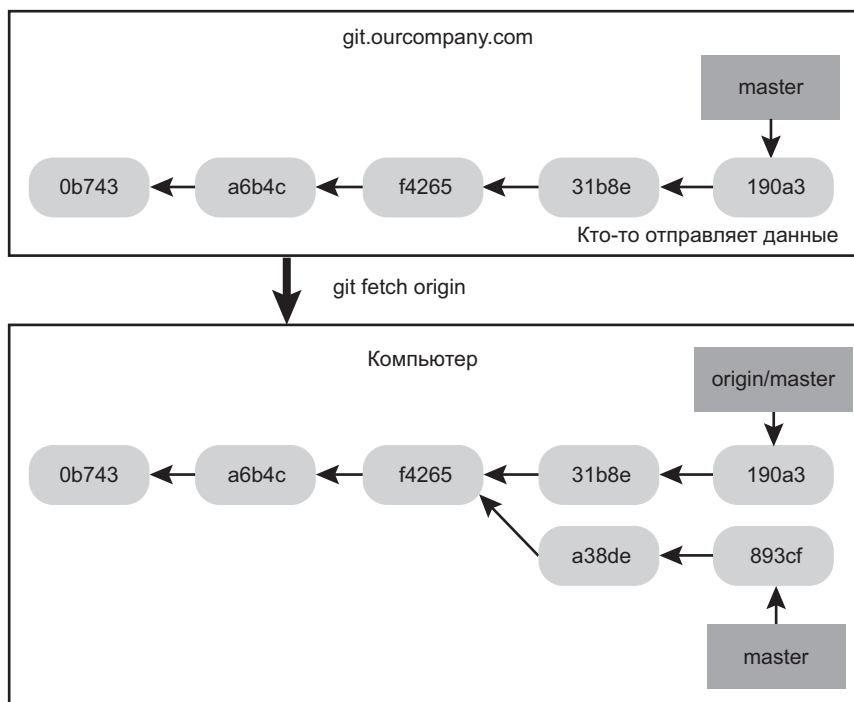
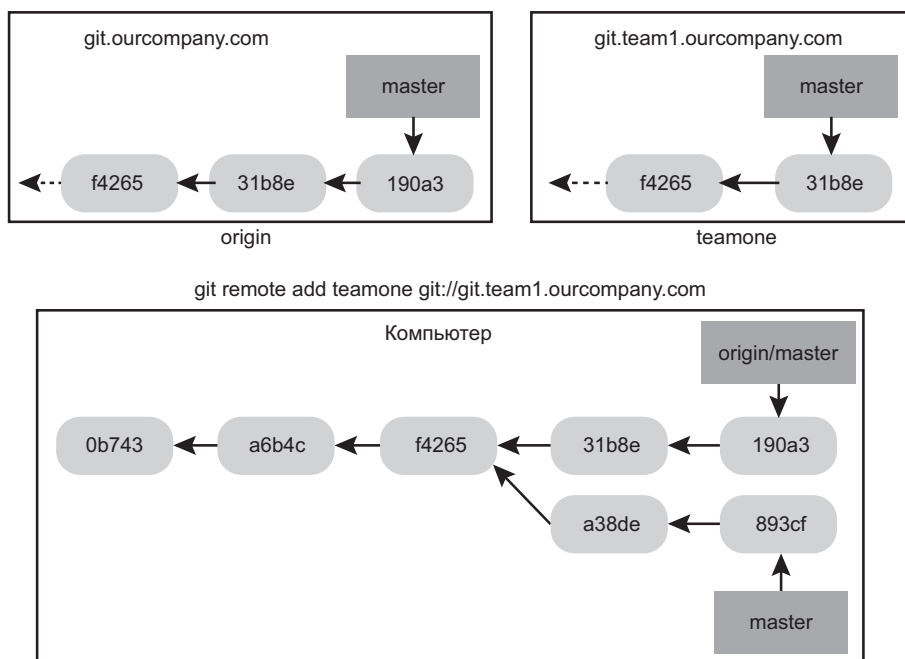
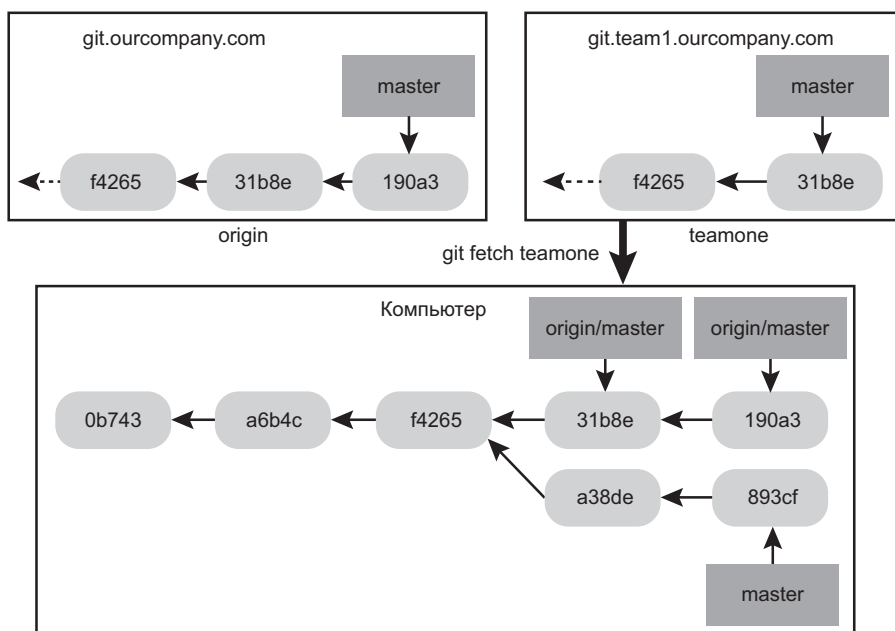


Рис. 3.24. Команда `git fetch` обновляет ссылки на удаленный репозиторий

Чтобы продемонстрировать, как будут выглядеть удаленные ветки при наличии нескольких удаленных серверов, предположим, что у вас есть еще один внутренний сервер Git, которым пользуется только одна из групп разработчиков. Этот сервер находится по адресу `git.team1.ourcompany.com`. Добавить его в качестве новой удаленной ссылки на проект, над которым вы сейчас работаете, можно уже знакомой вам командой `git remote add` (рис. 3.25). Присвойте этому удаленному серверу имя `teamone`.

Теперь можно воспользоваться командой `git fetch teamone` и получить все данные, отсутствующие у вас, но имеющиеся на сервере `teamone`. Однако поскольку пока этот сервер содержит всего лишь часть данных с сервера `origin`, система Git ничего не скачает, а только сгенерирует удаленную ветку `teamone/master`, указывающую на тот же коммит, что и ветка `master` на сервере `teamone` (рис. 3.26).

**Рис. 3.25.** Добавление еще одного удаленного сервера**Рис. 3.26.** Удаленная отслеживающая ветка для ветки teamone/master

Отправка данных

Чтобы поделиться содержимым своей ветки с окружающими, ее нужно отправить на удаленный сервер, на котором у вас есть права на запись. Автоматической синхронизации локальных веток с удаленными серверами не происходит, поэтому ветки, которые вы хотите выставить на всеобщее обозрение, следует отправлять вручную. Такой подход дает возможность работать с личными ветками независимо от коллег, предоставляя доступ только к тематическим веткам, предназначенным для совместной работы.

Предположим, у вас есть ветка **serverfix**, над которой вы хотите работать вместе с коллегами. Ее следует отправить на сервер тем же способом, каким была отправлена ваша первая ветка, то есть командой **git push (удаленный сервер) (ветка)**:

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

Это в некотором роде сокращение. Система Git автоматически превращает имя ветки **serverfix** в запись **refs/heads/serverfix:refs/heads/serverfix**, что означает «возьмите мою локальную ветку **serverfix** и используйте для обновления удаленной ветки **serverfix**». Фрагмент **refs/heads/** мы будем обсуждать в главе 10, но, как правило, его можно просто опустить. Вы также можете воспользоваться командой **git push origin serverfix:serverfix**, которая делает то же самое. Кроме того, данный формат позволяет переслать содержимое локальной ветки в удаленную ветку с другим именем. Достаточно вместо имени **serverfix** на удаленном сервере указать другой вариант, например **git push origin serverfix:awesomebranch**. В этом случае содержимое локальной ветки **serverfix** будет передано в ветку **awesomebranch** на удаленном сервере.

ПРИМЕЧАНИЕ

Каждый раз вводить свой пароль не нужно. Если для отправки данных вы используете протокол HTTPS, сервер Git будет просить вас указать имя пользователя и пароль для проверки прав доступа. По умолчанию эту информацию предлагается ввести в терминале, чтобы сервер мог сообщить вам, имеете ли вы право отправлять ему данные. Чтобы не делать этого при каждой отправке, настройте «кэш учетных данных». Проще всего несколько минут держать данную информацию в памяти, что легко достигается командой **git config --global credential.helper cache**.

Когда кто-то из ваших коллег в следующий раз решит скачать обновления с сервера, он получит ссылку на место, куда серверная версия ветки **serverfix** указывает как на удаленную 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
```

Имейте в виду, что получение новых удаленных веток при скачивании данных не означает автоматического появления их доступных для редактирования копий. Другими словами, у вас появляется не новая ветка **serverfix**, а только недоступный для редактирования указатель **origin/serverfix**.

Тем не менее данные оттуда можно слить в текущую рабочую ветку, воспользовавшись командой **git merge origin/serverfix**. Если вам требуется собственная копия ветки **serverfix**, достаточно создать ее, взяв за основу удаленную ветку:

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

В результате вы получаете для работы локальную ветку, которая начинается там же, где и ветка **origin/serverfix**.

Слежение за ветками

При переходе в локальную ветку, созданную из удаленной, автоматически появляется так называемая ветка наблюдения (tracking branch). Это локальная ветка, напрямую связанная с удаленной. Если, находясь на этой ветке, вы наберете команду **git push**, система Git автоматически поймет, на какой сервер и в какую ветку нужно отправлять данные. А команда **git pull** в этом случае скачивает все удаленные ссылки и после этого автоматически выполняет слияние в соответствующую удаленную ветку.

При клонировании репозитория автоматически создается ветка **master**, следящая за веткой **origin/master**. Именно поэтому команды **git push** и **git pull** выполняются из этой ветки без аргументов. При желании можно создавать и другие ветки наблюдения, следящие за ветками на других удаленных серверах или не отслеживающие происходящее на ветке **master**. Простой пример реализации этого сценария вы только что видели — команда **git checkout -b [ветка] [имя удаленного сервера]/[ветка]**. Это достаточно распространенная операция, для которой в Git существует параметр **--track**:

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

Если вы хотите, чтобы имя локальной ветки отличалось от имени удаленной ветки, используйте первую версию команды с указанием нового имени:

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

Теперь локальная ветка **sf** поддерживает автоматический обмен данными с удаленной веткой **origin/serverfix**.

Если вы хотите сопоставить только что скачанной удаленной ветке существующую локальную ветку или поменять удаленную ветку, за которой вы следите, используйте параметр **-u** или **--set-upstream-to** команды **git branch**.

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

СОКРАЩЕННОЕ ОБОЗНАЧЕНИЕ

Для обращения к существующей ветке наблюдения есть сокращенные формы **@{upstream}** или **@{u}**. К примеру, если из ветки **master** вы следите за веткой **origin/master**, для краткости можно писать **git merge @{u}** вместо **git merge origin/master**.

Получить список веток наблюдения позволяет параметр **-vv** команды **git branch**. В результате выводится перечень локальных веток с дополнительной информацией, касающейся того, за чем следит каждая ветка и на сколько она опережает соответствующую ветку на удаленном сервере или отстает от нее:

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

Здесь мы видим, что ветка **iss53** связана с веткой **origin/iss53** и при этом запись «ahead 2» означает наличие двух локальных коммитов, пока не отправленных на сервер. Наша ветка **master** связана с веткой **origin/master** и содержит актуальную информацию. А вот ветка **serverfix**, связанная с веткой **server-fix-good** на сервере **teamone**, помечена как «ahead 3, behind 1», то есть на сервере существует коммит, который в нее пока не слит, а на ней присутствуют три коммита, пока не отправленные на сервер. Наконец, мы видим, что ветка **testing** не связана ни с одной удаленной веткой.

Отметим, что все цифры представляют собой показатели, зафиксированные в момент последнего скачивания данных с каждого сервера. Данная команда не обращается к серверам, а просто сообщает локальные данные из кэша. Для получения актуальной информации о количестве новых коммитов на локальных и удаленных ветках следует извлечь данные со всех удаленных серверов и только затем воспользоваться этой командой. Это можно сделать так: **\$ git fetch --all; git branch -vv**.

Получение данных с последующим слиянием

Хотя команда `git fetch` забирает с сервера всю пока отсутствующую у вас новую информацию, на состояние рабочей папки она никак не влияет. Она всего лишь предоставляет данные, которые можно подвергнуть слиянию. Но существует и команда `git pull`, по сути, представляющая собой команду `git fetch`, за которой немедленно следует команда `git merge`. При наличии ветки наблюдения, настроенной, как показано в предыдущем разделе, вручную или в момент ее генерации командой `clone` или `checkout`, команда `git pull` будет обращаться к серверу и ветке, за которой наблюдает ваша текущая ветка, скачивать данные с этого сервера и пытаться слить их с этой удаленной веткой.

В общем случае лучше отдельно пользоваться командами `fetch` и `merge`, в явном виде указывая подлежащую слиянию информацию, так как автоматизм команды `git pull` может привести к путанице.

Ликвидация веток с удаленного сервера

Предположим, работа с удаленной веткой закончилась, и программный компонент, над которым вы с коллегами трудились, слит в удаленную ветку `master` (или в ту ветку, где вы храните стабильный вариант кода). Теперь можно избавиться от удаленной ветки, добавив команде `git push` параметр `--delete`. А вот как выглядит удаление ветки `serverfix` с сервера:

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

По сути, эта команда просто удаляет с сервера указатель. Как правило, Git-сервер хранит данные, пока в дело не вступит сборщик мусора, поэтому, в случае ошибочного удаления, информацию зачастую легко восстановить.

Перемещение данных

В Git есть два основных способа включения изменений из одной ветки в другую: слияние (`merge`) и перемещение (`rebase`). В этом разделе мы рассмотрим процесс перемещения, детали его реализации, достоинства и случаи, когда его стоит и не стоит использовать.

Основы перемещения данных

Вспомним пример из раздела, посвященного слиянию. Мы разделили работу на две части и фиксировали изменения в двух разных ветках (рис. 3.27).

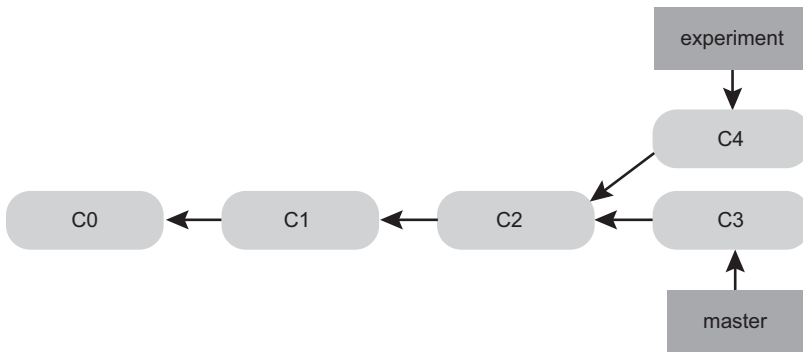


Рис. 3.27. Простой случай разделения истории

Простейшим средством объединения этих веток, как мы уже выяснили, является команда **merge** (рис. 3.28). Она в три шага выполняет слияние последних снимков состояния из разных веток (C3 и C4) и последнего общего предка этих состояний (C2), генерируя при этом новый снимок состояния (и коммит).

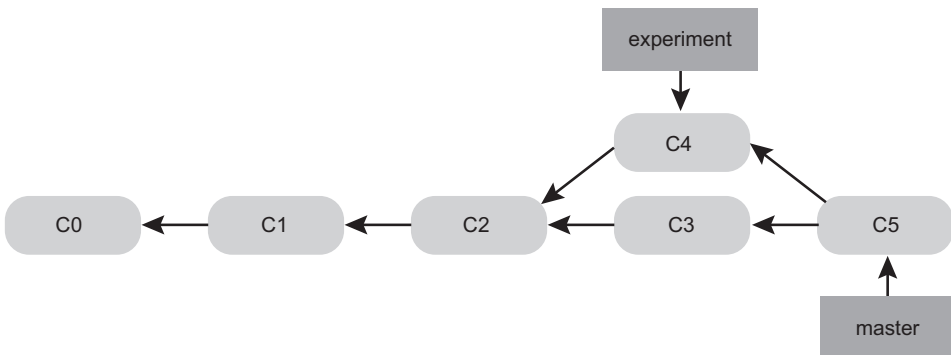


Рис. 3.28. Слияние для объединения разделившихся историй разработки

Но существует и другой способ: можно взять фрагмент изменений, появившийся в коммите C4, и применить его поверх коммита C3. В Git эта операция называется *перемещением* (rebasing). Команда **rebase** позволяет взять изменения, зафиксированные в одной ветке, и повторить их в другой.

В рассматриваемом примере это будет выглядеть так:

```
$ 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: ищется общий предок двух веток (текущей ветки и ветки, в которую вы выполняете перемещение), вычисляется разница, вносимая

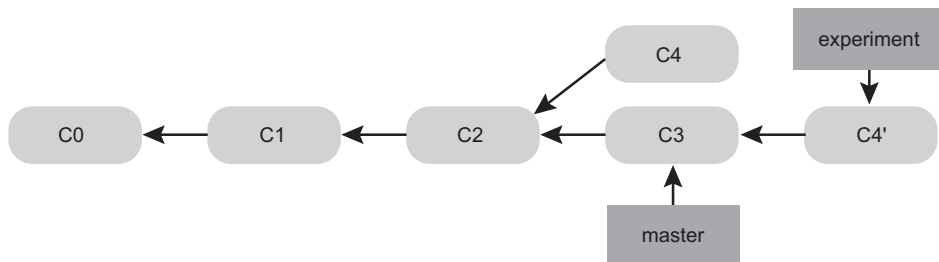


Рис. 3.29. Перемещение изменений, внесенных в коммите C3, в коммит C4

На этом этапе можно вернуться в ветку **master** и выполнить слияние перемотки (рис. 3.30).

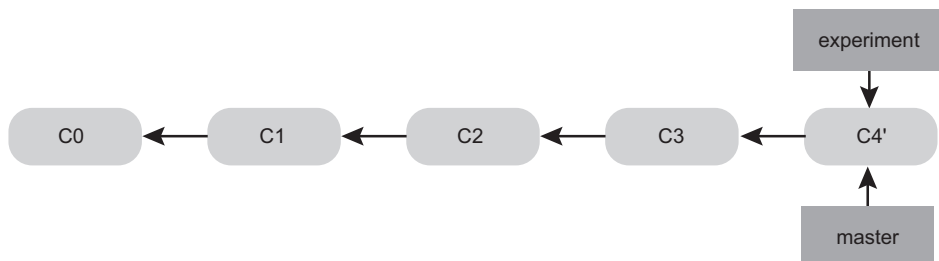


Рис. 3.30. Перемотка ветки master

Теперь снимок состояния, на который указывает коммит С3, полностью совпадает со снимком состояния, на который указывал коммит С5 в примере со слиянием. Разницы в конечном результате слияния нет, но в случае перемещения мы получаем более аккуратную историю. В журнале регистрации перемещенной ветки вы увидите линейное развитие событий. Все будет выглядеть так, как будто работа выполнялась последовательно, в то время как она велась параллельно.

Часто это делается, чтобы убедиться в четкости фиксируемых вами состояний на удаленной ветке, например в случае чужого проекта, в который вы хотите внести вклад. В такой ситуации оптимально работать в отдельной ветке, а затем, когда вы почувствуете, что готовы добавить свои исправления к основной версии проекта, переместить данные в ветку **origin/master**. Это избавит владельца проекта от необходимости заниматься интеграцией — достаточно прибегнуть к перемотке или просто применить добавленные изменения.

Обратите внимание, что в обоих случаях — последнего перемещенного коммита и итогового коммита слияния — коммиты указывают на один и тот же снимок

состояния, разной оказывается только история. При перемещении изменения из одной линии разработки применяются к другой в порядке их появления, в то время как при слиянии берутся конечные точки двух веток и соединяются друг с другом.

Более интересные варианты перемещений

Можно сделать так, чтобы воспроизведение результатов перемещения начиналось не с ветки, куда они были перемещены. В качестве примера рассмотрим историю разработки с рис. 3.31. Для добавления к проекту функциональности на стороне сервера была создана тематическая ветка (**server**), состояние которой в какой-то момент было зафиксировано. После этого была создана еще одна ветка (**client**) для внесения изменений в клиентскую часть, и в ней несколько раз выполнялась фиксация состояний. Наконец, произошло возвращение в ветку **server**, где также создано несколько коммитов.

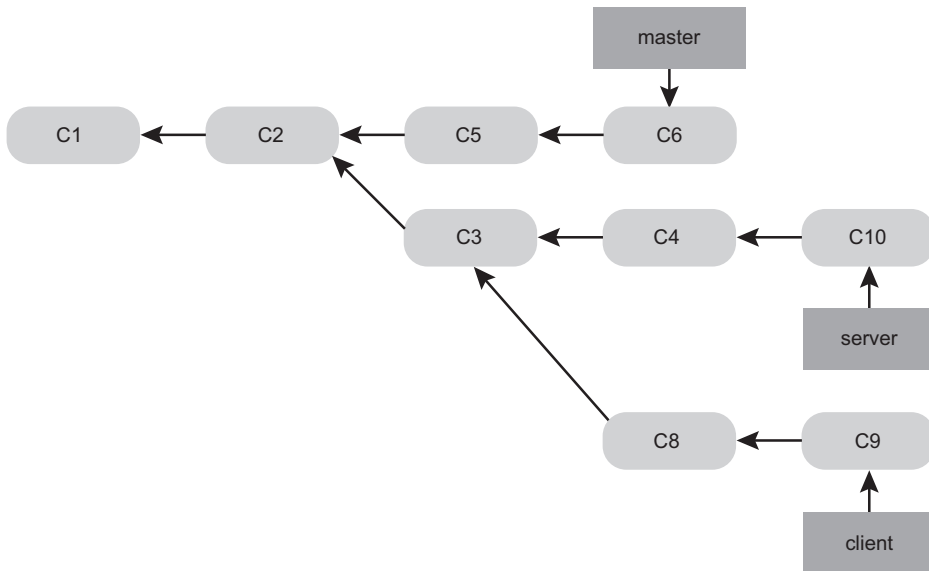


Рис. 3.31. История с тематической веткой, отходящей от другой тематической ветки

Предположим, вы хотите внести изменения клиентской части в окончательную версию кода, оставив изменения серверной части для дальнейшего тестирования. Взять изменения клиентской части (коммиты C8 и C9), не связанные с изменениями на серверной стороне, и воспроизвести их в ветке **master** позволяет команда **git rebase** с параметром **--onto**:

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

По сути, она приказывает «перейти в ветку `client`, найти исправления от общего предка веток `client` и `server` и повторить их в ветке `master`». Выглядит сложно, но дает интересный результат (рис. 3.32).

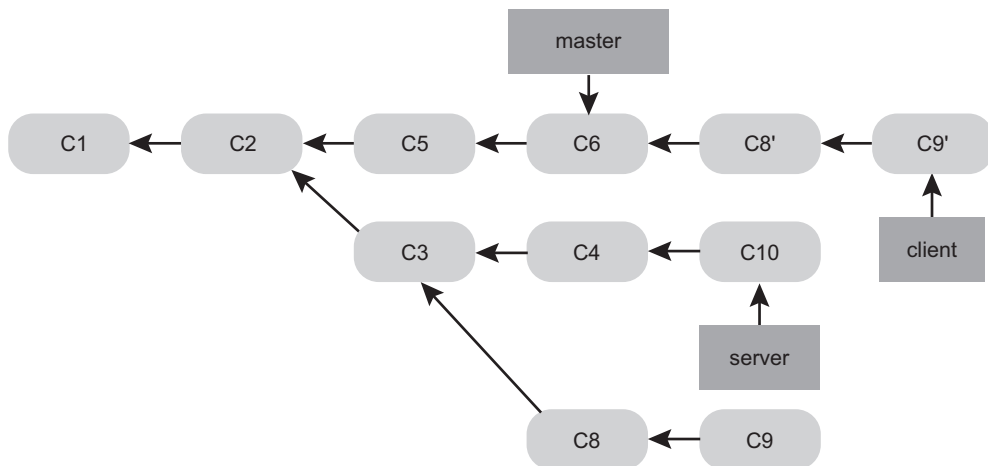


Рис. 3.32. Перемещение тематической ветки, начатой от другой тематической ветки

Теперь можно выполнить перемотку ветки `master` (рис. 3.33):

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

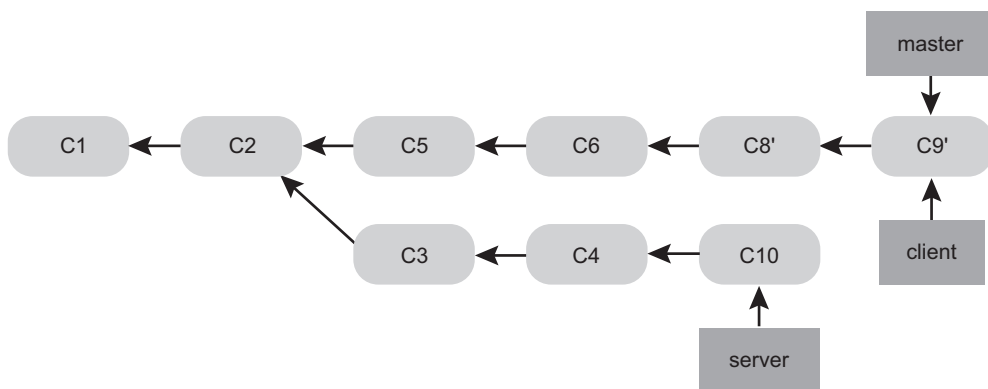


Рис. 3.33. Слияние перемотки ветки `master` для добавления изменений из ветки `client`

Предположим, вы решили добавить в основной код также работу из ветки `server`. Переместить эту ветку в ветку `master`, вне зависимости от того, в какой ветке вы сейчас находитесь, позволяет команда `git rebase [основная ветка]`

[тематическая ветка]. Она переключает вас на тематическую ветку (в данном случае — на ветку **server**) и воспроизводит ее содержимое в основной ветке (**master**):

```
$ git rebase master server
```

Это добавляет наработки из ветки **server** к наработкам в ветке **master** (рис. 3.34).

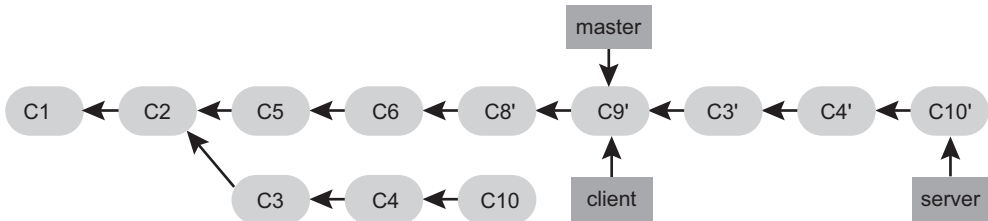


Рис. 3.34. Перемещение ветки **server** в конец ветки **master**

После этого можно выполнить перемотку основной ветки (**master**):

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

Ветки **client** и **server** теперь можно удалить, так как все их содержимое встроено в основную код и они больше не нужны (рис. 3.35).

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

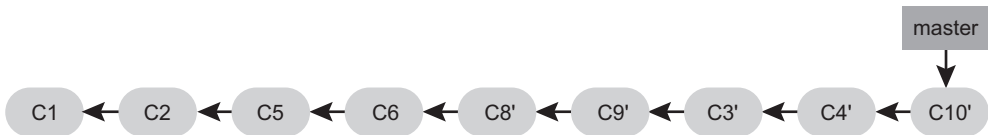


Рис. 3.35. Окончательный вид истории коммитов

Риски, связанные с перемещением

К сожалению, такая удобная операция, как перемещение, не лишена подводных камней. Следует знать о недопустимости перемещения коммитов, существующих вне вашего репозитория.

Пока вы следуете этому правилу, все хорошо. Нарушите его и на вас обрушится всеобщее осуждение.

В процессе перемещения вы прекращаете работу с существующими коммитами и создаете новые, которые в целом аналогичны старым, но кое в чем отличаются. Если вы выкладываете зафиксированные результаты своего труда в публичный доступ,

а кто-то скачивает их и использует как основу для своей работы, при переписывании коммитов командой `git rebase` и повторной отправке их на сервер вашим коллегам придется заново выполнять слияние. В результате при попытке инкорпорировать их работу в свою возникнет путаница.

Продemonстрируем, каким образом перемещение выложенных в общий доступ наработок становится источником проблем. Предположим, вы клонировали репозиторий с центрального сервера и выполнили с ним некие действия (рис. 3.36).

Кто-то из ваших коллег также делает работу, производит слияние и отправляет результат на центральный сервер. Вы извлекаете эти данные и присоединяете к своей работе новую удаленную ветку (рис. 3.37).

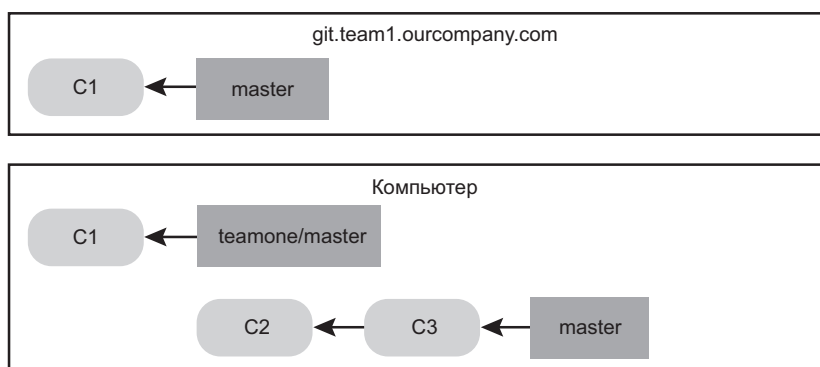


Рис. 3.36. Клонирование репозитория и выполнение на его основе какой-то работы

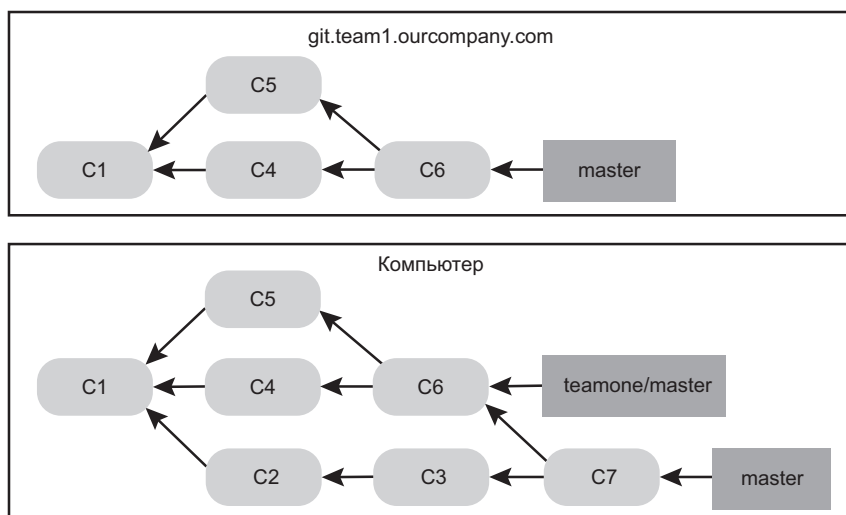


Рис. 3.37. Извлечение дополнительных коммитов и добавление их к своей работе

Но человек, выложивший подвергшийся слиянию коммит, решает все переиграть и выполнить вместо слияния перемещение; он использует команду `git push -force`, чтобы переписать историю на сервере. А после этого вы скачиваете с этого сервера изменения, включающие новые коммиты (рис. 3.38).

В результате вы оба попадаете в затруднительное положение. Команда `git pull` создаст у вас слитый коммит, включающий оба варианта развития истории (рис. 3.39).

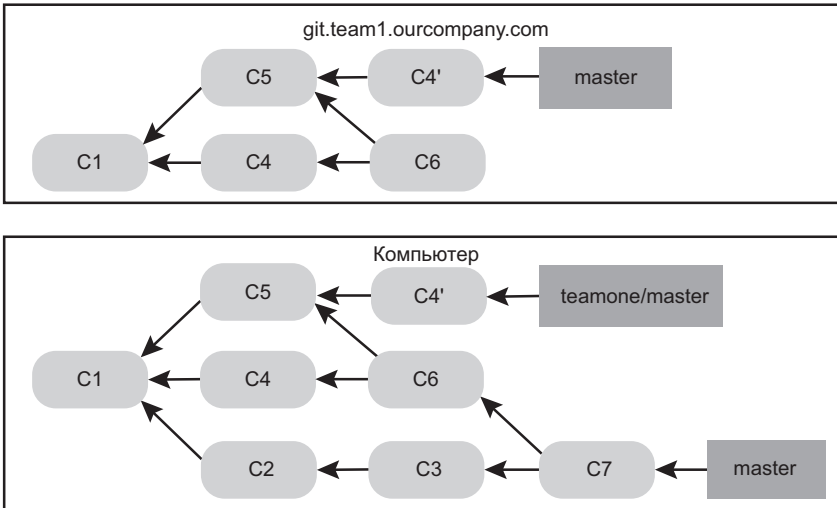


Рис. 3.38. Кто-то выложил перемещенные коммиты, убрав коммиты, на которых базировалась ваша работа

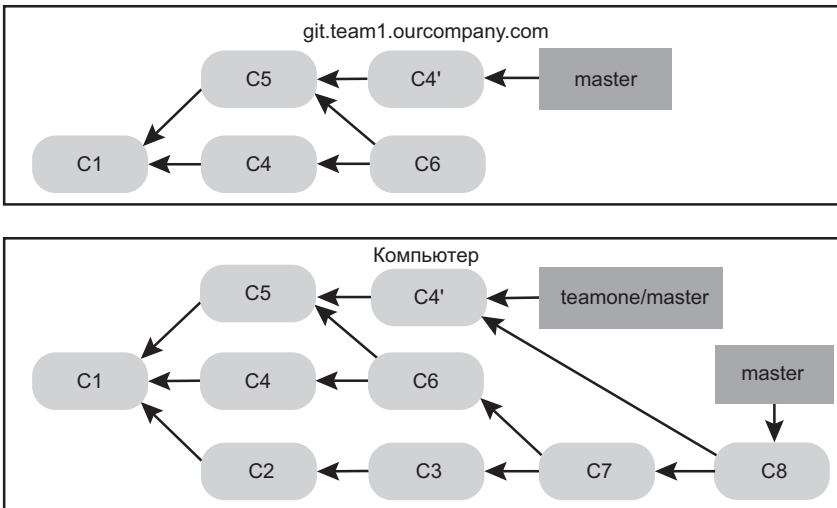


Рис. 3.39. Вы снова добавляете те же самые наработки в новый коммит слияния

Команда `git log` в случае подобной истории показывает два коммита одного и того же автора с одинаковой датой и одним сообщением слияния, что приводит к путанице. Более того, отправка этой истории обратно в общий доступ добавляет перемещенные коммиты в репозиторий центрального сервера, что еще сильнее запутывает пользователей. В подобном случае разумнее предположить, что другой разработчик не хотел видеть в истории коммиты C4 и C6; именно по этой причине он изначально прибег к перемещению.

Перемещение после перемещения

Для оказавшихся в подобной ситуации у Git есть палочка-выручалочка. Если кто-то из вашей рабочей группы волевым решением вносит изменения, меняющие данные, на которых вы базируете свою работу, первым делом следует определить, где ваши данные, а где данные, подвергшиеся перезаписи.

Оказывается, кроме контрольной суммы SHA коммита система Git вычисляет контрольную сумму, базирующуюся на внесенном вместе с коммитом исправлении. Это называется идентификатором исправления (`patch-id`).

В случае скачивания и добавления к вашей работе переопределенных данных с последующим перемещением их на новые коммиты от партнера Git зачастую успешно выделяет ваши уникальные данные и накладывает их на верхнюю часть новой ветки.

Если бы в ситуации с рис. 3.38 мы прибегли не к слиянию брошенных коммитов, служащих основой нашей работы, а воспользовались командой `git rebase teamone/master`, действия Git были бы следующими:

1. Определить работу, уникальную для нашей ветки (C2, C3, C4, C6, C7).
2. Определить, что не является коммитами слияния (C2, C3, C4).
3. Определить, что не нужно переписывать в целевую ветку (только C2 и C3, так как C4 — это то же самое исправление, что и C4').
4. Применить эти коммиты на ветке `teamone/master`.

И вместо картинки, показанной на рис. 3.39, мы получили бы результат, больше напоминающий рис. 3.40.

Это работает только в случае, когда коммиты C4 и C4' от вашего партнера содержат практически одинаковые исправления. В противном случае процедура перемещения не распознает их как дубликаты и добавит еще одно C4-образное исправление (которое, скорее всего, не удастся без проблем применить, так как эти изменения, по крайней мере до какой-то степени, уже присутствуют).

Упростить ситуацию можно, применив команду `git pull --rebase` вместо обычной команды `git pull`. Все то же самое можно также сделать вручную командой `git fetch`, за которой в данном случае следует команда `git rebase teamone/master`.

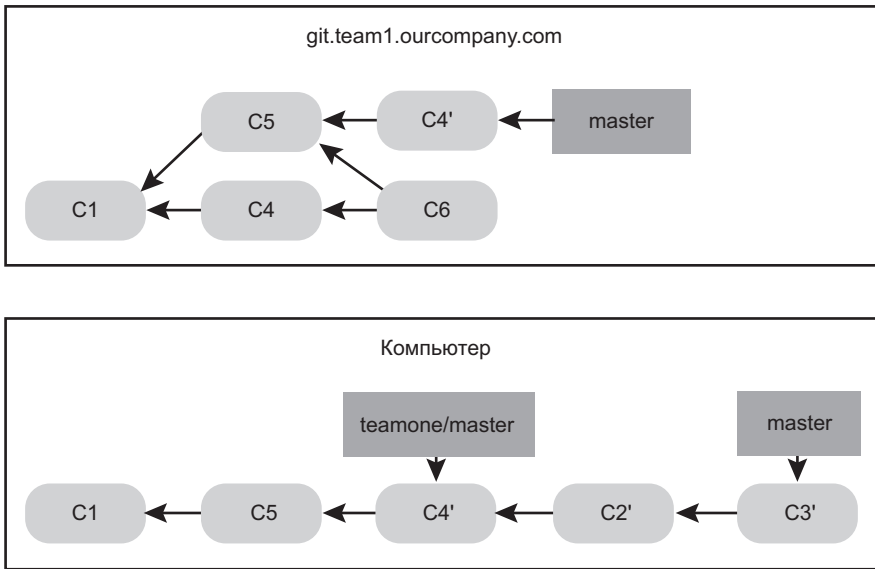


Рис. 3.40. Перемещение после перемещений, приведших к путанице

Если вы хотите, чтобы команда `git pull` по умолчанию запускалась с параметром `--rebase`, значению `pull.rebase config` присваивается нечто вроде `git config --global pull.rebase true`.

Если вы рассматриваете перемещение как операцию для наведения порядка и для работы с коммитами, которые пока не отправлены в общий доступ, все будет в порядке. Перемещение же коммитов, которые уже оказались в общем доступе и могли послужить основой для работы ваших коллег, может стать источником массы проблем.

В ситуации, когда вы или кто-то из ваших коллег все-таки считает необходимым перемещение коммитов, находящихся в общем доступе, доведите до сведения всех заинтересованных лиц, что им нужно воспользоваться командой `git pull --rebase`, чтобы хоть немного смягчить последствия данной операции.

Сравнение перемещения и слияния

Теперь, когда мы рассмотрели несколько процедур перемещения и слияния, может возникнуть вопрос, какая же из операций лучше. Перед тем как ответить на него, вернемся немного назад и поговорим о том, какой смысл несет история.

Существует точка зрения, что история коммитов вашего репозитория — это хроника работы с системой. Это документ, ценный сам по себе, и его подделка недопустима. Соответственно вмешательство в историю коммитов представляет собой

почти кошунство — вы фальсифицируете то, что происходило. Что делать в случае запутанной истории коммитов слияния? Ничего. Таким способом велась работа, и репозиторий должен сохранить эти подробности для будущих поколений.

Но есть и другая точка зрения. История коммитов — это то, что дает представление о структуре вашего проекта. Вы же не публикуете первый черновик книги, и даже руководство по поддержке выпускаемого вами программного обеспечения подвергается тщательному редактированию. Представители этого лагеря пользуются такими инструментами, как перемещение и фильтрация веток, чтобы представить историю в удобном для будущих читателей виде.

Наверное, вы уже сами понимаете, что на вопрос о превосходстве одной операции над другой однозначного ответа не существует. Система Git является мощным инструментом для реализации множества вещей, как строящих историю, так и меняющих ее, но в конечном счете все зависит от конкретной рабочей группы и конкретного проекта. Теперь, когда вы знакомы с деталями функционирования каждой команды, вы можете самостоятельно выбрать оптимальный в вашей ситуации инструмент.

В общем случае самый лучший результат обеспечивает перемещение локальных изменений, которые вы внесли, но пока не отправили в общий доступ, с целью приведения истории в порядок. Главное — никогда не перемещать данные, оказавшиеся за пределами вашего локального репозитория.

Заключение

Итак, мы рассмотрели основы ветвления и слияния в Git. Вы должны уверенно создавать новые ветки и переходить на них, без проблем выполнять переходы между ветками и слияние локальных веток. Кроме того, вы должны уметь выкладывать свои ветки в общий доступ, работать вместе с коллегами на таких ветках и выполнять перемещения своих веток до отправки их содержимого на всеобщее обозрение. В следующей главе мы поговорим о деталях поддержки вашего собственного сервера с Git-репозиторием.

4

Git на сервере

К этому моменту вы должны уметь решать большую часть рутинных задач, возникающих при работе с Git. Но для совместной работы требуется удаленный репозиторий. Разумеется, ничто не мешает вам добавлять изменения в чужие личные репозитории и скачивать оттуда обновления, но такая практика не приветствуется, так как легко приводит к путанице. К тому же вряд ли вам хотелось бы, чтобы у кого-то постороннего был доступ к вашему репозиторию и тогда, когда ваш компьютер выключен, а значит, требуется более надежный общественный репозиторий. Соответственно, для совместной работы лучше всего организовать промежуточный репозиторий, доступ на запись и на скачивание информации из которого будет у всех заинтересованных лиц.

Запуск сервера Git реализуется достаточно просто. Первым делом выбирается протокол для связи с этим сервером. Первая часть данной главы посвящена описанию доступных протоколов с их достоинствами и недостатками. Затем мы обсудим типичные конфигурации с применением этих протоколов и настройку сервера для работы с ними. Напоследок мы рассмотрим варианты хостинга, подходящие тем, кто готов разместить свой код на чужом сервере, так как не хочет возиться с настройкой и поддержанием собственного.

Если настройка собственного сервера вас не интересует, сразу переходите к концу главы, где обсуждаются варианты настройки учетных записей для существующего хостинга, а затем к следующей главе, посвященной тонкостям работы с распределенной системой контроля версий.

Удаленный репозиторий, как правило, представляет собой репозиторий без рабочей папки. Так как он предназначен исключительно для обмена данными, создавать рабочую копию на диске попросту незачем. Проще говоря, такой репозиторий содержит только каталог `.git` проекта и больше ничего.

Протоколы

Для передачи данных система Git пользуется четырьмя основными протоколами: локальным, HTTP, SSH (Secure Shell) и Git. В этом разделе мы поговорим об их особенностях и о том, в каких обстоятельствах имеет смысл применять каждый из них.

Локальный протокол

Основным является так называемый *локальный протокол* (local protocol), используемый в случаях, когда роль удаленного репозитория играет папка на этом же диске. Чаще всего он используется при наличии у всех членов группы доступа к общей файловой системе, например к NFS, или в менее распространенном случае, когда все авторизуются на одном компьютере. Второй вариант не рекомендуется, так как нахождение всех экземпляров репозитория на одной машине повышает риск катастрофической потери результатов.

Смонтированная общая файловая система дает возможность клонировать локальный репозиторий и обмениваться с ним данными. Чтобы осуществить клонирование или добавить такой репозиторий в качестве удаленного в существующий проект, следует указать путь к нему в виде адреса URL. К примеру, вот как выглядит клонирование локального репозитория:

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

Допустим и такой вариант команды:

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

Наличие приставки `file://` в начале URL-адреса несколько меняет поведение Git. Если указан только путь, Git пытается использовать жесткие ссылки или непосредственно копировать необходимые файлы. А приставка `file://` заставляет Git запустить процессы, обычно используемые для передачи данных по сети, что в целом менее эффективно. Но в этом случае вы получаете полную копию репозитория с дополнительными ссылками и оставшимися объектами — обычно после импорта из другой системы контроля версий (см. информацию об инструментах обслуживания в главе 10). Мы будем пользоваться стандартными путями, так как практически всегда это более быстрый способ.

Для добавления в существующий проект локального репозитория применяется следующая команда:

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

Теперь вы можете отправлять в этот репозиторий данные об изменениях и скачивать оттуда обновления точно так же, как это делалось по сети.

Преимущества

К преимуществам файлового репозитория относится его простота и возможность использовать существующие полномочия на доступ к файлам и сетевой доступ. При наличии доступной всей группе файловой системы настроить репозиторий очень легко. Достаточно поместить его основу в общедоступное место и указать права на чтение/запись, как в случае с любой совместно используемой папкой. Экспорт голой копии репозитория мы обсудим в разделе «Настройка Git на сервере».

Кроме того, это хорошая возможность быстро забрать данные из чужого рабочего репозитория. Если вы с коллегой работаете над одним проектом и он просит что-то проверить, применить команду `git pull /home/john/project` зачастую намного проще и быстрее, чем получать с удаленного сервера данные, которые он предварительно туда отправил.

Недостатки

Недостаток этого метода состоит в том, что общий доступ из разных мест настроить и получить сложнее, чем обычный сетевой доступ. К примеру, для отправки данных в репозиторий со своего домашнего компьютера вам нужно будет смонтировать удаленный диск, а это более медленная и сложная процедура, чем доступ по сети.

Также следует упомянуть, что использование папок общего доступа далеко не всегда является самым быстрым способом работы. Локальный репозиторий функционирует быстро только при наличии быстрого доступа к данным. Доступ к репозиторию в системе NFS часто медленнее доступа по протоколу SSH на том же самом сервере, при котором Git может работать с локальными дисками на каждой системе.

Протоколы семейства HTTP

Взаимодействия Git по протоколу HTTP осуществляются в двух режимах. До версии Git 1.6.6 существовал всего один способ, очень простой и в общем случае дающий доступ только на чтение. В версии 1.6.6 появился новый, более интеллектуальный протокол, позволяющий системе Git реализовать передачу данных примерно таким

же образом, как это происходит по протоколу SSH. В последние годы популярность этого нового протокола HTTP крайне возросла, так как пользователям проще с ним работать, а взаимодействия организованы намного аккуратнее. Более новая версия часто называется «интеллектуальным» протоколом HTTP, в то время как более старую называют «простым» протоколом HTTP. Первым делом мы рассмотрим «интеллектуальную» версию.

Интеллектуальный протокол HTTP

Интеллектуальный протокол HTTP функционирует во многом аналогично протоколу SSH или Git, но передает данные через стандартные порты HTTP/S и пользуется различными механизмами HTTP-аутентификации. То есть пользователям проще работать с ним, чем, например, с SSH, так как возможна аутентификация по имени пользователя/паролю, позволяющая избежать настройки ключей SSH.

В настоящее время это, пожалуй, самый популярный способ работы с Git, так как наряду с анонимным обслуживанием, как в случае протокола `git://`, он допускает аутентификацию и шифрование, аналогично протоколу SSH. И вместо набора URL-адресов для этого можно обойтись всего одним адресом. Если попытка отправить данные в репозиторий сопровождается требованием аутентификации (что, как правило, является нормой), сервер предлагает указать имя пользователя и пароль. То же самое происходит при получении доступа на чтение.

По сути, для таких служб, как GitHub, URL-адрес, используемый для просмотра репозитория в сети (например, `https://github.com/schacon/simplegit[]`), может применяться также и для клонирования при наличии у вас прав на запись.

Простой протокол HTTP

Если сервер не реагирует на интеллектуальный вариант HTTP, Git-клиент пользуется предшествующей, более простой версией протокола HTTP. С точки зрения этого протокола голый Git-репозиторий обслуживается как обычные файлы с веб-сервера. Прелесть простого протокола HTTP состоит в простоте его настройки. По сути, достаточно поместить голый Git-репозиторий в корневую папку с HTTP-документами и настроить хук, срабатывающий после обновления (`post-update hook`). После этого клонировать репозиторий сможет любой, кто имеет доступ к веб-серверу, где этот репозиторий расположен. Предоставить доступ на чтение по протоколу HTTP можно, например, следующим образом:

```
$ 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 все. Срабатывающий после обновления хук, по умолчанию устанавливаемый с Git, будет запускать команду (`git update-server-info`), обеспечивающую

корректное скачивание информации и клонирование по протоколу HTTP. Эта команда выполняется после каждой отправки изменений в репозиторий (скажем, по протоколу SSH); после этого другие пользователи могут выполнить клонирование командой:

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

В этом примере фигурировал путь `/var/www/htdocs`, общепринятый при настройке серверов Apache, но вы можете воспользоваться любым статическим веб-сервером — достаточно поместить в нужную папку голый репозиторий. Данные Git в таком случае обслуживаются как обычные статические файлы (подробно эта процедура рассматривается в главе 10).

В общем случае вы можете либо запустить сервер с доступом на чтение/запись по интеллектуальному протоколу HTTP, либо предоставить доступ к файлам только на чтение через обычный протокол. Вместе эти два варианта протокола практически никогда не используются.

Достоинства

Достоинства, о которых мы упомянем, относятся к интеллектуальной версии протокола. Наличие единого URL-адреса для всех типов доступа и приглашение на ввод личных данных, только когда это требуется для аутентификации, упрощают работу конечного пользователя. Аутентификация по имени пользователя и паролю также является огромным преимуществом перед протоколом SSH, ведь пользователю не приходится локально генерировать SSH-ключи и отправлять свои открытые ключи на сервер, чтобы получить возможность с ним взаимодействовать. Для менее опытных пользователей и пользователей систем, в которых протокол SSH не столь распространен, решающим фактором становится простота применения. К тому же, рассматриваемый протокол работает быстрее и рациональнее, чем SSH.

Кроме того, доступ на чтение репозитория можно организовать по протоколу HTTPS, дающему возможность шифровать трафик. Можно даже заставить клиентов пользоваться специальными подписанными SSL-сертификатами.

Еще одним плюсом является тот факт, что благодаря широкой распространенности протокола HTTP/S корпоративные фаерволы зачастую настроены на пропуск проходящего через эти порты трафика.

Недостатки

На некоторых серверах настроить Git по протоколу HTTP/S сложнее, чем по SSH. Но в остальном при работе с Git другие протоколы демонстрируют мало преимуществ перед интеллектуальным протоколом HTTP.

Если вы используете протокол HTTP для авторизованной отправки информации на сервер, постоянно вводить учетные данные сложнее, чем один раз воспользоваться

ключами в случае SSH. Однако существуют различные инструменты кэширования учетных данных, например Keychain в OSX и Credential Manager в Windows, облегчающие эту процедуру.

Протокол SSH

В случае самостоятельного хостинга для Git чаще всего используется протокол SSH. Дело в том, что доступ по SSH уже настроен на многих серверах, и даже если он отсутствует, это легко можно исправить. Кроме того, SSH — это протокол с проверкой подлинности, а благодаря тому, что он используется повсеместно, его легко настраивать и применять.

Для клонирования Git-репозитория по SSH указывается префикс `ssh://URL`, например:

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

Протокол можно не указывать — в этом случае Git по умолчанию предполагает SSH:

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

Кроме того, можно не указывать имя пользователя. В этом случае Git задействует данные пользователя, авторизовавшегося в системе.

Достоинства

У протокола SSH множество достоинств. Во-первых, он легко настраивается — SSH-демоны встречаются повсеместно, с ними умеют работать многие сетевые администраторы, они уже настроены во многих дистрибутивах операционных систем или же там есть инструменты для управления ими. Во-вторых, доступ по SSH безопасен — все данные передаются в зашифрованном виде и с проверкой подлинности. Наконец, подобно протоколам HTTP/S, Git и локальному, SSH делает данные перед передачей максимально компактными.

Недостатки

К сожалению, протокол SSH не дает возможности анонимного доступа к репозиторию. У каждого должен быть доступ к вашей машине по SSH, даже если речь идет только о чтении, что совершенно не подходит для проектов с открытым исходным кодом. Если работа осуществляется только в пределах вашей корпоративной сети, протокол SSH может оказаться единственным, с которым вам придется иметь дело. Если же требуется анонимный доступ на чтение материалов проекта, но при этом вы хотите использовать SSH, отправка данных в репозиторий настраивается по этому протоколу, а для их скачивания другими пользователями вам придется придумать что-то другое.

Протокол Git

Вместе с Git поставляется специальный демон, который слушает порт 9418. Этот демон предоставляет сервис, напоминающий протокол SSH, но без необходимости аутентификации. Для работы репозитория по протоколу Git необходимо создать файл `git-export-daemon-ok`, без него демон обслуживать репозиторий не будет. Но о средствах безопасности в данном случае речь не идет. Репозиторий Git может быть доступен для клонирования или всем, или никому. Это означает, что в общем случае по этому протоколу данные на сервер отправлять нельзя. Если открыть доступ на запись, то из-за отсутствия аутентификации кто угодно, зная URL-адрес вашего проекта, сможет вносить туда изменения. Но такое бывает крайне редко.

Достоинства

Зачастую протокол Git является самым быстрым из доступных сетевых протоколов передачи данных. В случае проекта с открытым доступом и мощным трафиком или большого проекта, в котором для получения доступа на чтение не требуется аутентификация, скорее всего, вам имеет смысл настроить Git-демон. Он использует тот же самый механизм передачи данных, что и протокол SSH, но без затрат на шифрование и аутентификацию.

Недостатки

Недостатком протокола Git является отсутствие аутентификации. Поэтому использовать его в качестве единственного средства доступа к проекту нежелательно. В общем случае он работает в паре с протоколом SSH или HTTPS, который обеспечивает нескольким разработчикам доступ на передачу данных (запись), в то время как нужды всех остальных обслуживает протокол `git://`, дающий доступ только на чтение. Также это один из самых сложно настраиваемых протоколов. Для него необходим собственный демон, требующий настройки службы `xinetd` или ей подобной, что далеко не всегда легко. Кроме того, фаервол должен предоставлять доступ к порту 9418, который не относится к стандартным портам, всегда открытым корпоративными фаерволами. Незвестные порты последними обычно блокируются.

Настройка Git на сервере

А теперь мы поговорим о настройке Git-службы, запускающей эти протоколы на вашем собственном сервере. Для начальной настройки любого Git-сервера требуется экспортировать существующий репозиторий в новый репозиторий, пока не имеющий рабочей папки. Обычно это достаточно простая процедура.

Для ее выполнения вам потребуется команда `clone` с параметром `--bare`. По существующему соглашению папки с голыми репозиториями имеют расширение `.git`:

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

ПРИМЕЧАНИЕ

В этом разделе демонстрируются команды и действия, необходимые для базовой, упрощенной установки на сервере с операционной системой Linux, хотя данные службы могут работать и на серверах с операционными системами Mac и Windows. Разумеется, настройка рабочего сервера внутри конкретной инфраструктуры потребует разных мер безопасности и своих инструментов в каждой системе, но в данном случае мы пытаемся сформировать у вас общее представление о порядке действий.

Копия данных из папки `Git` должна появиться в папке `my_project.git`.

Это примерно эквивалентно следующей записи:

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

Конфигурационный файл в этом случае будет содержать пару небольших отличий, но для наших целей они несущественны. Команда берет `Git`-репозиторий без его рабочей папки и создает папку конкретно для него.

Размещение на сервере голого репозитория

Теперь, когда у вас есть копия голого репозитория, ее нужно поместить на сервер и настроить протоколы. Мы предполагаем, что у вас уже есть полностью готовый к работе сервер `git.example.com` с доступом по протоколу `SSH`, а все свои репозитории вы хотите поместить в папку `/opt/git`. Предполагается, что такая папка на сервере уже существует, поэтому новый репозиторий можно получить простым копированием:

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

После этой операции другие пользователи, имеющие доступ на этот сервер по протоколу `SSH` и право читать содержимое папки `/opt/git`, могут клонировать ваш репозиторий командой:

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

Если у такого пользователя будут еще и права на запись в папку `/opt/git/my_project.git`, у него автоматически появится возможность отправлять свои файлы в репозиторий. Также автоматически `Git` корректно реализует права на запись для группы, если запустить команду `git init` с параметром `--shared`:

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

Вы видите, насколько просто взять Git-репозиторий, создать его голую версию и поместить на сервере, к которому у вас и ваших коллег есть доступ по протоколу SSH. После этого вы можете совместно работать над общим проектом.

Важно отметить, что это все, что требуется сделать для получения рабочего Git-сервера с доступом для нескольких человек, — добавьте на этот сервер учетные записи, умеющие работать с SSH, и поместите голый репозиторий в такое место, чтобы у всех заинтересованных лиц были права на чтение и запись. Больше никаких манипуляций не требуется.

Далее мы рассмотрим более сложные варианты настройки. К примеру, вы узнаете, как обойтись без создания учетных записей для каждого пользователя, как обеспечить общий доступ на чтение репозитория, как настроить пользовательские веб-интерфейсы, как работать с инструментом Gitis и многое другое. Но помните, что сервера с доступом по протоколу SSH и голого репозитория более чем достаточно, если речь идет о работе небольшой группы над закрытым проектом.

Простые настройки

Если вы работаете в небольшой фирме или только пытаетесь внедрить у себя Git, имея небольшое число разработчиков, процесс настройки будет достаточно простым. Одним из самых сложных аспектов настройки Git-сервера является управление пользователями. Ведь когда нужно дать некоторым пользователям права только на чтение из репозитория, а другим права на чтение и запись, задача усложняется.

Доступ по протоколу SSH

При наличии сервера, к которому у всех разработчиков уже есть доступ по протоколу SSH, проще всего расположить первый репозиторий там, так как в этом случае практически больше ничего не нужно делать (как вы могли убедиться в предыдущем разделе). Но если требуются более сложные права доступа к репозиториям, можно реализовать это через обычные права доступа файловой системы, используемой у вас на сервере.

Если же репозиторий будет располагаться на сервере, где не у всех членов рабочей группы, которым требуются права на запись, есть учетные записи, доступ по протоколу SSH придется настраивать вручную. Предполагается, что сервер с доступом по протоколу SSH у вас уже есть.

Существуют разные способы предоставления доступа членам группы. Во-первых, можно создать для каждого из них учетную запись. Это самое простое, что можно

сделать, но процедура может оказаться утомительной. Вряд ли вам захочется каждый раз выполнять команду `adduser` и придумывать временные пароли.

Во-вторых, можно создать на рабочей машине единого пользователя с именем `git`, попросить каждого, кому требуется доступ на запись, прислать свой открытый ключ SSH и добавить все присланные ключи в файл `~/.ssh/authorized_keys` нового пользователя `git`. Именно через этого пользователя все получают доступ к компьютеру. Это никак не повлияет на данные коммитов — учетная запись, из которой вы соединяетесь с сервером по протоколу SSH, не затрагивает записанные вами коммиты.

Еще можно сделать так, чтобы SSH-сервер опознавал вас через LDAP-сервер или другое централизованное средство аутентификации, которое у вас уже есть. При условии доступа каждого пользователя к консоли будет работать любой механизм аутентификации по протоколу SSH.

Создание открытого ключа SSH

Как уже говорилось, многие Git-серверы производят аутентификацию по открытым ключам SSH. Каждый пользователь вашей системы должен сгенерировать себе такой ключ, если он пока отсутствует. Этот процесс происходит одинаково во всех операционных системах. Первым делом нужно убедиться в отсутствии ключа. По умолчанию пользовательские SSH-ключи хранятся в папке `~/.ssh` каждого пользователя. Проверить, есть ли ключ, можно, зайдя в эту папку и отобразив ее содержимое:

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

Нам нужна пара файлов с такими именами, как, к примеру, `id_dsa` или `id_rsa`, и соответствующий файл с расширением `.pub`. Файл с расширением `.pub` содержит открытый ключ, а второй файл — закрытый ключ. Если таких файлов нет (а может быть, нет и папки `.ssh`), их можно создать в программе `ssh-keygen`, которая в операционных системах Linux/Mac поставляется вместе с пакетом SSH, а в Windows — вместе с пакетом MSysGit:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):
Created directory '/home/schacon/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
```



```
The key fingerprint is:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3  schacon@mylaptop.local
```

Первым делом вы указываете, где вы хотите сохранить ключ (`.ssh/id_rsa`), затем вас дважды просят ввести парольную фразу. Если вы не хотите вводить пароль, когда используете ключ, пропустите эту операцию.

Теперь ключ следует отослать администратору Git-сервера (предполагается, что настройки вашего SSH-сервера требуют открытых ключей). Для этого нужно скопировать содержимое файла `.pub` и отправить его по электронной почте. Открытые ключи выглядят примерно так:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAQEAK10UpkDhrfHY17SbrmTIpNLTKG9Tjom/BWDSU
GP1+nafz1HDTYW7hdI4yZ5ew18JH4JW9jbhUfrviQzM7x1ELEVf4h91FX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyB1WFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSLVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUF1jQJKprX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2so1d01QraT1MqVSsbx
NrRFi9wrf+m7Q==  schacon@mylaptop.local
```

Более подробную инструкцию по созданию SSH-ключей в разных операционных системах можно найти в руководстве GitHub по SSH-ключам, которое находится по адресу <https://help.github.com/articles/generating-ssh-keys>.

Настройка сервера

Рассмотрим настройку доступа по протоколу SSH на стороне сервера. В нашем примере для аутентификации пользователей будет применяться метод `authorized_keys`. Предполагается, что у вас установлен стандартный дистрибутив Linux, например Ubuntu. Первым делом нужно создать пользователя `git` и папку `.ssh` для этого пользователя:

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh
```

Затем для этого пользователя нужно добавить открытые SSH-ключи какого-либо разработчика в файл `authorized_keys`. Предположим, что вы получили по электронной почте несколько ключей и сохранили их во временных файлах. Еще раз напомним, как выглядят открытые ключи:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x41hJA0F3FR1rP6kYBRswj2aThGw6HXLm9/5zytK6Ztg3RPPK+4k
Yjh6541NysnEAZuXz0jTtYAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpGW1GYEIGS9Ez
Sdfd8ACCIicTDwbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv
07TCUSBdLQlgmVOFq1I2uPWQOkOWQAhuKEOmFjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5  gsg-keypair
```

Вы просто добавляете эти ключи в свой файл `authorized_keys`:

```
$ 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 можно настроить для них голый репозиторий, запустив команду `git init` с параметром `-bare`. В таком виде она инициализирует репозиторий без рабочей папки:

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

Теперь пользователи John, Josie и Jessica могут добавить этот репозиторий как удаленный и отправить туда первую версию проекта, над которым они работают. Имейте в виду, что каждый раз, когда вы хотите добавить проект, кто-то должен создавать на сервере голый репозиторий. Пусть сервер, на котором вы настроили пользователя `git` и репозиторий, называется `gitserver`. Если он находится во внутренней сети и вы настроили ссылающуюся на этот сервер DNS-запись для `gitserver`, командами можно пользоваться следующим образом:

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

Остальные могут без проблем клонировать это хранилище и отправлять туда изменения:

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

Этот способ позволяет быстро настроить Git-сервер с доступом на чтение и запись для небольшой группы разработчиков.

Следует заметить, что в настоящее время все эти пользователи также могут авторизоваться на сервере и получить доступ к консоли как пользователь `git`. Если вы хотите ограничить эту возможность, поменяйте в файле `passwd` командную оболочку на что-то другое.

Ограничить активность пользователя `git` только действиями, связанными с Git, позволяет поставляемая с этой системой ограниченная оболочка `git-shell`. Установив ее в качестве оболочки для авторизации пользователя `git`, вы ограничите доступ

этого пользователя к обычной оболочке. Для этого нужно указать **git-shell** вместо **bash** или **csh** как оболочку авторизации в файле **/etc/passwd**:

```
$ sudo vim /etc/passwd
```

В нижней части этого файла найдите примерно такую строку:

```
git:x:1000:1000::/home/git:/bin/sh
```

Замените **/bin/sh** на **/usr/bin/git-shell** (или запустите команду **which git-shell**, чтобы посмотреть, куда эта оболочка установлена). После редактирования строка примет вид:

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

Теперь пользователь Git сможет использовать SSH-соединение только для обмена данными с Git-репозиториями. При попытке же зайти на сервер через консоль он получит отказ в авторизации:

```
$ 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.
```

Сетевые Git-команды будут работать, но на сервер пользователи зайти не смогут. Как следует из вывода команды, вы можете изменить домашний каталог пользователя **git**, что слегка изменит поведение команды **git-shell**. К примеру, можно ограничить количество принимаемых сервером Git-команд или отредактировать сообщение, которое увидят пользователи при попытке зайти на сервер по протоколу SSH. Для получения дополнительной информации по настройке оболочки используйте команду **git help shell**.

Git-демон

Теперь рассмотрим процесс настройки демона, обслуживающего репозитории по протоколу Git. Это распространенный вариант доступа к Git-данным без аутентификации. Напоминаем, что в этом случае все обслуживаемое по данному протоколу общедоступно в сети.

Если сервер, на котором запущен демон, находится не за фаерволом, использовать его можно только для открытых проектов. Находящийся под защитой фаервола сервер может обеспечивать доступ на чтение большому числу людей или компьютеров (серверы непрерывной интеграции или сборки), если вы не хотите добавлять SSH-ключи для всех пользователей.

В любом случае настройка протокола Git осуществляется достаточно просто. По сути, достаточно этой команды:

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

Параметр `--reuseaddr` позволяет серверу перезагрузиться, не ожидая истечения времени ожидания старого подключения, а благодаря параметру `--base-path` пользователи могут клонировать проекты без указания полного пути — путь в конце команды сообщает демону Git, где находятся экспортируемые репозитории. Если вы используете фаервол, в нем должен быть открыт порт 9418 на компьютере, на котором вы все это настраиваете.

Автоматизировать запуск процесса в режиме демона можно разными способами, зависящими от операционной системы, с которой вы работаете. На машине с Ubuntu можно использовать сценарий Upstart:

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

В этот файл помещается такой сценарий:

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

По соображениям безопасности запускать этого демона желательно как пользователя с правами только на чтение репозитория. Для этого нужно создать нового пользователя `git-ro` и запустить из-под него демона. Для простоты мы запустим его из-под того же пользователя, из-под которого запущена ограниченная оболочка.

После перезапуска компьютера демон Git стартует автоматически. Автоматическим является и его перезапуск после завершения работы. Без перезагрузки его можно привести в активное состояние командой:

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

В других операционных системах можно воспользоваться сценарием `xinetd` системы инициализации `sysvinit` или чем-то еще, — главное, чтобы команду можно было запустить как демона и каким-то образом за ней присматривать.

Затем нужно указать, к каким репозиториям предоставляется доступ через Git-сервер без аутентификации. Для каждого репозитория при этом создается файл `git-daemon-export-ok`.

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

Наличие этого файла является для Git сигналом того, что с проектом можно работать без аутентификации.

Интеллектуальный протокол HTTP

Итак, мы организовали аутентифицированный доступ по протоколу SSH и доступ без аутентификации по протоколу `git://`, но существует и протокол, позволяющий объединить эти вещи. Настройка этого протокола, по сути, представляет собой добавление на сервер поставляемого вместе с Git CGI-сценария `git-http-backend`. Этот сценарий считывает путь и заголовки, пересылаемые в URL-адресах, которые передаются по протоколу HTTP командами `git fetch` и `git push`, и определяет, может ли клиент пользоваться этим протоколом для работы (все клиенты, начиная с версии 1.6.6, допускают эту возможность). Если выяснится, что клиент не в состоянии работать с более новой, интеллектуальной версией протокола, для него будет взята старая простая версия (то есть поддерживается обратная совместимость, обеспечивающая старым версиям клиента возможность чтения).

Рассмотрим основные этапы установки. В данном случае в качестве CGI-сервера будет фигурировать Apache. Читатели, у которых нет доступа к Apache, могут сделать на Linux-машине примерно следующее:

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

При этом будут активированы необходимые для корректной работы модули `mod_cgi`, `mod_alias` и `mod_env`.

Теперь внесем дополнения в конфигурацию Apache, так как нам требуется превратить сценарий `git-http-backend` в обработчик всей информации, приходящей по пути `/git` вашего веб-сервера.

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

Без переменной среды `GIT_HTTP_EXPORT_ALL` Git будет только отдавать не прошедшим аутентификацию клиентам репозитории, содержащие файл `git-daemon-export-ok`, как это делает Git-демон.

Затем нужно сделать так, чтобы сервер Apache разрешил запросы к данному пути:

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

Наконец, следует установить аутентификацию на запись, например при помощи вот такого блока `Auth`:

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

```
Require valid-user
</LocationMatch>
```

Для этого потребуется файл `.htaccess` с паролями всех имеющих доступ пользователей. Вот как выглядит добавление в этот файл пользователя `schacon`:

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

Существует множество способов аутентификации пользователей на сервере Apache, вам остается выбрать и реализовать какой-нибудь из них. Мы привели простейший пример. Мы почти уверены, что заодно вы захотите настроить работу по протоколу SSL для обеспечения шифрования данных.

Мы не будем слишком детально рассматривать особенности конфигурации серверов Apache, так как вы можете работать с другим сервером и иметь другие требования к аутентификации. Основная идея состоит в наличии у системы Git CGI-сценария `git-http-backend`, который берет на себя все процедуры согласования для отправки и приема данных по протоколу HTTP. Сам по себе этот сценарий не выполняет аутентификацию, но легко может контролироваться на уровне запускающего его веб-сервера. Его можно использовать практически на любом веб-сервере, поддерживающем CGI, так что вы можете выбрать наиболее удобный для вас вариант.

ПРИМЕЧАНИЕ

Дополнительные сведения о настройке аутентификации в Apache вы найдете в документации по адресу: <http://httpd.apache.org/docs/current/howto/auth.html>.

Интерфейс GitWeb

Итак, вы настроили для вашего проекта базовый доступ на чтение/запись и доступ только на чтение и, возможно, не отказались бы от простого веб-интерфейса. Система Git поставляется с CGI-сценарием GitWeb, который порой используется именно для этой цели (рис. 4.1).

В Git существует команда, позволяющая владельцам облегченных серверов, таких как `lighttpd` или `webrick`, установить временный экземпляр GitWeb, чтобы посмотреть, как он будет выглядеть в случае конкретного проекта. На машинах с операционной системой Linux сервер `lighttpd` зачастую уже установлен, так что достаточно выполнить команду `git instaweb` из папки вашего проекта. Операционная система Leopard на компьютерах Mac поставляется с предустановленным языком Ruby, поэтому, наверное, в этом случае лучше пользоваться сервером `webrick`. По умолчанию команда `instaweb` вызывает обработчик `lighttpd`, поэтому вам потребуется параметр `--httpd`.

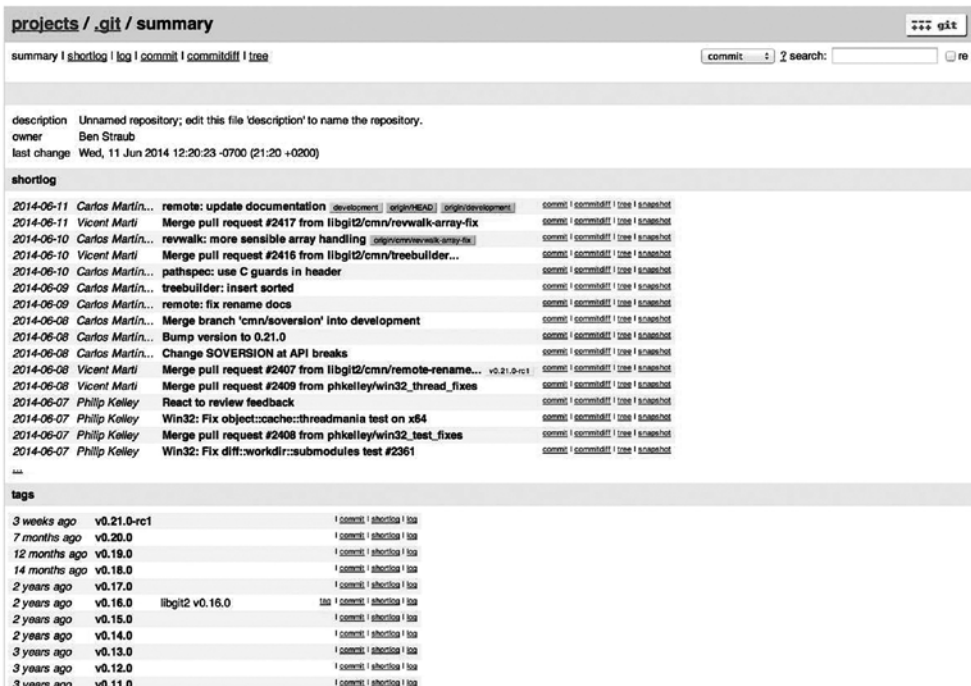


Рис. 4.1. Пользовательский веб-интерфейс на базе GitWeb

```
$ 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]
```

Запустится HTTPD-сервер на порту 1234, и автоматически откроется веб-браузер с нужной страницей. Как видите, все очень просто. Когда вы завершите работу, для остановки сервера нужно будет выполнить эту же команду с параметром `--stop`:

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

Если вашей группе или вашему проекту с открытым исходным кодом требуется постоянно работающий веб-интерфейс, установите этот CGI-сценарий на своем обычном веб-сервере. В некоторых дистрибутивах Linux имеется пакет `gitweb`, который можно установить с помощью программ `apt` или `yum`, — возможно, первым делом вы захотите испробовать этот способ. Мы же кратко рассмотрим процесс ручной установки GitWeb. Для начала следует получить исходный Git-код, с которым поставляется GitWeb, и сгенерировать CGI-сценарий для своей системы:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" prefix=/usr gitweb
SUBDIR gitweb
```

```
SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
GEN gitweb.cgi
GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

Напоминаем, что команде следует указать местоположение ваших Git-репозитория. Это делается при помощи переменной `GITWEB_PROJECTROOT`. Далее нужно сделать так, чтобы сервер Apache использовал для этого CGI-сценарий, для чего можно добавить директиву `VirtualHost`:

```
<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>
```

Еще раз напомним, что интерфейс GitWeb может быть установлен на любом веб-сервере, умеющем работать с CGI или Perl; если вы предпочитаете какой-то другой сервер, настройка не должна вызвать у вас трудностей. После этого вы сможете просматривать свои репозитории в сети по адресу `http://gitserver/`.

Приложение GitLab

Интерфейс GitWeb чрезмерно упрощен. В качестве более современного, полнофункционального Git-сервера можно взять одно из существующих решений с открытым исходным кодом. Наиболее популярным из них является приложение GitLab, поэтому именно его мы рассмотрим в качестве примера. Оно несколько сложнее интерфейса GitWeb и требует куда большей поддержки, но предлагает несоизмеримо более богатую функциональность.

Установка

Так как GitLab представляет собой веб-приложение на основе базы данных, установить его сложнее, чем другие Git-серверы. К счастью, для этого процесса существует подробная документация и он хорошо поддерживается.

Существуют разные методы установки GitLab. Если требуется быстрый результат, можно скачать с сайта <https://bitnami.com/stack/gitlab> образ виртуальной машины или

Пользователи

Пользователи представлены в GitLab в виде набора учетных записей. Эти записи не очень сложные и состоят по большей части из персональной информации, связанной с данными входа в систему. Каждой учетной записи соответствует собственное пространство имен, логически объединяющее принадлежащие конкретному пользователю проекты (рис. 4.4). Скажем, URL-адрес проекта **project** пользователя **jane** будет выглядеть так: `http://server/jane/project`.

Удалить учетную запись можно двумя способами. *Блокировка* пользователя не дает ему авторизоваться в экземпляре GitLab, но в соответствующем пространстве имен сохраняются все его данные, а подписанные его адресом электронной почты коммиты будут содержать ссылки на его профиль.

Уничтожение же пользователя удаляет все его следы из базы данных и файловой системы. Исчезают как все проекты и данные из его пространства имен, так и все принадлежащие ему группы. Разумеется, этот радикальный и необратимый способ применяется куда реже.

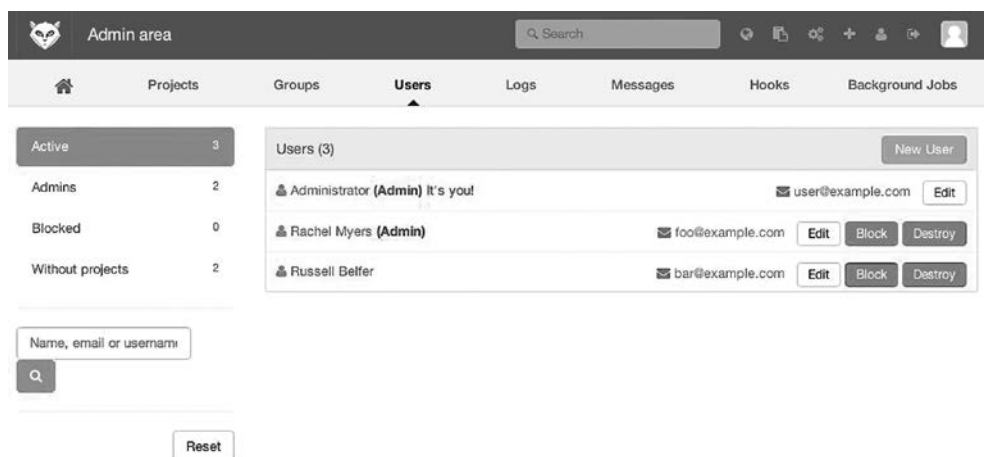


Рис. 4.4. Экран управления пользователями в GitLab

Группы

Группа в GitLab представляет собой набор проектов вместе с данными о том, как пользователи могут получить доступ к этим проектам (рис. 4.5). Каждой группе соответствует пространство имен проекта (аналогичное пространству имен

пользователей), то есть группа **training** с проектом **materials** будет доступна по адресу <http://server/training/materials>.

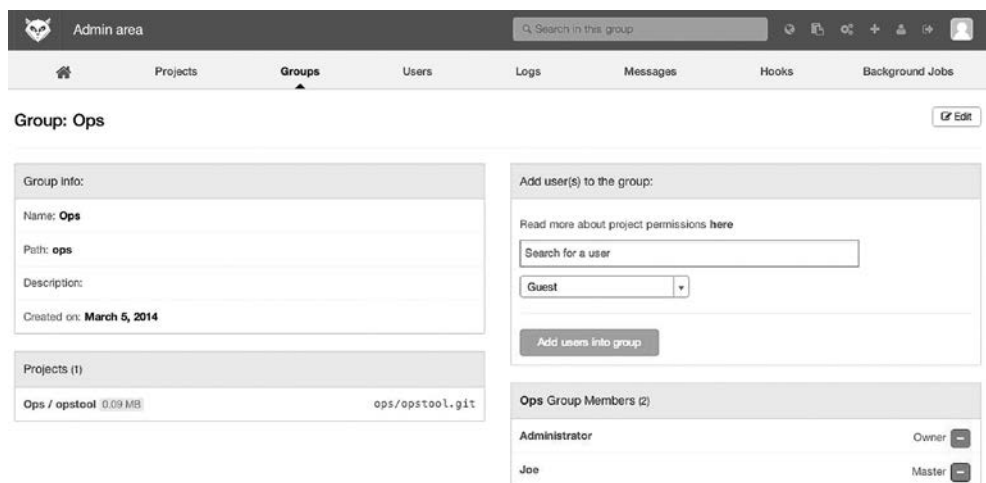


Рис. 4.5. Экран управления группами в GitLab

С группой связан набор пользователей, каждый из которых имеет свой уровень доступа к проектам группы и самой группе. Диапазон уровней варьируется от «гостя» (возможность задавать вопросы и принимать участие в чате) до «владельца» (полный контроль группы, ее членов и ее проектов). Количество уровней доступа слишком велико, чтобы перечислять их на страницах нашей книги, но на административном экране GitLab есть ссылка на их детальное описание.

Проекты

Проект в GitLab с некоторыми допущениями можно сопоставить с одним Git-репозиторием. Каждый проект является частью одного пространства имен, группового или пользовательского. Если проект принадлежит пользователю, доступ к нему контролирует владелец; в групповых проектах существуют допуски для разных категорий пользователей.

Кроме того, у каждого проекта есть свой уровень видимости, определяющий, у кого есть доступ к чтению страниц проекта или репозитория. В случае закрытого (private) проекта владелец в явном виде предоставляет доступ определенным пользователям. Внутренний (internal) проект виден любому авторизованному пользователю, а открытый (public) проект могут просматривать все желающие. Это относится как к информации, получаемой посредством команды **git fetch**, так и к просмотру проекта через веб-интерфейс.

Хуки

Поддержка хуков в GitLab реализована на уровне как проектов, так и системы в целом. В обоих случаях при возникновении заданного события GitLab-сервер передает по протоколу HTTP запрос POST с описанием в формате JSON. Это замечательный способ объединить ваши Git-репозитории и экземпляр GitLab с остальной автоматизированной структурой разработки, например серверами непрерывной интеграции, комнатами чатов или инструментами развертывания.

Базовое применение

Установив GitLab, первым делом создают новый проект. Для этого нужно щелкнуть на кнопке со значком + на панели инструментов приложения. Вам будет предложено указать имя проекта, пространство имен, которому он должен принадлежать, и уровень его видимости. Большую часть этих данных в будущем при желании можно изменить через интерфейс настроек. Щелкните на кнопке **Create Project**, чтобы создать проект.

Существующие проекты вы, скорее всего, захотите соединить с локальным Git-репозиторием. Все проекты доступны по протоколу HTTPS или SSH. Соответственно оба этих протокола могут применяться для настройки удаленного Git-репозитория. Адреса URL можно увидеть в верхней части домашней страницы проекта. Вот команда, которая для существующего локального репозитория создает удаленный репозиторий с именем **gitlab** на указанном вами сервере:

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

Если локальная копия репозитория отсутствует, ее можно легко создать:

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

Пользовательский веб-интерфейс обеспечивает доступ к разным представлениям репозитория. На домашней странице каждого проекта отображается информация о последних действиях, а ссылки в ее верхней части ведут к файлам проекта и журналу регистрации коммитов.

Совместная работа

Самым простым способом совместной работы над GitLab-проектом является предоставление другому пользователю непосредственного доступа к Git-репозиторию. Добавление пользователей происходит в разделе **Members** окна настроек проекта, где указывается уровень доступа каждого нового пользователя (уровни доступа

кратко обсуждались в разделе «Группы»). Уровень разработчика (developer) и выше позволяет отправлять коммиты и ветки непосредственно в репозиторий.

Менее плотным способом совместной работы являются запросы на слияние. Эта функциональность позволяет любому пользователю, который видит проект, контролируемым образом вносить в него свой вклад. Пользователи с непосредственным доступом могут создавать ветки, присылать коммиты для этих веток и открывать запросы на слияние из своих веток в ветку **master** или любую другую. Пользователи, не имеющие доступа на запись в репозиторий, могут создать собственную копию репозитория, отправить туда коммиты и открыть запрос на слияние их копии с основным проектом. Эта модель позволяет владельцу репозитория, принимая помощь от ненадежных пользователей, полностью контролировать, что именно и когда попадает в репозиторий.

Запросы на слияние и проблемы (issues) являются основой длительных обсуждений в GitLab. Каждый запрос на слияние порождает как построчное обсуждение предлагаемых изменений (чему способствует облегченное рецензирование кода), так и дискуссию в целом. Оба варианта могут быть как назначены пользователям, так и превращены в промежуточные этапы (milestones).

В этом разделе мы рассмотрели в основном аспекты GitLab, имеющие отношение к Git, но это хорошо продуманная система с массой функциональных возможностей, которые могут пригодиться при совместной работе. Сюда относятся, например, вики-страницы проектов, дискуссионные «стены» и инструменты поддержки системы.

Весомое преимущество GitLab состоит в том, что когда сервер настроен и работает, необходимость вносить изменения в конфигурационный файл или обращаться к нему по протоколу SSH возникает крайне редко; большая часть действий, связанных с администрированием и использованием этого приложения, выполняется через встроенный в браузер интерфейс.

Сторонний хостинг

Для тех, кто не хочет заниматься настройкой собственного Git-сервера, существуют несколько вариантов размещения проектов на специальных сайтах. У такого подхода есть ряд преимуществ: настройка и запуск проекта на таком сайте обычно не требуют много времени и вам не приходится заниматься поддержкой сервера и наблюдением за ним. Даже при наличии собственного внутреннего сервера в случае проектов с открытым исходным кодом сторонний хостинг порой оказывается более предпочтительным — в таком виде сообществу проще оказывать вам помощь.

В наши дни существует огромное количество вариантов хостинга с различными достоинствами и недостатками. Их список доступен по адресу <https://git.wiki.kernel.org/index.php/GitHosting>.

Работа с крупнейшим на сегодняшний день хостингом GitHub подробно рассмотрена в главе 6, так как вам может потребоваться взаимодействие с расположенными на нем проектами. Но есть и множество других вариантов для тех, кто не хочет заниматься настройкой собственного Git-сервера.

Заключение

Существуют различные варианты устройства удаленного Git-репозитория, позволяющего принимать участие в совместных проектах и делиться собственными наработками.

Собственный сервер дает вам полный контроль над происходящим; кроме того, его можно защитить своим фаерволом, но вам потребуется изрядное количество времени на его настройку и поддержку. Размещение данных с помощью стороннего хостинга упрощает эти операции, но код в этом случае будет храниться на чужом сервере, а далеко не все организации допускают подобное.

В данном случае вам остается только выбрать решение, устраивающее одновременно и вас, и фирму, в которой вы работаете.

5

Распределенная система Git

Итак, вы обзавелись удаленным Git-репозиторием, посредством которого все разработчики могут обмениваться своим кодом, и освоили основные Git-команды для локальной работы. Значит, пришло время познакомиться с распределенными рабочими процессами, которые вам предлагает Git.

В этой главе вы научитесь работать с Git в распределенной среде в качестве как простого разработчика, так и системного интегратора. То есть вы узнаете, как вносить вклад в проекты, максимально упрощая эту процедуру для себя и для пользователя, отвечающего за поддержку проекта, а также получите навыки сопровождения проектов, над которыми работает большой коллектив.

Распределенные рабочие процессы

В отличие от централизованных систем контроля версий (CVCS), Git благодаря своему распределенному характеру допускает большую гибкость при совместной работе над проектами. В централизованных системах всех разработчиков можно представить в виде узлов сети, каждый из которых делает свою часть работы на центральном концентраторе. В Git же каждый разработчик может выступать как в роли узла, так и в роли концентратора. То есть он может вносить код в чужие репозитории и поддерживать открытый репозиторий. Такой подход дает рабочей группе широкие возможности. Некоторые распространенные варианты работы над проектами, доступные благодаря гибкости системы Git, рассмотрены в этой

главе. Мы постараемся учесть как сильные стороны, так и возможные недостатки каждого варианта, в результате вы сможете выбрать для себя наиболее подходящий или скомбинировать функциональные возможности сразу нескольких подходов.

Централизованная работа

В централизованных системах, как правило, существует всего одна модель совместной работы. Один центральный концентратор (репозиторий) принимает код, а все остальные синхронизируют с ним свои действия. Все разработчики являются узлами — клиентами этого концентратора — и сверяются только с ним (рис. 5.1).

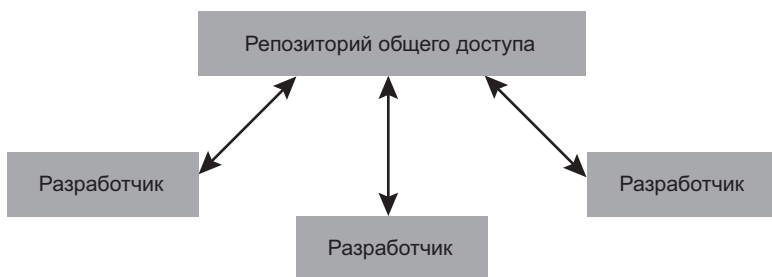


Рис. 5.1. Централизованный рабочий процесс

Если два разработчика клонируют концентратор и каждый из них вносит изменения, первый может без проблем отправить результаты своего труда назад. А вот второй, чтобы избежать перезаписи внесенных коллегой изменений, перед этим должен выполнить слияние с его наработками. В Git этот принцип справедлив в той же степени, что и в Subversion (или любой другой централизованной системе контроля версий), — модель превосходно функционирует.

Если в своей фирме или группе вы уже привыкли к централизованному рабочему процессу, можете работать с Git в аналогичной манере. Настройте один репозиторий и предоставьте всем членам группы доступ на запись; система Git сама не позволит пользователям переписывать наработки друг друга. Представим, что Джон и Джессика начинают работать одновременно. Джон вносит изменение и отправляет его на сервер. На попытку Джессики сделать то же самое сервер ответит отказом. Он сообщит, что Джессика пытается добавить изменения, для которых невозможно выполнить перемотку, поэтому сначала ей нужно скачать данные с сервера, осуществить слияние и только после этого отправлять результаты своего труда. Такой вариант многим по душе, потому что он знаком и привычен.

И это касается не только небольших групп. Модель ветвлений в Git позволяет сотням разработчиков успешно трудиться в рамках одного проекта, одновременно работая над десятками веток.

Диспетчер интеграции

Так как количество удаленных репозиторий в Git неограниченно, рабочий процесс можно организовать так, чтобы у каждого разработчика было право на запись в собственный открытый репозиторий и право на чтение чужих репозиторий. Этот сценарий зачастую подразумевает наличие общего репозитория, представляющего собой «официальный» проект. Чтобы принять участие в работе над таким проектом, нужно создать его открытую копию, куда и будут отправляться все изменения. Позже пользователю, отвечающему за поддержку основного проекта, отправляется запрос на внесение ваших изменений. После этого ваш репозиторий могут добавить как удаленный, протестировать ваши изменения локально, слить их со своей веткой и добавить в свой репозиторий (рис. 5.2). Вот общая последовательность действий:

1. Владелец проекта выкладывает файлы в открытый репозиторий.
2. Участники проекта клонируют этот репозиторий и вносят свои изменения.
3. Каждый участник выкладывает изменения в свой собственный открытый репозиторий.
4. Каждый участник отправляет владельцу письмо с просьбой включить его изменения в общий проект.
5. Владелец добавляет репозиторий каждого участника как удаленный и выполняет локальное слияние.
6. Владелец отправляет слитые изменения в основной репозиторий.

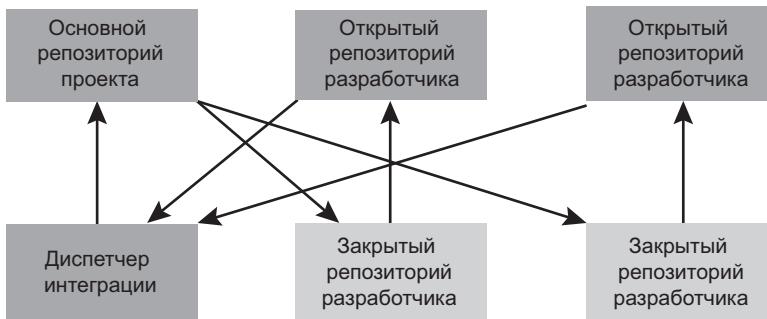


Рис. 5.2. Рабочий процесс с диспетчером интеграции

Рабочий процесс такого типа широко применяется с такими инструментами, как GitHub или GitLab, работающими по принципу концентратора и позволяющими легко выполнить ветвление проекта и выложить изменения на всеобщее обозрение в собственную ветвь. Одним из основных преимуществ такого подхода является

возможность продолжать работу, в то время как владелец основного репозитория может в любой момент включить туда ваши изменения. Участники проекта не должны ждать момента, когда их изменения будут встроены в общий проект, — каждый может работать в собственном темпе.

Диктатор и помощники

Существует еще одна разновидность рабочего процесса с набором репозитория. Как правило, она применяется в крупных проектах с сотнями участников, таких как работа над ядром Linux. За отдельные части репозитория отвечают несколько диспетчеров интеграции; этих людей называют помощниками. Над ними главенствует один диспетчер интеграции, которого называют диктатором. Именно он владеет эталонным репозиторием, откуда все остальные участники проекта должны скачивать изменения (рис. 5.3). Вот описание рабочего процесса:

1. Обычные разработчики трудятся в тематических ветках и перемещают результаты своей работы на вершину ветки **master**. Ветка **master** принадлежит диктатору.
2. Помощники сливают тематические ветки разработчиков в свои ветки **master**.
3. Диктатор сливает содержимое веток **master** своих помощников в свою ветку **master**.
4. Диктатор отправляет свою ветку **master** в эталонный репозиторий, что дает остальным разработчикам возможность перемещать туда свои данные.

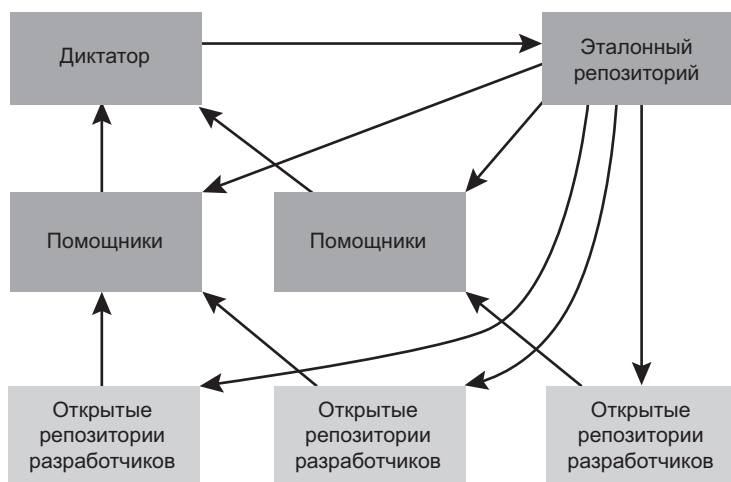


Рис. 5.3. Рабочий процесс с диктатором и помощниками

Рабочий процесс этого типа встречается нечасто, но может быть полезен в очень больших проектах или в строгом иерархическом окружении. Он позволяет лидеру проекта (диктатору) делегировать большую часть работы и собирать из разных мест большие фрагменты кода перед его интеграцией.

Заключение

Мы рассмотрели несколько распространенных вариантов рабочего процесса, доступных для таких распределенных систем, как Git, но позже вы убедитесь, что существует еще множество вариантов, зависящих от конкретных рабочих условий. Надеемся, что теперь вы в состоянии выбрать подходящий лично вам вариант, а значит, можно перейти к рассмотрению конкретных примеров реализации основных ролей в рамках рабочих процессов. В следующем разделе вы познакомитесь с распространенными вариантами содействия проекту.

Содействие проекту

Описать процесс содействия проекту сложно, главным образом из-за наличия множества подходов к его организации. Крайняя гибкость Git обеспечивает большую вариативность способов совместной работы, а значит, конкретный способ содействия проекту будет зависеть от конкретной ситуации. Следует учитывать количество активных участников, выбранную схему работы, ваши права доступа к коммитам, а возможно, и способ принятия изменений от сторонних разработчиков.

Первый фактор — количество активных участников. Сколько пользователей добавляет к проекту свой код и насколько часто? Зачастую это два-три разработчика с несколькими коммитами в день, а порой и меньше. В более крупных компаниях или проектах число разработчиков может измеряться сотнями или даже тысячами ежедневных коммитов. С увеличением числа разработчиков становится все труднее гарантировать возможность аккуратного применения изменений или легкого выполнения слияний. Изменения, которые вы отправляете, могут оказаться устаревшими или частично недействительными из-за данных, которые были вставлены в проект, пока вы работали или пока результаты вашего труда ждали одобрения и применения. Как сделать так, чтобы код оставался согласованным, а коммиты — актуальными?

Следующим фактором является выбранный для проекта тип рабочего процесса. Это должен быть централизованный процесс с наличием у каждого разработчика права на запись в главный репозиторий? Или существует человек, отвечающий за поддержку проекта и проверку всех присланных исправлений перед их интеграцией? Все ли исправления проверяются и одобряются специалистами? Вовлечены ли вы в этот процесс? Присутствуют ли в системе помощники и нужно ли первым делом отправлять свою работу им?

Следующий вопрос касается ваших прав на добавление коммитов. Схема вклада в проект зависит от наличия у вас доступа на запись. Если его у вас нет, каким образом осуществляется вклад в проект? Существуют ли какие-либо правила и политики? Какой объем работы вы вносите за один раз? Насколько часто вы это делаете?

От ответов на эти вопросы зависят эффективность вашего содействия проекту и оптимальный в вашей ситуации рабочий процесс. Все эти вещи мы рассмотрим в виде набора примеров, переходя от простых к более сложным; надеемся, что данные иллюстрации в будущем позволят вам организовывать собственные варианты рабочих процессов в зависимости от конкретных обстоятельств.

Рекомендации по созданию коммитов

Прежде чем мы приступим к рассмотрению конкретных сценариев, поговорим немного о сообщениях фиксации. Подробная инструкция по созданию коммитов значительно облегчает работу с Git и сотрудничество с другими разработчиками. У Git-проекта есть документация с хорошими советами по созданию коммитов, на основе которых делаются исправления. Ознакомиться с ней можно в файле `Documentation/SubmittingPatches`, входящем в исходный Git-код.

Во-первых, в отправляемых вами фрагментах кода не должно быть ошибок, связанных с пробелами. В Git существует команда, легко позволяющая проверить их наличие, — `git diff --check`. Обязательно применяйте ее перед отправкой коммитов (рис. 5.4).



```
bash
lib/simplegit.rb:5: trailing whitespace.
+ @git_dir = File.expand_path(git_dir)
lib/simplegit.rb:7: trailing whitespace.
+
lib/simplegit.rb:20: trailing whitespace.
+end
(END)
```

Рис. 5.4. Результат выполнения команды `git diff --check`

Эта команда сообщит, не собираетесь ли вы отправить коммит, содержащий так раздражающие других разработчиков ошибки, связанные с пробелами.

Во-вторых, старайтесь, чтобы каждый коммит представлял собой логически обособленный набор изменений. Желательно, чтобы изменения были не очень громоздкими, — не стоит все выходные работать над пятью проблемами, чтобы в понедельник

отправить один огромный коммит. Даже если в течение выходных изменения не фиксировались, в понедельник воспользуйтесь областью индексирования и разбейте результаты своего труда на части, по одному коммиту на проблему с содержательным сообщением фиксации в каждом случае. Если какие-то изменения относятся к одному и тому же файлу, попробуйте воспользоваться командой `git add --patch` для индексации файлов по частям. Снимок состояния проекта на вершине ветки не зависит от того, один коммит вы сделали или пять, если все изменения добавляются одновременно. Поэтому постарайтесь облегчить жизнь коллегам-разработчикам, которые будут просматривать плоды ваших усилий. Кроме того, такой подход упрощает процедуру извлечения или отмены одного изменения из внесенного набора, если вдруг в будущем появится такая необходимость.

Не следует забывать и про сообщение фиксации. Привычка к написанию содержательных сообщений упрощает работу с Git и облегчает сотрудничество. Как правило, сообщение должно начинаться со строки с лаконичным описанием изменений, содержащей не более 50 символов. Затем следует пустая строка и более подробное описание. Git-проект требует включить в этот текст информацию о причине изменений и сопоставление вашей реализации с предыдущим поведением. Желательно действовать в соответствии с этими рекомендациями. Кроме того, в этих сообщениях рекомендуют использовать повелительное наклонение. Другими словами, следует писать команды. Вместо «я добавил тест для» пишите «добавьте тест для». Вот пример шаблона, написанный Тимом Поупом (Tim Pope):

Краткое (до 50 символов) описание изменений

Более детальный, поясняющий текст, если он требуется. Ограничьтесь примерно 72 символами. В некоторых контекстах первая строка рассматривается как тема электронного письма, а остальной текст — как тело письма. Важна пустая строка, отделяющая краткое описание от тела письма (если тело письма присутствует); в противном случае такие инструменты, как `rebase`, могут воспринять написанное некорректно.

Дальнейшие абзацы идут после пустых строк.

- Возможны маркированные списки
- Роль маркера обычно играет дефис или звездочка, с одним пробелом перед ними и пустыми строками между пунктами, но возможны и другие реализации

Если ваши сообщения фиксации выглядят подобным образом, это весьма удобно как для вас, так и для ваших коллег-разработчиков. Примеры хорошо сформулированных сообщений фиксации можно найти в Git-проекте — попробуйте запустить для него команду `git log --no-merges` и почитать корректно отформатированную историю коммитов проекта.

В примерах из этой книги вы таких сообщений не увидите. Это сделано для экономии места; вместо них мы применяли параметр `-m` команды `git commit`. Поэтому не стоит считать примеры из книги руководством к действию.

Работа в маленькой группе

Простейшей организационной структурой, которая может вам встретиться, является закрытый проект с одним или двумя другими разработчиками. Определение «закрытый» в данном случае используется потому, что создаваемый этой группой код недоступен остальному миру. Право на запись в репозиторий есть как у вас, так и у остальных разработчиков.

В такой среде возможен рабочий процесс, которого придерживаются при работе с Subversion или другой централизованной системой. При этом у вас сохраняются такие преимущества, как локальные коммиты и упрощенные процедуры слияния и ветвления, но рабочий процесс будет отличаться в основном тем, что во время фиксации изменений слияние выполняется на стороне клиента, а не на сервере. Рассмотрим, как это выглядит, на примере двух разработчиков, пользующихся общим репозиторием. Джон клонирует репозиторий, добавляет свои изменения и фиксирует их локально. (Для сокращения фрагментов кода служебные сообщения заменены многоточием.)

```
# Компьютер Джона
$ git clone john@github:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Джессика — второй разработчик — делает то же самое, то есть клонирует репозиторий и фиксирует внесенные в копию изменения:

```
# Компьютер Джессики
$ git clone jessica@github:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Затем Джессика отправляет свою работу на сервер:

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

Джон также пытается отправить на сервер результаты своего труда:

```
# Компьютер Джона
$ git push origin master
```

```
To john@github:simplegit.git
! [rejected] master -> master (non-fast forward)
error: failed to push some refs to 'john@github:simplegit.git'
```

Джон получает отказ в совершении операции, так как Джессика успела отправить свои изменения первой. Этот момент крайне важно понять, особенно если вы привыкли работать с системой Subversion. Мы видим, что разработчики редактировали разные файлы. В этом случае Subversion выполняет слияние на сервере автоматически, а работая с Git, вы должны сделать это вручную на своей машине. Джону следует скачать изменения, внесенные Джессикой, и выполнить их слияние, и только после этого он получит разрешение на отправку результатов своего труда:

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

То, как при этом будет выглядеть локальный репозиторий Джона, показано на рис. 5.5.

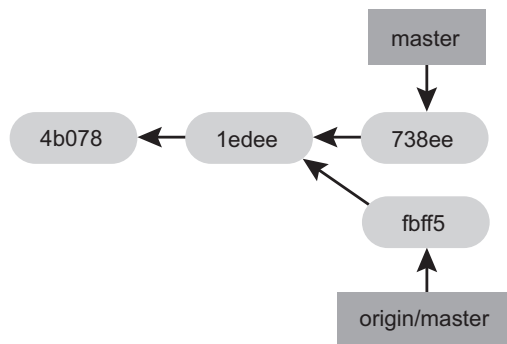


Рис. 5.5. Расходящаяся история Джона

У Джона есть ссылка на выложенные Джессикой изменения, и он должен осуществить их слияние со своей работой:

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

Слияние в данном случае происходит без проблем. История коммитов Джона начинает выглядеть так, как показано на рис. 5.6.

Теперь Джон может протестировать свой код, убедиться, что он по-прежнему корректно работает, и выложить на сервер свои наработки, уже объединенные с наработками Джессики:

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

История коммитов Джона после этой операции показана на рис. 5.7.

В это время Джессика работала над тематической веткой. Она создала ветку `issue54` и выполнила в ней три коммита. Внесенные Джоном изменения она пока не скачала, поэтому история ее изменений будет выглядеть так, как показано на рис. 5.8.

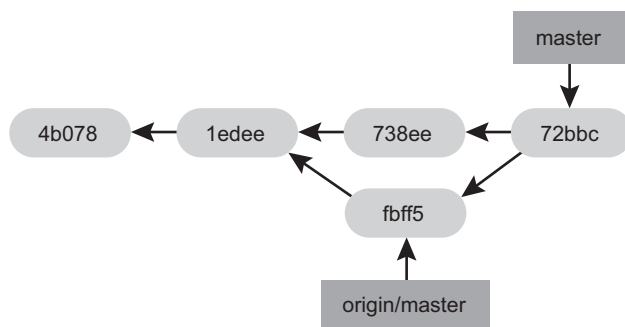


Рис. 5.6. Репозиторий Джона после слияния с веткой `origin/master`

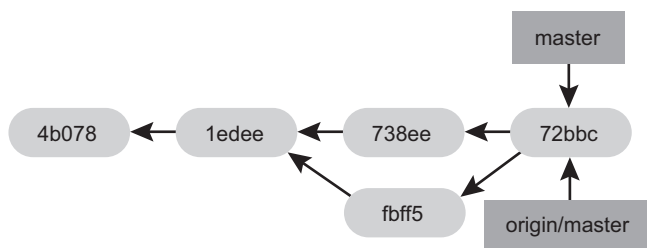


Рис. 5.7. История коммитов Джона после отправки изменений на сервер `origin`

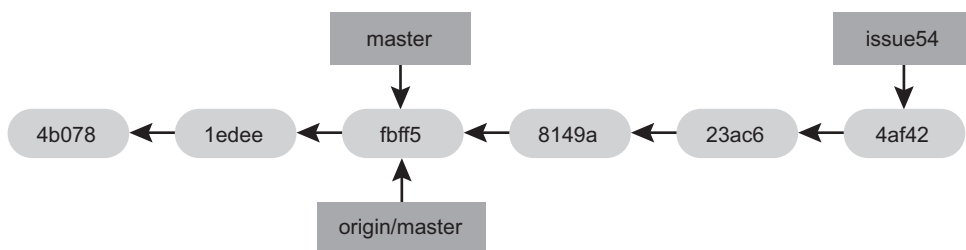


Рис. 5.8. Исходная история коммитов Джессики

Чтобы синхронизировать свою работу с наработками Джона, Джессика скачивает с сервера изменения:

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

После получения выложенных Джоном изменений история коммитов Джессики выглядит так, как показано на рис. 5.9.

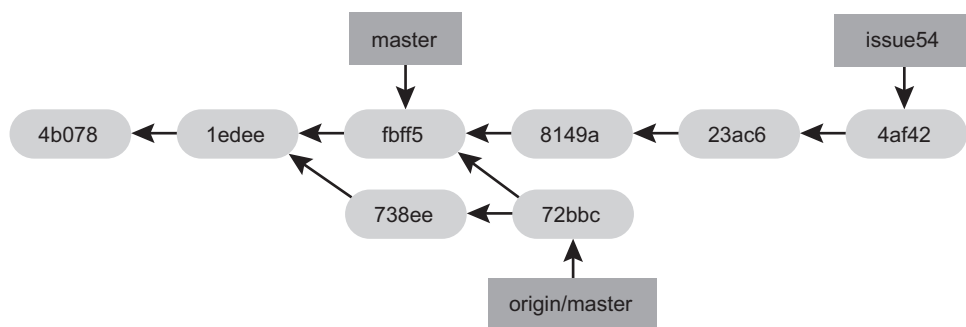


Рис. 5.9. История Джессики после скачивания изменений Джона

Джессика считает свою тематическую ветку законченной и хочет узнать, с чем нужно слить ее работу, чтобы получить возможность отправить ее на сервер. Для этого она использует 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

removed invalid default value
```

Запись **issue54..origin/master** служит фильтром для команды **log**, указывающим, что отображать нужно только те коммиты из последней ветки (в данном случае это ветка **origin/master**), которых нет в первой ветке (в данном случае это ветка **issue54**).

Эта команда показывает, что Джон успел зафиксировать одно изменение, которое Джессика пока не добавила к своей работе. Если она сливает к себе ветку **origin/master**, этот единственный коммит модифицирует ее локальные изменения.

После этого Джессика может слить свою тематическую ветку в свою ветку **master**, туда же слить работу Джона (**origin/master**), а потом отправить все изменения на сервер. Для интеграции всего этого она первым делом переходит в свою ветку **master**:

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

Теперь можно слить к себе ветку `origin/master` или ветку `issue54` — обе они в истории коммитов находятся выше, поэтому порядок слияния не играет роли. Конечное состояние репозитория в обоих случаях получится идентичным, слегка отличаться будет только история. Джессика решает сначала выполнить слияние ветки `issue54`:

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

Как видите, проблем не возникло. Это обычное слияние перемотки. Теперь Джессика добавляет к себе работу Джона (`origin/master`):

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

Слияние проходит нормально, и история коммитов Джессики начинает выглядеть так, как показано на рис. 5.10.

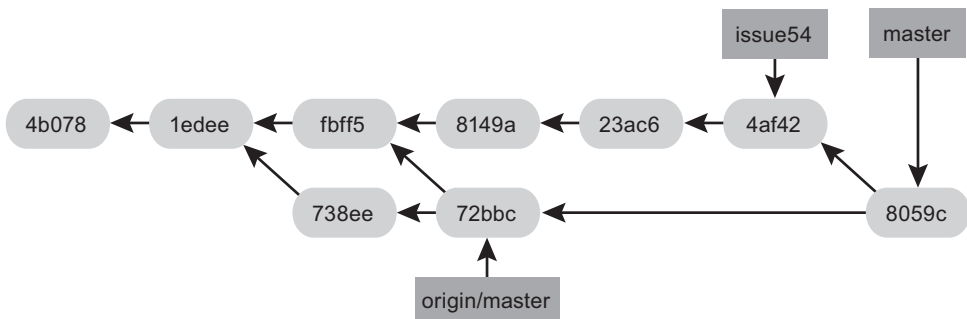


Рис. 5.10. История коммитов Джессики после добавления коммитов Джона

Теперь ветка `origin/master` достижима из ветки `master` Джессики, и она может без проблем отправить свою работу на сервер при условии, что Джон за это время не успел ничего добавить (рис. 5.11):

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

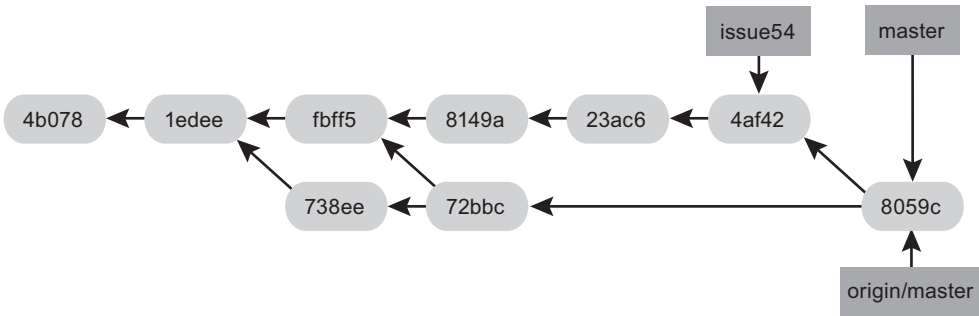


Рис. 5.11. История Джессики после отправки внесенных ею изменений на сервер

Оба разработчика несколько раз фиксировали изменения и успешно выполняли слияние своей работы с работой другого.

Это одна из простейших рабочих схем. Некоторое время вы работаете в тематической ветке, и когда приходит время, сливаете результаты своего труда в ветку **master**. Решив, что пришло время поделиться своими наработками с коллегами, вы скачиваете данные с сервера, и если там появились изменения, сливаете к себе ветку **origin/master**, после чего содержимое ветки **master** можно отправить на сервер. В общем виде эта последовательность соответствует рис. 5.12.

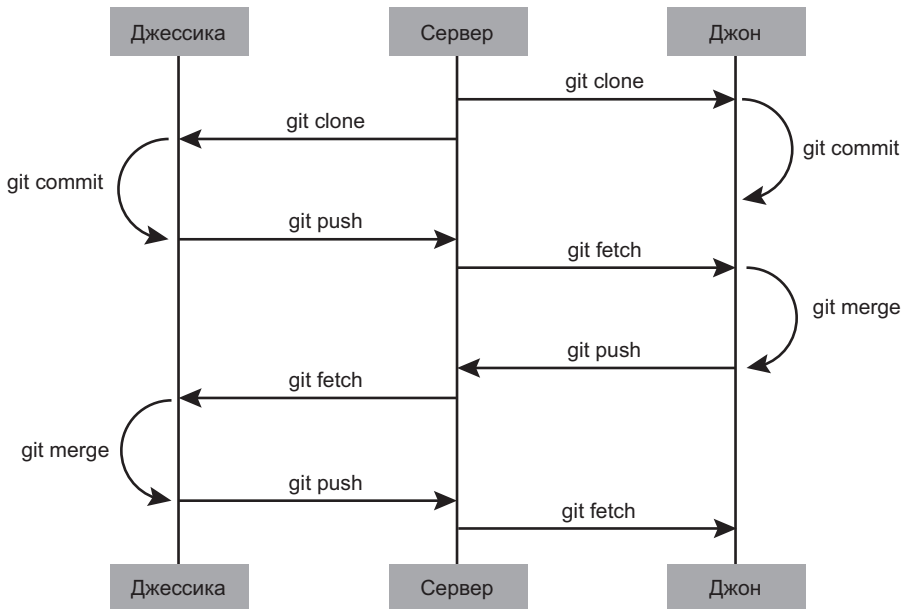


Рис. 5.12. Общая последовательность событий в простой схеме с несколькими разработчиками

Маленькая группа с руководителем

Следующий сценарий демонстрирует роли участников проекта в закрытой группе большого размера. Мы рассмотрим схему работы в среде, где небольшие группы занимаются отдельными задачами, а интеграцию результатов их труда осуществляет третья сторона.

Предположим, что над одной задачей работают Джон и Джессика, а над другой Джессика и Джози. В фирме есть также должность диспетчера интеграции, то есть интеграцией результатов труда отдельных групп занимаются специальные люди, имеющие право обновлять ветку **master** главного репозитория. При таком сценарии вся работа выполняется в ветках отдельных команд, а ее объединение осуществляется позднее.

Рассмотрим рабочий процесс занятой в решении двух задач Джессики, которая при этом сотрудничает с двумя разными разработчиками. Будем считать, что у нее уже есть собственная копия репозитория, и первым делом Джессика занялась задачей **featureA**. Для этой задачи создается новая ветка, в которой производятся некие действия:

```
# 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(-)
```

Теперь ей нужно поделиться своей работой с Джоном, поэтому коммиты из ветки **featureA** требуется отослать на сервер. Но у Джессики нет прав на запись в ветку **master** — этим занимается диспетчер интеграции, — поэтому для совместной работы с Джоном она отправляет свои данные в другую ветку:

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

По электронной почте она сообщает Джону, что он может ознакомиться с выложенной в ветку **featureA** информацией. Ожидая реакции Джона, Джессика решает приступить к работе в ветке **featureB** вместе с Джози. Для начала следует создать эту ветку, используя в качестве основы ветку **master** с сервера:

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

После этого Джессика фиксирует в ветке **featureB** несколько изменений:

```
$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
```

```
[featureB e5b0fdc] made the ls-tree function recursive
1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
1 files changed, 5 insertions(+), 0 deletions(-)
```

После этого репозиторий Джессики выглядит так, как показано на рис. 5.13.

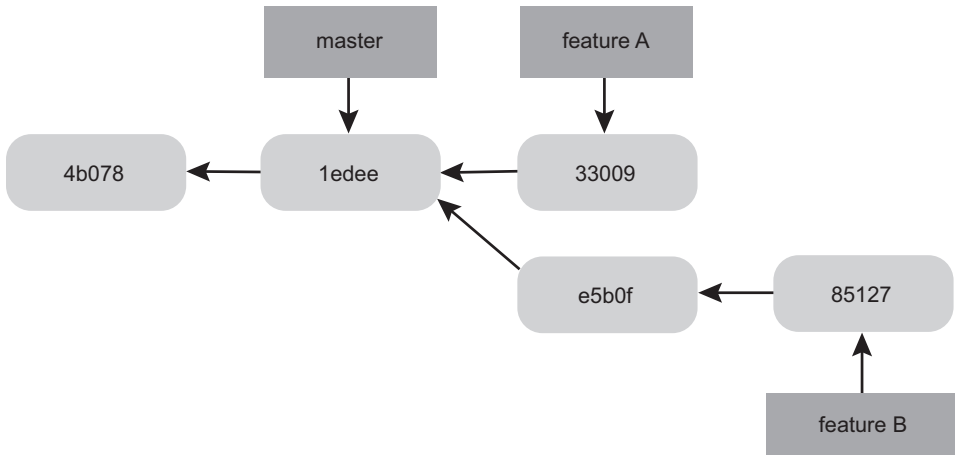


Рис. 5.13. Исходная история коммитов Джессики

Готовясь выложить свою работу в общий доступ, Джессика получает от Джози сообщение, информирующее о наработках, уже выложенных на сервер в ветку **featureBee**. Значит, для получения возможности отправки данных на сервер ей следует слить эти изменения со своими. Коммит Джози извлекается командой **git fetch**:

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

Теперь можно воспользоваться командой **git merge** и добавить эти изменения к результатам своего труда:

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

Но есть небольшая проблема. Полученные после слияния данные из ветки **featureB** Джессике нужно выложить в ветку **featureBee** на сервере. Это делается командой **git push** с указанием разделенных двоеточием названий локальной и удаленной веток:

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

Такая запись называется спецификацией ссылок (**refspec**). Заодно обратите внимание на флаг **-u**, представляющий собой краткое обозначение параметра **--set-upstream**, который устанавливает конфигурацию ветки, облегчающую дальнейшую отправку данных на сервер и скачивание их оттуда.

Внезапно Джессика получает сообщение от Джона. Он информирует, что добавил в ветку **featureA** кое-какие изменения и просит проверить их. Джессика скачивает эти изменения командой **git fetch**:

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

Затем с помощью команды **git log** она смотрит, что именно изменилось:

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

```
changed log output to 30 from 25
```

После этого она сливает работу 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 хочет внести кое-какие дополнения, после чего она фиксирует изменения и снова отправляет коммит на сервер:

```
$ git commit -am 'small tweak'
[featureA ed774b3] small tweak
1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@github:simplegit.git
 3300904..ed774b3 featureA -> featureA
```

После этого ее история коммитов выглядит так, как показано на рис. 5.14.

Джессика, Джози и Джон информируют человека, отвечающего за интеграцию, о том, что ветки **featureA** и **featureBee** на сервере готовы к добавлению в основную ветку разработки. После завершения интеграции скачивание данных с сервера приведет к появлению новых коммитов слияния (рис. 5.15).

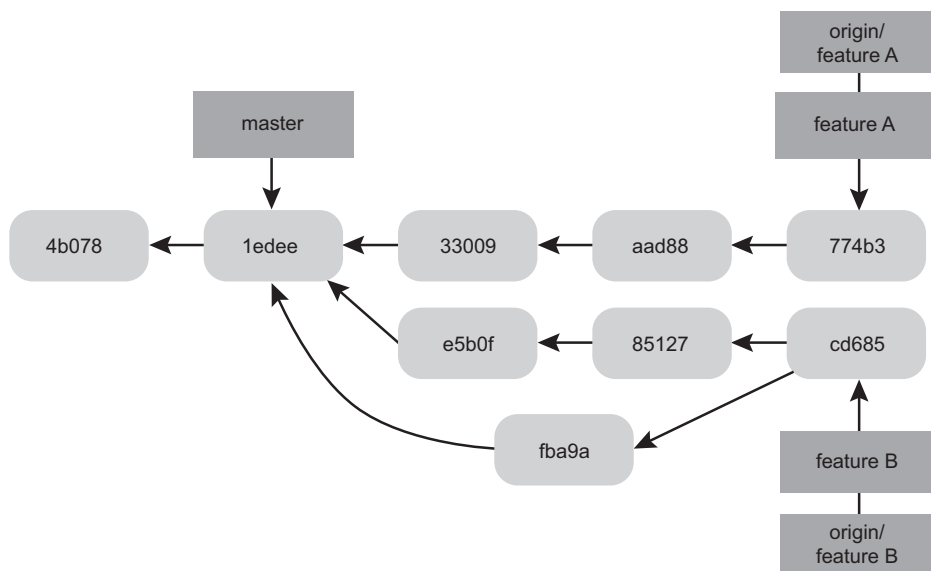


Рис. 5.14. История коммитов Джессики после фиксации изменений в ветке с решаемой задачей

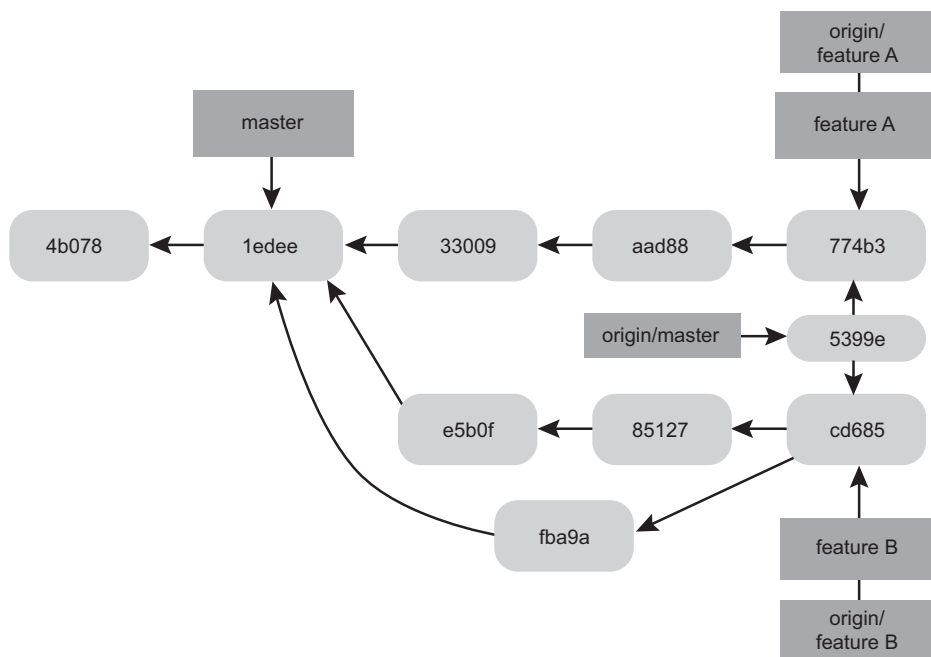


Рис. 5.15. История Джессики после слияния обеих ее тематических веток

Многие группы переходят на систему Git именно ради разработок в параллельном режиме с последующим их объединением. Возможность организовать работу небольших групп в удаленных ветках, не мешая при этом всей команде, является огромным преимуществом Git. Последовательность действий в описанной схеме работы представлена на рис. 5.16.

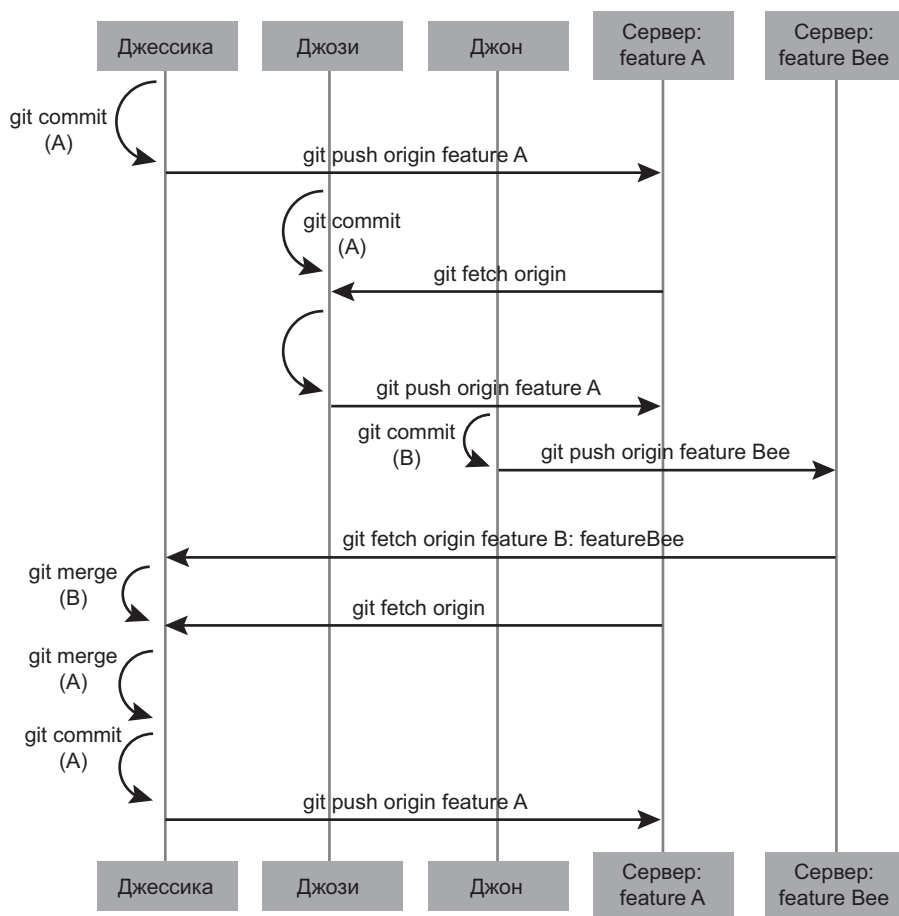


Рис. 5.16. Базовая последовательность действий при наличии диспетчера интеграции

Открытый проект, ветвление

Вклад в открытый проект — это совсем другое дело. Так как права на прямое обновление веток проекта у вас отсутствуют, нужно каким-то способом организовать

работу с лицами, отвечающими за его поддержку. Сначала мы рассмотрим участие в проекте посредством ветвлений. Многие сайты, предоставляющие хостинг для Git (в том числе GitHub, BitBucket, Google Code, repo.or.cz), дают такую возможность, да и владельцы проектов, ожидающие именно такой формы взаимодействия, тоже весьма многочисленны. Следующий же раздел посвящен проектам, владельцы которых предпочитают принимать исправления по электронной почте.

Первым делом требуется клонировать основной репозиторий, создать тематическую ветку для задачи или набора задач, которые вы собираетесь решать, и приступить к работе. Последовательность ваших действий будет следующей:

```
$ git clone (url)
$ cd project
$ git checkout -b featureA
# (выполнение работы)
$ git commit
# (выполнение работы)
$ git commit
```

ПРИМЕЧАНИЕ

Команда `rebase -i` позволяет объединить все наработки в один коммит или реорганизовать их внутри коммита таким образом, чтобы владельцу проекта было проще с ними знакомиться.

Когда вы завершите работу с веткой и решите поделиться результатами с владельцами проекта, перейдите на главную страницу проекта и щелкните на кнопке **Fork**, чтобы получить собственную копию проекта с правом на запись. URL-адрес этого нового репозитория нужно добавить в список удаленных репозиториях. Мы дадим ему имя `myfork`:

```
$ git remote add myfork (url)
```

Теперь следует отправить в этот репозиторий ваши наработки. Проще всего сразу отправить ветку, в которой вы работаете, а не сливать ее предварительно с веткой **master**. Это избавит вас от необходимости откатывать изменения ветки **master**, если вашу работу не примут или примут только частично. Если владельцы проекта выполняют слияние, перемещение или частичное включение вашей работы в основной код, вы в конечном счете все равно получите все обратно, скачав изменения из главного репозитория:

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

После отправки наработок в свою копию следует послать уведомление лицу, отвечающему за поддержку проекта. Часто эту процедуру называют запросом на включение (`pull request`). Запрос можно сгенерировать на сайте: например, у GitHub есть собственный механизм таких запросов, который рассматривается в следующей главе, — или воспользоваться командой `git request-pull`

и по электронной почте отправить результат ее применения ответственному за поддержку проекта.

В качестве аргумента команда `request-pull` принимает имя базовой ветки, в которую вы хотите включить свою тематическую ветку, и URL-адрес Git-репозитория, из которого диспетчер интеграции может взять ваши наработки. Результатом работы команды становится краткая сводка изменений, которые вы попросили включить в проект. К примеру, если Джессика после пары коммитов в тематической ветке и отправки этой ветки на сервер хочет послать Джону запрос на включение, ей нужно сделать следующее:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
    added a new function
are available in the git repository at:

  git://githost/simplegit.git featureA

Jessica Smith (2):
  add limit to log function
  change log output to 30 from 25

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

Вывод этой команды можно отправить диспетчеру интеграции — в нем сообщается, где начинается ветка с изменениями, кратко отображаются все коммиты и указывается, откуда можно забрать изменения.

Для проекта, поддержкой которого занимаются другие люди, в общем случае имеет смысл завести ветку `master`, непрерывно следящую за веткой `origin/master`, а всю работу выполнять в тематических ветках, которые легко можно удалить, если ваши коммиты не будут приняты. Наличие тематических веток для отдельных задач также облегчает процедуру перемещения работы в случае, когда за время ее выполнения верхняя точка главного репозитория сместилась и применение коммитов без конфликтов стало невозможным. Например, если вы хотите добавить к проекту работу по другой теме, не нужно продолжать работу над тематической веткой, содержимое которой вы только что отправили на сервер. Начните с ветки `master` главного репозитория:

```
$ git checkout -b featureB origin/master
# (выполнение работы)
$ git commit
$ git push myfork featureB
# (письмо диспетчеру интеграции)
$ git fetch origin
```

Теперь каждая тема будет находиться внутри собственного контейнера (похоже на очередь из исправлений), которые вы можете перезаписывать, перемещать и редактировать, не затрагивая остальных тем (рис. 5.17).

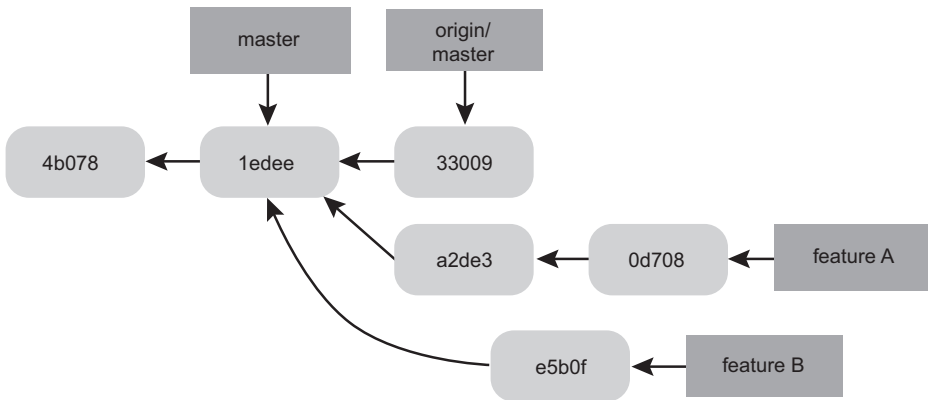


Рис. 5.17. Исходная история коммитов с веткой featureB

Предположим, диспетчер интеграции скачал набор чьих-то исправлений, и после этого оказалось, что ваша первая ветка не допускает бесконфликтного слияния. Вы можете попробовать переместить эту ветку на вершину ветки `origin/master`, разрешить конфликт и повторно отправить свои изменения:

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

История коммитов при этом изменится (рис. 5.18).

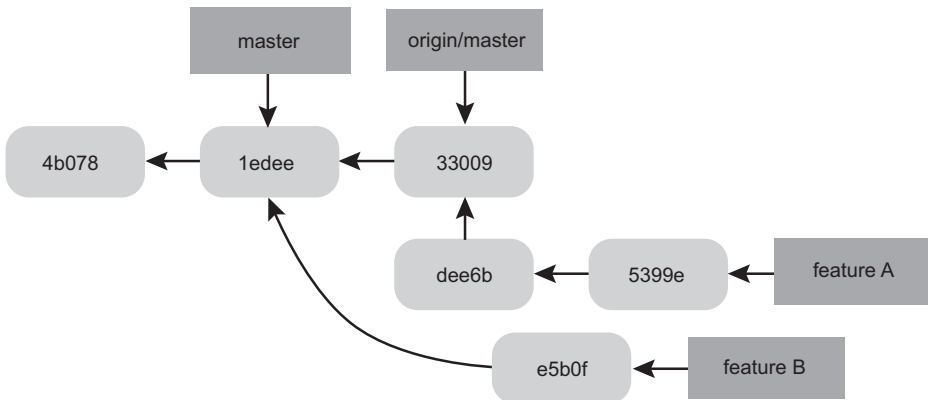


Рис. 5.18. История коммитов после работы в ветке featureA

Так как ветка была перемещена, чтобы заместить ветку `featureA` на сервере коммитом, который не является ее потомком, нужно воспользоваться командой `push` с параметром `-f`. В качестве альтернативы эти наработки можно выложить в другую ветку на сервере (назвав ее, к примеру, `featureAv2`).

Однако возможен и другой сценарий: диспетчер интеграции ознакомился с вашей работой на второй ветке и одобрил идею, но хотел бы поменять некоторые детали реализации. Для вас это возможность переместить свою работу таким образом, чтобы она базировалась в текущей ветке **master** проекта. Вы создаете новую ветку на базе ветки **origin/master**, вставляете туда изменения из ветки **featureB**, разрешаете все конфликты, вносите требуемые изменения в реализацию и отправляете это все на сервер как новую ветку:

```
$ git checkout -b featureBv2 origin/master^{  
$ git merge --no-commit --squash featureB  
# (изменение реализации)  
$ git commit  
$ git push myfork featureBv2
```

Параметр **--squash** превращает все содержимое сливаемой ветки в один коммит, не являющийся коммитом слияния, и помещает его на вершину текущей ветки. Параметр **--no-commit** отменяет автоматическую запись коммита. В этом случае вы сможете перед записью нового коммита добавить туда все изменения из другой ветки и внести прочие коррективы (рис. 5.19).

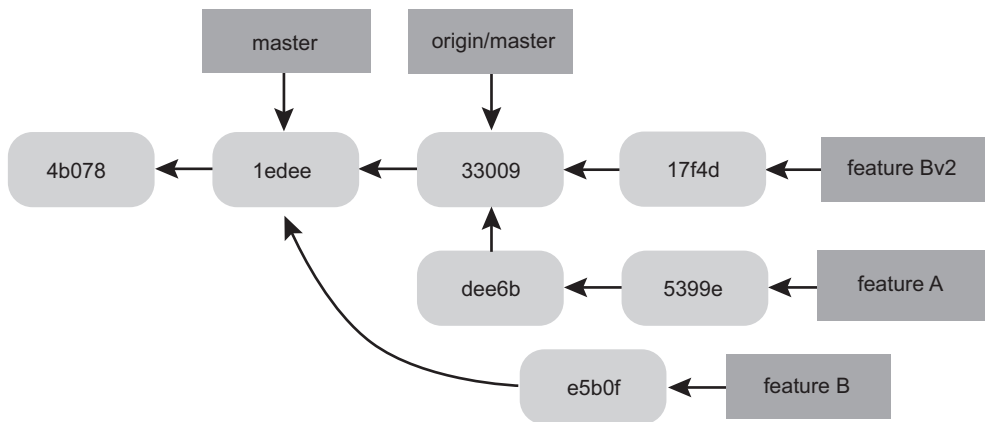


Рис. 5.19. История коммитов после работы на ветке **featureBv2**

Теперь диспетчеру интеграции можно отправить сообщение, информирующее о том, что все требуемые изменения сделаны и находятся в ветке **featureBv2**.

Открытый проект, электронная почта

Во многих проектах существуют установленные процедуры приема изменений. Точные правила каждый раз нужно выяснять отдельно. Тем не менее в некоторых старых больших проектах изменения принимаются именно через списки рассылки для разработчиков, поэтому мы рассмотрим этот вариант на примере.

Отличие от предыдущего случая состоит в способе внесения изменений. Вместо создания собственной копии с правом на запись в случае каждого набора коммитов генерируется версия для электронной почты с целью отправки в список рассылки разработчиков:

```
$ git checkout -b topicA
# (выполнение работы)
$ git commit
# (выполнение работы)
$ git commit
```

Итак, у нас есть два готовых к отправлению коммита. Команда **git format-patch** превратит их в сообщения электронной почты. Темой послужит первая строка сообщения фиксации, остальная часть этого сообщения и исправления войдут в тело письма. Причем в изменениях из сгенерированного командой **format-patch** письма корректно сохраняется вся информация о коммите.

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

Команда **format-patch** выводит имена создаваемых ею файлов с исправлениями. Параметр **-M** заставляет Git следить за переименованиями файлов. В итоге каждый файл приобретает примерно вот такой вид:

```
$ 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
```

Эти файлы с исправлениями можно отредактировать, добавив к электронному письму еще какую-то информацию, которую вы не захотели включать в сообщение

фиксации. Добавленный между строкой --- и началом исправления (строка `diff --git`) текст сможет прочитать разработчик, но после применения исправлений он исчезнет.

Для отправки в список рассылки файл нужно либо вставить в почтовый клиент, либо воспользоваться специальной программой, осуществляющей отправку из командной строки. В первом случае зачастую возникают ошибки форматирования, особенно в «интеллектуальных» клиентах, не сохраняющих символы новой строки и пробелы. К счастью, Git предоставляет инструмент для корректной отправки отформатированных изменений через IMAP. Мы также познакомимся с отправкой изменений через службу Gmail, так как этот агент является, наверное, самым распространенным; подробные инструкции для разных почтовых программ вы найдете в уже упоминавшемся тут файле `Documentation/SubmittingPatches`, входящем в исходный Git-код.

Первым делом нужно настроить раздел `imap` в файле `~/.gitconfig`. Значения можно добавлять как по одному набором команд `git config`, так и вручную. В итоге конфигурационный файл должен принять следующий вид:

```
[imap]
  folder = «[Gmail]/Drafts»
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = p4ssw0rd
  port = 993
  sslverify = false
```

В случае IMAP-серверов, не работающих с протоколом SSL, последние две строки не требуются, а параметр `host` должен иметь значение `imap://` вместо `imaps://`. После завершения настроек набор исправлений следует поместить в папку `Drafts` на указанном IMAP-сервере командой `git send-email`:

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

После этого Git добавит множество служебной информации, которая для каждого отсылаемого исправления будет иметь вот такой вид:

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
      \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] added limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
```

```
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty  
In-Reply-To: <y>  
References: <y>
```

```
Result: OK
```

Теперь вы можете зайти в свою папку **Drafts**, ввести в поле **To** адрес списка рассылки и, возможно, добавить в поле **CC** адрес человека, отвечающего за поддержку той части проекта, к которой относятся ваши изменения. После этого остается только отправить письмо.

Заключение

Мы рассмотрели ряд общепринятых рабочих схем, применяемых для разных типов Git-проектов, с которыми вам наверняка предстоит столкнуться. Вы познакомились с несколькими новыми инструментами, помогающими в организации рабочего процесса. Теперь мы рассмотрим ситуацию с другой стороны: как осуществляется сопровождение Git-проекта? Вы узнаете, что следует делать на месте диктатора или диспетчера интеграции.

Сопровождение проекта

Вы должны уметь не только результативно вносить свой вклад в проекты, но и поддерживать их. Сопровождение проекта может включать в себя как прием и применение исправлений, сгенерированных командой **format-patch** и присланных вам по электронной почте, так и интеграцию изменений из веток тех репозиторий, которые вы добавили к своему проекту в качестве удаленных. Не важно, отвечаете вы за поддержку эталонного репозитория или просто хотите помочь с проверкой и одобрением исправлений. В любом случае вам следует выработать метод приема присылаемых данных, максимально прозрачный для остальных участников проекта и подходящий вам в долгосрочной перспективе.

Работа с тематическими ветками

Новые наработки, предназначенные для интеграции, как правило, желательно протестировать, поместив во временную специально созданную для этой цели ветку. Это упрощает редактирование отдельных изменений и позволяет оставить их до лучших времен, если выяснится, что что-то не работает. Простое, легко запоминающееся имя, связанное с темой содержащейся в ветке работы, например **ruby_client** или что-то столь же наглядное, позволяет без проблем вспомнить назначение ветки, если вам пришлось на время прекратить работу с ней. Человек,

отвечающий за поддержку Git-проекта, как правило, добавляет к этим именам название пространства имен, например `sc/ruby_client`, где `sc` — инициалы того, кто добавил это исправление. Как вы помните, создать ветку на основе вашей ветки `master` можно следующим образом:

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

Мгновенно перейти в создаваемую ветку позволяет команда `checkout -b`:

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

Теперь в эту тематическую ветку можно принимать присылаемые изменения и смотреть, имеет ли смысл сливать их в свои стабильные ветки.

Исправления, присланные по почте

Полученное по электронной почте исправление первым делом следует поместить в тематическую ветку, чтобы оценить его. Это можно сделать как командой `git apply`, так и командой `git am`.

Команда `apply`

Если вы получили коммит, сгенерированный командой `git diff` или Unix-командой `diff` (что, как вы вскоре убедитесь, не рекомендуется), его можно применить при помощи команды `git apply`. Вот как выглядит эта процедура после сохранения присланного исправления по адресу `/tmp/patch-ruby-client.patch`:

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

Эта команда меняет файлы в вашей рабочей папке. Она практически идентична команде `patch -p1`, но более подозрительна и допускает меньше нечетких совпадений. Кроме того, она осуществляет добавление, удаление и переименование файлов, если это задано при форматировании командой `git diff`, чего команда `patch` делать не умеет. Наконец, команда `git apply` работает по принципу «все или ничего», в то время как команда `patch` допускает частичное принятие изменений, оставляя рабочую папку в странном состоянии. В целом команда `git apply` куда более консервативна. Она не создает коммиты — после нее следует вручную осуществить индексацию и зафиксировать изменения.

Еще команда `git apply` позволяет заранее узнать, аккуратно ли применяется исправление. Для этого нужно запустить ее с параметром `--check`, указав нужный файл:

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

Отсутствие вывода в данном случае означает отсутствие проблем в процессе применения исправления. В случае неудачного результата проверки команда завершается с ненулевым статусом, что позволяет использовать ее в сценариях.

Команда `am`

Если исправление прислано опытным Git-пользователем, который сгенерировал его командой `format-patch`, ваша задача упрощается, ведь этот файл содержит сведения об авторе и сообщение фиксации. По возможности рекомендуем участникам проекта пользоваться командой `format-patch`, а не командой `diff`. Команду `git apply` нужно оставить для устаревших исправлений и таких вещей, о которых мы рассказали.

Исправления, сгенерированные командой `format-patch`, применяются командой `git am`. 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 уже знакомого вам по предыдущему разделу вывода команды `format-patch`. Кроме того, это корректный формат mbox для электронной почты. Если кто-то правильно воспользовался командой `git send-email` и прислал вам исправление, которое вы сохранили в формате mbox, сохраненный файл можно передать команде `git am`, и она начнет применять все обнаруженные ею исправления. Если ваш почтовый клиент в состоянии сохранить несколько электронных писем в одном mbox-файле, то команда `git am`, которой вы передадите этот файл, начнет применять их по очереди.

Однако если исправление, сгенерированное командой `format-patch`, было загружено в систему отслеживания ошибок или нечто подобное, файл можно сохранить локально и уже после этого передать его команде `git am`:

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

Как видите, исправление было наложено без ошибок, и для вас автоматически создали новый коммит. Сведения об авторе взяты из полей `From` и `Date`, а сообщение коммита — из поля `Subject` и тела письма (до начала исправления). Скажем, если применить исправление из представленного примера, получится следующий коммит:

```
$ 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
```

В поле **Commit** указаны человек, применивший исправление, и дата этого события. Поле **Author** содержит информацию о том, кто создал исправление, и дату создания.

Однако применение исправлений далеко не всегда происходит гладко. Скажем, может возникнуть ситуация, когда основная ветка слишком расходится с веткой, послужившей основой для исправления, или, к примеру, исправление зависит от другого исправления, которое пока не применили. В этом случае выполнение команды **git am** прекращается, а вы получаете запрос, как следует поступить:

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

Эта команда ставит метки о конфликтах на все файлы, с которыми возникает проблема, аналогично тому, как это происходит при конфликтах слияния или перемещения. Сходен и способ решения этой проблемы: вы редактируете файл, разрешая конфликт, индексируете получившуюся новую версию и запускаете команду **git am --resolved**, чтобы перейти к следующему исправлению:

```
$ (исправление файла)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

Можно заставить Git прибегнуть к более интеллектуальному решению конфликта, добавив к команде параметр **-3** и инициировав трехэтапное слияние. По умолчанию этот параметр отсутствует, так как если в вашем репозитории нет коммита, послужившего основой для исправления, команда работать не будет. При наличии же этого коммита, например если исправление основано на открытом коммите, параметр **-3** в общем случае помогает действовать более интеллектуально в плане применения конфликтующего исправления:

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

В данном случае рассматриваемое исправление уже было применено. Но без параметра **-3** оно имело бы вид конфликтующего.

При применении серии исправлений из **mbox**-файла команду **am** можно запустить в интерактивном режиме. В этом случае после обнаружения очередного исправления будет появляться запрос о необходимости его применения:

```
$ git am -3 -i mbox
Commit Body is:
-----
seeing if this helps the gem
-----
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

Это удобно при работе с большим количеством исправлений, так как если вы забыли, что представляет собой конкретное исправление, на него сначала можно посмотреть, кроме того, можно отказаться от применения уже сделанных ранее исправлений.

После применения всех исправлений по выбранной теме и слияния их с вашей веткой можно принимать решение об их интеграции в стабильную ветку.

Проверка удаленных веток

Если вы получили наработки от Git-пользователя, настроившего свой собственный репозиторий, отправившего туда ряд изменений и приславшего вам его URL-адрес и имя удаленной ветки с изменениями, этот репозиторий можно добавить в качестве удаленного и выполнять все слияния локально.

К примеру, предположим, вы получаете от Джессики сообщение о том, что у нее в ветке **ruby-client** появился классный новый программный компонент. Протестировать этот компонент можно, добавив репозиторий Джессики в качестве удаленного и проверив указанную ветку локально:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Получив от Джессики еще одно письмо с другой веткой и новым замечательным программным компонентом, вы можете сразу извлечь все наработки и перейти в указанную ветку, так как ее хранилище уже фигурирует у вас как удаленное.

Этот метод особенно эффективен, когда вы работаете с человеком на постоянной основе. Если пользователь изредка присылает вам какие-то исправления, проще принять их по электронной почте, чем требовать от него поддержки собственного сервера и постоянно добавлять и ликвидировать удаленные ветки. Вряд ли вы захотите держать у себя сотни удаленных репозиториях, принадлежащих пользователям, которые когда-то прислали вам исправление-другое. Ситуация слегка упрощается в случае сценариев и служб при использовании чужого хостинга, но опять же все зависит от способа ведения разработки вами и остальными участниками проекта.

Другим достоинством данного подхода является получение вместе с исправлениями истории коммитов. Даже если вы унаследуете проблемы со слиянием, всегда можно будет узнать, где кроется их причина; в данном случае по умолчанию используется корректное трехэтапное слияние вместо добавления параметра **-3** в надежде, что

исправление было сгенерировано на основе открытого коммита, который вы сможете посмотреть.

Если же вы хотите принять таким способом изменения от человека, с которым вы не работаете на постоянной основе, можно передать URL-адрес его удаленного репозитория команде `git pull`. Это даст возможность выполнить однократное скачивание данных без сохранения URL-адреса в списке удаленных репозиториях:

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
* branch HEAD -> FETCH_HEAD
Merge made by recursive.
```

Просмотр вносимых изменений

Итак, у вас есть тематическая ветка с наработками участников проекта. Пришла пора решать, что с ними делать. В этом разделе вы заново встретите знакомые команды, но на этот раз они помогут точно определить, к каким результатам приведет включение в вашу основную ветку присланных изменений.

Зачастую имеет смысл изучить все коммиты, присутствующие в этой ветке, но отсутствующие в вашей ветке `master`.

Убрать из списка коммиты, которые уже есть в ветке `master`, позволит параметр `--not`, добавляемый перед именем ветки. Он делает то же самое, что и знакомая вам запись `master..contrib`. К примеру, если участник проекта прислал два изменения, а вы создали ветку `contrib` и применили их там, можно написать:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date: Fri Oct 24 09:53:59 2008 -0700
```

```
    seeing if this helps the gem
```

```
commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date: Mon Oct 22 19:38:36 2008 -0700
```

```
    updated the gemspec to hopefully work better
```

Чтобы увидеть, к каким изменениям приведет каждый коммит, достаточно добавить к команде `git log` параметр `-p`. После этого к каждому коммиту будет добавлен фрагмент с нововведениями.

Чтобы понять, что случится после слияния тематической ветки с другой веткой, потребуется странный трюк. Дело в том, что следующая команда выводит перечень внесенных изменений, но этот результат далеко не всегда является корректным:

```
$ git diff master
```

К примеру, если с момента создания тематической ветки ветка **master** была перемотана вперед, результат окажется непонятным. Дело в том, что Git напрямую сравнивает снимок последнего коммита тематической ветки, в которой вы находитесь, со снимком последнего коммита ветки **master**. Скажем, если вы добавили строку в файл из ветки **master**, прямое сравнение состояний покажет, что в тематической ветке эту строку собираются удалить.

Если ветка **master** является прямым предком вашей тематической ветки, проблем нет; но если история в какой-то точке расходится, выводимые изменения будут иметь такой вид, как будто все новое вы добавляете в тематическую ветку, а все уникальное удаляете из ветки **master**.

Нам же всего лишь требуется посмотреть изменения, добавленные в тематическую ветку — то есть данные, вносимые путем слияния в ветку **master**. Для этого нужно заставить Git сравнить последний коммит тематической ветки с первым предком, который будет у него общим с веткой **master**.

Технически можно вручную определить такого предка и выполнить команду **diff**:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

Но это не очень удобно, поэтому в Git для этого случая предусмотрена особая нотация: запись с тремя точками. Если в контексте команды **diff** поставить после имени одной из веток три точки, вы увидите разницу между последним коммитом ветки, в которой вы находитесь, и ее общим предком с другой веткой:

```
$ git diff master...contrib
```

В такой форме команда выводит вам только те наработки из тематической ветки, которые появились там после расхождения с веткой **master**. Это очень удобный синтаксис, так что его имеет смысл запомнить.

Интеграция чужих наработок

В конце концов наступает момент, когда все наработки в тематической ветке готовы к интеграции в более стабильную ветку. Остается вопрос, как это сделать. Какую схему работы вы предпочтете для поддержки своего проекта? Существует множество вариантов, некоторые из них рассмотрены в данном разделе.

Схемы слияния

Одна из простых рабочих схем состоит в добавлении наработок в вашу ветку **master**. В этом сценарии ветка **master** содержит основную стабильную версию кода. Если на тематической ветке находится сделанная вами или присланная кем-то и проверенная вами работа, содержимое этой ветки сливают в ветку **master**, затем ветку

удаляют и продолжают процесс. Если в вашем репозитории находятся наработки из двух веток `ruby_client` и `php_client`, имеющие такой же вид, как показано на рис. 5.20, и вы выполняете сначала слияние ветки `ruby_client`, а потом — ветки `php_client`, история будет выглядеть в соответствии с рис. 5.21.

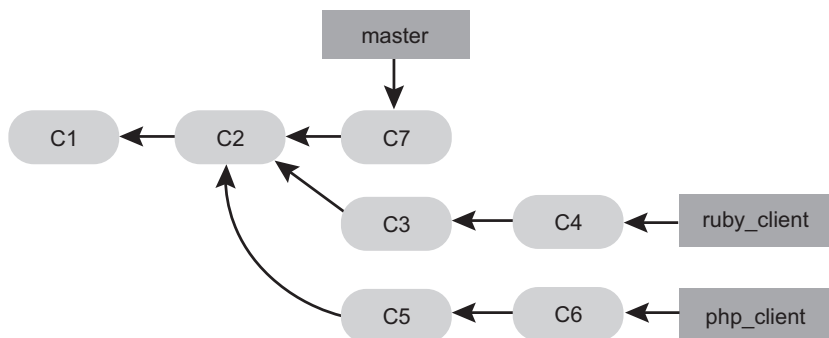


Рис. 5.20. История коммитов с несколькими тематическими ветками

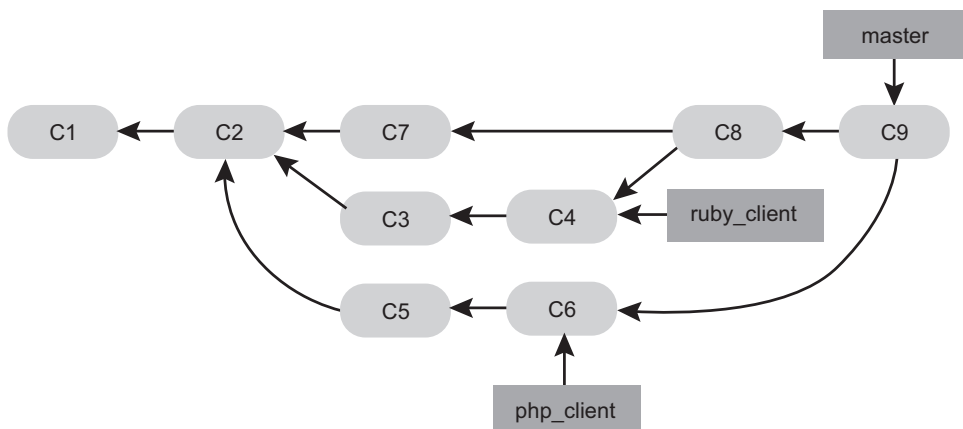


Рис. 5.21. Вид после слияния тематических веток

Это, возможно, самая простая рабочая схема, но в случае более крупных или более стабильных проектов, когда нужно очень аккуратно подходить к добавлению изменений, она может привести к проблемам.

Для важных проектов имеет смысл пользоваться двухэтапным циклом слияний. В этом случае у вас будут две долгоживущие ветки `master` и `develop`, при этом первая ветка обновляется только при добавлении крайне стабильного кода, а весь новый код попадает в ветку `develop`. Содержимое этих веток регулярно отправляется в открытый репозиторий. При каждом появлении новой тематической ветки

для слияния (рис. 5.22) ее содержимое сливают в ветку **develop** (рис. 5.23). Затем, когда вы помечаете новую версию тегом, ветка **master** перематывается до стабильного коммита ветки **develop** (рис. 5.24).

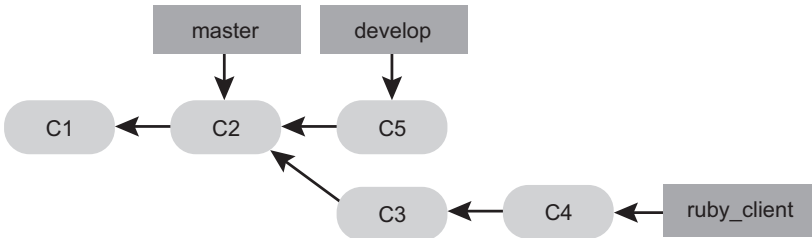


Рис. 5.22. До слияния с тематической веткой

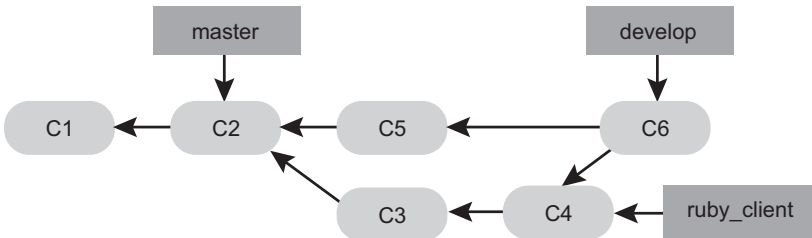


Рис. 5.23. После слияния с тематической веткой

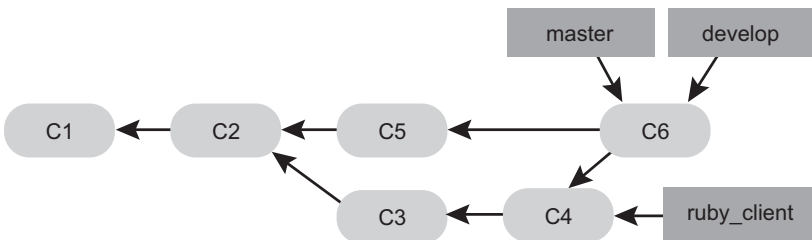


Рис. 5.24. После выхода версии проекта

В этом случае клонирующие ваш репозиторий пользователи могут либо получить последнюю стабильную версию из ветки **master** и легко поддерживать актуальность кода, либо перейти в ветку **develop**, содержащую новейшие обновления. Данный подход можно развить, добавив ветку для интеграции, в которой будет осуществляться слияние всех наработок. А когда код этой ветки станет стабильным и пройдет тестирование, его можно будет слить в ветку **develop**; если в течение некоторого времени он будет работать нормально, вы выполните перематку ветки **master**.

Схема с большим количеством слияний

В Git-проекте есть четыре долгоживущие ветки: **master**, **next** и **pu** (последняя расшифровывается как *proposed updates* — предложенные обновления) для новых наработок и **maint** для поддержки совместимости с более старыми версиями. Предложенные участниками проекта наработки накапливаются в тематических ветках в репозитории отвечающего за поддержку проекта лица описанным ранее способом. На этом этапе производится оценка содержимого тематических веток, чтобы определить, работают ли предложенные фрагменты так, как положено, или им требуется доработка. Если все в порядке, тематические ветки сливаются в ветку **next**, которая отправляется на сервер, чтобы у каждого была возможность опробовать результат интеграции (рис. 5.25).

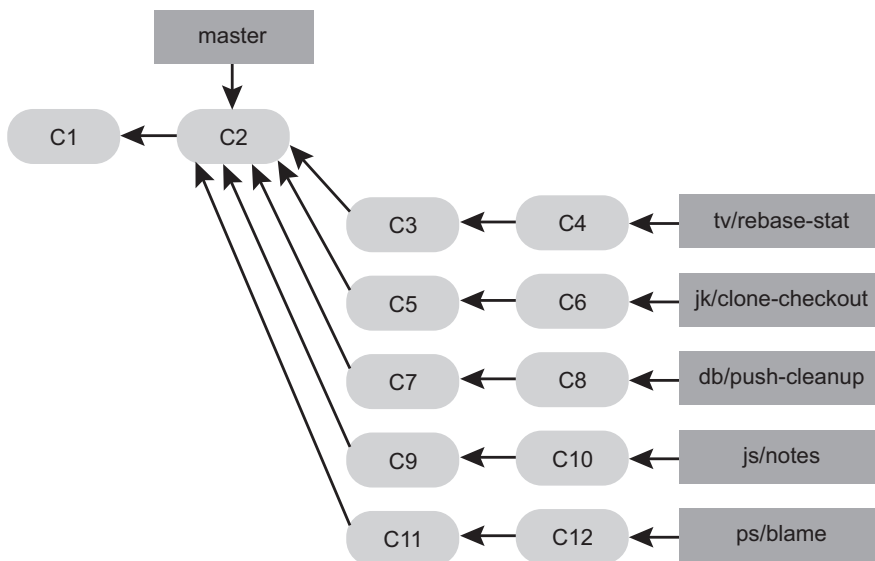
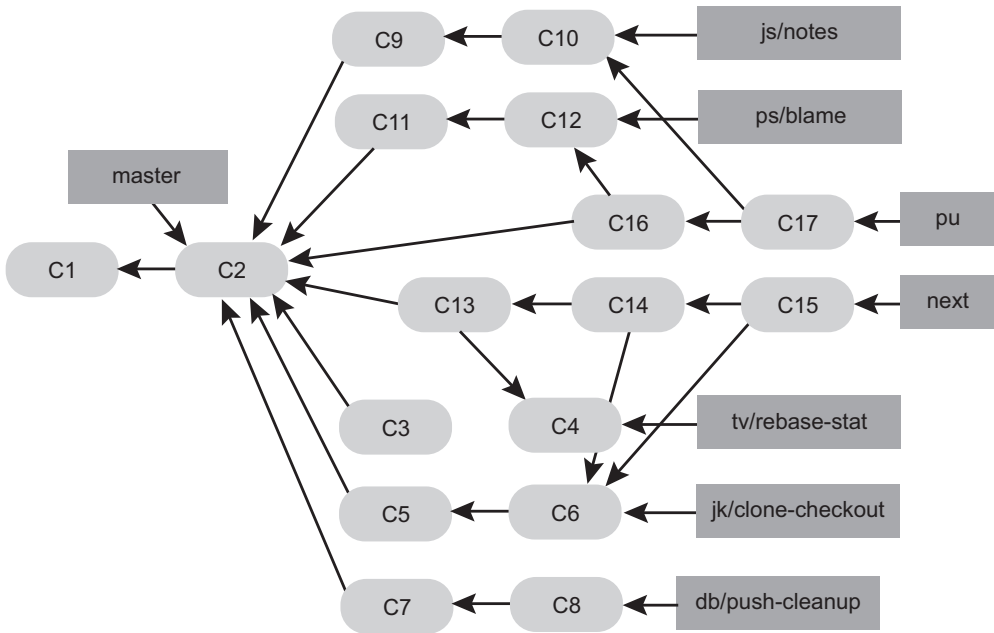


Рис. 5.25. Управление группой параллельных тематических веток участников проекта

Если содержимое тематических веток требует доработки, оно сливается в ветку **pu**. Когда выясняется, что предложенный код полностью стабилен, он сливается в ветку **master** и перестраивается с учетом данных из ветки **next**, пока еще не попавших в ветку **master**. Это означает, что ветка **master** почти всегда движется вперед, ветка **next** периодически подвергается перемещениям, а содержимое ветки **pu** перемещается еще чаще (рис. 5.26).

Когда, наконец, тематическая ветка полностью сливается в ветку **master**, она удаляется из репозитория. В проекте Git также присутствует ветка **maint**, которая является ответвлением от последней выпущенной версии, предназначенным

для исправлений более старой версии. В результате после клонирования Git-репозитория у вас оказывается четыре ветки, переключаясь между которыми вы можете оценить проект на разных стадиях разработки в зависимости от того, какую версию вы хотите получить или каким образом собираетесь сделать вклад в проект. Лицо, отвечающее за поддержку проекта, при этом действует в рамках структурированного рабочего процесса, помогающего ему изучать новые присланные изменения.



коммитов. Берутся представленные в коммите изменения и делается попытка применить их в вашей текущей ветке. Это удобно, если из набора коммитов в тематической ветке вы хотите интегрировать только один или если в этой ветке есть всего один коммит, но вы предпочитаете произвести операцию отбора, а не перемещения. Предположим, ваш проект выглядит так, как показано на рис. 5.27.

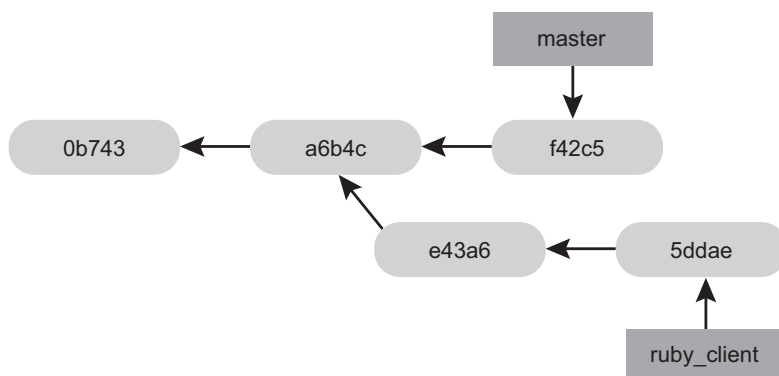


Рис. 5.27. Пример истории, из которой нужно отобрать лучшее

Чтобы поместить коммит **e43a6** в свою ветку **master**, выполните команду:

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcd
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
3 files changed, 17 insertions(+), 3 deletions(-)
```

Эта команда извлечет изменения, появившиеся в коммите **e43a6**, но при этом изменится контрольная сумма SHA-1 коммита, так как у него будет другая дата применения (рис. 5.28).

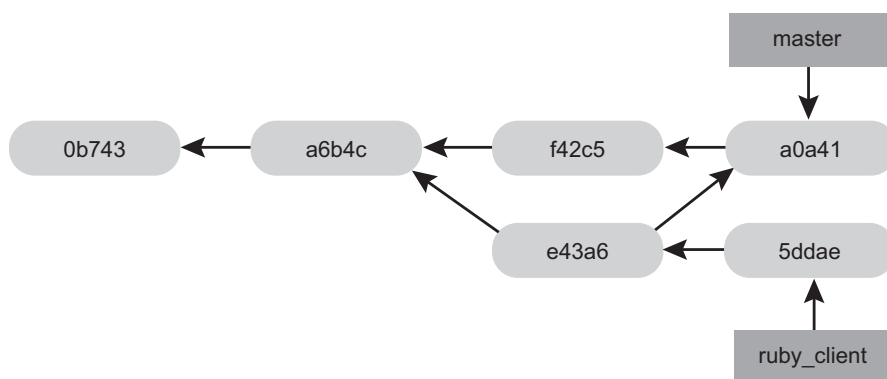


Рис. 5.28. История после отбора лучшего коммита из тематической ветки

Теперь тематическую ветку можно удалить, отбросив коммиты, которые вы не собираетесь включать в проект.

Программный компонент `rerere`

Для тех, кто часто выполняет слияния и перемещения или поддерживает долгоживущие ветки, в Git есть вспомогательный программный компонент `rerere`.

Как следует из названия, которое представляет собой аббревиатуру фразы «reuse recorded resolution» (повторное использование записанного решения), он дает возможность сократить процедуру ручного разрешения конфликтов. После включения этого программного компонента Git начинает сохранять набор пред- и постобразов успешных слияний. Обнаружив, что текущий конфликт точно напоминает какой-то из ранее решавшихся, система разрешает его аналогичным образом, не требуя вашего участия в этом процессе.

Этот программный компонент представлен в двух формах: в виде конфигурационного параметра и команды. Параметр `rerere.enabled` настолько удобен, что имеет смысл поместить его в `global config`:

```
$ git config --global rerere.enabled true
```

Теперь результат каждого слияния, сопровождающегося разрешением конфликтов, будет записываться в кэш.

При необходимости этим кэшем можно воспользоваться при помощи команды `git rerere`. После этой команды Git проверяет базу данных решений, пытаясь найти там совпадение с текущим конфликтом слияния и решить его (впрочем, после присвоения параметру `rerere.enabled` значения `true` это делается автоматически). Существуют и дополнительные варианты команды, позволяющие посмотреть, что именно будет записываться, убрать отдельные решения из кэша или очистить кэш целиком.

Идентификация устойчивых версий

Решив выпустить устойчивую версию, вы, скорее всего, захотите присвоить ей тег, чтобы в будущем его можно было легко в любой момент восстановить. Если вы предпочитаете создать новый тег как владелец версии, процедура будет выглядеть примерно 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 подписываете свои теги, может возникнуть проблема с распространением используемого для этой цели открытого PGP-ключа. Лица, ответственные за поддержку

Git-проекта, решили эту проблему, добавив этот открытый ключ в репозиторий в виде массива двоичных данных и установив тег, указывающий непосредственно на эту информацию. Если вы хотите пойти по данному пути, первым делом определите, какой ключ вам нужен, запустив команду `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]
```

Выбранный ключ можно вставить непосредственно в базу данных Git-проекта, экспортировав его и передав команде `git hash-object`, которая создаст из содержимого этого ключа новый массив двоичных данных и вернет вам его контрольную сумму SHA-1:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Теперь, когда в системе Git хранится содержимое этого ключа, легко можно создать указывающий на него тег, воспользовавшись данным вам командой `git hash-object` значением контрольной суммы SHA-1:

```
$ git tag -a maintainer-gpg-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Команда `git push --tags` делает тег `maintainer-gpg-pub` общедоступным. Для проверки тега любой пользователь может напрямую импортировать ваш PGP-ключ, взяв информацию непосредственно из базы данных и вставив ее в программу GPG:

```
$ git show maintainer-gpg-pub | gpg --import
```

Этим ключом пользователи могут проверять все подписанные вами теги. Если в сообщении тега добавлена инструкция, команда `git show <тег>` выдаст конечному пользователю инструкцию по проверке тегов.

Генерация номера сборки

Поскольку равномерно возрастающие номера, такие как v123, в Git для коммитов не используются, присвоить удобное для восприятия имя позволяет команда `git describe`. Она возвращает имя ближайшего тега с количеством сделанных поверх этого тега коммитов и частичной контрольной суммой SHA-1 описываемого коммита:

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

Это дает возможность присвоить снимку состояния проекта или сборке понятное для пользователей имя. По сути, при сборке Git-кода из кода, клонированного из

Git-репозитория, нечто подобное возвращает команда `git --version`. При описании коммита, которому был в явном виде присвоен тег, команда возвращает имя этого тега.

Команда `git describe` хорошо работает в случае тегов, снабженных комментариями (то есть созданных с флагом `-a` или `-s`), поэтому именно таким способом нужно создавать теги для устойчивых версий. Ведь при помощи команды `git describe` можно удостовериться в корректности имени тега. К этой строке также можно применять команды `checkout` и `show`, но в будущем они могут перестать работать из-за сокращенного значения SHA-1. Например, в ядре для обеспечения уникальности объектов с недавнего времени требуется не в 8, а 10 символов SHA-1, поэтому старые имена, сгенерированные командой `git describe`, стали недействительными.

Подготовка устойчивой версии

Итак, вы готовы выпустить свою сборку как основную версию. Возможно, вы захотите создать архив последнего состояния кода для тех пользователей, у которых отсутствует система Git. Это делается командой `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 последний снимок состояния вашего проекта в папке `project`. Можно создать и zip-архив, добавив команде `git archive` параметр `--format=zip`:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

В итоге вы получите tarball- и zip-архивы с устойчивой версией вашего проекта, которые можно загрузить на сайт или отправить пользователям по электронной почте.

Команда `shortlog`

Чтобы отправить коллегам по своему списку рассылки сообщение с информацией о последних новостях вашего проекта, используется команда `git shortlog`, позволяющая создать нечто вроде журнала, описывающего, что было добавлено к проекту после выхода последней версии или последней рассылки. Журнал включает в себя все коммиты в указанном вами диапазоне; к примеру, следующая команда возвращает перечень коммитов, сделанных с момента выхода версии v1.0.1:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
    Add support for annotated tags to Grit::Tag
```

```
Add packed-refs annotated tag support.  
Add Grit::Commit#to_patch  
Update version and History.txt  
Remove stray `puts`  
Make ls_tree ignore nils
```

```
Tom Preston-Werner (4):  
  fix dates in history  
  dynamic version method  
  Version bump to 1.0.2  
  Regenerated gemspec for version 1.0.2
```

То есть мы получаем аккуратную сводку всех коммитов, начиная с версии v1.0.1, сгруппированных по авторам, которую можно разослать своим подписчикам.

Заключение

К этому моменту вы должны уметь не только вносить свой вклад в чужие Git-проекты, но и поддерживать собственный проект, интегрируя в него изменения, присылаемые другими участниками. Фактически вас можно поздравить с вхождением в клан опытных Git-разработчиков! В следующей главе вы научитесь пользоваться самым крупным и самым популярным Git-хостингом, который называется GitHub.

6 GitHub

Сайт GitHub является крупнейшим на сегодняшний день хостингом для Git-репозиторий и центральным местом сотрудничества миллионов разработчиков и проектов. Именно на этом сайте располагается подавляющая часть всех Git-репозиторий, а многие проекты с открытым исходным кодом используют его для хостинга Git, отслеживания ошибок, рецензирования кода и других вещей. И хотя этот сайт пока не является частью открытого Git-проекта, велики шансы, что когда-нибудь вам потребуется воспользоваться им в рабочих целях.

В этой главе мы поговорим о том, как добиться максимально эффективной работы с GitHub. Вы узнаете, как создать и поддерживать свою учетную запись, научитесь создавать и использовать Git-репозитории, познакомитесь с общепринятыми схемами внесения изменений в чужие проекты и принятия изменений, присланных в ваш проект, освоите программный интерфейс для GitHub и получите множество советов, позволяющих изрядно облегчить жизнь.

Если вы не собираетесь использовать GitHub для размещения собственных проектов или сотрудничать в проектах, использующих этот хостинг, можете сразу перейти к чтению главы 7.

ИЗМЕНЕНИЯ ИНТЕРФЕЙСА

Следует заметить, что интерфейс сайта GitHub, как и интерфейс многих других активно развивающихся сайтов, со временем меняется, соответственно представленные на снимках элементы пользовательского интерфейса на момент чтения данной книги могут иметь немного другой вид. Но основной смысл выполняемых операций от этого не меняется. Более актуальную версию снимков экрана в некоторых случаях можно найти в онлайн-версии этой книги.

Настройка и конфигурирование учетной записи

Первым делом нужно создать бесплатную учетную запись. Для этого перейдите на страницу <https://github.com>, выберите свободное имя пользователя, укажите свой адрес электронной почты и пароль, после чего щелкните на большой зеленой кнопке Sign up for GitHub (рис. 6.1).

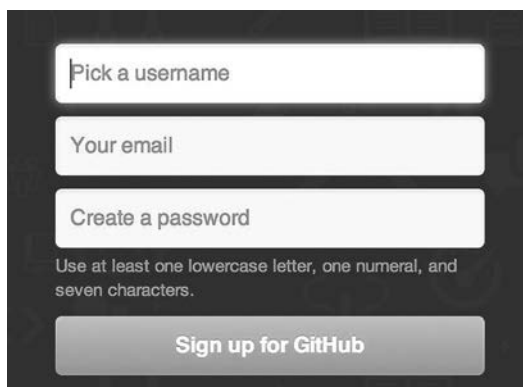
The image shows a registration form on a dark background. It consists of three white input fields stacked vertically. The first field is labeled 'Pick a username', the second 'Your email', and the third 'Create a password'. Below the password field, there is a small text requirement: 'Use at least one lowercase letter, one numeral, and seven characters.' At the bottom of the form is a large, rounded rectangular button with the text 'Sign up for GitHub'.

Рис. 6.1. Регистрационная форма на сайте GitHub

После этого вы попадете на страницу с тарифными планами, которую на данном этапе можно пропустить. Вам будет отправлено письмо для проверки указанного при регистрации адреса электронной почты. Выполните присланную в письме инструкцию. Как вы скоро убедитесь, это действительно необходимо.

ПРИМЕЧАНИЕ

Для бесплатных учетных записей сайт GitHub предоставляет всю функциональность, но ваши проекты будут полностью открытыми (у любого пользователя будет доступ на чтение). Платные учетные записи в GitHub дают возможность создавать различные варианты закрытых проектов, но их рассмотрение выходит за рамки темы данной книги.

Щелчок на расположенном в верхнем левом углу экрана логотипе, изображающем гибрид кота и осьминога (его называют осьмикот), откроет панель управления. Теперь все готово для работы с GitHub.

Доступ по протоколу SSH

С этого момента вы можете подключаться к Git-репозиториям, используя протокол `https:// protocol`. Для аутентификации достаточно указать имя пользователя и пароль. Впрочем, клонировать открытые проекты можно и без регистра-

ции — созданная вами учетная запись пригодится, когда вам потребуется выполнить ветвление какого-либо проекта и загрузить в новую ветку свои изменения.

Если же для удаленного доступа вы предпочитаете протокол SSH, нужно будет настроить открытый ключ. (Если такого ключа у вас пока нет, см. раздел «Создание открытого ключа SSH» в главе 4.) Перейдите к настройкам своей учетной записи, воспользовавшись расположенной в правом верхнем углу окна ссылкой (рис. 6.2).

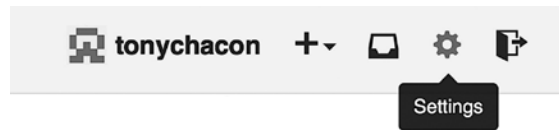


Рис. 6.2. Ссылка на страницу настройки учетной записи

Затем выберите на левой панели раздел SSH Keys (рис. 6.3).

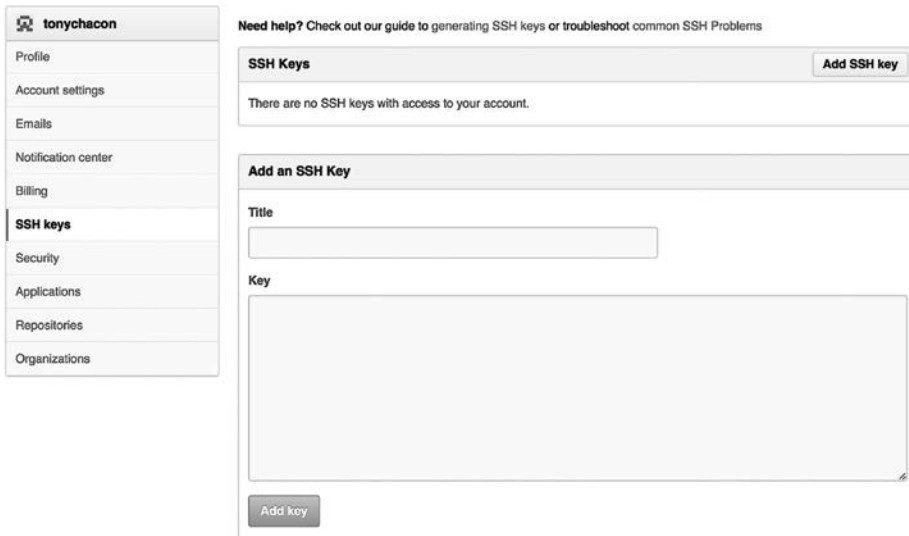


Рис. 6.3. Вход в раздел SSH keys

Щелкните на кнопке Add an SSH key, укажите имя ключа, скопируйте и вставьте сам открытый ключ из файла `~/.ssh/id_rsa.pub` (у вас этот файл может называться по-другому) и щелкните на кнопке Add key.

ПРИМЕЧАНИЕ

Присваивайте SSH-ключам имена, которые вы в состоянии запомнить. Выбирайте разные имена (например, «My Laptop» или «Work Account»), чтобы в будущем, если вдруг возникнет необходимость убрать ключ, можно было безошибочно найти нужный.

Аватар

Теперь при желании можно заменить сгенерированный аватар собственным изображением. На левой панели выберите вкладку Profile и щелкните на кнопке Upload new picture (рис. 6.4).

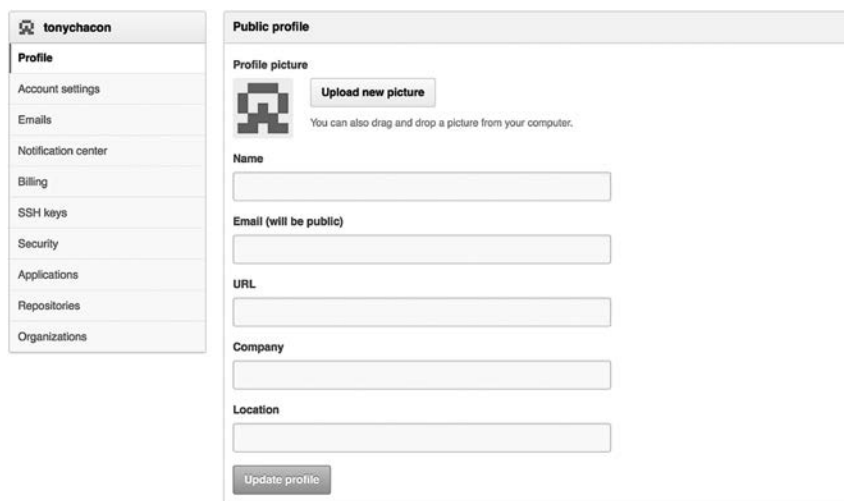


Рис. 6.4. Кнопка перехода на вкладку Profile

Мы выбрали на своем жестком диске копию логотипа Git. При загрузке из нее можно вырезать произвольный фрагмент (рис. 6.5).

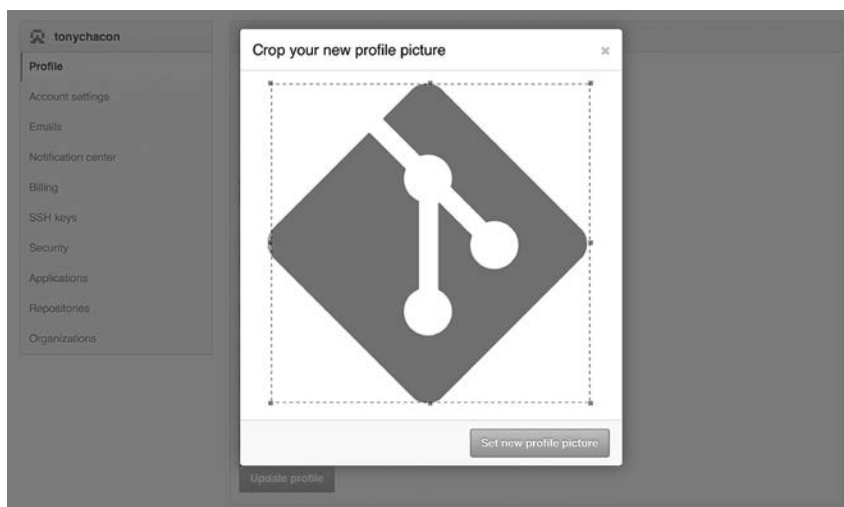


Рис. 6.5. Обрезка выбранного аватара

Теперь при любых ваших действиях на сайте рядом с вашим именем пользователя будет фигурировать выбранный аватар.

Если вы зарегистрированы в популярной службе Gravatar (она часто используется для учетных записей в WordPress), по умолчанию будет отображаться аватар оттуда, и данный этап вы можете просто пропустить.

Адреса электронной почты

GitHub проецирует ваши Git-коммиты на адрес электронной почты. Если в коммитах фигурируют разные адреса, все их нужно добавить в раздел **Emails** панели управления, чтобы сайт GitHub корректно выполнял проецирование.

На рис. 6.6 показан перечень возможных ситуаций. Верхний адрес проверен и установлен в качестве основного (**primary**), то есть именно на него вы будете получать все уведомления и подтверждения. Второй адрес проверен и поэтому тоже может быть установлен в качестве основного, если вдруг у вас возникнет желание переключиться на него. Проверка последнего адреса пока не произведена, поэтому в качестве основного он фигурировать не может. Обнаружив любой из этих адресов в сообщениях фиксации в любом своем репозитории, GitHub сразу же свяжет их с вашей учетной записью.

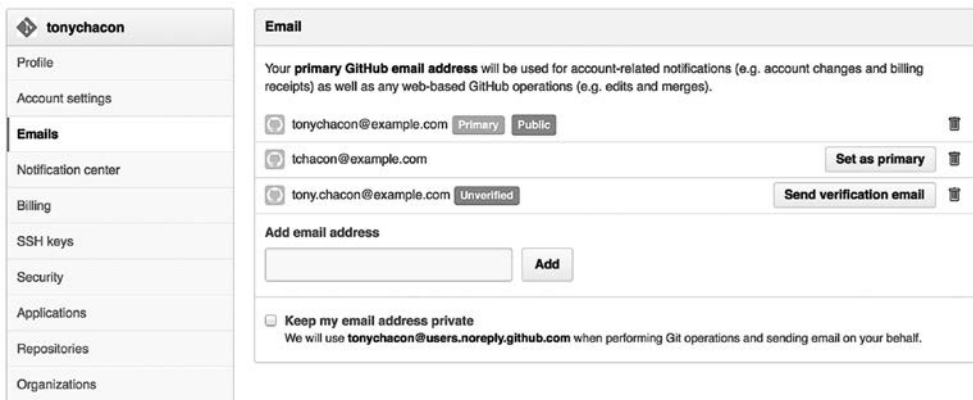


Рис. 6.6. Добавление адреса электронной почты

Аутентификация по двум признакам

Наконец, в качестве дополнительной меры безопасности можно настроить аутентификацию по двум признакам (Two-factor Authentication, 2FA). Этот механизм

в последнее время набирает все большую популярность как средство снизить риск взлома учетной записи в случае, если ваш пароль каким-то образом попадет в руки злоумышленника. После включения этого механизма GitHub начинает запрашивать у вас аутентификацию двумя способами, в результате даже взлом пароля не дает атакующему доступа к вашей учетной записи.

Аутентификация по двум признакам включается в разделе **Security** настроек учетной записи (рис. 6.7).

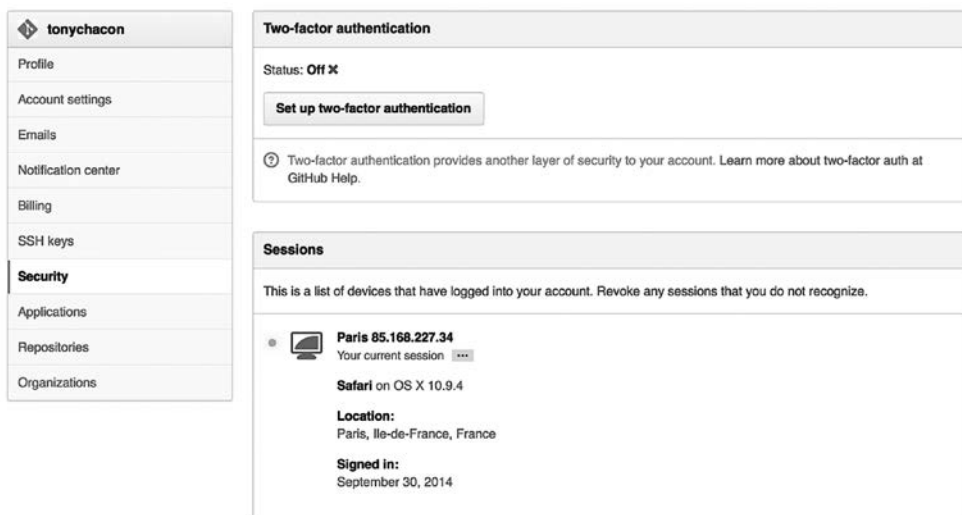


Рис. 6.7. Включение режима 2FA на вкладке Security

Щелчок на кнопке **Set up two-factor authentication** открывает конфигурационную страницу, на которой можно выбрать мобильное приложение для генерации вторичного кода проверки (однократный пароль с временным критерием). Кроме того, можно сделать так, чтобы при попытке авторизации сайт GitHub отправлял вам SMS с кодом.

После того как вы выберете метод и выполните настройку режима 2FA, безопасность вашей учетной записи возрастет, и при переходе в нее вам потребуется вводить не только пароль, но и дополнительный код.

Содействие проекту

Теперь, когда у нас есть настроенная учетная запись, поговорим о вещах, которые могут пригодиться тем, кто хочет внести свой вклад в существующие проекты.

Ветвления проектов

Для участия в чужом проекте, где у вас нет права на запись, нужно создать ветку (fork) этого проекта. То есть GitHub генерирует отдельную копию проекта, находящуюся в вашем пространстве имен, и предоставляет вам доступ на запись туда.

ПРИМЕЧАНИЕ

Исторически термин ветвление (fork) имел негативный оттенок, так как означал, что кто-то поменял направление проекта с открытым исходным кодом, например, создав конкурирующий проект и переманив в него участников. Однако в контексте GitHub этим термином обозначается клон проекта, находящийся в вашем пространстве имен и позволяющий открыто вносить изменения в основную версию проекта.

Такой подход избавляет владельцев проектов от необходимости добавлять пользователей, желающих принять участие в работе, и предоставлять им доступ на запись. Любой человек может создать собственную ветку, внести туда изменения и отправить их в исходный репозиторий путем запроса на включение. Такие запросы подробно рассматриваются далее в этой главе. Они открывают цепочки обсуждения с анализом кода, в которых владелец и участник проекта могут согласовывать изменения, пока они не устроят владельца и он не осуществит их слияние.

Для ветвления проекта зайдите на его страницу и щелкните на кнопке Fork в ее верхнем правом углу (рис. 6.8).



Через несколько секунд вы попадете на новую страницу проекта с доступной для редактирования копией кода.

Рис. 6.8. Кнопка Fork

Схема работы с GitHub

Сайт GitHub спроектирован под определенную схему работы, сконцентрированную вокруг запросов на включение. Она применяется как сплоченными командами, пользующимися общим репозиторием, так и распределенными группами и даже группами не знакомых друг с другом людей, добавляющих свою работу к проекту на десятках ветвлений. Эта схема организована по принципу тематических веток, с которыми вы познакомились в главе 3.

Вот как это работает в общем случае:

1. На основе ветки **master** создается тематическая ветка.
2. Делается ряд коммитов с целью внести улучшения в проект.
3. Ветвь отсылается в ваш GitHub-проект.
4. На сайте GitHub открывается запрос на включение.

5. Предложенное изменение обсуждается и, возможно, подвергается редактированию.
6. Владелец проекта выполняет слияние или закрывает запрос на включение.

По сути, это рассмотренная в главе 5 схема работы при наличии диспетчера интеграции, но обсуждение и анализ кода осуществляются не по электронной почте, а посредством GitHub-инструментария.

Рассмотрим работу этой схемы на примере.

Запрос на включение

В поисках кода для программируемого микроконтроллера Arduino Тони обнаруживает на странице <https://github.com/schacon/blink> замечательную программу (рис. 6.9).

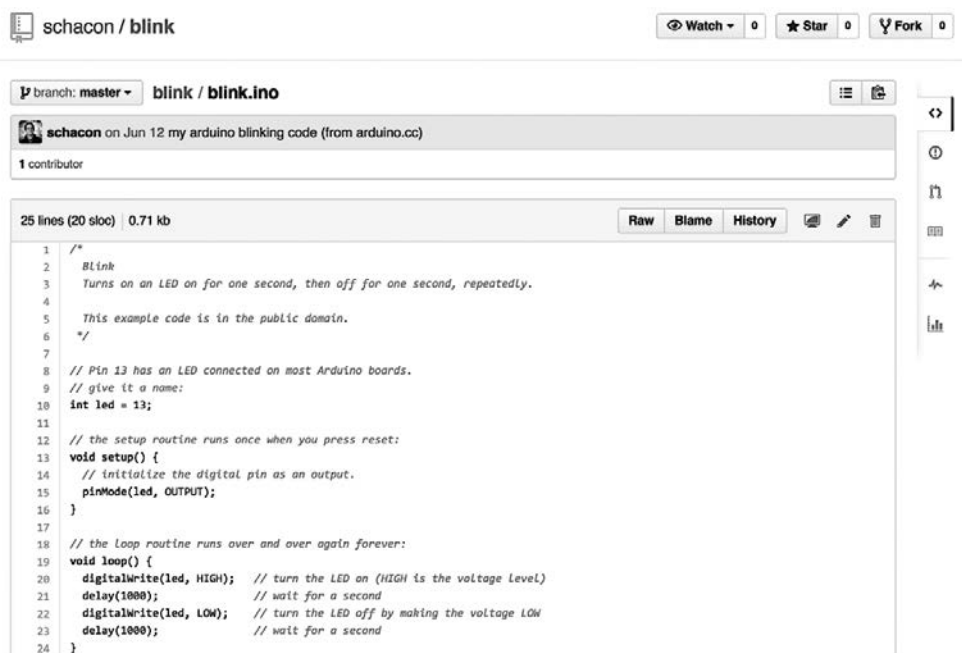


Рис. 6.9. Проект, в который мы хотим внести свой вклад

Единственной проблемой является слишком высокая скорость мигания. С нашей точки зрения, между изменениями состояния лучше ждать три секунды, а не одну. Поэтому отредактируем программу и отправим ее в проект как предлагаемое изменение.

Первым делом мы щелкаем на кнопке Fork, как описывалось в предыдущем разделе, для получения себе копии проекта. Имя пользователя в данном случае **tonychacon**, поэтому копия проекта оказывается по адресу <https://github.com/tonychacon/blink>, где мы и сможем заняться ее редактированием. Мы локально ее клонируем, создадим тематическую ветку, внесем в код изменения и, наконец, отправим исправленный код обратно на сайт GitHub.

```
$ git clone https://github.com/tonychacon/blink (1)
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink (2)
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino (3)

$ git diff --word-diff (4)
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
    digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
    [-delay(1000);-]{+delay(3000);+} // wait for a second
    digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
    [-delay(1000);-]{+delay(3000);+} // wait for a second
}

$ git commit -a -m 'three seconds is better' (5)
[master 5ca509d] three seconds is better
1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink (6)
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch] slow-blink -> slow-blink
```

Вся процедура выглядит так:

1. Локальное клонирование нашей ветки проекта.
2. Создание наглядной тематической ветки.
3. Внесение изменений в код.
4. Проверка корректности изменений.

5. Фиксация наших изменений в тематической ветке.
6. Отправка новой тематической ветки в нашу ветку на сайте GitHub.

Вернувшись к нашей ветке на сайте GitHub, мы увидим, что система получила нашу тематическую ветку и предоставила большую зеленую кнопку для перехода к нашим изменениям и открытия запроса на включение в исходный проект (рис. 6.10).

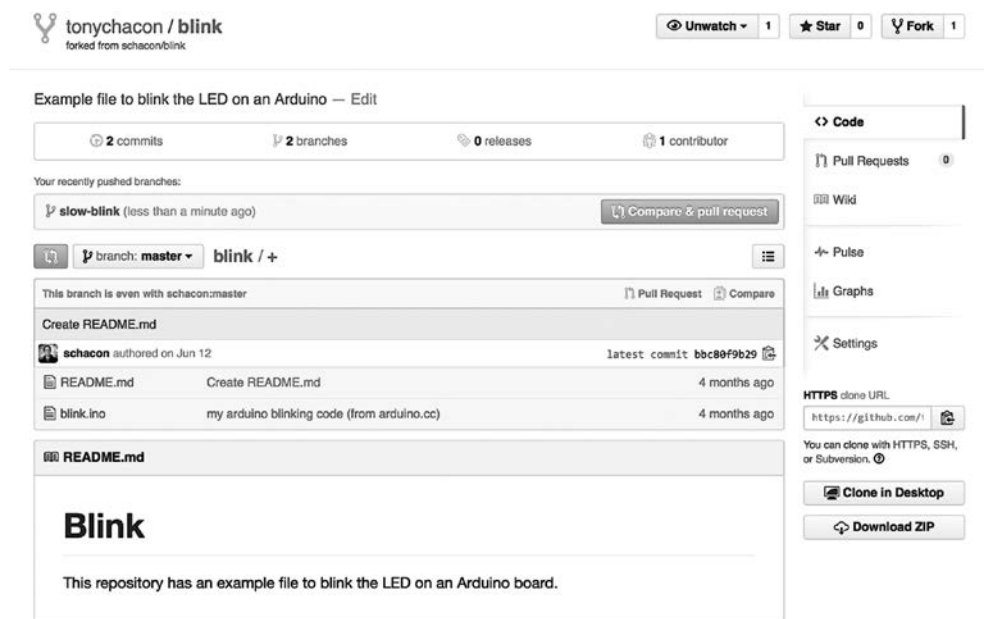


Рис. 6.10. Кнопка Pull Request

Кроме того, вы можете перейти на страницу с перечнем веток по адресу <https://github.com/<имя пользователя>/<название проекта>/branches>, найти там нужную ветку и открыть запрос на включение оттуда.

Щелчок на этой зеленой кнопке откроет экран, где изменению, для которого мы собираемся создать запрос на включение, можно присвоить заголовок и снабдить его описанием, чтобы у владельца проекта появился стимул с ним ознакомиться. Как правило, имеет смысл приложить усилия и сделать описание как можно более содержательным, чтобы сразу можно было понять, зачем предлагается данное изменение и почему его желательно принять (рис. 6.11).

Кроме того, мы увидим список коммитов в нашей тематической ветке, которые «опережают» ветку **master** (в данном случае есть всего один такой коммит) и унифицированное описание всех изменений, к которым приведет слияние этой ветки с проектом.

После щелчка на кнопке Create Pull Request владелец проекта, для которого вы создали ответвление, получит уведомление о предлагаемом изменении и ссылку на страницу с полной информацией о нем.

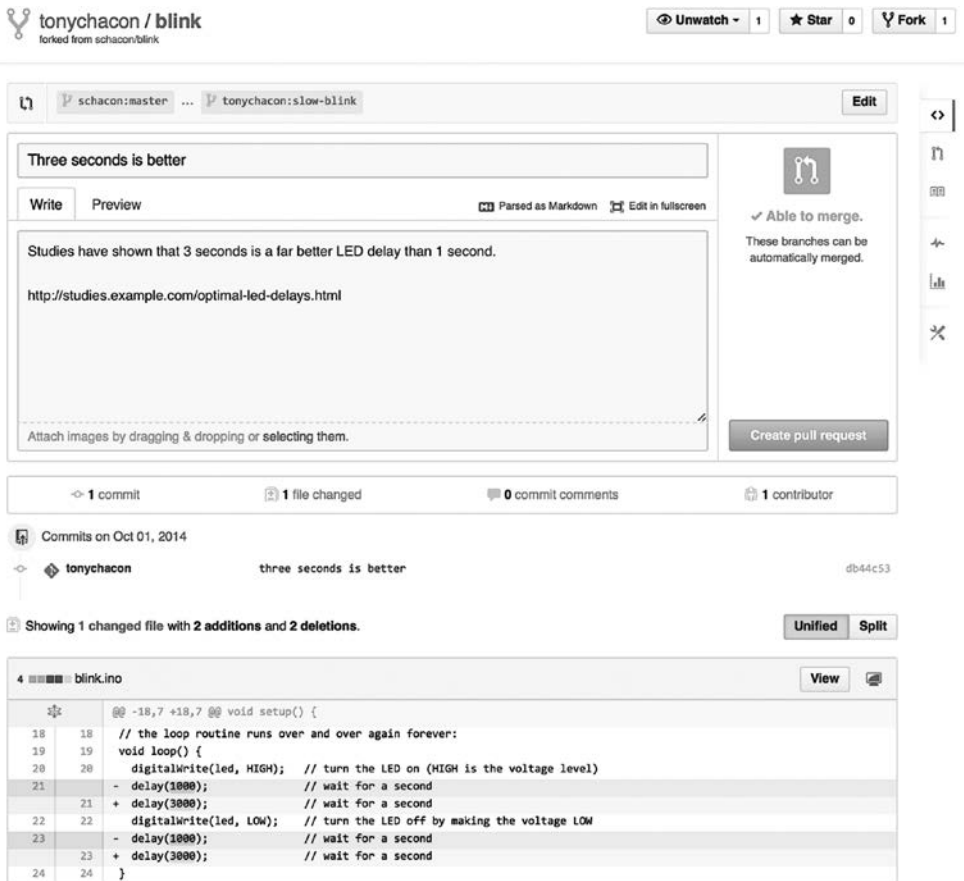


Рис. 6.11. Страница создания запроса на включение

ПРИМЕЧАНИЕ

Как правило, запросы на включение используются в открытых проектах, в рамках которых участники предлагают полностью готовые к внедрению решения, но прибегают к ним и во внутренних проектах **в начале** цикла разработки. Так как добавлять информацию в тематическую ветку можно и **после** открытия запроса на включение, запрос часто открывается на ранней стадии и используется как способ выполнения циклов работы одной командой.

Стадии обработки запроса на включение

Теперь владелец проекта может ознакомиться с предложенным изменением и выполнить его слияние, отвергнуть его или оставить к нему комментарий. Предположим, общая идея ему понравилась, но ему бы хотелось, чтобы в выключенном состоянии лампочка находилась дольше, чем во включенном.

В рабочей схеме, рассмотренной в главе 5, обсуждение этого аспекта велось бы по электронной почте, но GitHub позволяет делать это непосредственно на сайте. Владелец проекта может изучить вносимые поправки, выделить строку кода и написать для нее комментарий (рис. 6.12).

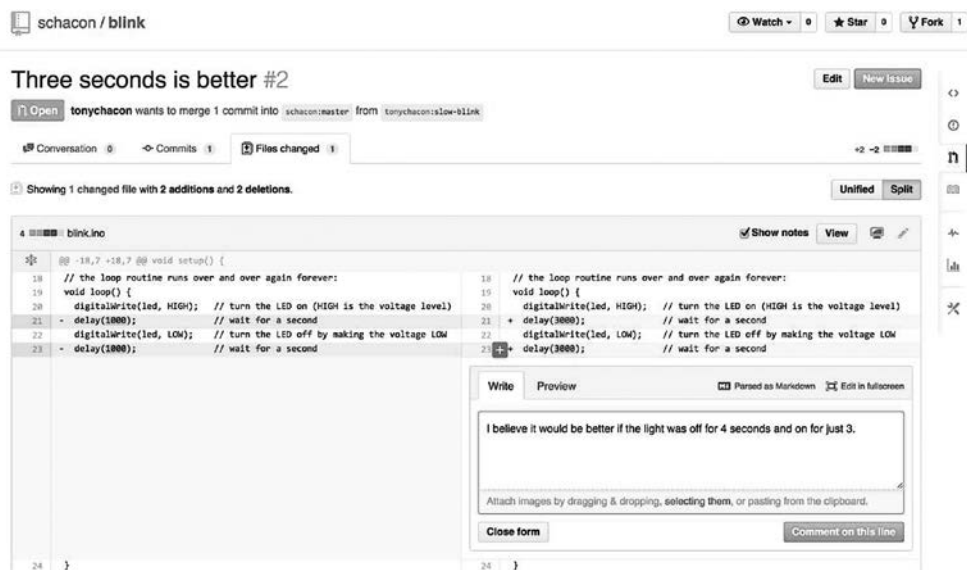


Рис. 6.12. Комментарий к строке кода в запросе на включение

Как только владелец добавит этот комментарий, пользователь, открывший запрос на включение (как и все остальные пользователи, работающие с данным репозиторием), получит уведомление. О том, как настроить уведомления, мы поговорим чуть позже, но если у Тони соответствующий режим включен, он получит по электронной почте сообщение, представленное на рис. 6.13.

Все пользователи могут оставлять комментарии к запросу на включение. На рис. 6.14 показана ситуация, когда владелец проекта сначала комментирует строку кода, а затем оставляет общий комментарий в разделе обсуждений. Обратите внимание, что в обсуждение включены также комментарии к коду.

Теперь предложивший изменение участник видит, что следует сделать, чтобы это изменение было принято. К счастью, это тоже очень просто. Если в случае обмена

данными по электронной почте вам пришлось бы заново создавать архив и повторно отправлять его в рассылку, GitHub позволяет выполнить коммит в тематической ветке и отправить его.

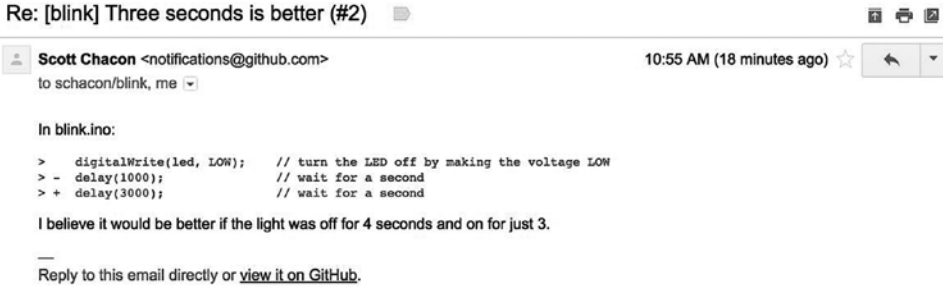


Рис. 6.13. Комментарий, отправленный как уведомление по электронной почте

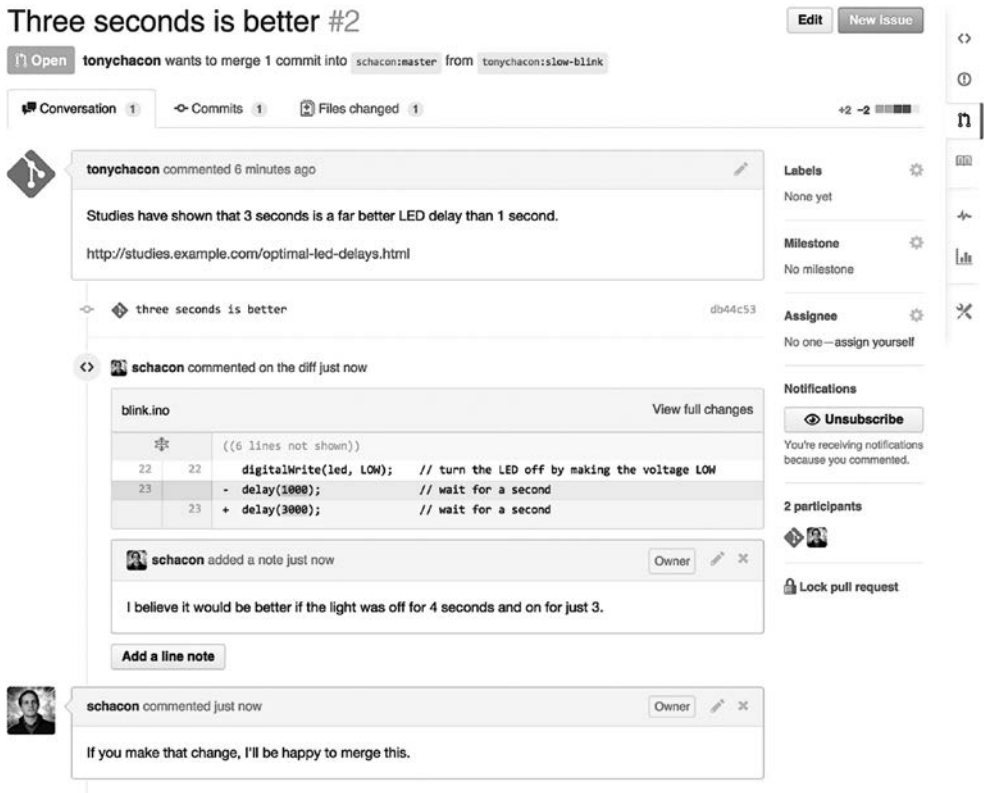


Рис. 6.14. Страница обсуждения запроса на включение

После этого владелец проекта снова получит уведомление, а перейдя на страницу, увидит, что проблема решена. Более того, так как строка кода, которой был посвящен комментарий, отредактирована, GitHub сворачивает утратившую актуальность ветку (рис. 6.15).

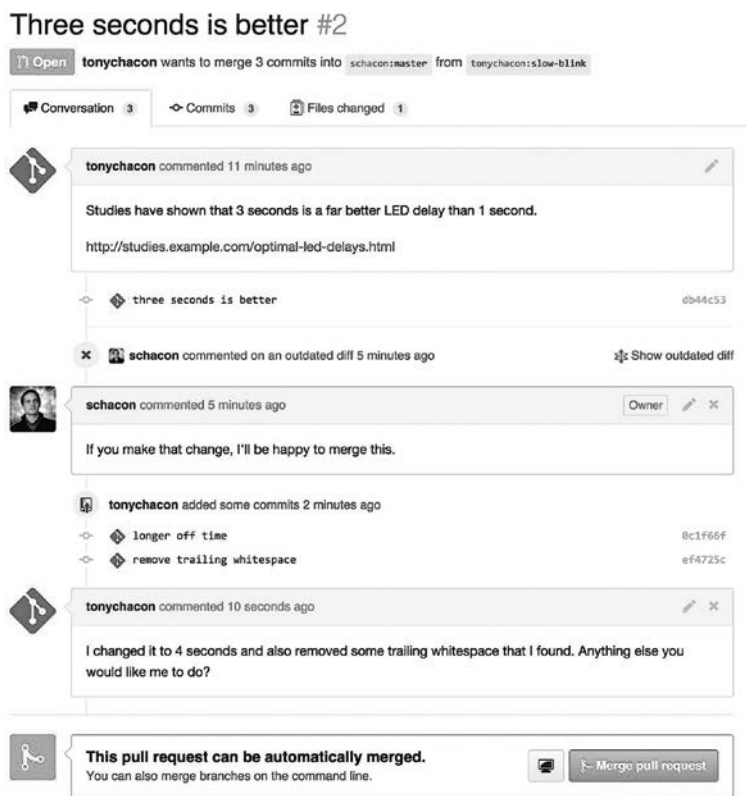


Рис. 6.15. Финал запроса на включение

Имейте в виду, что на вкладке **Files Changed** этого запроса на включение вы найдете описание изменений в унифицированном формате, то есть совокупные изменения, которые произойдут с главной веткой после того, как туда будет слито содержимое тематической ветки. Фактически вам покажут результат действия команды `git diff master...<имя ветки>` для ветки, на основе которой был сформирован данный запрос на включение. Подробно эта команда рассматривалась в разделе «Просмотр вносимых изменений» главы 5.

Кроме того, обращаем ваше внимание, что GitHub проверяет, корректно ли происходит слияние изменений, предлагаемых в запросе, и предоставляет кнопку для выполнения этой операции на сервере. Эта кнопка появляется только при наличии у вас доступа на запись в репозиторий и возможности аккуратного слияния. После

щелчка на ней GitHub выполняет слияние без перемотки. Даже если перемотка *возможна*, будет создан только коммит слияния.

При желании вы можете извлечь данные из ветки и выполнить слияние локально. Когда вы сольете эту ветку в свою ветку **master** и отправите ее на сервер GitHub, запрос на включение автоматически закрывается.

Так выглядит базовая рабочая схема большинства проектов на сайте GitHub. Создаются тематические ветки, для них открываются запросы на включение, затем следует обсуждение, возможно, в ветке выполняется дополнительная работа, и в конечном счете запрос либо закрывается, либо подвергается слиянию.

НЕ ТОЛЬКО ВЕТВЛЕНИЯ

Имейте в виду, что запрос на включение может быть открыт для двух веток одного репозитория. Если вы работаете над каким-то программным компонентом вместе с коллегой и у вас обоих есть доступ на запись, можно отправить тематическую ветку в репозиторий и открыть запрос на ее включение в ветку **master** этого же проекта, чтобы инициировать анализ кода и процесс обсуждения. Как видите, ветвления не требуется.

Более сложные запросы на включение

Теперь, когда вы знаете базовую схему участия в проектах на сайте GitHub, рассмотрим интересные аспекты применения запросов на включение, позволяющие повысить эффективность вашей работы.

Запросы на включение как исправления

Важно понимать, что в большинстве случаев никто не воспринимает запросы на включение как очередь ценных исправлений, которые следует применять строго в определенном порядке, как это бывает в проектах, где наборы исправлений присылают по электронной почте. Для большинства GitHub-проектов ветки с запросами на включение — это повод многократно обсудить предлагаемые изменения, получив в конце унифицированный код, который применяется путем слияния.

Это важное отличие, ведь, как правило, изменение предлагается до того, как код можно будет считать идеальным, — для вкладов, осуществляемых через списки расылки, это практически невероятная ситуация. Однако такой подход способствует возникновению обсуждения на ранних стадиях работы, и появившееся в итоге корректное решение представляет собой в большей степени плод коллективных усилий. Когда появляется код с запросом на включение, а владелец предлагает внести в него изменения, последовательность исправлений не перематывается, вместо этого добавление отправляется в ветку как новый коммит, двигая обсуждение вперед, но в контексте предыдущей работы.

К примеру, вернемся к рис. 6.15. Обратите внимание, что участник не перемещает свой коммит, посылая еще один запрос на включение. Он добавляет новые коммиты и отправляет их в существующую ветку. При таком подходе, вернувшись в будущем к данному запросу на включение, вы легко сможете понять, почему было принято то или иное решение. Щелчок на кнопке Merge приводит к появлению коммита слияния со ссылкой на запрос, что дает возможность при необходимости легко вернуться к исходному обсуждению.

Сохранение актуальности данных

Если ваш запрос на включение устаревает или по каким-то другим причинам не допускает чистого слияния, ситуацию следует исправить, чтобы владелец проекта смог без проблем добавить предложенные вами изменения. GitHub проверяет этот аспект и в нижней части каждого запроса на включения сообщает, можно ли без проблем осуществить слияние.

Если вы видите такую запись, как на рис. 6.16, значит, в ветку следует внести исправления, чтобы владельцу проекта не пришлось делать дополнительную работу. Как только конфликт слияния будет разрешен, цвет рамки изменится на зеленый.



Рис. 6.16. Запрос на включение, не допускающий аккуратного слияния

Это можно сделать двумя способами. Во-первых, переместив свою ветку на вершину целевой ветки (как правило, это ветка `master` репозитория, послужившего основой вашего ветвления), во-вторых, слив содержимое целевой ветки в вашу ветку.

Большинство использующих GitHub разработчиков выбирает второй вариант по причинам, изложенным в предыдущем разделе. Значение имеют только история и финальное слияние, а перемещение всего лишь делает вашу историю немного чище, являясь при этом операцией *намного более* сложной и способствующей ошибкам.

Если вы хотите выполнить слияние в целевую ветку, чтобы сделать ваш запрос на включение допускающим слияние, добавьте себе исходный репозиторий в качестве удаленного, извлеките оттуда данные, слейте содержимое основной ветки этого репозитория в вашу тематическую ветку, разрешите конфликты и, наконец, верните все в ветку, для которой открывался запрос на включение.

Предположим, что в рассмотренный ранее пример «topuchason» автор внес изменение, которое стало причиной конфликта в запросе на включение. Рассмотрим наши дальнейшие действия.

```

$ 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

```

Вся процедура выглядит так:

1. Добавление исходного хранилища как удаленного с именем **upstream**.
2. Извлечение последних наработок из этого удаленного репозитория.
3. Слияние основной ветки в вашу тематическую ветку.
4. Решение возникающего конфликта.
5. Возвращение данных в ту же самую тематическую ветку.

После этого запрос на включение автоматически обновляется и производится проверка допустимости его слияния (рис. 6.17).

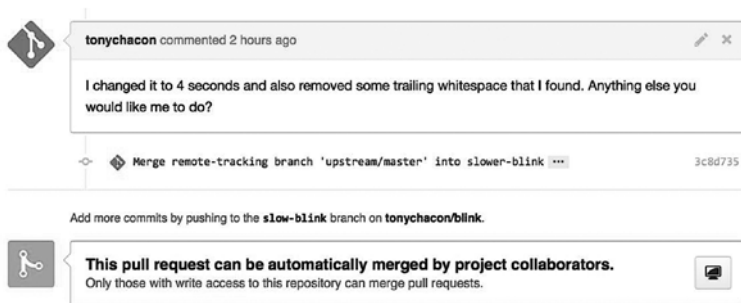


Рис. 6.17. Теперь слияние запроса на включение можно выполнить без проблем

Одним из замечательных аспектов Git является возможность выполнять эти действия в непрерывном режиме. В случае долгоживущего проекта слияние из целевой ветки можно выполнять снова и снова, просто разбираясь с конфликтами, появившимися после последнего слияния, что делает процесс полностью управляемым.

Если же вам очень хочется прибегнуть к перемещению, ни в коем случае не отправляйте данные в ветку, для которой уже открыт запрос на включение. Если другие пользователи извлекут оттуда информацию и начнут с ней работу, вы столкнетесь с проблемами, описанными в разделе «Риски, связанные с перемещением» главы 3. Вместо этого отправьте содержимое перемещенной ветки в новую ветку на сайте GitHub, откройте совершенно новый запрос на включение, ссылающийся на старый, а затем закройте оригинал.

Ссылки

У вас может возникнуть вопрос «Как сделать ссылку на старый запрос на включение?» Сослаться на другой элемент, расположенный практически в любом доступном для записи месте сайта GitHub, можно множеством способов.

Для начала рассмотрим перекрестные ссылки на другой запрос на включение или проблему. Всем запросам на включение и проблемам присваиваются уникальные в пределах проекта номера. К примеру, у вас не может быть одновременно запроса на включение под номером 3 и проблемы под номером 3. Для ссылки на чужой запрос или проблему в рамках одного проекта достаточно указать `#<номер>` в любом комментарии или описании. Если вы ссылаетесь на проблему или запрос в ветке репозитория, в котором вы в данный момент находитесь, укажите `имя пользователя#<номер>`. Запись вида `имя пользователя/репозиторий#<номер>` является ссылкой на запрос или проблему из чужого репозитория.

Рассмотрим пример. Предположим, мы переместили ветку из предыдущего примера, создали для нее новый запрос на включение и хотим добавить в него ссылку на старый вариант запроса. Еще нам требуется ссылка на проблему в ветке текущего репозитория и на еще одну проблему из чужого проекта. Заполним описание, как показано на рис. 6.18.

После отправки запроса на включение наши ссылки приобретут вид, показанный на рис. 6.19.

Обратите внимание, что полный URL-адрес, вставленный в комментарий, GitHub сокращает, оставляя только необходимую информацию.

Теперь если Тони закроет исходный запрос на включение, из-за того, что он упоминается в новом запросе, GitHub автоматически создаст событие уведомления (trackback) на временной шкале этого запроса. В итоге любой, кто зайдет на страницу этого запроса и увидит, что он закрыт, сможете легко перейти к замесившей его более новой версии. Пример такой ссылки показан на рис. 6.20.

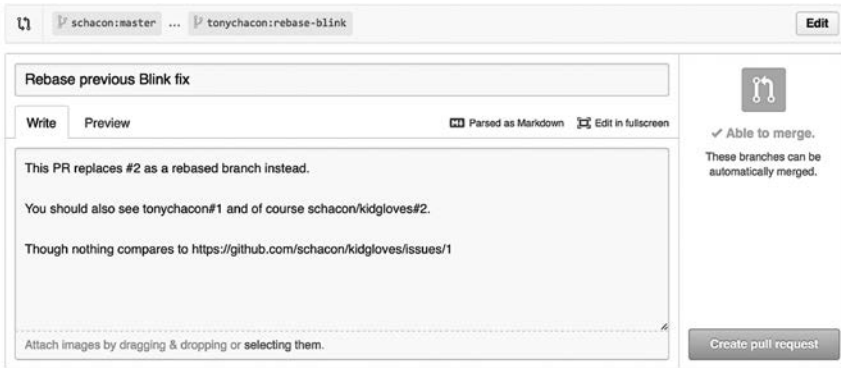


Рис. 6.18. Перекрестные ссылки в запросе на включение

Rebase previous Blink fix #4

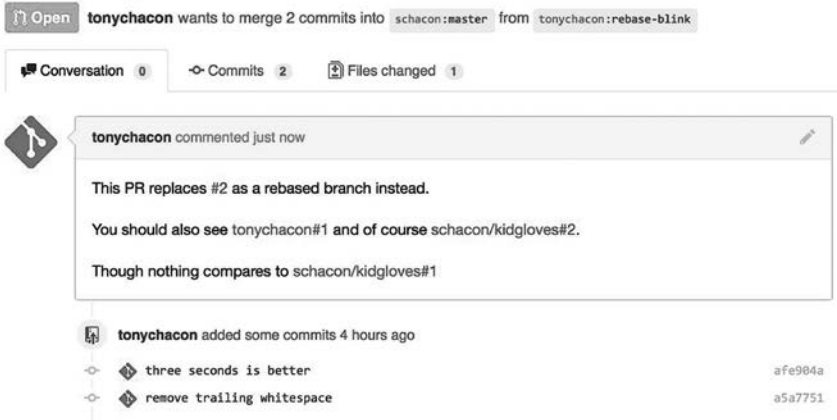


Рис. 6.19. Перекрестные ссылки в запросе на включение

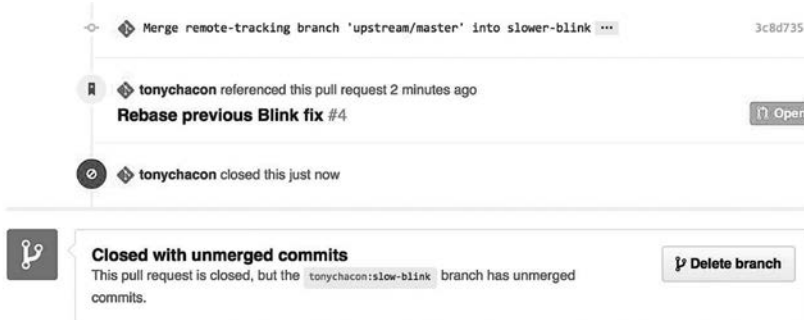


Рис. 6.20. Перекрестные ссылки в запросе на включение

Ссылаться можно не только на проблему по ее номеру, но и на коммит по его контрольной сумме SHA. Указывать нужно все 40 символов в этом случае GitHub, обнаружив контрольную сумму в комментарии, создает прямую ссылку на коммит. Ссылки на коммиты в ветвлениях и других репозиториях делаются таким же способом, как и ссылки на проблемы.

Язык разметки Markdown

Создание ссылок на проблемы — всего лишь первый шаг ко многим интересным вещам, которые можно проделывать практически с любым текстовым полем на сайте GitHub. В описаниях проблем и запросов на включение, комментариях, комментариях кода и пр. можно использовать так называемую GitHub-версию языка Markdown. Вы пишете обычный текст, который визуализируется в отформатированном виде, как показано на рис. 6.21.

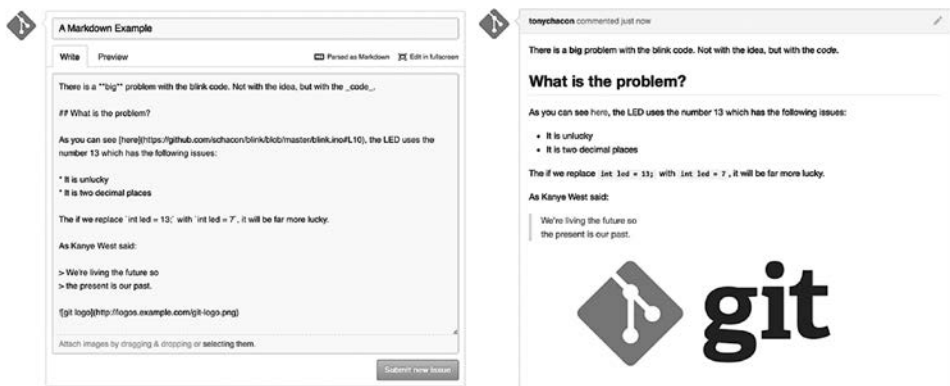


Рис. 6.21. Пример написания текста на языке Markdown с его последующей визуализацией

GitHub-версия языка Markdown

В GitHub-версии к базовому синтаксису языка Markdown добавляются новые элементы. Они пригодятся вам при подготовке как исчерпывающих комментариев к запросам на включение и проблемам, так и описаний.

Списки задач

Первой характерной для GitHub особенностью языка Markdown, особенно часто фигурирующей в запросах на включение, является список задач. Он представляет

собой набор флажков с перечнем вещей, которые требуется сделать. Вставка такого списка в проблему или запрос на включение обычно означает появление набора задач, без решения которых работу нельзя считать завершенной.

Вот пример списка, указывающего, что вам нужно написать код, все тесты к нему и снабдить его документацией:

- [X] Write the code
- [] Write all the tests
- [] Document the code

Вывод такого списка в составе запроса на включение или проблемы показано на рис. 6.22.

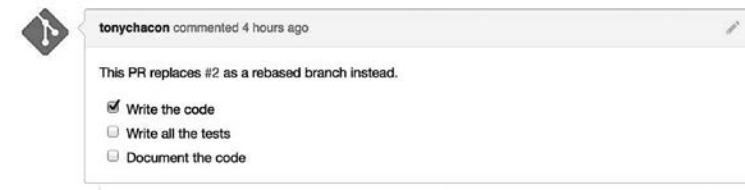


Рис. 6.22. Список задач, отображенный в виде комментария на языке Markdown

Такие списки часто вставляют в запросы на включение, указывая, что именно следует сделать, прежде чем запрос будет готов к слиянию. Что замечательно, для обновления комментария достаточно установить или сбросить флажок — редактировать сам Markdown-код не требуется.

Более того, GitHub ищет списки задач в ваших проблемах и запросах на включение и выводит их на страницах, где они фигурируют, в виде метаданных. К примеру, при просмотре страниц с перечнем всех запросов на включение можно увидеть результативность работы над каждым запросом (рис. 6.23). Подобное деление запросов на наборы задач помогает другим пользователям отслеживать ход работы в ветке.



Рис. 6.23. Отчет о решении задач из списка в перечне запросов на включение

Списки задач особенно полезны, когда вы открываете запрос на включение на ранней стадии и следите за тем, как продвигается реализация предложенного программного компонента.

Фрагменты кода

Еще к комментариям можно добавлять фрагменты кода. Это имеет смысл, например, чтобы показать, что вы собираетесь делать, перед *попыткой* реализовать задуманное в виде коммита в своей ветке. Также этой возможностью пользуются для демонстрации кода, над которым работа еще не ведется, но который может быть реализован в рамках текущего запроса на включение.

Для добавления фрагмента кода достаточно заключить его в обратные кавычки.

```
```java
for(int i=0 ; i < 5 ; i++)
{
 System.out.println("i is : " + i);
}
```
```

Если, как в приведенном примере, указать язык, на котором написан фрагмент, GitHub попытается выполнить подсветку синтаксиса. Результат визуализации этого фрагмента показан на рис. 6.24.



Рис. 6.24. Визуализация фрагмента кода, заключенного в кавычки

Цитирование

При ответе на небольшой фрагмент длинного комментария его можно выборочно вставить в свой текст, поставив в начале строки символ >. Этот прием применяется настолько часто, что для него существует даже клавиатурная комбинация. Нужно выделить текст, на который вы собираетесь отвечать, и нажать клавишу r. Выделенный текст немедленно будет оформлен в виде цитаты.

Цитата выглядит вот 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?
```

На рис. 6.25 демонстрируется ее вид после визуализации.

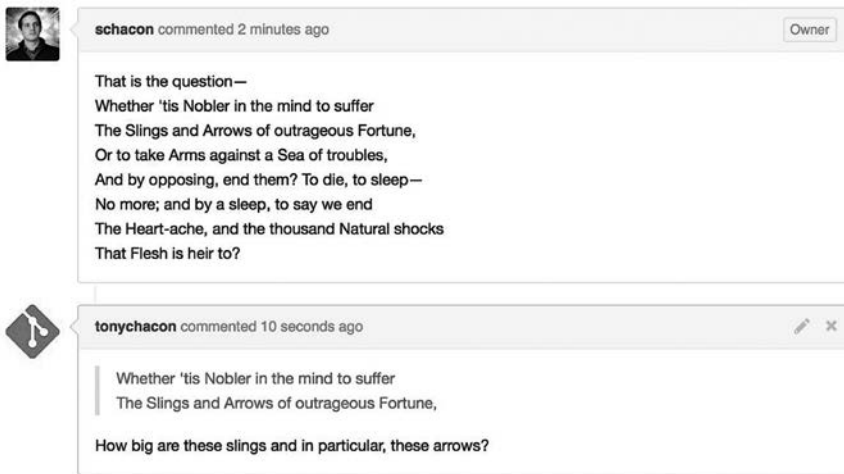


Рис. 6.25. Пример визуализации цитаты

Эмодзи

Наконец, в комментариях можно пользоваться эмодзи — языком идеограмм и смайликов. Элементы этого языка довольно широко применяются в комментариях, и вы часто будете встречать их на сайте GitHub. Существует даже руководство по этому языку. Например, если при наборе комментария ввести символ двоеточия, функция автозаполнения поможет найти подходящий вариант (рис. 6.26).

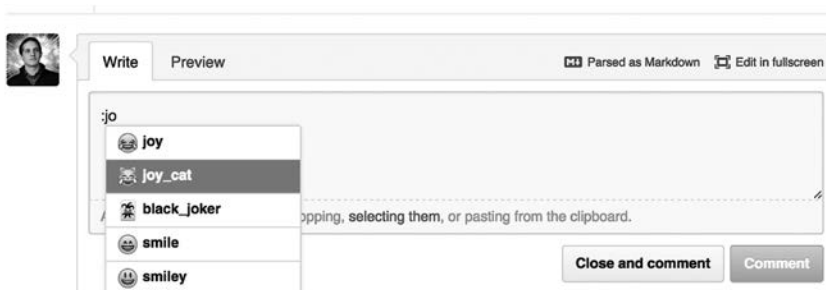


Рис. 6.26. Автозаполнение для языка эмодзи в действии

В комментариях выражения языка эмодзи имеют форму `:<имя>:`. К примеру, вы можете написать:

```
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:
```

После визуализации вы получите комментарий, показанный на рис. 6.27.



Рис. 6.27. Комментарий, перенасыщенный символами эмодзи

Особой смысловой нагрузки это не несет, просто добавляет немного эмоций в среду, в которой другим способом трудно выразить свои чувства.

ПРИМЕЧАНИЕ

Существует немало веб-служб, активно пользующихся символами языка эмодзи. Большая шпаргалка для выбора наиболее подходящего для выражения ваших чувств символа находится на странице <http://www.emoji-cheat-sheet.com/>.

Изображения

С технической точки зрения эта крайне полезная функция уже не относится к GitHub-версии языка Markdown. В дополнение к возможности добавлять в комментарии ссылки на изображения, для случаев, когда URL-адрес изображения бывает сложно найти и внедрить, GitHub позволяет осуществлять внедрение простым перетаскиванием картинок в текстовое поле (рис. 6.28).

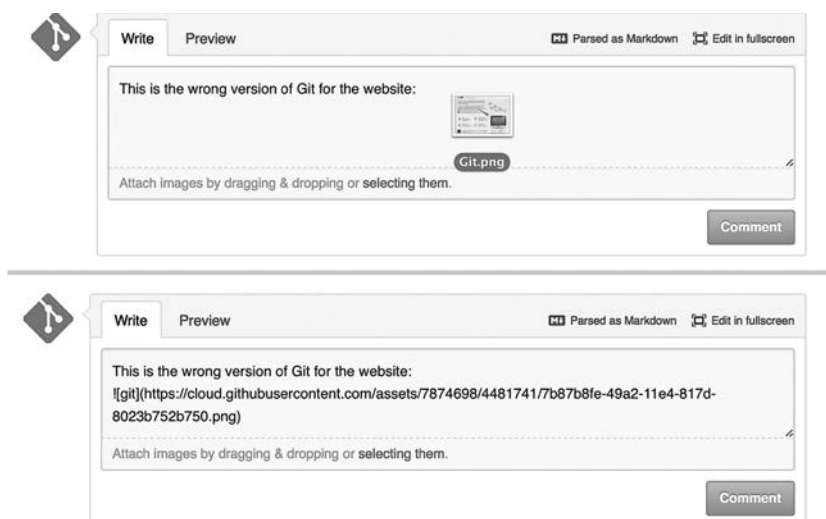


Рис. 6.28. Перетаскивание изображения в текстовое поле для его загрузки и последующего автоматического внедрения

Если вернуться к рис. 6.18, вы увидите справа над полем ввода текста небольшой значок `Parsed as Markdown`. Щелчок на нем вызывает шпаргалку с информацией обо всех операциях на языке Markdown в GitHub.

Сопровождение проекта

Теперь, когда вы знаете, каким образом вносить вклад в чужие проекты, посмотрим на ситуацию с другой стороны и поговорим о создании, сопровождении и администрировании собственных проектов.

Создание нового репозитория

Создадим новый репозиторий, чтобы получить возможность делиться кодом своего проекта с другими людьми. Первым делом щелкните на кнопке `New repository`, расположенной справа на панели инструментов (рис. 6.29), или на кнопке `+` панели инструментов рядом с вашим именем пользователя (рис. 6.30).

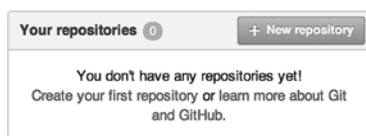


Рис. 6.29. Область Your repositories

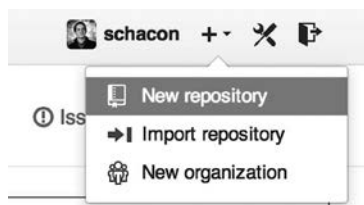


Рис. 6.30. Раскрывающийся список New repository

Появится форма создания нового репозитория (рис. 6.31).

Здесь в обязательном порядке следует указать только название проекта; все остальные поля заполняются по желанию. Достаточно щелчка на кнопке `Create Repository`, и у вас появится новый репозиторий на сайте GitHub с именем вида `<имя пользователя>/<название проекта>`.

Так как кода у вас пока нет, GitHub показывает инструкцию, касающуюся того, как создать совершенно новый Git-репозиторий или подключиться к существующему проекту Git. Мы не будем еще раз объяснять этот материал; если вам требуется освежить его в памяти, перечитайте главу 2.

URL-адрес размещенного на сайте GitHub проекта можно дать всем, с кем вы хотите поделиться результатами своего труда. Доступ к любому GitHub-проекту

осуществляется по протоколу HTTP в форме `https://github.com/<пользователь>/<название_проекта>` или по протоколу SSH в форме `git@github.com:<пользователь>/<название_проекта>`. Система Git может извлекать данные по любому из этих URL-адресов, но уровень доступа зависит от учетных данных человека, который осуществляет подключение.

The screenshot shows the GitHub interface for creating a new repository. At the top, there's a 'PUBLIC' label and a repository icon. The 'Owner' is set to 'ben' and the 'Repository name' is 'iOSApp'. Below this, a message says: 'Great repository names are short and memorable. Need inspiration? How about **drunken-dubstep**.' The 'Description (optional)' field contains 'iOS project for our mobile group'. Under the 'Visibility' section, the 'Public' radio button is selected, with the text 'Anyone can see this repository. You choose who can commit.' The 'Private' option is also visible with the text 'You choose who can see and commit to this repository.' There is a checkbox for 'Initialize this repository with a README' with the subtext 'This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.' Below this are two dropdown menus: 'Add .gitignore: None' and 'Add a license: None'. At the bottom is a 'Create repository' button.

Рис. 6.31. Форма создания нового репозитория

ПРИМЕЧАНИЕ

В случае открытого проекта зачастую лучше публиковать URL-адрес на базе HTTP, чтобы для его клонирования не требовалась учетная запись на сайте GitHub. Если вы дадите пользователю адрес доступа по протоколу SSH, ему потребуется создать учетную запись и загрузить свой ключ SSH. В случае же доступа по протоколу HTTP достаточно ввести URL-адрес в адресную строку браузера.

Добавление соавторов

Если вы работаете с другими пользователями, которым требуется право на запись коммитов, их следует добавить как соавторов. Если у Бена, Джеффа и Луизы есть учетные записи на сайте GitHub и вы хотите предоставить им доступ на запись в ваш репозиторий, добавьте их к своему проекту. После этого они смогут как читать материалы вашего проекта, так и вносить свой вклад в ваш Git-репозиторий.

Щелкните на кнопке **Settings** внизу расположенной справа панели (рис. 6.32).



Рис. 6.32. Ссылка на страницу настройки репозитория

Выберите в появившемся слева меню пункт **Collaborators**. После этого остается ввести в текстовое поле имя пользователя и щелкнуть на кнопке **Add collaborator**. Этот процесс можно повторять произвольное количество раз, предоставляя доступ всем, кому считаете нужным. Чтобы лишить кого-то права доступа, щелкните на крестике, расположенном справа от имени этого пользователя (рис. 6.33).

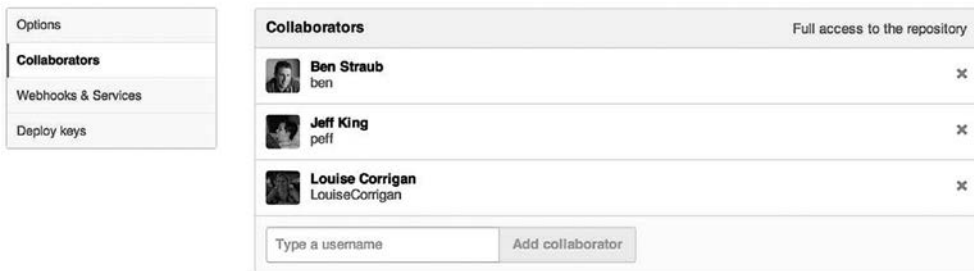


Рис. 6.33. Соавторы репозитория

Управление запросами на включение

Теперь, когда у вас есть проект с каким-то кодом, а возможно, и несколько соавторов, обладающих доступом на запись, рассмотрим процесс обработки запросов на включение.

Запросы на включение могут появиться как из ветки в ветвлении репозитория, так и из другой ветки вашего собственного репозитория. Единственное отличие состоит в том, что в первом случае это запросы от людей, не имеющих права на запись

в указанную ветку и в ваши ветки, а во втором их присылают люди, как правило, имеющие возможность делать запись в ветку.

Предположим, что вы — пользователь «tonychacon», который только что создал новый проект с кодом для системы Arduino и присвоил ему имя «fade».

Уведомления по электронной почте

Пусть какой-то пользователь внес изменения в ваш код и прислал вам запрос на включение. Вы должны получить сообщение электронной почты, уведомляющее вас о новом запросе. Примерно такое, как на рис. 6.34.

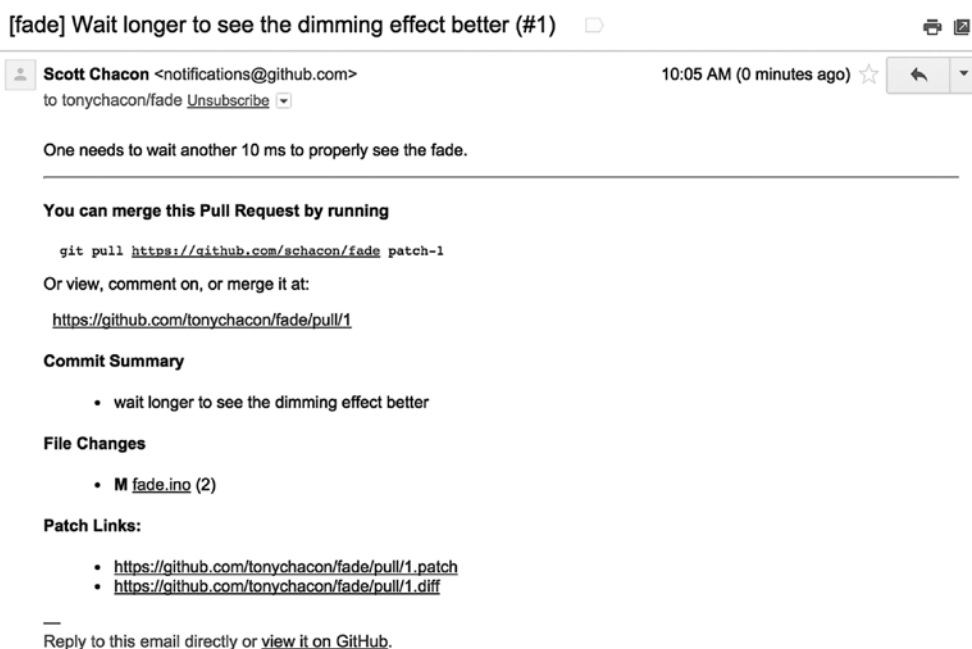


Рис. 6.34. Электронное письмо с уведомлением о новом запросе на включение

Следует отметить несколько моментов. В этом письме вы увидите сведения о внесенных изменениях — перечень отредактированных пользователем файлов с указанием степени их редактирования. Здесь же содержится ссылка с адресом запроса на включение на сайте GitHub. И несколько URL-адресов, которые можно использовать из командной строки.

Возможно, вы обратили внимание на строку `git pull <url> patch-1`. Это простой механизм извлечения данных из удаленной ветки без добавления удаленного репозитория. Этот процесс кратко рассматривался в разделе «Проверка удаленных

веток» главы 5. При желании вы можете создать тематическую ветку, перейти в нее и воспользоваться этой командой для добавления предложенных в запросе на включение изменений.

Другие интересные URL-адреса, `.diff` и `.patch`, как можно догадаться, ведут к унифицированным версиям различий и исправлений, связанных с запросом на включение. В принципе, наработки из запроса можно добавить вот такой командой:

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

Совместная работа над запросом на включение

Как следует из раздела «Схема работы с GitHub», теперь у вас поддерживается диалог с пользователем, открывшим запрос на включение. Вы можете прокомментировать отдельные строки кода, коммит в целом или весь запрос на включение, пользуясь GitHub-версией языка Markdown.

Каждый раз, когда кто-то оставляет комментарий к запросу на включение, вам приходит уведомление, чтобы ни одно связанное с запросом действие не осталось без вашего внимания. Уведомление содержит ссылку на запрос, кроме того, прокомментировать происходящее в цепочке обсуждения запроса можно, просто ответив на письмо (рис. 6.35).



Рис. 6.35. Ответы на электронную почту включаются в цепочку обсуждения

Когда код придет в нужное вам состояние и вы захотите добавить его к себе, его можно будет скачать и выполнить локальное слияние или воспользоваться уже знакомым вам синтаксисом `git pull <url> <ветка>`. Можно даже добавить ветвление как удаленный репозиторий и применить команды `fetch` и `merge`.

Если слияние не представляет проблемы, на сайте GitHub достаточно щелкнуть на кнопке **Merge**. При этом произойдет слияние без перемотки, то есть даже если перемотка возможна, у вас появится всего лишь коммит слияния. Это означает, что при каждом щелчке на кнопке **Merge** будет создаваться коммит слияния. Как показано на рис. 6.36, всю эту информацию GitHub предоставляет по щелчку на ссылке, ведущей к справочной информации.

При желании вы можете выполнить слияние. Или просто закройте запрос — тогда человеку, который его открыл, придет соответствующее уведомление.

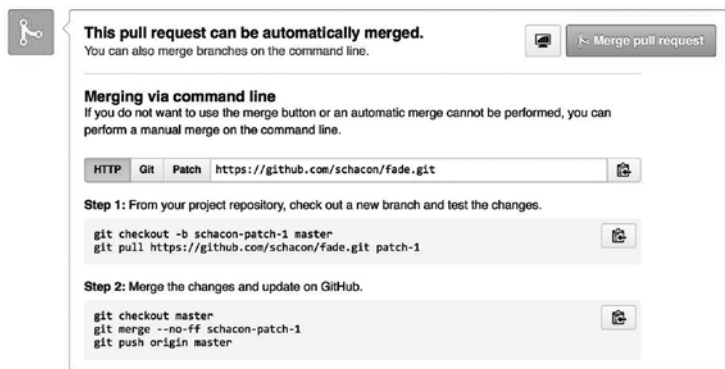


Рис. 6.36. Кнопка Merge и инструкции о том, как вручную выполнить слияние запроса на включение

Обращения к запросам на включение

Если вам приходится иметь дело с большим количеством запросов на включение и при этом вы не хотите добавлять множество удаленных репозиториях или каждый раз скачивать информацию, GitHub предлагает специализированный прием, который может быть вам крайне полезен, но детали реализации которого мы рассмотрим чуть позже.

Для репозитория GitHub визуализирует ветки запросов на включение как некие псевдоветки на сервере. По умолчанию их содержимое при клонировании не переносится, но присутствует в неявном виде и легко доступно.

Мы продемонстрируем это на примере низкоуровневой команды `ls-remote` (такие команды часто называют «служебными», подробно они рассматриваются в главе 10). При повседневной работе с Git эта команда не используется, но полезна, когда нам требуется посмотреть список всех ссылок на сервере.

Посмотрим, какой результат она даст в случае с репозиторием `blink`, с которым мы работали раньше. Мы получим список всех веток и тегов, а также других имеющих в репозитории ссылок.

```
$ 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

Именно такой результат вы получите, запустив из своего репозитория команду **git ls-remote origin** или указав другой удаленный репозиторий, который хотите проверить.

Если репозиторий находится на сайте GitHub и у вас есть открытые запросы на включение, появится набор ссылок с приставкой **refs/pull/**. По своей сути это ветки, но так как они не перечислены в разделе **refs/heads/**, при клонировании или извлечении данных с сервера вы их содержимого не получите — в обычном состоянии команда **fetch** их игнорирует.

Для каждого запроса на включение есть две ссылки — та, которая заканчивается символами **/head**, точно указывает на последний коммит в ветке запроса. Поэтому если пользователь откроет запрос на включение в наше хранилище, а его ветка будет называться **bug-fix** и указывать на коммит **a5a775**, в нашем репозитории ветка **bug-fix** не появится (так как она принадлежит ответившейся версии), но мы получим ссылку **pull/<pr#>/head** на коммит **a5a775**. Как видите, мы можем легко извлечь содержимое любой ветки запроса на включение, не добавляя к себе ветки из удаленных репозиториях.

Попробуем воспользоваться прямой ссылкой для извлечения данных:

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
* branch refs/pull/958/head -> FETCH_HEAD
```

Эта команда скажет Git: «Подключись к удаленной ветке **origin** и скачай данные по ссылке **refs/pull/958/head**». Система Git скачает все, что вам нужно для конструирования этой ссылки, и поместит указатель на нужный вам коммит в папку **.git/FETCH_HEAD**. После этого можно будет воспользоваться командой **git merge FETCH_HEAD** и слить данные в ветку, в которой вы собираетесь их тестировать, но такое сообщение фиксации выглядит несколько странно. Кроме того, вряд ли вам захочется многократно проделывать эту рутинную операцию, когда придется работать с большим количеством запросов на включение.

К счастью, есть способ извлечь данные из всех запросов на включение и поддерживать их в актуальном состоянии простым подключением к удаленному репозиторию.

Откройте в своем любимом редакторе файл **.git/config** и найдите удаленный репозиторий **origin**. Это может выглядеть вот так:

```
[remote "origin"]
  url = https://github.com/libgit2/libgit2
  fetch = +refs/heads/*:refs/remotes/origin/*
```

Строка, которая начинается с `fetch =`, обеспечивает проецирование имен в удаленном репозитории на имена в вашей локальной папке `.git`. По сути, она сообщает системе Git: «Все, что в удаленном репозитории находится по адресу `refs/heads`, должно попасть в папку моего репозитория `refs/remotes/origin`». Этот раздел можно отредактировать, добавив еще одну спецификацию ссылок:

```
[remote "origin"]
  url = https://github.com/libgit2/libgit2.git
  fetch = +refs/heads/*:refs/remotes/origin/*
  fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

Эта последняя строка скажет Git: «Все ссылки вида `refs/pull/123/head` должны сохраняться локально по адресу `refs/remotes/origin/pr/123`». Сохраните этот файл и выполните команду `git fetch`:

```
$ git fetch
# ...
* [new ref] refs/pull/1/head -> origin/pr/1
* [new ref] refs/pull/2/head -> origin/pr/2
* [new ref] refs/pull/4/head -> origin/pr/4
# ...
```

Теперь все запросы на извлечение информации из удаленного репозитория представлены в виде локальных ссылок, которые функционируют как ветки слежения; они предназначены только для чтения и обновляются при выполнении команды `fetch`. В итоге вы можете без проблем локально протестировать код из запроса на включение:

```
$ 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'
```

Проницательные читатели могли обратить внимание на слово `head` в конце спецификации удаленных ссылок. На стороне GitHub имеется также ссылка `refs/pull/#/merge`, ведущая к коммиту, который мог бы возникнуть, если бы вы щелкнули на кнопке `Merge`. Это позволяет посмотреть на результат слияния до его выполнения.

Запросы на включение запросов на включение

Можно открыть запрос на включение не только в ветку `main` или `master`, но и в любую другую ветку в сети. Более того, можно открыть запрос на включение в другой запрос.

Если вы обнаружили перспективный запрос на включение и у вас появилась базирующаяся на нем идея, но вы не уверены в ее правильности или просто у вас нет доступа на запись в целевую ветку, можно открыть запрос на включение своей информации в этот запрос.

В верхней части страницы открытия запроса находятся поля, в одном из которых указана ветка, предназначенная для внесения изменений, в другом — ветка, содержащая предназначенные для включения данные. Щелчок на расположенной справа от этих полей кнопке Edit дает возможность менять не только ветку, но и ветвление (рис. 6.37).



Рис. 6.37. Вручную меняем целевое ветвление и ветку запроса на включение

Здесь вы можете легко указать, что содержимое вашей ветки следует сливать в другой запрос на включение или в другое ветвление проекта.

Упоминания и уведомления

В GitHub встроена прекрасная система уведомлений, которая приходит на помощь, когда у вас появляются вопросы или возникает необходимость в обратной связи с конкретным человеком или группой.

Введенный в любой комментарий символ @ активизирует функцию автозаполнения, открывающую список всех соавторов или участников проекта (рис. 6.38).

Вы также можете упомянуть пользователя, имя которого отсутствует в раскрывающемся списке, но зачастую функция автозаполнения просто ускоряет данный процесс.

После публикации комментария, в котором упоминается пользователь, этот пользователь получает уведомление. Это весьма эффективный способ привлечения людей к обсуждению. На сайте GitHub авторы запросов на включение часто привлекают других пользователей из своей рабочей группы или фирмы для анализа проблемы или запроса.



Рис. 6.38. Наберите символ @, чтобы упомянуть конкретного пользователя

Человек, упомянутый в проблеме или запросе на включение, подписывается на него и начинает получать уведомления обо всех совершаемых в данной теме действиях. Подписка появляется и при открытии какого-то элемента, просмотре репозитория или комментировании в чужой теме. Если вы не хотите получать уведомления, воспользуйтесь кнопкой **Unsubscribe** (рис. 6.39).

Notifications

🔊 **Unsubscribe**

You're receiving notifications
because you commented.

Рис. 6.39. Кнопка, позволяющая отписаться от обновлений, связанных с проблемой или запросом на включение

Страница уведомлений

Когда мы говорим об «уведомлениях» в контексте GitHub, мы подразумеваем некий способ, которым сайт пытается сообщить о возникновении некоего события. Вы можете выбрать удобный для себя вариант. Это делается на вкладке **Notification center** страницы настройки (рис. 6.40).

Вы можете получать уведомления по электронной почте или просматривать их в Интернете. Существует возможность выбрать один из этих вариантов, ни одного или оба сразу, причем как для случая, когда вы принимаете активное участие в каких-то событиях, так и для деятельности, связанной с просматриваемыми вами репозиториями.

Уведомления через Интернет. Уведомления через Интернет находятся на сайте GitHub, а значит, увидеть их можно только на этом сайте. Если вы выбрали этот вариант получения уведомлений, об их появлении сообщит маленькая синяя точка над значком уведомлений в верхней части экрана, показанная на рис. 6.41.

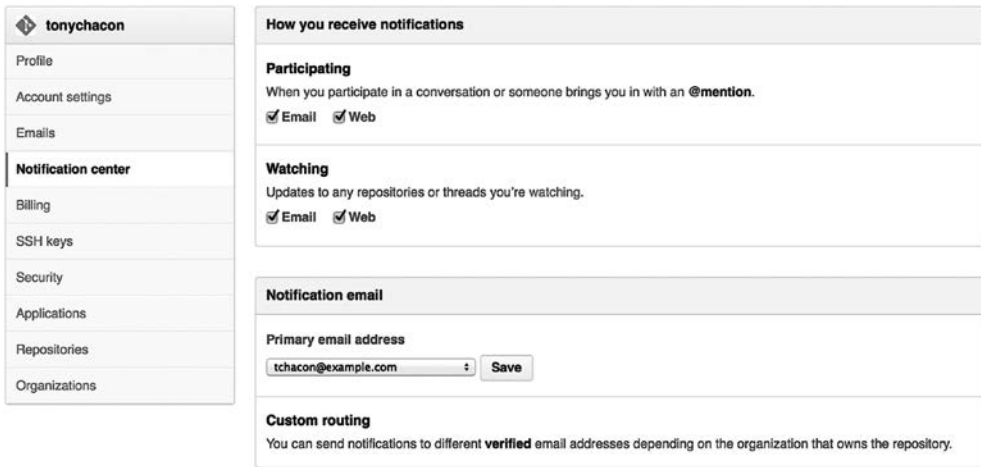


Рис. 6.40. Вкладка Notification center

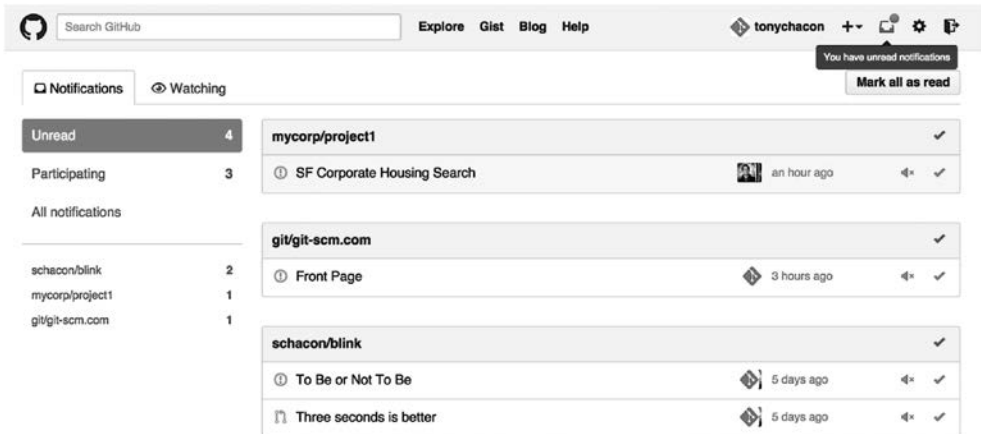


Рис. 6.41. Центр уведомлений

Щелчок на этом значке откроет сгруппированный по проектам список всех элементов, о которых вам пришли уведомления. Чтобы отобразить только уведомления, связанные с конкретным проектом, выделите имя этого проекта на панели слева. Можно подтвердить принятие уведомления, щелкнув на расположенной справа от него галочке. Щелчок на галочке рядом с именем группы отметит все связанные с этой группой уведомления как прочитанные. Рядом с каждой галочкой находится также значок в виде рупора, щелчок на котором отключает дальнейшие уведомления, связанные с этим элементом.

Все эти инструменты крайне полезны при работе с большим количеством уведомлений. Многие пользователи сайта GitHub предпочитают полностью отказаться

от уведомлений по электронной почте и просматривать их непосредственно на сайте.

Уведомления по электронной почте. Второй вариант работы с GitHub-уведомлениями связан с электронной почтой. Если вы выбрали этот вариант, каждое уведомление будет присылаться вам по электронной почте. Примеры таких уведомлений вы видели на рис. 6.13 и 6.34. Сообщения корректно разбиты на цепочки, что очень удобно, если ваш почтовый клиент умеет их визуализировать.

В заголовок сообщения встроено множество метаданных, позволяющих настраивать фильтры и правила. К примеру, в показанном на рис. 6.34 заголовке письма, присланного Тони, мы найдем следующую информацию:

```
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
```

Обратим ваше внимание на пару интересных моментов. Если вы хотите выделить или изменить маршрут почты для этого конкретного проекта или даже запроса на включение, в поле **Message-ID** вы найдете все нужные для этого данные в формате `<пользователь>/<проект>/<тип>/<id>`. К примеру, если бы сообщение касалось проблемы, в поле `<тип>` фигурировало бы слово «issues», а не «pull». Присутствие полей **List-Post** и **List-Unsubscribe** означает, что при наличии почтового клиента, который умеет с ними работать, вы можете легко отправить сообщение в обсуждение или отписаться от него. Последнее аналогично щелчку на значке с изображением рупора на интернет-версии страницы уведомлений или щелчку на кнопке **Unsubscribe** на странице запроса на включение или проблемы.

Заметим также, что если у вас включены режимы получения обоего типа уведомлений и в вашем почтовом клиенте разрешен показ изображений, прочитанное сообщение электронной почты приведет к тому, что в интернет-версии это уведомление тоже будет фигурировать как прочитанное.

Специальные файлы

Существуют файлы, наличие которых в вашем репозитории не пройдет незамеченным для GitHub.

README

Во-первых, это файл **README**, который может иметь любой формат, распознаваемый сайтом GitHub как текстовый. Например, он может называться **README**, **README.md**,

`README.asciidoc` и т. п. Обнаружив этот файл среди вашего кода, GitHub отобразит его на целевой странице проекта.

Многие группы помещают в этот файл всю связанную с проектом информацию, необходимую пользователям, которые впервые попали в данный репозиторий. Как правило, там указывается следующая информация:

- ☐ назначение проекта;
- ☐ особенности его конфигурации и установки;
- ☐ пример применения проекта или его запуска;
- ☐ лицензия, по которой доступен этот проект;
- ☐ способ внести в него вклад.

Так как GitHub показывает содержимое этого файла, туда можно встроить изображения или ссылки, облегчающие понимание материала.

CONTRIBUTING

Второй распознаваемый GitHub специальный файл называется `CONTRIBUTING`. Обнаружив файл с таким именем и произвольным расширением, GitHub в момент открытия пользователем запроса на включение визуализирует его содержимое, как показано на рис. 6.42.

The screenshot shows the GitHub pull request interface. At the top, a light gray box contains the text: "Please review the guidelines for contributing to this repository." Below this, the main content area is divided into two columns. The left column has a "Title" input field at the top. Below it are two tabs: "Write" (active) and "Preview". Under the "Write" tab, there is a large text area with the placeholder "Leave a comment". At the bottom of this text area, a small note says: "Attach images by dragging & dropping, selecting them, or pasting from the clipboard." To the right of the text area, there are two icons: a document with a checkmark labeled "Parsed as Markdown" and a full-screen icon labeled "Edit in fullscreen". The right column features a GitHub logo at the top, followed by the text: "We can't automatically merge these branches. Don't worry, you can still create the pull request." At the bottom of the right column is a button labeled "Create pull request".

Рис. 6.42. Открытие запроса на включение при наличии файла `CONTRIBUTING`

Здесь перечисляется то, что вы хотите или не хотите видеть в присылаемых в ваш проект запросах на включение. Пользователи могут ознакомиться с вашими пожеланиями перед открытием запроса.

Администрирование проекта

В общем случае существует не так уж много операций, связанных с администрированием единственного проекта, но с некоторыми возможностями нам бы хотелось вас познакомить.

Изменение ветки, предлагаемой по умолчанию

Если в качестве ветки, предлагаемой по умолчанию, у вас фигурирует ветка, отличная от ветки `master`, и вы хотите, чтобы именно для нее пользователями открывались запросы на включение и именно она визуализировалась по умолчанию, перейдите на вкладку `Options` страницы настройки вашего хранилища (рис. 6.43).

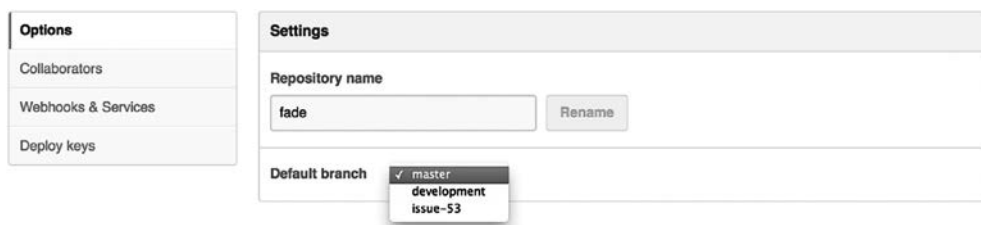


Рис. 6.43. Изменение ветки, по умолчанию используемой для проекта

Просто выберите в раскрывающемся списке ветку, которая будет фигурировать как предлагаемая по умолчанию для большинства операций, в том числе как ветка, на которую выполняется переход при клонировании репозитория.

Пересылка проекта

Для тех, кто хочет переслать проект другому пользователю или организации, на сайте GitHub в нижней части все той же вкладки `Options` есть кнопка `Transfer` (рис. 6.44).

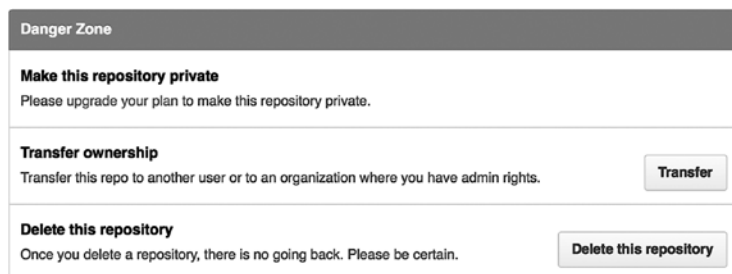


Рис. 6.44. Пересылка проекта другому пользователю сайта GitHub или организации

Это может пригодиться в ситуации, когда вы больше не хотите заниматься проектом, а кто-то готов взяться за его поддержку, или когда проект настолько разросся, что вы предпочитаете передать его какой-нибудь организации.

Эта операция не только перемещает в другое место репозиторий со всеми, кто за ним наблюдает, но и настраивает перенаправление с вашего URL-адреса на новый. Перенаправлению подвергаются даже Git-команды `clone` и `fetch`, а не только веб-запросы.

Управление организацией

Кроме учетных записей пользователей сайт GitHub допускает так называемые учетные записи организаций. Такие учетные записи имеют свое пространство имен, в котором существуют все их проекты, но ряд других аспектов работы с ними отличается от работы с обычными учетными записями. Ведь в данном случае речь идет о группе пользователей, которые одновременно являются владельцами проектов, поэтому существует множество инструментов для управления их подгруппами. Как правило, подобные учетные записи используются проектами с открытым исходным кодом (такими как Perl или Rails) или компаниями (такими как Google или Twitter).

Основные сведения об организации

Создать организацию очень просто: щелкните на значке + в верхнем правом углу любой страницы сайта GitHub и выберите в появившемся меню команду **New organization** (рис. 6.45).

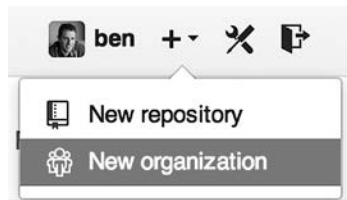


Рис. 6.45. Выбор в меню команды **New organization**

Первым делом следует указать имя вашей организации и адрес электронной почты, который будет служить для связи с группой. После этого можно пригласить других пользователей присоединиться к вам в качестве владельцев учетной

записи. Как и персональные учетные записи, учетные записи организаций бесплатны, пока речь идет о хранении открытого кода.

Как владелец при ветвлении репозитория вы можете поместить ветку в пространство имен вашей организации. Создавать новые репозитории вы можете как от имени своей личной учетной записи, так и от имени любой организации, владельцем которой вы являетесь. Кроме того, вы автоматически получаете доступ на просмотр любого репозитория, созданного от имени этих организаций.

Как и при работе с персональной учетной записью, вы можете добавить в учетную запись организации аватар, чтобы выделить ее на общем фоне. Будет у вас и основная страница с перечнем всех принадлежащих организации репозиториях, доступная для просмотра всем пользователям.

Теперь поговорим о том, что возможно только для учетной записи организации.

Группы

С отдельными пользователями организации связаны через группы, объединяющие пользовательские учетные записи и репозитории по уровням доступа.

Предположим, у организации есть три репозитория: `frontend`, `backend` и `deployscripts`. Вам хотелось бы, чтобы у разработчиков, использующих HTML/CSS/Javascript, был доступ к репозиторию `frontend` и, может быть, к репозиторию `backend`, а у отдела эксплуатации — доступ к репозиториям `backend` и `deployscripts`. Группы позволяют легко реализовать это на практике, избавляя вас от необходимости предоставлять коллегам доступ к каждому репозиторию в отдельности.

Страница **Organization** содержит простую панель управления всеми принадлежащими вашей организации репозиториями, пользователями и группами (рис. 6.46).

Чтобы перейти к управлению группами, щелкните справа на боковой панели **Teams** (см. рис. 6.46). Откроется страница, на которой вы можете добавлять в группу новых членов, добавлять доступные группе репозитории и управлять настройками уровня доступа для группы. У каждой группы может быть доступ только на чтение, на чтение/запись или на администрирование репозиториях. Для изменения уровня доступа щелкните на кнопке **Settings**, как показано на рис. 6.47.

Человек, приглашенный в группу, получает уведомление. Также не стоит забывать про упоминания (например, `@acmecorp/frontend`), которые функционируют так же, как в случае индивидуальных пользователей, но подписку на обсуждение получают все члены группы. Это полезно в ситуации, когда требуется привлечь внимание кого-то из членов группы, но вы не знаете, к кому именно имеет смысл обратиться.

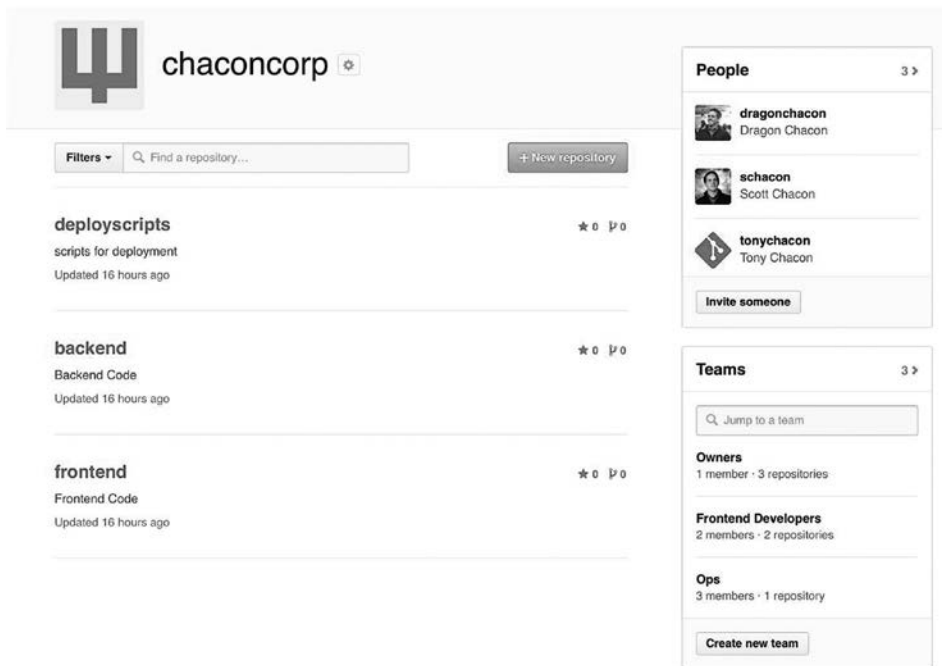


Рис. 6.46. Страница Organization

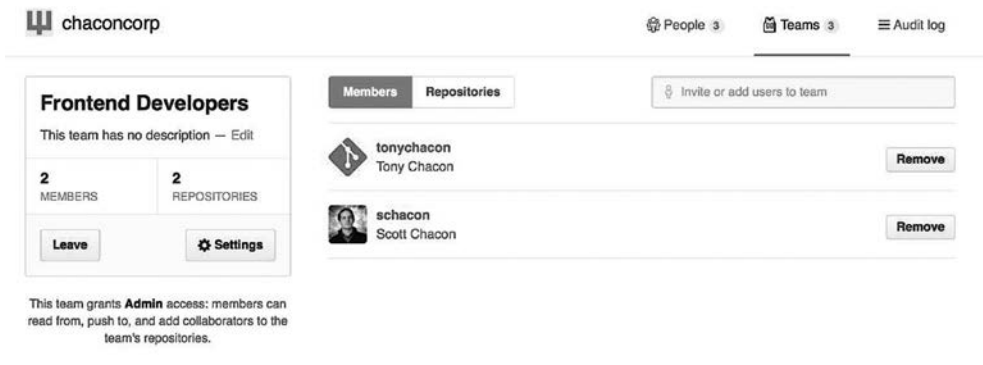


Рис. 6.47. Страница Team

Пользователь может состоять в произвольном числе групп, поэтому не ограничивайте свое членство группами контроля доступа. Группы по интересам, например `ux`, `css` или `refactoring`, пригодятся при обсуждении одного типа вопросов, в то время как в группах `legal` и `colorblind` обсуждаются проблемы совсем другого рода.

Журнал регистрации

Владелец имеет доступ ко всей информации, связанной с управляемой им организацией. На вкладке **Audit Log** можно увидеть все события, происшедшие на уровне организации, а также кем и где они были инициированы (рис. 6.48).

Вы можете выполнять фильтрацию по типам событий, по местам или по конкретным пользователям.

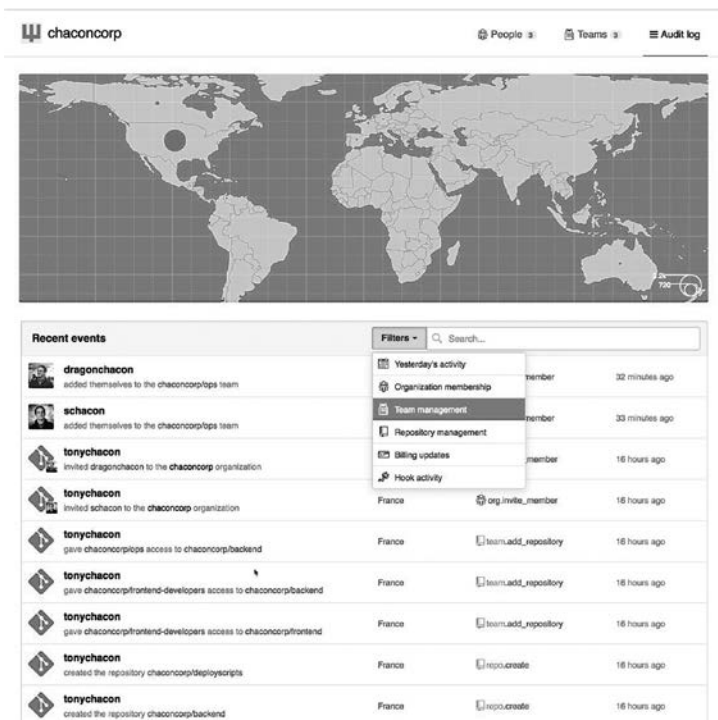


Рис. 6.48. Вкладка Audit log

GitHub-сценарии

Итак, вы познакомились с основными функциональными особенностями и рабочими схемами сайта GitHub, но в любой крупной группе проектов периодически возникает необходимость сделать какие-то настройки или добавить к ней какие-то внешние службы.

К счастью для нас, GitHub допускает разного рода вмешательства в свой код. В этом разделе мы поговорим о работе с GitHub-системой хуков и API этого сайта. Именно эти вещи позволяют добиться нужных нам результатов.

Хуки

Раздел управления хуками и службами, относящийся к администрированию GitHub-репозитория, является самым простым инструментом, чтобы наладить взаимодействие сайта GitHub с внешними системами.

Службы

Начнем мы со служб. Интеграция хуков и служб осуществляется в разделе **Settings** вашего репозитория, которым вы уже пользовались для добавления соавторов и изменения ветки, предлагаемой по умолчанию. Вам на выбор предоставляются десятки служб, большая часть которых осуществляет интеграцию в другие коммерческие системы и системы с открытым исходным кодом. По большей части это службы непрерывной интеграции, системы слежения за ошибками и проблемами, системы чатов и системы документирования. Рассмотрим процесс настройки на простом примере. Выбрав в раскрывающемся списке **Add Service** вариант **email**, вы попадете на экран, показанный на рис. 6.49.

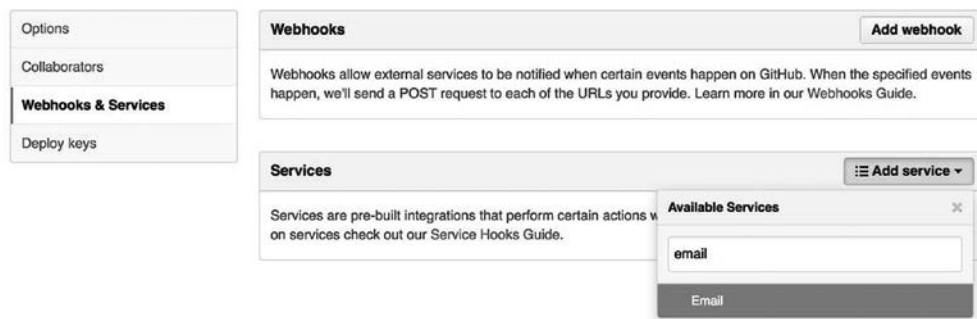


Рис. 6.49. Конфигурирование службы электронной почты

В данном случае щелчок на кнопке **Add service** приведет к тому, что каждый раз при добавлении кем-то данных в наш репозиторий нам будет приходить сообщение. Службы могут слушать события множества типов, но чаще всего они отслеживают события добавления и выполняют с полученными данными какие-то действия.

Если вы хотите осуществить интеграцию какой-то системы с GitHub, первым делом проверьте наличие готовой службы интеграции. К примеру, если для тестирования на своей базе кода вы пользуетесь системой Jenkins, можно включить встроенный режим интеграции с этой системой, и она будет запускаться для любого добавляемого в ваш репозиторий кода.

Хуки

Если вам требуется что-то более специфическое или служба, с которой вы хотите выполнить интеграцию, отсутствует в списке, можно прибегнуть к более универсальной системе хуков. Хуки, связанные с GitHub-репозиторием, крайне просты. Вы указываете адрес URL, а GitHub по протоколу HTTP отправляет на этот адрес полезные данные для любого нужного вам события.

В общем случае вы настраиваете небольшую веб-службу, слушающую информацию от GitHub-хука и затем совершающую какие-то действия с полученными данными.

Для включения хука щелкните на кнопке **Add webhook**, как показано на рис. 6.49. Вы окажетесь на странице, представленной на рис. 6.50.

The image shows the GitHub 'Webhooks / Add webhook' configuration page. On the left, a sidebar contains links for 'Options', 'Collaborators', 'Webhooks & Services' (which is highlighted), and 'Deploy keys'. The main content area is titled 'Webhooks / Add webhook'. It contains the following elements:

- A text block explaining that a POST request will be sent to the provided URL with details of subscribed events, and that the user can specify the data format (JSON, x-www-form-urlencoded, etc.).
- A 'Payload URL' field with the example value 'https://example.com/postreceive'.
- A 'Content type' dropdown menu set to 'application/json'.
- A 'Secret' text input field.
- A section titled 'Which events would you like to trigger this webhook?' with three radio button options:
 - ☒ Just the push event.
 - ☐ Send me **everything**.
 - ☐ Let me select individual events.
- An 'Active' checkbox, which is checked, with the text 'We will deliver event details when this hook is triggered.'
- An 'Add webhook' button at the bottom.

Рис. 6.50. Выбор конфигурации хука

Конфигурационные настройки в данном случае очень просты. В большинстве случаев достаточно указать URL-адрес и секретный ключ, а затем щелкнуть на кнопке **Add webhook**. Возможны разные варианты событий, в случае которых вы захотите получать данные от GitHub, — по умолчанию это событие **push**, то есть добавление нового кода в любую ветку вашего репозитория.

Вот небольшой пример веб-службы, которую можно настроить для обработки хука. Мы воспользуемся фреймворком Sinatra на языке Ruby, так как он

в достаточной степени интуитивно понятен и позволяет легко посмотреть, что именно мы делаем.

Предположим, мы хотим получать письмо, когда определенный человек отправляет код в определенную ветку нашего проекта, редактируя определенный файл. В этом нам поможет следующий код:

```
require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON

  # сбор данных, которые нам нужны
  pusher = push["pusher"]["name"]
  branch = push["ref"]

  # получение списка всех затронутых файлов
  files = push["commits"].map do |commit|
    commit['added'] + commit['modified'] + commit['removed']
  end
  files = files.flatten.uniq

  # проверка по заданным критериям
  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 берем получаемые от GitHub данные в формате JSON и смотрим, кто их автор, в какую ветку они были оправлены, какие файлы оказались затронутыми присланными коммитами. Затем мы проверяем заданные критерии и посылаем сообщение в случае совпадения.

Для разработки и тестирования подобных вещей потребуется консоль разработчика на том же экране, где вы настраиваете хук. Вы можете посмотреть несколько последних доставок, которые для этого хука попытался сделать сайт GitHub. Для каждого хука можно посмотреть время доставки данных, успешность ее прохождения, а также тело и заголовки запроса и ответа. Все это сильно упрощает тестирование и отладку хуков (рис. 6.51).

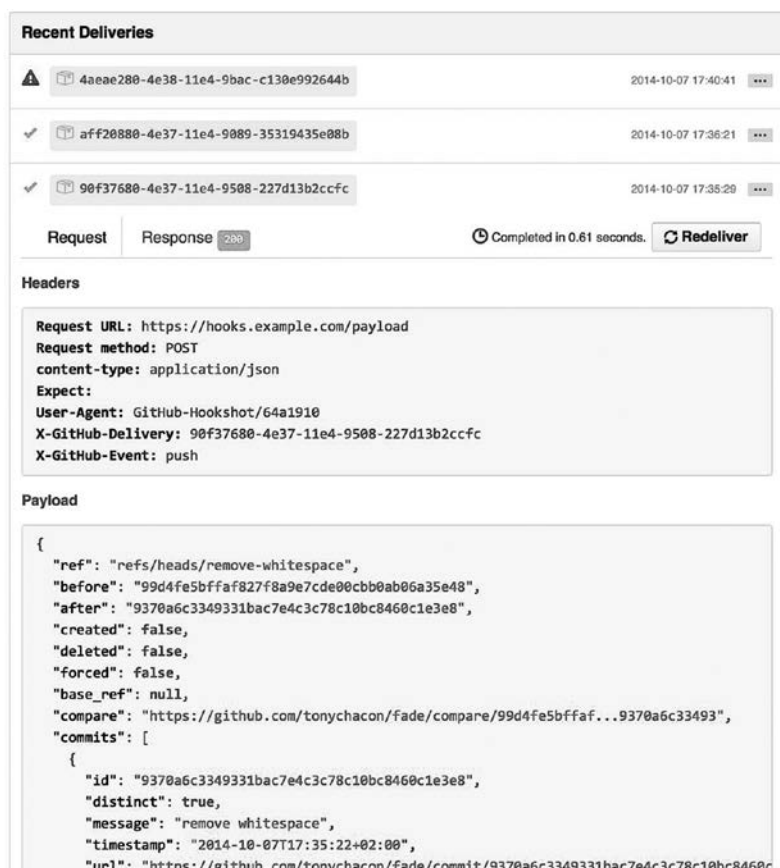


Рис. 6.51. Отладочная информация для хуков

Другой замечательной возможностью является возможность повторной доставки любых данных, полезная при тестировании службы.

Дополнительную информацию о написании хуков и типах доступных для прослушивания событий вы найдете в документации GitHub-разработчика по адресу: <https://developer.github.com/webhooks/>.

API для GitHub

Службы и хуки уведомляют о событиях, происходящих в ваших хранилищах. Но как быть, если требуется более подробная информация? Что делать, если вы хотите автоматизировать такие действия, как добавление соавторов или пометка проблем?

Здесь вам на помощь приходит API для GitHub. На сайте GitHub вы найдете множество прикладных программных интерфейсов (Applications Programming Interface, API), автоматизирующих практически любые действия. В этом разделе мы поговорим о способах аутентификации и подключения к API, а также о способах добавления комментариев к проблемам и изменения статуса запроса на включение через API.

Основы работы

Базовой вещью является простой запрос GET к конечной точке, не требующей аутентификации. Это могут быть сведения о пользователе или предназначенная только для чтения информация из проекта с открытым исходным кодом. Вот пример получения сведений о пользователе **schacon**:

```
$ 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"
}
```

Существуют миллионы аналогичных конечных точек, предоставляющих данные об организациях, проектах, проблемах, коммитах — практически обо всем, что находится в открытом доступе на сайте GitHub. Воспользоваться API можно даже для визуализации произвольной разметки на языке Markdown или поиска шаблона **.gitignore**.

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "/*.class

# Мобильные инструменты для Java (J2ME)
.mtj.tmp/

# Пакетные файлы #
*.jar
*.war
*.ear

# просмотр журнала аварии виртуальной машины http://www.java.com/en/download/
help/error_hotspot.xml
hs_err_pid*
"
```

Комментирование проблем

Однако для различных действий на сайте, таких как комментирование проблемы или запроса на включение либо получение закрытых данных, вам потребуется аутентификация.

Ее можно выполнить разными способами. Можно ограничиться базовой аутентификацией по имени пользователя и паролю, но в общем случае лучше воспользоваться личным маркером доступа. Он генерируется на вкладке Applications страницы настройки (рис. 6.52).

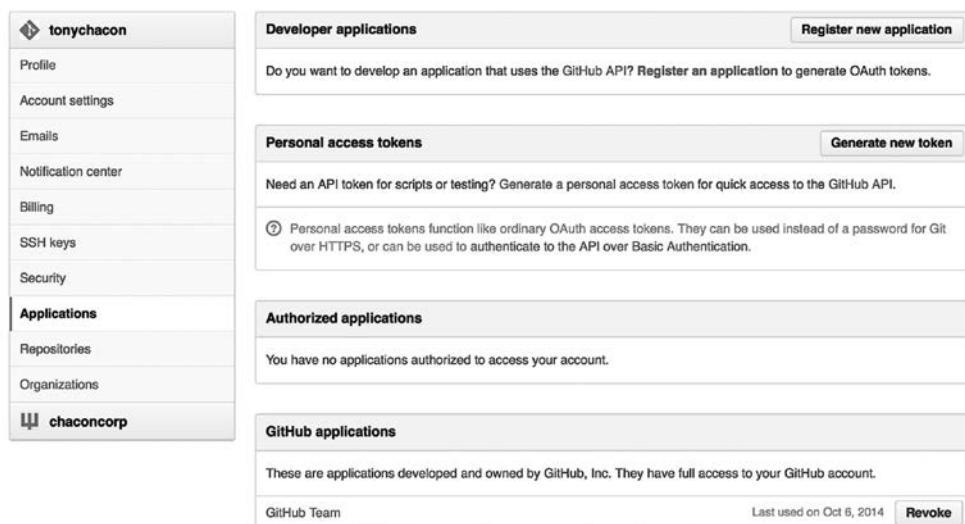


Рис. 6.52. Генерация маркера доступа на вкладке Applications страницы настройки

Вас попросят указать сферу действия этого маркера и дать его описание. Постарайтесь сделать описание таким, чтобы по нему можно было легко найти маркер, когда надобность в нем отпадет и потребуется его удалить.

Сайт GitHub показывает маркер всего один раз, поэтому не забудьте его скопировать. После этого вы можете указывать его вместо имени пользователя и пароля для аутентификации в ваших сценариях. Это очень удобно, так как вы можете ограничить диапазон своих действий, а маркер в любой момент можно уничтожить.

Кроме того, вы получаете дополнительное преимущество в виде повышения быстродействия. Без аутентификации вы можете делать всего 60 запросов в час, а после ее прохождения количество запросов в час возрастает до 5000.

Давайте посмотрим на процедуру добавления комментария к одной из наших проблем. К примеру, мы хотим прокомментировать проблему номер 6. Для этого нам потребуется сделать HTTP-запрос POST к странице `repos/<пользователь>/<хранилище>/issues/<номер>/comments` с только что сгенерированным как заголовок `Authorization` маркером.

```
$ 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:"
}
```

Если теперь посмотреть на цепочку обсуждения данной проблемы, вы увидите только что отправленный комментарий (рис. 6.53).

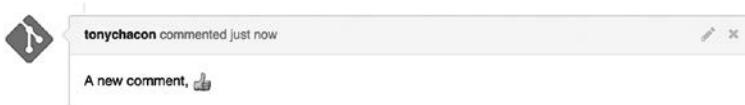


Рис. 6.53. Комментарий, отправленный посредством API для GitHub

Применять API можно практически для любых операций на сайте: создания и настройки контрольных точек, связывания пользователей с проблемами и запросами на включение, создания и редактирования меток, доступа к данным коммита, создания новых коммитов и веток, открытия, закрытия и слияния запросов на включение, создания и редактирования групп, комментирования строк кода в запросе на включение, поиска по сайту и многого другого.

Изменение статуса запроса на включение

Вот еще один последний пример, который мы рассмотрим в этой главе, так как он может пригодиться вам при работе с запросами на включение. С каждым коммитом может быть связано одно или несколько состояний, поэтому существует

прикладной программный интерфейс, позволяющий добавлять состояние или делать к нему запрос.

К этому интерфейсу прибегает большинство служб непрерывной интеграции и тестирования с целью проверки присланного кода и формирования отчета в случае, если коммит прошел все тесты. Вы можете воспользоваться им, к примеру, чтобы проверить корректность форматирования сообщения фиксации, следование автора коммита вашим пожеланиям, правильность подписи коммита — словом, выполнить любые проверки.

Предположим, вы настроили для своего репозитория хук, связывающийся с компактной веб-службой, которая проверяет строку **Signed-off-by** в сообщении фиксации.

```
require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON
  repo_name = push['repository']['full_name']

  # просмотр всех сообщений фиксации
  push["commits"].each do |commit|

    # поиск строки Signed-off-by
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end

    # отправка состояния на сайт 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
```


К счастью, понять этот код просто. В данном обработчике хука мы просматриваем все только что присланные коммиты, ищем в их сообщениях фиксации строку **Signed-off-by** и выполняем HTTP-запрос **POST** к странице `/repos/<пользователь>/<репозиторий>/statuses/<sha_коммита>` конечной точки API с данными о состоянии.

В данном случае можно послать данные состояния (**success**, **failure**, **error**), описание происшедшего, URL-адрес страницы, на которой пользователь может получить дополнительную информацию, а в случае, если с одним коммитом связаны несколько состояний, еще и контекст. Ведь отчет о состоянии может послать как служба тестирования, так и служба проверки, подобная описанной; различаться они будут только контекстом.

Если при наличии этого хука кто-то откроет на сайте GitHub новый запрос на включение, может возникнуть ситуация, показанная на рис. 6.54.

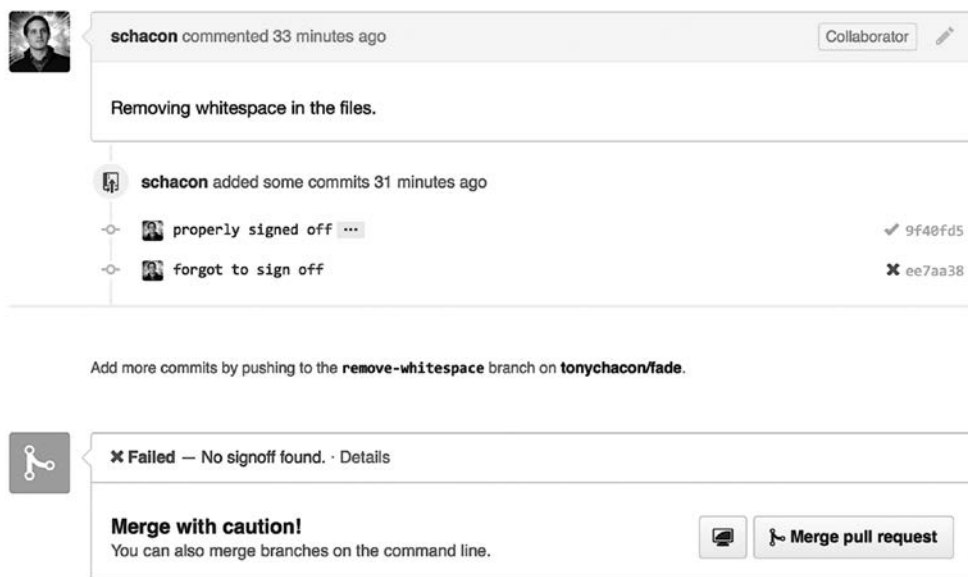


Рис. 6.54. Состояние коммита, полученное через API

Обратите внимание на галочку рядом с коммитом, у которого есть строка **Signed-off-by**, и крестик рядом с другим коммитом, автор которого забыл добавить подпись. При этом запрос на включение принимает состояние последнего коммита в ветке и выводит предостережение в случае, когда его слияние невозможно. Крайне рекомендуем вам пользоваться этим API для результатов тестирования, чтобы избежать случайного слияния с коммитом, не прошедшим тестирование.

От пользователя Octokit

Хотя в данной главе мы рассмотрели практически все, что касается сложных и простых HTTP-запросов, существует несколько библиотек открытого исходного кода, позволяющих работать с этим API в более свободном стиле. На момент написания данной книги поддерживались такие языки, как Go, Objective-C, Ruby и .NET. Дополнительную информацию об этом можно получить на странице <http://github.com/octokit>.

Надеемся, что эти инструменты помогут вам настроить сайт GitHub, оптимизировав его под собственную схему работы. Полная документация на API и руководство по решению типовых задач находятся на странице <https://developer.github.com>.

Заключение

Теперь вы — пользователь сайта GitHub. Вы умеете создавать учетные записи, руководить организацией, создавать и обновлять репозитории, вносить вклад в чужие проекты и принимать вклады от других пользователей. В следующей главе вы познакомитесь с еще более мощными инструментами и найдете ряд советов, помогающих отыскать решение в сложных ситуациях. Все это сделает вас настоящим экспертом по Git.

7

Git-инструментарий

К настоящему моменту вы уже изучили большинство необходимых для повседневной работы команд и рабочих схем, используемых для управления Git-репозиторием и исходным кодом. Вы решили ряд задач, связанных с файлами слежения и фиксации состояния, а также ощутили на собственном опыте, насколько мощными инструментами являются область индексирования и облегченные процедуры ветвления и слияния.

Пришла пора детально рассмотреть ряд высокопроизводительных операций, которые, возможно, не потребуются вам для решения рутинных задач, но в какой-то момент могут пригодиться.

Выбор версии

В Git есть несколько способов выбора определенных коммитов или диапазона коммитов. Далеко не все из них очевидны, но их следует знать.

Одна версия

Понятно, что обращение к коммиту можно выполнить через его контрольную сумму SHA-1, но существуют и более удобные для восприятия варианты. В этом разделе мы перечислим различные способы ссылки на отдельный коммит.

Короткий вариант SHA

Система Git достаточно умна, чтобы по нескольким первым символам хеша распознать, какой коммит вы имеете в виду. Главное, чтобы частичная контрольная сумма SHA-1 состояла хотя бы из четырех символов и была уникальной, — то есть в текущем репозитории ей должен соответствовать всего один объект.

Предположим, вы запускаете команду `git log` и находите коммит, в который вы добавили какие-то конструктивные особенности:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

В данном случае выберем коммит `1c002dd...`. Если теперь воспользоваться командой `git show` для разных вариантов написания контрольной суммы, то результат во всех показанных далее случаях будет одинаковым (разумеется, при условии, что минимальная длина контрольной суммы соответствует единственной версии):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Для ваших значений SHA-1 Git может представить короткие, уникальные сокращения. Добавив к команде `git log` параметр `--abbrev-commit`, вы получите набор сокращенных и вместе с тем уникальных значений; по умолчанию длина составляет семь символов, но при необходимости она увеличивается до размера, обеспечивающего уникальность контрольной суммы SHA-1:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

В общем случае для уникальности в рамках проекта достаточно указать от восьми до десяти символов.

К примеру, в таком крупном проекте, как ядро Linux с его более 450 тысячами коммитов и 3,6 миллиона объектов, не существуют двух объектов, контрольная сумма SHA которых совпадала бы более чем в первых 11 символах.

ПРИМЕЧАНИЕ ПО ПОВОДУ SHA-1

Многие пользователи озабочены тем, что в какой-то момент в их репозитории могут оказаться два объекта с одинаковым значением SHA-1. Что произойдет в этом случае?

Предположим, вы пытаетесь выполнить фиксацию объекта, хеш SHA-1 которого совпадает с хешем другого объекта, уже существующего в данном репозитории. Система Git обнаружит этот предыдущий объект в базе данных, и с ее точки зрения он уже будет записан. В результате при будущих проверках вы все время будете получать данные первого объекта.

Другое дело, что подобный сценарий практически невероятен. Размер хеш-суммы SHA-1 составляет 20 байт, или 160 бит. Количество случайным образом хешированных объектов, необходимое для гарантирования 50-процентной вероятности единственного совпадения, составляет примерно 2^{80} (вот формула расчета вероятности совпадения $p = (n(n-1)/2) \times (1/2^{160})$). Это $1,2 \times 10^{24}$, или более триллиона триллионов вариантов, что в 1200 раз больше количества песчинок на земле.

Вот пример, демонстрирующий, насколько невероятной вещью является совпадение значений SHA-1. Если бы все 6,5 миллиарда жителей нашей планеты занимались программированием, каждую секунду каждый из них производил бы количество кода, равное коду, написанному за всю историю существования ядра Linux (а это 3,6 миллиона Git-объектов), и отправлял бы этот код в один огромный Git-репозиторий, потребовалось бы почти 2 года, чтобы число объектов в этом репозитории достигло величины, необходимой для обеспечения 50-процентной вероятности совпадения SHA-1 у пары объектов. Куда выше вероятность того, что два члена рабочей группы будут одновременно, но независимо друг от друга съедены волками.

Ссылки из веток

Самый простой способ выбора определенного коммита требует наличия ссылки на этот коммит из ветки. В этом случае в любой команде, где требуется указать объект-коммит, можно будет использовать имя ветки или значение SHA-1. Например, если ветка `topic1` ссылается на коммит `ca82a6d`, просмотр последнего коммита в ветке можно осуществить двумя командами:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

Посмотреть значение SHA, на которое ссылается ветка, или понять, как выглядят приведенные примеры в терминах SHA, позволяет служебная Git-программа `rev-parse`. Служебные программы будут подробно рассматриваться в главе 10; а пока скажем только, что эта программа предназначена для низкоуровневых операций и в повседневной работе к ней прибегают не часто. Но в ситуациях, когда нужно

понять, что именно происходит, она незаменима. Вот результат ее применения к вашей ветке.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

Сокращения журнала ссылок

Одной из вещей, которую Git делает в фоновом режиме, является ведение журнала ссылок, в котором хранятся ссылки указателей **HEAD** и веток за последние несколько месяцев.

Для просмотра этого журнала используется команда **git reflog**:

```
$ git reflog
734713b... HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970... HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd... HEAD@{2}: commit: added some blame and merge stuff
1c36188... HEAD@{3}: rebase -i (squash): updating HEAD
95df984... HEAD@{4}: commit: # This is a combination of two commits.
1c36188... HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5... HEAD@{6}: rebase -i (pick): updating HEAD
```

В виде этой временной истории Git сохраняет данные о каждом изменении вершины ветки. Позволяют эти данные указывать и на более старые коммиты. Например, чтобы посмотреть, куда ссылался указатель **HEAD** пять шагов назад, используйте ссылку **@{n}**, 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. Например, чтобы увидеть, как выглядела ветка вчера, следует написать:

```
$ git show master@{yesterday}
```

Вы увидите, что было на вершине ветки вчера. Этот способ работает только для данных, которые все еще находятся в вашем журнале ссылок, поэтому его невозможно использовать для просмотра коммитов, возраст которых превышает несколько месяцев.

Для просмотра информации из журнала ссылок в таком же формате, как в выводимых данных команды **git log**, используйте вариант **git log -g**:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800
```

```
fixed refs handling, added gc auto, updated tests
```

```
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 15:08:43 2008 -0800
```

```
Merge commit 'phedders/rdocs'
```

Важно понимать, что выводимые этой командой данные касаются исключительно локальной информации, — это журнал событий вашего репозитория. Для чужой копии вашего репозитория ссылки будут уже другими; а сразу после клонирования журнал окажется пустым, так как в репозитории еще не совершалось никаких операций. Команда `git show HEAD@{2.months.ago}` даст результат только в случае, когда проект клонирован более двух месяцев назад, — если вы выполнили клонирование пять минут назад, результата не будет.

Ссылки на предков

Указать коммит можно и через его родителя. Символ `^` в конце ссылки с точки зрения Git соответствует предку коммита. Предположим, у вас есть проект со следующей историей:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
* d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

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'
```

После символа `^` можно указать число: например, запись `d921970^2` означает «второй предок коммита d921970». Этот синтаксис применяется только в случае коммитов слияния, у которых существует несколько предков. Первый родитель — это ветка, на которой вы находились в момент слияния, а второй — коммит на ветке, которая подверглась слиянию:

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
```

```
Date: Thu Dec 11 14:58:32 2008 -0800
    added some blame and merge stuff

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul@git@mjr.org>
Date: Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

Другое распространенное обозначение предка — символ `~`. Он также соответствует ссылке на первого родителя, поэтому записи `HEAD~` и `HEAD^` эквивалентны. Различия проявляются, когда вы указываете номер. Запись `HEAD~2` означает «первый предок первого предка», то есть фактически «дедушка», — при этом происходит переход от заданного предка вглубь указанное число раз. К примеру, для показанной ранее истории `HEAD~3` дает следующий результат:

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

Еще это можно записать как `HEAD^^^`, что опять же означает первого предка первого предка первого предка:

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

Указанные обозначения можно комбинировать. К примеру, второго родителя предыдущей ссылки (при условии, что это коммит слияния) можно получить, написав `HEAD~3^2`.

Диапазоны коммитов

Теперь, когда вы научились ссылаться на отдельные коммиты, разберемся с процедурой ссылки на диапазон коммитов. Это часто нужно при управлении ветками: если веток много, именно ссылка на диапазон отвечает на такой вопрос, как, к примеру: «Какие наработки из этой ветки еще не были слиты в основную ветку?»

Две точки

Чаще всего диапазон коммитов обозначают двумя точками. По сути, так вы обращаетесь к Git с командой определить диапазон коммитов, достижимый из

одного коммита, но недостижимый из другого. Рассмотрим историю коммитов с рис. 7.1.

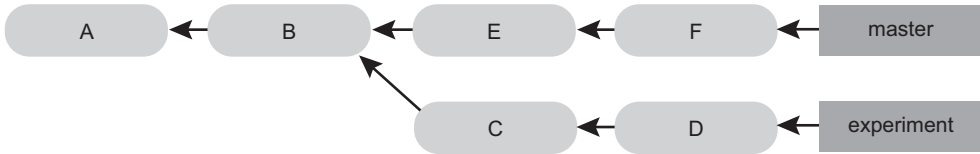


Рис. 7.1. История коммитов, на примере которой мы будем указывать диапазон

Предположим, нам нужно определить, какая информация из ветки **experiment** пока не слита в ветку **master**. Чтобы система Git показала вам только список таких коммитов, следует написать **master..experiment** — это означает «все коммиты из ветки **experiment**, отсутствующие в ветке **master**». Для краткости и наглядности в рассматриваемых примерах мы будем указывать на диаграммах не реальные данные, выводимые командой **log**, а буквы в порядке их появления:

```
$ git log master..experiment
D
C
```

Чтобы рассмотреть обратную ситуацию — все коммиты из ветки **master**, отсутствующие в ветке **experiment**, — достаточно поменять порядок следования веток. Список таких коммитов обеспечит вам запись **experiment..master**:

```
$ git log experiment..master
F
E
```

Это пригодится, когда требуется поддерживать ветку **experiment** в актуальном состоянии и осуществлять предварительный просмотр данных, которые вы собираетесь в нее слить. Еще данный синтаксис часто используется для просмотра информации, которую вы собираетесь отправить на удаленный сервер:

```
$ git log origin/master..HEAD
```

Эта команда покажет вам все коммиты в текущей ветке, отсутствующие в ветке **master** на сервере **origin**. При запуске команды **git push** из ветки, которая следит за веткой **origin/master**, на сервер будут переданы именно коммиты, фигурирующие в выводимых командой **git log origin/master..HEAD** данных. Если опустить одну из частей записи, Git подставит туда **HEAD**. К примеру, команда **git log origin/master..** дает такой же результат, как и показанная ранее команда, потому что вместо пропущенного фрагмента Git подставляет **HEAD**.

Множественная выборка

Запись с двумя точками полезна как сокращенный вариант. Но порой, чтобы указать нужный вам коммит, требуется перечислить более двух веток, — например,

для просмотра коммитов из веток, отсутствующих в той ветке, в которой вы находитесь. В Git это достигается при помощи символа `^` или параметра `--not` перед именем ветки, коммиты из которой вы не хотите видеть. Следующие три команды эквивалентны:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

Этот синтаксис удобен тем, что позволяет указать в запросе более двух ссылок, что невозможно в синтаксисе с двумя точками. Например, если вы хотите увидеть все коммиты, достижимые по ссылке `refA` или `refB`, но недостижимые по ссылке `refC`, можно написать так:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

Это позволяет получить очень мощную систему запросов на изменение, призванную помочь вам выяснить реальную ситуацию с ветками.

Три точки

Последний вариант записи для выбора диапазона коммитов — три точки, обозначающие, что нас интересуют коммиты, достижимые по одной из двух ссылок, но не по обеим одновременно. Вернемся к рассмотренному примеру. Чтобы посмотреть коммиты, которые находятся только в ветке `master` или только в ветке `experiment`, но не в обеих ветках одновременно, можно написать:

```
$ git log master...experiment
F
E
D
C
```

Еще раз напомним, что эта команда даст нам стандартный вывод, просто в нем будет содержаться информация только об этих четырех коммитах, как обычно, упорядоченных по дате создания.

С этой командой часто используют параметр `--left-right`, позволяющий посмотреть, с какой стороны диапазона находится каждый коммит. Это увеличивает практическую ценность данных:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

Эти инструменты дают вам возможность объяснить системе Git, какой коммит или коммиты вы хотите изучить.

Интерактивное индексирование

Вместе с Git поставляется пара сценариев, облегчающих решение некоторых задач из командной строки. В этом разделе мы рассмотрим несколько интерактивных команд, позволяющих включать в коммиты только определенные комбинации или части файлов. Это полезно в случаях, когда результат редактирования файлов вы хотите сохранить в нескольких коммитах, а не в одном большом. Такой подход гарантирует, что ваши коммиты будут представлены как логически разделенные наборы изменений, которые легко задействовать сотрудничающим с вами разработчикам. Добавление к команде `git add` параметра `-i` или `--interactive` переводит Git в режим интерактивной оболочки и дает примерно такой результат:

```
$ git add -i
      staged unstaged path
1:  unchanged   +0/-1  TODO
2:  unchanged   +1/-1  index.html
3:  unchanged   +5/-1  lib/simplegit.rb

*** Commands ***
1: status   2: update   3: revert   4: add untracked
5: patch    6: diff      7: quit      8: help
What now>
```

Фактически эта команда показывает содержимое области индексации — по сути, эту информацию вы получили бы и при помощи команды `git status`, просто в данном случае она представлена в более сжатом и информативном виде. Индексированные изменения отображаются слева, а неиндексированные — справа.

Следом идет раздел **Commands**. Здесь можно выполнить ряд операций, включая индексирование файлов, отмену этой операции, индексирование частей файлов, добавление неотслеживаемых файлов и просмотр новой информации, которая подверглась индексированию.

Индексирование файлов и его отмена

Если ввести 2 или `u` в ответ на приглашение `What now>`, сценарий спросит, какие файлы вы хотите проиндексировать:

```
What now> 2
      staged unstaged path
1:  unchanged   +0/-1  TODO
2:  unchanged   +1/-1  index.html
3:  unchanged   +5/-1  lib/simplegit.rb
Update>>
```

Для индексации файлов `TODO` и `index.html` достаточно указать 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 `*` рядом с именем файла означает, что он выбран для индексирования. Если, ничего не добавляя, нажать клавишу `Enter` в ответ на приглашение `Update>>`, Git возьмет все выбранные на данный момент файлы и проиндексирует их:

```
Update>>
updated 2 paths

*** Commands ***
  1: status   2: update  3: revert  4: add untracked
  5: patch    6: diff    7: quit    8: help
What now> 1
      staged   unstaged path
  1:    +0/-1   nothing TODO
  2:    +1/-1   nothing index.html
  3: unchanged   +5/-1 lib/simplegit.rb
```

Как видите, теперь файлы `TODO` и `index.html` проиндексированы, а файл `simplegit.rb` — пока нет. Если в этот момент у вас появится желание отменить индексирование файла `TODO`, воспользуйтесь параметром `3` или `r` (от слова `revert` — возвращать в первоначальное состояние):

```
*** Commands ***
  1: status   2: update  3: revert  4: add untracked
  5: patch    6: diff    7: quit    8: help
What now> 3
      staged   unstaged path
  1:    +0/-1   nothing TODO
  2:    +1/-1   nothing index.html
  3: unchanged   +5/-1 lib/simplegit.rb
Revert>> 1
staged unstaged path
* 1:    +0/-1   nothing TODO
  2:    +1/-1   nothing index.html
  3: unchanged   +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

Если сейчас посмотреть на состояние Git, вы обнаружите, что файл `TODO` удален из области индексирования:

```
*** Commands ***
  1: status   2: update  3: revert  4: add untracked
  5: patch    6: diff    7: quit    8: help
What now> 1
staged unstaged path
  1: unchanged   +0/-1 TODO
  2:    +1/-1   nothing index.html
  3: unchanged   +5/-1 lib/simplegit.rb
```

Чтобы посмотреть, какие же изменения вы проиндексировали, используйте команду **б** или **d** (от слова *diff* — различия). Вы получите список индексированных файлов, и останется указать, вносимые какими файлами изменения вас интересуют. Это аналог ввода в командной строке команды **git diff --cached**:

```
*** Commands ***
  1: status    2: update    3: revert    4: add untracked
  5: patch     6: diff      7: quit      8: help
What now> 6
      staged      unstaged path
  1:      +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

    <p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

<script type="text/javascript">
```

Эти базовые команды позволяют добавлять файлы в область индексирования в интерактивном режиме, несколько упрощая вам жизнь.

Индексирование изменений

Еще Git позволяет индексировать только отдельные части файлов. К примеру, если вы внесли в файл **simplegit.rb** два изменения и хотите проиндексировать только одно из них, делается это очень просто. Введите в строку приглашения интерактивной оболочки параметр **5** или **p** (от слова *patch* — исправление). Вас спросят, какие файлы вы собираетесь индексировать частями; затем для каждой части выбранного файла будут по очереди отображаться добавленные изменения с вопросом, хотите ли вы их индексировать:

```
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,]?
```

Как видите, у вас есть множество вариантов действий. Ввод символа `?` позволяет увидеть перечень доступных вам вариантов:

```
Stage this hunk [y,n,a,d,/,j,J,g,e,?]? ?
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

Как правило, самыми востребованными оказываются параметры `y` и `n`, указывающие, хотите вы или нет индексировать конкретный фрагмент, но иногда требуется индексировать несколько фрагментов вместе или отложить решение на потом. После частичного индексирования файла информация о его состоянии будет выглядеть так:

```
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
```

Посмотрите, как интересно выглядит информация о состоянии файла `simplegit.rb`. Она показывает, что пара строчек проиндексирована, а вторая пара — нет. То есть вы частично проиндексировали этот файл. Теперь можно завершить работу с интерактивным сценарием и запустить команду `git commit` для фиксации состояния частично индексированных файлов.

Впрочем, для частичной индексации файлов вам не обязательно переходить в режим интерактивного добавления — для запуска этого сценария достаточно набрать в командной строке команду `git add -p` или `git add --patch`.

Режим работы с фрагментами файлов также может применяться для частичного восстановления файлов командой `reset --patch`, для перехода к нужному фрагменту при помощи команды `checkout --patch` и для скрытия частей файлов командой `stash save --patch`. Подробно эти команды будут описываться при рассмотрении более специализированных вариантов использования.

Скрытие и очистка

Часто во время работы над проектом, когда все еще находится в беспорядочном состоянии, возникает необходимость перейти в другую ветку и поработать над

другим аспектом. Проблема в том, что фиксировать работу, сделанную наполовину, чтобы позже к ней вернуться, вы не хотите. В такой ситуации вам на помощь придет команда **git stash**.

При ее выполнении все содержимое рабочей папки — то есть отредактированные отслеживаемые файлы и индексированные изменения — сохраняется в стеке незавершенных изменений, откуда вы их можете достать в любой удобный для вас момент.

Скрытие вашей работы

Чтобы посмотреть, как это работает, отредактируйте пару файлов своего проекта и проиндексируйте один из них. Команда **git status** покажет, что состояние проекта изменилось:

```
$ 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
```

Если теперь вы решите перейти в другую ветку, не фиксируя результатов своей работы, их можно просто скрыть. Спрятать все в буфер позволяет команда **git stash** или **git stash save**:

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

В результате рабочая папка оказывается очищенной:

```
$ git status
# На ветке master
nothing to commit (working directory clean)
```

Теперь можно легко менять ветки и работать над другим фрагментом проекта — все изменения хранятся в стеке. Увидеть содержимое стека позволяет команда **git stash list**:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
```

В нашем случае операция скрытия уже два раза выполнялась, поэтому в буфере хранится три варианта наработок. Последняя строка вывода исходной команды

stash подсказывает нам, как вернуть спрятанные в буфер изменения в рабочее состояние. Для этого используется команда **git stash apply**. Если же вы хотите вернуться к работе над версией, сохраненной в буфер ранее, укажите ее номер, например **git stash apply stash@{2}**. Без указания номера Git возвращает в работу последние сохраненные в буфере изменения:

```
$ git stash apply
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   index.html
#       modified:   lib/simplegit.rb
#
```

Как видите, система Git вернула в файлы все изменения, которые вы отменили, сохранив их в буфере. В данном случае мы возвращали содержимое буфера в чистый рабочий каталог и в ту же ветку, из которой они были сохранены; но для успешного возвращения состояния в работу эти условия обязательными не являются. Можно скрыть изменения одной ветки, перейти на другую и попытаться вставить измененное состояние туда. Более того, на момент извлечения содержимого из буфера в рабочей папке могут находиться отредактированные файлы с незафиксированными изменениями — если вставить данные без проблем не получится, Git сообщит о конфликте слияния.

После извлечения информации из буфера файлы, которые до помещения в буфер были проиндексированы, автоматически в это состояние не вернуться. Вам потребуется выполнить команду **git stash apply** с параметром **--index**, которая осуществит повторную индексацию. Чтобы сразу вернуть данные из буфера в исходное состояние, нужно написать:

```
$ git stash apply --index
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
#
```

При этом команда **apply** только возвращает данные в ветке, но из стека они никуда не деваются. Убрать их из стека позволяет команда **git stash drop** с именем удаляемого фрагмента:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
```



```
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Впрочем, существует также команда **git stash pop**, которая возвращает сохраненную в буфере информацию в ветку и немедленно удаляет ее из буфера.

Более сложные варианты скрытия

У команды **stash** существует ряд параметров, которые могут вам пригодиться. Во-первых, достаточно популярен параметр **--keep-index** команды **stash save**. Наличие этого параметра сообщает Git о том, что не нужно скрывать данные, проиндексированные командой **git add**.

Она применяется в случаях, когда вы хотите зафиксировать только часть внесенных изменений, а остальные хотите сохранить, чтобы вернуться к работе с ними позже.

```
$ git status -s
M index.html
M lib/simplelegit.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
```

Также часто возникает ситуация, когда требуется скрыть неотслеживаемые файлы вместе с отслеживаемыми. По умолчанию команда **git stash** сохраняет только файлы из области индексирования. Но параметр **--include-untracked** или **-u** заставляет систему Git сохранять также все неотслеживаемые файлы.

```
$ git status -s
M index.html
M lib/simplelegit.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
$
```

Наконец, флаг **--patch** приводит к тому, что Git вместо скрытия всех модифицированных файлов начинает спрашивать вас в интерактивном режиме, какие файлы вы предпочитаете спрятать, а какие следует оставить в рабочей папке.

```
$ git stash --patch
diff --git a/lib/simplelegit.rb b/lib/simplelegit.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

```

Отмена скрытых изменений

Может возникнуть ситуация, когда после возвращения изменений из буфера вы выполняете некую работу, а потом хотите отменить изменения, внесенные из буфера. Команды `stash unapply` в Git нет, но нужный эффект можно получить, если сначала извлечь связанные с буфером исправления, а затем применить их в реверсивном виде:

```
$ git stash show -p stash@{0} | git apply -R
```

Еще раз напоминаем, что если номер версии не указан в явном виде, Git берет фрагмент данных, скрытый последним:

```
$ git stash show -p | git apply -R
```

Существует возможность создать псевдоним и добавить в свою версию Git команду `stash-unapply`. Например:

```

$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash
$ #... work work work
$ git stash-unapply

```

Создание ветки из скрытого фрагмента

Если скрыть некие наработки, оставить их на некоторое время в буфере, а тем временем продолжить работу в ветке, из которой была скрыта информация, в итоге можно столкнуться с ситуацией, когда просто взять и вернуть данные из буфера не удастся. Ситуация, когда возвращение данных сопровождается попыткой отредактировать файл, который с момента скрытия был модифицирован, приводит

к конфликту слияния, требующему вашего разрешения. Намного проще протестировать скрытые изменения командой `git stash branch`. Она создает новую ветку, переходит к коммиту, в котором вы находились на момент скрытия работы, копирует в новую ветку содержимое буфера и очищает его, если изменения прошли успешно:

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
# On branch testchanges
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

Это удобный способ легко восстановить скрытые изменения и продолжить работу с ними в новой ветке.

Очистка рабочей папки

В некоторых ситуациях лучше не скрывать результаты своего труда или файлы, а избавиться от них. Это можно сделать командой `git clean`.

Удаление требуется, чтобы убрать мусор, сгенерированный путем слияний или внешними инструментами, или чтобы избавиться от артефактов сборки в процессе ее очистки.

С этой командой нужно быть крайне аккуратным, так как она предназначена для удаления неотслеживаемых файлов из рабочей папки. Даже если вы передумаете, восстановить содержимое таких файлов, как правило, будет невозможно. Безопаснее воспользоваться командой `git stash --all`, удаляющей из папки все содержимое, но с последующим его сохранением в буфере.

Предположим, вы все-таки хотите удалить командой `git clean` мусорные файлы или очистить вашу рабочую папку. Для удаления из этой папки всех неотслеживаемых файлов используйте команду `git clean -f -d`, которая полностью очищает папку, убирая не только файлы, но и вложенные папки. Параметр `-f` (сокращение от слова *force* — заставлять) означает принудительное удаление, подчеркивая, что вы действительно хотите это сделать.

Если же вам интересно посмотреть на результаты удаления, используйте параметр `-n`, который говорит системе: «Сымитируй удаление и покажи мне, что будет удалено».

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

По умолчанию команда **git clean** удаляет только неотслеживаемые файлы, не добавленные в список игнорируемых. Любой файл, имя которого совпадает с шаблоном в файле **.gitignore**, сохранится. Чтобы удалить и их, например убрав все генерируемые в процессе сборки файлы с расширением **.o** с целью полной очистки сборки, добавьте к команде **clean** параметр **-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 точно не знаете, к каким последствиям приведет выполнение команды **git clean**, всегда добавляйте к ней параметр **-n**, чтобы посмотреть на результат перед тем, как указать параметр **-f** и проводить реальное удаление. Другой способ контроля над процессом дает параметр **-i**, включающий интерактивный режим.

Вот пример выполнения команды 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>
```

Это дает вам возможность по очереди рассмотреть все удаляемые файлы или в интерактивном режиме указать шаблон удаления.

Подпись

Благодаря шифрованию система Git является безопасной, но полностью она не защищена. Для проверки того факта, что пересылаемые через Интернет файлы исходят из доверенного источника, в Git существует несколько вариантов подписи и проверки исходного кода при помощи программы GPG.

Знакомство с GPG

Чтобы иметь возможность что-то подписывать, вам прежде всего нужен настроенный в программе GPG и установленный личный 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 ключа у вас пока нет, сгенерируйте его командой **gpg --gen-key**.

```
gpg --gen-key
```

Чтобы заставить Git использовать для подписи ваш закрытый ключ, установите значение конфигурационного параметра **user.signingkey**:

```
git config --global user.signingkey 0A46826A
```

Теперь Git будет по умолчанию использовать этот ключ для подписи тегов и коммитов.

Подпись тегов

После настройки закрытого GPG-ключа его можно использовать для подписи новых тегов. Достаточно заменить параметр **-a** параметром **-s**:

```
$ 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 запустить для этого тега команду **git show**, вы увидите присоединенную к тегу 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/09PFE51KPVP1anr6q1v4/Ut
LQxfojUWiLQdg2ESJItkuweYg+kc3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLowZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfvfciMnSDeSvzCpWAH17h8Wj6hhqePmLm9lAYqnKp
855B/1SSQuEAjRZgI4IexpZoeKGVdptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RlYsgqjcpT8+iQM1Pb1gFHR4XAhuQnN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----
```

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

changed the verison number

Проверка тегов

Проверка подписанного тега выполняется командой `git tag -v [имя-тега]`. Эта команда использует для проверки GPG. Для ее корректной работы нужно, чтобы в вашем хранилище ключей присутствовал открытый ключ автора, поставившего подпись:

```
$ git tag -v v1.4.2.1
object 883653babb8ee7ea23e6a5c392bb739348b1eb61
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 открытый ключ автора у вас отсутствует, вы увидите нечто 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 (от v1.7.9 и выше) появилась возможность подписывать отдельные коммиты. Если вы предпочитаете подписывать не теги, а сами коммиты, добавьте к команде `git commit` параметр `-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 команда `git log` снабжена параметром `--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
```

Кроме того, существует параметр форматирования `%G?`, заставляющий команду `git log` проверять все обнаруженные подписи и выводить их в виде списка:

```
$ git log --pretty="format:%h %G? %aN %s"

5c3386c G Scott Chacon signed commit
ca82a6d N Scott Chacon changed the verison number
085bb3b N Scott Chacon removed unnecessary test code
a11bef0 N Scott Chacon first commit
```

Здесь мы видим, что подписан и действителен только последний коммит, а все остальные — нет.

Начиная с версии 1.8.3, команды `git merge` и `git pull` с помощью параметра `--verify-signatures` можно заставить проверять слияния и отклонять их, если коммит не содержит доверенной GPG-подписи.

Если вы воспользуетесь этим параметром при слиянии с веткой, содержащей неподписанные и недействительные коммиты, слияние выполнено не будет:

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

Если же ветка, с которой осуществляется слияние, содержит только корректно подписанные коммиты, команда `merge` сначала покажет все проверенные ею подписи, а потом перейдет непосредственно к слиянию:

```
$ 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(+)
```

Можно также воспользоваться параметром `-S` команды `git merge` для подписи коммита, образующегося в результате слияния. В следующем примере выполняется проверка наличия подписи у каждого предназначенного для слияния коммита, а затем итоговый коммит слияния получает подпись:

```
$ 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(+)
```

Подпись должна быть у всех

Подписывать теги и коммиты важно и нужно, но если вы решите ввести эту операцию в обычный рабочий процесс, следует убедиться, что делать это умеют все члены группы. В противном случае придется потратить уйму времени, объясняя коллегам, как перезаписать их коммиты в подписанном виде. Перед тем как внедрить процедуру подписи в рабочий процесс, убедитесь, что вы хорошо понимаете, что такое GPG и какие преимущества дает эта программа.

Поиск

Практически при любом размере базы кода рано или поздно возникает задача поиска места, в котором вызывается или определяется функция, или поиска истории метода. Для поиска в коде и коммитах, хранящихся в базе данных Git, есть ряд полезных инструментов. Некоторые из них мы рассмотрим в этом разделе.

Команда `git grep`

В Git есть команда `git grep`, позволяющая искать строки и регулярные выражения в дереве коммитов или в рабочей папке. Для иллюстрации ее применения мы возьмем код самой системы Git.

По умолчанию команда `git grep` выполняет поиск среди файлов вашей рабочей папки. Параметр `-n` указывает максимальное количество обнаруженных совпадений, которые будут фигурировать в выводимых командой данных:

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8:     return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct tm *result)
compat/gmtime.c:16:     ret = gmtime_r(timep, result);
compat/mingw.c:606:struct tm *gmtime_r(const time_t *timep, struct tm *result)
compat/mingw.h:162:struct tm *gmtime_r(const time_t *timep, struct tm *result);
date.c:429:         if (gmtime_r(&now, &now_tm))
date.c:492:         if (gmtime_r(&time, tm)) {
git-compat-util.h:721:struct tm *git_gmtime_r(const time_t *, struct tm *);
git-compat-util.h:723:#define gmtime_r git_gmtime_r
```

У команды `grep` есть еще ряд интересных параметров.

Например, можно заставить Git обобщить выводимые командой данные, показав только файлы, в которых обнаружены совпадения, вместе с количеством этих совпадений. Для этого вам потребуется параметр `--count`:

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:2
git-compat-util.h:2
```

Чтобы посмотреть, в каком методе или функции было обнаружено совпадение, используйте параметр `-p`:

```
$ git grep -p gmtime_r *.c
date.c:static int match_multi_number(unsigned long num, char c, const char *date,
    char *end, struct
    tm *tm)
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)) {
```

Вывод этой команды показывает, что метод `gmtime_r` вызывается функциями `match_multi_number` и `match_digit`, находящимися в файле `date.c`.

Флаг `--and` позволяет искать сложные комбинации строк. Он гарантирует наличие в одной строке нескольких совпадений. Давайте, к примеру, найдем в базе Git-кода более старой версии 1.8.0 строки, содержащие определение константы, посредством строковых переменных `LINK` или `BUF_MAX`.

Заодно мы воспользуемся параметрами `--break` и `--heading`, чтобы придать выводимым данным более читабельный вид:

```
$ 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 ряд преимуществ перед обычными командами поиска `grep` и `ack`. Во-первых, она работает действительно быстро, а во-вторых, позволяет искать не только в рабочей папке, но и в произвольном дереве системы Git. Скажем, в предыдущем примере мы искали термины в более старой, а не в текущей версии Git-кода.

Поиск в Git-журнале

Иногда нужно увидеть не текущее местоположение некоторого выражения, а его место в прошлом или даже место, где оно впервые появилось. Команда `git log` обладает мощным инструментарием, позволяющим искать коммиты по содержанию их сообщений фиксации или даже по содержанию появившихся в них изменений.

Предположим, нам нужно определить, когда появилась константа `ZLIB_BUF_MAX`. При помощи параметра `-S` мы попросим Git показать нам только те коммиты, в которых эта строка добавлялась или удалялась:

```
$ git log -SZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

Посмотрев на изменения, вносимые этими коммитами, мы обнаружим, что в коммите `ef49a7a` эта константа появилась, а в коммите `e01503b` была отредактирована.

Более сложные поисковые запросы реализуются через передаваемые в параметр `-G` регулярные выражения.

Поиск по строкам кода

Существует еще один вариант поиска в журнале, который часто оказывается довольно востребованным — поиск в истории строк. Эта возможность появилась относительно недавно и пока не приобрела широкой известности. Она реализуется добавлением к команде `git log` параметра `-L` и позволяет увидеть историю функции или строки кода в вашей кодовой базе.

К примеру, для просмотра изменений, вносившихся в функцию `git_deflate_bound` в файле `zlib.c`, следует написать:

```
git log -L :git_deflate_bound:zlib.c
```

Эта команда определит границы функции, а затем после поиска в истории покажет в обратном хронологическом порядке все вносившиеся в нее в виде набора исправлений изменения с 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 система Git не может найти способ сопоставить функцию или метод на вашем языке программирования, следует использовать регулярные выражения. К примеру, следующая команда выполнит такой же поиск, как и в показанном ранее случае:

```
git log -L 'unsigned long git_deflate_bound',/^}/:zlib.c
```

Для получения таких же выводимых данных можно также указать интервал строк или номер строки.

Перезапись истории

Во время работы с Git периодически возникает необходимость внести исправления в историю коммитов. Система Git примечательна тем, что позволяет вносить изменения в самый последний момент. Благодаря наличию области индексирования перед фиксацией состояния вы выбираете, какие файлы должны войти в коммит. Вы можете скрыть наработки, работу над которыми пока не хотите продолжать, или можете внести изменения в сделанные коммиты, придав их истории совсем другой вид. Этот процесс включает в себя изменение порядка следования коммитов, редактирование сообщений фиксации или редактирование входящих в коммит

файлов, объединение набора коммитов или, наоборот, разбиение одного коммита на несколько отдельных и даже полное удаление коммитов. И все это делается до выкладывания ваших наработок в общий доступ.

В этом разделе вы познакомитесь со способами решения всех этих задач и научитесь перед публикацией данных приводить историю коммитов в нужный вам вид.

Редактирование последнего коммита

Редактирование последнего коммита представляет собой, наверное, самый распространенный вариант переписывания истории. Как правило, при этом выполняются два основных действия: меняется сообщение фиксации или только что записанный снимок состояния путем добавления, изменения или удаления файлов.

Отредактировать сообщение фиксации последнего коммита очень просто:

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

Откроется текстовый редактор с готовым к редактированию сообщением. Как только вы сохраните результаты редактирования и закроете редактор, будет записан новый вариант коммита с новым сообщением фиксации, и с этого момента он будет считаться последним.

Если вы решите изменить снимок состояния, добавив или отредактировав файлы, например, потому что в момент фиксации вы забыли добавить в коммит только что созданный файл, порядок действий будет аналогичным. Вы индексируете дополнительные изменения, отредактировав файл и выполнив для него команду **git add** (или **git rm** для отслеживаемого файла), а затем — команду **git commit --amend**. Последняя берет содержимое области индексирования и превращает его в снимок состояния для нового коммита.

Эту технику нужно применять с осторожностью, так как она меняет контрольную сумму SHA-1 коммита. Как и в случае с небольшим перемещением, нельзя править последний коммит, если вы уже отправили его в общий доступ.

Редактирование нескольких сообщений фиксации

Для редактирования коммита, сделанного в более ранний момент времени, потребуются более сложные инструменты. Специального инструмента для редактирования истории в Git не существует, но можно воспользоваться процедурой перемещения и поместить коммиты там, где изначально находился их указатель HEAD, а не в другое место. Инструмент **rebase** поддерживает интерактивный режим, то есть каждый коммит, который вы хотите отредактировать, показывается вам отдельно, и можно поменять сообщение фиксации, добавить в него файлы или выполнить другие действия. Для входа в интерактивный режим команду **git rebase**

запускают с параметром `-i`. Следует указать, насколько давние коммиты вы собираетесь переписывать, сообщив команде, для какого коммита будет выполняться перемещение.

Например, чтобы отредактировать сообщения фиксации последних трех коммитов или сообщения только для некоторых коммитов из этой группы, в качестве аргумента команде `git rebase -i` передается родитель последнего коммита, который вы собираетесь менять. В данном случае это `HEAD~2^` или `HEAD~3`. Второй вариант записи запомнить проще, потому что вы пытаетесь внести изменения в три последних коммита; но при этом обратите внимание, что на самом деле вы нацеливаетесь на четвертый сзади коммит — родителя последнего интересующего вас коммита:

```
$ git rebase -i HEAD~3
```

Еще раз напоминаем, что это команда служит для перемещения, то есть будут переписаны все коммиты в диапазоне `HEAD~3..HEAD` вне зависимости от того, меняете вы для них сообщение или нет. Ни в коем случае не включайте в этот набор коммиты, уже отправленные на центральный сервер, — сделав так, вы запутаете других разработчиков, предоставив им альтернативную версию уже имеющихся изменений.

Эта команда откроет вам текстовый редактор со списком коммитов следующего вида:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Важно запомнить, что эти коммиты перечисляются в обратном порядке по сравнению с тем, что вы привыкли видеть в выводимых командой `log` данных. Запустив команду `log`, вы получите нечто подобное:

```
$ git log --pretty=format:»%h %s» HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

Обратите внимание на обратный порядок. Запускаемый сценарий начинает работу с коммита, фигурирующего в командной строке (**HEAD~3**), и сверху вниз воспроизводит появившиеся в каждом из коммитов изменения. Самый старый коммит отображается сверху, так как именно он будет воспроизводиться первым.

Вам нужно отредактировать сценарий таким образом, чтобы на коммитах, в которые вы хотите внести изменения, он останавливался. Для этого замените у каждого такого коммита слово **pick** на слово **edit**. К примеру, для редактирования сообщения фиксации только в третьем коммите в файл следует внести вот такие изменения:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Когда вы сохраните этот файл и закроете редактор, Git перебросит вас к последнему коммиту в списке и откроет для вас командную строку со следующим сообщением:

```
$ git rebase -i HEAD~3
Stopped at 7482e0d... updated the gemspec to hopefully work better
You can amend the commit now, with
git commit --amend
Once you're satisfied with your changes, run
git rebase --continue
```

Эта инструкция точно сообщает вам, что надо делать. Введите:

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

Измените сообщение фиксации и закройте редактор. Затем запустите команду:

```
$ git rebase --continue
```

Данная команда автоматически применяет остальные два коммита и на этом заканчивает работу. Поменяв **pick** на **edit** в большем числе строк, вы сможете повторить эти шаги для всех коммитов, рядом с которыми будет фигурировать слово **edit**. Каждый раз Git будет останавливаться, предоставляя вам возможность внести в коммит исправления и после завершения редактирования продолжить свою работу.

Изменение порядка следования коммитов

Перемещение в интерактивном режиме может также использоваться для изменения порядка следования коммитов или их удаления. К примеру, чтобы удалить коммит, связанный с добавлением файла **cat-file**, и изменить порядок следования двух оставшихся коммитов, нужно изменить сценарий перемещения. Вот исходный вариант:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

А вот новый вариант сценария:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

После сохранения новой версии сценария и выхода из редактора система Git перемотает ветку до предка этих коммитов, применит коммиты **310154e** и **f7f3f6d**, после чего остановится. Как видите, вы эффективно изменили порядок следования указанных коммитов и удалили коммит, связанный с добавлением файла `cat-file`.

Объединение коммитов

Инструмент интерактивного перемещения позволяет также превратить несколько коммитов в один коммит. Сценарий добавляет в сообщение перемещения полезные инструкции:

```
#
# Commands:
# p, pick = use commit
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

Если вместо **pick** или **edit** указать **squash**, Git применит указанное изменение и непосредственно предшествующее ему изменение и заставит вас объединить сообщения фиксации. Фактически, для того чтобы превратить эти три коммита в один, вам потребуется следующий сценарий:

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

После того как вы сохраните сценарий и выйдете из редактора, Git применит все три изменения и вернет вас в редактор, чтобы вы выполнили объединение трех сообщений фиксации:

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:

added cat-file
```

После сохранения результатов редактирования у вас появится единственный коммит, содержащий изменения из трех исходных коммитов.

Разбиение коммита

Процедура разбиения коммита отменяет внесенные им изменения, затем индексирует его по частям и фиксирует столько раз, сколько коммитов вы в итоге хотите получить. Предположим, вы хотите разбить средний коммит на три части. Вместо коммита «updated README formatting and added blame» (обновлен файл README, отформатирован и добавлен файл blame) вы хотите сделать так, чтобы первый коммит обновлял и форматировал файл README, а второй добавлял файл blame. Для этого в сценарии **rebase -i** нужно заменить инструкцию для разбиваемого коммита на **edit**:

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

После того как сценарий вернет вас в командную строку, вы отмените действие коммита и создадите из этих отмененных изменений нужное количество новых коммитов. Как только вы сохраните сценарий и выйдете из редактора, Git перейдет к родителю первого коммита из списка, применит первый коммит (**f7f3f6d**), затем второй (**310154e**) и вернет вас в консоль. Здесь изменения, внесенные этим коммитом, можно отменить командой **git reset HEAD^**, которая эффективно возвращает все в предшествующее состояние, причем модифицированные файлы оказываются неиндексированными. После этого можно начать индексацию и фиксацию файлов, пока у вас не появится несколько коммитов. Затем останется выполнить команду **git rebase --continue**:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git применит последний коммит (**a5f4a0d**) из этого сценария, и история приобретет такой вид:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Имейте в виду, что это меняет контрольные суммы SHA всех коммитов в списке, поэтому важно следить за тем, чтобы список содержал только коммиты, которые еще не отправлялись в общее хранилище.

Последнее средство: команда `filter-branch`

Существует еще один способ переписывания истории, к которому прибегают, когда при помощи сценария нужно внести изменения в большое количество коммитов, например везде поменять ваш адрес электронной почты или убрать какой-то файл из всех коммитов. В таких случаях вам на помощь приходит команда `filter-branch`, позволяющая переписывать большие фрагменты истории. Как обычно бывает в подобной ситуации, ее можно использовать только для проектов, которые еще не попали в общий доступ и переписываемые коммиты не служат основой ничьей работы. Впрочем, эта команда может быть весьма полезной. Далее мы рассмотрим примеры ее применения, которые дадут вам представление о ее возможностях.

Удаление файла из всех коммитов

Часто случается так, что пользователь, необдуманно выполнив команду `git add`, включил в коммиты огромный бинарный файл и его требуется отовсюду удалить. Или вы можете сами включить в коммит файл, содержащий пароль, а потом решить сделать проект открытым. Поиск во всей истории проекта вам поможет выполнить утилита `filter-branch`. Вот как выглядит удаление из истории проекта файла `passwords.txt`. Вам потребуется параметр `--tree-filter`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

Параметр `--tree-filter` заставляет выполнить указанную команду после перехода к каждой следующей версии проекта, а затем повторно фиксирует результаты. В этом случае вы удаляете файл `passwords.txt` из каждого снимка состояния системы вне зависимости от того, существует он или нет. Для удаления всех случайно зафиксированных резервных копий файла, созданных текстовым редактором, можно написать:

```
git filter-branch --tree-filter 'rm -f *~' HEAD
```

Вы сможете посмотреть, как Git переписывает деревья и коммиты, а затем перемещает указатель в конец ветки. В общем случае это рекомендуется делать в тестовой ветке, а потом, если выяснится, что именно такой результат вам и нужен, выполнить полную перезагрузку ветки `master`. Чтобы команда `filter-branch` работала со всеми вашими ветками, добавьте к ней параметр `--all`.

Превращение вложенной папки в корневую

Предположим, в результате импорта из другой системы контроля версий у вас появился набор вложенных папок, не имеющих никакого смысла (`trunk`, `tags` и т. п.). Чтобы превратить папку `trunk` в корневую папку нового проекта,

в которой будут сохраняться все коммиты, вам опять потребуется команда **filter-branch**:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Теперь все время в роли новой корневой папки проекта будет выступать папка **trunk**. Заодно Git автоматически удалит коммиты, не связанные с этой папкой.

Изменение адресов электронной почты в глобальном масштабе

Также часто возникает ситуация, когда пользователь перед началом работы забывает воспользоваться командой **git config** и указать свой адрес электронной почты или, к примеру, для открытия проекта с открытым исходным кодом нужно заменить все адреса вашим персональным адресом. В подобных случаях на помощь вам приходит все та же команда **filter-branch**, позволяющая одновременно поменять адреса электронной почты в наборе коммитов. Эту операцию следует проводить аккуратно, чтобы замена не затронула не принадлежащие вам адреса электронной почты, поэтому рекомендуем воспользоваться параметром **--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 по очереди переписывает все коммиты, вставляя туда ваш новый адрес электронной почты. Так как коммиты содержат значения SHA-1 своих предков, эта команда меняет значения SHA всех коммитов в истории, а не только тех, в которых был обнаружен указанный вами электронный адрес.

Команда reset

Перед тем как мы перейдем к рассмотрению более специализированных инструментов, поговорим о командах **reset** и **checkout**. На начальных этапах знакомства с Git они кажутся одними из самых непонятных. Их богатая функциональность создает впечатление, что понять и научиться правильно использовать

эти команды невозможно. Мы же попробуем пояснить их смысл при помощи простой метафоры.

Три дерева

Понять суть команд `reset` и `checkout` будет проще, если представить, что Git управляет содержимым трех деревьев. В данном случае термин «дерево» относится не к специализированной структуре данных, а к обычному набору файлов. (Область индексирования далеко не всегда ведет себя как дерево, но для наших целей сейчас проще всего представить ее именно так.)

В обычном режиме система Git управляет и манипулирует тремя деревьями:

Дерево	Назначение
HEAD	Снимок последнего коммита, следующий родитель
Индекс	Снимок, предложенный для следующего коммита
Рабочая папка	Изолированная программная среда (песочница)

Указатель HEAD

HEAD — это указатель на текущую ветку, которая в свою очередь является указателем на последний сделанный в ней коммит. Это означает, что HEAD будет родителем вашего следующего коммита. В общем случае проще всего воспринимать его как снимок вашего последнего коммита.

Увидеть, как выглядит этот снимок, достаточно просто. Вот пример получения списка содержимого папки и контрольных сумм SHA для каждого файла в снимке HEAD:

```
$ git cat-file -p HEAD
tree cfd3bf379e4f8dba8717dee55aab78aef7f4daf
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
```

Низкоуровневые команды `cat-file` и `ls-tree` являются служебными и при ежедневной работе не требуются, но в данном случае именно они помогут нам понять, что происходит.

Индекс

В области индексирования находятся снимки состояний, *которые вы предлагаете зафиксировать*. Именно из этой области Git берет данные для команды `git commit`.

Система Git заполняет область индексирования списком содержимого всех файлов, в последний раз выгруженных в рабочую папку. Вид этого содержимого такой же, как на момент выгрузки. После этого вы заменяете некоторые из этих файлов новыми версиями, а команда `git commit` превращает их в дерево для нового коммита:

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0    README
100644 8f94139338f9404f26296befa88755fc2598c289 0    Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0    lib/simplegit.rb
```

Для получения вышеуказанных данных мы пользуемся служебной командой `ls-files`, которая и отображает текущее содержимое нашего индекса.

С технической точки зрения индекс не является древовидной структурой — на самом деле он реализован как манифест, из которого вся иерархия удалена, — но для наших целей такое приближение вполне допустимо.

Рабочая папка

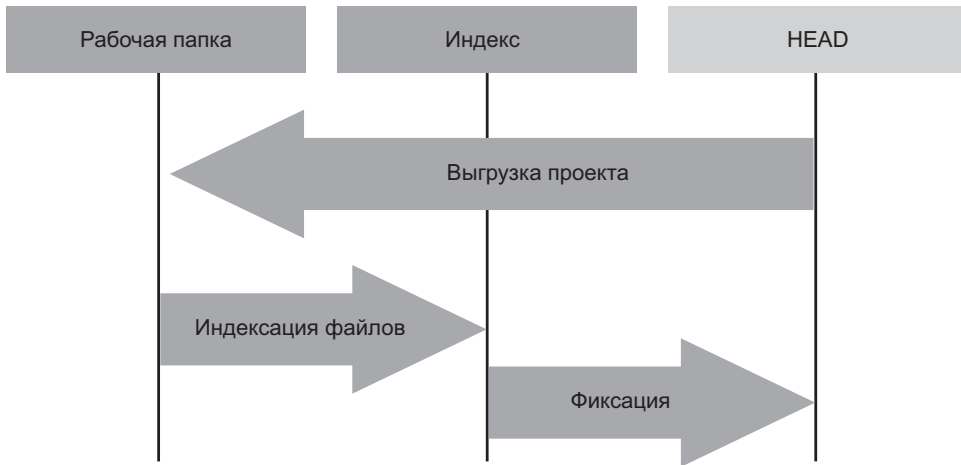
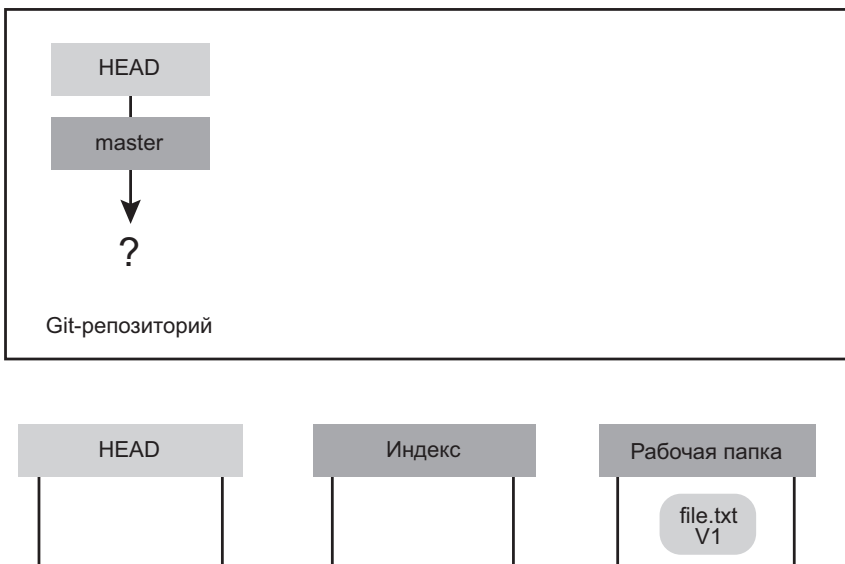
Последним нашим деревом является рабочая папка. Два первых дерева хранят свое содержимое в рациональном, но неудобном виде — внутри папки `.git`. Рабочая папка распаковывает их, превращая в реальные файлы, что упрощает процесс их редактирования. Рабочую папку можно считать изолированной программной средой, или песочницей, в которой можно посмотреть, как будут выглядеть изменения перед тем, как их зафиксируют в области индексирования, а затем и в истории:

```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb
1 directory, 3 files
```

Рабочий процесс

Основным назначением системы Git является запись снимков последовательно улучшающихся состояний проекта, реализуемая через управление нашими тремя деревьями (рис. 7.2).

Предположим, вы перешли в новую папку, в которой присутствует единственный файл. Назовем этот файл именем `v1` (рис. 7.3). Выполним команду `git init`, которая создаст Git-репозиторий со ссылкой `HEAD` на несуществующую пока ветку (ветки `master` у нас еще нет).

**Рис. 7.2****Рис. 7.3**

Пока что содержимым обладает только дерево рабочей папки.

Теперь мы хотим зафиксировать этот файл, поэтому прибегаем к команде **git add**, которая берет содержимое рабочей папки и копирует его в области индексирования (рис. 7.4).

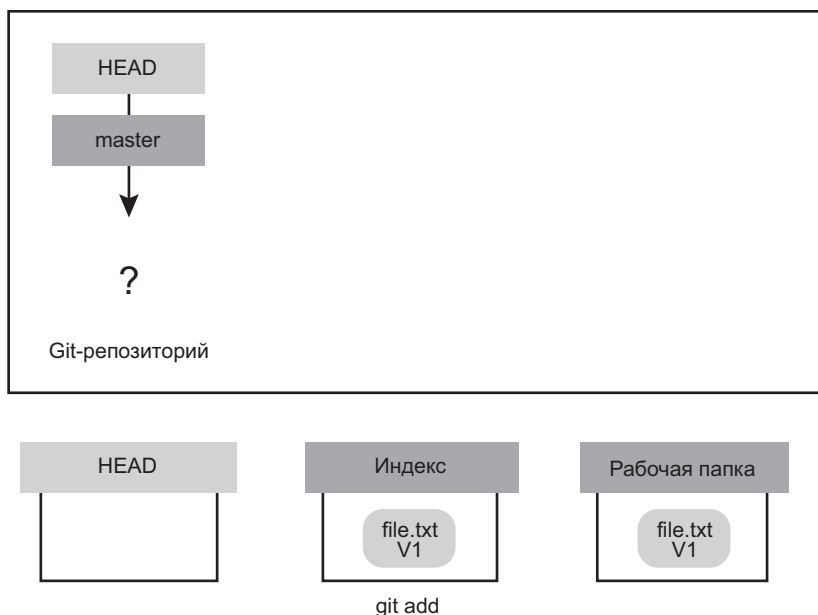


Рис. 7.4

Далее мы запускаем команду **git commit**, которая берет содержимое области индексирования, сохраняет его как постоянный снимок, создавая указывающий на этот снимок объект-коммит, и обновляет ветку **master** таким образом, чтобы она тоже была нацелена на этот коммит (рис. 7.5).

Команда **git status** пока не покажет нам никаких изменений, так как все три дерева идентичны.

Но мы хотим отредактировать этот файл и зафиксировать его новое состояние. Для этого мы вернемся в начало рабочего процесса и внесем изменения в файл, находящийся в рабочей папке. Эту версию файла мы назовем v2 (рис. 7.6).

Если сейчас запустить команду **git status**, мы увидим, что файл выделен красным и помечен как «изменения, не индексированные для коммита», так как именно наличие этого элемента отличает область индексирования от рабочей папки. Но после выполнения команды **git add** файл появится в области индексирования (рис. 7.7).

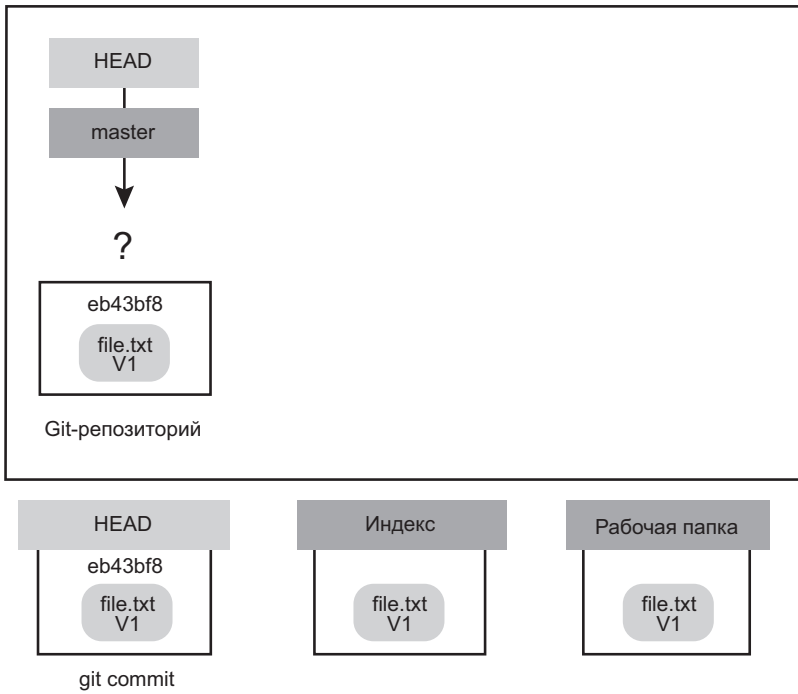


Рис. 7.5

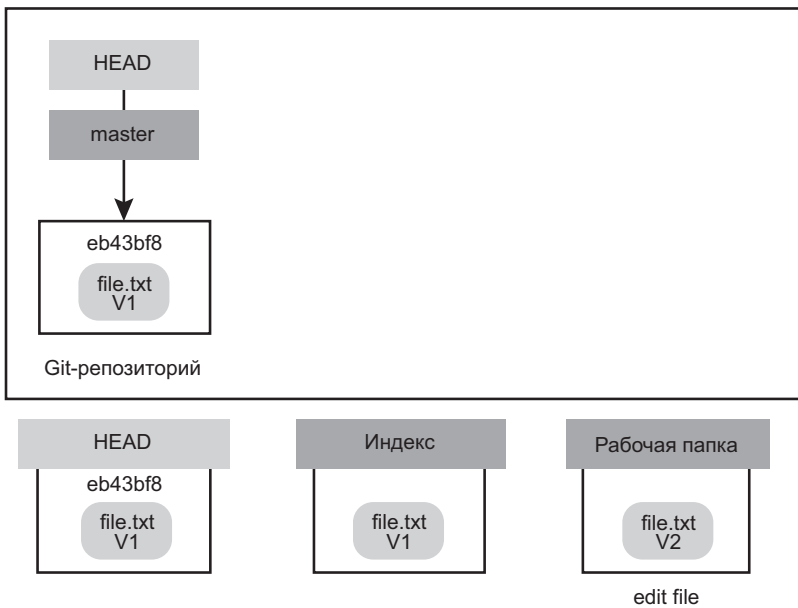
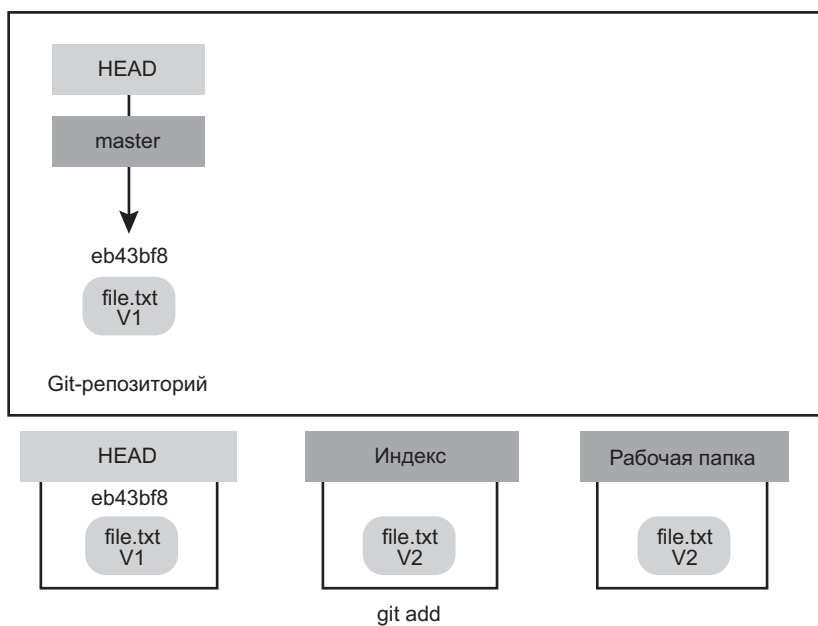
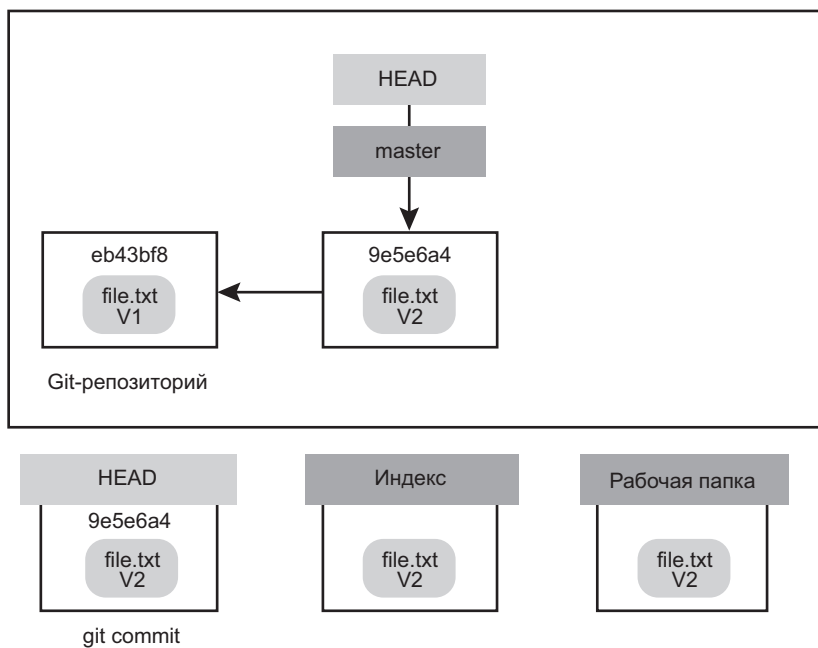


Рис. 7.6

**Рис. 7.7****Рис. 7.8**

Теперь команда `git status` покажет нам этот файл, выделенный зеленым цветом с пометкой «изменения, предназначенные для фиксации», так как в данном случае у нас различаются индекс и HEAD. Другими словами, данные, предложенные для следующего коммита, отличаются от нашего последнего коммита. Окончательно зафиксировать изменения нам поможет команда `git commit` (рис. 7.8). После этого команда `git status` опять перестанет давать результат, так как все три дерева снова станут одинаковыми. Процессы перехода с ветки на ветку и клонирования происходят по аналогичной схеме. При переходе в новую ветку система переключает туда же указатель HEAD, заполняет область индексирования снимком коммита из этой ветки и копирует содержимое этой области в рабочую папку.

Роль команды reset

В этом контексте становится более понятной и команда `reset`.

Предположим, мы снова отредактировали файл `file.txt` и в третий раз зафиксировали его состояние. Теперь его история выглядит так, как показано на рис. 7.9.

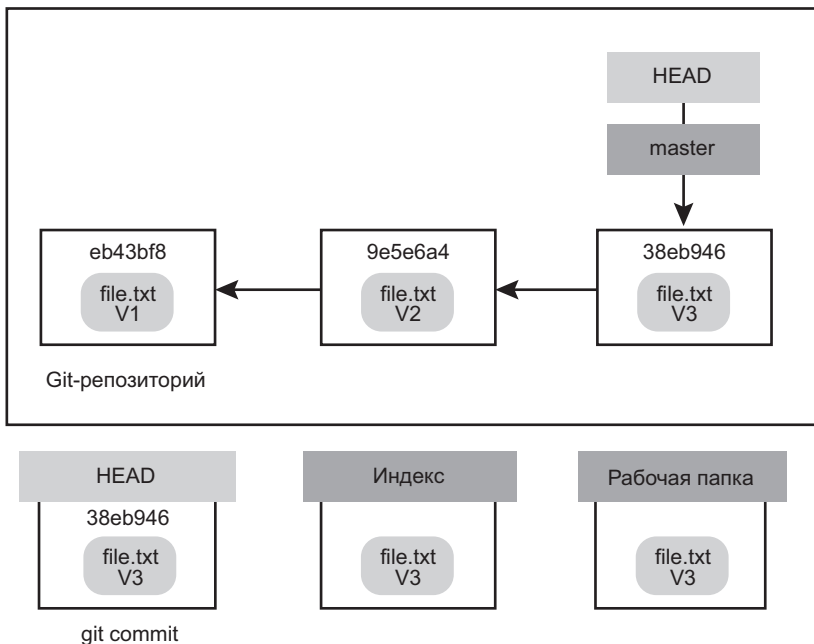


Рис. 7.9

А теперь посмотрим, что происходит при вызове команды `reset`. Эта команда простым и предсказуемым образом воздействует непосредственно на три дерева. При этом выполняются три основные операции.

Шаг 1. Перемещение указателя HEAD

Первым делом команда **reset** переместит элемент, на который нацелен указатель **HEAD**. При этом не происходит изменения положения самого указателя **HEAD** (как это бывает при переключении); команда **reset** двигает ветку, на которую нацелен данный указатель. Фактически если указатель **HEAD** нацелен на ветку **master** (то есть в настоящий момент вы находитесь в ветке **master**), команда **git reset 9e5e64a** первым делом сделает так, что ветка **master** начнет указывать на коммит **9e5e64a** (рис. 7.10).

Какую бы форму команды **reset** с указанием коммита вы ни использовали, этот шаг всегда будет первым. В случае команды **reset --soft** этот же шаг окажется и последним.

Теперь посмотрим на диаграмму и попробуем понять, что произошло. Фактически был отменен результат действия последней команды **git commit**. Команда **git commit** заставляет Git создать новый коммит и перемещает в него ветку, на которую нацелен указатель **HEAD**. Если команда **reset** делается для **HEAD~** (родителя **HEAD**), то ветка просто перемещается в предшествующее положение, никак не затрагивая ни область индексирования, ни рабочую папку. Вы можете обновить индекс и снова запустить команду **git commit**, получив в итоге результат, который дала бы вам команда **git commit --amend**.

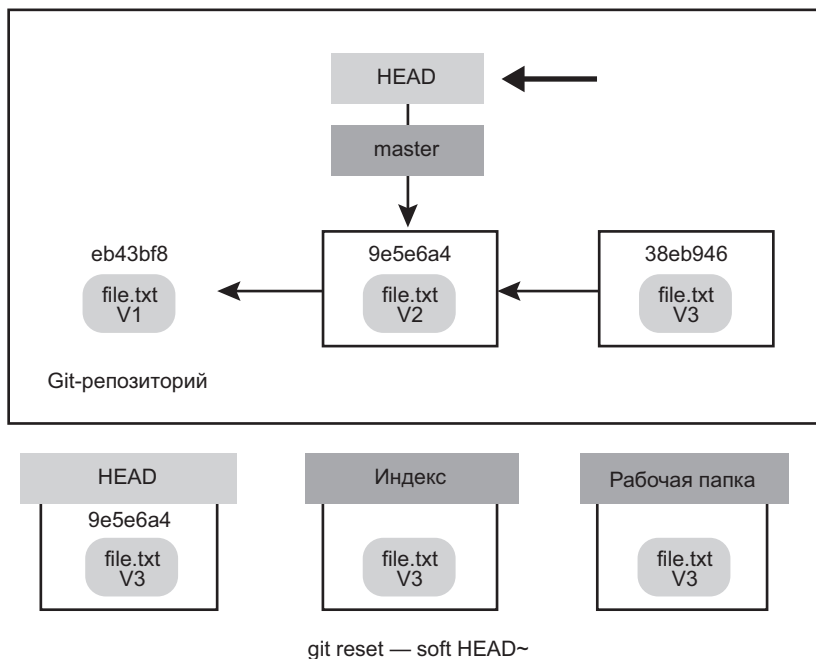


Рис. 7.10

Шаг 2. Обновление индекса

Выполнив прямо сейчас команду `git status`, вы увидите выделенные зеленым цветом изменения между областью индексации и тем местом, куда теперь нацелен указатель `HEAD`.

Следующим действием команды `reset` станет обновление области индексирования путем добавления туда содержимого снимка, на который нацелен указатель `HEAD`.

Наличие параметра `--mixed` останавливает работу команды `reset` на этом шаге. Аналогичное поведение предлагается по умолчанию, то есть в ситуации, когда параметры отсутствуют (в нашем случае это будет команда `git reset HEAD~`).

Теперь снова обратимся к диаграмме и постараемся понять, что произошло (рис. 7.11). Команда не только отменила последний коммит, но и убрала из области индексирования все находившиеся там файлы. То есть вы вернулись к состоянию, которое мы имели до выполнения всех команд `git add` и `git commit`.

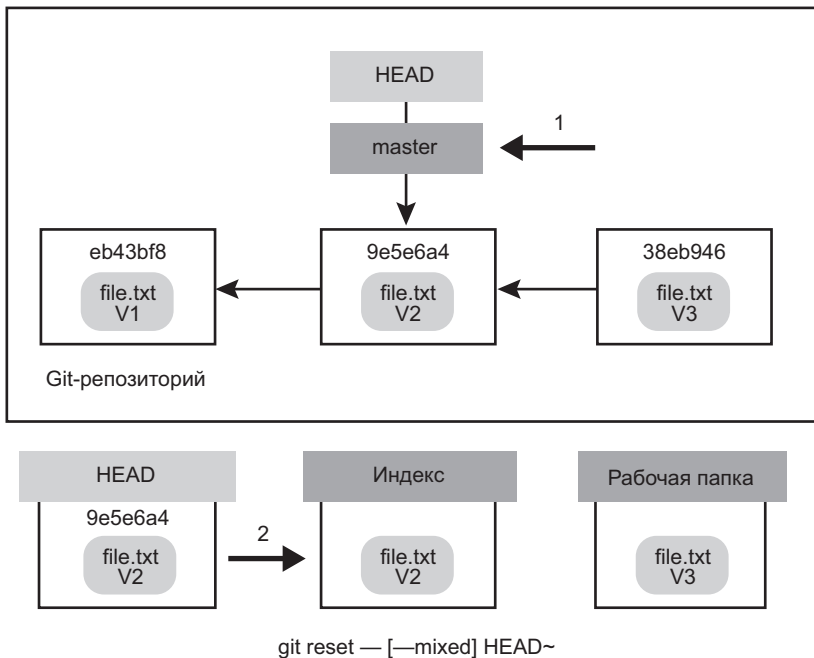


Рис. 7.11

Шаг 3. Обновление рабочей папки

Третьим действием команды `reset` станет приведение рабочей папки к виду, который имеет область индексирования. До этой стадии команда работает при наличии параметра `--hard` (рис. 7.12).

Давайте подумаем, что же сейчас произошло. Вы убрали свой последний коммит, результаты работы команд `git add` и `git commit` и все наработки из рабочей папки. Имейте в виду, что только параметр `--hard` делает команду `reset` по-настоящему опасной. Это один из немногочисленных случаев, когда Git реально удаляет данные. Результат применения всех остальных вариантов команды `reset` отменяется достаточно легко, но наличие параметра `--hard` делает ваши действия необратимыми из-за принудительной перезаписи файлов в рабочей папке. В рассматриваемом в этом разделе случае версия файла `v3` пока еще существует в виде коммита в нашей базе данных Git, и есть возможность вернуться к ней, просматривая журнал при помощи команды `reflog`. Но если вы не фиксировали эту версию, система Git перезапишет файл, и восстановить его будет уже невозможно.

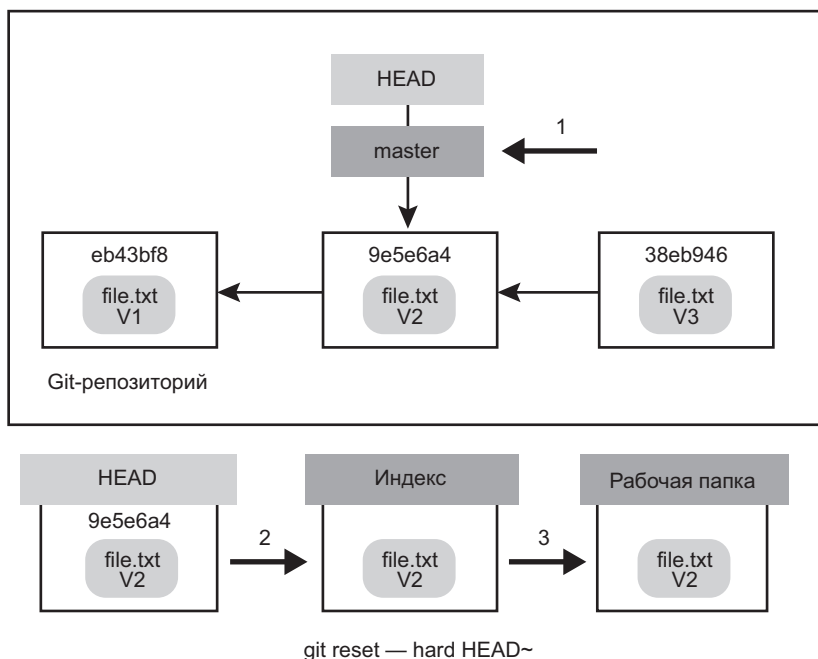


Рис. 7.12

Заключение

Команда `reset` в определенном порядке переписывает три дерева, останавливаясь в указанном вами месте:

1. Перемещает ветку, на которую нацелен указатель `HEAD` (и останавливается при наличии параметра `--soft`).
2. Приводит вид области индексирования в соответствии с данными, на которые нацелен указатель `HEAD` (и без параметра `--hard` на этом останавливается).
3. Приводит вид рабочей папки в соответствии с видом области индексирования.

Команда reset с указанием пути

Итак, вы познакомились с работой базовой формы команды **reset**, но кроме этого ей можно передавать путь. В этом случае команда пропускает первый этап и ограничивает свою работу определенным файлом или набором файлов. Такое поведение имеет смысл, ведь указатель **HEAD** не может быть нацелен частично на один коммит, а частично на другой. А вот частичное обновление области индексирования и рабочей папки — вполне реальная операция, поэтому команда **reset** сразу переходит к этапам 2 и 3.

Итак, предположим, вы выполнили команду **git reset file.txt**. В такой форме (ведь вы не указали ни контрольную сумму SHA коммита, ни ветку, ни параметр **--soft** или **--hard**) команда является сокращением для команды **git reset --mixed HEAD file.txt**, которая:

1. Перемещает ветку, на которую нацелен указатель **HEAD** (пропущено).
2. Приводит содержимое области индексирования к такому же виду, как и в месте, на которое нацелен указатель **HEAD** (на этом команда останавливается).

То есть фактически происходит обычное копирование файла **file.txt** из места, отмеченного указателем **HEAD**, в область индексирования (рис. 7.13).

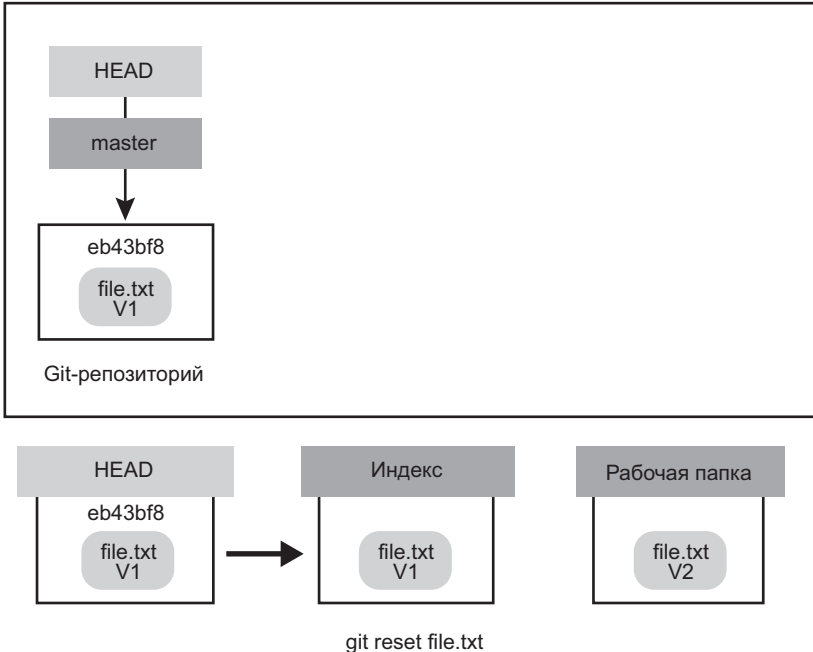
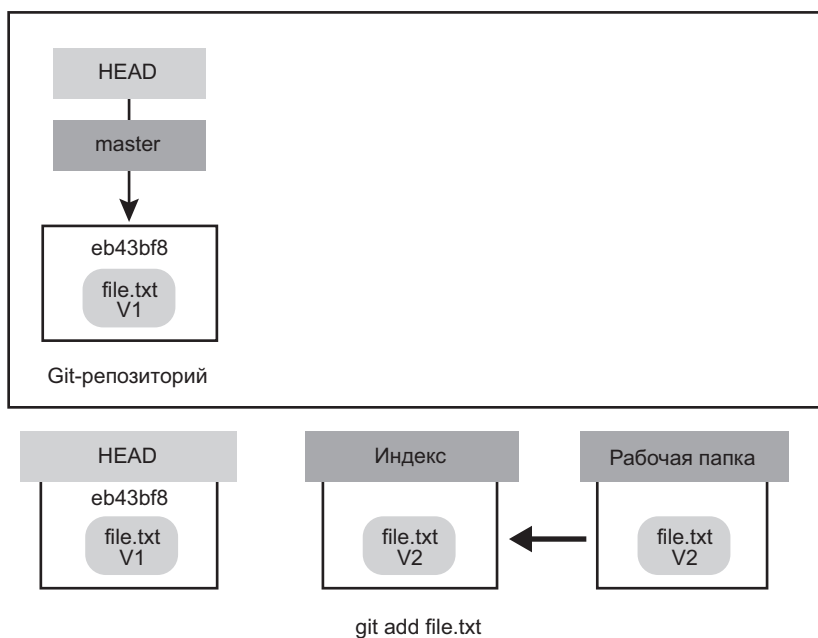
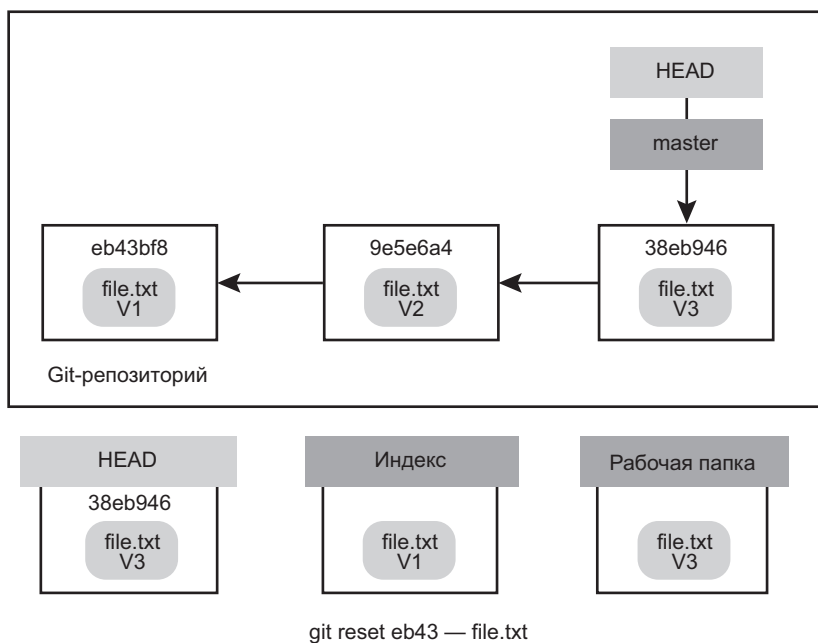


Рис. 7.13

**Рис. 7.14****Рис. 7.15**

При этом мы получаем эффект отмены индексирования файла. Если посмотреть на эту диаграмму и вспомнить, что делает команда `git add`, вы увидите, что их действия прямо противоположны друг другу (рис. 7.14).

Именно поэтому в выводимых командой `git status` данных предлагается использовать для отмены индексирования указанную форму команды.

Так же легко мы можем заставить Git брать данные не из того места, куда нацелен указатель `HEAD`. Достаточно в явном виде указать коммит, из которого следует взять версию файла. Команда в этом случае примет вид `git reset eb43bf file.txt` (рис. 7.15).

Аналогичный эффект мы получили бы, вернув рабочую папку в состояние, когда там находился файл `v1`, запустив для этого файла команду `git add`, а затем снова вернувшись к версии `v3` (не выполняя на самом деле всех этих действий). Если сейчас воспользоваться командой `git commit`, будут зафиксированы изменения, возвращающие файл к версии `v1`, хотя фактически этот файл больше ни разу не попадал в рабочую папку.

Следует также заметить, что аналогично команде `git add` команда `reset` допускает параметр `--patch`, реализующий снятие индексации с части содержимого. В результате вы можете избирательно отменять результаты индексирования или откатывать изменения.

Объединение коммитов

А теперь с учетом вышеизложенного сделаем кое-что интересное — объединим несколько коммитов друг с другом.

Предположим, у вас есть набор коммитов с сообщениями «ой», «ведется работа» и «забыл про этот файл». Команда `reset` позволяет легко и просто объединить их друг с другом.

Допустим, что в рамках вашего проекта первый коммит содержит один файл, во втором коммите добавляется новый файл и редактируется уже имеющийся, но при этом в третьем коммите в этот исходный файл снова вносятся изменения (рис. 7.16). Поскольку второй коммит появился в процессе работы, хотелось бы объединить его с третьим.

Команда `git reset --soft HEAD~2` возвращает ветку с указателем `HEAD` на более старый коммит — первый из тех, которые мы хотим сохранить (рис. 7.17).

После этого достаточно будет команды `git commit` (рис. 7.18).

Как видите, теперь доступная нам история, которую мы подготовили к отправке на сервер, включает в себя один коммит с файлом `file-a.txt` версии `v1` и второй, содержащий одновременно отредактированный файл `file-a.txt` версии `v3` и добавленный файл `file-b.txt`. Коммита с версией `v2` этого файла в истории больше нет.

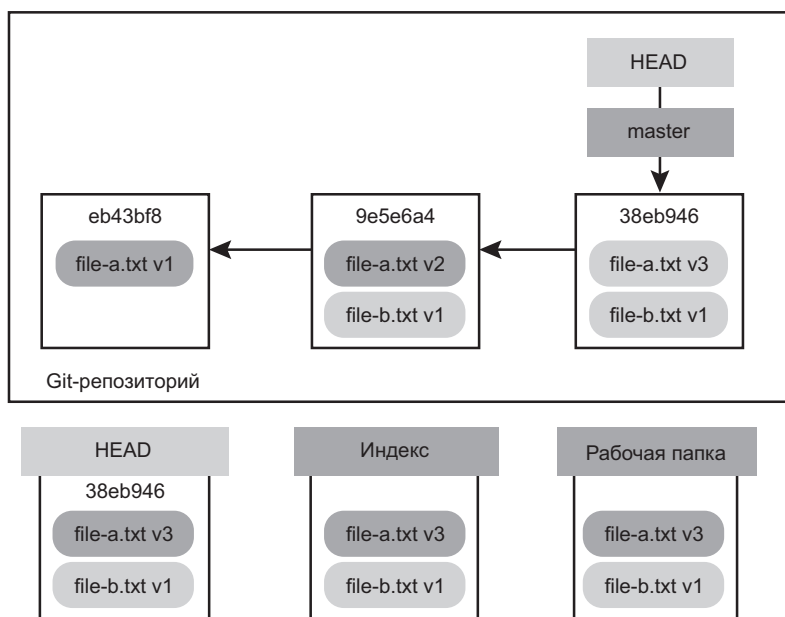
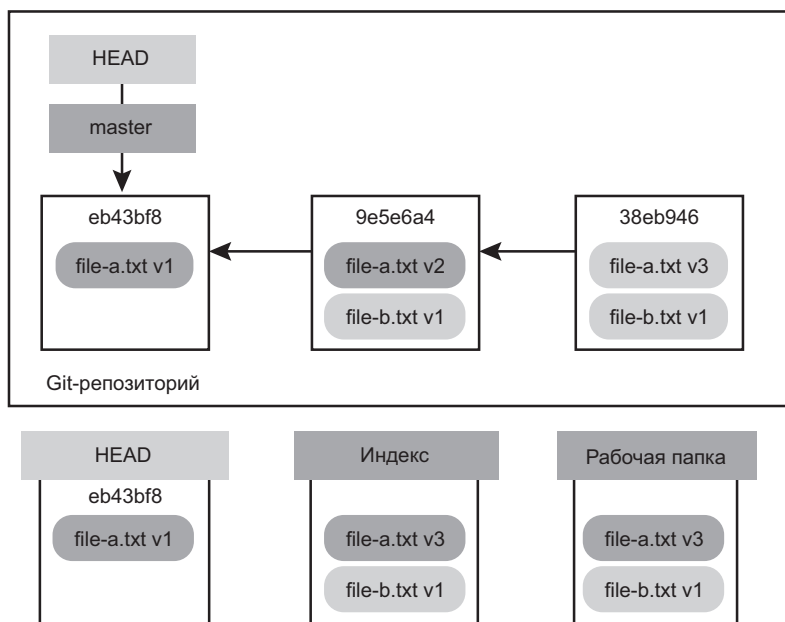


Рис. 7.16



git reset — soft HEAD~2

Рис. 7.17

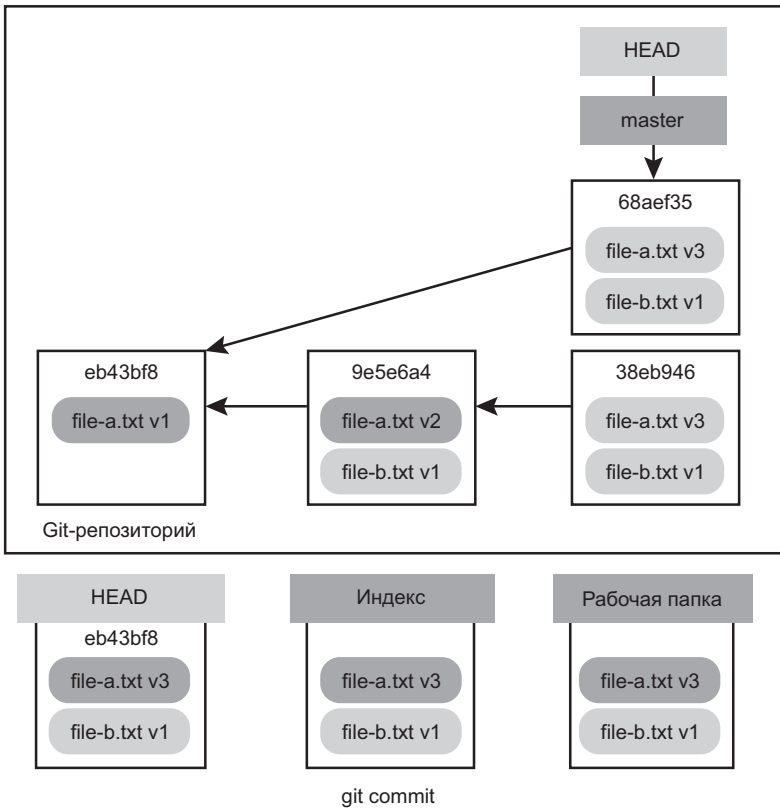


Рис. 7.18

Сравнение с командой checkout

Возможно, у вас возник вопрос, в чем же состоит разница между командами `checkout` и `reset`, ведь команда `checkout` тоже манипулирует тремя деревьями и меняет свое поведение после того, как ей указывают путь к файлу.

Без указания пути

Команда `git checkout [ветка]` напоминает команду `git reset --hard [ветка]`, но есть и два существенных отличия.

Во-первых, в отличие от команды `reset --hard`, команда `checkout` не ставит под угрозу содержимое рабочей папки; она проверяет, чтобы файлы, содержащие какие-либо изменения, остались незатронутыми. Более того, она пытается выполнить тривиальное слияние в рабочей папке, в итоге обновляются все файлы, которые не

претерпели изменений. Команда же **reset --hard** просто замещает все содержимое, не выполняя никаких проверок.

Вторым важным отличием является способ обновления указателя **HEAD**. Если команда **reset** смещает ветку, на которую нацелен указатель **HEAD**, команда **checkout** смещает сам указатель, и он начинает ссылаться на другую ветку.

Предположим, у нас есть ветки **master** и **develop**, указывающие на разные коммиты. Указатель **HEAD** нацелен на вторую из них. После команды **git reset master** ветка **develop** станет ссылаться на тот же самый коммит, что и ветка **master**. Если же вместо этого воспользоваться командой **git checkout master**, сдвинется уже не ветка **develop**, а указатель **HEAD**, который начнет ссылаться на ветку **master**.

То есть в обоих случаях мы перемещаем указатель **HEAD** на коммит *A*, но делаем это по-разному. Команда **reset** двигает ветку, на которую нацелен указатель **HEAD**, в то время как команда **checkout** смещает сам указатель (рис. 7.19).

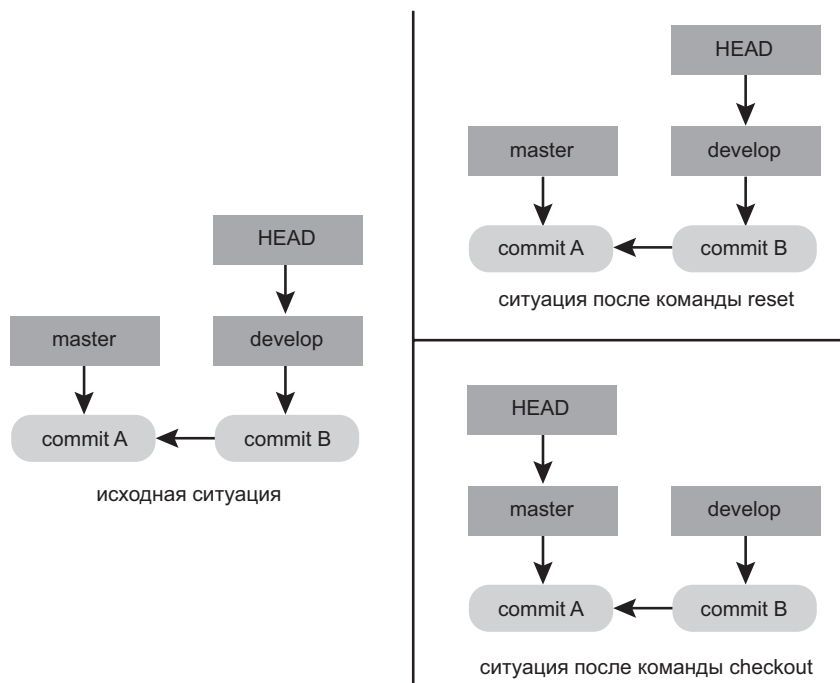


Рис. 7.19

С указанием пути

Вторым вариантом команды **checkout** является вариант с указанием пути. При этом, как и в случае команды **reset**, указатель **HEAD** остается на своем месте. Аналогично команде **git reset [ветка]** файл наша команда обновляет содержимое области

индексирования файлом из коммита, заодно переписывая и файл в рабочей папке. Такое же действие произвела бы команда `git reset --hard [ветка] файл` (если бы команда `reset` допускала подобную модификацию) — потенциально опасное для содержимого рабочей папки и не затрагивающее указателя `HEAD`.

Подобно командам `git reset` и `git add`, команда `checkout` принимает параметр `--patch`, давая вам возможность отменять изменения в файле по частям.

Заключение

Надеемся, вы разобрались с командой `reset` и сможете корректно ею пользоваться. Возможно, пока вы немного путаетесь с отличиями этой команды от команды `checkout` и не помните все существующие варианты вызова.

Для таких случаев мы приводим сводную таблицу с информацией о том, как каждая из команд воздействует на все три дерева. Пометка `REF` в столбце `HEAD` означает, что команда перемещает ссылку (ветку), на которую нацелен указатель `HEAD`, в то время как пометка `HEAD` означает смещение самого указателя. Особое внимание обращайте на содержимое столбца «Сохранность рабочей папки» — если там написано `НЕТ`, дважды подумайте перед тем как выполнить такую команду.

	HEAD	Индекс	Рабочая папка	Сохранность рабочей папки
На уровне коммитов				
<code>reset --soft [коммит]</code>	REF	НЕТ	НЕТ	ДА
<code>reset [коммит]</code>	REF	ДА	НЕТ	ДА
<code>reset --hard [коммит]</code>	REF	ДА	ДА	НЕТ
<code>checkout [коммит]</code>	HEAD	ДА	ДА	ДА
На уровне файлов				
<code>reset (коммит) [файл]</code>	НЕТ	ДА	НЕТ	ДА
<code>checkout (коммит) [файл]</code>	НЕТ	ДА	ДА	НЕТ

Более сложные варианты слияния

Обычно операция слияния в Git происходит достаточно просто. Дело в том, что в Git легко выполняются многократные слияния с одной и той же веткой, что дает возможность поддерживать актуальное состояние долгоживущей ветки, разрешая

по мере появления небольшие конфликты и не доводя дело до одного глобального конфликта при попытке слияния конечного результата работы.

Тем не менее сложные конфликты периодически возникают даже при таком подходе. В отличие от других систем контроля версий, Git не пытается проявлять чрезмерный интеллект в деле разрешения конфликтов слияния. Эта система направляет ресурсы на выявление ситуаций, в которых слияние производится однозначным образом; если же возникает конфликт, она не пытается автоматически его разрешить. Соответственно, слишком долго откладывая слияние быстро расходящихся веток, вы рискуете попасть в сложную ситуацию.

В этом разделе мы рассмотрим некоторые из возникающих проблем и изучим инструменты, которые Git предоставляет как раз для подобных нетривиальных случаев. Кроме того, мы поговорим о ряде других нестандартных типов слияния, а также о том, как вернуть в исходное состояние уже слитые друг с другом ветки.

Конфликты слияния

Ранее мы уже вскользь касались процедуры разрешения конфликтов слияния, но для более сложных ситуаций Git предоставляет инструментарий, позволяющий понять, что именно происходит, и выбрать оптимальный подход к решению проблемы.

Перед потенциально конфликтным слиянием первым делом следует по возможности очистить рабочую папку. Незаконченные наработки желательно либо зафиксировать во временной ветке, либо скрыть. Это позволит вам отменить любые сделанные в этой папке изменения. Если на момент слияния в рабочей папке присутствуют несохраненные данные, есть риск потерять их.

Рассмотрим пример. Пусть у нас есть простейший файл, написанный на языке Ruby и выводящий на экран строку **hello world**:

```
#!/usr/bin/env ruby

def hello
  puts 'hello world'
end

hello()
```

Создадим в репозитории ветку **whitespace** и заменим все окончания строк в стиле Unix окончаниями строк в стиле DOS. По сути, изменения будут внесены в каждую строку файла, но затронут только пробел. После этого мы заменим строку **hello world** строкой **hello mundo**:

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'

$ unix2dos hello.rb
```

```
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
1 file changed, 7 insertions(+), 7 deletions(-)
```

```
$ vim hello.rb
$ git diff -w
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
  #! /usr/bin/env ruby
```

```
def hello
-   puts 'hello world'
+   puts 'hello mundo'^M
end
```

```
hello()
```

```
$ git commit -am 'hello mundo change'
[whitespace 6d338d2] hello mundo change
1 file changed, 1 insertion(+), 1 deletion(-)
```

Вернемся в ветку **master** и снабдим нашу функцию некой документацией:

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

```
$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
  #! /usr/bin/env ruby
  +# prints out a greeting
  def hello
    puts 'hello world'
  end
```

```
$ git commit -am 'document the function'
[master bec6336] document the function
1 file changed, 1 insertion(+)
```

При попытке выполнить слияние с веткой **whitespace** мы получим конфликт из-за изменения пробелов.

```
$ 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

Теперь у нас есть несколько вариантов действия. Во-первых, рассмотрим пути выхода из этой ситуации. Если вы не ожидали конфликта слияния и не хотите заниматься его разрешением, можно просто отменить результат слияния командой `git merge --abort`:

```
$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

Команда `git merge --abort` пытается вернуть все в состояние, которое имело место до попытки слияния. Но если перед слиянием в вашей рабочей папке находились незафиксированные изменения, возвращение в исходное состояние может оказаться невозможным. Во всех же остальных случаях оно прекрасно работает.

Если в какой-то момент вы решили, что текущее состояние вас совсем не устраивает и имеет смысл начать все заново, можно воспользоваться командой `git reset --hard HEAD`, хотя при этом можно указать и другую точку, в которую вы хотели бы вернуться. Но еще раз напомним, что это не оставит камня на камне от содержимого вашей рабочей папки, поэтому удостоверьтесь, что содержащиеся в ней наработки вам действительно не требуются.

Игнорирование пробелов

В рассматриваемой ситуации конфликты связаны с символами пробела. В данном случае это нам заранее известно, но и в реальных ситуациях понять причину конфликта не составляет труда, так как каждая строка, удаляемая с одной стороны, добавляется с другой. По умолчанию система Git рассматривает все эти строки как измененные, и это мешает ей выполнить слияние.

К счастью, используемая по умолчанию стратегия слияния допускает наличие дополнительных аргументов, а среди них есть и связанные с корректным игнорированием изменений в символах пробела. Если выясняется, что изрядная часть конфликтов слияния вызвана этими символами, просто отмените операцию и проведите ее заново, но на этот раз с параметром `-Xignore-all-space` или `-Xignore-space-change`. Первый параметр игнорирует изменения в любом количестве существующих символов пробела, второй игнорирует все связанные с ними изменения в совокупности:

```
$ git merge -Xignore-all-space whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
```

```
hello.rb | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

Так как в рассматриваемом примере реальные изменения, внесенные в файл, не конфликтуют между собой, игнорирование измененных пробелов позволит без проблем выполнить слияние.

Это палочка-выручалочка для тех рабочих групп, в которых есть любитель периодически переходить от пробелов к символам табуляции или наоборот.

Слияние файлов вручную

С предварительной обработкой пробелов Git справляется довольно успешно, но существуют и другие типы изменений, которые система уже не может выполнить автоматически, и для них вам потребуются отдельные сценарии. Для примера предположим, что Git не умеет обрабатывать изменения пробелов и это нужно сделать вручную.

В этом случае нам нужно пропустить предназначенный для слияния файл через программу `dos2unix`. Как мы будем это делать?

Первым делом мы спровоцируем конфликт слияния. Затем нужно получить копии нашей версии файла, версии файла из ветки, в которую сливаются данные, и общей версии, от которой произошли обе наши ветки. После этого мы внесем исправления в нашу или вторую версию и заново выполним слияние только для одного файла.

Получить три версии файла очень легко. Система Git хранит их все в области индексирования в разных «состояниях», каждое из которых имеет связанный с ним номер. Состояние 1 — это общий предок, состояние 2 — ваша версия, а состояние 3 берется из `MERGE_HEAD` и представляет собой версию, возникающую в процессе слияния (общая версия).

Копию каждой из этих версий конфликтующего файла позволяет извлечь команда `git show` и особый синтаксис:

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

Для получения более конкретной информации можно прибегнуть к служебной команде `ls-files -u` и получить контрольные суммы SHA массивов двоичных Git-данных 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` представляет собой всего лишь сокращение для поиска хеша SHA этого массива двоичных данных.

Итак, в нашей рабочей папке теперь находится содержимое, связанное с каждым из трех состояний, и можно вручную решить проблему с символами пробела, а затем повторить процедуру слияния, прибегнув к малоизвестной команде `git merge-file`, предназначенной именно для таких случаев:

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...

$ git merge-file -p \
    hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -w
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
    #! /usr/bin/env ruby
    # prints out a greeting
    def hello
-     puts 'hello world'
+     puts 'hello mundo'
    end
    hello()
```

Итак, мы корректно выполнили слияние файла. По большому счету, этот вариант решения проблемы лучше применения параметра `ignore-all-space`, так как мы вносим исправления в измененные пробелы, а не просто игнорируем их. При слиянии с применением параметра `ignore-all-space` у нас окажется несколько строк с окончаниями в стиле DOS, то есть в одном файле будут использоваться разные стили.

Если перед окончательной фиксацией изменений вы хотите посмотреть, что именно изменилось с одной или с другой стороны, можно с помощью команды `git diff` в качестве результата слияния сравнить подготовленное к фиксации содержимое вашей рабочей папки с любым из трех состояний. Проведем процедуру сравнения для каждого из них.

Чтобы сравнить результат с тем, что было в этой ветке до слияния, другими словами, увидеть, к чему привело слияние, используем команду `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()
```

Здесь можно легко увидеть, что в нашей ветке в результате слияния изменилась всего одна строка.

Чтобы посмотреть, чем результат слияния отличается от содержимого ветки, в которую сливались данные, используем команду **git diff --theirs**. В этом и следующем примерах мы будем добавлять параметр **-w**, чтобы не учитывать проблемы с пробелами, ведь сравнение выполняется с версией Git, а не с нашим исправленным файлом **hello.theirs.rb**:

```
$ git diff --theirs -w
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
  #! /usr/bin/env ruby

  # prints out a greeting
  def hello
    puts 'hello mundo'
  end
```

Наконец, узнать, как изменился файл относительно сразу двух веток, позволяет команда **git diff --base**:

```
$ git diff --base -w
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
  #! /usr/bin/env ruby

  # prints out a greeting
  def hello
-   puts 'hello world'
+   puts 'hello mundo'
  end
  hello()
```

Теперь можно воспользоваться командой **git clean** и удалить дополнительные файлы, которые требовались в процессе ручного слияния файлов, — больше они нам не нужны:

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

Применение команды checkout

Бывают ситуации, когда по каким-то причинам мы не хотим выполнять слияние прямо сейчас или же эта процедура пока не работает и для ее проведения требуется дополнительная информация.

Внесем в наш пример небольшие изменения. Предположим, у нас есть пара долгоживущих веток, в каждой из которых присутствует несколько коммитов, но при этом попытки их слияния не удаются по причине конфликта:

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) update README
* 9af9d3b add a README
* 694971d update phrase to hola world
| * e3eb223 (mundo) add more tests
| * 7cff591 add testing script
| * c3ffff1 changed text to hello mundo
|/
* b7dcc89 initial hello world code
```

У нас есть три уникальных коммита в ветке **master** и еще три коммита в ветке **mundo**. При попытке слить содержимое ветки **mundo** мы 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.
```

Хотелось бы понять, что именно является причиной проблемы. Открыв файл, мы увидим примерно такую картину:

```
#!/usr/bin/env ruby

def hello
  <<<<<< HEAD
    puts 'hola world'
  =====
    puts 'hello mundo'
  >>>>>> mundo
end

hello()
```

На обеих ветках в этот файл добавлялось некое содержимое, но некоторые коммиты модифицировали одни и те же строки, что и стало причиной конфликта.

Есть пара инструментов, позволяющих определить, что именно мешает успешному слиянию. Возможно, что вы пока не понимаете, каким образом можно разрешить конфликт, и вам нужна дополнительная информация.

На помощь придет команда **git checkout** с параметром **--conflict**. Она повторно выгружает содержимое файла и заменяет маркеры конфликта слияния. Это может пригодиться в ситуации, когда вы хотите сбросить маркеры и снова попробовать разрешить конфликт.

Параметру `--conflict` можно передать значение `diff3` или `merge` (последнее используется по умолчанию). Значение `diff3` заставляет Git использовать слегка измененную версию маркеров конфликта — помимо «нашей» и «их» версий файлов отображается также «базовая» версия, что дает дополнительную информацию:

```
$ git checkout --conflict=diff3 hello.rb
```

После этой команды наш файл примет следующий вид:

```
#!/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, можете сделать его форматом, который по умолчанию будет использоваться для будущих конфликтов слияния. Для этого параметру `merge.conflictstyle` нужно присвоить значение `diff3`:

```
$ git config --global merge.conflictstyle diff3
```

Команде `git checkout` также можно добавить параметры `--ours` и `--theirs`, которые позволяют быстро выбрать одну из версий файла, не выполняя слияния.

Это особенно полезно в случае конфликта бинарных файлов, при которых можно просто выбрать одну из сторон, или в ситуации, когда необходимо слить из другой ветки только определенные файлы, — вы можете выполнить слияние и перед фиксацией состояния перейти к файлам на одной или на другой стороне.

Протоколирование слияния

Пригодится нам при разрешении конфликтов слияния и команда `git log`. Она помогает получить информацию о возможных причинах конфликта. Иногда просмотр истории позволяет вспомнить, почему в двух ветках разработки редактировалась одна и та же область кода.

Получить полный список всех уникальных коммитов, сделанных в любой из участвующих в слиянии веток, помогает уже знакомая вам нотация трех точек:

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 update README
< 9af9d3b add a README
< 694971d update phrase to hola world
```

```
> e3eb223 add more tests
> 7cfff591 add testing script
> c3fffff1 changed text to hello mundo
```

В итоге мы получили замечательный список из шести участвующих в слиянии коммитов с информацией о том, какой строки разработки касается каждый из них.

Эти данные можно упростить, дав команду предоставить нам более конкретные сведения. Добавив к команде `git log` параметр `--merge`, мы получим список только тех коммитов с каждой стороны слияния, которые касаются файла, являющегося в данный момент причиной конфликта:

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3fffff1 changed text to hello mundo
```

Добавив вместо этого к команде параметр `-p`, мы получим список изменений, внесенных в файл, в котором возник конфликт. Все это позволяет вам быстро получить все сведения, необходимые для понимания причины конфликта и поиска оптимального способа его разрешения.

Комбинированный формат

Так как Git индексирует все успешные результаты слияния, команда `git diff`, запущенная во время конфликта слияния, дает возможность вывести на экран только конфликтующие фрагменты. Это позволяет посмотреть, какие еще конфликты требуется разрешить.

Запущенная сразу после конфликта слияния команда `git diff` предоставляет информацию в необычном 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()
```

Это так называемый комбинированный формат, в котором рядом с каждой строкой выводятся два столбца данных. В первом столбце показывается, есть ли в строке

отличия (добавленная или удаленная информация) между «нашей» веткой и файлом из рабочей папки, а вторая строка отображает данные об отличиях между «их» веткой и копией вашей рабочей папки.

В итоге в приведенном примере мы видим строки <<<<<< и >>>>>> в рабочей копии, при том что они отсутствуют в обеих сливаемых версиях. Это имеет смысл, ведь инструмент слияния вставляет их, предоставляя нам дополнительную информацию, а впоследствии мы должны их удалить.

Если мы разрешим конфликт и снова прибегнем к команде `git diff`, эта информация будет предоставлена нам в более удобном виде:

```
$ 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()
```

Здесь показано, что строка «hola world», присутствующая в «нашей» ветке, отсутствовала в рабочей копии, в то время как строка «hello mundo» была в «их» ветке, но не в рабочей копии, а вот строка «hola mundo» не наблюдалась ни в одной из сливаемых веток, зато есть в рабочей копии. Подобную информацию полезно получить перед фиксацией результатов разрешения конфликта.

Эту информацию для любого выполненного слияния можно также получить с помощью команды `git log`. Она позывает, каким образом был разрешен конфликт. В таком формате Git выводит сведения, если к коммиту слияния применяется команда `git show` или к команде `git log -p` добавляется параметр `--cc` (по умолчанию эта команда показывает только исправления для коммитов, не входящих в слияние):

```
$ 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 результатов слияния

Теперь, когда вы умеете создавать коммиты слияния, вы можете попасть в ситуацию, когда какой-то из коммитов будет сделан по ошибке. Но одной из замечательных особенностей системы Git является возможность делать ошибки, потому что их можно исправить (и зачастую это делается очень легко).

Не стали исключением и коммиты слияния. Предположим, вы начали работу в тематической ветке и случайно слили ее в ветку **master**, после чего история коммитов стала выглядеть так, как показано на рис. 7.20.

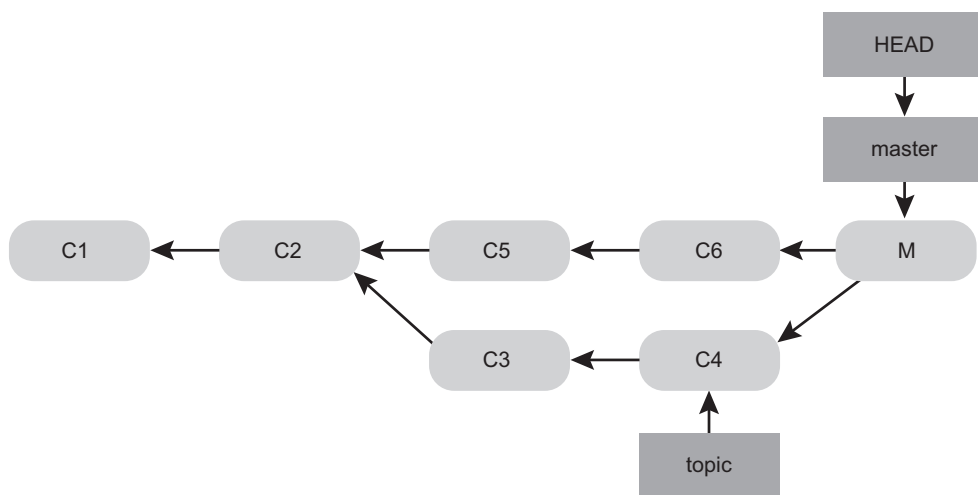


Рис. 7.20. Неправильный коммит слияния

В этой ситуации у вас есть два варианта действий — в зависимости от того, какой результат вы хотите получить.

Исправление ссылок

Если нежелательный коммит слияния присутствует только в вашем локальном репозитории, проще и лучше всего переместить ветки, чтобы они указывали туда, куда нужно. В большинстве случаев выполнив после ошибочной команды `git merge` команду `git reset --hard HEAD~`, вы установите указатели веток так, как показано на рис. 7.21.

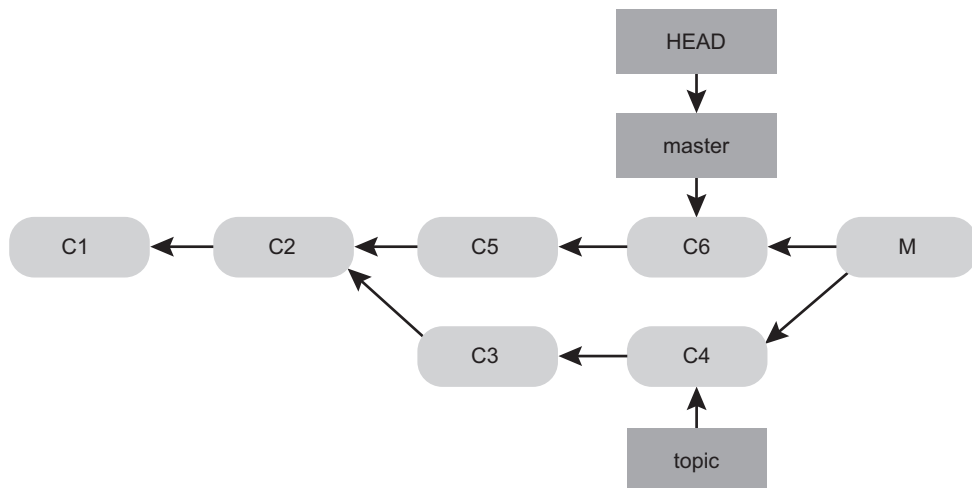


Рис. 7.21. Вид истории после выполнения команды `git reset --hard HEAD~`

С особенностями команды `reset` вы уже знакомы, поэтому понять, что произошло, не сложно. На всякий случай напомним, что команда `reset --hard`, как правило, выполняется в три этапа:

1. Перемещается ветка, на которую нацелен указатель `HEAD`. В рассматриваемом случае мы хотим вернуть ветку `master` в положение, в котором она была до фиксации слияния (C6).
2. Область индексирования приводится к такому же виду, как в ветке, отмеченной указателем `HEAD`.
3. Рабочая папка приводится к такому же виду, как и область индексирования.

Недостатком этого метода является внесение изменений в историю, что может привести к проблемам при наличии репозитория общего доступа. Если другие пользователи работают с коммитами, которые вы собираетесь переписать, от использования команды `reset` лучше отказаться. Кроме того, данный подход не будет работать, если с момента слияния был создан хотя бы один новый коммит, — перемещение ссылок, по сути, приведет к потере внесенных изменений.

Отмена коммита

Если перемещение указателей ветки в вашей ситуации не помогает, система Git дает возможность выполнить новый коммит, отменяющий все изменения, внесенные предыдущим коммитом. Эта операция называется восстановлением (*revert*) и в рассматриваемой ситуации она выполняется следующим образом:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

Флаг `-m 1` указывает, какой из предков является «основной веткой» и подлежит сохранению. После слияния в ветку, на которую нацелен указатель `HEAD` (`git merge topic`), у нового коммита окажутся два предка: первый с указателем `HEAD` (`C6`) и второй — вершина сливаемой ветки (`C4`). Мы хотим отменить все изменения, внесенные слиянием родителя 2 (`C4`), сохранив все содержимое родителя 1 (`C6`).

После отмены коммита история будет выглядеть так, как показано на рис. 7.22.

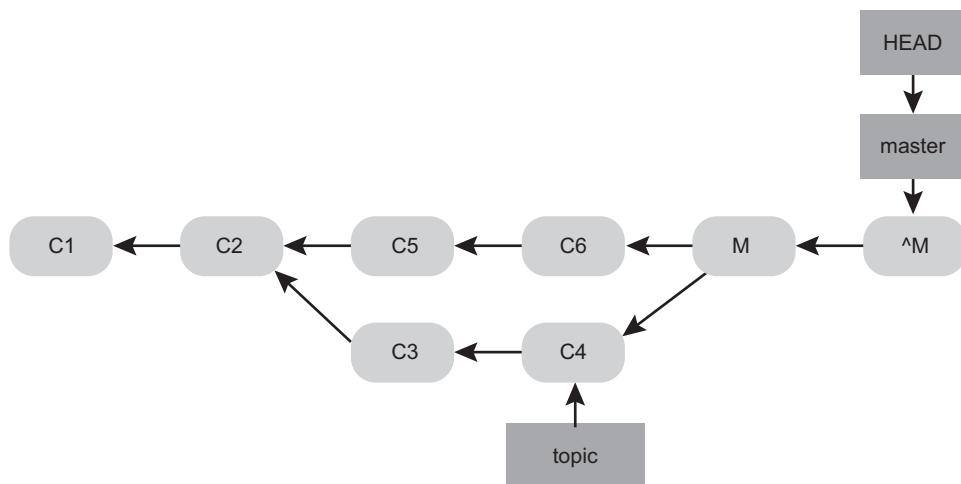


Рис. 7.22. История после выполнения команды `git revert -m 1`

Содержимое нового коммита `^M` полностью совпадает с содержимым коммита `C6`, то есть вы можете начать работу, как будто слияния никогда не было, только коммиты, не входящие в слияние, все еще присутствуют в истории перемещений указателя `HEAD`. В системе Git возникнет путаница, если сейчас вы попытаетесь снова слить содержимое ветки `topic` в ветку `master`:

```
$ git merge topic
Already up-to-date.
```

В ветке `topic` сейчас нет ничего, что было бы недоступно из ветки `master`. Что еще хуже, если выполнить какую-то работу в ветке `topic` и снова произвести слияние,

Git добавит только те изменения, которые были сделаны после отмены слияния (рис. 7.23).

Лучше всего решить эту проблему отменой возврата исходного слияния, то есть нужно добавить изменения, которые были отменены, и создать новый коммит слияния (рис. 7.24):

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'""
$ git merge topic
```

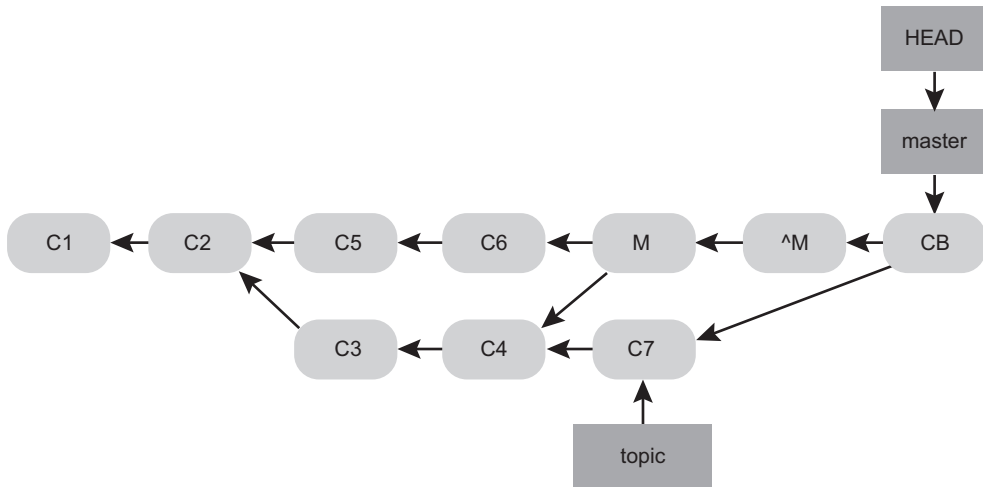


Рис. 7.23. История с некорректным слиянием

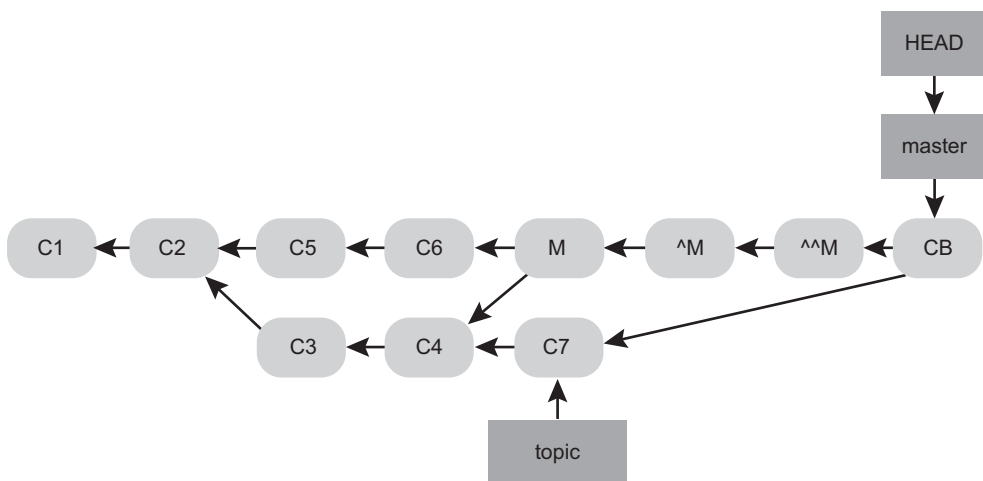


Рис. 7.24. История после повторения отмененного слияния

В приведенном примере коммиты М и ^М отменены. В коммит ^М фактически сливаются изменения из коммитов С3 и С4, а в коммит С8 — изменения из коммита С7. Таким образом, слияние ветки `topic` полностью завершается.

Другие типы слияния

До сих пор мы рассматривали обычное слияние двух веток, реализуемое на основе так называемой «рекурсивной» стратегии слияния. Но существуют и другие типы данной операции, и в этом разделе мы кратко рассмотрим некоторые из них.

Выбор версии

Прежде всего, существует еще один полезный прием, который может вам пригодиться в обычном «рекурсивном» режиме слияния. Вам уже знакомы параметры `ignore-all-space` и `ignore-space-change`, которые передаются с флагом `-X`, но кроме того, мы можем попросить Git при обнаружении конфликта использовать файлы с одной или с другой стороны.

По умолчанию, обнаружив конфликт слияния двух веток, Git добавляет в код маркеры конфликта, помечает файл как содержащий конфликт и дает вам возможность разрешить его. Но бывают ситуации, когда вместо решения конфликта было бы лучше, чтобы система Git просто выбрала одну из версий, проигнорировав все остальное. В этом случае к команде `merge` следует добавить параметр `-Xours` или `-Xtheirs`.

Встретив такой параметр, Git не будет добавлять маркеры конфликта. Система просто сольет все допускающие слияние изменения. А для конфликтующих изменений она целиком выберет указанный вами вариант файла, причем это может быть двоичный файл.

Возвращаясь к примеру с выводом строки `hello world`, мы видим, что слияние в нашу ветку сопровождается конфликтами:

```
$ 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.
```

А вот при наличии параметра `-Xours` или `-Xtheirs` конфликт не возникает.

```
$ 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 данном случае, вместо того чтобы добавить маркеры конфликта в файл с версией строки `hello mundo`, с одной стороны, и версией `hola world` — с другой, система просто выбирает файл со строкой `hola world`. При этом все прочие изменения, не приводящие к конфликту, сливаются успешно.

Эти параметры можно добавить также к уже знакомой нам команде `git merge-file`, используя для отдельных вариантов слияния файлов, например, команду `git merge-file --ours`.

Для ситуаций, когда вам нужен подобный результат, но при этом вы не хотите, чтобы система Git пыталась слить изменения из другой версии, существует более суровый вариант — стратегия слияния «ours» (она не имеет отношения к параметру «ours» рекурсивной стратегии слияния).

По сути, в данном случае выполняется фиктивное слияние. Записывается новый коммит слияния, родителями которого выступают обе ветки, при этом сливаемая вами ветка игнорируется, а в качестве результата слияния записывается точный код из вашей текущей ветки:

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

Можно убедиться в отсутствии отличий между веткой, в которой вы находились, и результатом слияния.

Этот трюк полезен в ситуациях, когда нужно обмануть Git, заставив систему думать, что слияние ветки выполнено, в то время как фактическая операция откладывается «на потом». Например, представим, что вы создали ветку `release` и выполнили в ней некую работу, которую в какой-то момент требуется слить обратно в ветку `master`. Но при этом в ветку `master` были внесены некие исправления, которые необходимо перенести в ветку `release`. В такой ситуации можно слить ветку с исправлениями в ветку `release`, а затем командой `merge -s ours` слить содержимое этой ветки в ветку `master` (несмотря на то, что исправления в ней уже присутствуют). Именно благодаря этому позже, когда вы будете снова сливать содержимое ветки `release`, не возникнет конфликтов из-за исправлений.

Слияние поддеревьев

Идея слияния поддеревьев состоит в том, что один из проектов проецируется во вложенную папку другого, и наоборот. При указании этой операции система Git зачастую в состоянии распознать, что один элемент является поддеревом другого, и корректно выполнить слияние.

Рассмотрим пример добавления нового проекта в уже существующий, при этом код из первого проекта записывается в подпапку второго.

Первым делом добавим к нашему проекту приложение Rack. Мы присоединим его как удаленный репозиторий и выгрузим его содержимое в отдельную ветку:

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
* [new branch] build -> rack_remote/build
* [new branch] master -> rack_remote/master
* [new branch] rack-0.4 -> rack_remote/rack-0.4
* [new branch] rack-0.9 -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

Теперь корень проекта Rack находится в ветке **rack_branch**, в то время как наш собственный проект располагается в ветке **master**. По очереди проверив обе эти ветки, вы убедитесь, что они имеют разный корневой каталог:

```
$ ls
AUTHORS      KNOWN-ISSUES  Rakefile     contrib      lib
COPYING      README        bin          example      test
$ git checkout master
Switched to branch "master"
$ ls
README
```

Такая конфигурация выглядит несколько странно. Но ветки в вашем хранилище вовсе не обязаны быть ветками одного и того же проекта. Это не распространенная ситуация, так как с прикладной точки зрения этого практически никогда не требуется, но вы легко можете получить ветки с совершенно разной историей.

В рассматриваемой ситуации мы хотим выгрузить содержимое проекта Rack в подпапку нашего основного проекта. В Git это реализуется командой **git read-tree**. Подробно она вместе с сопутствующими ей командами будет рассматриваться позже, а пока вам достаточно понимать, что она считывает корень дерева одной ветки в текущую область индексации и рабочую папку. Мы просто вернемся в ветку **master** и извлечем содержимое ветки **rack** в подпапку **rack** нашей ветки **master** основного проекта:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

При фиксации все это будет выглядеть как результат добавления во вложенную папку всех файлов из проекта Rack — как будто мы просто скопировали их из

архива. Примечательна легкость, с которой мы можем слить изменения из одной ветки в другую. Все обновления проекта Rack можно получить, перейдя на его ветку и выполнив скачивание данных из удаленного репозитория:

```
$ git checkout rack_branch
$ git pull
```

Скачанные изменения можно слить в нашу ветку `master`. Для этого можно воспользоваться командой `git merge -s subtree`; впрочем, в этом случае Git заодно сольет истории проектов, что не всегда желательно. Чтобы извлечь изменения и предварительно заполнить сообщение фиксации, пользуйтесь параметрами `--squash` и `--no-commit` вместе с параметром стратегии слияния поддереьев `-s`:

```
$ git checkout master
$ git merge --squash -s subtree --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Итак, мы слили все изменения из проекта Rack, подготовив их к локальной фиксации. Но можно поступить и наоборот: внести изменения во вложенную папку `rack` ветки `master` и позже слить их в ветку `rack_branch` для передачи лицам, отвечающим за поддержку проекта, или для отправки на сервер.

В результате мы получаем такую же рабочую схему, как и в случае с вложенными модулями, но без использования самих модулей. Ветки можно хранить в нашем репозитории с другими проектами, периодически сливая их в наш проект как поддереья. В некотором смысле это удобно: например, фиксация состояния всего кода происходит в одном месте. Однако при этом рабочий процесс несколько усложняется и повышается вероятность ошибки при повторной интеграции изменений, а также вероятность случайно отправить ветку не в тот репозиторий.

Еще необычно то, что для просмотра разницы содержимого подпапки `rack` и кода в ветке `rack_branch` (это нужно, чтобы понять, не пришло ли время выполнить слияние) обычную команду `diff` использовать уже нельзя. 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 подпапки `rack` с содержимым ветки `master` на сервере после последнего скачивания изменений:

```
$ git diff-tree -p rack_remote/master
```

Команда `rerere`

Команда `git rerere` относится к частично скрытым компонентам. Ее название представляет собой аббревиатуру фразы «reuse recorded resolution» (повторное использование записанного решения), и как следует из этого названия, она позволяет

сделать так, чтобы система Git запомнила, как вы разрешили конкретный конфликт и в следующий раз подобный конфликт можно было разрешить автоматически.

Существует ряд сценариев, в которых эта функциональность действительно полезна. Один из примеров упоминается даже в документации и касается возможности обеспечить чистоту слияния долгоживущей ветки, не создавая при этом набор промежуточных коммитов слияния. Подключив команду **rerere**, можно периодически проводить слияния, разрешать конфликты, а затем отменять результаты слияния. Выполняя эту последовательность в процессе работы, вы обеспечите чистое финальное слияние, так как команда **rerere** сможет делать все автоматически.

Аналогичная тактика применяется для сохранения мобильности ветки. Благодаря ей вам не придется все время сталкиваться с одними и теми же конфликтами перемещения. Представьте, что вам нужно переместить ветку, для которой уже было выполнено слияние с разрешением целого набора конфликтов. Вряд ли вам захочется разрешать эти конфликты по второму разу.

Полезна эта команда и при периодическом слиянии набора тематических веток в тестовую ветку. В случае неудачного тестирования можно отменить все слияния и выполнить их еще раз, исключив ветку, послужившую причиной неудачи. При этом вам не потребуется снова разрешать конфликты.

Для подключения команды **rerere** достаточно поменять конфигурационные настройки:

```
$ git config --global rerere.enabled true
```

Кроме того, можно создать папку **.git/rr-cache** в определенном репозитории, но подключение через изменение настроек выглядит понятнее, кроме того, может выполняться в глобальном масштабе.

А теперь рассмотрим простой пример. Предположим, у нас есть вот такой файл:

```
#!/usr/bin/env ruby

def hello
  puts 'hello world'
end
```

Подобный пример мы уже рассматривали, и сейчас, как и в предыдущем случае, мы изменим строку «hello» на «hola», а затем в другой ветке — «world» на «mundo».

При слиянии этих веток возникает 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.
```

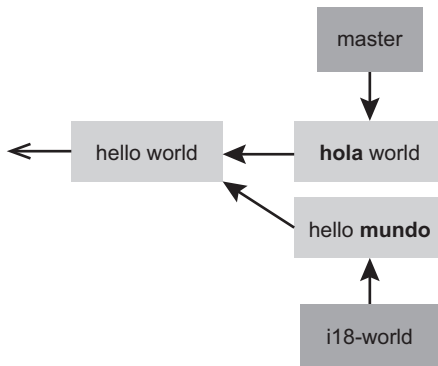


Рис. 7.25

Возможно, вы уже обратили внимание на новую строку **Recorded preimage for FILE**. В остальном выводимые командой данные такие же, как и при обычных конфликтах слияния. Здесь нам может дать подсказку команда **rerere**. Обычно в такой ситуации мы используем команду **git status**, чтобы узнать причину конфликта:

```
$ 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
#
```

Но команда **git rerere status** также может рассказать, что именно было записано в состоянии, предшествующем слиянию:

```
$ git rerere status
hello.rb
```

А команда **git rerere diff** демонстрирует текущую стадию разрешения конфликта — то, с чего было начато разрешение, и то, что в итоге получилось:

```
$ 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
```

Кроме того (это уже не связано с командой **rerere**), для просмотра конфликтующих файлов с их предшествующими конфликту версиями используется команда **ls-files -u**:

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1 hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2 hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3 hello.rb
```

Теперь конфликт можно разрешить вставкой строки **'hola mundo'**, а затем снова запустить команду **rerere diff**, чтобы посмотреть, что именно она запомнила:

```
$ 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
```

Теперь, когда Git обнаружит в файле **hello.rb** конфликт, в котором, с одной стороны, фигурирует строка **«hello mundo»**, а с другой — строка **«hola world»**, система разрешит его вставкой строки **«hola mundo»**.

То есть мы можем пометить конфликт как разрешенный и выполнить фиксацию состояния:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

Обратите внимание, что теперь в выводимых командой данных появилась строка **Recorded resolution for FILE** (сохранено разрешение конфликта для ФАЙЛА).

Теперь давайте отменим это слияние и вместо него переместим данные на вершину ветки **master**. Сместим ветку назад командой **reset**:

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

Слияние отменено. Теперь выполним перемещение тематической ветки:

```
$ 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
```

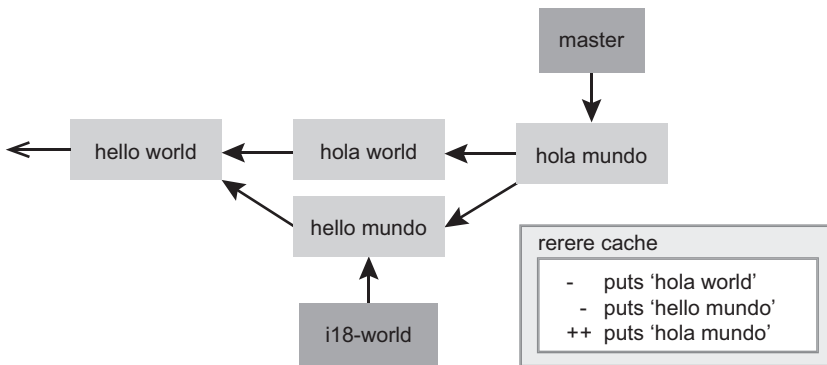


Рис. 7.26

Мы получили ожидаемый конфликт слияния, но обратите внимание на строку **Resolved FILE using previous resolution**. Если посмотреть на содержимое файла, вы увидите, что конфликт уже разрешен и маркеры конфликта слияния в файле отсутствуют:

```
$ cat hello.rb
#!/usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

Детали автоматического повторного разрешения конфликта вам покажет команда **git diff**:

```
$ 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
rerere3
```

Команда `checkout` позволяет вернуть файл в состояние конфликта:

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#!/usr/bin/env ruby
```

```
def hello
<<<<<<< ours
    puts 'hola world'
=====
    puts 'hello mundo'
>>>>>>> theirs
end
```

Давайте повторно разрешим конфликт средствами команды `rerere`:

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#!/usr/bin/env ruby
```

```
def hello
    puts 'hola mundo'
end
```

Итак, мы автоматически разрешили конфликт, воспользовавшись вариантом, который команда `rerere` сохранила для нас в кэше. Теперь можно проиндексировать изменения и продолжить перемещение ветки:

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

Если вам часто приходится выполнять повторные слияния, либо вы хотите поддерживать актуальное состояние тематической ветки без ее многочисленных слияний с веткой `master`, либо вы часто прибегаете к перемещениям, задействуйте команду `rerere`, чтобы немного упростить себе жизнь.

Отладка с помощью Git

Есть в Git и инструменты, способные помочь в отладке ваших проектов. Так как система Git способна работать с проектами практически всех типов, эти инструменты являются в изрядной степени обобщенными, но зачастую они позволяют выявить ошибку и ее причину.

Примечания к файлам

Если вы обнаружили в коде ошибку и хотите узнать, когда и почему она появилась, лучше всего посмотреть примечания к файлам. Именно там вы найдете информацию о том, какие коммиты в последний раз вносили изменения в каждую строку произвольного файла. Словом, обнаружив в своем коде ошибочный метод, откройте командой `git blame` примечания к файлу, указывающие, когда и кем в последний раз редактировалась каждая строка метода. В данном примере мы используем параметр `-L`, чтобы ограничить выводимые данные строками с 12-й по 22-ю:

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree = 'master')
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13)   command("git show #{tree}")
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree = 'master')
79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame #{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end
```

Первое поле показывает частичную контрольную сумму SHA-1 коммита, в котором в последний раз редактировалась строка. Следующие два поля содержат извлеченную из коммита информацию — имя автора и дату создания. То есть вы сразу видите, кто и когда редактировал данную строку. После этого идут номера строк и содержимое файла. Обозначение `^4832fe2` указывает на строки, которые входили в самый первый коммит — сделанный после добавления файла в проект — и остались с той поры неизменными. Здесь легко запутаться, так как теперь мы знаем по меньшей мере три способа, которыми символ `^` меняет контрольную сумму SHA коммита, но в данном случае используется такое обозначение.

Еще одной примечательной особенностью системы Git является то, что она не отслеживает переименования файлов в явном виде. Она записывает снимок состояния и пытается косвенным образом постфактум определить, что именно подверглось переименованию. Именно благодаря этой конструктивной особенности можно заставить систему выявить все виды перемещения кода. Если добавить команде `git blame` параметр `-C`, Git проанализирует файл, для которого выводятся примечания, и попытается понять, откуда взялись скопированные в него фрагменты кода. К примеру, представим, что вы производите переработку файла `GITServerHandler.m` в несколько файлов, один из которых называется `GITPackUpload.m`. Вызвав для последнего команду `blame` с параметром `-C`, вы определите происхождение кода:

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
```

```

70befddd GITServerHandler.m (Scott 2009-03-22 144)           //NSLog(@"GATHER COMM
ad11ac80 GITPackUpload.m (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 146)             NSString *parentSha;
ad11ac80 GITPackUpload.m (Scott 2009-03-24 147)             GITCommit *commit = [g
ad11ac80 GITPackUpload.m (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 149)             //NSLog(@"GATHER COMM
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 setObject
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)

```

Это очень удобно. Ведь обычно в качестве исходного фигурирует коммит, в который вы скопировали код, так как это первый случай изменения строк в файле. А Git показывает вам, из какого коммита были скопированы эти строки, даже если они находились в другом файле.

Двоичный поиск

Примечания к файлу помогают, когда вы знаете, в каком месте следует искать источник проблемы. Если же вы понятия не имеете, куда именно вкралась ошибка, а с момента последнего рабочего состояния кода образовались десятки или даже сотни коммитов, лучше прибегнуть к команде **git bisect**. Она выполняет двоичный поиск в истории коммитов, с максимально возможной скоростью определяя проблемный коммит.

Предположим, вы только что запустили в эксплуатацию новую версию своего кода, периодически получаете отчеты об ошибках, которые в среде разработки не возникали, и не можете взять в толк, почему код ведет себя подобным образом. Вы возвращаетесь к своему коду и вам удается воспроизвести проблему, но понять ее причины вы все равно не в состоянии. Здесь вам может помочь команда **bisect**. Первым делом мы запускаем процесс командой **git bisect start**, после чего выполняем команду **git bisect bad**, сообщая системе о неполадках в текущем коммите. Теперь остается запустить двоичный поиск последнего известного рабочего состояния. Это делается командой **git bisect good [исправный_коммит]**:

```

$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo

```

Система Git посчитала, что между коммитом, который вы поместили как последний корректный (v1.0), и текущей неработающей версией было сделано примерно 12 коммитов, и выгрузила вам центральный коммит этой последовательности. Теперь можно провести тестирование и посмотреть, проявляется ли проблема.

Положительный результат означает, что неполадки возникли до центрального коммита; в случае же отрицательного результата проблему следует искать после этого коммита. Предположим, наше тестирование прошло успешно, и мы информируем об этом Git командой `git bisect good`:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Теперь вам предлагается следующий коммит, расположенный посередине между протестированным и неработающим коммитами. Предположим, в этом случае тестирование покажет наличие ошибки. Тогда мы воспользуемся командой `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

Этот коммит корректен, и теперь у системы Git есть вся необходимая информация, чтобы определить, где именно впервые возникла проблема. Она сообщает нам контрольную сумму SHA-1 первого коммита с ошибкой, некоторые сведения о нем, а также перечень файлов, которые претерпели изменения в этом коммите. Все это дает возможность понять, как появилась ошибка:

```
$ 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
f24d3c6ebcf639b1a3814550e62d60b8e68a8e4 M config
```

После завершения поиска следует воспользоваться командой `git bisect reset`, чтобы вернуть указатель `HEAD` туда, где он был до начала процедуры:

```
$ git bisect reset
```

Это мощный инструмент, помогающий за считанные минуты проверить сотни коммитов на наличие ошибки. При этом, если у вас есть сценарий, возвращающий значение 0 в случае корректного проекта и отличное от нуля значение в случае обнаружения ошибок, работу команды `git bisect` можно полностью автоматизировать. Для начала вы задаете область поиска, указав работающий и неработающий коммиты. Их можно перечислить в команде `bisect start`, первым указав коммит с ошибкой, а вторым — коммит, который исправно работает:

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

В этом случае выгрузка каждого коммита будет автоматически запускать сценарий `test-error.sh`, пока Git не обнаружит дефектный коммит. Также можно воспользоваться командой `make` или `make tests`, если именно она инициирует автоматизированное тестирование.

Подмодули

Нередкой является ситуация, когда внутри одного проекта нужно задействовать другой проект. Это может быть библиотека сторонних разработчиков или ваша собственная библиотека, отдельно написанная и используемая в разных проектах. Распространенный вопрос, возникающий при таком сценарии: как, имея дело с двумя независимыми проектами, использовать один из них внутри другого?

Представим, что для веб-сайта вы создаете ленту в формате Atom. Вместо написания собственного генератора кода в этом формате вы решили воспользоваться библиотекой. Скорее всего, вам придется подключить код через библиотеки общего доступа, например CPAN или Rubygem, или скопировать его в дерево своего проекта. В первом случае вы столкнетесь с проблемой настройки библиотеки под свои нужды, кроме того, могут возникнуть сложности с внедрением вашего кода, так как потребуется гарантировать доступность этой библиотеки для всех пользователей. Решив же включить код в свой проект, вы столкнетесь со сложностями при слиянии ваших данных с изменениями из основного репозитория.

В Git эта задача решается с помощью подмодулей. Именно они позволяют превратить один Git-репозиторий в подпапку другого. Вы можете клонировать в проект дополнительный репозиторий, храня свои коммиты отдельно.

Начало работы

Рассмотрим процесс разработки простого проекта, разбитого на основную часть и несколько дополнительных.

Первым делом добавим существующий Git-репозиторий в качестве подмодуля нашего рабочего репозитория. Добавление новых подмодулей осуществляется командой `git submodule add` с указанием URL-адреса проекта, который вы хотите начать отслеживать. В данном случае добавим библиотеку `DbConnector`.

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

По умолчанию подмодули добавляют вложенные проекты в папки, имена которых совпадают с именами репозитория. То есть в нашем случае проект окажется в папке **DbConnector**. В конце данной команды можно указать путь к другому месту.

Если сейчас воспользоваться командой **git status**, станут очевидными несколько вещей.

```
$ 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, появился новый файл **.gitmodules**. Это конфигурационный файл, проецирующий URL-адрес проекта на локальную папку, в которую вы его скачали:

```
$ cat .gitmodules
[submodule "DbConnector"]
  path = DbConnector
  url = https://github.com/chaconinc/DbConnector
```

В случае набора подмодулей в этом файле будет несколько записей. Важно понимать, что Git контролирует этот файл наравне с остальными вашими файлами, такими как **.gitignore**. То есть вы можете отправлять его на сервер и скачивать оттуда. Это сделано для того, чтобы другие пользователи, клонировавшие ваш проект, могли узнать местоположение подмодулей.

ПРИМЕЧАНИЕ

Так как фигурирующий в файле **.gitmodules** URL-адрес является первым местом, откуда другие пользователи будут клонировать/скачивать информацию, вы должны по возможности гарантировать к нему доступ. Например, если для отправки информации на сервер вы пользуетесь не тем URL-адресом, с которого осуществляется скачивание данных, задействуйте адрес, доступ к которому есть и у остальных. Вы можете локально, исключительно для собственных нужд, переопределить этот адрес командой **git config submodule.DbConnector.url PRIVATE_URL**.

Следующим пунктом в списке вывода команды **git status** будет запись о папке проекта. Команда **git diff** покажет для нее 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
```

В данном случае **DbConnector** — это папка, вложенная в вашу рабочую папку, но Git рассматривает ее как подмодуль, и когда вы находитесь за ее пределами, не следит за ее содержимым. Вместо это система считает данную папку отдельным коммитом из репозитория.

Параметр `--submodule` делает вывод команды `diff` более понятным:

```
$ 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)
```

Вот результат фиксации текущего состояния:

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
2 files changed, 4 insertions(+)
create mode 100644 .gitmodules
create mode 160000 DbConnector
```

Обратите внимание на режим `160000` для записи **DbConnector**. Это специальный режим Git, по сути означающий, что вы записываете коммит как папку, а не как подпапку или файл.

Клонирование проекта с подмодулями

Посмотрим, каким образом осуществляется клонирование проекта, содержащего подмодуль. По умолчанию в этом случае вы получаете папки с подмодулями, но файлов в них пока нет:

```
$ 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
$
```

Итак, у нас появилась папка **DbConnector**, но пока что она пуста. Теперь нам потребуются две команды: **git submodule init** для инициализации локального конфигурационного файла и **git submodule update** для извлечения всех данных проекта и перехода к соответствующему коммиту, указанному 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 'c3f01dc8862123d317dd46284b05b6892c7b29
bc'
```

Теперь ваша папка **DbConnector** пришла в состояние, в котором она была на момент выполнения коммита.

Впрочем, этот результат можно получить и более простым способом. Добавив к команде **git clone** параметр **--recursive**, вы включите автоматическую инициализацию и обновление всех подмодулей в репозитории:

```
$ git clone --recursive https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for
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 'c3f01dc8862123d317dd46284b05b6892c7b29
bc'
```

Работа над проектом с подмодулями

Итак, мы получили копию проекта с подмодулями. Теперь можно рассмотреть процесс коллективной работы как над основным, так и над вложенным проектом.

Извлечение изменений из хранилища

Простейший вариант работы с подмодулями возникает, когда у вас есть вложенный проект и вы время от времени получаете оттуда обновления, не меняя ничего в копии данных. Давайте рассмотрим этот простой пример более подробно.

Чтобы проверить, какие изменения появились в подмодуле, перейдите в его папку и запустите команды `git fetch` и `git merge`, чтобы ваш локальный код обновился данными из вышележащей ветки:

```
$ git fetch
From https://github.com/chaconinc/DbConnector
   c3f01dc..d0354fc master   -> origin/master
Scotts-MacBook-Pro-3:DbConnector schacon$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
 scripts/connect.sh | 1 +
 src/db.c           | 1 +
 2 files changed, 2 insertions(+)
```

Теперь вернемся к основному проекту и воспользуемся командой `git diff --submodule`, чтобы удостовериться в обновлении подмодуля и получить список добавленных коммитов. Если вы не хотите вручную добавлять параметр `--submodule` при каждом вызове команды `git diff`, установите данный формат вывода как принятый по умолчанию, присвоив конфигурационному параметру `diff.submodule` значение `log`:

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
 > more efficient db routine
 > better connection routine
```

Зафиксировав данное состояние, вы сделаете так, что подмодуль начнет получать новый код при любом обновлении со стороны других пользователей.

Впрочем, если вы не хотите вручную извлекать данные и сливать их в подпапку, существует и более простой способ. Команда `git submodule update --remote` заставляет Git перейти к вашим подмодулям, извлечь данные и произвести обновление:

```
$ 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'
```

По умолчанию эта команда предполагает, что вы хотите обновить локальную копию до состояния ветки **master** из хранилища подмодуля. Но ее поведение можно изменить. Скажем, если вы хотите, чтобы подмодуль **DbConnector** следил за веткой **stable**, достаточно указать это в файле **.gitmodules** (в этом случае остальные пользователи также будут отслеживать эту ветку) или в вашем локальном файле **.git/config**. Рассмотрим первый вариант:

```
$ 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'
```

Без фрагмента **-f .gitmodules** команда сделает свои изменения локальными, но, как правило, имеет смысл следить за этими обновлениями через репозиторий, чтобы другие пользователи также могли это делать.

Выполненная на этом этапе команда **git status** покажет наличие в подмодуле новых коммитов:

```
$ 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 конфигурационных настройках параметр **status.submodulesummary**, Git начнет показывать и краткую сводку изменений в 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
```

Команда **git diff** покажет изменения в файле **.gitmodules** и извлеченные с сервера новые коммиты, которые можно зафиксировать в рамках вложенного проекта:

```
$ 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
```

Очень удобно, что мы можем посмотреть список коммитов, которые готовы к фиксации в нашем подмодуле. Эта информация будет доступна и после того, как у нас появится коммит, достаточно выполнить 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
```

По умолчанию команда `git submodule update --remote` будет приводить к обновлению всех ваших подмодулей, поэтому, если их у вас много, имеет смысл указывать имя того подмодуля, обновление которого вам требуется.

Работа с подмодулем

К подмодулям, как правило, прибегают, когда требуется сделать что-то с кодом подмодуля одновременно с работой над кодом основного проекта. В других случаях обычно используют более простую систему управления зависимостями (например, Maven или Rubygems).

Давайте посмотрим, как на практике осуществляется внесение изменений в подмодуль и основной проект и их одновременная фиксация и публикация.

До этого момента мы извлекали изменения из хранилищ подмодуля командой `git submodule update`. При этом система Git получала изменения и обновляла файлы в подпапке, но вложенное хранилище оставалось в так называемом состоянии обособленного указателя HEAD. То есть у нас отсутствовала локальная рабочая ветка (например, `master`), отслеживающая изменения. Соответственно, вносимые вами изменения не отслеживались.

Можно настроить подмодуль таким образом, чтобы с ним было проще работать. Для этого нам потребуются две вещи. Во-первых, в каждом подмодуле перейти в ветку, в которой вы собираетесь работать. Во-вторых, научить систему Git, что она должна делать, если вы внесли изменения, а затем командой `git submodule update --remote` получили новые наработки из репозитория. Можно слить их в локальную версию или, наоборот, попробовать переместить локальные наработки, расположив их поверх новых изменений.

Итак, перейдем в папку нашего подмодуля и переключимся в нужную ветку:

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

Попробуем выполнить вариант со слиянием. Чтобы проделать его вручную, достаточно добавить к команде `update` параметр `--merge`.

Теперь мы увидим все изменения состояния этого подмодуля на сервере и сможем добавить их к своей работе:

```
$ 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 папку **DbConnector**, мы увидим, что новые изменения уже слиты в локальную ветку **stable**. А теперь посмотрим, что произойдет, если мы отредактируем библиотеку локально, в то время как другой пользователь отправит собственный вариант изменений в вышележащее хранилище:

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'unicode support'
[stable f906e16] unicode support
 1 file changed, 1 insertion(+)
```

Обновим наш подмодуль, чтобы рассмотреть ситуацию, когда мы вносим локальные изменения при наличии на сервере изменений, которые нужно встроить в нашу работу:

```
$ 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'
```

Без параметров **--rebase** или **--merge** Git просто обновит подмодуль до состояния, которое он имеет на сервере, и переведет указатель **HEAD** в обособленное состояние:

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Ничего страшного в такой ситуации нет, достаточно вернуться в папку, переключиться в вашу ветку (все еще содержащую ваши наработки) и вручную слить или переместить ветку **origin/stable** (или другую нужную вам удаленную ветку).

Если на момент обновления подмодуля вы не зафиксировали локальные результаты его редактирования, Git извлечет изменения с сервера, но не перезапишет несохраненную работу в папке 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 конфликтуют с изменениями на сервере, Git сообщит вам об этом, когда вы решите выполнить обновление:

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path
'DbConnector'
```

После этого можно перейти в папку подмодуля и обычным образом разрешить конфликт.

Публикация результатов редактирования подмодуля

Итак, мы внесли некие изменения в папку нашего подмодуля. Часть из них получена путем скачивания обновлений с сервера, часть внесена нами локально и другим пользователям пока недоступна, ведь мы еще не отправили их на сервер:

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > updated setup script
  > unicode support
  > remove unnessesary method
  > add new option for conn pooling
```

Если, зафиксировав состояние основного проекта, мы отправим этот коммит на сервер, не добавив туда изменений, внесенных в подмодуль, другие пользователи, попытавшиеся скачать результаты нашего труда, столкнутся с проблемами, так как возможности заодно скачать изменения подмодуля, влияющие на общую ситуацию, у них не будет. Эти изменения существуют только в нашей локальной копии.

Чтобы избежать подобной ситуации, можно заставить систему Git перед отправкой на сервер результатов работы в рамках основного проекта удостовериться, что на сервере уже есть все результаты редактирования подмодулей. Команда **git push** принимает аргумент **--recurse-submodules**, который может иметь значение **check** или **on-demand**. При значении **check** попытка отправки данных на сервер просто завершится неудачей, если на сервере пока отсутствуют результаты редактирования подмодулей:

```
$ 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.
```

Обратите внимание, что эта команда дает нам советы, помогающие выбрать дальнейшее направление действий. Проще всего вручную войти в каждый подмодуль и отправить сведения о результатах его редактирования в общий доступ, а затем повторить общую команду **push**.

Или же вы можете воспользоваться значением **on-demand**, чтобы все эти действия были выполнены автоматически:

```
$ 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
```

В данном случае перед отправкой на сервер изменений, связанных с основным проектом, система Git переходит в папку модуля **DbConnector** и выполняет отправку изменений, связанных с подмодулем. Без успешного выполнения последней операции первая не осуществляется.

Слияние результатов редактирования подмодуля

Изменение ссылки на подмодуль одновременно с другим пользователем может создать проблемную ситуацию. Если истории подмодуля разошлись и были зафиксированы в разошедшихся ветках основного проекта, от вас потребуются некие корректирующие действия.

Когда один коммит является непосредственным предком другого (в случае слияния перемотки), для слияния Git выбирает последний коммит, и все прекрасно работает.

Но даже простейшего слияния система не выполнит вместо вас. Если зафиксированные состояния подмодуля расходятся и нужно объединить их друг с другом, вы получите примерно такую ситуацию:

```
$ 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.
```

Фактически система Git обнаружила, что в истории подмодуля присутствуют две разошедшиеся ветки, которые следует объединить. При этом вы видите непонятную запись «merge following commits not found» (слияние следующих коммитов не обнаружено). Позже мы поговорим о том, откуда она взялась.

Для решения проблемы следует выбрать, какое состояние подмодуля нам требуется. Как ни странно, в данном случае Git не дает нам вспомогательной информации. Вы не найдете сведений даже о контрольных суммах SHA коммитов из обоих фрагментов истории. К счастью, получить эти сведения несложно. Команда **git diff** даст вам значения SHA обоих коммитов, которые вы пытаетесь слить друг с другом:

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

Итак, в данном случае **eb41d76** — это коммит в нашей версии подмодуля, а **c771610** — коммит из вышележащего хранилища. При переходе в папку подмодуля мы окажемся в коммите **eb41d76**, так как слияние его не затронуло. Если по каким-то причинам это не так, достаточно создать указывающую на этот коммит ветку и перейти в нее.

Куда важнее в данной ситуации значение SHA второго коммита. Ведь именно в него нам нужно слить наши изменения, попутно разрешив конфликт. Можно попытаться сделать это, непосредственно указав SHA, а можно создать для этого коммита отдельную ветку и выполнить слияние уже в нее. Мы предлагаем использовать второй вариант хотя бы потому, что сообщение фиксации в этом случае получается более аккуратным.

Итак, перейдем в папку подмодуля, создадим на основе второй контрольной суммы SHA, которую нам дает команда **git diff**, новую ветку и выполним слияние вручную:

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

В данном случае имеет место настоящий конфликт слияния, поэтому разрешим его и зафиксируем состояние, после чего останется только обновить основной проект:

```
$ vim src/main.c (1)
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ cd .. (2)
$ git diff (3)
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
-Subproject commit c77161012afb1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector (4)

$ git commit -m "Merge Tom's Changes" (5)
[master 10d2c60] Merge Tom's Changes
```

Вся процедура выглядит так:

1. Сначала мы разрешаем конфликт.
2. Затем возвращаемся в папку основного проекта.
3. Еще раз проверяем контрольные суммы SHA.
4. Разрешаем конфликтующие записи подмодуля.
5. Фиксируем результат слияния.

Это может показаться сложным, но на самом деле ничего сложного в этой ситуации нет.

Что интересно, существует один аспект, в котором система Git проявляет интеллект. Если в папке подмодуля, содержащей оба коммита из истории этого подмодуля, существует и коммит слияния, Git предлагает его в качестве возможного решения. Она видит, что в какой-то точке проекта слияние веток, содержащих эти два коммита, уже было выполнено, и предполагает, что, возможно, именно этот вариант вам и требуется.

По этой причине в приведенных выходных данных команды появилась запись об ошибке: «merge following commits not found». Система просто не смогла предложить вам коммит, полученный в результате слияния, так как его пока не существует. Но откуда нам было заранее знать, что она попытается так поступить?

Если бы система обнаружила подходящий коммит, вы увидели бы примерно следующее:

```
$ 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.
```

Предполагается, что вы должны обновить область индексирования, выполнив команду `git add`, убирающую сведения о конфликте, а затем создать коммит. Но можно пойти и другим путем. Например, перейти в папку подмодуля, посмотреть, какие изменения в ней появились, выполнить перемотку до указанного коммита, произвести его тестирование и после этого зафиксировать его состояние:

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward
$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forwarded to a common submodule child'
```

Вы получите тот же самый результат, но попутно удостоверитесь в том, что все работает, а после завершения работы будете иметь данный код в папке подмодуля.

Полезные советы

Существуют приемы, несколько упрощающие работу с подмодулями.

Команда `foreach`

Команда `foreach` позволяет выполнить в каждом подмодуле произвольную команду. Она крайне облегчает работу с проектами, в которых существует огромное количество подмодулей.

Предположим, вы хотите добавить новую конструктивную особенность или исправить какую-то ошибку, причем в процессе этой работы вам придется иметь дело с несколькими подмодулями. Все наработки из этих подмодулей легко можно скрыть:

```
$ 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
```

После этого можно создать новую ветку и во всех подмодулях перейти в нее:

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Switched to a new branch 'featureA'
Entering 'DbConnector'
Switched to a new branch 'featureA'
```

Надеемся, основную идею вы поняли. Одним из самых полезных вариантов применения этой команды является создание унифицированного списка изменений, внесенных как в основной проект, так и во все его подмодули:

```
$ 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;
```

Обратите внимание, что в данном фрагменте кода мы определили в подмодуле функцию, которая затем вызывается из основной части проекта. Это крайне упрощенный пример, но он прекрасно иллюстрирует принцип работы команды **foreach**.

Псевдонимы

В случае длинных команд, для конфигурационных параметров которых зачастую нельзя задать значения, предлагаемые по умолчанию, имеет смысл настроить псевдонимы. Ранее мы уже рассматривали эту процедуру довольно подробно, теперь коснемся только аспектов, которые могут понадобиться при активной работе с подмодулями.

```
$ 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'
```

После этого для обновления подмодулей достаточно будет воспользоваться командой **git supdate**, а отправку изменений на сервер с проверкой подмодулей будет выполнять команда **git spush**.

Проблемы, возникающие при работе с подмодулями

При работе с подмодулями часто возникают различные затруднения.

Например, нетривиальной порой оказывается смена веток, в которых присутствуют подмодули. Если создать новую ветку, добавить туда подмодуль и вернуться в ветку, в которой он отсутствует, у вас все равно обнаружится неотслеживаемая папка подмодуля:

```
$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...
$ git commit -am 'adding crypto library'
[add-crypto 4445836] adding crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
```

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

```
CryptoLibrary/
```

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

Эту папку можно просто удалить, но вопрос в том, почему она вообще появилась. Вернувшись после удаления этой папки в ветку с подмодулем, вы обнаружите, что тут для повторного создания папки следует воспользоваться командой **submodule update --init**:

```
$ 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
```

Ничего сложного в этом нет, но логика происходящего не совсем понятна.

Другой подводный камень подстерегает нас при переходе от подпапок к подмодулям. Следует с крайней осторожностью перемещать в подмодуль отслеживаемые файлы. Предположим, что в подпапке проекта есть некие файлы, которые вы хотите поместить в подмодуль. Команда **submodule add** после удаления подпапки вызовет вот такую реакцию:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

Сначала нужно было удалить папку **CryptoLibrary** из области индексирования. И только после этого 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.
```

А теперь предположим, что вы сделали это в какой-то ветке. При попытке перейти в ветку, в которой эти файлы все еще располагаются в основном дереве, а не в подмодуле, вы получите сообщение об ошибке:

```
$ 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
```

Можно перейти в эту ветку принудительно, воспользовавшись командой **checkout -f**, но при этом обязательно удостоверьтесь, что у вас нет несохраненных изменений, которые эта команда может уничтожить:

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Вернувшись обратно, вы неожиданно обнаружите пустую папку **CryptoLibrary**, и команда **git submodule** исправить эту ситуацию не поможет. Чтобы вернуть файлы, перейдите в папку подмодуля и воспользуйтесь командой **git checkout**. Сценарий **submodule foreach** позволяет запустить ее сразу для нескольких подмодулей.

Важно отметить, что в настоящее время все свои служебные данные подмодули хранят в папке **.git** основного проекта, поэтому в отличие от более старых версий Git удаление папки подмодуля не приводит к потере коммитов или веток.

Все эти инструменты делают подмодули достаточно простым и эффективным средством одновременной разработки нескольких связанных друг с другом отдельных проектов.

Пакеты

Основные средства передачи Git-данных по сети (HTTP, SSH и т. п.) мы уже рассмотрели, но существует еще один, менее распространенный способ, который в некоторых ситуациях может оказаться весьма полезным.

Система Git умеет «упаковывать» данные в один файл. Существуют разные сценарии, в которых такое поведение востребовано. Представьте, что вам срочно нужно отправить наработки коллегам, а ваша сеть вышла из строя. Или вы находитесь вне офиса и по соображениям безопасности не имеете доступа к локальной сети. Или вышла из строя плата беспроводной связи либо сети ethernet. Или в текущий момент у вас нет доступа к общему серверу, а вы хотите переслать кому-то по электронной почте свои обновления, не передавая 40 коммитов командой **format-patch**.

Именно в таких случаях вам на помощь приходит команда **git bundle**. Она упаковывает в бинарный файл все данные, которые в обычной ситуации вы отправили бы на сервер командой **git push**, и этот файл можно будет без проблем переслать по электронной почте или записать на флеш-накопитель, а затем распаковать в целевом хранилище.

Рассмотрим простой пример. Предположим, у вас есть хранилище 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 переслать кому-то этот репозиторий, но не имеете доступа на запись или не хотите его настраивать, воспользуйтесь 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)
```

Появится файл **repo.bundle** со всеми необходимыми для воссоздания ветки **master** вашего репозитория данными. Для команды **bundle** следует составить список всех ссылок или указать диапазон коммитов, которые следует включить в пакет. Если репозиторий предназначен для клонирования на новом месте, в пакет добавляется еще и указатель **HEAD**, как и было сделано в данном случае.

Теперь этот файл **repo.bundle** можно отправить по электронной почте или записать на USB-диск.

А что делать, если вы получили этот файл и собираетесь работать над проектом? Репозиторий из бинарного файла можно клонировать в папку совершенно так же, как вы делали это в случае URL-адреса:

```
$ git clone repo.bundle repo
Initialized empty Git repository in /private/tmp/bundle/repo/.git/
$ cd repo
$ git log --oneline
9a466c5 second commit
b1ec324 first commit
```

Если указатель **HEAD** не входит в список ссылок, при распаковке нужно будет указать **-b master** или другую присутствующую в пакете ветку, в противном случае система не будет знать, в какую ветку ей следует перейти.

Предположим, вы сформировали три коммита и хотите отправить их обратно в виде пакета:

```
$ git log --oneline
71b84da last commit - second repo
c99cf5b fourth commit - second repo
```



```
7011d3d third commit - second repo
9a466c5 second commit
b1ec324 first commit
```

Первым делом определяется диапазон коммитов, которые мы хотим включить в пакет. Если сетевые протоколы самостоятельно определяют минимальный размер передаваемых по сети данных, то сейчас это нужно сделать вручную. В рассматриваемом примере можно просто упаковать весь репозиторий, но лучше включить в пакет только изменения — наши три локальных коммита.

Для этого требуется определить, в чем состоят различия. Указать диапазон коммитов, как вы уже знаете, можно разными способами. Чтобы получить три коммита из ветки `master`, которых не было на момент ее клонирования, можно написать `origin/master..master` или `master ^origin/master`. Проверить результат можно командой `log`:

```
$ git log --oneline master ^origin/master
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
```

Итак, теперь у нас есть список коммитов, которые будут включены в пакет, и можно приступить к процедуре создания пакета. Для этого мы воспользуемся командой `git bundle create`, передав ей имя будущего пакета и диапазон коммитов, которые следует в этот пакет включить:

```
$ 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)
```

Теперь в нашей папке появился файл `commits.bundle`. Если отправить его коллеге, тот сможет импортировать наши коммиты в исходный репозиторий, даже если там параллельно велась некая работа.

Перед импортированием пакета в репозиторий можно посмотреть его содержимое. Это делается командой `bundle verify`, которая проверяет корректность пакета и наличие всех предков коммитов, необходимых для правильного восстановления данных:

```
$ 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 создать пакет только с двумя последними коммитами, репозиторий не сможет импортировать его содержимое из-за отсутствия необходимого фрагмента истории. В подобных случаях команда `verify` возвращает вот такой результат:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second repo
```

Тем не менее созданный нами пакет совершенно корректен, поэтому мы можем извлечь оттуда коммиты. Если вы хотите посмотреть, какие ветки внутри пакета доступны для импорта, можно вывести только список веток:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

Команда **verify**, примененная к папке **heads**, покажет вам уже ее содержимое. Предположим, мы хотим посмотреть, что именно можно извлечь, чтобы импортировать из пакета коммиты командой **fetch** или **pull**. Извлечем ветку **master**, поместив ее содержимое 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 в ветке **other-master** импортированные коммиты, а заодно и все коммиты, сделанные за это 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
```

Как видите, команда **git bundle** позволяет делиться своими наработками и, по сути, выполнять операции, даже если у вас нет сетевого доступа или возможности воспользоваться общим репозиторием.

Замена

Хотя Git-объекты неизменяемы, существует интересный вариант, эмулирующий замену одних объектов в базе другими.

Команда **replace** позволяет указать объект и объяснить Git, что его следует принимать за другой объект. Эта возможность пригодится вам, если нужно будет заменить один коммит в истории другим.

Предположим, у вас есть проект с длинной историей и вы хотите разбить репозиторий на две части: с короткой историей для новых разработчиков и с более подробной историей для тех, кого интересует анализ данных. Можно присоединить одну историю к другой, заменив первый коммит в коротком варианте последним

коммитом из длинного. При этом вам не придется переписывать каждый коммит в новой истории, как это обычно происходит при объединении (так как отношения родства влияют на контрольные суммы SHA).

Посмотрим на практике, как это работает. Разобьем существующий репозиторий на две части, одна из которых содержит свежие результаты редактирования, а вторая — коммиты, составляющие историю проекта. А затем посмотрим, каким образом команда **replace** позволяет объединить их, не меняя их SHA.

Возьмем 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 это на две исторические линии. Первая начинается в первом и заканчивается четвертым коммитом. Вторая будет содержать только четвертый и пятый коммиты, иллюстрируя новейшую историю проекта (рис. 7.27).

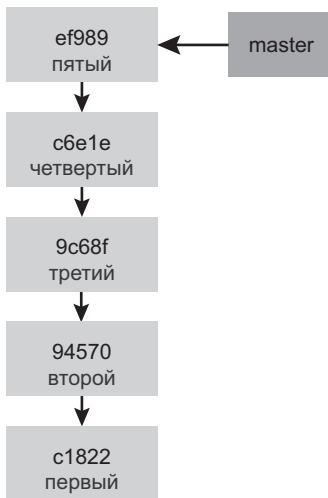


Рис. 7.27

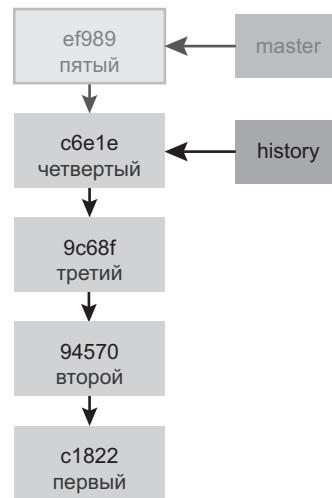


Рис. 7.28

Создать исторический фрагмент очень просто. Достаточно вставить ветку в историю и отправить ее содержимое в ветку **master** нового удаленного репозитория (рис. 7.28):

```
$ 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 только что созданную ветку `history` можно отправить 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
```

Итак, историю проекта мы опубликовали, теперь осталась более сложная часть — выделение в отдельную ветку новейшей истории. Два наших варианта истории должны перекрываться, чтобы коммит из одной части можно было заменить эквивалентным коммитом из другой, поэтому мы оставим четвертый и пятый коммиты (четвертый коммит сформирует область пересечения):

```
$ 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 имеет смысл создать базовый коммит с инструкцией получения доступа к истории, чтобы другие разработчики знали, как поступить, если, встретив коммит с урезанной историей, они захотят узнать ее целиком. Поэтому мы создадим начальный объект-коммит, который послужит отправной точкой, и поместим поверх него оставшиеся коммиты (четыре и пять).

Для этого нам нужно выбрать точку разбиения. В нашем случае это будет третий коммит, то есть коммит с контрольной суммой `9c68fdc`. Наш базовый коммит будет основываться на этом дереве. Для его создания мы воспользуемся командой `commit-tree`, которая берет дерево и возвращает нам контрольную сумму SHA нового, не имеющего предков коммита:

```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

ПРИМЕЧАНИЕ

Команда `commit-tree` входит в набор команд, которые обычно называются служебными. Как правило, непосредственно они не применяются, но ими пользуются другие Git-команды для решения небольших задач. Периодически, когда нам, как в данном случае, нужно выполнить нестандартную операцию, именно такие команды позволяют действовать на низком уровне, но в повседневной работе они практически не применяются.

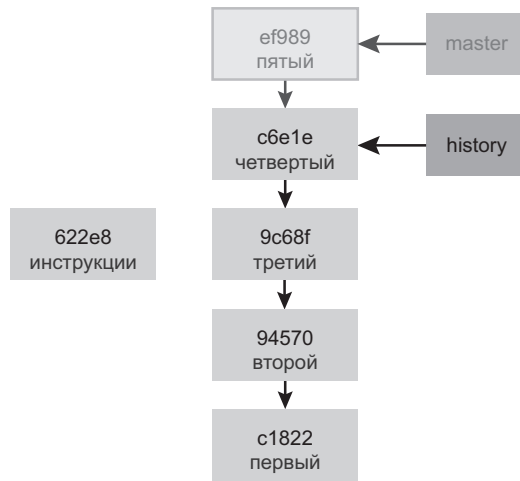


Рис. 7.29

Итак, у нас есть базовый коммит, и при помощи команды `git rebase --onto` мы переместим в него остальную историю. Значением аргумента `--onto` послужит контрольная сумма SHA, только что полученная нами от команды `commit-tree`, а точкой перемещения станет третий коммит (предок коммита `9c68fdc`, который мы хотим сохранить):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```

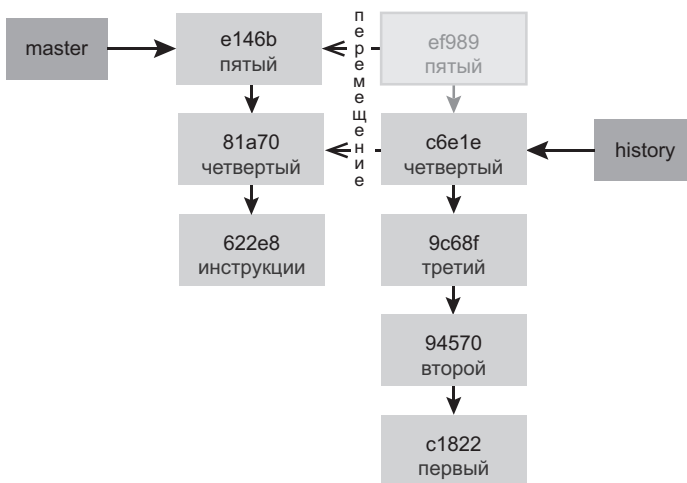


Рис. 7.30

Таким образом, мы записали нашу новейшую историю поверх вспомогательного базового коммита, содержащего инструкцию по восстановлению полной версии. Можно отправить эту новую историю в новый проект, и клонировавшие его пользователи получат только два последних коммита и базовый коммит с инструкцией.

Теперь представим, что мы клонировали чужой проект и хотим узнать его полную историю. Чтобы получить нужные нам данные после клонирования усеченной версии репозитория, требуется добавить в список своих удаленных репозиториив репозиторий, содержащий историю, и извлечь оттуда всю информацию:

```
$ 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 коллеги, с которым вы сотрудничаете, последние коммиты будут находиться в ветке **master**, а связанные с историей 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 можно воспользоваться командой **git replace**, указав коммит, который вы хотите заменить, а затем коммит, который должен послужить заменой. Вот как будет выглядеть команда, замещающая «четвертый» коммит из ветки **master** «четвертым» коммитом из ветки **project-history/master**:

```
$ git replace 81a708d c6e1e95
```

Теперь история ветки **master** должна выглядеть примерно так:

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Здорово, правда? Нам не пришлось менять контрольные суммы SHA всех коммитов из вышележащего репозитория, чтобы заместить один коммит в истории другим. При этом обычные инструменты (**bisect**, **blame** и т. п.) работают именно так, как мы того ожидаем.

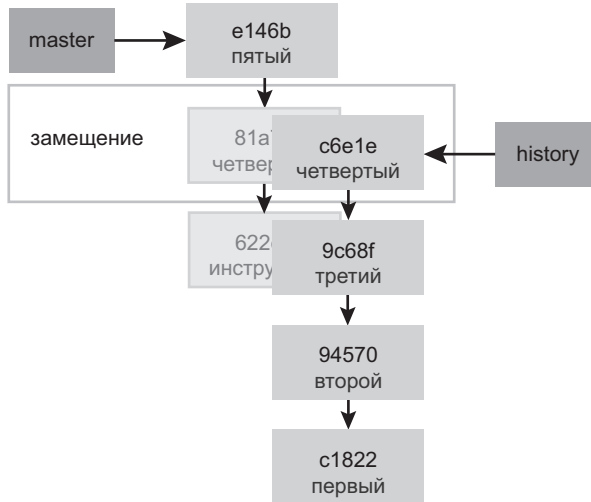


Рис. 7.31

Что интересно, для замещенного коммита все равно выводится контрольная сумма SHA **81a708d**, хотя на самом деле там используется контрольная сумма **с6e1e95** коммита, послужившего заменой. И даже такая команда, как **cat-file**, выводит замещенные 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

Напоминаем, что реальным предком коммита **81a708d** является наш вспомогательный коммит (**622e88e**), а не фигурирующий здесь коммит **9c68fdce**.

Также следует отметить, что информация о замене сохраняется в наших ссылках:

```
$ git for-each-ref
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master
с6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/remotes/history/master
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/master
с6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/replace/81a708dd0e167a3f69154
1c7a6463343b
с457040
```

То есть результатами замещения легко поделиться с другими пользователями, так как их можно отправить на сервер, где они будут доступны для скачивания. В рассмотренном случае замещения истории это особого смысла не имеет (так как все равно будут скачиваться оба варианта истории и непонятно, зачем мы их разбивали), но в других случаях подобная возможность может пригодиться.

Хранение учетных данных

Если для подключения к удаленному серверу вы пользуетесь протоколом SSH, у вас может быть ключ без парольной фразы, что позволяет безопасно передавать данные без ввода имени пользователя и пароля. Однако с протоколами семейства HTTP подобное невозможно — ввод имени пользователя и пароля требуется при каждом подключении. Еще сложнее дело обстоит в системах с двухфакторной аутентификацией, где используемый в качестве пароля токен генерируется случайным образом.

К счастью, для таких случаев в Git существует система управления учетными данными. Возможны несколько вариантов:

- ❑ По умолчанию учетные данные не кэшируются. При каждом подключении вам предлагается ввести имя пользователя и пароль.
- ❑ В режиме кэша учетные данные определенное время хранятся в памяти. Пароли никогда не сохраняются на диске и удаляются из кэша через 15 минут.
- ❑ В режиме сохранения учетные данные сохраняются в обычном текстовом файле. Поэтому пока вы не поменяете пароль к Git-серверу, учетные данные вводить не придется. Но, к сожалению, в данном случае ваши пароли хранятся в открытом виде в домашней папке.
- ❑ Для пользователей Mac в Git есть режим «osxkeychain», в котором учетные данные помещаются в защищенный репозиторий, привязанный к системной учетной записи. Хранение осуществляется в течение неограниченного времени, но для шифрования применяется та же самая система, которая сохраняет HTTPS-сертификаты и данные автозаполнения для браузера Safari.
- ❑ В операционной системе Windows доступен помощник «winstore». Он напоминает описанный режим «osxkeychain», но для управления конфиденциальной информацией применяется репозиторий Windows Credential Store. Его можно найти по адресу <https://gitcredentialstore.codeplex.com>.

Выбор режима осуществляется путем редактирования конфигурации системы Git:

```
$ git config --global credential.helper cache
```

Некоторые из этих режимов имеют параметры. Например, для режима **store** можно указать аргумент **--file <путь>**, который задает место хранения текстового файла с учетными данными (по умолчанию он находится в папке `~/.git-credentials`).

Для режима **cache** указывается аргумент **--timeout <секунд>**, меняющий время работы демона (по умолчанию этот аргумент имеет значение «900», что составляет 15 минут). Зададим в режиме **store** собственный файл:

```
$ git config --global credential.helper store --file ~/.my-credentials
```

В Git можно настраивать сразу несколько режимов. При поиске учетных данных для сервера Git по очереди их проверяет, прекращая проверку после получения первого ответа. Сохраняемые учетные данные рассылаются всему списку режимов, которые самостоятельно решают, что им с этими данными делать. Вот как будет выглядеть файл **.gitconfig**, когда файл с учетными данными хранится на диске, но на случай его отсутствия вы хотите кэшировать данные в оперативной памяти:

```
[credential]
  helper = store --file /mnt/thumbdrive/.git-credentials
  helper = cache --timeout 30000
```

Взгляд изнутри

Как же это все работает? Корневой командой системы режимов авторизации в Git является команда **git credential**, принимающая указания через аргумент, а остальные данные — через стандартный поток ввода.

Понять это проще на примере. Предположим, вы настроили режим авторизации, сохранив учетные данные для сервера **mygithost**. Вот сеанс, использующий команду **fill**, которая вызывается при попытке Git найти учетные 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
```

Вся процедура выглядит так:

1. Эта строка иницирует взаимодействие.
2. После этого учетные Git-данные ожидают сведений из стандартного потока ввода. Мы передаем туда известную нам информацию: протокол и имя сервера.

3. Пустая строка означает завершение ввода, а система учетных записей должна показать, что она это понимает.
4. После этого начинается работа с учетными Git-данными, а обнаруженные биты информации система записывает в стандартный поток вывода.
5. Если учетные данные не обнаружены, система Git запрашивает имя пользователя и пароль и возвращает их в активированный поток вывода (в данном примере эти потоки связаны с одной и той же консолью).

Но на самом деле система управления учетными данными вызывает программу, существующую отдельно от Git; что это за программа и каким образом она вызывается, зависит от значения конфигурационной переменной `credential.helper`. Возможны несколько вариантов:

Значение переменной	Поведение
<code>foo</code>	Запускается <code>git-credential-foo</code>
<code>foo -a --opt=bcd</code>	Запускается <code>git-credential-foo -a --opt=bcd</code>
<code>/absolute/path/foo -xyz</code>	Запускается <code>/absolute/path/foo -xyz</code>
<code>!f() { echo "password=s3cre7"; }; f</code>	Указанное после символа <code>!</code> оценивается в интерпретаторе команд

Итак, полные названия описанных режимов: `git-credential-cache`, `git-credential-store` и т. п. Их можно настроить на прием аргументов командной строки. В обобщенной форме это выглядит так: `git-credential-foo [аргументы] <действие>`. Протокол ввода-вывода такой же, как и у команды `git-credential`, просто теперь используется немного другой набор операций:

- ❑ `get` — запрос пары имя пользователя/пароль;
- ❑ `store` — запрос на сохранение набора учетных данных в памяти;
- ❑ `erase` — удаление учетных данных из памяти для указанных свойств.

Операции `store` и `erase` обратной реакции не требуют (впрочем, Git их в любом случае игнорирует). Но для операции `get` реакция активируемого режима крайне важна. Если полезная информация отсутствует, команда включения режима просто завершает свою работу, если же дополнительные данные появляются, они добавляются к сохраненной информации. Выводимое командой выглядит как набор операторов присваивания; все введенные значения будут заменены имеющимися в Git данными.

Снова рассмотрим пример, который приводился ранее, но на этот раз пропустим часть `git-credential` и перейдем непосредственно к команде `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
```

Вся процедура выглядит так:

1. Здесь мы просим команду **git-credential-store** сохранить учетные данные: имя пользователя **bob** и пароль **s3cre7**, которые потребуются для доступа к серверу **https://mygithost**.
2. Далее мы извлекаем эти учетные данные. Мы передаем часть уже известных нам параметров подключения (**https://mygithost**) и пустую строку.
3. Команда **git-credential-store** возвращает ранее сохраненные нами имя пользователя и пароль.

Вот как выглядит содержимое файла **~/git.store**:

```
https://bob:s3cre7@mygithost
```

Это обычный набор строк, каждая из которых содержит URL-адрес с добавленными к нему учетными данными. В режимах **osxkeychain** и **winstore** используется собственный формат хранилищ, в то время как режим кэша задействует свой формат хранения во внутренней памяти (недоступный для чтения другими процессами).

Нестандартный вариант хранения учетных данных

Так как рассмотренная команда **git-credential-store** и ей подобные вызывают не связанные с Git программы, несложно сообразить, что вспомогательным средством для работы с учетными данными может быть любая программа. Программы, связанные с Git, подходят к многим, но не ко всем возможным случаям. К примеру, предположим, что для развертывания все члены вашей группы пользуются одними и теми же учетными данными. Они хранятся в общедоступной папке, а в собственный репозиторий учетных данных вы их копировать не хотите, так как они часто меняются. Подходящей вспомогательной программы для этого случая нет, а значит, нужно писать собственную. При этом должны использоваться следующие критерии:

1. Единственные действия, которым нужно уделить внимание — **get**, **store** и **erase**, — являются операциями записи, поэтому после их выполнения программа должна просто прекращать работу.

2. Для файла с общедоступными учетными данными мы будем использовать формат, знакомый нам по команде **git-credential-store**.
3. Файл будет располагаться в обычном месте, но пользователь при желании должен иметь возможность указать собственный путь к файлу.

Это расширение мы тоже напишем на языке Ruby, но вы можете применить любой другой язык при условии, что Git сможет выполнить готовую программу. Вот полный исходный код нашей новой программы для работы с учетными данными:

```
#!/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'] then
    puts "protocol=#{prot}"
    puts "host=#{host}"
    puts "username=#{user}"
    puts "password=#{pass}"
    exit(0)
  end
end
```

Вся процедура выглядит так:

1. Здесь мы анализируем параметры командной строки, позволяя пользователю указать входной файл. По умолчанию это файл **~/.git-credentials**.
2. Эта программа отвечает только в случае операции **get** и при наличии файла хранилища.
3. Этот цикл читает данные стандартного ввода, пока не появляется пустая строка. Введенные данные сохраняются в известный хеш для дальнейших ссылок.

4. Этот цикл читает содержимое хранилища, выполняя поиск соответствия. Если протокол и сервер, указанные в хеше, совпадают с данной строкой, программа выводит результат стандартным способом и прекращает свою работу.

Сохраним нашу вспомогательную программу под именем **git-credential-read-only**, сделаем ее исполняемой и поместим в одну из папок, вложенных в папку PATH. Вот вариант интерактивного сеанса:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

Так как имя программы начинается с **git-**, для конфигурационного значения можно использовать простой синтаксис:

```
$ git config --global credential.helper read-only --file /mnt/shared/creds
```

Как видите, расширение системы выполняется очень просто, что позволяет решать распространенные проблемы, возникающие в процессе работы.

Заключение

Вы познакомились с инструментами с расширенной функциональностью, увеличивающими точность управления коммитами и областью индексирования. Именно они позволят вам в случае проблем легко определить, какой коммит стал их источником, а также когда и кем он добавлен. Вы узнали, каким образом добавить в свой проект вложенные проекты. Фактически вы уже должны быть в состоянии уверенно решать большинство рутинных задач, возникающих при работе с Git.

8 Настройка системы Git

До этого момента мы говорили о том, как функционирует система Git и как ею пользоваться, попутно рассмотрев ряд инструментов, делающих работу с Git простой и эффективной. В этой главе вы узнаете, как заставить Git работать нужным вам образом. Мы покажем вам ряд важных конфигурационных параметров и познакомим с системой хуков. Эти инструменты дадут возможность легко настроить Git нужным вам, вашей фирме или вашей рабочей группе образом.

Конфигурирование системы Git

Как вы узнали в главе 1, конфигурирование системы Git выполняется командой `git config`. С ее помощью мы меняли имя и электронный адрес пользователя:

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

Теперь пришла пора познакомиться с более интересными вариантами настройки Git.

Напомним, что задание желаемого поведения в Git осуществляется с помощью набора конфигурационных файлов. Первым делом конфигурационные параметры Git берет из файла `/etc/gitconfig`, содержащего значения для всех пользователей системы и всех репозиториях этих пользователей. Параметр `--system`, добавленный к команде `git config`, инициирует именно чтение из этого файла и запись в него.

Затем Git смотрит в файл `~/.gitconfig` (или `~/.config/git/config`), привязанный к конкретному пользователю. Чтение и запись этого файла активизирует передача параметра `--global`.

Последним местом поиска конфигурационных параметров является файл в папке используемого в текущий момент репозитория (`.git/config`). Находящиеся в этом файле значения связаны с конкретным репозиторием. Значения с каждого уровня (системный, глобальный, локальный) переопределяют значения, заданные на предыдущем уровне. Например, значения в файле `.git/config` переписывают значения в файле `/etc/gitconfig`.

ПРИМЕЧАНИЕ

Свои параметры Git хранит в конфигурационных файлах настройки в виде обычного текста, поэтому при условии применения корректного синтаксиса менять эти значения можно вручную, просто редактируя конфигурационные файлы. Однако, как правило, проще воспользоваться командой `git config`.

Основные настройки на стороне клиента

Конфигурационные параметры, с которыми работает Git, делятся на две категории: находящиеся на стороне клиента и на стороне сервера. Большинство параметров клиентские — они задают ваши персональные предпочтения. Из огромного числа существующих параметров большая часть используется только в редких случаях. Мы рассмотрим только самые распространенные и самые полезные варианты. Посмотреть список всех доступных параметров позволяет команда `$ man git-config`. Выводимые ею данные содержат довольно подробное описание каждого параметра. Кроме того, этот справочный материал доступен на странице <http://git-scm.com/docs/git-config.html>.

Параметр `core.editor`

Для создания и редактирования сообщений фиксации и комментариев к тегам в Git применяется либо текстовый редактор, который система выбирает по умолчанию (`$VISUAL` или `$EDITOR`), либо редактор `vi`. Изменить эту ситуацию позволяет параметр `core.editor`:

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

Теперь вне зависимости от того, каким редактором оболочка пользуется по умолчанию, для редактирования сообщений система Git будет вызывать редактор Emacs.

Переменная `commit.template`

В этой переменной можно задать путь к файлу, содержимое которого Git будет по умолчанию использовать как сообщение фиксации при создании коммитов. Предположим, у вас есть файл с шаблоном по адресу `~/ .gitmessage.txt`. Вот как он выглядит:

```
Строка заголовка
Описание действия
[карточка: X]
```

Чтобы система Git начала по умолчанию показывать это сообщение в редакторе после команды `git commit`, присвойте переменной `commit.template` следующее значение:

```
$ git config --global commit.template ~/.gitmessage.txt
$ git commit
```

Теперь при создании коммита содержимое вашего редактора будет выглядеть так:

строка заголовка

описание действия

```
[карточка: 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 у вашей рабочей группы существуют определенные правила для сообщений фиксации, создание соответствующего шаблона и настройка Git, заставляющая по умолчанию использовать этот шаблон, повышают вероятность их соблюдения.

Параметр `core.pager`

Параметр `core.pager` задает способ организации постраничного вывода результатов работы таких команд, как `log` и `diff`. Вы можете присвоить ему значение `more`, указать другую служебную программу (по умолчанию используется программа `less`) или вообще отключить эту настройку, присвоив параметру пустую строку:

```
$ git config --global core.pager ''
```

После этого Git будет выводить результат работы команды на одной странице вне зависимости от объема этих данных.

Параметр `user.signingkey`

Чтобы облегчить процесс создания подписанных тегов с комментариями, можно в настройках указать ключ GPG-подписи. Вот процедура задания идентификатора ключа:

```
$ git config --global user.signingkey <gpg-key-id>
```

Теперь для подписи тега не требуется каждый раз при вызове команды `git tag` указывать ключ:

```
$ git tag -s <tag-name>
```


Параметр `core.excludesfile`

Чтобы система Git не рассматривала определенные файлы как неотслеживаемые и не пыталась индексировать их при выполнении команды `git add`, следует создать шаблон в файле `.gitignore` вашего проекта. Но иногда возникает необходимость сделать так, чтобы система игнорировала определенные файлы для всех ваших репозиториев. Если вы работаете на компьютере с операционной системой Mac OS X, скорее всего, вам знакомы файлы `.DS_Store`. Если же вы предпочитаете пользоваться редактором Emacs или Vim, значит, вы знакомы с файлами, имена которых оканчиваются символом `~`.

Параметр `core.excludesfile` позволяет записать файл `.gitignore`, действующий на глобальном уровне. Достаточно создать файл `~/.gitignore_global` с вот таким содержимым:

```
*~
.DS_Store
```

Если после этого запустить команду `git config --global core.excludesfile ~/.gitignore_global`, Git никогда не будет беспокоить вас по поводу этих файлов.

Параметр `help.autocorrect`

Неправильный набор команды приводит к вот такому результату:

```
$ git chekcout master
git: 'chekcout' is not a git command. See 'git --help'.

Did you mean this?
    checkout
```

Система Git вежливо пытается выяснить, что вы имели в виду, но никаких действий не предпринимает. Если же присвоить параметру `help.autocorrect` значение 1, Git автоматически запустит команду, если она является единственным подходящим вариантом:

```
$ 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...
```

Учитывайте, что запись `0.1 seconds` (0,1 секунды) в переменной `help.autocorrect` на самом деле соответствует десятой части секунды и представляет собой целое число. Если присвоить этой переменной значение 50, система Git даст вам 5 секунд на размышление и только после этого выполнит скорректированную команду.

Цвета в Git

В системе Git поддерживается цветовое оформление консольного вывода, что упрощает визуальный анализ результатов применения команд. Ряд параметров поможет вам выбрать цвета по своему вкусу.

Параметр `color.ui`

В системе Git большая часть выводимых данных выделяется цветами автоматически, но это поведение можно поменять. Для отключения цветового оформления выполните команду:

```
$ git config --global color.ui false
```

По умолчанию этот параметр имеет значение **auto**. В этом случае цветовое оформление добавляется только при выводе на терминал. Когда вывод перенаправляется на конвейер или в файл, оно опускается.

Кроме того, существует значение **always**, при котором цвета добавляются всегда. Оно практически никогда не используется; в большинстве случаев, если вы хотите оформить цветом перенаправленный вывод, достаточно добавить к команде флаг `--color`. А для подавляющего большинства случаев прекрасно подходит значение этого параметра, предлагаемое по умолчанию.

Параметр `color.*`

Для тех, кто хочет более подробно указать, какие команды какими цветами следует выделить, существуют дополнительные параметры. Каждому из них можно присвоить значение **true**, **false** или **always**:

```
color.branch  
color.diff  
color.interactive  
color.status
```

Кроме того, у каждого из этих параметров есть собственные параметры, задающие цвета частей вывода и позволяющие переопределить каждый цвет. Например, вот как можно добиться того, чтобы выводимые командой `diff` метаданные окрасились синим цветом на черном фоне и были выделены полужирным шрифтом:

```
$ git config --global color.diff.meta "blue black bold"
```

Отвечающий за задание цвета параметр может принимать следующие значения: **normal**, **black**, **red**, **green**, **yellow**, **blue**, **magenta**, **cyan** или **white**. Для задания дополнительных характеристик (например, в представленном примере мы делали шрифт полужирным) используйте значения **bold**, **dim**, **ul** (подчеркивание), **blink** и **reverse** (меняет местами цвет букв и фона).

Внешние инструменты для слияния и индикации изменений

Хотя в систему Git встроена прекрасно вам знакомая команда `diff`, вместо нее можно воспользоваться внешней программой. Кроме того, существует возможность не разрешать конфликты слияния вручную, а прибегнуть к специальной программе

с графическим интерфейсом. Далее описан процесс настройки служебной программы Perforce Visual Merge Tool (P4Merge), которая поможет вам с просмотром добавленных изменений и разрешением конфликтов слияния. Она обладает прекрасным графическим интерфейсом и к тому же распространяется бесплатно.

Существуют версии программы P4Merge для всех основных платформ, так что вы в любой момент можете ею воспользоваться. В представленных далее примерах фигурируют пути к файлам, принятые в системах Mac и Linux; для операционной системы Windows нужно заменить путь `/usr/local/bin` тем путем к исполняемому файлам, который используется в вашей среде.

Первым делом следует скачать программу P4Merge со страницы <http://www.perforce.com/downloads/Perforce/>. Затем мы настроим внешние сценарии-оболочки для исполнения наших команд. Здесь показан путь к исполняемому файлам на компьютере Mac; в других операционных системах нужно указать путь к бинарному файлу `p4merge`. Создадим сценарий-оболочку `extMerge` для процедуры слияния, он будет вызывать бинарный файл со всеми переданными аргументами:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

Оболочка для команды `diff` проверяет наличие всех семи аргументов, которые ей должны передать, и отправляет два из них вашему сценарию слияния. По умолчанию Git передает команде `diff` следующие аргументы:

```
путь старый-файл старый-хеш старые-права новый-файл новый-хеш новые-права
```

Так как нам требуются только второй и пятый аргументы, то есть имена старого и нового файлов, воспользуемся сценарием-оболочкой, который передаст только нужные сведения.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Заодно следует убедиться, что наши инструменты представляют собой исполняемые файлы:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Теперь можно сделать так, чтобы конфигурационный файл вызывал наши инструменты разрешения конфликтов слияния и индикации добавленных изменений. Для этого потребуются несколько настроек: параметр `merge.tool` укажет Git, какую стратегию следует использовать, параметр `mergetool.*.cmd` определит способ запуска команды, параметр `mergetool.trustExitCode` даст Git понять, указывает код завершения программы на успешное разрешение конфликта или нет, а параметр `diff.external` объяснит Git, какой командой следует воспользоваться для получения изменений. То есть вы можете задать четыре варианта команды `config`:

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
  'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
$ git config --global mergetool.trustExitCode false
$ git config --global diff.external extDiff
```

Либо можно отредактировать файл `~/.gitconfig`, добавив в него указанные строки:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

После этого можно запустить команду `diff` вот в такой форме:

```
$ git diff 32d1776b1^ 32d1776b1
```

Это приведет не к выводу команды `diff` в командной строке, а к запуску программы P4Merge (рис. 8.1).

Если попытка слияния двух веток оканчивается конфликтом, запустите команду `git mergetool`. В результате откроется программа P4Merge с графическим интерфейсом, в котором вы и разрешите конфликт.

Использование сценариев-оболочек допускает простую замену инструментов слияния и индикации изменений. К примеру, чтобы вместо программ `extDiff` и `extMerge` начала запускаться программа `KDiff3`, достаточно отредактировать файл `extMerge`:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

В результате для просмотра добавленных изменений и разрешения конфликтов слияния Git будет пользоваться программой `KDiff3`.

В Git существуют предустановленные параметры для запуска разных программ разрешения конфликтов слияния, что избавляет вас от необходимости полностью редактировать команду запуска. Вот как посмотреть список всех поддерживаемых программ:

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
  emerge
  gvimdiff
  gvimdiff2
  opendiff
  p4merge
  vimdiff
  vimdiff2
```

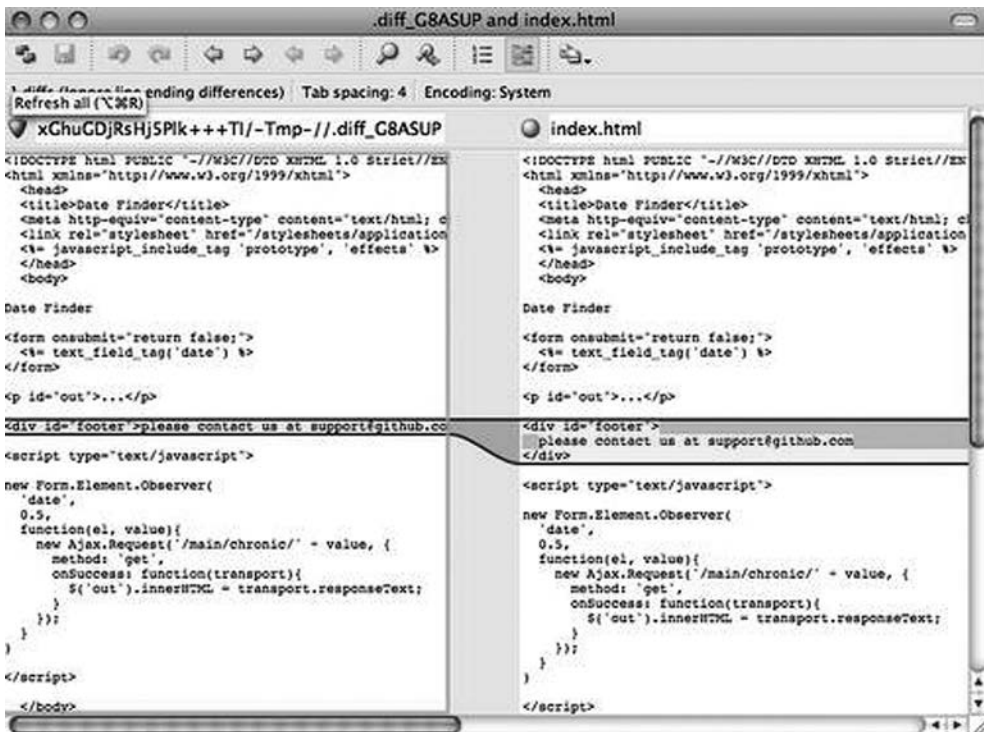


Рис. 8.1. P4Merge

А это инструменты, которые могут применяться, но в настоящее время недоступны:

```
araxis
bc3
codecompare
deltawalker
diffmerge
diffuse
ecmerge
kdiff3
meld
tkdiff
tortoisemerge
xxdiff
```

Некоторые из перечисленных инструментов работают лишь в средах с графическим интерфейсом. В рамках сеанса, ограниченного только терминалом, они работать не смогут.

Если вы предпочитаете использовать программу KDiff3 только для разрешения конфликтов слияния, при этом у вас задан путь, запускающий команду `kdiff3`, можно написать:

```
$ git config --global merge.tool kdiff3
```

Если вместо настройки файлов `extMerge` и `extDiff` вы воспользуетесь этой командой, для разрешения конфликтов слияния Git начнет вызывать программу KDiff3, а за индикацию изменений будет по-прежнему отвечать команда `diff`.

Форматирование и пробелы

Проблемы, связанные с формированием и пробелами, возникающие в процессе совместной работы над проектами, особенно если она ведется на разных платформах, крайне сложно распознаются. Дополнительные пробелы появляются при добавлении исправлений или разных вариантах редактирования, так как тестовые редакторы вставляют их без предупреждения, а при взаимодействии ваших файлов с операционной системой Windows может произойти замена символов конца строки. В системе Git существует ряд конфигурационных параметров, позволяющих облегчить жизнь при столкновении с подобными проблемами.

Параметр `core.autocrlf`

Если вы работаете в операционной системе Windows, а ваши коллеги нет (или наоборот), проблемы, связанные с символами конца строки, неизбежны. Дело в том, что в Windows-файлах для новых строк используется как символ перевода строки, так и символ возврата каретки, в то время как в операционных системах Mac и Linux — только символ перевода строки. Это небольшое, но крайне раздражающее отличие, с которым приходится сталкиваться, работая на разных платформах; многие редакторы в Windows незаметно для пользователя заменяют существующие концы строк типа LF концами строк типа CRLF или вставляют при нажатии пользователем клавиши `Enter` символы обоего типа.

С этой ситуацией Git справляется, автоматически конвертируя окончания строк типа CRLF в тип LF в момент индексирования файла и производя обратное преобразование при выгрузке кода из репозитория в файловую систему. Данную функциональность иницирует параметр `core.autocrlf`. В операционной системе Windows присвоение этому параметру значения `true` преобразует при выгрузке кода концы строк типа LF в тип CRLF:

```
$ git config --global core.autocrlf true
```

Если же вы работаете в операционной системе семейства Linux или Mac, в которой используются окончания строк типа LF, автоматическое преобразование при выгрузке файлов вам не требуется; но если вдруг у вас случайно появится файл с символами конца строки типа CRLF, желательно, чтобы система Git исправила этот недостаток. Можно сделать так, чтобы преобразование CRLF в LF происходило исключительно при фиксации состояний. Для этого нужно присвоить параметру `core.autocrlf` значение `input`:

```
$ git config --global core.autocrlf input
```

После этого у вас будут оставаться концы строк типа CRLF при выгрузке в операционной системе Windows, в то время как в операционных системах семейства Mac и Linux и в репозитории останутся концы строк типа LF.

Если вы работаете в операционной системе Windows над проектом, предназначенным исключительно для Windows, данную функциональность можно отключить и записывать в репозиторий символы возврата каретки. Для этого данной переменной команды **config** нужно присвоить значение **false**:

```
$ git config --global core.autocrlf false
```

Параметр core.whitespace

В Git существует ряд предустановленных параметров для распознавания и исправления некоторых проблем, связанных с пробелами. Система в состоянии обнаружить шесть основных типов проблем: три типа по умолчанию отслеживаются, но слежение можно отключить; слежение за остальными тремя типами по умолчанию отключено, но его можно включить.

По умолчанию работают следующие функции:

- ❑ **blank-at-eol** ищет пробелы в конце строк;
- ❑ **blank-at-eof** ищет пустые строки в конце файла;
- ❑ **space-before-tab** ищет пробелы перед символами табуляции в начале строк.

По умолчанию не работают, но в любой момент могут быть включены следующие функции:

- ❑ **indent-with-non-tab** ищет строки, начинающиеся с пробелов, а не с символов табуляции (количество пробелов задается параметром **tabwidth**);
- ❑ **tab-in-indent** ищет символы табуляции в отступах строк;
- ❑ **cr-at-eol** сообщает системе Git о допустимости символов возврата каретки в конце строк.

Чтобы указать Git, какие из этих функций вы хотите задействовать, присвойте их переменной **core.whitespace**, перечислив через запятую. Для отключения функции ее можно опустить в списке или же поставить перед ее именем знак минус (-). К примеру, если вы хотите задействовать все функции, кроме **cr-at-eol**, можно сделать так:

```
$ git config --global core.whitespace \
trailing-space,space-before-tab,indent-with-non-tab
```

Теперь при запуске команды **git diff** система Git будет распознавать перечисленные проблемы и выделять их цветом, чтобы их можно было решить до фиксации состояния. Это поможет вам и при наложении исправлений командой **git apply**. А можно сделать так, чтобы в момент применения исправления система Git предупреждала вас о наличии перечисленных проблем с пробелами:

```
$ git apply --whitespace=warn <patch>
```

Или можно сделать так, чтобы система Git автоматически пыталась решить проблему перед наложением исправлений:

```
$ git apply --whitespace=fix <patch>
```

Данные параметры применимы и к команде **git rebase**. Если вы зафиксировали состояние, которому присущи проблемы с пробелами, но еще не отправили коммит в репозиторий, запустите команду **git rebase --whitespace=fix**, чтобы система Git автоматически исправила ошибки при перезаписи исправлений.

Настройка сервера

Для настройки серверной части Git-параметров куда меньше, но на некоторые из них имеет смысл обратить внимание.

Параметр `receive.fsckObjects`

Система Git умеет проверять, совпадает ли контрольная сумма присланного на сервер объекта с исходной и указывает ли он на допустимый объект. Однако по умолчанию эта функциональность отключена, так как данные проверки требуют большого объема ресурсов и могут замедлить работу, особенно в случае большого репозитория или большого количества отправляемых на сервер данных. Но если вы хотите, чтобы целостность объектов проверялась при каждой отправке, достаточно присвоить параметру `receive.fsckObjects` значение `true`:

```
$ git config --system receive.fsckObjects true
```

Теперь перед приемом данных от клиента система Git будет проверять целостность вашего репозитория, гарантируя, что неисправный (или вредоносный) клиент не пришлет поврежденные данные.

Параметр `receive.denyNonFastForwards`

Если вы перемещаете уже отправленные на сервер коммиты, а затем пытаетесь отправить их туда еще раз, другими словами, хотите отправить результат фиксации в удаленную ветку, не имеющую коммита, на который она в данный момент указывает, вы столкнетесь с отказом. В общем случае это совершенно правильная стратегия; но если вы сознательно, полностью понимая, что делаете, перемещаете коммиты, можно принудительно обновить удаленную ветку, добавив к команде `push` флаг `-f`.

Чтобы отключить режим принудительного обновления веток, задайте переменную `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```


Это можно сделать и с помощью работающих на стороне сервера хуков `receive`, о которых мы подробно поговорим позже. Данный подход позволяет делать более сложные вещи, например запрещать коммиты без перемотки определенным группам пользователей.

Переменная `receive.denyDeletes`

Пользователь может обойти запрет, установленный с помощью переменной `denyNonFastForwards`, удалив ветку и снова отправив ее на сервер с новой ссылкой. Чтобы этого избежать, присвойте переменной `receive.denyDeletes` значение `true`:

```
$ git config --system receive.denyDeletes true
```

После этого удаление веток и тегов пользователями станет невозможным. Для ликвидации удаленной ветки вам придется вручную удалять с сервера файлы ссылок. Позже вы узнаете, как сделать это на уровне отдельных пользователей с помощью списков контроля доступа.

Git-атрибуты

Существует группа параметров, которую можно указывать для определенного пути, чтобы система Git применяла указанные настройки только к вложенной папке или к группе файлов. Эти связанные с путями параметры называют Git-атрибутами. Они задаются в файле `.gitattributes` одной из папок (обычно это корневая папка проекта) или в файле `.git/info/attributes`. Последнее делается, когда вы не хотите, чтобы файл с атрибутами включался в коммиты вашего проекта.

Атрибуты позволяют задавать различные стратегии слияния для отдельных файлов или папок проекта. Они же объясняют Git, каким образом осуществляется сравнение нетекстовых файлов, и заставляют фильтровать содержимое перед его выгрузкой или записью в репозиторий. В этом разделе вы познакомитесь с некоторыми атрибутами и примерами их применения.

Бинарные файлы

Атрибуты позволяют указать системе Git, какие файлы являются бинарными (на случай, если по-другому это определить невозможно), и дать инструкции по работе с этими файлами. К примеру, некоторые текстовые файлы могут быть сгенерированы программой и поэтому для них нет смысла рассчитывать изменения, в то время как для других бинарных файлов эта операция необходима. Вы узнаете, как объяснить системе Git, в какую категорию попадает тот или иной файл.

Обнаружение бинарных файлов

Некоторые файлы выглядят как текстовые, но по своему назначению являются бинарными. Например, Xcode-проекты на компьютерах Mac содержат файлы, имена которых заканчиваются на `.pbxproj`. Это набор данных в формате JSON (текстовый формат для JavaScript-данных), содержащий параметры сборки и т. п., который интегрированная среда разработки записывает на диск. С технической точки зрения это текстовый файл (в кодировке UTF-8), но по сути это облегченная база данных — вы не можете объединить ее содержимое, если туда внесут изменения разные пользователи, ничем не помогает и вычисление этих изменений. Этот файл предназначен для машинной обработки, и лучше рассматривать его как бинарный.

Чтобы система Git воспринимала таким образом все файлы с расширением `.pbxproj`, добавьте в файл `.gitattributes` следующую строку:

```
*.pbxproj binary
```

Теперь система Git не будет преобразовывать или корректировать концы строк типа CRLF. Не станет она также вычислять или выводить внесенные в этот файл изменения при запуске в рамках вашего проекта команд `git show` и `git diff`.

Выявление изменений в бинарных файлах

Git-атрибуты позволяют эффективно определять, какие изменения были внесены в бинарные файлы. Для этого нужно научить Git преобразовывать бинарные данные в текстовый формат, допускающий сравнение обычной командой `diff`.

К сожалению, эта замечательная функциональность не слишком известна, поэтому мы рассмотрим несколько примеров. Начнем с того, что покажем, как эта техника решает одну из самых раздражающих проблем: управление версиями документов Microsoft Word. Всем известно, что Word является одним из самых ужасных редакторов, но, как ни странно, все им пользуются. Для управления версиями документов Word можно поместить их в Git-репозиторий и время от времени фиксировать текущее состояние. Но что мы получим в результате? Обычная команда `git diff` даст нам примерно такой результат:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

Чтобы напрямую сравнить две версии, их нужно выгрузить на диск и просканировать вручную. Впрочем, существует еще одна возможность, реализуемая через Git-атрибуты. Поместите в файл `.gitattributes` следующую строку:

```
*.docx diff=word
```

Она информирует систему Git о том, что любой файл, соответствующий этому шаблону (`.docx`), должен использовать при просмотре изменений фильтр «word».

Что это за фильтр? Мы должны его создать сами. Сейчас мы заставим Git пользоваться программой `docx2txt` для преобразования документов Word в читабельные текстовые файлы, к которым потом можно будет без проблем применить команду `diff`.

Первым делом следует скачать и установить программу `docx2txt`. Она доступна по адресу <http://docx2txt.sourceforge.net>. Следуйте инструкциям в файле `INSTALL`, чтобы поместить программу в место, где ее сможет обнаружить наша командная оболочка. Затем мы напишем сценарий-оболочку, который будет преобразовывать данные в понятный системе Git формат. Создайте в какой-нибудь из ваших папок файл `docx2txt` и добавьте туда следующие строки:

```
#!/bin/bash
docx2txt.pl $1 -
```

Не забудьте выполнить для этого файла команду `chmod a+x`. Затем можно настроить Git на использование этого сценария:

```
$ git config diff.word.textconv docx2txt
```

Теперь система Git знает, что когда ей нужно определить разницу между двумя снимками состояния и какой-то из файлов имеет расширение `.docx`, следует пропустить эти файлы через фильтр «word», который определен как программа `docx2txt`. Это фактически преобразует файлы Word в текстовый формат, и только после этого система попытается выявить внесенные изменения.

Рассмотрим пример: я преобразовал главу 1 этой книги в формат Word и зафиксировал в Git-репозитории. Затем добавил еще один абзац текста. Вот что показала мне 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.

Система Git коротко и ясно дала мне понять, что я добавил строку **Testing: 1, 2, 3.**, что и произошло в реальности. То есть все работает, хотя и не идеально, так как изменения в форматировании не выявляются.

Этот способ позволяет решить и другую интересную проблему: сравнение графических файлов. Можно прогнать файлы JPEG через фильтр, извлекающий данные EXIF — метаданные, которые дописываются в большинстве форматов изображений. Скачайте и установите программу `exiftool`, и вы сможете преобразовывать метаданные графических файлов в текст, давая команде `diff` возможность вывести на экран текстовое представление внесенных изменений:

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

Вот результат, который дает команда `git diff` после замены изображения в проекте:

```
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 изменились размер файла и размеры изображения.

Развертывание ключа

Развертывание ключа в стиле SVN или CVS зачастую востребовано разработчиками, привыкшими к данным системам. Реализации этой функциональности в Git мешает отсутствие возможности записывать в файл сведения о коммите после фиксации

состояния, так как Git первым делом считает контрольную сумму файла. Тем не менее текст можно вставлять в файл, когда он выгружен, удаляя эту информацию перед добавлением в коммит. Git-атрибуты позволяют делать это двумя способами.

Во-первых, контрольную сумму SHA-1 массива двоичных данных можно автоматически вставлять в поле `$Id$`. Если установить для файла или набора файлов данный атрибут, при следующей выгрузке данных из ветки система Git заместит это поле значением SHA-1 массива двоичных данных. Имейте в виду, что это — контрольная сумма не коммита, а массива:

```
$ echo '*.txt ident' >> .gitattributes
$ echo '$Id$' > test.txt
```

При следующей выгрузке этого файла Git вставит SHA массива двоичных данных:

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Впрочем, применимость этого подхода ограничена. При развертывании ключа в CVS или Subversion можно добавлять метку данных — это несет куда больше информации, чем SHA. Ведь контрольная сумма формируется случайным образом, и по виду двух сумм невозможно сказать, какая из них принадлежит более новому коммиту.

Однако можно написать собственные фильтры, которые будут делать подстановки при фиксации состояния/выгрузке файлов. Эти фильтры называются **clean** (рис. 8.2) и **smudge** (рис. 8.3). В файле `.gitattributes` указывается путь для фильтра и настраиваются сценарии, которые будут обрабатывать файлы перед выгрузкой (фильтр **smudge**) и перед индексацией (фильтр **clean**). Данные фильтры можно настроить на выполнение любых действий.

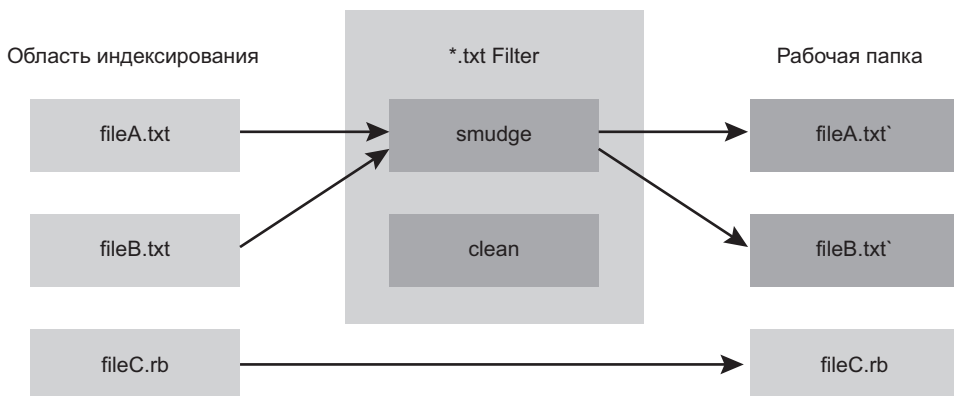


Рис. 8.2. Фильтр **smudge** срабатывает перед выгрузкой

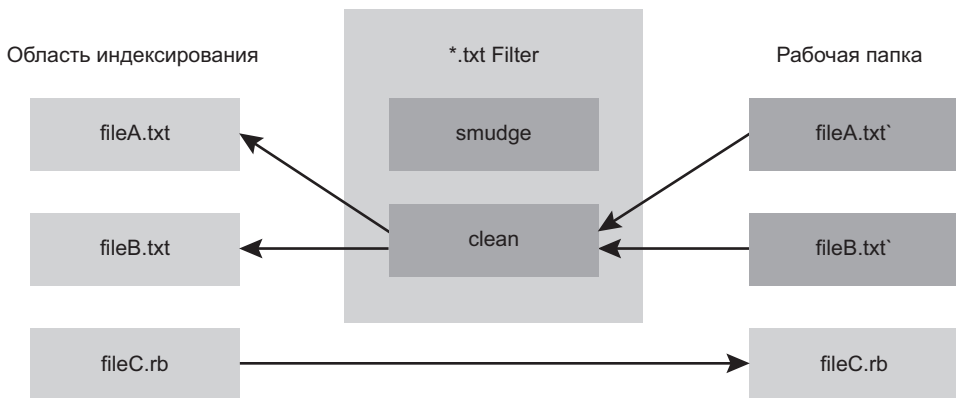


Рис. 8.3. Фильтр `clean` срабатывает перед индексацией

В исходном сообщении фиксации коммита, добавляющего эту функциональность, дан простой пример того, как код на языке C перед фиксацией пропускается через программу `indent`. Для этого нужно задать атрибут `filter` в файле `.gitattributes` таким образом, чтобы он пропускал файлы `*.c` через фильтр `indent`:

```
*.c filter=indent
```

Теперь объясним системе Git, что фильтр `indent` должен делать с фильтрами `smudge` и `clean`:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

В данном случае при фиксации состояния файлов, которые подходят под шаблон `*.c`, система Git пропустит их через программу `indent` перед индексированием, а затем, перед тем как выгрузить их обратно на диск, — через программу `cat`. Программа `cat`, по сути, ничего не делает: она передает те же данные, которые получила. Такая комбинация фильтров эффективно пропускает все файлы с кодом на языке C через программу `indent` до выполнения коммита.

Другим интересным примером является развертывание ключа `$Date$`, как это делается в системе RCS. Для корректной реализации нам потребуется небольшой сценарий, который берет имя файла, определяет дату последней фиксации его состояния в рамках проекта и вставляет эту дату в наш файл. Вот маленький сценарий на языке Ruby, который все это делает:

```
#!/usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Сценарий получает от команды `git log` дату последнего коммита, вставляет ее во все строки `$Date$`, обнаруженные в стандартном потоке ввода, и выводит результат. Подобную последовательность действий легко реализовать на любом языке.

Присвоим этому файлу имя `expand_date` и вставим в наш путь. Теперь нужно настроить фильтр (назовем его `dater`) и указать, что он должен применяться при проходе файлов через фильтр `smudge` при выгрузке. Чтобы убраться эти изменения из коммита, воспользуемся выражением на языке Perl:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\\$Date[^\\\\$]*\\\\$Date\\\\$/'"
```

Этот фрагмент кода на языке Perl вырезает все обнаруженное в строке `$Date$`, возвращая нас в исходное состояние. Итак, наш фильтр готов и его можно протестировать, создав файл с ключевым словом `$Date$` и настроив Git-атрибут, активизирующий для этого файла новый фильтр:

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

Зафиксировав эти изменения и снова выгрузив файл, вы обнаружите, что ключевое слово корректно замещено:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Как видите, у вас есть мощная техника настройки приложений под свои нужды. Но применять ее следует аккуратно, так как файл `.gitattributes` добавляется в коммит и распространяется с проектом, а его основной рабочий элемент (в данном случае — фильтр `dater`) нет, поэтому работать данная техника будет не везде. Протестировать фильтры следует с таким расчетом, чтобы при их отказе проект мог продолжить корректное функционирование.

Экспорт репозитория

Git-атрибуты позволяют делать интересные вещи, связанные с экспортом архива проекта.

Атрибут `export-ignore`

Можно сделать так, чтобы система Git при создании архива не экспортировала определенные файлы и папки. Файлы и папки, которые вы не хотите видеть в архиве, но которые должны присутствовать в проекте, задаются посредством атрибута `export-ignore`.

Предположим, в папке `test/` находятся некие тестовые файлы, которые при экспорте проекта нет никакого смысла добавлять в архив. Добавим в файл с Git-атрибутами следующую строку:

```
test/ export-ignore
```

Теперь команда `git archive`, создающая архив `tar` с вашим проектом, не будет включать в архив указанную папку.

Атрибут `export-subst`

Еще в архивах можно выполнять простую замену ключевых слов. Система Git позволяет добавить в любой файл строку `$Format:$` с любым коротким кодом форматирования `--pretty=format`. Варианты этого кода вы видели в главе 2. Предположим, вы хотите добавить в проект файл `LAST_COMMIT`, в который при запуске команды `git archive` будет автоматически вставляться дата последнего коммита. Это делается следующим образом:

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

После выполнения команды `git archive` содержимое файла для открывших архив пользователей будет выглядеть так:

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

Стратегии слияния

Еще Git-атрибуты позволяют выбирать для файлов проекта различные стратегии слияния. Крайне полезна возможность сделать так, чтобы система Git в случае конфликтов слияния в определенных файлах просто выбирала ваш вариант файла, предпочитая его чужому.

Это имеет смысл делать, например, в ситуации, когда ветка вашего проекта разошлась с исходной или стала специализированной, но вам требуется возможность сливать из нее изменения, игнорируя при этом некоторые файлы. Предположим, у вас есть файл с настройками базы данных `database.xml`. Две ветки содержат разные модификации этого файла, а вы хотите, не затрагивая его, слить данные в чужую ветку. Настроим атрибут следующим образом:

```
database.xml merge=ours
```

При слиянии в другую ветку вместо конфликта для файла `database.xml` вы получите следующий вариант:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

В данном случае файл `database.xml` остается в том же состоянии, в каком он был изначально.

Git-хуки

Подобно другим системам контроля версий, Git в ответ на различные действия позволяет запускать ваши собственные сценарии. Существует две группы хуков: работающие на стороне сервера и работающие на стороне клиента. Последние активизируются такими операциями, как фиксация и слияние, в то время как на стороне сервера триггерами служат сетевые операции, например прием коммитов. Хуки применяются для решения самых разных задач.

Установка хука

Все хуки хранятся в папке **hooks**, вложенной в папку с системой Git. В большинстве проектов это папка **.git/hooks**. При инициализации нового репозитория командой **git init** Git заполняет эту папку примерами сценариев, многие из которых полезны сами по себе; кроме того, там перечислены входные значения для каждого сценария. Все примеры представляют собой сценарии-оболочки с добавками на языке Perl, но работать в данной ситуации будет любой корректно именованный исполняемый сценарий — вы можете писать их на языке Ruby или Python или любом другом. Чтобы воспользоваться этими хуками, их следует переименовать, поскольку все имена файлов с хуками имеют окончание **sample**.

Для активизации хука поместите корректно названный исполняемый файл в подпапку **hooks** папки с системой Git. С этого момента его можно будет вызывать. Основные имена хуков мы рассмотрим в этом разделе.

Хуки на стороне клиента

Существует множество хуков, работающих на стороне клиента. Мы поделим их на хуки, связанные с коммитами, хуки для работы электронной почты и все прочие.

ПРИМЕЧАНИЕ

Следует помнить, что хуки, работающие на стороне клиента, не копируются при клонировании репозитория. Если вы хотите обеспечить с помощью хуков соблюдение некоего правила, лучше делать это на стороне сервера.

Хуки для работы с коммитами

Первые четыре хука связаны с процессом фиксации состояний.

Первым (еще до ввода сообщения фиксации) срабатывает хук **pre-commit**. Он инспектирует снимок подготовленного к фиксации состояния, проверяя, не забыто

ли что, проведено ли тестирование, следует ли обратить внимание на какие-то фрагменты кода. Ненулевой результат работы этого хука прерывает создание коммита, хотя этого можно избежать при помощи команды `git commit --no-verify`. Вы можете, к примеру, проверить стиль кода (воспользовавшись анализатором `lint` или подобной ему программой), наличие символов пробела в конце строк (по умолчанию хук занимается именно этим) или наличие корректной документации для новых методов.

Хук `prepare-commit-msg` запускается до того, как откроется редактор с сообщением фиксации, но после того, как появится версия этого сообщения, заданная по умолчанию. Он позволяет отредактировать данное сообщение перед тем, как оно станет доступно автору коммита. Этот хук принимает несколько параметров: путь к файлу, в котором в настоящее время хранится сообщение фиксации, тип коммита и его контрольная сумма SHA-1, если это коммит, в который вносится правка при помощи параметра `--amend`. Для обычных коммитов этот хук, как правило, не применяется. Он требуется в основном коммитам с автоматически генерируемыми сообщениями: например, при шаблонных сообщениях фиксации, коммитах слияния, объединенных коммитах и коммитах, подвергшихся исправлениям. Он может использоваться совместно с шаблоном коммита, реализуя программное добавление туда информации.

У хука `commit-msg` параметр всего один — путь к временному файлу с написанным разработчиком сообщением фиксации. Если этот сценарий завершает свою работу с ненулевым кодом, Git прерывает процесс фиксации. Этим можно воспользоваться для проверки состояния проекта или сообщения фиксации. В последнем разделе этой главы вы увидите, каким образом хук `commit-msg` позволяет проверить, соответствует ли сообщение фиксации требуемому шаблону.

После завершения процесса фиксации запускается хук `post-commit`. Параметров у него нет, но вы легко можете получить последний коммит командой `git log -1 HEAD`. Как правило, этот сценарий применяется для разного рода уведомлений.

Хуки для работы с электронной почтой

Для работ, связанных с электронной почтой, можно настроить три хука на стороне сервера. Все они вызываются командой `git am`, поэтому если вы этой командой не пользуетесь, можете смело переходить к следующему разделу. Если же вы принимаете по электронной почте исправления, созданные командой `git format-patch`, эти хуки могут вам пригодиться.

Первым запускается хук `applypatch-msg`. Он принимает единственный аргумент: имя временного файла, содержащего предложенное сообщение фиксации. Если этот сценарий завершается ненулевым кодом, Git прерывает применение исправления. Этим можно воспользоваться, чтобы гарантировать корректность

форматирования сообщения фиксации или стандартизировать его средствами самого сценария.

Следующий хук **pre-applypatch** запускается во время применения исправлений командой **git am**. По непонятной причине он активизируется после применения исправления, но до его фиксации, поэтому им можно воспользоваться для проверки снимка состояния перед созданием коммита. Средствами этого сценария можно провести тестирование или другим способом исследовать рабочее дерево папок. При нехватке каких-либо элементов или завершившихся неудачей тестах хук завершает свою работу с ненулевым кодом, прерывая работу команды **git am** и не давая зафиксировать исправления.

Последний хук, запускаемый во время работы команды **git am**, называется **post-applypatch**. Он активизируется после создания коммита. Им можно пользоваться для уведомления группы или автора скачанного вами исправления, что вы его применили. Этот сценарий уже не может остановить процесс применения исправления.

Другие клиентские хуки

Хук **pre-rebase** запускается перед перемещением данных и может остановить процесс, завершившись с ненулевым кодом. С его помощью можно запретить перемещение уже отправленных на сервер коммитов, хотя данный механизм подразумевает ряд допущений, которые могут оказаться несовместимыми с вашим рабочим процессом.

Хук **post-rewrite** запускается командами, которые отвечают за замену коммитов, например **git commit --amend** и **git rebase** (но в эту группу не попадает команда **git filter-branch**). Его единственным аргументом является команда, вызывающая перезапись, при этом он получает список перезаписанного при стандартном вводе. Применяется этот хук в основном теми же способами, что и хуки **post-checkout** и **post-merge**.

После успешного выполнения команды **git checkout** запускается хук **post-checkout**; вы можете воспользоваться им для корректной настройки своей рабочей папки под среду проекта. Настройка может состоять, например, в перемещении больших бинарных файлов, для которых не требуется управление версиями, в автоматической генерации документации и прочих действиях подобного плана.

Хук **post-merge** запускается после успешного слияния. Он позволяет восстанавливать в рабочем дереве данные, отследить которые Git не в состоянии, например сведения о правах доступа. Аналогичным образом этот хук проверяет наличие внешних по отношению к Git файлов, которые иногда требуется скопировать при изменении рабочего дерева.

Хук `pre-push` запускается вместе с командой `git push` после обновления удаленных ссылок, но до передачи каких-либо объектов. В качестве параметров он принимает имя и местоположение удаленного сервера, а также список ссылок, которые требуется обновить в процессе стандартного ввода. Этот хук позволяет проверить набор обновлений ссылок до отправки данных на сервер (ненулевой код выхода прерывает выполнение команды `push`).

Время от времени Git выполняет сборку мусора в рамках своей обычной жизнедеятельности, вызывая для этого команду `git gc --auto`. Перед сборкой мусора вызывается хук `pre-auto-gc`, который может применяться для уведомления о начале процесса сборки или для его прерывания.

Хуки на стороне сервера

В дополнение к хукам на стороне клиента вы, как системный администратор, можете применить пару важных хуков на стороне сервера, принудительно реализуя в рамках своего проекта практически любую политику. Данные сценарии запускаются до и после отправки данных на сервер. Хуки, активирующиеся до отправки, в любой момент могут завершить свою работу с ненулевым кодом и прервать процесс передачи данных, отправив клиенту сообщение об ошибке. Вы можете установить сколь угодно сложные правила приема данных.

Хук `pre-receive`

Первый сценарий запускается при обработке отправленных клиентом данных и называется `pre-receive`. Он принимает отправленные через стандартный ввод ссылки; при завершении сценария с ненулевым кодом ни одна из этих ссылок не принимается. Этот хук позволяет, например, удостовериться, что все обновленные ссылки связаны с перемоткой, или выполнить процедуру контроля доступа для всех ссылок и файлов, редактируемых в процессе приема новой информации.

Хук `update`

Сценарии `update` и `pre-receive` очень похожи, просто первый выполняется для каждой ветки, которую отправитель пытается обновить. При попытке обновить одновременно несколько веток хук `pre-receive` запускается всего один раз, в то время как `update` будет вызываться для каждой обновляемой ветки отдельно. Вместо чтения из стандартного ввода сценарий принимает три аргумента: имя ссылки (ветки), контрольную сумму SHA-1 коммита, на который ссылка указывала перед отправкой, и SHA-1 коммита, который пользователь пытается передать на сервер. Если сценарий `update` завершает работу с ненулевым кодом, отклоняется только одна ссылка, остальные допускают обновление.

Хук `post-receive`

Хук `post-receive` запускается после завершения процесса приема данных и применяется для обновления других служб и уведомления пользователей. Он принимает из стандартного ввода те же самые данные, что и хук `pre-receive`. Он используется, например, при отправке писем в рассылку, уведомлении сервера непрерывной интеграции или обновлении в системе отслеживания ошибок — вы можете даже анализировать сообщения фиксации, выясняя, не нужно ли открыть, отредактировать или закрыть какую-либо заявку. Процесс приема данных этот сценарий останавливать не умеет, а так как клиент не разрывает соединение до завершения процесса, будьте осторожны с действиями, выполнение которых может занять длительное время.

Пример принудительного внедрения политики

Сейчас мы воспользуемся ранее полученными знаниями для организации рабочего процесса, который будет проверять формат сообщения фиксации и допускать к редактированию определенных подпапок проекта только некоторых пользователей. Мы напишем сценарии, работающие на стороне клиента, которые дадут разработчику понять, будут ли приняты отправляемые им данные, и сценарии, работающие на стороне сервера и принудительно внедряющие заданные нами правила.

Все примеры сценариев в этом разделе написаны на языке Ruby; отчасти потому, что мы привыкли им пользоваться, а отчасти потому, что Ruby-код достаточно прост и понятен даже тем, кто им не пользуется. Впрочем, в данном случае сгодится и любой другой язык — все распространяющиеся с системой Git сценарии хуков написаны на языке Perl или Bash, так что вы можете найти множество самых разных примеров.

Хук на стороне сервера

Все действия на стороне сервера мы запрограммируем в файле `update`, находящемся в вашей папке `hooks`. Хук `update` запускается для каждой отправляемой на сервер ветки и принимает три аргумента:

- ☐ ссылку на ветку, в которую отправляются данные;
- ☐ старую версию этой же ветки;
- ☐ новую отправляемую версию данных.

Кроме того, в случае передачи данных по протоколу SSH вам будет доступно имя пользователя, осуществляющего отправку. Если вы позволяете всем подключаться

под одним и тем же именем пользователя (например, «git») с аутентификацией по открытому ключу, может потребоваться предоставить этому пользователю командную оболочку, которая будет по ключу определять, кто именно выполнил подключение, и соответствующим образом задавать переменную окружения. Предположим, что информация о пользователе содержится в переменной `$USER` и наш сценарий `update` начинается со сбора всех необходимых сведений:

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$soldrev = ARGV[1]
$newrev = ARGV[2]
$user = ENV['USER']

puts "Enforcing Policies..."
puts "({$refname}) ({$soldrev[0,6]}) ({$newrev[0,6]}")"
```

В данном случае используются глобальные переменные, потому что так получается нагляднее.

Формат сообщения фиксации

Первым делом следует обеспечить определенный формат всех сообщений фиксации. Например, пусть в каждом сообщении будет строка вида `ref: 1234`, чтобы каждый коммит был связан с рабочим элементом в системе отслеживания ошибок. Мы должны изучать все присылаемые коммиты, проверять наличие в сообщении фиксации указанной строки, а при ее отсутствии хотя бы в одном коммите завершать работу сценария ненулевым кодом и отказывать в приеме данных.

Чтобы получить список контрольных сумм SHA-1 всех присланных коммитов, передайте служебной команде `git rev-list` значения `$newrev` и `$oldrev`. Фактически это команда `git log`, которая по умолчанию выводит только значения SHA-1. Вот как получить перечень всех контрольных сумм коммитов, появившихся в промежутке между двумя коммитами:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Теперь можно взять выводимые этой команды данные, циклически просмотреть все контрольные суммы SHA коммитов, извлекая их сообщения фиксации и сравнивая с шаблоном при помощи регулярного выражения.

Осталось только понять, каким образом мы будем извлекать сообщения фиксации. Для получения исходных данных коммита применяется еще одна служебная

команда `git cat-file`. Подробно служебные команды рассматриваются далее, а пока посмотрим, что мы получим в данном случае:

```
$ git cat-file commit ca82a6
tree cfd3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Проще всего узнать сообщение фиксации, если известна контрольная сумма SHA-1 коммита, найдя в выводе команды пустую строку и скопировав то, что находится после нее. В операционных системах семейства Unix это можно делать командой `sed`:

```
$ git cat-file commit ca82a6 | sed '1,/^\$/d'

changed the version number
```

Данная схема позволяет извлекать сообщения фиксации из всех присылаемых коммитов, прерывая выполнение сценария в случае несовпадения с шаблоном. Для отказа в приеме присланных данных нам требуется ненулевой код при завершении работы сценария. Целиком метод выглядит так:

```
$regex = /\[ref: (\d+)\]/

# заставляем использовать конкретный формат сообщений фиксации
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] Некорректное форматирование сообщения"
      exit 1
    end
  end
end
end
check_message_format
```

Поместив этот код в сценарий `update`, мы запретим обновления, содержащие коммиты с сообщениями фиксации, отформатированными не по нашим правилам.

Система контроля доступа пользователей

Теперь мы добавим механизм, в котором применяются списки контроля доступа (Access Control List, ACL), указывающие, какие пользователи могут присылать обновления к различным частям проектов. У некоторых будет полный доступ на запись, другие же смогут редактировать содержимое только определенных подпапок или вообще только конкретные файлы. Запишем наши правила в файл `acl`, который находится в голом Git-репозитории на сервере. Нужно будет сделать так, чтобы

хук `update` брал эти правила и определял, к каким файлам относятся присланные коммиты и имеет ли отправитель право на изменение этих файлов.

Первым делом мы создадим список контроля доступа. В данном случае будет использоваться формат, напоминающий механизм ACL в системе CVS. Он предполагает наличие набора строк, в котором первое поле имеет значение `avail` или `unavail` (есть или нет доступ), следующее поле содержит разделенные запятыми имена пользователей, которых касается данное правило, а последнее поле задает путь применения правила (пустое поле соответствует открытому доступу). Все поля разделяются вертикальной чертой (`|`).

В нашем случае нужно внести в список пару администраторов, лиц, ответственных за написание документации (с доступом к папке `doc`), и одного разработчика (с доступом к папкам `lib` и `tests`). Наш ACL-файл должен выглядеть следующим образом:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Теперь нужно передать эту информацию в пригодную для работы структуру. Для простоты мы будем требовать соблюдения только директив `avail`. Здесь показан метод, дающий нам ассоциативный массив, ключом в котором является имя пользователя, а значением — массив путей к местам, куда этот пользователь может записывать информацию:

```
def get_acl_access_data(acl_file)
  # чтение данных ACL
  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 ACL-файла, который мы создали ранее, метод `get_acl_access_data` вернет вот такую структуру данных:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```


Итак, мы произвели сортировку прав доступа. Теперь нужно понять, где располагаются файлы, редактируемые присланными коммитами, и убедиться, что отправитель коммитов имеет право сделать туда запись.

Определить, какие файлы были изменены внутри одного коммита, позволяет добавление к команде `git log` параметра `--name-only` (об этом коротко упоминалось в главе 2):

```
$ git log -1 --name-only --pretty=format:'' 9f585d
```

```
README
lib/test.rb
```

Сопоставив ACL-структуру, которую нам вернул метод `get_acl_access_data`, со списком файлов внутри каждого коммита, вы определите, имеет ли пользователь право на отправку своих коммитов:

```
# только определенным пользователям разрешается
# редактировать определенные подпапки проекта
def check_directory_perms
  access = get_acl_access_data('acl')
  # проверяем, не пытается ли прислать данные тот, у кого нет на это права
  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 # полный доступ у пользователя
          || (path.start_with? access_path) # доступ к этой папке
          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
```

Команда `git rev-list` предоставляет список новых коммитов, присланных на сервер. Мы определяем, какие файлы меняет каждый из них, и проверяем наличие у приславшего коммиты пользователя права на запись в эти файлы.

Теперь работающие с нами пользователи не смогут присылать коммиты с некорректно отформатированными сообщениями фиксации и будут редактировать только те файлы, которые мы позволяем им менять.

Тестирование

Если выполнить команду `chmod u+x .git/hooks/update`, запустив файл, в который мы поместили весь показанный в предыдущем разделе код, и попытаться отправить коммит с некорректно отформатированным сообщением фиксации, результат должен быть примерно таким:

```
$ 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'
```

В данном случае нужно обратить внимание на пару моментов. Во-первых, при запуске хука мы видим следующее:

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Напоминаем, что мы выводили это на экран в самом начале сценария `update`. Все, что сценарий отправляет в стандартный поток вывода, будет передано клиенту.

Затем мы видим 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 наша, а в остальных двух система Git сообщает, что сценарий `update` завершил работу с ненулевым кодом и поэтому сервер отказывается принять пришедшие коммиты. Наконец, мы видим следующее:

```
To git@gitserver:project.git
! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Сообщение `remote rejected` будет появляться для каждой отклоненной хуком ссылки. Оно информирует, что отказ вызван сбоем хука.

Более того, аналогичное сообщение получит пользователь, попытавшийся, отправив коммит, отредактировать файл, к которому у него нет доступа. Например, если человек, отвечающий за документацию, попытается отправить коммит, редактирующий что-то в папке `lib`, он увидит следующее:

```
[POLICY] You do not have access to push to lib/test.rb
```

С этого момента, пока сценарий `update` находится на своем месте и является исполняемым, в ваш репозиторий не попадут коммиты, сообщения фиксации которых не соответствует заданному шаблону, и пользователи смогут работать только в указанных вами пределах.

Хуки на стороне клиента

Следствием рассмотренного подхода являются многочисленные жалобы, которые вам неизбежно начнут поступать после отказа в приеме коммитов. Когда их старательно выполненную работу в последний момент отвергают, пользователи расстраиваются; кроме того, в этой ситуации им приходится редактировать историю своих наработок, а это занятие не для слабых духом.

Поэтому имеет смысл создавать хуки, работающие на стороне клиента и уведомляющие пользователей о потенциально недопустимых действиях. Это даст возможность внести коррективы перед фиксацией состояния, когда решение проблемы еще не требует больших усилий.

Так как при клонировании проекта хуки не копируются, их следует передавать каким-то другим способом, попутно заставляя пользователей поместить их в папку `.git/hooks` и сделать исполняемыми. Распространять хуки можно как в рамках существующего проекта, так и создав для них отдельный проект, но настраивать их автоматически система Git не будет.

Для начала перед записью каждого коммита следует проверять корректность сообщения фиксации, чтобы сервер не отклонял исправления по этой причине. Для этого можно добавить хук `commit-msg`. Если сделать так, чтобы хук читал сообщение из файла, переданного ему в качестве первого аргумента, и сравнивал его с шаблоном, при обнаружении несовпадения можно заставить систему Git прервать фиксацию состояния:

```
#!/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 сценарий находится в нужной папке (`.git/hooks/commit-msg`) и является исполняемым, при попытке создания коммита с некорректно отформатированным сообщением фиксации вы получите следующее:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

Как видите, коммит в данном случае не создается. Если же сообщение соответствует шаблону, система Git позволяет зафиксировать состояние:

```
$ git commit -am 'test [ref: 132]'  
[master e05c914] test [ref: 132]  
1 file changed, 1 insertions(+), 0 deletions(-)
```

Теперь нужно убедиться, что пользователь не пытается отредактировать файлы, права на запись в которые у него нет. Если папка `.git` вашего проекта содержит копию ранее использовавшегося ACL-файла, показанный здесь сценарий `pre-commit` принудительно введет указанное ограничение:

```
#!/usr/bin/env ruby  
  
$user = ENV['USER']  
  
# [ вставьте код метода acl_access_data ]  
  
# только определенным пользователям разрешается  
# редактировать определенные подпапки проекта  
def check_directory_perms  
  access = get_acl_access_data('.git/acl')  
  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")  
  files_modified.each do |path|  
    next if path.size == 0  
    has_file_access = false  
    access[$user].each do |access_path|  
      if !access_path || (path.index(access_path) == 0)  
        has_file_access = true  
      end  
    end  
    if !has_file_access  
      puts "[POLICY] You do not have access to push to #{path}"  
      exit 1  
    end  
  end  
end  
  
check_directory_perms
```

Это почти тот же самый сценарий, что и на стороне сервера, но с двумя существенными отличиями. Во-первых, в другом месте располагается ACL-файл, так как сценарий запускается из вашей рабочей папки, а не из папки `.git`. То есть мы поменяли путь к этому файлу. Раньше это был такой путь:

```
access = get_acl_access_data('acl')
```

А теперь он выглядит следующим образом:

```
access = get_acl_access_data('.git/acl')
```

Отличается и способ получения списка отредактированных файлов. Так как метод, работающий на стороне сервера, сверяется по журналу коммитов, а в настоящее

время информации о данном коммите там пока нет, список берется из области индексирования. Таким образом, ранее мы использовали команду:

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

А теперь требуется следующая команда:

```
files_modified = `git diff-index --cached --name-only HEAD`
```

Но за исключением этих двух отличий сценарий работает абсолютно так же. При этом он полагает, что вы запускаете его локально от имени того же пользователя, от имени которого отправляете изменения на сервер. В противном случае нужно вручную задать значение переменной `$user`.

Ну и последнее, что мы можем сделать, — это удостовериться, что пользователь отправляет только ссылки, связанные с коммитами перемотки. Для получения ссылок, не допускающих перемотки, следует переместить последний отправленный коммит или попытаться отправить в ту же удаленную ветку содержимое другой локальной ветки.

Предполагается, что на сервере уже установлены переменные `receive.denyDeletes` и `receive.denyNonFastForwards`, обеспечивающие выполнение этого требования, поэтому единственным случайным действием, которое вы можете попытаться перехватить, является перемещение уже отправленных на сервер коммитов.

Вот пример проверяющего данный аспект сценария `pre-rebase`. Он получает список всех коммитов, которые вы собираетесь переписать, и проверяет, существуют ли они в виде удаленных ссылок. Обнаружив коммит, достижимый из какой-либо удаленной ветки, хук прерывает перемещение:

```
#!/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
```

Обратите внимание, что здесь используется синтаксис, который мы не рассматривали в разделе «Выбор версии» главы 7. Список отправленных на сервер коммитов мы получаем следующей командой:

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

Запись `SHA^@` соответствует всем предкам указанного коммита. Мы ищем коммит, достижимый из последнего коммита в удаленной ветке, но при этом недостижимый со стороны предков коммитов, которые вы пытаетесь отправить на сервер. Все это указывает на перемтку.

Основным недостатком такого подхода является медленное выполнение, а зачастую и избыточность проверки. Ведь если вы не пытаетесь принудительно отправлять данные на сервер с параметром `-f`, сервер сообщит вам о недопустимости выполняемой операции и прервет ее. Но это интересное упражнение, которое теоретически в состоянии помочь вам избежать перемещения и исправления его последствий в дальнейшем.

Заключение

Мы рассмотрели большинство основных средств настройки клиентской и серверной частей системы Git, позволяющих максимально приспособить ее под ваши нужды. Вы изучили все виды конфигурационных параметров, атрибутов файлов и даже хуков, а также познакомились с примером задания политики работы сервера. Теперь вы сможете настроить систему Git под любой тип рабочего процесса.

9 Git и другие системы контроля версий

Этот мир далек от совершенства. Возможность сразу же использовать Git в любом вашем проекте предоставляется далеко не всегда. Иногда приходится пользоваться другой VCS, только мечтая о Git. Поэтому первую часть этой главы мы посвятим изучению способов применения системы Git в качестве клиента при работе с проектами, реализуемыми в другой системе.

Возможно, в какой-то момент у вас появится возможность полностью перейти на Git. Поэтому во второй части главы мы расскажем о способах перехода на Git с различных систем. Более того, вы узнаете, как написать собственный метод на случай отсутствия встроенных инструментов импорта.

Git в качестве клиента

Система Git производит на разработчиков настолько хорошее впечатление, что многие ищут способы использовать ее на своей машине, в то время как остальная часть группы работает с совершенно другой VCS. Существует множество устройств сопряжения, которые называются «мостами». Здесь мы рассмотрим варианты, с которыми вам, скорее всего, придется столкнуться в процессе работы.

Git и Subversion

В огромном количестве проектов с открытым исходным кодом и в корпоративных проектах для управления версиями кода используется система Subversion. Эта система существует более десяти лет и фактически именно она в большинстве случаев выбиралась для проектов с открытым исходным кодом. Кроме того, она во многом похожа на систему CVS, считающуюся прародителем всех современных систем контроля версий.

Одним из замечательных инструментов системы Git является двусторонний мост, связывающий ее с Subversion. Он называется **git svn**. Именно этот инструмент превращает Git в полноценный клиент для Subversion-сервера, позволяющий пользоваться функциональностью Git с отправкой результатов на Subversion-сервер, если локально вы применяете именно систему Subversion. Это означает, что вам будут доступны операции локального создания веток и слияния, область индексирования, перемещения, отбор лучшего и многое другое, в то время как ваши коллеги продолжают работать дедовскими методами. Это хороший способ ввести систему Git в корпоративную среду и предложить ее коллегам как вариант повышения эффективности работы, постепенно добиваясь изменения инфраструктуры в сторону полной поддержки Git. Мост для Subversion может стать тропинкой в мир DVCS.

Команда git svn

В системе Git основой всех команд для совместной работы с Subversion служит **git svn**. Она используется с рядом дополнительных команд, наиболее распространенные из которых будут описаны далее.

Важно понимать, что каждое применение команды **git svn** означает взаимодействие с системой Subversion, функционирующей совсем не так, как Git. Разумеется, вам будет доступно локальное создание веток и слияние, но в общем случае лучше пользоваться процедурой перемещения, сохраняя историю по возможности линейной и избегая одновременного взаимодействия с удаленным репозиторием Git.

Не переписывайте уже отправленную на сервер историю. Не отправляйте ничего в параллельный Git-репозиторий для взаимодействия с коллегами, которые занимаются разработкой с помощью Git. В Subversion допустима только одна линейная история, и туда очень легко внести путаницу. Если в вашей рабочей группе часть сотрудников пользуется SVN, а часть предпочитает Git, организуйте совместную работу через SVN — это сильно облегчит вам жизнь.

Настройка

Для знакомства с данной функциональностью потребуется обычный SVN-репозиторий с правом на запись. Тем, кто хочет повторить приводимые далее примеры, потребуется копия моего тестового репозитория. Ее легко можно получить при помощи встроенного в Subversion инструмента **svnsync**. Для своих тестов мы

создали новый репозиторий Subversion в службе Google Code. Этот репозиторий представляет собой частичную копию проекта **protobuf**, который является инструментом, шифрующим структурированные данные для их передачи по сети.

Итак, если вы хотите воспроизводить все наши действия, создайте локальный репозиторий Subversion:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Затем разрешите всем пользователям вносить изменения в **revprops**. Для этого проще всего добавить сценарий **pre-revprop-change**, всегда завершающий работу с нулевым кодом:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

На этом этапе можно синхронизировать этот проект со своим локальным компьютером, вызвав команду **svnsync init** и указав для нее исходный и целевой репозитории:

```
$ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
```

Это настроит свойства необходимым для синхронизации образом. Теперь выполним клонирование кода:

```
$ 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.
[...]
```

Эта операция занимает всего несколько минут. Но если попытаться скопировать исходный репозиторий не в локальный, а в другой удаленный репозиторий, процесс может занять почти час, хотя копированию подвергаются менее 100 коммитов. Система Subversion сначала копирует каждый файл, затем пересылает его в другой репозиторий. Это до смешного нерационально, но другого простого способа выполнения данной операции не существует.

Начало работы

Итак, у вас есть Subversion-репозиторий с правом записи, а значит, можно познакомиться со стандартным рабочим процессом. Начнем с команды **git svn clone**, которая целиком импортирует Subversion-репозиторий в локальный Git-репозиторий. Напоминаем, что при импорте из настоящего репозитория в системе Subversion следует заменить в коде адрес **file:///tmp/test-svn** реальным URL-адресом:

```
$ 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-calcbranch,
75
Found branch parent: (refs/remotes/origin/my-calc-branch) 556a3e1e7ad1fde0a32823fc
7e4d046bcfd86dae
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
```

Этот код эквивалентен выполнению по указанному вами URL-адресу двух команд: **git svn init**, а затем **git svn fetch**. Процесс извлечения данных занимает некоторое время. Тестовый проект имеет всего 75 коммитов, и база его кода невелика, но системе Git приходится по отдельности проверять каждую версию файла и делать для нее коммит. В случае проекта с тысячами коммитов этот процесс может занять несколько часов или даже дней.

Запись **-T trunk -b branches -t tags** сообщает системе Git, что в данном репозитории Subversion используется стандартная система создания веток и тегов. Если вы применяете для стволов, веток или тегов нестандартные имена, поменяйте данные параметры. Но так как указанный вариант является общепринятым, этот фрагмент можно заменить сокращением **-s**. Оно указывает на стандартную компоновку и применяет все перечисленные параметры. То есть первую строку кода можно записать и так:

```
$ git svn clone file:///tmp/test-svn -s
```

К этому моменту у вас должен быть функционирующий Git-репозиторий с импортированными туда 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, каким образом этот инструмент представляет Subversion-теги в виде удаленных ссылок. Рассмотрим это более подробно при помощи служебной Git-команды **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 клонировании с Git-сервера система Git ничего подобного не делает; вот как сразу же после этой процедуры выглядит репозиторий с тегами:

```
$ 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
6dcb09b5b57875f334f61aebd695e2e4193db5e refs/tags/v1.0.0
```

Система Git извлекает теги напрямую в папку `refs/tags`, а не рассматривает их как удаленные ветки.

Отправка коммитов в систему Subversion

Теперь, когда у вас есть рабочий репозиторий, можно поработать над проектом и отправить изменения обратно на сервер, по сути, используя систему Git как клиент для SVN. Отредактировав файл и зафиксировав внесенные в него изменения, вы получите коммит, существующий локально в Git, но отсутствующий на Subversion-сервере:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
1 file changed, 5 insertions(+)
```

Значит, изменения следует отправить на сервер. Обратите внимание, как изменился привычный способ работы с Subversion, — теперь можно сделать сразу несколько коммитов локально и одновременно отправить их на Subversion-сервер. Это делается командой `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
```

Этот код берет все коммиты, которые вы добавили к коду с Subversion-сервера, фиксирует состояние каждого из них в Subversion и переписывает локальные

Git-коммиты, добавляя к ним уникальный идентификатор. Важно понимать, что при этом меняются все контрольные суммы SHA-1 ваших коммитов. Частично именно поэтому не стоит одновременно работать с удаленными версиями вашего проекта на основе Git и на Subversion-сервере. Посмотрев на последний коммит, вы обнаружите, что там появился новый идентификатор `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 контрольная сумма SHA, которая после исходной фиксации состояния начиналась с `4af61fd`, теперь начинается с `95e0222`. Если вы хотите отправлять результат своего труда одновременно на Git- и Subversion-серверы, первым делом выполните отправку (командой `dcommit`) на Subversion-сервер, потому что это меняет данные вашего коммита.

Скачивание изменений

При совместной работе то и дело возникает ситуация, когда кто-то из коллег успевает отправить изменения на сервер раньше, в результате возникает конфликт слияния. Ваши коммиты будут отвергаться, пока вы не добавите наработки коллеги к своим. В случае с командой `git svn` эта ситуация 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 c80b6127dd04f5fcda218730dd
 f3a2da4eb39138
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.
```

Для решения проблемы воспользуйтесь командой `git svn rebase`, которая заберет с сервера все изменения, пока отсутствующие у вас, и переместит ваши локальные наработки поверх содержащей их ветки:

```
$ 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 b34372b25ccf4945fe5658fa38
 1b075045e7702a
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.

```

После этого результаты вашего труда расположатся поверх данных с Subversion-сервера. То есть теперь история стала линейной и можно спокойно воспользоваться командой `dcommit`:

```

$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M   README.txt
Committed r85
M   README.txt
r85 = 9c29704cc0bbb7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and refs/remotes/
origin/trunk
Resetting to the latest refs/remotes/origin/trunk

```

Обратите внимание, что в отличие от Git, где перед отправкой данных на сервер всегда требуется выполнять локальное слияние свежих изменений из удаленного репозитория с результатами вашего труда, команда `git svn` требует этого только в случае конфликта (именно так функционирует Subversion). Если кто-то внес изменения в один файл, а вы — в другой, команда `dcommit` сработает без проблем:

```

$ 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: a0253d06732169107aa020390d9fef9d2b1d92806 and refs/remotes/origin/trunk differ,
  using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 e757b59a9439312d80d5d43bb6
 5d4a7d0389ed6d
M   autogen.sh
First, rewinding head to replay your work on top of it...

```

Об этом важно помнить, так как в результате проект оказывается в состоянии, не существующем ни у вас, ни на машине коллеги. В случае несовместимых, но не конфликтующих изменений могут возникнуть сложно диагностируемые проблемы. Git позволяет перед отправкой изменений на сервер полностью протестировать

состояние проекта на стороне клиента, в то время как в SVN нельзя быть уверенным даже в идентичности состояний непосредственно перед выполнением коммита и после него. Поэтому извлекайте изменения с Subversion-сервера почаще, даже если вы не собираетесь фиксировать состояние своих файлов. Для скачивания новых данных можно пользоваться командой `git svn fetch`, но лучше применять команду `git svn rebase`, которая не только извлекает данные, но и обновляет ваши локальные коммиты:

```
$ 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.
```

Периодическое выполнение команды `git svn rebase` поддерживает актуальное состояние локальной версии кода. Но применять ее следует только при пустой рабочей папке. Если у вас есть локальные изменения, скройте их или на время зафиксируйте, иначе команда `git svn rebase`, обнаружив, что перемещение приведет к конфликту слияния, остановит свою работу.

Проблемы с ветками в Git

Как только рабочий процесс с использованием Git станет для вас привычным, вы, скорее всего, захотите создавать тематические ветки, работать в них и сливать результаты труда в основную ветку. Но если вы отправляете данные на Subversion-сервер командой `git svn`, нужно будет каждый раз перемещать свою работу в одну ветку, а не сливать ветки друг с другом. Ведь в системе Subversion используется линейная история, и слияние в стиле системы Git не практикуется. Соответственно преобразуя снимок состояния в Subversion-коммит, команда `git svn` учитывает только первого предка.

Предположим, наша история выглядит следующим образом: вы создали ветку `experiment`, два раза зафиксировали внесенные в нее изменения и слили ее обратно в ветку `master`. В этом случае команда `dcommit` даст нам следующий результат:

```
$ 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
```

Итак, команда `dcommit` прекрасно работает в случае ветки с объединенной историей. Однако посмотрев на историю Git-проекта, вы обнаружите, что ни один из ваших коммитов в ветке `experiment` не переписан — все изменения появились только в SVN-версии как один общий коммит.

Пользователь, скопировавший себе результаты вашего труда, получит единственный коммит со всеми изменениями, как будто вы прибегли к команде `git merge --squash`; данных о том, кто внес изменения и когда они были зафиксированы, не будет.

Ветки в Subversion

Ветвление в системе Subversion отличается от этой процедуры в Git; поэтому по возможности прибегайте к ней как можно реже. Тем не менее команда `git svn` позволяет создавать ветки в системе Subversion и работать с ними.

Создание новой SVN-ветки

Новую ветку в системе Subversion создает команда `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) cb522197870e61467473391799148f672
1bcf9a0
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

Мы получаем тот же самый результат, что и в случае команды `svn copy trunk branches/opera`, просто действие выполняется на Subversion-сервере. Имейте в виду, что команда не переводит вас автоматически в эту ветку; если зафиксировать в этот момент состояние и отправить его на сервер, оно попадет в ветку `trunk`, а не в ветку `opera`.

Переход с ветки на ветку

Система Git определяет, в какую ветку поместить коммиты, отправляемые командой `dcommit`, по виду вершины любой из Subversion-веток в вашей истории — ведь у вас должна быть всего одна такая ветка, последняя в истории ветвлений с меткой `git-svn-id`.

Для одновременной работы с несколькими ветками можно настроить локальные ветки таким образом, чтобы команда `dcommit` отправляла коммиты из них в определенные Subversion-ветки. Для этого следует начинать локальные ветки от импортированного Subversion-коммита из нужной удаленной ветки. Например, чтобы получить ветку `opera` для независимой работы, выполните следующую команду:

```
$ git branch opera remotes/origin/opera
```

Теперь слить данные из ветки **opera** в ветку **trunk** (вашу ветку **master**) можно обычной командой **git merge**. Но нужно будет создать детальное сообщение фиксации (воспользовавшись параметром **-m**), иначе вместо полезной информации там будет написано «Merge branch opera».

Тем не менее следует помнить, что пусть вы и выполняете эту операцию командой **git merge**, объединение происходит намного проще, чем обычно в Subversion (так как система Git автоматически выбирает подходящую основу для слияния), и традиционного для Git коммита слияния вы не получите. Ведь эти данные нужно отправлять на Subversion-сервер, который не умеет работать с коммитами, имеющими более одного предка; именно поэтому после отправки вы получите единый коммит, в который втиснуты все изменения из другой ветки. После слияния одной ветки с другой нельзя просто вернуться к работе, как это обычно делается в Git. Команда **dcommit** удаляет все данные о том, из какой именно ветки были слиты изменения, что делает все последующие вычисления базы слияния некорректными. Фактически команда **dcommit** в данном случае дает вам такой результат, как если бы вместо команды **git merge** вы воспользовались командой **git merge --squash**. Простых способов избежать подобной ситуации не существует, потому что система Subversion не умеет сохранять эту информацию. Поэтому используя ее в качестве своего сервера, вы постоянно будете сталкиваться с ограничениями. Избавиться от проблем можно, только удалив локальную ветку (в данном случае ветку **opera**) после слива данных из нее в ветку **trunk**.

Subversion-команды

Инструментарий команды **git svn** включает в себя набор команд, облегчающих переход к системе Git, предлагая функциональность, аналогичную той, которая применяется в Subversion. Рассмотрим несколько таких команд.

История в стиле SVN

Если вы привыкли работать с Subversion и хотите просматривать историю коммитов в том виде, в каком она выводится именно этой системой, используйте команду **git svn log**:

```
$ 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
```

У команды **git svn log** существуют две важные особенности. Во-первых, она работает локально, в то время как команда **svn log** запрашивает данные с Subversion-сервера. Во-вторых, она показывает только коммиты, переданные на Subversion-сервер.

Локальные Git-коммиты, к которым еще не была применена команда `dcommit`, не показываются; как и коммиты, которые за это время добавили на Subversion-сервер другие пользователи. Фактически вам предоставляется последнее известное состояние коммитов на Subversion-сервере.

Комментарии в стиле SVN

Подобно тому, как команда `git svn log` во многом эмулирует команду `svn log`, эквивалентом команды `svn annotate` является команда `git svn blame [ФАЙЛ]`. Результат ее работы выглядит так:

```
$ 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
```

Эта команда тоже не показывает коммиты, выполненные локально средствами Git или отправленные за это время на Subversion-сервер другими пользователями.

Данные SVN-сервера

Для получения сведений, предоставляемых командой `svn info`, используйте 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)
```

Аналогично командам `blame` и `log`, эта команда действует локально и показывает информацию, актуальную на момент последнего обращения к Subversion-серверу.

Игнорирование того, что игнорирует Subversion

Если вы скопировали себе Subversion-репозиторий с заданными свойствами `svn:ignore`, скорее всего, вы захотите настроить соответствующие файлы

.gitignore, чтобы случайно не зафиксировать файлы, которые вам не требуются. С этой задачей вам помогут справиться две команды, связанные с **git svn**. Во-первых, это команда **git svn create-ignore**, автоматически создающая соответствующие файлы **.gitignore**, которые можно будет включить в следующий коммит.

Вторая команда, **git svn show-ignore**, вводит в стандартный поток вывода строки, которые следует включить в файл **.gitignore** для перенаправления вывода в файл исключений проекта:

```
$ git svn show-ignore > .git/info/exclude
```

В этом случае файлы **.gitignore** не будут захламлять проект. Именно так следует поступать, если вы являетесь единственным пользователем системы Git в группе, члены которой работают с Subversion и которым не нужны файлы **.gitignore** в папках проекта.

Подводя итоги

Инструменты, которые предоставляет команда **git svn**, полезны в ситуации, когда вам приходится работать с Subversion-сервером или ваша среда разработки по каким-то причинам требует взаимодействия с этим сервером. Но следует помнить, что в этом случае вы имеете дело с урезанной версией Git. В противном случае можно столкнуться с проблемами преобразования, сбивающими с толку как вас, так и ваших коллег. Избежать неприятностей поможет несколько рекомендаций:

- ❑ Сохраняйте Git-историю линейной, без созданных командой **git merge** коммитов слияния. Вся работу, которая выполняется вне основной ветки, добавляйте туда путем перемещения, а не слияния.
- ❑ Не нужно настраивать для себя отдельный Git-сервер. Вы можете пользоваться таким сервером, чтобы ускорить процедуру клонирования для новых разработчиков, но не отправляйте туда материалы, не имеющие метки **git-svn-id**. Возможно, имеет смысл добавить хук **pre-receive**, который будет проверять все сообщения фиксации на наличие этой метки и прерывать отправку не имеющих ее коммитов.

При следовании этим правилам работа с Subversion-сервером становится более или менее сносной. Но если есть возможность перехода на обычный Git-сервер, это нужно сделать, так как проект и рабочая группа от этого только выиграют.

Git и Mercurial

DVCS-пространство не ограничивается системой Git. В нем существует и множество других систем, у каждой из которых свой подход к управлению версиями. Второй по популярности после Git является система Mercurial, во многих отношениях сходная с Git.

Если вы предпочитаете пользоваться системой Git, но вынуждены работать с проектом, управление версиями которого отдано на откуп системе Mercurial, есть способ превратить Git в клиента для работы с репозиторием Mercurial. Так как Git прекрасно общается с удаленными репозиториями, вряд ли вас удивит тот факт, что в данном случае мост реализуется в виде удаленной вспомогательной функции. Она называется `git-remote-hg` и доступна по адресу <https://github.com/felipec/git-remote-hg>.

Утилита git-remote-hg

Первым делом следует установить программу `git-remote-hg`. Достаточно поместить ее по доступному адресу:

```
$ curl -o ~/bin/git-remote-hg \
https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

Предполагается, что папка `~/bin` входит в вашу переменную `$PATH`. У этой вспомогательной программы есть еще одна зависимость: Mercurial-библиотека для языка Python. Если этот язык у вас установлен, просто выполните команду:

```
$ pip install mercurial
```

(Если этого языка у вас пока нет, скачайте установщик с сайта <https://www.python.org/>.)

Наконец, нам потребуется Mercurial-клиент. Если он пока у вас отсутствует, скачайте его со страницы <http://mercurial.selenic.com/>.

Теперь все готово к работе. Вам нужен только Mercurial-репозиторий, в который будут отправляться результаты вашего труда. К счастью, для этого подходит любой репозиторий Mercurial, поэтому мы просто воспользуемся репозиторием «hello world», предназначенным для освоения системы Mercurial:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

Начало работы

Теперь у нас есть репозиторий на стороне сервера и можно рассмотреть типичный рабочий процесс. Вы убедитесь, что в данном случае системы во многом похожи, поэтому особых проблем не возникает.

Как всегда при работе с Git, мы первым делом выполняем клонирование:

```
$ 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
```

Обратите внимание, что для клонирования Mercurial-репозитория мы воспользовались стандартной командой `git clone`. Дело в том, что программа `git-remote-hg` функционирует на достаточно низком уровне, используя тот же самый механизм, который реализован в протоколе HTTP/S в Git (вспомогательные программы для работы с удаленным сервером). Так как работа с Git и Mercurial предполагает наличие у каждого клиента полной истории проекта, эта команда выполняет полное клонирование, причем достаточно оперативно.

Команда `log` показывает два коммита, последний из которых указывает на целый набор ссылок. Но далеко не все они существуют на самом деле. Посмотрим повнимательнее, что же хранится в папке `.git`:

```
$ tree .git/refs
.git/refs
├── heads
│   └── master
├── hg
│   └── origin
│       ├── bookmarks
│       │   └── master
│       └── branches
│           └── default
├── notes
│   └── hg
├── remotes
│   └── origin
│       └── HEAD
└── tags
```

```
9 directories, 5 files
```

Вспомогательная программа `git-remote-hg` пытается нивелировать различия между двумя системами, незаметно для нас преобразуя форматы. На самом деле ссылки на удаленные объекты хранятся в папке `refs/hg`. Например, `refs/hg/origin/branches/default` — это ссылка на Git-файл, содержащий контрольные суммы SHA, начиная с коммита `ac7955c`, на который указывает ветка `master`. Соответственно, папка `refs/hg` является своего рода имитацией папки `refs/remotes/origin`, но с добавленной способностью отличать закладки и ветки.

Отправной точкой, позволяющей определить, каким образом программа `git-remote-hg` сопоставляет хеши коммитов в Git и идентификаторы наборов изменений в Mercurial, служит файл `notes/hg`. Посмотрим, что в нем содержится:

```
$ 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, ссылка `refs/notes/hg` указывает на дерево, которое в базе данных Git-объектов представляет собой список других объектов с именами. Команда `git ls-tree` показывает права доступа, тип, хеш объекта и имя файла для элементов дерева. Наконец, добравшись до одного из элементов, мы обнаруживаем внутри у него двоичный массив данных с именем `ac9117f` (хеш SHA-1 коммита, на который указывает ветка `master`), содержащий внутри значение `0a04b98` (идентификатор Mercurial-набора изменений на вершине ветки `default`).

К счастью, вся эта детализированная информация нам по большей части не требуется. Типичная рабочая схема в данном случае не будет особо отличаться от привычного для нас порядка действий в Git.

Но есть одна вещь, которую следует учитывать: игнорируемые файлы. Механизм работы с ними в системах Mercurial и Git очень похож, но вряд ли стоит сохранять файл `.gitignore` в виде коммита в Mercurial-репозитории. К счастью, Git позволяет игнорировать файлы в локальной копии репозитория, а список исключений в Mercurial имеет совместимый с Git формат, поэтому файл достаточно скопировать поверх существующего:

```
$ cp .hgignore .git/info/exclude
```

Файл `.git/info/exclude` функционально аналогичен файлу `.gitignore`, но не включается в коммиты.

Рабочий процесс

Предположим, вы проделали некую работу в ветке `master` и несколько раз зафиксировали внесенные изменения. Теперь коммиты нужно отправить в удаленный репозиторий. Вот как он выглядит в данный момент:

```
$ 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` на два коммита опережает ветку `origin/master`, но эти коммиты существуют только на нашем компьютере. Посмотрим, не внес ли кто за это время важные изменения в репозиторий:

```
$ 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

```

Так как в данном случае был установлен флаг `--all`, мы видим ссылки `notes`, которые используются внутри программы `git-remote-hg`. Впрочем, на них можно просто не обращать внимания. В остальном все так, как мы и ожидали; в ветке `origin/master` появился один новый коммит, и наша история разошлась. В отличие от остальных систем контроля версий, рассматриваемых в этой главе, система Mercurial умеет работать со слияниями, так что ничего необычного делать не потребуется.

```

$ 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 инициировали тестирование, и оно прошло гладко. Ничто не мешает нам поделиться своими наработками с остальными членами группы:

```

$ git push
To hg::tmp/hello
    df85e87..0c64627 master -> master

```

Готово! Взглянув в Mercurial-репозиторий, вы обнаружите там именно то, что ожидали:

```

$ 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 с номером 2 сделан системой Mercurial, а наборы изменений с номерами 3 и 4 мы внесли программой `git-remote-hg`, отправив на сервер коммиты, созданные системой Git.

Ветки и закладки

В Git существует всего один вид веток — ссылка, которая перемещается по мере фиксации изменений. В Mercurial такой тип ссылок называется закладками (bookmarks). Закладки функционируют практически аналогично веткам в Git.

В системе Mercurial понятие «ветка» включает в себя куда больше. Ветка, в которой выполняется набор изменений, записывается вместе с этим набором и, таким образом, всегда остается в истории репозитория. Вот пример коммита, сделанного 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, которая начинается со слова `branch`. В системе Git это невоспроизводимо (да в этом и нет нужды, потому что оба типа веток допускают представление в виде ссылок), но программа `git-remote-hg` должна улавливать эту разницу, так как для Mercurial это существенно.

Создавать закладки в Mercurial так же просто, как и ветки 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
```

И все. А в 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 новый тег `[featureA]` в версии 5. Таким образом, со стороны Git закладки выглядят как обычные ветки, но с одним исключением: закладку невозможно удалить средствами Git (это одно из ограничений вспомогательных программ для работы с удаленными серверами).

Вы можете работать с и «утяжеленными» Mercurial-ветками: достаточно поместить такую ветку в пространство имен `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 выглядит со стороны системы 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** записано вместе с набором изменений номер 7.

Со стороны Git работа с ветками обоего типа выглядит одинаково: мы переходим в ветку, фиксируем изменения, скачиваем чужие наработки, производим слияние и отправляем результаты нашего труда в репозиторий, как это обычно происходит. Но есть один нюанс, о котором нужно знать. Система Mercurial не поддерживает изменение истории. Вы можете только добавлять туда новые изменения. Вот как будет выглядеть репозиторий Mercurial после интерактивного перемещения и принудительной отправки данных на сервер:

```

$ 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 наборы изменений 8, 9 и 10, которые теперь принадлежат ветке **permanent**, но старые наборы изменений никуда не делись. Такая ситуация может запутать ваших коллег, привыкших к системе Mercurial, поэтому старайтесь ее не допускать.

Подводя итоги

Системы Git и Mercurial настолько похожи, что их совместная эксплуатация происходит достаточно просто. Если избегать изменения уже опубликованной истории (а этого имеет смысл избегать и в общем случае, а не только при работе с Mercurial), вы можете даже не заметить, что на другом конце располагается другая система контроля версий.

Git и Perforce

Система контроля версий Perforce также крайне популярна в корпоративной среде. Она появилась еще в 1995 году и из рассматриваемых в этой главе систем является самой древней. Соответственно она спроектирована с ограничениями, присущими тому времени; вы должны быть постоянно подключены к единому центральному серверу, а локально хранится единственная версия файлов. Разумеется, функциональность и ограничения данной системы прекрасно подходили для решения ряда конкретных проблем, и для многих проектов, использующих Perforce, система Git подошла бы намного лучше.

Существуют два варианта совместного использования Perforce и Git. Во-первых, можно задействовать программу Git Fusion от создателей системы Perforce, позволяющую превратить поддеревья из Perforce-депо в Git-репозитории с доступом на чтение и запись. Во-вторых, можно применить клиентскую оболочку **git-p4**, позволяющую использовать Git в качестве клиента для работы с Perforce без изменения конфигурации Perforce-сервера.

Программа Git Fusion

Для системы Perforce существует программа Git Fusion (доступная по адресу <http://www.perforce.com/git-fusion>), синхронизирующая Perforce-сервер с Git-репозиториями на стороне сервера.

Настройка

Рассмотрим простейший способ установки программы Git Fusion. Вам нужно скачать со страницы <http://www.perforce.com/downloads/Perforce/20-User> образ виртуальной машины с предустановленным демоном и собственно программой. После завершения загрузки программа импортируется в ваше любимое средство виртуализации (мы воспользуемся VirtualBox).

При первом запуске виртуальная машина попросит вас указать пароли для трех пользователей Linux (**root**, **perforce** и **git**) и имя хоста, по которому данный компьютер идентифицируется в сети. После завершения этих операций вы увидите то, что показано на рис. 9.1.

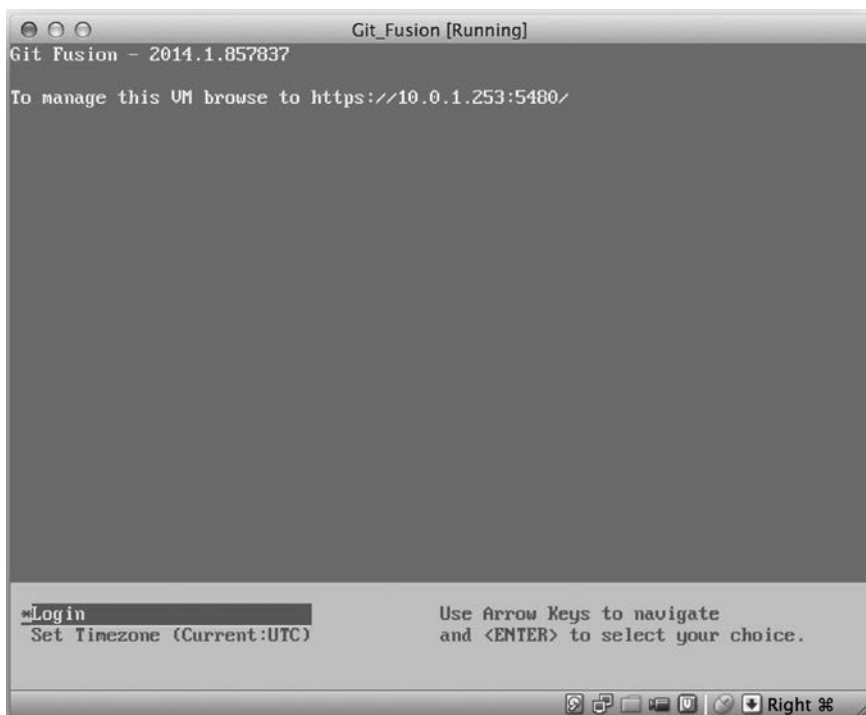


Рис. 9.1. Экран загрузки виртуальной машины Git Fusion

Запомните показанный на этом экране IP-адрес, потому что в будущем он вам потребуется. Теперь создадим пользователя **Perforce**. Выберите в нижней части экрана вариант **Login** и нажмите клавишу **Enter** (или воспользуйтесь **SSH**). Авторизуйтесь как пользователь **root**. Вот команды создания нового пользователя:

```
$ p4 -p localhost:1666 -u super user -f john
$ p4 -p localhost:1666 -u john passwd
$ exit
```

Первая команда открывает редактор **vi** для настройки пользователя, но вы можете принять параметры, предлагаемые по умолчанию, введя команду **:wq** и нажав клавишу **Enter**. Затем вас попросят дважды указать пароль. На этом работа с приглашением командного процессора заканчивается и можно завершить сеанс.

Теперь нужно запретить системе Git проверять сертификаты SSL. Виртуальная машина для Git Fusion имеет сертификат, но для домена, не совпадающего с IP-адресом

вашей виртуальной машины, поэтому Git будет разрывать HTTPS-соединения. Если вы собираетесь пользоваться этой виртуальной машиной постоянно, изучите руководство Perforce Git Fusion, чтобы узнать, как установить другой сертификат. А для наших целей вполне можно ограничиться командой:

```
$ export GIT_SSL_NO_VERIFY=true
```

Проверим, что все 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.
```

Виртуальная машина поставляется с примером проекта, который мы можем клонировать. Мы воспользуемся для этого протоколом HTTPS и ранее созданным пользователем `john`. При подключении система Git запросит ваши учетные данные. Благодаря кэшу учетных данных это единственный раз, когда вам нужно будет указать пароль, — в дальнейшем подключения этого не потребуют.

Конфигурирование программы Git Fusion

Установив программу Git Fusion, вы, возможно, захотите внести изменения в ее конфигурацию. Ваш любимый Perforce-клиент легко позволяет это сделать; просто спроецируйте папку `/.git-fusion` с Perforce-сервера на ваше рабочее пространство.

Структура файлов будет выглядеть так:

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
├── p4gf_config
├── repos
│   └── Talkhouse
│       └── p4gf_config
├── users
│   └── p4gf_usermap
```

```
498 directories, 287 files
```

Папкой `objects` программа Git Fusion пользуется для проецирования Perforce-объектов на Git-объекты и наоборот, поэтому здесь ничего трогать не нужно. В этой папке находится глобальный конфигурационный файл `p4gf_config` плюс по одному

такому же файлу для каждого репозитория — именно эти файлы и задают поведение программы Git Fusion. Посмотрим на файл в корневом каталоге:

```
[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
```

Не будем вдаваться в смысл каждого флага — достаточно понять, что это обычный текстовый файл в формате INI, напоминающий файлы, задающие конфигурацию Git. Этот файл содержит глобальные параметры, которые могут быть переопределены в конфигурационных файлах конкретных репозиториях, например `repos/Talkhouse/p4gf_config`. Внутри такого файла есть раздел `[@repo]` с настройками, отличающимися от используемых по умолчанию глобальных параметров. Найдете вы там и вот такие разделы:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ..
```

Это проецирование Perforce-ветки на Git-ветку. Названия таких разделов выбираются произвольным образом — главное, чтобы они были уникальными. Команда `git-branch-name` преобразует пути в депо, которые в Git выглядели бы громоздко, к более приемлемому виду. Параметр `view` отвечает за проецирование Perforce-файлов в репозитории Git, используя стандартный для таких случаев синтаксис. Можно указать несколько вариантов проецирования, как в приведенном примере:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
      //depot/project2/mainline/... project2/...
```

Таким образом, если ваше обычное проецирование рабочего пространства подразумевает изменения в структуре папок, это можно воспроизвести в репозитории Git.

Последним мы обсудим файл `users/p4gf_usermap`, проецирующий Perforce-пользователей на Git-пользователей. Возможно, он вам никогда не понадобится. Преобразуя Perforce-набор изменений в Git-коммит, программа Git Fusion по умолчанию ищет Perforce-пользователя и использует хранящиеся в этом файле адрес электронной почты и полное имя пользователя для заполнения поля автор/автор коммита в Git. При обратном преобразовании Perforce-пользователь по умолчанию ищется по адресу электронной почты, хранящемуся в поле `author` Git-коммита, и уже для этого пользователя предоставляется набор изменений (с соответствующими правами доступа). В большинстве случаев это прекрасно работает, но рассмотрим файл со следующими вариантами проецирования:

```
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 `<пользователь> <email> "<полное имя>"` и задает проекцию для одного пользователя. Первые две строки проецируют два адреса электронной почты одной и той же учетной записи в Perforce. Такое случается при создании Git-коммитов с использованием различных адресов (или с заменой адресов в процессе), но в данном случае мы хотим, чтобы эти два адреса проецировались на одного Perforce-пользователя. При создании Git-коммита из набора изменений в Perforce первая строка, совпадающая с Perforce-пользователем, служит для установления авторства коммита.

Последние две строки скрывают реальные имена и адреса пользователей `bob` и `joe` из созданных ими Git-коммитов. Так поступают, когда хотят превратить внутренний проект в открытое программное обеспечение, не раскрывая при этом данных о сотрудниках. Следует иметь в виду, что адреса и полные имена должны быть уникальными, в противном случае все коммиты система Git припишет одному фиктивному автору.

Рабочий процесс

Программа Git Fusion — это мост между системами контроля версий Perforce и Git. Посмотрим на рабочий процесс со стороны Git. Предполагается, что мы уже задали проекции для проекта `Jam`, используя показанный конфигурационный файл, так что теперь проект можно клонировать:

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://ben@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
```

```

Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest
  metrowerks on Beos -- 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 клонирование может занять некоторое время, ведь программа Git Fusion преобразует все необходимые наборы изменений из Perforce-истории в Git-коммиты. Эта операция выполняется локально на сервере, поэтому все делается относительно быстро, тем не менее какое-то время на это потребуется. При последующих скачиваниях преобразованию подвергаются только появившиеся за прошедшее время новые данные и скорость операций сравнима со скоростью работы обычного Git-сервера.

Как видите, наш репозиторий напоминает любой другой репозиторий Git. В нем три ветки, кроме того, система Git любезно создала локальную ветку **master**, следящую за веткой **origin/master**. Давайте выполним какие-нибудь действия и создадим пару новых коммитов:

```

# ...
$ 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.
[...]
```

У нас два новых коммита. Посмотрим, не внесли ли другие пользователи за это время каких-либо изменений:

```

$ 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.
[...]
```

Кажется, кто-то успел добавить новую информацию! Хотя из выводимых командой данных этого понять нельзя, коммит `6afeb15` создан Perforce-клиентом. С точки зрения системы Git он не отличается от любого другого коммита, что нам, собственно, и требуется. Посмотрим, как Perforce-сервер обрабатывает 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 все работает как положено. Посмотрим, как выглядит история файла `README` с точки зрения Perforce. Для этого мы воспользуемся функцией `r4v` для воспроизведения графа изменений (рис. 9.2).

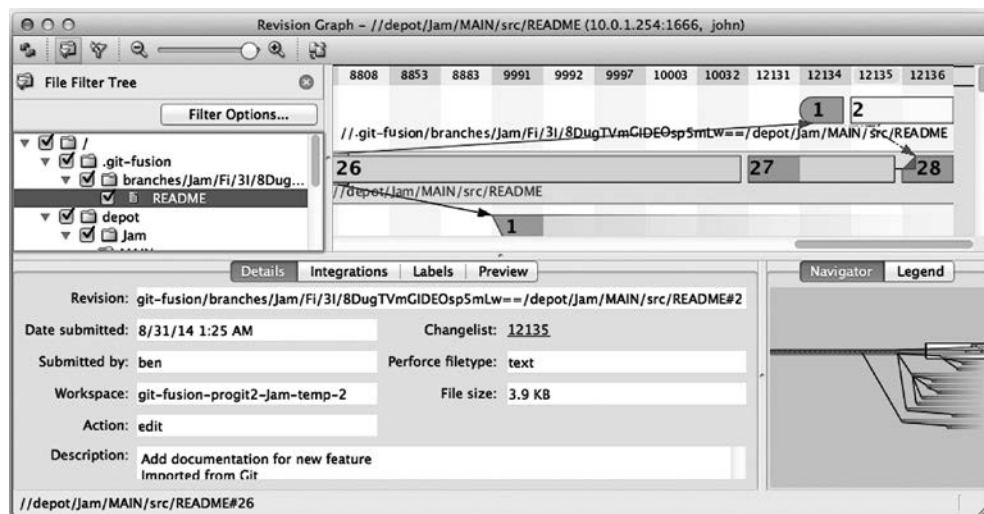


Рис. 9.2. Графическое представление версий в Perforce после отправки данных из Git

Тех, кто раньше никогда не пользовался этим инструментом, обилие информации может сбить с толку, но концептуально это то же самое, что визуализатор версий для Git-истории. Так как в данном случае мы рассматриваем историю файла README, дерево папок в левом верхнем углу демонстрирует этот файл, появляющийся в разных ветках. Справа сверху показано, как связаны друг с другом различные версии этого файла, а целиком этот граф представлен в правом нижнем углу. В остальной части окна мы видим подробности, связанные с выбранной версией (в данном случае это версия 2).

Следует отметить, что этот граф выглядит совершенно так же, как в Git-истории. Так как в системе Perforce отсутствует именованная ветка для хранения коммитов 1 и 2, была создана анонимная ветка в папке `.git-fusion`. Ровно то же самое произойдет с именованными ветками в Git, не имеющими соответствующих именованных веток в Perforce (позже вы можете выполнить проецирование через конфигурационный файл).

Большинство операций в данном случае происходит незаметно для нас, поэтому вполне возможна ситуация, когда кто-то в рабочей группе использует Git, кто-то — Perforce, и при этом ни один не подозревает о выборе другого.

Подводя итоги

Если у вас имеется (или вы можете получить) доступ к Perforce-серверу, программа Git Fusion станет прекрасным средством организации совместного использования Git и Perforce. Вам немного придется повозиться с настройками, но в целом освоение этой программы не должно быть очень сложным. Этот раздел — один из немногих в главе, где мы не предостерегаем вас против использования всей функциональности системы Git. Разумеется, не все ваши действия будут приняты: например, программа Git Fusion не даст вам переписывать уже опубликованную историю, но по большому счету, вы сможете работать привычным для вас образом. Вы сможете даже пользоваться подмодулями (хотя для Perforce-пользователей это будет выглядеть странно) и выполнять слияние веток (со стороны Perforce это будет фиксироваться как интеграция).

Впрочем, если у вас не получится убедить администратора вашего сервера настроить программу Git Fusion, не расстраивайтесь. Потому что существует еще одно средство объединения двух систем.

Мост Git-p4

Существует еще один мост между Git и Perforce — Git-p4. Он запускается внутри Git-репозитория, соответственно доступ к Perforce-серверу вам уже не потребуется (хотя вводить учетные данные, конечно, придется). Этот инструмент не так гибок и полнофункционален, как программа Git Fusion, но позволяет совершать большинство необходимых действий без вторжения в среду сервера.

ПРИМЕЧАНИЕ

Для работы с `git-p4` следует поместить исполняемый файл `p4` куда-нибудь в папку `PATH`. На момент написания данной книги он был доступен для бесплатного скачивания по адресу <http://www.perforce.com/downloads/Perforce/20-User>.

Настройка

Для демонстрационных целей мы воспользуемся Perforce-сервером из показанного ранее примера с Git Fusion, просто в данном случае мы минуем сервер Git Fusion, обращаясь непосредственно к системе контроля версий Perforce.

Чтобы воспользоваться клиентом командной строки `p4` (зависящим от `git-p4`), установим следующие переменные окружения:

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

Начало работы

Как это обычно бывает при работе с Git, первым делом следует выполнить 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 данном случае мы получаем так называемый поверхностный клон; в Git импортируется только самая последняя версия Perforce. В конце концов, система Perforce не предназначена для выдачи любому пользователю всех существующих версий. Полученной информации вполне достаточно, чтобы использовать Git в качестве клиента для Perforce, а вот другие задачи решать вы уже не сможете.

После завершения клонирования у нас появится полнофункциональный репозиторий Git:

```
$ 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 удаленный репозиторий `p4` для сервера Perforce. В остальном же у нас стандартный клонированный репозиторий. Впрочем, написанное не совсем точно отражает истинное положение дел. На самом деле никакого удаленного репозитория у нас нет.

```
$ git remote -v
```

Инструмент `git-p4` просто создал ссылки, представляющие состояние сервера. Именно они для команды `git log` выглядят как удаленные серверы, но при этом система Git ими не управляется и вы не можете отправлять туда свои данные.

Рабочий процесс

Приступим к работе. Предположим, вы внесли существенные изменения в важный программный компонент и хотите показать их остальным членам рабочей группы:

```
$ 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 сделали два коммита, готовых к отправке на Perforce-сервер. Посмотрим, не появилось ли там за это время новой информации от коллег:

```
$ 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
```

Как видите, ветки **master** и **p4/master** успели разойтись. Система ветвлений в Perforce совсем не похожа на то, к чему мы привыкли в Git, поэтому отправка коммитов слияния не имеет смысла. Инструмент **git-p4** рекомендует перемещать коммиты и даже предоставляет для этого сокращенную команду:

```
$ 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(-)
```

Из выводимых этой командой данных можно понять, что **git p4 rebase** — краткая запись для команды **git p4 sync**, за которой следует команда **git rebase p4/master**. На самом деле эта команда гораздо более интеллектуальна, особенно при работе с несколькими ветками, но, по сути, это верное описание.

Теперь наша история снова линейна и все готово для отправки наших изменений на Perforce-сервер. Команда **git p4 submit** попытается создать новые версии Perforce для всех Git-коммитов между коммитами **p4/master** и **master**. Она открывает наш любимый редактор примерно с 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>

```

Практически эти же данные вы получили бы, запустив команду `p4 submit`, за исключением нескольких строк в конце, которые услужливо добавила программа `git-p4`. Эта программа, представляя имя для коммита или набора изменений, старается отдельно учитывать настройки Git и Perforce, но в некоторых случаях вам потребуется переопределить их. Например, если импортируемый вами Git-коммит был создан пользователем, у которого отсутствует учетная запись в Perforce,

но вы все равно хотите, чтобы итоговый набор изменений был приписан его, а не вашему авторству.

Программа **git-p4** любезно вставила сообщение из Git-коммита внутрь данного Perforce-набора изменений, поэтому нам остается только дважды выполнить операции сохранения и выхода из редактора (по одной на каждый коммит). Вот какой результат мы после этого получим:

```
$ 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
```

Результат выглядит так, как будто мы выполнили команду **git push**, что по большому счету недалеко от истины.

Обратите внимание, что во время этого процесса каждый Git-коммит превращается в Perforce-набор изменений; до выполнения команды **git p4 submit** их можно превратить в один набор изменений, воспользовавшись интерактивным перемещением. Также следует обратить внимание на изменение хешей SHA всех превращенных в наборы изменений коммитов. Это связано с тем, что программа **git-p4** добавляет в конце каждого преобразуемого ею коммита строку:

```
$ 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 произойдет, если отправить этот коммит слияния на сервер? Давайте проверим это на практике. Пусть наша история имеет вот такой вид:

```
$ 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
```

Git- и Perforce-истории расходятся после коммита **775a46f**. Со стороны Git есть два коммита, затем коммит слияния с состоянием Perforce, после этого еще один коммит. Попробуем переместить это наверх единого набора изменений в Perforce. Вот что произойдет при отправке данных:

```
$ 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** является сокращением для параметра **--dry-run**, который отвечает за предварительный вид результата отправки данных. В нашем случае появляются три Perforce-набора изменений, что соответствует трем коммитам, не являющимся результатами слияния, которые пока отсутствуют на Perforce-сервере. В принципе, это именно то, что нам требуется, посмотрим, что получится на практике:

```
$ 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, как будто мы выполнили перемещение перед отправкой данных (что на самом деле и произошло). Фактически вы можете свободно создавать ветки на стороне Git, работать в них, удалять их и выполнять их слияние, не боясь, что ваша история станет несовместимой с системой Perforce. Если данные допускают перемещение, их можно добавлять на Perforce-сервер.

Ветвление

Если Perforce-проект включает в себя несколько веток, ничего страшного; программа **git-p4** позволяет работать с ними как с обычными Git-ветками. Предположим, наше Perforce-депо выглядит следующим образом:

```
//depot
└─ project
   └─ main
      └─ dev
```

При этом у вас есть ветка **dev**, которая имеет вот такой вид:

```
//depot/project/main/... //depot/project/dev/...
```

Программа **git-p4** автоматически распознает ситуацию и корректно на нее реагирует:

```
$ 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 спецификатор **@all** в составе пути к депо; он указывает программе **git-p4**, что клонировать следует не только самый последний набор изменений для этого поддева, но и все наборы изменений, когда-либо находившиеся в указанном месте. Это больше напоминает процедуру клонирования в Git, но при работе с проектом, имеющим долгую историю, может занять изрядное время.

Флаг **--detect-branches** заставляет программу **git-p4** использовать настройки Perforce-веток для поиска соответствующих им ссылок в Git. Если на Perforce-сервере эти соответствия не заданы (потому что для работы Perforce это не обязательно), можно задать их вручную и получить в итоге тот же самый 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.
```

Присвоив конфигурационной переменной `git-p4.branchList` значение `main:dev`, мы сообщим `git-p4`, что `main` и `dev` — это ветки, причем вторая является потомком первой.

Если теперь выполнить команду `git checkout -b dev p4/project/dev` и создать несколько коммитов, программа `git-p4` догадается, в какую ветку отправлять данные при выполнении команды `git p4 submit`. К сожалению, `git-p4` не позволяет одновременно использовать «поверхностные» клоны и набор веток; если вам нужно работать с несколькими ветками, команда `git p4 clone` выполняется для каждой ветки, в которую вы планируете отправлять данные.

Для создания и интеграции веток вам потребуется Perforce-клиент. Программа `git-p4` умеет только синхронизировать и отправлять на сервер содержимое существующих веток, причем только по одному линейному набору изменений за раз. При попытке отправить на сервер набор изменений, полученный слиянием двух веток в Git, записаны будут только результаты изменения файлов; все метаданные об участвовавших в интеграции ветках теряются.

Подводя итоги

Программа `git-p4` дает возможность реализовывать рабочие схемы системы Git для Perforce-сервера и весьма успешно с этим справляется. Но не стоит забывать, что в такой ситуации источником данных является Perforce, а Git используется лишь для локальной работы. Будьте осторожны с публикацией Git-коммитов; если у вас есть удаленный сервер, которым пользуется кто-то еще, не отправляйте туда коммиты, которые пока отсутствуют на Perforce-сервере.

Если вы хотите свободно комбинировать системы Perforce и Git, используя их в качестве клиентов для контроля версий, и в состоянии уговорить администратора сервера установить программу Git Fusion, вы получите первоклассный Git-клиент для работы с Perforce-сервером.

Git и TFS

Система Git набирает популярность среди Windows-разработчиков, и если вы относитесь к их числу, велика вероятность, что вы знакомы с таким продуктом Microsoft, как Team Foundation Server (TFS). Это комплексное решение, включающее в себя систему отслеживания ошибок, систему учета рабочего времени, поддержку Scrum-процессов, анализ кода и контроль версий. Существует небольшая терминологическая путаница. Дело в том, что TFS — это сервер, поддерживающий управление

версиями как посредством Git, так и при помощи собственной системы контроля версий, которая называется TFVC (Team Foundation Version Control). Поддержка Git появилась в TFS относительно недавно (начиная с версии 2013), поэтому названия всех инструментов, применявшихся в предшествующих версиях, включают в себя аббревиатуру «TFS», хотя работают они в основном с TFVC.

Если все члены вашей рабочей группы пользуются TFVC, в то время как вы предпочитаете для контроля версий задействовать систему Git, для вас существует свой инструмент.

Выбор инструмента

На самом деле инструментов два: `git-tf` и `git-tfs`.

Второй (находящийся по адресу <http://git-tfs.com>) написан на языке .NET и на момент написания данной книги работал только в операционной системе Windows. Для работы с Git-репозиториями он использует .NET-привязки для библиотеки `libgit2`. Это крайне гибкая и производительная библиотека, позволяющая выполнять множество низкоуровневых операций с Git-репозиторием. Но полностью функциональность Git эта библиотека не покрывает; впрочем, в таких случаях проект `git-tfs` вызывает Git-клиент командной строки, поэтому с Git-репозиториями вы сможете делать практически что угодно. Впечатляет своей полнотой и поддержка TFVC, так как для работы с серверами используются сборки Visual Studio (хотя это и требует установленной версии Visual Studio с доступом к TFVC; на момент написания данной книги бесплатных версий Visual Studio, умеющих подключаться к TFS-серверу, не было).

Инструмент `git-tf` (расположенный по адресу <https://gittf.codeplex.com>) написан на языке Java и запускается на любом компьютере со средой исполнения Java. Взаимодействие с Git-репозиториями осуществляется посредством библиотеки JGit (JVM-реализация системы Git), что означает практически полное отсутствие ограничений на использование функциональности системы Git. К сожалению, поддержка этого проекта для TFVC ограничена по сравнению с тем, что мы имеем в `git-tfs`, — например, не поддерживаются ветки.

Итак, каждый из инструментов имеет свои сильные и слабые стороны, для каждого из них существует множество ситуаций, в которых именно он оказывается наиболее предпочтительным. Далее мы рассмотрим основы их применения.

ПРИМЕЧАНИЕ

Чтобы проверить на практике примеры из книги, потребуется репозиторий на базе TFVC. Это не настолько распространенная вещь, как репозитории Git или Subversion, поэтому, возможно, вам придется создавать такой репозиторий самостоятельно. Для этой цели лучше воспользоваться системой Codeplex (<https://www.codeplex.com>) или Visual Studio Online (<http://www.visualstudio.com>).

Начало работы: git-tf

Первое, что нужно сделать в любом Git-проекте, — выполнить клонирование. Вот как это выглядит с инструментом **git-tf**:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/  
Main project_git
```

Первым аргументом тут является URL-адрес TFVC-коллекции, второй аргумент представляет собой строку вида `$/project/branch`, третий — путь к локальному Git-репозиторию, который следует создать (последний аргумент является необязательным). Инструмент **git-tf** умеет работать только с одной веткой за раз; для перехода на другую ветку в TFVC следует создать еще один клон репозитория уже из этой ветки.

Приведенная команда создает полнофункциональный Git-репозиторий:

```
$ cd project_git  
$ git log --all --oneline --decorate  
512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

В данном случае у нас снова имеется так называемая поверхностная копия, то есть загружен только самый последний набор изменений. Система TFVC не предусматривает предоставления каждому клиенту полной копии истории, поэтому инструмент **git-tf** по умолчанию скачивает только последнюю версию, экономя ресурсы.

Если у вас есть на это время, возможно, имеет смысл скачать всю историю, добавив к команде параметр `--deep`:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main \  
project_git --deep  
Username: domain\user  
Password:  
Connecting to TFS...  
Cloning $/myproject into /tmp/project_git: 100%, done.  
Cloned 4 changesets. Cloned last changeset 35190 as d44b17a  
$ cd project_git  
$ git log --all --oneline --decorate  
d44b17a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Goodbye  
126aa7b (tag: TFS_C35189)  
8f77431 (tag: TFS_C35178) FIRST  
0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \  
Team Project Creation Wizard
```

Обратите внимание на теги с такими именами, как `TFS_C35189`, — они помогают увидеть, как Git-коммиты связаны с TFVC-наборами изменений. Это очень удобное представление, так как простая команда `log` показывает, какие из ваших коммитов связаны со снимком состояния, существующим и в TFVC. Это дополнительная функциональность (ее можно отключить командой `git config git-tf.tag false`), потому что инструмент **git-tf** хранит проекции коммитов на наборы изменений в файле `.git/git-tf`.

Начало работы: git-tfs

Клонирование инструментом **git-tfs** выглядит немного по-другому:

```
PS> git tfs clone --with-branches \
    https://username.visualstudio.com/DefaultCollection \
    $/project/Trunk project_git
Initialized empty Git repository in C:/Users/ben/project_git/.git/
C15 = b75da1aba1ffb359d00e85c52acb261e4586b0c9
C16 = c403405f4989d73a2c3c119e79021cb2104ce44a
Tfs branches found:
- $/tfvc-test/featureA
The name of the local branch will be : featureA
C17 = d202b53f67bde32171d5078968c644e562f1c439
C18 = 44cd729d8df868a8be20438fdeeeefb961958b674
```

Обратите внимание на флаг **--with-branches**. Инструмент **git-tfs** умеет проецировать TFVC-ветки на Git-ветки, и этот флаг является командой завести по локальной Git-ветке для каждой TFVC-ветки. Всем, кто когда-либо создавал ветки или выполнял слияния в TFS, имеет смысл это сделать, но для более старых, чем TFS 2010, версий эта функциональность не поддерживается. До появления этой версии «ветки» представляли собой обычные папки, поэтому инструмент **git-tfs** не может отличить их от папок.

Рассмотрим полученный нами Git-репозиторий:

```
PS> git log --oneline --graph --decorate --all
* 44cd729 (tfs/featureA, featureA) Goodbye
* d202b53 Branched from $/tfvc-test/Trunk
* c403405 (HEAD, tfs/default, master) Hello
* b75da1a New project
PS> git log -1
commit c403405f4989d73a2c3c119e79021cb2104ce44a
Author: Ben Straub <ben@straub.cc>
Date: Fri Aug 1 03:41:59 2014 +0000
```

Hello

```
git-tfs-id: [https://username.visualstudio.com/DefaultCollection]$/myproject/
Trunk;C16
```

У нас есть две локальные ветки — **master** и **featureA**, которые представляют собой изначальную отправную точку клона (TFVC-ветка **Trunk**) и дочернюю ветку (TFVC-ветка **featureA**). Кроме того, «удаленный» репозиторий **tfs** тоже имеет пару ссылок: **default** и **featureA**, соответствующих TFVC-веткам. Инструмент **git-tfs** проецирует ветку, клонированную вами от ветки **tfs/default**, а остальные ветки получают собственные имена.

Следует обратить внимание и на строки **git-tfs-id**: в сообщениях фиксации. Вместо тегов проецирование TFVC-наборов изменений на Git-коммиты осуществляется посредством этих маркеров. Последствием этого становится изменение контрольной суммы SHA-1 Git-коммитов после их отправки на TFVC-сервер.

Рабочий процесс с git-tf[s]

ПРИМЕЧАНИЕ

Вне зависимости от того, какой инструмент вы выбрали для работы, задайте пару конфигурационных параметров:

```
$ git config set --local core.ignorecase=true
$ git config set --local core.autocrlf=false
```

Очевидно, что после клонирования проекта вам захочется над ним поработать. Как инструменты, TFVC и TFS обладают рядом конструктивных особенностей, которые могут затруднить рабочий процесс:

1. Осложняют дело отсутствующие на TFVC-сервере функциональные ветки (feature branches). Это связано с тем, что TFVC и Git по-разному осуществляют ветвление.
2. Следует помнить, что в TFVC можно запретить другим пользователям редактировать файлы на сервере. Разумеется, это не мешает вносить изменения в ваш локальный репозиторий, но отправить результат редактирования на TFVC-сервер вы не сможете.
3. В TFS существует концепция «условных» наборов изменений, в рамках которой изменения принимаются сервером только после успешного прохождения встроенных в TFS процедур сборки и тестирования. Это реализуется через TFVC-функцию включения в набор отложенных изменений, которую мы подробно рассматривать не будем. Инструмент **git-tf** позволяет вручную эмулировать подобное поведение, а в **git-tfs** существует умеющая работать с такими наборами команда **checkintool**.

Для краткости мы рассмотрим простой сценарий работы, не касающийся вышеупомянутых особенностей.

Рабочий процесс: git-tf

Итак, вы выполнили некую работу, создали в ветке **master** пару коммитов и готовы отправить результаты своего труда на TFVC-сервер. Наш Git-репозиторий выглядит так:

```
$ git log --oneline --graph --decorate --all
* 4178a82 (HEAD, master) update code
* 9df2ae3 update readme
* d44b17a (tag: TFS_C35190, origin_tfs/tfs) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

Мы хотим отправить на TFVC-сервер снимок состояния, находящийся в коммите 4178a82. Но первым делом нужно убедиться, что с момента последнего подключения к серверу там не появилось новых коммитов от коллег:

```
$ git tf fetch
Username: domain\user
Password:
Connecting to TFS...
Fetching $/myproject at latest changeset: 100%, done.
Downloaded changeset 35320 as commit 8ef06a8. Updated FETCH_HEAD.
$ git log --oneline --graph --decorate --all
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
| * 4178a82 (HEAD, master) update code
| * 9df2ae3 update readme
|/
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

Похоже, вы работаете над проектом не в одиночку, так как истории разошлись. Система Git отлично справляется с такими ситуациями, но в данном случае мы можем действовать двумя способами:

1. Пользователи Git в таких случаях делают коммит слияния (в конце концов, именно это реализует команда `git pull`), а `git-tf` делает это простой командой `git tf pull`. Но следует помнить, что система TFVC воспринимает эту ситуацию по-другому, и после отправки коммита слияния на сервер ваша история начнет по-разному выглядеть со стороны Git и TFVC, что может привести к путанице. Но это, наверное, самый быстрый способ отправить все изменения одним набором.
2. Перемещение дает нам линейную историю коммитов, что позволяет преобразовать каждый из наших Git-коммитов в TFVC-набор изменений. Это наиболее гибкий способ, поэтому мы рекомендуем пользоваться именно им; еще проще этот результат можно получить в `git-tf`, где есть команда `git tf pull --rebase`.

Выбор за вами. Вот пример, в котором используется перемещение:

```
$ git rebase FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

Теперь можно отправить данные на TFVC-сервер. В **git-tf** вы можете создать единый набор изменений, который будет представлять все изменения, начиная с последнего (за это отвечает заданный по умолчанию параметр **--shallow**), или создать новый набор изменений для каждого Git-коммита (за это отвечает параметр **--deep**). В рассматриваемом примере мы создадим всего один набор изменений:

```
$ git tf checkin -m 'Updating readme and code'
Username: domain\user
Password:
Connecting to TFS...
Checking in to $/myproject: 100%, done.
Checked commit 5a0e25e in as changeset 35348
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, tag: TFS_C35348, origin_tfs/tfs, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
    Team Project Creation Wizard
```

Появился новый тег **TFS_C35348**, указывающий на то, что система TFVC хранит такое же состояние, как коммит **5a0e25e**. Важно понимать, что далеко не у всех Git-коммитов есть точное соответствие в TFVC: например, коммит **6eb3eb5** на сервере отсутствует.

Так выглядит базовый рабочий процесс. Есть еще пара моментов, о которых следует помнить:

1. Процедуры ветвления не существует. **Git-tf** за один раз создает Git-репозиторий только из одной TFVC-ветки.
2. Для обмена результатами работы пользуйтесь TFVC- или Git-сервером, но не обоими одновременно. Разные клоны одного TFVC-репозитория могут обладать различными контрольными суммами SHA, что становится причиной множества проблем.
3. Если рабочий процесс вашей группы включает обмен данными через Git и периодическую синхронизацию с TFVC, всегда подключайтесь к TFVC из одного и того же Git-репозитория.

Рабочий процесс: git-tfs

Рассмотрим тот же сценарий в **git-tfs**. Вот новые коммиты, созданные в ветке **master** нашего репозитория Git:

```
PS> git log --oneline --graph --all --decorate
* c3bd3ae (HEAD, master) update code
* d85e5a2 update readme
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
```

```
|/
* c403405 (tfs/default) Hello
* b75da1a New project
```

Посмотрим, появились ли на сервере новые изменения:

```
PS> git tfs fetch
C19 = aea74a0313de0a391940c999e51c5c15c381d91d
PS> git log --all --oneline --graph --decorate
* aea74a0 (tfs/default) update documentation
| * c3bd3ae (HEAD, master) update code
| * d85e5a2 update readme
|/
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Действительно, кто-то из коллег успел добавить новый TFVC-набор изменений, который для нас выглядит как новый коммит **aea74a0** и перемещение удаленной ветки **tfs/default**.

Как и в случае **git-tf**, избавиться от расходящихся историй можно двумя способами:

1. Выполнить перемещение, сделав историю линейной.
2. Выполнить слияние, сохранив историю без изменений.

В данном случае мы хотим отправить данные на сервер таким образом, чтобы каждый Git-коммит превратился в TFVC-набор изменений, поэтому выполним перемещение:

```
PS> git rebase tfs/default
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
PS> git log --all --oneline --graph --decorate
* 10a75ac (HEAD, master) update code
* 5cec4ab update readme
* aea74a0 (tfs/default) update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Теперь мы готовы внести свой вклад в проект, отправив код на TFVC-сервер. Воспользуемся командой **rcheckin**, чтобы создать TFVC-набор изменений для каждого Git-коммита, от отмеченного указателем **HEAD** до первой обнаруженной удаленной ветки **tfs remote**. Команда **checkin** создает только один набор изменений, соединяя все коммиты друг с другом:

```
PS> git tfs rcheckin
Working with tfs remote: default
```

```

Fetching changes from TFS to minimize possibility of late conflict...
Starting checkin of 5cec4ab4 'update readme'
add README.md
C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017
Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto new TFS-
commit...
Rebase done successfully.
Starting checkin of b1bf0f99 'update code'
edit .git\tfs\default\workspace\ConsoleApplication1\ConsoleApplication1\Program.cs
C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b
Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto new TFS-
commit...
Rebase done successfully.
No more to rcheckin.
PS> git log --all --oneline --graph --decorate
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project

```

Обратите внимание, как после каждой удачной отправки данных на TFVC-сервер инструмент **git-tfs** перемещает еще не отправленные данные. Это связано с добавлением в нижнюю часть сообщений фиксации поля **git-tfs-id**, что приводит к изменению контрольных сумм SHA-1. Беспокоиться в данном случае не о чем, просто таким образом работает система. Но вы должны быть в курсе этого обстоятельства, особенно если делитесь своими Git-коммитами с другими пользователями.

В TFS есть множество функциональных особенностей, интегрированных в систему контроля версий, например рабочие элементы, специализированные средства обзора кода, условные наборы изменений и многое другое. Их применение через инструмент командной строки может показаться сложным, но, к счастью, в проекте **git-tfs** есть и программа с графическим интерфейсом:

```

PS> git tfs checkintool
PS> git tfs ct

```

Она выглядит примерно так, как показано на рис. 9.3.

Пользователи TFS сразу узнают это окно диалога, так как именно оно открывается в Visual Studio.

Кроме того, **git-tfs** позволяет из Git-репозитория контролировать TFVC-ветки. К примеру, создадим новую ветку:

```

PS> git tfs branch $/tfvc-test/featureBee
The name of the local branch will be : featureBee
C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5
PS> git lga
* 1d54865 (tfs/featureBee) Creation branch $/myproject/featureBee
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme

```



```
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

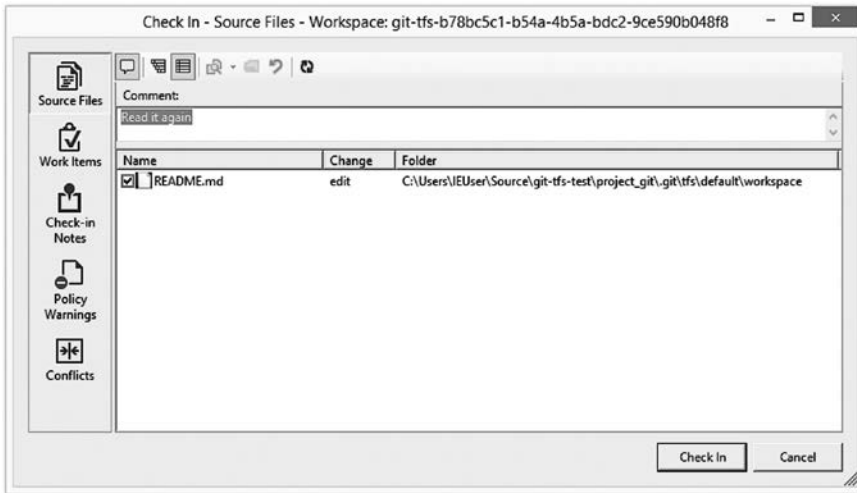


Рис. 9.3. Инструмент git-tfs

В TFVC создание ветки означает создание первого набора изменений, с которого она, собственно, начнется, и все это выглядит как Git-коммит. Обратите внимание, что хотя инструмент **git-tfs** создал удаленную ветку **tfs/featureBee**, указатель **HEAD** по-прежнему нацелен на ветку **master**. Чтобы работа продолжилась в новой ветке, новые коммиты нужно перенести к коммиту **1d54865**, например, формируя с этого места тематическую ветку.

Подводя итоги

Инструменты **git-tf** и **git-tfs** отлично подходят для взаимодействия с TFVC-сервером. Они позволяют локально использовать потрясающие возможности системы Git без постоянного взаимодействия с центральным TFVC-сервером, что облегчает жизнь разработчика, не вынуждая всю группу переходить на Git. Если вы работаете в операционной системе Windows (а раз ваша группа пользуется TFS, скорее всего, это именно так), вам лучше обратиться к инструменту **git-tfs** с его более полной функциональностью, а на долю работающих на других платформах остается инструмент **git-tf**, возможности которого куда более ограничены. Как и в случае остальных описывавшихся в этой главе инструментов, следует выбрать одну систему контроля версий в качестве основной. Для обмена результатами работы с коллегами следует использовать либо Git, либо TFVC, но никак не обе системы сразу.

Переход на Git

Если при наличии базы кода в другой VCS вы решили перейти на Git, следует каким-то образом выполнить перенос вашего проекта. В этом разделе мы сначала рассмотрим варианты импорта, поддерживаемые в распространенных системах, а затем покажем пример разработки собственного средства переноса данных. Выбор систем, о которых пойдет речь в данном разделе, обусловлен количеством пользователей, выполняющих перенос, а также наличием качественного инструментария.

Subversion

Тем, кто прочитал раздел о работе с `git svn`, не составит труда воспользоваться командой `git svn clone` для клонирования репозитория. После этого про Subversion-сервер можно забыть, отправляя все новые изменения уже на новый Git-сервер. Скопировать историю целиком вам ничто не мешает, хотя этот процесс может занять длительное время.

Однако процедура импорта не идеальна. И так как она является длительной, лучше сразу делать все правильно. Первая проблема связана с данными об авторстве. В системе Subversion все, кто отправляет результат своих трудов на сервер, обладают учетными записями, сведения о которых фиксируются в данных коммитов. К примеру, в предыдущем разделе в выводимых командами `blame` и `git svn log` данных фигурировал пользователь `schacon`. Поиск соответствующей информации в Git возможен только при наличии файла проекций Subversion-пользователей на авторов в Git. Создадим файл `users.txt`, содержащий данную информацию в следующем формате:

```
schacon = Scott Chacon <schacon@gmail.com>
selse = Someo Nelse <selse@gmail.com>
```

Получим список авторов, которые пользуются SVN:

```
$ svn log --xml | grep author | sort -u | \
perl -pe 's/.*>(.*?)<.*$/1 = /'
```

Эта команда генерирует список в формате XML, оставляя только строки с данными об авторе, избавляясь от дубликатов и удаляя XML-теги. (Разумеется, все это будет работать только на компьютере с установленными приложениями `grep`, `sort` и `perl`.) Вывод этой команды можно направить в файл `users.txt`, после чего в каждой строке остается только дописать соответствующих Git-пользователей.

Этот файл можно передать инструменту `git svn`, обеспечивая более корректное проецирование данных об авторах. При этом параметр `--no-metadata`, добавленный к команде `clone` или `init`, означает, что `git svn` не будет добавлять обычно импортируемые системой Subversion метаданные. В результате команда импорта примет такой вид:

```
$ git-svn clone http://my-project.googlecode.com/svn/ \
--authors-file=users.txt --no-metadata -s my_project
```

После этого в папке `my_project` появится более корректный результат импорта из системы Subversion. Например:

```
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-proje ct.googlecode.com/svn/trunk@94 4c93b258-373f-11debe05-5f7a86268029
```

Вместо этих коммитов 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 мы не только имеем более удобное представление поля **Author**, но и убрали строку `git-svn-id`.

Кроме того, импортированные данные следует почистить. Сначала надо удалить ненужные ссылки, которые добавляет инструмент `git svn`. Первым делом мы переместим все теги таким образом, чтобы из непонятных удаленных веток они действительно превратились в теги, а затем переместим остальные ветки, сделав их локальными.

Вот как выглядит перемещение тегов:

```
$ cp -Rf .git/refs/remotes/origin/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/origin/tags
```

Мы берем все ссылки на удаленные ветки с адресом `remotes/origin/tags/` и превращаем их в легковесные (`lightweight`) теги.

Теперь переместим ветки из папки `refs/remotes`, сделав их локальными:

```
$ cp -Rf .git/refs/remotes/* .git/refs/heads/
$ rm -Rf .git/refs/remotes
```

Теперь все старые ветки стали настоящими Git-ветками, а все старые теги — настоящими Git-тегами. Осталось только добавить наш новый Git-сервер в качестве удаленного репозитория и отправить туда первые данные. Вот пример превращения сервера в удаленный репозиторий:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Так как все наши ветки и теги должны находиться на этом сервере, выполним команду:

```
$ git push origin --all
```

Итак, все наши ветки и теги аккуратно импортированы на новый Git-сервер.

Mercurial

Так как модели представления версий в системах Mercurial и Git во многом схожи, причем Git обладает несколько большей гибкостью, процедура перевода репозитория с Mercurial на Git во многом очевидна. Она выполняется инструментом **hg-fast-export**, копию которого вам нужно будет получить:

```
$ git clone http://repo.or.cz/r/fast-export.git /tmp/fast-export
```

Первым этапом преобразования является получение полного клона нужного нам репозитория Mercurial:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

Следующим шагом является создание файла проецирования авторов. Система Mercurial не так строго подходит к данным, помещаемым в поле **author** для наборов изменений, поэтому сейчас удобный момент навести порядок. Сгенерировать такой файл позволяет однострочная команда для командной оболочки **bash**:

```
$ cd /tmp/hg-repo
$ hg log | grep user: | sort | uniq | sed 's/user: */' > ../authors
```

Через несколько секунд в зависимости от размера истории вашего проекта у вас появится вот такой файл **/tmp/authors**:

```
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 рассматриваемом примере один и тот же человек с именем **Bob** создал наборы изменений под четырьмя разными именами, одно из которых вполне корректно с точки зрения Git-коммита, в то время как другие совсем не соответствуют формату. Исправить ситуацию поможет команда **hg-fast-export**. С ее помощью мы добавляем в конец каждой изменяемой строки строку **= {новое имя и адрес электронной почты}** и удаляем строки для имен пользователей, которые мы оставляем без изменений. Если все имена пользователей выглядят корректно, этот файл не требуется. В нашем примере он будет иметь такой вид:

```
bob=Bob Jones <bob@company.com>
bob@localhost=Bob Jones <bob@company.com>
bob jones <bob <AT> company <DOT> com>=Bob Jones <bob@company.com>
bob <bob@company.com>=Bob Jones <bob@company.com>
```

Теперь нужно создать новый Git-репозиторий и запустить сценарий экспорта:

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

Флаг `-r` указывает команде `hg-fast-export`, где находится подлежащий преобразованию репозиторий, а флаг `-A` указывает место хранения файла с проекциями имен авторов. Этот сценарий анализирует Mercurial-наборы изменений и преобразует их в сценарий для Git-функции, которая называется `fast-import` (подробно она рассматривается чуть позже). Преобразование займет некоторое время (не слишком большое), и мы получим достаточно подробный результат:

```
$ /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
```

На этом, собственно, и все. Все Mercurial-теги преобразованы в Git-теги, а Mercurial-ветки и Mercurial-закладки превратились в Git-ветки. Теперь можно отправить репозиторий на новый сервер:

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin -all
```

Perforce

Следующей системой, для которой мы рассмотрим процесс импорта, является Perforce. Как уже обсуждалось, взаимодействие Git и Perforce реализуется посредством двух программ: git-p4 and Git Fusion.

Git Fusion

Приложение Git Fusion делает процесс практически безболезненным. Достаточно в конфигурационном файле указать параметры проекта, проекции пользователей и ветки (как было показано ранее в разделе «Программа Git Fusion»), а затем клонировать репозиторий. После этого вы получите настоящий Git-репозиторий, готовый к отправке на Git-сервер. В качестве такового при желании может использоваться даже Perforce-сервер.

Git-p4

Программа git-p4 также может использоваться как инструмент импорта. В качестве примера рассмотрим импорт проекта Jam из открытого Perforce-депо. Для настройки клиента нужно экспортировать переменную окружения P4PORT, которая будет указывать на Perforce-депо:

```
$ export P4PORT=public.perforce.com:1666
```

ПРИМЕЧАНИЕ

Для выполнения остальных операций вам потребуется подключение к Perforce-депо. В наших примерах используется открытое депо на сайте public.perforce.com, вы же можете указать любое доступное вам депо.

Для импорта проекта с Perforce-сервера воспользуйтесь командой `git p4 clone`, указав ей путь к депо и к проекту, а также к месту, куда вы планируете поместить проект:

```
$ 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%)
```

В данном случае проект содержит всего одну ветку, но при наличии нескольких веток, конфигурация которых задана функцией `branch views` (то есть представляет собой набор папок), достаточно воспользоваться флагом `--detect-branches`, реализующим импорт всех веток проекта. Подробно этот аспект рассматривался ранее в разделе «Мост Git-p4» данной главы.

Теперь наш репозиторий практически готов. Зайдя в папку `p4import` и запустив команду `git log`, вы увидите результат импорта:

```
$ 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]
```

Как видите, программа `git-p4` оставила во всех сообщениях фиксации свои идентификаторы. Их можно оставить на случай, если в будущем потребуется сделать в системе Perforce ссылку на номер изменений. Но если вы предпочитаете избавиться от них, самое время сделать это сейчас, пока не началась работа с новым репозиторием. Удалим все идентификаторы командой `git filter-branch`:

```
$ git filter-branch --msg-filter ,sed -e „/^[git-p4:/d"
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref ,refs/heads/master' was rewritten
```

Теперь команда `git log` покажет изменение контрольных сумм SHA-1 у всех коммитов, при том что добавленных инструментом `git-p4` строк в сообщениях фиксации вы больше не найдете:

```
$ 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 3e68c2e26cd89cb983eb52c024ecd6b1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).
```

Итак, импортированный репозиторий готов к отправке на новый сервер.

TFS

Если ваша рабочая группа решила передать процесс управления версиями от TFVC к Git, процесс перехода должен выполняться с максимально доступной точностью. Поэтому хотя ранее мы познакомились с двумя инструментами — `git-tfs` и `git-tf`, сейчас речь пойдет только о `git-tfs`, так как именно он поддерживает ветки. Кроме того, импорт средствами `git-tf` представляет собой чрезмерно сложную процедуру.

ПРИМЕЧАНИЕ

В данном случае речь идет о необратимом преобразовании. Полученный в итоге Git-репозиторий подключить к исходному TFVC-проекту вы уже не сможете.

Первым делом следует выполнить проецирование имен пользователей. Система TFVC достаточно вольно подходит к данным, сохраняемым для наборов изменений в поле `author`, в то время как Git требует удобочитаемого имени и адреса электронной почты. Получить эту информацию можно с помощью клиента командной строки `tf`:

```
PS> tf history $/myproject -recursive | cut -b 11-20 | tail -n+3 | uniq | sort >
AUTHORS
```

В данном случае мы рассматриваем все наборы изменений в истории проекта. Команда `cut` игнорирует все, кроме символов 11–20 из каждой строки (в данном случае для получения корректных цифр вам придется поэкспериментировать с длиной строк). Команда `tail` пропускает первые две строки, содержащие заголовки полей и подчеркивания, выполненные ASCII-графикой. Затем все данные передаются команде `uniq`, которая убирает дубликаты, а конечный результат сохраняется в файле `AUTHORS`. Дальше приходится действовать вручную; инструмент `git-tfs` может результативно пользоваться этим файлом только в случае, когда каждая строка имеет вот такой формат:

```
DOMAIN\username = User Name <email@address.com>
```

Слева от знака равенства мы видим содержимое поля `User` в системе TFVC, в то время как справа фигурирует имя пользователя, которое применяется в Git-коммитах.

Теперь нужно выполнить клонирование интересующего нас TFVC-проекта:

```
PS> git tfs clone --with-branches --authors=AUTHORS https://username.visualstudio.com/
DefaultCollection $/project/Trunk project_git
```

Затем вы, возможно, захотите почистить разделы `git-tfs-id` в нижней части сообщений фиксации. Это делается следующей командой:

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$/g"' -- --all
```

В данном случае команда `sed` для среды `Git-bash` заменяет все строки, начинающиеся с префикса `git-tfs-id:`, пустой строкой, которую система Git впоследствии успешно игнорирует.

После этого остается только добавить новый удаленный репозиторий, отправить туда все ветки и приступить к работе с Git.

Другие варианты импорта

Если вы пользуетесь системой контроля версий, не входящей в число рассмотренных ранее, можно поискать средство импорта в Интернете. Существуют качественные программы для множества систем, в том числе CVS, Clear Case, Visual Source Safe и даже обычной папки с архивами. Если же ни одна из готовых программ вам не подойдет, потому что вы работаете с малоизвестной системой контроля версий или вам требуется больший контроль над процедурой импорта, остается команда **git fast-import**. Она читает простые инструкции из стандартного потока ввода и записывает определенные данные в Git. Это намного более простой способ создания Git-объектов, чем запуск низкоуровневых Git-команд или попытка записи низкоуровневых объектов (более подробно эта тема рассматривается в главе 10). В данном же случае можно написать сценарий импорта, читающий из исходной системы нужную информацию и отправляющий четкие инструкции в стандартный поток вывода. После чего остается запустить этот сценарий и передать результат его работы непосредственно команде **git fast-import**.

Рассмотрим написание простой программы импорта. Предполагается, что мы работаем в папке **current** и периодически делаем резервные копии проекта в папках, имена которых строятся по принципу **back_ГГГГ_ММ_ДД**.

Итак, у нас возникла необходимость импортировать эти данные в Git. Вот структура наших папок:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

Чтобы импортировать папку Git, следует освежить в памяти способ хранения данных в Git. Как вы, возможно, помните, система Git, по сути, представляет собой связанный список объектов-коммитов, каждый из которых указывает на снимок состояния данных. Поэтому нам нужно всего лишь указать команде **fast-import** снимки состояния, входящие в них данные коммитов и порядок их следования. Мы по очереди рассмотрим все снимки и создадим коммиты с содержимым каждой папки, связывая каждый следующий коммит с предыдущим.

Как и в главе 8, мы напишем сценарий на языке Ruby, потому что обычно пишем именно на этом языке, кроме того, он прост для понимания. Вы же можете воспользоваться своим любимым языком программирования, ведь нам требуется только включить нужную информацию в стандартный поток вывода.

Кроме того, тем, кто работает в операционной системе Windows, следует особо позаботиться о том, чтобы в конце строк не появлялись символы возврата каретки. Дело в том, что команда `git fast-import` работает только с символами перевода строки (LF).

Первым делом зайдем в исходную папку и определимся с подпапками, содержащими снимки состояния, которые мы хотим импортировать как коммиты. Мы будем по очереди заходить в каждую такую подпапку и запускать команды экспорта. Вот главный базовый цикл:

```
last_mark = nil

# по очереди просматриваем папки
Dir.chdir(ARGV[0]) do
  Dir.glob("**").each do |dir|
    next if File.file?(dir)

    # переход в целевую папку
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

Внутри каждой папки запускается метод `print_export`, который берет манифест и метку предыдущего снимка состояния и возвращает манифест и метку текущего; именно это обеспечивает корректность их связывания. Термин «метка» в контексте команды `fast-import` означает идентификатор, который вы присваиваете коммиту в момент создания и который впоследствии используется для связи с другими коммитами. Соответственно первым действием метода `print_export` будет генерация метки из имени папки:

```
mark = convert_dir_to_mark(dir)
```

Мы создадим массив папок, взяв в качестве метки значение индекса, так как в роли метки может выступать только целое число. Наш метод мог бы выглядеть так:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Теперь, когда у нас есть целочисленное представление нашего коммита, необходимо получить дату для его метаданных. Эта информация извлекается из названий папок, поэтому соответствующая строка в файле `print_export` выражается следующим образом:

```
date = convert_dir_to_date(dir)
```

Здесь метод `convert_dir_to_date` определен так:

```
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 метод возвращает целочисленное значение даты для каждой из папок. Последним фрагментом метаданных, которые требуются в каждом коммите, являются сведения о создавшем его человеке. Эта информация берется из глобальной переменной:

```
$author = 'John Doe <john@example.com>'
```

Все готово к вводу данных коммита в наш сценарий импорта. Исходная информация указывает, что вы создаете объект-коммит в определенной ветке, далее следует сгенерированная нами метка, информация о создателе коммита и сообщение фиксации, а затем — ссылка на предыдущий коммит, если таковой существует в природе. В итоге у нас получается вот такой код:

```
# вывод импортируемой информации
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
```

Временную зону (-0700) мы для простоты указали прямо в коде. При импорте из другой системы следует задать ее как смещение. Сообщение фиксации должно быть отформатировано следующим образом:

```
data (size)\n(contents)
```

Первым идет слово **data**, затем указываются длина сообщения, знак новой строки и, наконец, само сообщение. Так как позднее в этом формате нужно будет задавать и содержимое файла, создадим вспомогательный метод **export_data**:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Осталось только указать содержимое файлов для каждого снимка состояния. Это не сложно, так как все они хранятся в отдельных папках. Достаточно воспользоваться командой **deleteall**, за которой указывается содержимое всех файлов в папке. После этого система Git сможет корректно записать снимки состояния:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

ПРИМЕЧАНИЕ

Так как многие системы воспринимают версии данных как наборы изменений, сделанных между двумя коммитами, команда `fast-import` может для каждого коммита использовать дополнительные команды, указывающие, какие файлы были добавлены, удалены или отредактированы и как выглядит их новое содержимое. Можно рассчитать разницу между снимками состояния и предоставить только эти данные, но это сложнее. Хотя если вам больше подходит такой вариант, подробности его реализации можно узнать на странице справочника по команде `fast-import`.

Вот формат для перечисления нового содержимого файла или указания модифицированного файла с новым содержимым:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Число 644 здесь указывает на права доступа к файлу (в случае исполняемых файлов следует распознать этот факт и передать значение 755). Слово `inline` сообщает, что после этой строки пойдет вывод содержимого файла. В итоге метод `inline_data` будет выглядеть так:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

Мы повторно используем заданный ранее метод `export_data`, так как в данном случае форматирование выполняется тем же способом, что и в сообщениях фиксации.

После этого остается только вернуть текущую метку, чтобы ее можно было передать следующей итерации:

```
return mark
```

ПРИМЕЧАНИЕ

Для пользователей операционной системы Windows существует еще один этап. Как уже упоминалось, Windows использует для перевода на новую строку управляющие символы CRLF, в то время как команда `git fast-import` понимает только LF. Для решения этой проблемы достаточно дать интерпретатору языка Ruby команду использовать символы LF вместо CRLF:

```
$stdout.binmode
```

Вот и все. Это код нашего сценария:

```
#!/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

  # По очереди просматриваем папки
  last_mark = nil
  Dir.chdir(ARGV[0]) do
    Dir.glob("**").each do |dir|
      next if File.file?(dir)

      # переходим в целевую папку
      Dir.chdir(dir) do
        last_mark = print_export(dir, last_mark)
      end
    end
  end
end

```

Запустив этот сценарий, мы получим примерно такой результат:

```
$ 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
(...)
```

Для запуска сценария импорта направьте этот вывод команде **git fast-import**, находясь в папке **Git**, в которую вы хотите поместить результаты импорта. Можно создать новую папку, выполнить в ней команду **git init**, а затем запустить наш сценарий:

```
$ 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 видите, после успешного завершения появляется множество статистических данных, информирующих нас о проделанных операциях. В данном случае импорту подверглось 13 объектов для четырех коммитов в одной ветке. Теперь можно воспользоваться командой `git log` и посмотреть новую историю:

```
$ 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

Итак, у нас появился прекрасный, чистый Git-репозиторий. Обратите внимание, что данные пока не скачивались — рабочая папка изначально пуста. Для получения данных нужно установить нашу ветку туда, где в настоящее время находится ветка `master`:

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

Инструмент `fast-import` позволяет делать еще множество вещей — обрабатывает различные права доступа, бинарные данные, наборы веток и слияния, теги, индикаторы выполнения и многое другое. Ряд более сложных сценариев для этой команды доступен в папке `contrib/fast-import` исходного Git-кода.

Закключение

Теперь вы умеете пользоваться системой Git как клиентом для остальных систем контроля версий и импортировать в Git практически любой существующий репозиторий без потери данных. В следующей главе мы познакомимся с особенностями низкоуровневого функционирования системы Git, что даст вам возможность контролировать каждый байт данных, если возникнет такая необходимость.

10 Git изнутри

Вы могли бы сразу перейти к этой главе, пропустив остальной материал, а может быть, вы добрались сюда в процессе вдумчивого чтения — как бы то ни было, именно тут мы рассмотрим внутреннюю работу и особенности реализации системы Git. Оказалось, что изучение именно этой информации позволяет понять, насколько полезным и мощным инструментом является Git, хотя некоторые считают, что попытка углубиться во все эти детали неоправданно сложна для новичков и только все запутывает. Поэтому эта глава и оказалась в самом конце книги, и вы можете самостоятельно выбрать, на каком этапе процесса обучения она вам потребуется.

Впрочем, приступим к делу. Сначала напомним, что Git — это в своей основе контентно-адресуемая файловая система, на которую наложен пользовательский интерфейс от VCS. Что это означает вы узнаете чуть позже.

В ранние годы своего существования (в основном до версии 1.5) пользовательский интерфейс системы Git был намного сложнее, так как был призван подчеркивать, что перед нами файловая система, а не законченная VCS. За последние годы интерфейс значительно улучшился и по удобству не уступает другим системам; но зачастую до сих пор можно столкнуться со стереотипным мнением, что пользовательский интерфейс системы Git запутан и труден в освоении.

Слой контентно-адресуемой файловой системы — это именно та замечательная вещь, которую мы рассмотрим в первую очередь; затем вы познакомитесь с транспортными механизмами и обслуживанием репозитория, с чем вам, возможно, придется сталкиваться в процессе использования Git.

Канализация и фарфор

В этой книге описывается, как работать с Git при помощи трех десятков команд, таких как `checkout`, `branch`, `remote` и т. п. Но так как система Git изначально представляла собой не VCS с удобным интерфейсом, а скорее инструментарий для создания VCS, Git предлагает множество команд для низкоуровневой работы, а также поддерживает использование конвейера в стиле UNIX и вызов из сценариев. Эти служебные команды иногда сравнивают с водопроводно-канализационными трубами (*plumbing*), а более удобные для пользователей команды — с фарфоровыми унитазами и раковинами (*porcelain*), облегчающими нам доступ к канализации.

В первых девяти главах мы рассматривали преимущественно «фарфоровые» команды. Теперь же пришло время более подробно познакомиться с низкоуровневыми командами, которые предоставляют контроль над внутренними процессами системы Git, помогая продемонстрировать, как функционирует Git и почему это происходит так, а не иначе. Многие из этих команд не предназначены для непосредственного вызова из командной строки, а являются строительными кирпичиками для новых инструментов и пользовательских сценариев.

При выполнении команды `git init` из новой или существующей папки система Git создает папку `.git`, в которой в дальнейшем сохраняются почти все используемые данные. При резервировании или клонировании репозитория достаточно скопировать эту папку, чтобы получить практически все необходимое. В этой главе мы в основном будем работать с содержимым этой папки. Вот как она выглядит:

```
$ ls -F1
HEAD
config*
description
hooks/
info/
objects/
refs/
```

Там могут находиться и другие файлы, но в данном случае перечислено только то, что по умолчанию оказывается в ней после выполнения команды `git init`. Файл `description` используется только программой GitWeb, поэтому на него можно не обращать внимания. Файл `config` содержит конфигурационные параметры, связанные с вашим проектом, а в папке `info` хранится глобальный файл исключений с игнорируемыми шаблонами, которые вы не хотите помещать в файл `.gitignore`. Папка `hooks` содержит сценарии хуков, работающих как на стороне клиента, так и на стороне сервера. Хуки подробно обсуждались в разделе «Git-хуки» главы 8.

Осталось четыре важных элемента: файл `HEAD`, еще не созданный файл `index` и папки `objects` и `refs`. Это ключевые элементы системы Git. В папке `objects` находится все содержимое вашей базы данных, папка `refs` хранит указатели на объекты-коммиты в этой базе (ветки), файл `HEAD` указывает, в какой ветке вы сейчас находитесь, а в файле `index` система Git хранит информацию из области индексирования. Далее мы детально рассмотрим все эти элементы, чтобы понять, как функционирует Git.

Объекты в Git

Система Git представляет собой контентно-адресуемую файловую систему. Но что это означает на практике? Это означает, что в своей основе Git является хранилищем данных вида «ключ-значение». В нее можно добавить любое содержимое, а в ответ вы получите ключ, по которому это содержимое позднее можно будет извлечь, когда оно вам понадобится. Для демонстрации этого принципа воспользуемся служебной командой **hash-object**, которая сохраняет данные в папке **.git** и возвращает ключ. Первым делом инициализируем новый репозиторий Git и удостоверимся, что в папке **objects** ничего нет:

```
$ 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
```

Система Git инициализировала папку **objects**, создав внутри нее папки **pack** и **info**, но обычные файлы в ней пока отсутствуют. Добавим в базу данных системы Git какой-нибудь текст:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Параметр **-w** заставляет команду **hash-object** сохранить объект; без нее команда просто возвращает вам ключ. Параметр **--stdin** заставляет команду читать содержимое из стандартного потока ввода; без этого параметра команда **hash-object** будет искать путь к файлу с содержимым. Выводимые командой данные представляют собой контрольную хеш-сумму, состоящую из 40 символов. Это хеш SHA-1 — контрольная сумма сохраняемого вами содержимого плюс заголовок, о котором мы поговорим чуть позже. Теперь можно посмотреть, в каком виде Git сохраняет ваши данные:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

В папке **objects** появился файл. Именно так Git изначально сохраняет содержимое — один файл на единицу хранения с именем, которое представляет собой контрольную сумму SHA-1 содержимого и его заголовок. Подпапка именуется первыми двумя символами SHA, а имя файла формируют остальные 38 символов.

Извлечь содержимое обратно позволяет команда **cat-file**. Ее можно сравнить со швейцарским армейским ножом для вскрытия Git-объектов. Параметр **-p** заставляет эту команду определить тип содержимого и корректно его воспроизвести:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Теперь вы можете добавлять данные в систему Git и извлекать их оттуда. Эти операции доступны и для содержимого файлов. Рассмотрим пример управления версиями файла. Первым делом создадим новый файл и сохраним его в базе данных:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Запишем в этот файл какие-то новые данные и снова его сохраним:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Теперь в базе данных есть две новые версии этого файла, а также его исходное содержимое:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Мы можем вернуть файл к его первой версии:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

А можем вернуть его ко второй версии:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Однако запомнить ключ SHA-1 для каждой версии файла нереально; кроме того, в системе сохраняется не имя файла, а только его содержимое. Объекты такого типа называют массивами двоичных данных, или большими бинарными объектами (Binary Large Object, blob). Можно попросить систему Git по контрольной сумме SHA-1 определить тип любого объекта. Для этого используется команда `cat-file -t`:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Объекты-деревья

Рассмотрим другой тип объекта — дерево (tree). Объекты этого типа решают проблему хранения имен файлов, а также позволяют хранить группы файлов. Способ хранения содержимого в Git во многом такой же, как и в файловой системе UNIX, но несколько упрощенный. Вся информация хранится в виде объектов-деревьев

и blob-объектов, причем первые соответствуют записям UNIX-каталога, а вторые больше напоминают индексные дескрипторы, или содержимое файлов. Объект-дерево может содержать одну или несколько записей, каждая из которых включает в себя указатель SHA-1 на двоичный массив данных или на поддерево со связанными с ним правами доступа, типом и именем файла. Например, дерево последнего коммита в проекте может выглядеть так:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859 README
100644 blob 8f94139338f9404f26296befa88755fc2598c289 Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0 lib
```

Синтаксис `master^{tree}` определяет объект-дерево, на который указывает последний коммит в ветке `master`. Обратите внимание, что подпапка `lib` представляет собой не двоичный массив данных, а указатель на другое дерево:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b simplegit.rb
```

По существу, данные, которые сохраняет Git, выглядят так, как показано на рис. 10.1.

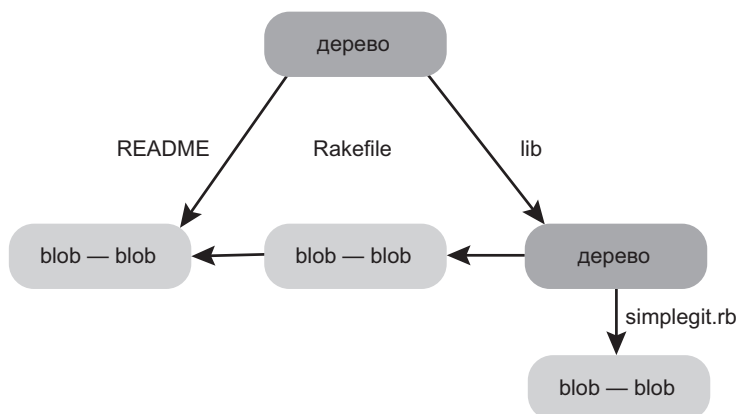


Рис. 10.1. Упрощенная версия модели данных в Git

Вы можете легко создать собственное дерево. Обычно Git берет состояние области индексирования и записывает оттуда набор объектов-деревьев. Поэтому для создания дерева первым делом нужно проиндексировать какие-либо файлы. Чтобы получить индекс с одной записью — первой версией файла `text.txt`, — воспользуемся служебной командой `update-index`. Она позволит нам принудительно поместить более раннюю версию файла `text.txt` в новую область индексации. Следует добавить к команде параметр `--add`, так как файл еще не индексирован (более того, у нас пока отсутствует настроенная область индексирования), а также параметр `--cacheinfo`, потому что добавляемый файл находится не в папке, а в базе данных. Затем мы указываем права доступа, контрольную сумму SHA-1 и имя файла:

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

В данном случае заданные права доступа 100644 означают обычный файл. Вариант 100755 соответствует исполняемому файлу, а 120000 — символической ссылке. Здесь мы имеем дело с аналогом прав доступа в операционной системе UNIX, но намного менее гибким — упомянутые три варианта являются единственными доступными для файлов (массивов двоичных данных) в Git (хотя для папок и подмодулей допустимы и другие варианты прав доступа).

Теперь с помощью команды **write-tree** добавим область индексирования к нашему объекту-дереву. Параметр **-w** в данном случае не требуется — вызов команды **write-tree** автоматически создает объект-дерево из состояния области индексирования, если такого дерева пока не существует:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Удостоверимся в том, что мы действительно получили дерево:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

Создадим новое дерево со второй версией файла **test.txt** и новым файлом:

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

В области индексирования появилась новая версия файла **test.txt**, а также новый файл **new.txt**. Запишем это дерево (сохранив состояние области индексирования в объекте-дереве) и посмотрим, что из этого получилось:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
```

Обратите внимание, что наше дерево содержит записи для обоих файлов, а контрольная сумма SHA файла **test.txt** представляет собой SHA «версии 2», которая появлялась раньше (**1f7a7a**). Просто из интереса добавим первое дерево как подпапку для текущего. Чтение деревьев в область индексирования осуществляется командой **read-tree**. Так как в данном случае нам нужно прочитать существующее дерево как поддереву, потребуется параметр **--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 создать рабочую папку из только что записанного дерева, мы получим два файла в корне и подпапку **bak** с первой версией файла **test.txt**. Данные, которые система Git содержит для таких структур, можно представить так, как показано на рис. 10.2.

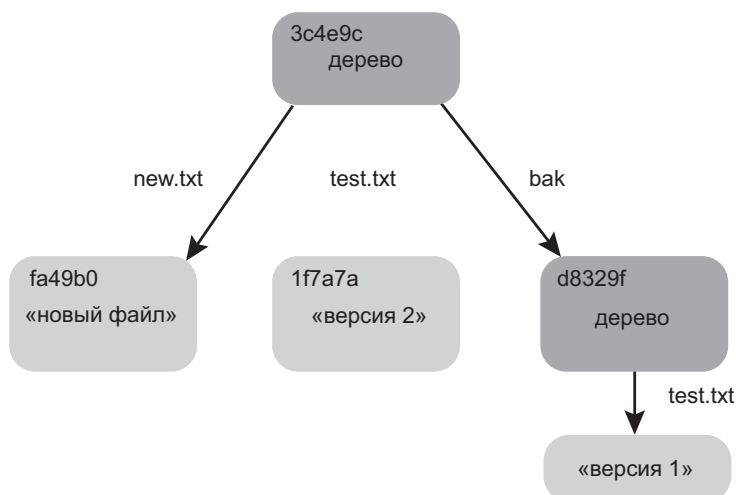


Рис. 10.2. Структура данных для текущего дерева в Git

Объекты-коммиты

У нас есть три дерева, соответствующие трем состояниям проекта, которые мы хотим отслеживать. Но это не решило стоящую перед нами проблему. Нам по-прежнему приходится запоминать все три значения SHA-1, чтобы иметь возможность доступа к снимкам состояний. К тому же у нас нет информации о том, кто сохранил эти состояния, когда это произошло и почему они были сохранены. Это именно та базовая информация, которую сохраняет для вас объект-коммит.

Создать объект-коммит позволяет команда **commit-tree**, которой мы сообщаем контрольную сумму SHA-1 нужного дерева и непосредственно предшествующие этому дереву объекты-коммиты, если таковые существуют. Начнем с первого записанного нами дерева:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Теперь посмотрим на полученный объект с помощью команды **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: указываются дерево верхнего уровня для состояния проекта на определенный момент времени; сведения об авторе/создателе коммита (они берутся из конфигурационных настроек `user.name` и `user.email`) и временная метка; а также пустая строка, за которой следует сообщение фиксации.

Создадим еще два объекта-коммита, каждый из которых будет ссылаться на предыдущий коммит:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Каждый из этих трех объектов-коммитов указывает на одно из трех созданных нами деревьев для снимков состояний. Как ни странно это звучит, но теперь у нас есть полноценная Git-история, которую можно увидеть с помощью команды `git log`, указав хеш SHA-1 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(+)
```

Поразительно! Мы только что создали Git-историю с помощью исключительно низкоуровневых операций, не прибегая к командам пользовательского уровня. По сути, именно это продельывает система Git в ответ на команды `git add` и `git commit` — сохраняет двоичные массивы данных для измененных файлов, обновляет область индексирования, записывает деревья, объекты-коммиты, ссылающиеся на деревья верхнего уровня, а также коммиты, которые им предшествуют. Эти три основных вида Git-объектов — массивы двоичных данных, деревья и коммиты — изначально хранятся как отдельные файлы в папке `.git/objects`. Вот перечень объектов из рассматриваемой в данном примере папки с комментариями о том, что хранится внутри:

```
$ 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
```

Проследив за внутренними указателями, получаем соответствующий граф объектов (рис. 10.3).

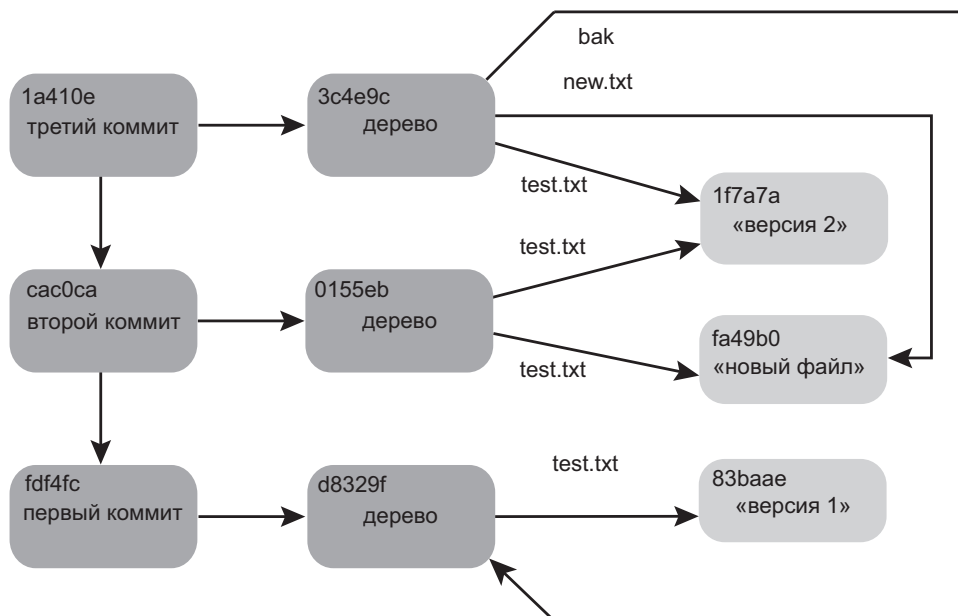


Рис. 10.3. Все объекты в вашей папке Git

Хранение объектов

Как уже упоминалось, заголовок сохраняется вместе с содержимым. Давайте более подробно рассмотрим, каким образом Git сохраняет свои объекты. Сохраним в интерактивном режиме массив двоичных данных — строку «что нового, доктор?», — воспользовавшись для этого языком Ruby.

Интерактивный режим интерпретатора Ruby запускает команда `irb`:

```
$ irb
>> content = "что нового, доктор?"
=> "что нового, доктор?"
```

Система Git конструирует заголовок, начинающийся с типа объекта — в данном случае это массив двоичных данных (**blob**). Затем добавляется пробел, указывается размер содержимого и, наконец, нулевой байт:

```
>> header = "blob #{content.length}\0"
=> "blob 16\u0000"
```

Система Git склеивает заголовок и исходное содержимое, а затем вычисляет для новых данных контрольную сумму SHA-1. В языке Ruby значение SHA-1 строки вычисляется путем подключения соответствующей библиотеки командой `require` и вызовом для строки метода `Digest::SHA1.hexdigest()`:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Для сжатия новых данных Git пользуется библиотекой **zlib**. Поэтому первым делом нужно подключить эту библиотеку, а потом вызвать для нашего содержимого функцию `Zlib::Deflate.deflate()`:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\x9CK\xCA\xC90R04c(\xCFH,Q\xC8,V(-\xD0QH\xC90\xB6\a\x00_\x1C\a\x9D"
```

Наконец, запишем сжатое библиотекой **zlib** содержимое в объект на диске. Укажем путь к объекту, в котором мы хотим записать данные (первые два символа контрольной суммы SHA-1 являются именем подпапки, а остальные 38 — именем файла внутри этой папки). В языке Ruby создать такую подпапку, если у нас ее пока нет, поможет функция `FileUtils.mkdir_p()`. После чего мы откроем этот файл функцией `File.open()` и запишем туда ранее сжатое библиотекой **zlib** содержимое, вызвав функцию `write()` для дескриптора итогового файла:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
```

```
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

Вот вы и создали в Git настоящий массив двоичных данных. Все Git-объекты хранятся одинаково, меняется только тип: вместо строки **blob** заголовок будет начинаться со строки **commit** или **tree**. Кроме того, в массив двоичных данных можно поместить практически любую информацию, а вот содержимое коммитов и деревьев форматируется особым образом.

Ссылки в Git

Команда **git log 1a410e** даст вам возможность целиком посмотреть историю, пройдясь по всем коммитам, но для этого вы в любом случае должны помнить, что последний коммит носит имя **1a410e**. Нам нужен файл, в котором значение SHA-1 можно было бы сохранить под простым именем и впоследствии указывать имя этого файла, а не саму контрольную сумму.

В Git такие файлы называются ссылками (references, или refs) и хранятся в папке **.git/refs**. В рассматриваемом проекте эта папка пока пуста, но в ней уже существует простая структура:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

Создание новой ссылки, которая поможет запомнить, где именно находится последний коммит, с технической точки зрения выглядит просто:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Теперь в Git-командах вместо контрольной суммы SHA-1 можно указывать ссылку, только что созданную нами в папке **head**:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Но напрямую редактировать файлы ссылок не рекомендуется. Для обновления ссылок Git предоставляет безопасную команду **update-ref**:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Именно это, по сути, представляет собой ветка в Git — простой указатель, или ссылка, на начало рабочей линии. Чтобы создать ветку, начинающуюся из второго коммита, напишите:

```
$ git update-ref refs/heads/test cac0ca
```

Эта ветка будет содержать только коммиты, появившиеся после указанного:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Теперь наша база данных Git должна выглядеть так, как показано на рис. 10.4.

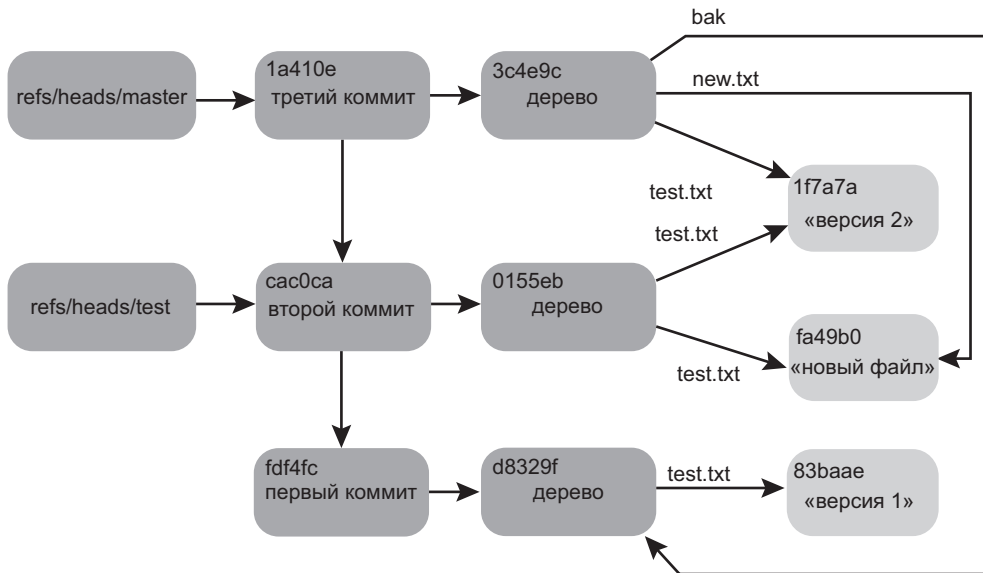


Рис. 10.4. Папка Git-объектов со ссылками на вершины веток

Такая команда, как `git branch` (имя ветки), заставляет Git выполнить команду `update-ref`, чтобы добавить контрольную сумму SHA-1 последнего коммита ветки, в которой вы находитесь, в создаваемую вами новую ссылку.

Указатель HEAD

Вопрос в том, каким образом при выполнении команды `git branch` (имя ветки) Git узнает контрольную сумму SHA-1 последнего коммита? Ответом является файл HEAD.

Этот файл представляет собой символическую ссылку на вашу текущую ветку. Символическая ссылка в отличие от обычной вместо контрольной суммы SHA-1 содержит указатель на другую ссылку. Вот как выглядит содержимое такого файла:

```
$ cat .git/HEAD
ref: refs/heads/master
```

По команде **git checkout test** система Git обновит файл, и он начнет выглядеть так:

```
$ cat .git/HEAD
ref: refs/heads/test
```

Команда **git commit** создает объект-коммит, указывая в качестве его предка коммит, на контрольную сумму SHA-1 которого указывает ссылка в файле **HEAD**.

Этот файл можно редактировать вручную, но существует и более безопасная команда **symbolic-ref**. Она позволяет узнать значение вашего указателя **HEAD**:

```
$ git symbolic-ref HEAD
refs/heads/master
```

Вы также можете задать значение **HEAD**:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

При этом символическую ссылку, ведущую за пределы папки **refs**, создать невозможно:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

Теги

Мы обсудили три основных типа Git-объектов, но существует еще один. Объект-тег во многом напоминает объект-коммит — он содержит имя автора тега, дату его создания, комментарий и указатель. Основное отличие состоит в том, что объект-тег, как правило, указывает на коммит, а не на дерево. Тег напоминает ссылку на ветку, которая никогда не перемещается, — она всегда нацелена на один и тот же коммит, предоставляя ему более понятное имя.

Как вы уже знаете из главы 2, существует два типа тегов: снабженные комментарием и легковесные. Вторые создаются такой командой:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Именно так выглядит легковесный тег — ссылка, которая никогда не перемещается. Тег с комментарием имеет более сложную структуру. При его создании Git создает

объект-тег и записывает ссылку, которая указывает на этот тег, а не непосредственно на коммит. Посмотрим, как это выглядит на практике (флаг `-a` указывает, что нам требуется тег, снабженный комментарием):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Вот значение контрольной суммы SHA-1 созданного объекта:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Выполним для этого значения команду `cat-file`:

```
$ 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 запись `object` указывает на значение SHA-1 коммита, для которого создавался тег. Но в принципе она может указывать не только на коммит; создать тег можно для любого Git-объекта. К примеру, в исходный Git-код человек, отвечающий за его поддержку, добавил свой открытый ключ GPG как blob-объект и затем создал для этого объекта тег. Увидеть этот открытый ключ можно, запустив в клоне Git-репозитория команду:

```
$ git cat-file blob junio-gpg-pub
```

В репозитории ядра Linux также существует тег, указывающий не на коммит, — первый созданный тег указывает на основное дерево импорта исходного кода.

Удаленные ветки

Третий тип ссылок, о которых вам следует знать, — это ссылки на удаленные ветки. После добавления удаленного репозитория и отправки туда изменений Git сохраняет контрольные суммы отправленных во все ветки данных в папке `refs/remotes`. Например, добавим удаленный репозиторий `origin` и отправим туда содержимое ветки `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
```

Теперь вы можете определять состояние ветки **master** на сервере **origin** во время последнего подключения к этому серверу, просто заглянув в файл **refs/remotes/origin/master**:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Ссылки на удаленные ветки отличаются от обычных ссылок вида **refs/heads** в основном тем, что считаются предназначенными только для чтения. Вы можете выполнить команду **git checkout**, но Git не нацелит указатель **HEAD** на выбранную вами удаленную ветку и вы не сможете обновлять ее командой **commit**. Система Git работает с ними как с закладками на последнее известное состояние указанных веток на ваших серверах.

Реск-файлы

Вернемся к базе данных объектов нашего тестового Git-репозитория. К этому моменту там должно находиться 11 объектов, включая 4 двоичных массива данных, 3 дерева, 3 коммита и 1 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/95/85191f37f7b0fb9444f35a9bf50de191bead2 # 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
```

Содержимое этих файлов было сжато с помощью библиотеки **zlib**, к тому же информации пока немного, соответственно общий объем всех файлов составляет всего 925 байт. Для демонстрации одной интересной функциональной особенности Git добавим в репозиторий более объемный файл. Это будет файл **repo.rb** из библиотеки **Grit**, его исходный код «весит» примерно 22 Кбайт:

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb >
  repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 709 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

Внимательно посмотрев на полученное дерево, мы увидим контрольную сумму SHA-1, которую файл `repo.rb` получил после преобразования в blob-объект:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b test.txt
```

Узнаем размер этого объекта с помощью команды `git cat-file`:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Теперь внесем в этот файл небольшие изменения и посмотрим, что из этого получится:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master 2431da6] modified repo a bit
1 file changed, 1 insertion(+)
```

Проверив созданное этим коммитом дерево, мы увидим кое-что интересное:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b test.txt
```

У нас появился другой двоичный массив данных. То есть после добавления в конец 400-строчного файла единственной строки система Git сохранила новое содержимое как новый объект:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

У нас на диске есть два почти одинаковых объекта объемом по 22 Кбайт каждый. Разве не здорово было бы, если бы система Git сохраняла один из объектов целиком, а второй — как разницу между объектами?

Оказывается, такое возможно. Изначально Git сохраняет объекты на диске в так называемом свободном (loose) формате. Но время от времени такие объекты упаковываются в один бинарный файл для экономии места на диске и повышения эффективности. Это происходит автоматически, когда объектов в «свободном» формате становится слишком много, кроме того, можно вручную вызвать команду `git gc` или отправить данные на удаленный сервер. Посмотрим, как это происходит. Для этого попросим систему Git упаковать наши объекты, воспользовавшись командой `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)
```

Заглянув в папку с объектами, вы обнаружите, что большая часть объектов исчезла, зато появилась пара новых файлов:

```
$ 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 представляют собой двоичные массивы данных, на которые не указывает ни один коммит, — в нашем случае это созданная нами ранее строка «что нового, доктор?» и блоб с «тестовым содержимым». Так как эти файлы не добавлены ни в один коммит, они считаются «висячими» и не упаковываются вместе с остальными.

Остальные файлы — это наш новый pack-файл и индекс. Pack-файл теперь включает в себя содержимое всех удаленных из вашей файловой системы объектов. А индекс представляет собой файл, в котором записаны смещения данных внутри pack-файла, что позволяет быстро найти конкретный объект. Примечательно то, что объекты, занимавшие на диске примерно 22 Кбайт, после упаковки стали «весить» всего 7 Кбайт. То есть мы на две трети уменьшили занятое место на диске.

Каким образом Git это делает? При упаковке объектов система ищет файлы с похожими именами и примерно одного размера, после чего сохраняет только разницу между версиями. Можно заглянуть в pack-файл и посмотреть, что делает Git для экономии места. Для просмотра содержимого упакованного файла существует служебная команда `git verify-pack`:

```
$ git verify-pack -v .git/objects/pack/pack-978e03944f5c581011e6998cd0e
9e30000905586.idx
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcbda5f5328278ab1c0812ce0e07fa7d26a96d7 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
fe879577cb8cfcdf25441725141e310dd7d239b 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
    
```

В данном случае двоичный массив данных **033b4**, который, как вы помните, представляет собой первую версию файла **repo.rb**, ссылается на двоичный массив данных **b042a**, являющийся второй версией этого файла. Третий столбец в списке вывода содержит размер объекта в упакованном виде. Как видите, если **b042a** занимает 22 Кбайт, то **033b4** — всего 9 байт. Кроме того, примечательно, что вторая версия сохраняется в исходном виде, в то время как от оригинала остаются только данные о внесенных изменениях. Это делается потому, что чаще всего требуется доступ к последней версии файла.

Приятно то, что файл в любой момент можно перепаковать. Время от времени Git делает это со своей базой данных автоматически, чтобы сэкономить место, вы же для этого можете воспользоваться командой **git gc**.

Спецификация ссылок

В этой книге мы просто проецировали удаленные ветки на локальные, но варианты могут быть и более сложными. Предположим, вы добавили удаленный репозиторий:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

Эта команда добавляет в ваш файл **.git/config** раздел, задающий имя удаленного репозитория (**origin**), его URL-адрес и спецификацию ссылок для получения оттуда данных:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit
fetch = +refs/heads/*:refs/remotes/origin/*
```

Эта спецификация имеет следующий формат: по желанию символ **+**, за которым следуют **<источник>: <цель>**, где **<источник>** — шаблон для ссылок на сервере, а **<цель>** — место, куда эти ссылки будут записываться локально. Символ **+** заставляет Git обновлять ссылки, даже если речь идет не о перемотке.

В случае настроек, предлагаемых по умолчанию, которые автоматически используются командой `git remote add`, Git извлекает все ссылки из папки `refs/heads/` на сервере и записывает их в локальную папку `refs/remotes/origin/`. Соответственно если на сервере есть ветка `master`, локальный доступ к ее журналу можно получить так:

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Все эти команды дают один и тот же результат, так как Git разворачивает каждую из них до состояния `refs/remotes/origin/master`.

Если нужно, чтобы с удаленного сервера при каждом обращении к нему извлекалось содержимое только ветки `master`, а не всех веток подряд, измените соответствующую строку в конфигурационном файле:

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Эта спецификация ссылки, предлагаемой по умолчанию, будет использоваться командой `git fetch` при обращении к данному удаленному репозиторию. Если же вы хотите выполнить некое однократное действие, указать спецификацию можно и в командной строке. Чтобы извлечь данные из удаленной ветки `master` и поместить их в локальную ветку `origin/mymaster`, напишите:

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Можно задать и набор спецификаций. Извлечь данные из нескольких веток посредством 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 добавление данных из ветки `master` выполнить не удалось, так как здесь невозможно прибегнуть к перемотке. Данное поведение можно изменить, добавив перед спецификацией ссылки символ `+`.

Задать множественную спецификацию ссылок для извлечения данных можно и в конфигурационном файле. Чтобы все время получать обновления из веток `master` и `experiment`, 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
```

Использовать частичную маску нельзя. Следующая запись некорректна:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Впрочем, получить требуемый результат позволяют пространства имен (или папки). Если группа тестирования работает в некоторых ветках, а вам нужно получить данные из ветки **master** и из всех веток, используемых этой группой, добавьте в раздел конфигурации следующие строки:

```
[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/*
```

В случае сложного рабочего процесса с ветками, куда отправляет данные группа тестирования, ветками, куда отправляет данные группа разработчиков, и ветками, куда отправляет данные и в которых осуществляет совместную работу группа интеграции, вы можете легко поместить их в отдельные пространства имен.

Спецификация ссылок для отправки данных на сервер

Здорово, что извлекать данные можно с помощью ссылок, разделенных по пространствам имен, но сначала нужно дать группе тестирования возможность отправлять свои ветки в пространство имен **qa/**. Эта задача решается через спецификации ссылок для команды **push**.

Вот как отправить ветку **master**, принадлежащую группе тестирования, в ветку **qa/master** на удаленном сервере:

```
$ git push origin master:refs/heads/qa/master
```

Если нужно, чтобы система Git автоматически делала это при каждом запуске команды **git push origin**, добавьте в конфигурационный файл значение переменной **push**:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/*:refs/remotes/origin/*
    push = refs/heads/master:refs/heads/qa/master
```

В результате команда **git push origin** по умолчанию будет отправлять локальную ветку **master** в ветку **qa/master** на удаленном сервере.

Ликвидация ссылок

Спецификации ссылок, кроме всего прочего, можно использовать для удаления ссылок с сервера. Это может выглядеть так:

```
$ git push origin :topic
```

Так как спецификация задается в формате `<источник> : <цель>`, опустив часть `<источник>`, мы, по сути, скажем, что тематическую ветку на удаленном сервере следует сделать пустой, и это приведет к ее ликвидации.

Протоколы передачи данных

Передача данных между репозиториями осуществляется в Git двумя основными путями: по «простому» и по «интеллектуальному» протоколам. В этом разделе мы кратко рассмотрим принципы их работы.

Простой протокол

Если вы предоставляете к своему репозиторию доступ только на чтение по протоколу HTTP, скорее всего, вы воспользуетесь простым протоколом. Название «простой» было дано потому, что во время передачи данных не требуется исполнения на стороне сервера никакого характерного для Git кода. Процесс извлечения данных представляет собой набор HTTP-запросов `GET`, когда клиент предполагает, что имеет дело со стандартной структурой Git-репозитория на сервере.

ПРИМЕЧАНИЕ

Простой протокол в наши дни применяется крайне редко. При передаче данных по этому протоколу сложно обеспечить безопасность или сделать данные закрытыми, поэтому большинство Git-хостов (как облачных, так и локальных) отказывается с ним работать. В общем случае рекомендуется использовать «интеллектуальный» протокол, который мы рассмотрим чуть позже.

Рассмотрим процесс получения данных по протоколу HTTP для библиотеки `simplegit`:

```
$ git clone http://server/simplegit-progit.git
```

Первым делом эта команда загружает файл `info/refs`. Этот файл записывается командой `update-server-info`, поэтому для корректной передачи данных по протоколу HTTP его следует активировать как хук `post-receive`:

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949 refs/heads/master
```

Теперь у нас есть список удаленных ссылок и их контрольные суммы SHA. Нужно определить, куда нацелен указатель `HEAD`, чтобы вы знали, куда переключаться после завершения работы команды:

```
=> GET HEAD
ref: refs/heads/master
```

После завершения процесса следует перейти в ветку **master**, и все будет готово для обхода дерева. Так как отправной точкой является объект-коммит **ca82a6**, который мы наблюдали в файле **info/refs**, начнем мы именно с его загрузки:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

Вы получили свой объект. На сервере он находился в свободном формате, и вы извлекли его статическим HTTP-запросом **GET**. Теперь его можно извлечь из архива **zlib**, удалить заголовок и посмотреть на содержимое коммита:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfd3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Осталось затребовать еще два объекта — дерево данных **cfd3b**, на которое указывает только что скачанный нами коммит, и родительский коммит **085bb3**:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

Это даст нам следующий объект-коммит. Извлечем объект-дерево:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Кажется, в свободном формате это дерево на сервере не представлено, поэтому мы получили в ответ ошибку 404. Причиной может быть то, что объект находится в другом репозитории или внутри **pack**-файла. Первым делом Git проверяет список альтернативных репозиторий:

```
=> GET objects/info/http-alternates
(empty file)
```

Если ответом на этот запрос становится список URL-адресов, Git ищет файлы как в свободном, так и в упакованном виде по этим адресам. Этот прекрасный механизм позволяет проектам, которые являются ветками друг друга, совместно использовать объекты на диске. Но в данном случае списка альтернатив не появилось, а значит, искомый объект находится в **pack**-файле. Список таких файлов, доступных на данном сервере, содержится в файле **objects/info/packs**. Этот файл также генерируется командой **update-server-info**. Скачаем его:

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

На сервере есть всего один `pack`-файл, и очевидно, что наш объект находится именно там, но мы на всякий случай проверим индексный файл. При наличии нескольких `pack`-файлов эта операция позволяет определить, какой из них содержит нужный объект:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx  
(4k of binary data)
```

Итак, у нас есть индекс `pack`-файла и можно проверить его на наличие внутри интересующего нас объекта. Ведь в индексе перечислены все контрольные суммы SHA содержащихся в `pack`-файле объектов, а также смещения этих объектов относительно друг друга. Мы видим, что наш объект там, и скачиваем данный `pack`-файл целиком:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack  
(13k of binary data)
```

Итак, мы получили наш объект-дерево и можем выполнить обход коммитов. Все они находятся внутри только что скачанного `pack`-файла, поэтому запросов к серверу больше не потребуется. Git проверяет рабочую копию ветки `master`, на которую нацелен указатель `HEAD`.

Интеллектуальный протокол

При своей незамысловатости простой протокол не слишком эффективен, кроме того, он не позволяет записывать данные на сервер. Поэтому куда чаще для передачи данных используется интеллектуальный протокол, требующий на сервере специального процесса, осведомленного о структуре Git-репозитория. Этот процесс умеет читать локальные данные, определять, что именно требуется клиенту, и генерировать для него специальный `pack`-файл. Передачу данных в этом случае обеспечивают два набора процессов: два для отправки данных на сервер и два для скачивания оттуда информации.

Загрузка данных

Для загрузки данных на удаленный сервер Git пользуется процессами `send-pack` и `receive-pack`. Процесс `send-pack` запускается на стороне клиента и подключается к процессу `receive-pack` на стороне сервера.

SSH

Предположим, в рамках своего проекта вы выполняете команду `git push origin master`, при этом ветка `origin` определена как URL-адрес, использующий протокол SSH. Git запускает процесс `send-pack`, который инициирует подключение

к серверу по протоколу SSH. Система пытается запускать на удаленном сервере команды через вот такой SSH-вызов:

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status \
    delete-refs side-band-64k quiet ofs-delta \
    agent=git/2:2.1.1+github-607-gfba4028 delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

Команда **git-receive-pack** немедленно отвечает одной строкой на каждую имеющуюся ссылку, — в данном случае у нас есть только ветка **master** и ее контрольная сумма SHA. Кроме того, первая строка содержит список возможностей сервера (в данном случае это команды **report-status**, **delete-refs** и пр., включая идентификатор клиента).

Каждая строка начинается с 4-символьного шестнадцатеричного значения, задающего длину остальной части строки. В первой строке фигурирует значение **005b**, что в десятичной системе счисления соответствует значению 91. То есть в данной строке остался еще 91 байт. Следующая строка начинается с **003e**, что соответствует числу 62, то есть осталось прочитать еще 62 байта. Затем следует значение **0000**, указывающее на конец списка ссылок.

Теперь, когда состояние сервера известно, процесс **send-pack** определяет, какие из ваших коммитов отсутствуют на сервере, и сообщает эту информацию процессу **receive-pack** для всех ссылок, которые будут обновлены командой **push**. К примеру, если мы обновляем ветку **master**, добавляя ветку **experiment**, процесс **send-pack** даст примерно такой ответ:

```
0085ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b
3ff6 \
    refs/heads/master report-status
0067000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf6
3e8d \
    refs/heads/experiment
0000
```

Git отправляет для каждой обновляемой вами ссылки строку с информацией о длине этой строки, старой и новой контрольными суммами SHA и именем обновляемой ссылки. Кроме того, в первой строке перечисляются функциональные возможности клиента. Значение SHA-1, состоящее из одних нулей, указывает на отсутствие ранее данной ссылки — ведь в этом случае мы добавляем ветку **experiment**. При удалении ветки нули, наоборот, оказались бы справа.

Затем клиент отправляет **pack-файл** со всеми пока отсутствующими на сервере объектами. В конце сервер отчитывается, успешно ли была проведена операция (или возник сбой):

```
000Aunpack ok
```

HTTP(S)

Процесс во многом напоминает подключение по протоколу HTTP, просто в данном случае квити́рование выполняется немного по-другому. Подключение инициирует вот такой запрос:

```
=> 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
```

На этом первый обмен данными между клиентом и сервером завершается. Затем клиент делает второй запрос, на этот раз **POST**, с полученными от команды **git-upload-pack** данными:

```
=> POST http://server/simplegit-progit.git/git-receive/pack
```

Данный запрос **POST** включает в себя в качестве информационного наполнения результат работы процесса **send-pack** и **pack**-файл. А сервер через HTTP-код состояния отчитывается об успехе или сбое операции.

Скачивание данных

В скачивании данных из удаленных репозиторийе́в принимают участие процессы **fetch-pack** и **upload-pack**. Клиент инициирует процесс **fetch-pack**, который подключается к процессу **upload-pack** на стороне сервера, чтобы определить подлежащие передаче данные.

SSH

При скачивании данных по протоколу SSH процесс **fetch-pack** запускает примерно такую команду:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

После подключения процесса **fetch-pack** к процессу **upload-pack** последний отправляет обратно примерно следующее:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEADmulti_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

Это во многом напоминает ответ процесса **receive-pack**, но с другими функциональными возможностями. Кроме того, обратно отправляется ветка, на которую нацелен указатель **HEAD** (**symref=HEAD:refs/heads/master**), чтобы

клиент знал, в какую ветку нужно перейти, если выполняется клонирование. На этом этапе процесс **fetch-pack** смотрит на имеющиеся в наличии объекты, возвращая информацию об объектах, которых ему не хватает. Для этого используется ключевое слово **want** и контрольная сумма SHA недостающего объекта. Данные об имеющихся объектах процесс пересылается с ключевым словом **have** и их контрольными суммами SHA. В конце списка фигурирует слово **done**, заставляющее процесс **upload-pack** приступить к отправке pack-файла с необходимыми данными:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

HTTP(S)

Квитирование при скачивании данных состоит из двух HTTP-запросов. Сначала идет запрос **GET** к той же самой конечной точке, что и в случае простого протокола:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
000000e7ca82a6dff817ec66f44342007202690a93763949 HEADmulti_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed no-done symref=HEAD:refs/heads/master \
    agent=git/2.2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

Это очень похоже на вызов команды **git-upload-pack** по протоколу SSH, но обмен данными оформляется в виде отдельного запроса:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

И снова вы видите уже знакомый формат. Ответ на этот запрос сообщает об успехе или сбое операции и включает в себя pack-файл.

Подводя итоги

В этом разделе мы кратко рассмотрели протоколы передачи данных. Они включают в себя много других функциональных возможностей, таких как команда **multi_ack** или **side-band**, но их рассмотрение выходит за рамки данной книги. Мы попытались дать вам общее представление об обмене данными между клиентом и сервером; дополнительные сведения, если таковые вам потребуются, можно найти в исходном Git-коде.

Обслуживание репозитория и восстановление данных

Время от времени приходится чистить репозиторий, делая его более компактным, убирать лишнее, появившееся в результате импорта, или восстанавливать потерянное. Часть этих процедур рассмотрена в данном разделе.

Обслуживание репозитория

Время от времени система Git автоматически инициирует сбоку мусора. Чаще всего при этом ничего не происходит. Но если у вас скопилось слишком много объектов в свободном формате (не помещенных в pack-файлы) или слишком много pack-файлов, активизируется команда `git gc`. В данном случае аббревиатура `gc` означает сборку мусора (garbage collect), а команда выполняет несколько действий: собирает все свободные файлы и помещает их в pack-файлы, объединяет существующие pack-файлы в один большой файл и удаляет объекты, на которые не ссылается ни один коммит, и объекты, созданные несколько месяцев назад.

Сборку мусора можно запустить вручную:

```
$ git gc --auto
```

И снова в общем случае это не даст никакого результата. Для запуска настоящего сборщика мусора у вас должно быть примерно 7000 свободных объектов или более 50 pack-файлов. Хотя эти цифры можно поменять в параметрах `gc.auto` и `gc.autopacklimit` соответственно.

Кроме того, эта команда осуществляет упаковку ваших ссылок в единый файл. Предположим, ваш 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
```

Команда `git gc` удалит эти файлы из папки `refs`. Для повышения эффективности работы система Git перенесет их в файл `.git/packed-refs`, который имеет примерно такой вид:

```
$ 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 ссылок этот файл не редактируется. Вместо этого Git записывает новый файл в папку **refs/heads**. Чтобы получить корректную контрольную сумму SHA для указанной вами ссылки, Git сначала ищет эту ссылку в папке **refs**, и если поиск заканчивается неудачей, проверяет файл **packed-refs**. Так что если вы не можете найти ссылку в папке **refs**, скорее всего, она была перенесена в файл **packed-refs**.

Обратите внимание на последнюю строку файла, которая начинается с символа **^**. Это означает, что вышележащий тег снабжен комментарием и данная строка является коммитом, на который этот тег указывает.

Восстановление данных

В какой-то момент вы можете потерять коммит. Как правило, такая ситуация возникает в случае принудительного удаления ветки, содержащей какие-то наработки, когда впоследствии внезапно выясняется, что эта ветка все-таки была нужна; или вы выполняете для ветки команду **git reset --hard**, отказываясь от коммитов, в которых потом возникает необходимость. Существуют ли способы вернуть коммиты назад?

В качестве примера вернем ветку **master** в тестовом репозитории в более раннее состояние, а затем восстановим потерянные коммиты. Первым делом посмотрим, как выглядит наша история изменений в данный момент:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Теперь переместим 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
```

Вы результативно избавились от двух верхних коммитов — теперь они недостижимы ни из одной ветки. Нужно найти контрольную сумму SHA самого последнего коммита и добавить ветку, которая будет на него указывать. Сложность в данном случае состоит в определении контрольной суммы, ведь вряд ли вы будете ее специально запоминать.

Зачастую быстрее всего воспользоваться инструментом **git reflog**. Дело в том, что в процессе вашей работы Git незаметно для вас записывает все изменения

положения указателя HEAD. Каждый новый коммит или смена веток обновляют журнал ссылок. Обновляется этот журнал и командой `git update-ref`, поэтому имеет смысл ее использовать вместо записи контрольных сумм SHA в файлы ссылок, как описывалось в разделе «Ссылки в Git». В этом случае команда `git reflog` покажет ваше местоположение в любой момент времени:

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: modified repo.rb a bit
484a592 HEAD@{2}: commit: added repo.rb
```

Мы видим два коммита, с которыми раньше работали. Но в целом информации не очень много. Более информативной является команда `git log -g`, дающая нормальный вывод команды `log` для журнала ссылок.

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:22:37 2009 -0700
```

third commit

```
commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:15:24 2009 -0700
```

modified repo.rb a bit

Похоже, последний коммит является именно тем коммитом, который мы потеряли. Его можно восстановить, создав указывающую на него ветку. К примеру, начнем из этого коммита (`ab1afef`) ветку `recover-branch`:

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Теперь у нас есть ветка `recover-branch`, находящаяся там, где раньше была ветка `master`, и снова делающая доступными первые два коммита. А теперь рассмотрим ситуацию, когда потерянная вами информация по какой-то причине не попала в журнал ссылок. Для имитации такой ситуации достаточно удалить ветку `recover-branch` и журнал ссылок. Теперь первые два коммита в принципе недоступны:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Так как данные журнала ссылок хранятся в только что удаленной нами папке `.git/logs/`, фактически журнал у нас отсутствует. Как восстановить потерянные коммиты? Для этого можно воспользоваться служебной программой `git fsck`, которая проверяет целостность базы данных. Запустив ее с параметром `--full`, вы получите список всех объектов, недостижимых из других объектов:

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

В данном случае мы найдем свой утраченный коммит в строке `dangling commit`. Для его восстановления опять достаточно создать ветку, указывающую на объект с этим хешем SHA.

Удаление объектов

Система Git обладает множеством замечательных вещей, но далеко не все они безобидны. К примеру, источником проблем может стать команда `git clone`, которая скачивает историю проекта, включая в нее все версии всех файлов. В этом нет ничего страшного, когда в репозитории хранится только исходный код. В конце концов, Git умеет эффективно сжимать данные. Но если в проект был кем-то добавлен огромный файл, он будет скачиваться при всех процедурах клонирования, даже если в следующем коммите его удалили из проекта. Он навсегда останется в базе, так как в истории проекта на него есть ссылки.

При преобразовании репозитория Subversion или Perforce в репозитории Git это может стать большой проблемой. В этих системах полное скачивание истории не практикуется, поэтому подобные добавления проходят там практически без последствий и на них не обращают внимания. Но если вы после импорта из другой системы или при других обстоятельствах обнаружили, что ваш репозиторий стал слишком большим, значит, нужно найти и удалить оттуда крупные объекты. И сейчас мы поговорим о том, как это делается.

Имейте в виду, что описанная техника разрушает историю коммитов. Она переписывает все объекты-коммиты, начиная с самого первого дерева, которое нужно отредактировать для удаления ссылки на большой файл. Если вы решили заняться этим сразу после импорта данных, когда никто еще не успел начать работу на базе импортированных коммитов, беспокоиться не о чем. В противном случае нужно уведомить всех, кто вносит вклад в ваш проект, о необходимости перемещения результатов своего труда в новые коммиты.

Итак, добавим крупный файл в тестовый репозиторий, удалим его при следующей фиксации состояния, а потом найдем и навсегда удалим из репозитория. Начнем, естественно, с добавления файла в 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
```

Но мы вовсе не собирались вставлять в проект огромный tar-архив. Поэтому избавимся от него как можно быстрее:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'oops - removed large tarball'
[master dadf725] oops - removed large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Запустим сборщик мусора для нашей базы данных, а затем посмотрим, сколько места занимает репозиторий:

```
$ 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)
```

Сколько места занимает репозиторий, покажет команда `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
```

Элемент `size-pack` дает размер нашего pack-файла в килобайтах, то есть мы используем практически 5 Мбайт. А перед последним коммитом наш репозиторий занимал около 2 Кбайт. Как видите, удаление файла из предыдущего коммита не убрало его из истории. И теперь каждому, кто захочет клонировать ваш небольшой проект, придется скачивать все 5 Мбайт, потому что вы случайно добавили этот огромный файл.

Первым делом нам нужно найти файл, который является источником проблем. В данном случае мы уже знаем, что это за файл. Но представим, что этой информации у нас нет; как определить, какой файл или файлы занимают так много места?

Если вы пользовались командой `git gc`, все объекты уже упакованы в один файл, и для идентификации больших объектов нам потребуется другая служебная команда `git verify-pack`. Выводимые ею данные следует отсортировать по третьему столбцу, в котором указывается размер файла. Этот вывод можно дополнительно пропустить через команду `tail`, так как нас интересует только последний из нескольких крупных файлов:

```
$ 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 находится в самом низу, его размер составляет 5 Мбайт. Узнать, что это за файл, нам поможет команда `rev-list`, с которой вы уже встречались в разделе «Формат сообщения фиксации» главы 8. Добавленный к ней параметр `--objects` заставляет ее вывести список контрольных сумм SHA всех коммитов и всех двоичных массивов данных, а также пути к соответствующим файлам. Именно она поможет нам обнаружить имя нашего объекта:

```
$ git rev-list --objects --all | grep 82c99a3  
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Этот файл следует удалить из всех деревьев, присутствующих в истории вашего проекта. Вы можете легко узнать, какие коммиты модифицировали этот файл:

```
$ git log --oneline -- git.tgz  
dadf725 oops - removed large tarball  
7b30847 add git tarball
```

Для полного удаления этого файла из истории проекта нужно переписать все коммиты, появившиеся после коммита `7b30847`. Для этого мы прибегнем к команде `filter-branch`, уже знакомой вам по разделу «Перезапись истории» в главе 7:

```
$ git filter-branch --index-filter \  
'git rm --cached --ignore-unmatch git.tgz' -- 7b30847^..  
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm ,git.tgz'  
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)  
Ref 'refs/heads/master' was rewritten
```

Параметр `--index-filter` напоминает параметр `--tree-filter`, которым мы пользовались в разделе «Перезапись истории» главы 7, просто в данном случае наша команда модифицирует файлы не на диске, а в области индексирования.

Удаление файла также выполняется не командой `rm file`, а командой `git rm --cached` — ведь нам нужно удалить файл из области индексирования, а не с жесткого диска. В данном случае мы можем себе это позволить, так как системе Git не приходится перед применением фильтра скачивать каждую версию файла на диск, процесс идет намного быстрее. Если хотите, используйте для решения этой задачи

параметр `--tree-filter`. Добавление к команде `git rm` параметра `--ignore-unmatch` отключает вывод сообщения об ошибке в случае отсутствия файлов, совпадающих с шаблоном. Наконец, мы заставляем команду `filter-branch` переписать историю, только начиная с коммита `6df7640`, так как именно здесь впервые появилась проблема. Без этой явной инструкции перезапись началась бы с самого начала, что заняло бы намного больше времени.

Теперь история не содержит ссылок на данный файл. Но в журнале ссылок и в новом наборе ссылок, который система Git добавила в папку `.git/refs/original` при выполнении команды `filter-branch`, они еще есть, поэтому их нужно удалить и выполнить повторную упаковку базы данных. Перед процедурой повторной упаковки следует избавиться от всех элементов, имеющих указатель на эти старые коммиты:

```
$ 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)
```

Посмотрим, сколько места мы освободили.

```
$ 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 Кбайт, что намного лучше исходных 5 Мбайт. Значение поля `size` показывает, что большой объект все еще присутствует среди объектов свободного формата — он никуда не делся, но он не будет передаваться при выполнении команды `push` и при последующем клонировании, а именно этого мы и добивались. Если хотите удалить этот объект навсегда, воспользуйтесь командой `git prune` с параметром `--expire`:

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```


Переменные среды

Git всегда работает внутри командной оболочки `bash` и пользуется для задания своего поведения некоторыми доступными в этой оболочке переменными среды. Имеет смысл знать, где они находятся и как их использовать, чтобы заставить Git вести себя нужным вам образом. Приведенный нами в этом разделе список переменных среды нельзя назвать исчерпывающим, но мы постарались рассказать о самых полезных из них.

Глобальное поведение

Общее поведение системы Git как компьютерной программы зависит от переменных среды.

Переменная `GIT_EXEC_PATH` указывает, где находятся Git-подпрограммы (такие, как `git-commit`, `git-diff` и другие). Текущее значение этой переменной можно узнать командой `git --exec-path`.

Переменная `HOME` обычно считается неизменяемым параметром (потому что от нее зависит слишком много других вещей), но именно в ней Git ищет глобальный файл конфигураций. Чтобы получить переносимую Git-копию с полной глобальной конфигурацией, переопределите переменную `HOME` в профиле оболочки.

На нее похожа переменная `PREFIX`, работающая для общесистемной конфигурации. Git ищет этот файл в папке `$PREFIX/etc/gitconfig`.

Установленная переменная `GIT_CONFIG_NOSYSTEM` отключает использование общесистемного конфигурационного файла. Это требуется в случаях, когда конфигурация системы мешает выполнению ваших команд, а у вас нет доступа к ее редактированию или удалению.

Переменная `GIT_PAGER` управляет программой, выполняющей многостраничный вывод в командной строке. Если ее значение не задано, используется значение переменной `PAGER`.

Переменная `GIT_EDITOR` задает редактор, который Git загружает, когда пользователю нужно отредактировать какой-то текст (например, сообщение фиксации). Если она не задана, используется значение переменной `EDITOR`.

Расположение репозитория

Некоторые переменные среды Git использует для задания взаимодействия с текущим репозиторием.

Переменная `GIT_DIR` указывает местоположение папки `.git`. Если она не задана, Git поднимается по дереву папок, пока не дойдет до папки `~` (домашняя папка

пользователя) или / (корневая папка), проверяя на каждом шаге наличие папки `.git`.

Переменная `GIT_CEILING_DIRECTORIES` управляет поиском папки `.git`. При доступе к медленно загружающимся папкам (например, находящимся на ленточном накопителе или в сети с медленным доступом) имеет смысл останавливать этот поиск принудительно, особенно если Git начинает этим заниматься, например, при создании приглашения командного процессора.

Переменная `GIT_WORK_TREE` указывает местоположение корневой рабочей папки в случае обычного, а не голого репозитория. Если она не задана, используется папка, родительская для `$GIT_DIR`.

Переменная `GIT_INDEX_FILE` содержит путь к индексному файлу (только для репозитория с непустой рабочей папкой).

Переменная `GIT_OBJECT_DIRECTORY` может применяться для задания местоположения папки, которая обычно находится в папке `.git/objects`.

Переменная `GIT_ALTERNATE_OBJECT_DIRECTORIES` содержит разделенный двоеточиями список (отформатированный так: `/dir/one:/dir/two:..`), указывающий системе Git, где можно найти объекты, отсутствующие в папке `GIT_OBJECT_DIRECTORY`. При работе с несколькими проектами, в которых задействованы большие файлы с одним и тем же содержимым, это позволяет избежать появления большого числа дубликатов.

Пути доступа

Термин «путь доступа» относится к указанию местоположения различных Git-объектов, в том числе с применением групповых символов. Эти настройки применяются как в файле `.gitignore`, так и в командной строке (`git add *.c`).

Переменные `GIT_GLOB_PATHSPECS` и `GIT_NOGLOB_PATHSPECS` управляют поведением групповых символов в путях доступа. Если переменной `GIT_GLOB_PATHSPECS` присвоено значение 1 (а это значение, предлагаемое по умолчанию), групповые символы понимаются обычным образом; присвоение переменной `GIT_NOGLOB_PATHSPECS` значения 1 приводит к их буквальной трактовке. Например, запись `*` с будет означать только файл с именем `*.c`, а не любой файл с расширением `.c`. Это поведение допускает переопределение. Для этого в начало пути следует добавить префикс `:(glob)` или `:(literal)`, например `:(glob)*.c`.

Переменная `GIT_LITERAL_PATHSPECS` отключает оба указанных варианта поведения; перестают работать как групповые символы, так и специальные префиксы.

Переменная `GIT_ICASE_PATHSPECS` делает все пути независимыми от регистра символов.

Фиксация изменений

Окончательное создание объекта-коммита в Git обычно осуществляется командой `git-commit-tree`, которая берет информацию из следующих переменных среды, используя конфигурационные значения, только если эти переменные отсутствуют.

- ❑ Переменная `GIT_AUTHOR_NAME` определяет удобочитаемое имя в поле `author`.
- ❑ Переменная `GIT_AUTHOR_EMAIL` задает адрес электронной почты для поля `author`.
- ❑ Переменная `GIT_AUTHOR_DATE` представляет собой временную метку для поля `author`.
- ❑ Переменная `GIT_COMMITTER_NAME` задает удобочитаемое имя для поля `committer`.
- ❑ Переменная `GIT_COMMITTER_EMAIL` содержит адрес электронной почты для поля `committer`.
- ❑ Переменная `GIT_COMMITTER_DATE` служит временной меткой для поля `committer`.
- ❑ Переменная `EMAIL` содержит запасной адрес электронной почты на случай незаданного конфигурационного значения `user.email`. Если не задана окажется и эта переменная, Git начинает использовать имя пользователя в системе и имя хоста.

Работа в сети

Для сетевых операций по протоколу HTTP в Git используется библиотека `curl`, соответственно, переменная `GIT_CURL_VERBOSE` заставляет Git выводить все генерируемые этой библиотекой сообщения. Тот же самый результат дает команда `curl -v` в командной строке.

Переменная `GIT_SSL_NO_VERIFY` отключает проверку SSL-сертификатов. Такая возможность требуется, к примеру, если вы используете самостоятельно подписанный сертификат для обслуживания Git-репозитория по протоколу HTTPS, или вы находитесь в процессе настройки Git-сервера, а все необходимые сертификаты пока не установлены.

Если скорость передачи данных по протоколу HTTP ниже, чем `GIT_HTTP_LOW_SPEED_LIMIT` байт в секунду, и остается такой дольше, чем `GIT_HTTP_LOW_SPEED_TIME` секунд, операция прерывается. Эти переменные переопределяют значения конфигурационных параметров `http.lowSpeedLimit` и `http.lowSpeedTime`.

Переменная `GIT_HTTP_USER_AGENT` задает строку `user-agent`, которой Git пользуется при работе по протоколу HTTP. По умолчанию она имеет значение `git/2.0.0`.

Определение изменений и слияние

Переменная `GIT_DIFF_OPTS` — на самом деле не совсем переменная. У нее существуют только два допустимых значения: `-u<n>` и `--unified=<n>`, задающих количество контекстных строк, присутствующих в выводимых данных команды `git diff`.

Переменная `GIT_EXTERNAL_DIFF` используется как замещение конфигурационного значения `diff.external`. Если она задана, Git в ответ на команду `git diff` вызывает указанную программу.

Переменные `GIT_DIFF_PATH_COUNTER` и `GIT_DIFF_PATH_TOTAL` используются внутри программы, заданной переменной `GIT_EXTERNAL_DIFF` или конфигурационным значением `diff.external`. Первая показывает, для какого файла в последовательности (начиная с 1) вычисляются изменения, а вторая определяет общее число файлов в этом пакете.

Переменная `GIT_MERGE_VERBOSE` контролирует уровень детализации вывода при рекурсивном слиянии. Она может принимать следующие значения:

- ❑ 0 — выводится только возможное сообщение об ошибке;
- ❑ 1 — отображаются только конфликты;
- ❑ 2 — дополнительно отображаются изменения файлов;
- ❑ 3 — показывает, когда файлы пропускаются, так как в них не были внесены изменения;
- ❑ 4 — выводятся все пути по мере их обработки;
- ❑ 5 и выше — выводится детализированная отладочная информация.

По умолчанию эта переменная имеет значение 2.

Отладка

Хотите знать, на что способна система Git? В ней есть практически полный набор возможностей слежения, которые достаточно только подключить. Перечисленные далее переменные могут принимать одно из следующих значений:

- ❑ `true`, 1 или 2 — вывод записывается в стандартный поток ошибок;
- ❑ начинающийся с символа `/` абсолютный путь — вывод записывается в указанный файл.

Переменная `GIT_TRACE` контролирует протоколирование действий, не попадающих ни в одну из заданных категорий. К таким действиям относятся, например, расширение псевдонимов и вызовы внешних подпрограмм.

```
$ 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'

```

Переменная `GIT_TRACE_PACK_ACCESS` регистрирует доступ к pack-файлам. Первое поле содержит pack-файл, к которому вы обращаетесь, а второе — 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

```

Переменная `GIT_TRACE_PACKET` активирует регистрацию сетевых операций на уровне пакетов:

```

$ 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 ofsdelta
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
# [...]

```

Переменная `GIT_TRACE_PERFORMANCE` отвечает за регистрацию сведений о производительности. Выводимые данные показывают, сколько времени занимали 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'
                             'packobjects' '--keep-true-parents'
                             '--honor-pack-keep' '--non-empty' '--all'
                             '--reflog' '--unpackunreachable= 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'
                             'updateserver-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'

```

Переменная `GIT_TRACE_SETUP` показывает информацию, собранную системой Git о репозитории и среде, с которой она взаимодействует:

```

$ 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

Если переменная `GIT_SSH` задана, указанная в ней программа будет использоваться вместо `ssh` при попытках системы Git подключиться к SSH-хосту. Порядок вызова в данном случае следующий:

```
$GIT_SSH [имя пользователя@]хост [-p <порт>] <команда>
```

Это не самый простой способ настроить вызовы `ssh`; дополнительные параметры командной строки не поддерживаются, и вам, скорее всего, придется писать сценарий-надстройку, превращая переменную `GIT_SSH` в ссылку на этот сценарий. Вероятно, проще воспользоваться файлом `~/.ssh/config`.

Переменная `GIT_ASKPASS` переопределяет значение конфигурационного параметра `core.askpass`. Это программа вызывается каждый раз, когда системе Git нужно узнать учетные данные пользователя. Она ожидает текстового приглашения как

аргумента командной строки и возвращает ответ в стандартный поток вывода. (Более подробно см. раздел «Хранение учетных данных» в главе 7.)

Переменная `GIT_NAMESPACE` контролирует доступ к ссылкам внутри пространства имен и эквивалентна флагу `--namespace`. В основном она используется на стороне сервера, когда нужно сохранить в одном репозитории несколько версий этого репозитория, разделяя лишь ссылки.

Переменная `GIT_FLUSH` заставляет систему Git при инкрементной записи в стандартный поток вывода использовать небуферизованный ввод-вывод. Значение 1 увеличивает частоту сброса данных, а значение 0 делает вывод целиком буферизованным. Значение, предлагаемое по умолчанию (если эта переменная не задана), выбирается в зависимости от выполняемых действий и режима вывода данных.

Переменная `GIT_REFLOG_ACTION` позволяет задать описание в журнале ссылок. Вот пример:

```
$ 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
```

Заклучение

Теперь вы должны хорошо понимать, как Git функционирует в фоновом режиме, и до некоторой степени разбираться в том, как система реализована. В этой главе были рассмотрены некоторые служебные команды, более низкоуровневые и простые, чем команды, которыми вы пользовались в остальной части книги. Знание того, как Git функционирует на низком уровне, облегчит вам понимание действий этой системы и даст возможность писать собственные инструменты и вспомогательные сценарии для организации нестандартного рабочего процесса.

Будучи контентно-адресуемой файловой системой, Git является мощным инструментом, который может применяться не только как VCS. Надеемся, что сведения о внутреннем устройстве Git помогут вам в реализации собственных вариантов применения этой системы и дадут ощущение свободы даже при решении сложных задач.

Приложение А

Git в других средах

Если вы прочитали книгу целиком, значит, вы уже много знаете о том, как пользоваться системой Git из командной строки. Вы умеете работать с локальными файлами, синхронизировать репозитории по сети и эффективно работать с другими пользователями. Но это еще не все; Git обычно применяется как часть большой экосистемы, и терминал — далеко не лучший инструмент для работы с ней. Рассмотрим несколько вариантов окружения, в которых вам может пригодиться Git. Также вы узнаете, как другие приложения (в том числе и ваши собственные) работают с этой системой.

Графические интерфейсы

Терминал является родной средой для Git. Именно там первым делом рождаются новые функциональные возможности, и только из командной строки можно в полной мере ощутить всю мощь Git. Но текстовый интерфейс подходит для решения далеко не всех задач; иногда более предпочтительным является графическое представление, а некоторым пользователям намного удобнее работать мышью.

Важно понимать, что каждый интерфейс оптимизирован под свои рабочие схемы. Некоторые клиенты ограничены только теми функциональными возможностями, которые их автор считает наиболее востребованными или эффективными. С этой точки зрения ни один из инструментов, о которых мы будем говорить в этом приложении, не может быть «лучше» остальных; каждый из них просто

предназначен для решения конкретных задач. Кроме того, помните, что все задачи, решаемые с помощью графического интерфейса, могут решаться и из командной строки. И именно командная строка дает вам максимальный контроль над репозиториями.

Утилиты `gitk` и `git-gui`

После установки `Git` вы получаете два графических инструмента: `gitk` и `git-gui`.

Инструмент `gitk` предназначен для просмотра истории. Представьте, что у вас есть мощная GUI-оболочка для команд `git log` и `git grep`. Именно этим инструментом вы будете пользоваться для поиска прошедших событий и визуализации истории проекта.

Проще всего `gitk` вызывается из командной строки. Перейдите в `Git`-репозиторий и введите:

```
$ gitk [git log параметры]
```

Инструмент `gitk` принимает множество различных параметров командной строки, большинство из которых передается базовой команде `git log`. Одним из наиболее применяемых является параметр `--all`, заставляющий инструмент `gitk` показывать коммиты, достижимые по любой ссылке, а не только те, на которые нацелен указатель `HEAD` (рис. А.1).

Вверху находится рисунок, напоминающий вывод команды `git log --graph`; каждая точка соответствует коммиту, линии показывают соотношения между коммитами, а ссылки представлены цветными прямоугольниками. Желтая точка означает указатель `HEAD`, а красная — изменения, которые попадут в следующий коммит. В нижней части окна дано представление выделенного коммита; комментарии и исправления — слева, а обобщенный вид — справа. В центре расположен набор элементов управления для поиска в истории проекта.

Второй инструмент, `git-gui`, в основном предназначен для редактирования коммитов (рис. А.2). Он также легко вызывается из командной строки:

```
$ git gui
```

Слева находится область индексирования; неиндексированные изменения располагаются сверху, а индексированные — снизу. Файлы можно перемещать из одного состояния в другое, щелкая на их значках. Чтобы выбрать файл для просмотра, просто щелкните на его имени.

Справа вверху находится область просмотра изменений в выделенном файле. Можно индексировать отдельные фрагменты кода (или отдельные строки), щелкая на этой области правой кнопкой мыши.

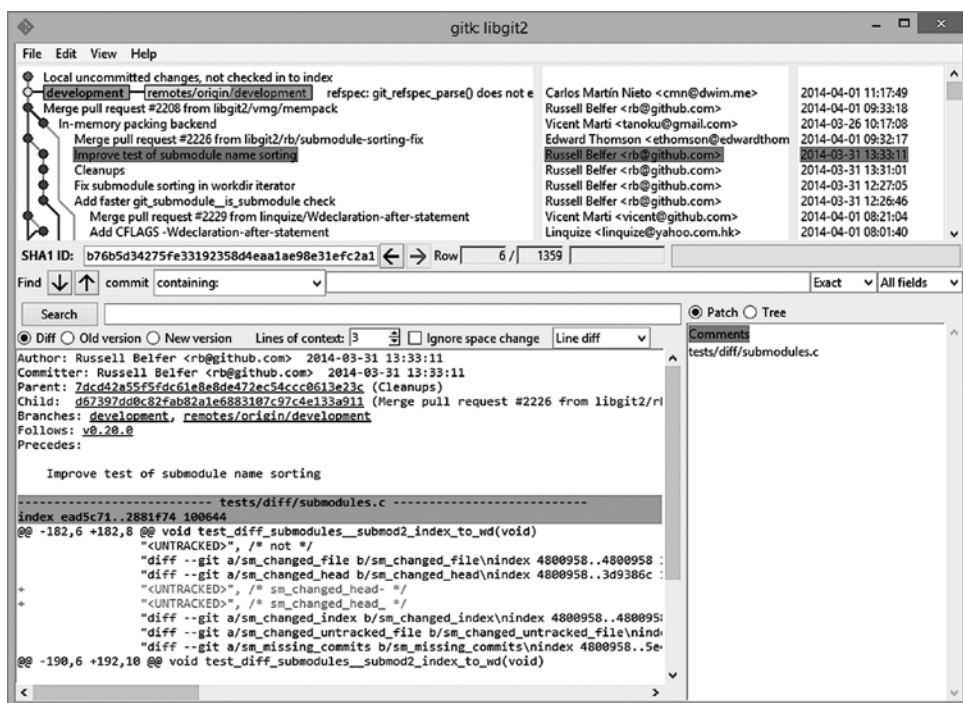


Рис. А.1. gitk — инструмент для просмотра истории

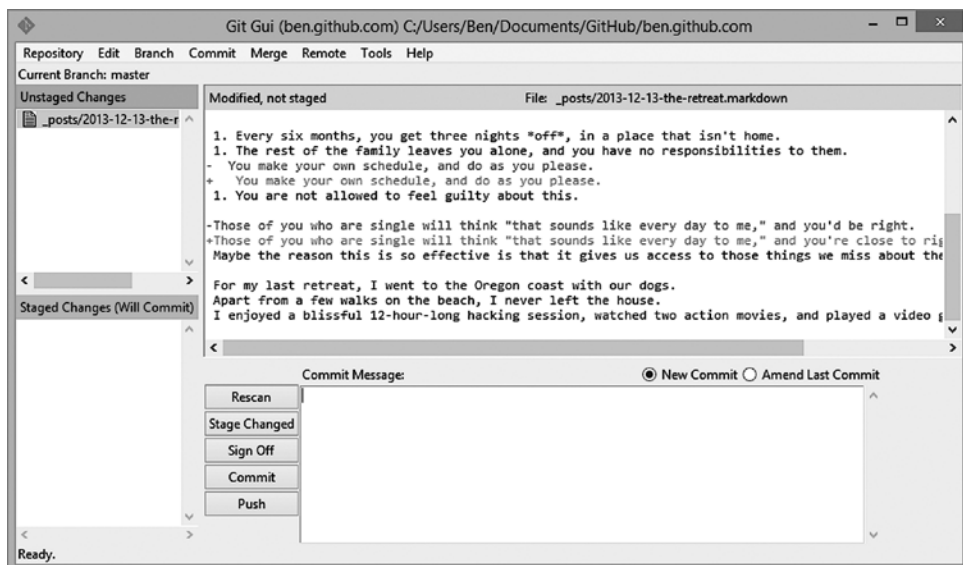


Рис. А.2. git-gui — инструмент для работы с коммитами

Справа снизу мы видим область для ввода сообщений фиксации и несколько кнопок. Введите в текстовое поле сообщение и щелкните на кнопке **Commit**, чтобы выполнить действие, напоминающее команду `git commit`. Кроме того, в последний коммит можно вносить правки, установив переключатель **Amend Last Commit**. Это обновляет область **Staged Changes**, добавляя туда содержимое последнего коммита. После этого вы можете индексировать изменения и убирать их из индекса, редактировать сообщение фиксации, а затем щелчком на кнопке **Commit** заменить старый коммит новым.

Инструменты `gitk` и `git-gui` являются примерами приложений, ориентированных на решение конкретных задач. У каждого из них есть определенное предназначение (просмотр истории и создание коммитов соответственно), поэтому функциональные возможности, которые не требуются для решения поставленных задач, там попросту не поддерживаются.

GitHub-клиенты для Mac и Windows

Компания GitHub создала два Git-клиента, ориентированных на рабочий процесс: для Windows и для Mac (рис. А.3 и А.4). Эти клиенты демонстрируют замечательный пример ориентированности на решение конкретных задач. Они предоставляют доступ не ко всем функциональным возможностям системы Git, а только к наиболее востребованным, хорошо работающим друг с другом.

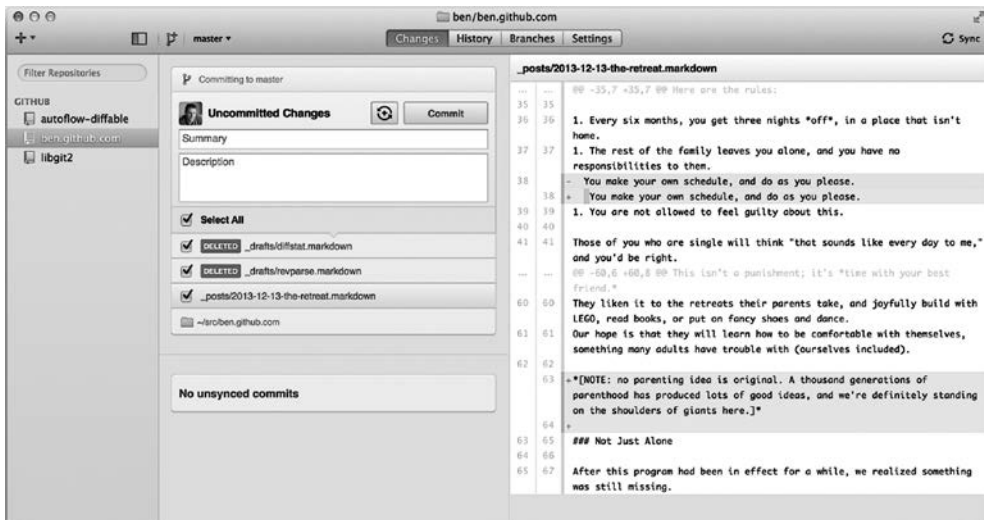


Рис. А.3. GitHub-клиент для Mac

Они спроектированы по одному шаблону, поэтому мы будем рассматривать их как один продукт. В детали мы углубляться не будем (в конце концов,

у этих инструментов существует документация), а в основном поговорим о представлении изменений (именно за этим занятием вы будете проводить большую часть времени).

Слева находится список отслеживаемых репозиторий; для добавления репозитория (путем клонирования или путем присоединения локальной копии) щелкните на кнопке + в верхней части этой области.

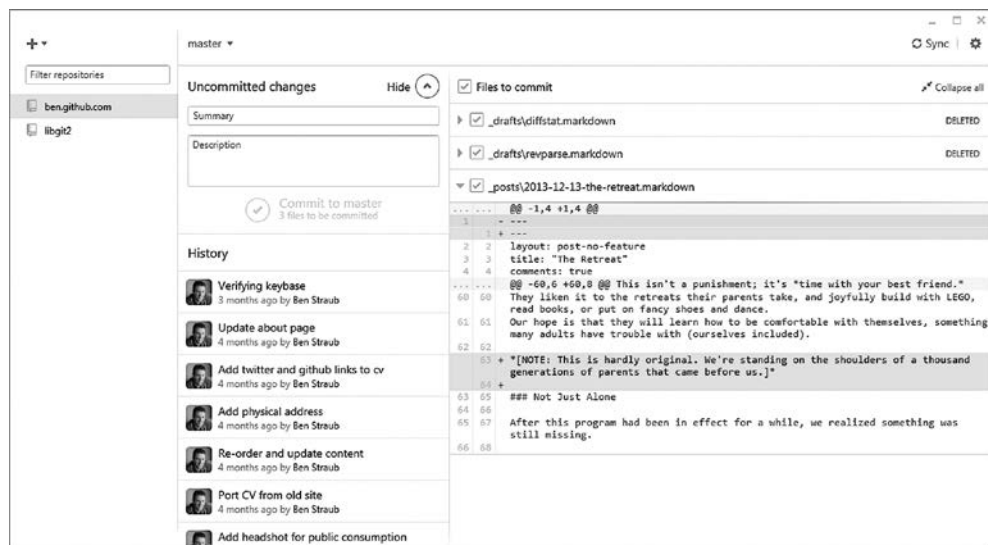


Рис. А.4. GitHub-клиент для Windows

В центре находится область ввода коммита. Здесь вы указываете сообщение фиксации и файлы, которые следует включить в коммит. (В версии для Windows история коммитов демонстрируется сразу под этой областью, а в версии для Mac она находится на отдельной вкладке.)

Справа располагается область просмотра изменений, показывающая, что изменилось в рабочей папке и какие изменения были добавлены в выбранный коммит.

Следует также обратить внимание на кнопку Sync в верхней правой части. Именно она предлагает основной способ взаимодействия по сети.

ПРИМЕЧАНИЕ

Для работы с этими инструментами учетная запись на сайте GitHub не требуется. Хотя они и спроектированы под GitHub-службу, их можно использовать с любым репозиторием и любым Git-сервером.

Установка

Интерфейс GitHub для Windows можно скачать со страницы <https://windows.github.com>, а GitHub для Mac находится по адресу <https://mac.github.com>. При первом запуске оба приложения выполняют первоначальную настройку Git, например укажут ваши имя и адрес электронной почты, а также зададут разумные значения, предлагаемые по умолчанию для многих распространенных конфигурационных параметров, таких как кэши учетных данных и обработка символов CRLF.

Оба инструмента поддерживают фоновое автоматическое скачивание и установку обновлений. Это относится и к связанной с ними версии системы Git, так что вам, возможно, никогда не придется обновлять ее вручную. В операционной системе Windows к клиенту прилагается также ярлык для запуска оболочки Powershell с пакетом Posh-Git, которые мы рассмотрим чуть позже.

Теперь нужно предоставить инструменту репозитории, с которыми он будет работать. Клиент показывает список репозиториях, доступных на сайте GitHub, любой из которых допускает клонирование одним щелчком. Если локальный репозиторий у вас уже есть, просто перетащите его папку из приложения Finder или Windows Explorer в окно GitHub-клиента, и оно окажется в расположенном слева списке репозиториях.

Рекомендуемый рабочий процесс

Установленный и настроенный GitHub-клиент позволяет решать многие типовые для Git задачи. Оптимальную рабочую схему для этого инструмента мы подробно рассмотрели в разделе «Схема работы с GitHub» главы 6, но вкратце она выглядит как регулярная фиксация состояний ветки и синхронизация с удаленным репозиторием.

Управление ветками относится к одной из немногих вещей, которая в этих инструментах реализована по-разному. В версии для Mac в верхней части окна находится кнопка создания новой ветки (рис. А.5).



Рис. А.5. Кнопка создания ветки в версии для Mac

В версии для Windows нужно ввести имя новой ветки в виджет, управляющий сменой веток (рис. А.6).

Добавление коммитов в готовую ветку осуществляется тривиально. Внесите изменения в рабочую папку, и при переходе в окно GitHub-клиента вы увидите, какие файлы были модифицированы. Введите сообщение фиксации, выберите файлы, которые вы планируете включить в коммит, и щелкните на кнопке Commit (либо нажмите клавиатурную комбинацию Ctrl+Enter или ⌘+Enter).

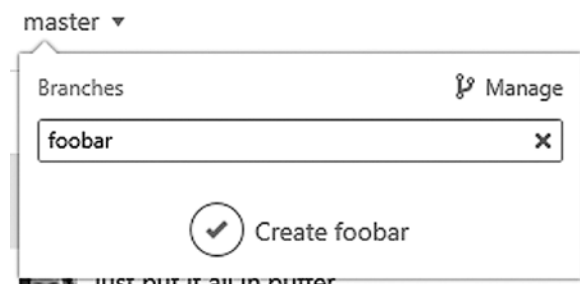


Рис. А.6. Создание ветки в версии для Windows

Основное средство взаимодействия с другими репозиториями — кнопка **Sync**. В Git существуют отдельные команды для отправки данных на сервер, скачивания с сервера информации, слияния и перемещения, а вот в GitHub-клиенте все это реализует один программный компонент. Вот что происходит после щелчка на кнопке **Sync**:

1. `git pull --rebase`. Если по причине конфликта слияния эта команда дает сбой, происходит возвращение в предшествующее состояние командой `git pull --norebase`.
2. `git push`.

Это наиболее распространенная последовательность команд при работе в данном стиле, поэтому их совмещение экономит массу времени.

Подводя итоги

Рассмотренные инструменты отлично справляются с задачами, для решения которых они были спроектированы. Разработчики (и не только они) могут с их помощью в считанные минуты начать работу над совместным проектом, причем многие распространенные операции реализованы в виде встроенных элементов интерфейса. Впрочем, если вы предпочитаете другой рабочий процесс или ожидаете большего контроля над выполнением сетевых операций, рекомендуем обратиться к другому клиенту или к командной строке.

Другие GUI

Существует множество графических клиентов для работы с Git, начиная со специализированных, предназначенных для решения конкретных задач, и заканчивая инструментами, пытающимися воплотить всю доступную в Git функциональность. На официальном сайте Git вы найдете актуальный список наиболее популярных клиентов. Он располагается на странице <http://git-scm.com/downloads/guis>. Еще более

подробный список доступен в Git-вики по адресу https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces.

Git в Visual Studio

Начиная с версии Visual Studio 2013 обновление 1, пользователям стал доступен встроенный в IDE Git-клиент. В Visual Studio уже довольно давно добавлена функциональность управления исходным кодом, но она была ориентирована на централизованные системы с блокировкой файлов, а система Git не очень вписывается в такой рабочий процесс. Поддержка Git в Visual Studio 2013 была отделена от более старых функциональных особенностей, что позволило добиться лучшей интеграции этих двух инструментов.

Откройте проект, который управляется системой Git (или выполните команду `git init` для существующего проекта), и выберите в меню View команду Team Explorer. Откроется окно Connect (рис. А.7).

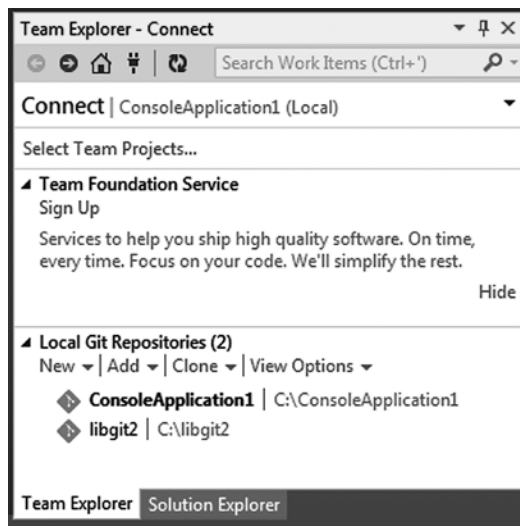


Рис. А.7. Подключение к репозиторию Git через Team Explorer

Приложение Visual Studio запоминает все открываемые вами проекты, управляемые системой Git, формируя их список в нижней части окна. Если вы не видите в списке нужного проекта, щелкните на ссылке **Add** и укажите путь к рабочей папке. Двойной щелчок на имени локального Git-репозитория откроет его главную страницу Home, которая показана на рис. А.8. Это центр действий, связанных с Git; при написании кода вы, как правило, проводите больше всего времени на странице **Changes**, а когда

дело доходит до скачивания результатов труда ваших коллег, переходите на страницы Unsynced Commits и Branches.

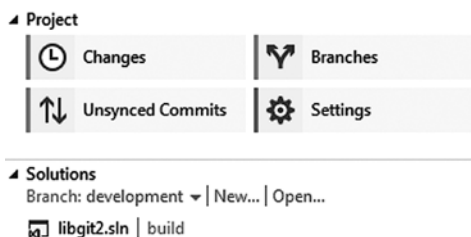


Рис. А.8. Главная страница Git-репозитория в Visual Studio

В Visual Studio в настоящее время имеется мощный, ориентированный на решение конкретных задач пользовательский интерфейс для Git. В него входят средства линейного представления истории, просмотра добавленных изменений, выполнения удаленных команд и многие другие. Познакомиться с их полным перечнем (рассмотрение которого выходит за рамки темы данной книги) вы можете на странице <http://msdn.microsoft.com/en-us/library/hh850437.aspx>.

Git в Eclipse

Приложение Eclipse поставляется с подключаемым модулем Egit, предоставляющим достаточно полный интерфейс для работы с Git (рис. А.9). Для доступа к нему следует перейти к Git-представлению (выбрав в меню Window команду Open Perspective, а затем вариант Other и выбрав Git).

Git в Bash

Пользователи командной оболочки Bash могут задействовать некоторые ее функциональные возможности для упрощения работы с Git. Система Git поставляется с подключаемыми модулями нескольких оболочек, но по умолчанию они не работают.

Первым делом скопируйте из репозитория с исходным Git-кодом файл `contrib/completion/git-completion.bash`. Поместите копию в доступное место, например в папку `home`, и добавьте в файл `.bashrc` такую строку:

```
. ~/git-completion.bash
```

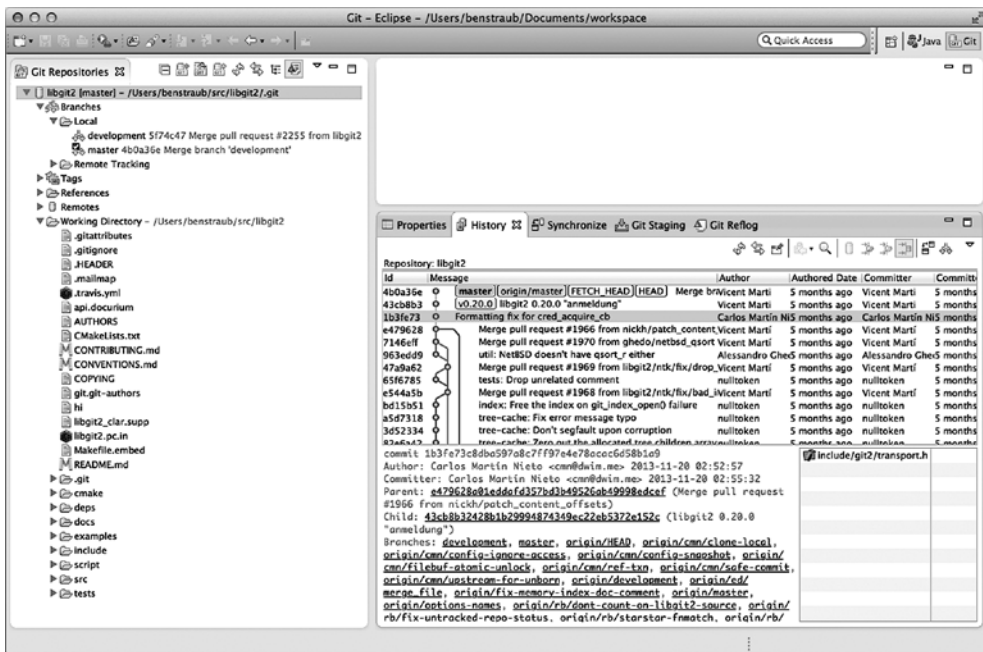



Рис. А.9. Среда EGit в Eclipse

Затем перейдите в Git-репозиторий и наберите следующую команду:

```
$ git che<tab>
```

В результате оболочка Bash автоматически добавит в строку команду `git checkout`. Этот прием работает для всех Git-команд, параметров командной строки, а где это возможно еще и для имен удаленных серверов и ссылок.

Кроме того, имеет смысл настроить в приглашении на ввод режим отображения информации о связанном с текущей папкой Git-репозитории. Можно вывести сколь угодно сложную информацию, но обычно большинству пользователей хватает сведений о текущей ветке и состоянии рабочей папки. Для добавления этой информации достаточно скопировать файл `contrib/completion/git-prompt.sh` из репозитория с исходным Git-кодом в свою папку `home` и добавить в файл `.bashrc`, к примеру, следующие строки:

```
~/.git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(__git_ps1 " (%s)")\$ '
```

Запись `\w` означает показ текущей рабочей папки, символы `\$` означают вывод в приглашении на ввод символа `$`, а запись `__git_ps1 " (%s)"` вызывает вместе с аргументом форматирования функцию, предоставляемую файлом `git-prompt.sh`.

После этого при вызове в рамках Git-проектов приглашение на ввод команды будет выглядеть так, как показано на рис. А.10.

Оба этих сценария снабжены исчерпывающей документацией; дополнительную информацию вы найдете в файлах `git-completion.bash` и `git-prompt.sh`.



```
~/src/libgit2 (development *)$ █
```

Рис. А.10. Специальный вариант приглашения на ввод команды в bash

Git в Zsh

Кроме того, система Git поставляется с библиотекой автоматического заполнения при нажатии клавиши табуляции в командной оболочке Zsh. Скопируйте файл `contrib/completion/git-completion.zsh` в папку `home` и укажите путь к нему в конфигурационном файле `.zshrc`. Командная оболочка Zsh обладает куда более мощным интерфейсом, чем 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
```

Допускающие двоякое толкование варианты автоматического заполнения не просто перечислены, а снабжены исчерпывающими описаниями, и вы можете перемещаться по списку, снова и снова нажимая клавишу `Tab`. Этот режим работает не только для Git-команд, но и для их аргументов и даже имен объектов (например, ссылок и удаленных серверов) внутри репозитория, а также для имен файлов и прочих вещей, для которых в Zsh реализована функция автоматического заполнения.

Настройка приглашения на ввод команды в Zsh происходит почти таким же способом, как и в Bash, но у вас есть возможность вывода информации справа. Чтобы показывать имя ветки в этой дополнительной строке, добавьте в файл `~/.zshrc` следующие строки:

```
setopt prompt_subst
. ~/git-prompt.sh
export RPROMPT='${__git_ps1 "%s"}'
```

В результате при каждом переходе командной оболочки внутрь репозитория Git с правой стороны будет появляться имя текущей ветки (рис. А.11).



Рис. А.11. Специальный вариант приглашения на ввод команды в zsh

Возможности настройки Zsh столько обширны, что существуют целые фреймворки, предназначенные для его улучшения. Один из таких проектов называется oh-my-zsh и доступен по адресу <https://github.com/robbyrussell/oh-my-zsh>. Это система подключаемых модулей с мощным набором правил автоматического заполнения для Git и различными вариантами приглашений на ввод команды, многие из которых показывают информацию, связанную с контролем версий. Рисунок А.12 демонстрирует один из примеров настройки.



Рис. А.12. Пример темы из проекта oh-my-zsh

Git в Powershell

Стандартный терминал командной строки в Windows (`cmd.exe`) не предназначен для работы с Git, но вы можете воспользоваться оболочкой Powershell. Пакет Posh-Git (<https://github.com/dahlbyk/posh-git>) предоставляет мощные функции автоматического заполнения и расширенные приглашения на ввод команд, помогая поддерживать состояние репозитория на высоком уровне (рис. А.13).

Если у вас установлено GitHub-приложение для Windows, значит, есть и Posh-Git, поэтому вам остается только добавить в файл `profile.ps1` (который обычно находится в папке `C:\Users\<имя пользователя>\Documents\WindowsPowerShell`) следующие строки:

```
. (Resolve-Path "$env:LOCALAPPDATA\GitHub\shell.ps1")  
. $env:github_posh_git\profile.example.ps1
```

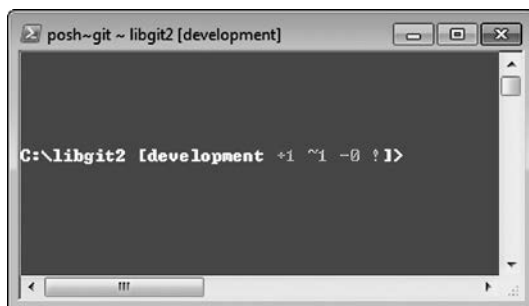


Рис. А.13. Командная оболочка Powershell с Posh-Git

Остальным же нужно загрузить последнюю версию пакета Posh-Git (<https://github.com/dahlbyk/posh-git>) и распаковать ее в папке `WindowsPowerShell`. Затем запускается Powershell с правами администратора и выполняются следующие команды:

```
> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser -Confirm  
> cd ~\Documents\WindowsPowerShell\posh-git  
> .\install.ps1
```

Это добавит в файл `profile.ps1` необходимые строки, и при следующем запуске терминала вы получите доступ к Posh-Git.

Заключение

Теперь вы умеете пользоваться системой Git в рамках инструментов, применяемых при решении каждодневных задач, и знаете, как получить доступ к репозиториям Git из своих программ.

Приложение Б

Встраивание Git в приложения

Если приложение, которое вы пишете, предназначено для разработчиков, скорее всего, от интеграции с системой контроля версий оно только выиграет. Впрочем, пользу из такой интеграции могут извлечь и приложения для обычных пользователей, например текстовые редакторы. Git замечательно работает в самых разных сценариях.

Интегрировать Git в приложение можно тремя способами: вызвать командную оболочку и воспользоваться Git-инструментами командной строки или библиотеками Libgit2 и JGit.

Командная строка

Итак, вы можете вызвать командную оболочку и воспользоваться для решения текущих задач командной строкой. Это канонический подход, кроме того, вы получаете поддержку всех функциональных возможностей системы Git. Более того, это очень простой способ, так как в большинстве сред выполнения предоставляется относительно удобный доступ к вызову процессов с аргументами командной строки. Но подход обладает и своими недостатками.

Во-первых, выводимые командами данные имеют вид обычного текста. То есть вам придется заставлять Git время от времени менять формат вывода для удобства анализа информации, что далеко не всегда эффективно и повышает вероятность ошибок.

Вторым недостатком является отсутствие способности восстанавливать работу системы после ошибок. При повреждении репозитория или в случае некорректного конфигурационного параметра Git просто отказывается выполнять большинство операций.

Кроме того, необходимо управлять процессом. Система Git требует для среды оболочки отдельного процесса, что усложняет ситуацию. Координация набора таких процессов (особенно если они работают с одним и тем же репозиторием) может оказаться нетривиальной задачей.

Libgit2

Следующий вариант дает нам библиотека Libgit2. Она представляет собой свободную от внешних зависимостей Git-реализацию, основной целью которой является предоставление хорошего API другим программам. Скачать эту библиотеку можно с сайта <http://libgit2.github.com>.

Первым делом посмотрим, как выглядит C API:

```
// 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 свойств коммита
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_object_free(head_commit);
git_repository_free(repo);
```

Первая пара строк открывает репозиторий Git. Тип `git_repository` является дескриптором на репозиторий с кэшем в памяти. Это самый простой метод, которым можно пользоваться, если вы знаете точный путь к рабочей папке репозитория или к папке `.git`. Существует также вариант `git_repository_open_ext`,

принимающий параметры поиска, вариант `git_clone` и ему подобные для создания клона удаленного репозитория, и вариант `git_repository_init` для создания нового репозитория.

Во втором фрагменте кода фигурирует синтаксис `rev-parse` (подробно он рассматривался в разделе «Ссылки из веток» главы 7), помогающий получить коммит, на который в данный момент нацелен указатель `HEAD`. Нам возвращают указатель `git_object`, представляющий собой нечто, существующее в базе данных Git-объектов. По существу, тип `git_object` является предком для ряда других объектов; во всех случаях используется одна и та же структура памяти, поэтому можно безопасно преобразовывать типы друг в друга. В данном случае `git_object_type(commit)` вернет нам тип `GIT_OBJ_COMMIT`, который можно будет безопасно преобразовать в указатель `git_commit`.

Следующий фрагмент кода показывает способ доступа к свойствам коммита. В последней строке фигурирует тип `git_oid`; он является внутренним представлением контрольной суммы SHA-1 в библиотеке Libgit2.

В этом примере прослеживаются следующие закономерности.

- ❑ Если вы объявили указатель и передали ссылку в какую-либо Libgit2-функцию, вам, скорее всего, вернут целочисленный код ошибки. Значение 0 указывает на успешное выполнение операции; все, что меньше, означает ошибку.
- ❑ Если библиотека Libgit2 загружает для вас указатель, его освобождение становится вашей задачей.
- ❑ Если библиотека Libgit2 возвращает после вызова функции указатель `const`, освобождать его не нужно. Он сам становится недействительным при уничтожении объекта, на который ссылается.
- ❑ Писать код на языке C не так-то просто.

Последний пункт означает, что вам вряд ли придется пользоваться языком C при работе с библиотекой Libgit2. К счастью, существует ряд привязок для различных языков, облегчающих работу с репозиториями Git из вашей среды исполнения. Перепишем предыдущий пример с привязкой к языку Ruby для Libgit2. Эта привязка называется Rugged и доступна по адресу <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, объем кода сократился. Во-первых, привязка Rugged использует исключения; она умеет сигнализировать о сбоях, вызывая `ConfigError` или `ObjectError`. Во-вторых, вам не требуется в явном виде освобождать ресурсы, так как в языке Ruby есть сборщик мусора. Рассмотрим 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)

```

Целиком процедура выглядит так:

1. Создается новый двоичный массив данных с содержимым нового файла.
2. Область индексации заполняется содержимым дерева коммита, на который нацелен указатель **HEAD**, и добавляется новый файл в путь к файлу **newfile.txt**.
3. Создается новое дерево в ODB, которое используется для нового коммита.
4. Для полей **author** и **committer** используется одна сигнатура.
5. Сообщение фиксации.
6. При создании коммитов требуется указывать их предков. В данном случае родителем является указатель **HEAD**.
7. При создании коммита привязка **Rugged** (и библиотека **Libgit2**) может обновить ссылку.
8. Возвращаемое значение — это контрольная сумма **SHA-1** нового коммита, которую впоследствии можно использовать для доступа к этому объекту.

Код на языке Ruby получается чистым и лаконичным, так как основную работу выполняет библиотека **Libgit2**. Выполняется он также достаточно быстро. Варианты, доступные тем, кто предпочитает писать код на других языках, мы рассмотрим в разделе «Другие привязки».

Нетривиальная функциональность

Некоторые возможности библиотеки **Libgit2** выходят за рамки системы **Git**. В частности, к ним относится расширяемость. Библиотека **Libgit2** позволяет для части

операций использовать нестандартные «серверные приложения», сохраняя объекты не так, как это делает система Git. Подобные «приложения» существуют для конфигурации, хранения ссылок и базы данных объектов.

Посмотрим, как это работает. Представленный здесь код заимствован из примеров, написанных разработчиками библиотеки Libgit2 (вы найдете их на странице <https://github.com/libgit2/libgit2-backends>). Вот настройка нестандартной базы данных объектов:

```
git_odb *odb;
int error = git_odb_new(&odb); (1)

git_repository *repo;
error = git_repository_wrap_odb(&repo, odb); (2)

git_odb_backend *my_backend;
error = git_odb_backend_mine(&my_backend, /*...*/); (3)

error = git_odb_add_backendodb, my_backend, 1); (4)
```

ПРИМЕЧАНИЕ

Ошибки в данном фрагменте перехватываются, но не обрабатываются. Надеемся, что ваш код будет лучше нашего.

Целиком процедура выглядит так:

1. Инициализируется «интерфейс» пустой базы данных объектов (Object DataBase, ODB), который послужит основой для реальной базы данных объектов.
2. Вокруг пустой базы данных объектов конструируется `git_repository`.
3. Инициализируется серверное приложение специализированной базы данных объектов.
4. Репозиторий настраивается на использование этого приложения.

Но что такое `git_odb_backend_mine`? Это ваша собственная реализация ODB, поэтому внутри вы можете делать все что угодно при условии корректного заполнения структуры `git_odb_backend`. Например, внутри может быть следующий код:

```
typedef struct {
    git_odb_backend parent;

    // Какой-то другой код
    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;
}
```

Здесь действует небольшое ограничение. Первым членом в `my_backend_struct` должна быть структура `git_odb_backend`; это гарантирует ожидаемое библиотекой Libgit2 распределение памяти. Остальные поля — на ваше усмотрение; структура может быть как большой, так и маленькой, в зависимости от того, что вам требуется.

Функция инициализации выделяет под эту структуру некий объем памяти, настраивает указанный вами контекст и заполняет поля поддерживаемой ею родительской структуры. Полный набор сигнатур вызовов вы найдете в файле `include/git2/sys/odb_backend.h` исходного кода библиотеки Libgit2; в каждом конкретном случае вы сами решаете, какие из них будут поддерживаться.

Другие привязки

Библиотека Libgit2 имеет привязки для множества языков. Здесь мы приведем только пару примеров. Полный список поддерживаемых языков очень широк и включает в себя, в частности, C++, Go, Node.js, Erlang и JVM на разных стадиях своего развития. Официальный список привязок можно найти, просматривая репозитории по адресу <https://github.com/libgit2>. Представленные здесь примеры кода демонстрируют, как получить сообщение фиксации для коммита, на который нацелен указатель HEAD (своего рода аналог команды `git log -1`).

Привязка LibGit2Sharp

Если вы пишете .NET- или Mono-приложения, вам поможет LibGit2Sharp (<https://github.com/libgit2/libgit2sharp>). В данном случае все написано на языке C#, и все прямые Libgit2-вызовы подаются через API из удобной среды CLR. Наш пример выглядит так:

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

Для приложений рабочего стола Windows существует даже пакет NuGet, позволяющий быстро приступить к работе.

Привязка objective-git

Если ваше приложение предназначено для платформы Apple, скорее всего, оно написано на языке Objective-C. Для этой среды в Libgit2 используется привязка Objective-Git (<https://github.com/libgit2/objective-git>). Вот пример программы:

```

GTRepository *repo =
[[GTRepository alloc] initWithURL:[NSURL URLWithString:@" /path/to/repo"]
 error:NULL];
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];

```

Привязка Objective-git полностью взаимозаменяема с привязкой Swift, так что не бойтесь отступить в сторону от языка Objective-C.

Привязка pygit2

Привязка Libgit2 для языка Python называется Pygit2 и доступна по адресу <http://www.pygit2.org/>. Наш пример программы:

```

pygit2.Repository(« /path/to/repo»)      # открытие репозитория
    .head.resolve()                       # получение прямой ссылки
    .get_object().message                  # получение коммита, чтение сообщения

```

Дополнительная информация

К сожалению, детальное рассмотрение возможностей библиотеки Libgit2 выходит за рамки темы данной книги. Чтобы лучше познакомиться с этой библиотекой, начните с чтения документации к API, расположенной по адресу <https://libgit2.github.com/libgit2>, а также с руководства на странице <https://libgit2.github.com/docs>. Сведения о привязках к другим языкам вы найдете в соответствующих файлах README и тестовых примерах; там часто встречаются небольшие учебные пособия и ссылки на дополнительные материалы.

Приложение В

Git-команды

В этой книге мы познакомили вас с десятками различных Git-команд, приложив немало усилий, чтобы рассказать о них по порядку, постепенно добавляя к повествованию новые команды. Но в результате информация об их смысле и предназначении оказалась рассеянной по всей книге.

В этом приложении вы найдете все упоминавшиеся в книге команды, сгруппированные по областям их применения. Вместе с кратким описанием каждой команды вы найдете ссылки на приведенные в книге примеры их применения.

Настройка и конфигурирование

Существуют две распространенные команды, востребованные как сразу после установки Git, так и в повседневной практике. Это команды **config** и **help**.

git config

Сотни операций в системе Git выполняются с параметрами, заданными по умолчанию. Но в большинстве случаев эти параметры можно заменить другими предустановленными значениями или вовсе воспользоваться собственным вариантом значения. Это относится к большинству вариантов настройки Git, от вашего имени до цветовой схемы терминала и используемого редактора. Есть несколько файлов, из которых команда **git config** считывает значения, что дает возможность задавать параметры как на глобальном уровне, так и на уровне конкретного репозитория.

Командой **git config** мы пользовались почти во всех главах.

- ❑ В разделе «Первая настройка Git» главы 1 мы, еще не начав работать с Git, воспользовались этой командой для задания своего имени, адреса электронной почты и предпочитаемого редактора.
- ❑ В разделе «Псевдонимы в Git» главы 2 мы показали, каким образом эта команда позволяет создавать краткие варианты команд с длинными списками параметров, избавляя нас от необходимости каждый раз все это набирать на клавиатуре.
- ❑ В разделе «Перемещение данных» главы 3 мы заставили команду **git pull** по умолчанию использовать параметр **--rebase**.
- ❑ В разделе «Хранение учетных данных» главы 7 мы настроили репозиторий, куда по умолчанию помещали HTTP-пароли.
- ❑ И наконец, этой команде посвящен практически весь раздел «Конфигурирование системы Git» главы 8.

git help

Команда **git help** служит для вывода на экран справочной информации по всем Git-командам. В этом приложении мы кратко описываем самые популярные команды, узнать же полный список всех их параметров и флагов поможет команда **git help <имя команды>**.

Мы познакомили вас с командой **git help** в разделе «Получение справочной информации» главы 1, а еще воспользовались ею в разделе «Настройка сервера» главы 4 для получения сведений о настройке ограниченной оболочки.

Копирование и создание проектов

Существуют два способа получения Git-репозитория. Можно скопировать существующий репозиторий или создать новый в какой-нибудь папке.

git init

Для превращения папки в новый Git-репозиторий, в котором можно будет контролировать версии файлов, достаточно воспользоваться командой **git init**.

- ❑ В разделе «Удаленные ветки» главы 3 мы кратко рассказали, как поменять заданное по умолчанию имя ветки **master**.
- ❑ Мы воспользовались этой командой для создания пустого голого репозитория для работы на сервере в разделе «Размещение на сервере голого репозитория» главы 4.
- ❑ Наконец, мы рассмотрели подробности функционирования этой команды в разделе «Канализация и фарфор» главы 10.

git clone

Команда **git clone** иногда выступает в роли оболочки для других команд. Она создает новую папку, переходит в нее и запускает команду **git init** для создания пустого Git-репозитория. Затем она добавляет удаленный репозиторий (**git remote add**) по URL-адресу, который вы ей передали (по умолчанию он получает имя **origin**), выполняет из этого репозитория команду **git fetch**, а затем выгружает самый последний коммит в рабочую папку командой **git checkout**.

Команда **git clone** встречается в этой книге десятки раз, но мы перечислим только самые интересные случаи ее применения.

- ❑ Первое знакомство и объяснение принципа действия этой команды вы найдете в разделе «Клонирование существующего репозитория» главы 2, где рассматривается несколько примеров.
- ❑ В главе 4 мы показали вам, как с помощью параметра **--bare** создать копию Git-репозитория без рабочей папки.
- ❑ В разделе «Пакеты» главы 7 мы пользовались этой командой для распаковки Git-репозитория.
- ❑ Наконец, в разделе «Клонирование проекта с подмодулями» главы 7 мы показали, как параметр **--recursive** до некоторой степени упрощает клонирование содержащего подмодули проекта.

По большому счету, эта команда фигурирует во всех главах, но мы перечислили только более-менее нестандартные варианты ее применения.

Фиксация состояния

Для базового рабочего процесса, включающего индексирование содержимого и его фиксацию в истории проекта, существует всего несколько команд.

git add

Команда **git add** перемещает из рабочей папки в область индексирования предназначенное для следующей фиксации содержимое. По умолчанию команда **git commit** работает только с областью индексирования, поэтому именно команда **git add** позволяет вам точно указать, что именно должно попасть в следующий коммит.

Это одна из ключевых Git-команд, соответственно она упоминается в книге десятки раз. Здесь же мы перечислим наиболее интересные варианты ее применения.

- ❑ Впервые команда появляется в разделе «Слежение за новыми файлами» главы 2, там же описаны детали ее применения.

- ❑ Мы упомянули, как эта команда используется для разрешения конфликтов слияния в разделе «Конфликты при слиянии» главы 3.
- ❑ Эта команда позволила нам добавить в область индексирования только определенные части измененного файла в разделе «Интерактивное индексирование» главы 7.
- ❑ Наконец, мы эмулировали низкоуровневую работу этой команды в разделе «Объекты-деревья» главы 10, чтобы дать вам представление о том, что скрыто от ваших глаз.

git status

Команда **git status** показывает состояние файлов в рабочей папке и области индексирования. С ее помощью вы можете узнать, какие файлы подверглись изменению, но пока не индексировались, а какие уже индексировались, но пока не зафиксированы. При обычном формате вывода также выводятся подсказки о том, как изменить состояние файлов.

В первый раз эта команда и оба формата ее вывода — как основной, так и упрощенный — рассматривается в разделе «Проверка состояния файлов» главы 2. И хотя эта команда неоднократно возникает в разных главах, практически все варианты ее использования описаны именно тут.

git diff

Команда **git diff** служит для вычисления разницы между любыми двумя деревьями. Это может быть разница между рабочей папкой и областью индексирования (собственно команда **git diff**), между областью индексирования и последним коммитом (**git diff --staged**) или между двумя коммитами (**git diff master branchB**).

- ❑ В первый раз мы воспользовались этой командой в разделе «Просмотр индексированных и неиндексированных изменений» главы 2, демонстрируя, как можно узнать, какие изменения уже находятся в области индексирования, а какие еще нет.
- ❑ В разделе «Рекомендации по созданию коммитов» главы 5 перед фиксацией состояния мы воспользовались этой командой с параметром **--check** для поиска возможных проблем с пробелами.
- ❑ В разделе «Просмотр вносимых изменений» главы 5 мы научили вас эффективно сравнивать ветки, применяя синтаксис **git diff A...B**.
- ❑ Мы воспользовались параметром **-w**, чтобы избавиться от проблем с пробелами, а также показали, как параметры **--theirs**, **--ours** и **--base** позволяют сравнивать различные варианты конфликтующих файлов в разделе «Более сложные варианты слияния» главы 7.

- ❑ Наконец, добавление к этой команде параметра `--submodule` позволило нам в разделе «Подмодули» главы 7 сравнить изменения в подмодулях проекта.

git difftool

Команда `git difftool` просто запускает внешний инструмент для вывода на экран разницы между двумя деревьями, когда вы хотите воспользоваться чем-то, отличным от встроенной команды `git diff`.

git commit

Команда `git commit` берет все содержимое файлов, проиндексированное с помощью команды `git add`, и записывает в базу данных новый постоянный снимок состояния, а затем сдвигает на этот снимок указатель ветки.

- ❑ С основами создания коммитов вы познакомились в разделе «Фиксация изменений» главы 2. Там же мы продемонстрировали, как при помощи флага `-a` убрать выполнение команды `git add` из рутинного рабочего цикла и как флаг `-m` передает сообщение фиксации в командную строку, избавляя вас от необходимости пользоваться редактором.
- ❑ В разделе «Отмена изменений» главы 2 мы рассказали о параметре `-amend`, позволяющем повторить последний коммит.
- ❑ В разделе «Суть ветвления» главы 3 было более подробно рассмотрено, какое действие производит команда `git commit` и почему она функционирует именно таким образом.
- ❑ Мы научили вас создавать зашифрованные подписи для коммитов, добавляя к этой команде флаг `-s` в разделе «Подпись коммитов» главы 7.
- ❑ Наконец, в разделе «Объекты-коммиты» главы 10 мы детально рассмотрели, как эта команда работает на низком уровне.

git reset

Команда `git reset`, как легко догадаться по ее названию, в основном используется для отмены совершенных действий. Она перемещает указатель `HEAD`, в некоторых случаях меняя область индексирования, а если вы воспользуетесь параметром `--hard`, может внести изменения и в рабочую папку. В последнем случае существует опасность потери данных, поэтому перед применением этого варианта команды следует четко понимать, что и зачем вы делаете.

- ❑ Простейший пример применения команды `git reset` был исчерпывающе описан в разделе «Отмена индексирования» главы 2, где с ее помощью мы убрали из области индексирования файл, добавленный туда командой `git add`.
- ❑ В разделе «Команда reset» главы 7 мы детально объяснили все особенности применения этой команды.

- ❑ Команда `git reset --hard` позволила нам остановить процедуру слияния в разделе «Прерывание слияния» главы 7. Там же мы показали работу с командой `git merge --abort`, которая представляет собой своего рода оболочку для команды `git reset`.

git rm

Команда `git rm` применяется для удаления файлов из области индексирования или рабочей папки. Фактически ее действие обратное действию команды `git add`, ведь она индексирует данные, которые будут удалены при следующей фиксации состояния.

- ❑ Более-менее подробно команда `git rm` рассмотрена в разделе «Удаление файлов» главы 2, где мы продемонстрировали рекурсивное удаление файлов, а также удаление файлов из области индексирования при добавлении к команде параметра `--cached`. В последнем случае в рабочей папке файлы сохраняются.
- ❑ Единственный альтернативный вариант применения команды `git rm` показан в разделе «Удаление объектов» главы 10, где мы вкратце объяснили назначение параметра `--ignore-unmatch` при выполнении команды `git filter-branch`, которая просто не дает нам вывести сообщение об ошибке, если удаляемого файла не существует. Эта возможность востребована при написании сценариев.

git mv

Команда `git mv` представляет собой всего лишь удобный инструмент перемещения файла, после которого для нового файла выполняется команда `git add`, а для старого — команда `git rm`. Мы коротко рассмотрели ее в разделе «Перемещение файлов» главы 2.

git clean

Команда `git clean` используется для удаления из рабочей папки ненужных файлов. К таким файлам относятся временные артефакты сборки или файлы конфликтов слияния. Различные параметры этой команды и сценарии ее применения мы рассмотрели в разделе «Очистка рабочей папки» главы 7.

Ветвления и слияния

За создание новых веток и их слияние друг с другом в системе Git отвечает всего несколько команд.

git branch

Команда `git branch`, по сути, представляет собой инструмент управления ветками. Она умеет выводить на экран список имеющихся у вас веток, создавать новые, а также удалять и переименовывать их.

- ❑ Этой команде посвящена большая часть главы 3, поэтому там она используется повсеместно. Впервые она появляется в разделе «Создание новой ветки», а большинство ее функциональных возможностей (вывод списков и удаление) рассматривается в разделе «Управление ветками».
- ❑ В разделе «Слежение за ветками» мы показали, как вариант команды **git branch** -и используется в слежении за ветками.
- ❑ Наконец, мы рассмотрели, как эта команда функционирует на низком уровне в разделе «Ссылки в Git» главы 10.

git checkout

Команда **git checkout** используется для перехода в другую ветку и выгрузки содержимого веток в рабочую папку.

- ❑ В первый раз она упоминается в разделе «Смена веток» главы 3 одновременно с командой **git branch**.
- ❑ Мы показали, как с помощью флага **--track** инициировать слежение за веткой в разделе «Слежение за ветками» главы 3.
- ❑ Мы использовали эту команду с параметром **--conflict=diff3** для повторного разрешения конфликтов файлов в разделе «Применение команды checkout» главы 7.
- ❑ А в разделе «Команда reset» все той же главы 7 мы подробно рассмотрели связь этой команды с командой **git reset**.
- ❑ Наконец, детали реализации этой команды были описаны в разделе «Указатель HEAD» главы 10.

git merge

Команда **git merge** служит для вставки содержимого одной или нескольких веток в ту ветку, в которой вы в данный момент находитесь. После этого команда двигает текущую ветку к результату слияния.

- ❑ Впервые команда **git merge** появляется в разделе «Основы ветвления» главы 3. После этого она в различных вариациях, но чаще всего в форме **git merge <имя ветки>**, где указывается ветка, содержимое которой вы хотите слить, появляется во всех главах.
- ❑ Мы рассказали вам, как слить все изменения в один коммит (система Git при этом выполняет слияние таким образом, что в результате возникает один новый коммит, не записывая историю ветки, куда вставляются данные), в самом конце раздела «Открытый проект, ветвление» главы 5.
- ❑ В разделе «Более сложные варианты слияния» главы 7 мы подробно рассмотрели процесс слияния и данную команду, в том числе такой ее вариант, как

-Xignore-all-whitespace, а также флаг --abort, позволяющий прервать проблемное слияние.

- ❑ Мы научили вас проверять подписи перед процедурой слияния в проектах, использующих GPG-ключи, в разделе «Подпись коммитов» главы 7.
- ❑ Наконец, в разделе «Слияние поддеревьев» все той же главы 7 вы познакомились с процедурой слияния двух поддеревьев.

git mergetool

Команда **git mergetool** просто загружает вспомогательную программу при возникновении проблем со слиянием в системе Git.

Мы вкратце упомянули о ней в разделе «Конфликты при слиянии» главы 3, а также подробно показали, как с ее помощью реализовать собственный инструмент слияния в разделе «Внешние инструменты для слияния и индикации изменений» главы 8.

git log

Команда **git log** используется для отображения достижимой записанной истории проекта, начиная с самого свежего коммита. По умолчанию она показывает только историю ветки, в которой вы сейчас находитесь, но можно настроить ее на вывод данных сразу о нескольких ветках. Кроме того, ее можно применять для индикации различий между ветками на уровне коммитов.

Эта команда фигурирует практически во всех главах книги, демонстрируя нам историю различных проектов.

- ❑ Мы познакомили вас с этой командой и более-менее подробно рассмотрели механизм ее применения в разделе «Просмотр истории версий» главы 2. Вы узнали, что параметры **-p** и **--stat** позволяют получить представление о том, что нового появилось в каждом коммите, а параметры **--pretty** и **--oneline** представляют историю более лаконично, особенно при выполнении фильтрации по дате создания и автору.
- ❑ В разделе «Создание новой ветки» главы 3 мы воспользовались параметром **--decorate** для индикации местоположения указателей веток в истории коммитов. А параметр **--graph** позволил нам посмотреть, как выглядит разошедшаяся история.
- ❑ В разделах «Работа в маленькой группе» главы 5 и «Диапазоны коммитов» главы 7 мы познакомили вас с синтаксисом **branchA...branchB**, позволяющим команде **git log** выбирать только коммиты, присутствующие в одной ветке, но отсутствующие в другой. В разделе «Диапазоны коммитов» этот прием применяется довольно интенсивно.
- ❑ В разделах «Протоколирование слияния» и «Три точки» главы 7 мы рассмотрели варианты синтаксиса **branchA...branchB** и **--left-right**, позволяющие

увидеть, что находится в одной или в другой ветке, но не в них обеих сразу. Кроме того, в разделе «Протоколирование слияния» мы познакомили вас с параметром `-merge`, помогающим при отладке конфликтов слияния, и показали, как с помощью параметра `--cc` увидеть конфликты слияния в истории проекта.

- ❑ В разделе «Сокращения журнала ссылок» главы 7 мы показали, как добавлением к команде параметра `-g` можно избавиться от необходимости обхода ветки, а сразу вывести журнал ссылок в Git.
- ❑ В разделе «Поиск» главы 7 мы рассмотрели, как с помощью параметров `-S` и `-L` реализуется достаточно сложный поиск в истории проекта. К примеру, было показано, как посмотреть историю развития какой-либо функциональной возможности.
- ❑ В разделе «Подпись коммитов» главы 7 вы увидели, как с помощью параметра `--show-signature` показать в выводе команды `git log` все коммиты со строкой проверки в зависимости от того, корректно ли они подписаны.

git stash

Команда `git stash` позволяет на время скрыть незафиксированные наработки, когда требуется освободить рабочую папку, а делать коммит для незавершенной работы вам не хочется. Эта команда детально рассмотрена в разделе «Скрытие и очистка» главы 7.

git tag

Команда `git tag` позволяет получить постоянную закладку на определенную точку в истории кода. Как правило, она используется для таких вещей, как версии.

- ❑ Эта команда впервые появилась и была детально рассмотрена в разделе «Теги» главы 2, а попрактиковались с ее применением мы в разделе «Идентификация устойчивых версий» главы 5.
- ❑ Кроме того, мы научили вас создавать подписанные GPG-ключом теги с помощью флага `-s` и проверять их с помощью флага `-v` в разделе «Подпись» главы 7.

Совместная работа и обновление проектов

В системе Git не так уж много команд, требующих доступа к сети. Практически все команды работают с локальной базой данных. Однако в ситуации, когда вы хотите поделиться с коллегами результатами своего труда или скачать чужие наработки, на помощь приходят команды для работы с удаленными репозиториями.

git fetch

Команда **git fetch** взаимодействует с удаленным репозиторием, скачивая оттуда всю отсутствующую у вас информацию и сохраняя ее в локальной базе данных.

- ❑ Впервые мы показали вам эту команду в разделе «Извлечение данных из удаленных репозиториях» главы 2, а затем представили ряд примеров ее применения в разделе «Удаленные ветки» главы 3.
- ❑ Кроме того, мы использовали эту команду в ряде примеров в разделе «Содействие проекту» главы 5.
- ❑ В разделе «Обращения к запросам на включение» главы 6 мы воспользовались ею для извлечения конкретной ссылки, лежащей за пределами нашего рабочего пространства, а в разделе «Пакеты» главы 7 вы узнали, как извлечь ссылку из пакета.
- ❑ В разделе «Спецификация ссылок» главы 10 мы рассмотрели специальные варианты спецификации ссылок, позволяющие команде **git fetch** отклоняться от предлагаемого по умолчанию поведения.

git pull

Команда **git pull**, по сути, представляет собой комбинацию Git-команд **fetch** и **git merge**. То есть система Git сначала извлекает информацию из указанного вами удаленного репозитория, а затем пытается вставить ее в текущую ветку.

- ❑ Кратко эта команда была описана в разделе «Извлечение данных из удаленных репозиториях» главы 2, а в разделе «Просмотр удаленных репозиториях» этой же главы было показано, как узнать, к каким изменениям приведет ее выполнение.
- ❑ Кроме того, в разделе «Перемещение после перемещения» главы 3 мы узнали, как эта команда помогает решать проблемы с перемещениями.
- ❑ В разделе «Проверка удаленных веток» главы 5 мы показали, как использовать эту команду с URL-адресом для извлечения изменений из удаленного репозитория без сохранения указанного адреса в списке.
- ❑ Наконец, в разделе «Подпись» главы 7 мы кратко упомянули, что параметр **--verify-signatures** позволяет проверять зашифрованные GPG-ключом подписи скачиваемых коммитов.

git push

Команда **git push** используется для взаимодействия с другим репозиторием, вычисления данных, которые есть в вашей локальной базе, но отсутствуют там, и передачи этих данных в упомянутый репозиторий. Так как в данном случае требуется право на запись в чужой репозиторий, команда прибегает к процедуре аутентификации.

- ❑ В первый раз эта команда появилась в разделе «Отправка данных в удаленный репозиторий» главы 2. Там мы рассмотрели основы отправки веток в удаленные репозитории. В разделе «Отправка данных» главы 3 мы подробнее рассмотрели специфику пересылки веток, а в следующем разделе — «Слежение за ветками» — вы узнали, как настроить в отслеживаемых ветках автоматическую передачу данных в удаленный репозиторий. В разделе «Ликвидация веток с удаленного севера» все той же главы 3 мы воспользовались флагом `--delete`, чтобы с помощью команды `git push` убрать ветку с удаленного сервера.
- ❑ В разделе «Содействие проекту» главы 5 вы найдете ряд примеров применения команды `git push` для публикации данных из веток в нескольких удаленных репозиториях одновременно.
- ❑ В разделе «Обмен тегами» главы 2 мы показали, как эта команда после добавления параметра `--tags` позволяет выкладывать в общий доступ созданные вами теги.
- ❑ В разделе «Публикация результатов редактирования подмодуля» главы 7 мы воспользовались параметром `--recurse-submodules`, чтобы проверить возможность публикации всех наших подмодулей перед их фактической отправкой на сервер.
- ❑ В разделе «Другие клиентские хуки» главы 8 мы коротко рассмотрели хук `pre-push`, который представляет собой сценарий, запускаемый перед отправкой данных на сервер, чтобы проверить допустимость этой отправки.
- ❑ Наконец, в разделе «Спецификация ссылок для отправки данных на сервер» главы 10 мы рассмотрели процесс передачи данных с полной спецификацией ссылок, а не с используемой обычно сокращенной версией. Это повышает точность выбора данных, которые вы хотите отправить в общий доступ.

git remote

Команда `git remote` представляет собой инструмент управления удаленными репозиториями. Она позволяет сохранять длинные URL-адреса в виде понятных коротких строк, например «origin», что избавляет вас от необходимости все время вводить длинные адреса. У вас может быть несколько удаленных репозиториях, а команда `git remote` дает возможность добавлять, редактировать и удалять их.

- ❑ Подробно эта команда рассматривается в разделе «Удаленные репозитории» главы 2, где представлены процессы вывода списка удаленных репозиториях, их добавления, удаления и переименования.
- ❑ Она появляется практически во всех последующих главах, причем всегда в одном и том же стандартном формате `git remote add <имя> <url>`.

git archive

Команда `git archive` создает архив файла из определенного снимка проекта. С ее помощью мы создали tarball-архив проекта для передачи его по сети в разделе «Подготовка устойчивой версии» главы 5.

git submodule

Команда **git submodule** используется для управления внешними репозиториями, вложенными в наши обычные репозитории. В роли таких вложенных репозиторий могут выступать библиотеки или другие ресурсы общего доступа. У команды **submodule** есть ряд внутренних команд (**add**, **update**, **sync** и т. п.) для управления такими ресурсами. Эта команда полностью описана в разделе «Подмодули» главы 7.

Проверка и сравнение

git show

Команда **git show** выводит Git-объекты на экран в простом и понятном пользователям виде. Обычно она применяется для получения информации о теге или коммите.

- ❑ Впервые мы воспользовались ею в разделе «Теги с комментариями» главы 2 для вывода сведений о теге, снабженном комментарием.
- ❑ Затем мы прибегли к этой команде в разделе «Выбор версии» главы 7, чтобы отобразить коммиты в зависимости от выбранной версии.
- ❑ Одним из самых интересных примеров применения команды **git show** продемонстрирован в разделе «Слияние файлов вручную» главы 7, где показано, как извлечь определенное содержимое файла на разных стадиях конфликта слияния.

git shortlog

Команда **git shortlog** кратко воспроизводит вывод команды **git log**. Она принимает многие из параметров этой команды, но выводит не список всех коммитов, а только их краткое описание, сгруппированное по авторам. В разделе «Команда **shortlog**» главы 5 вы найдете пример создания с помощью этой команды аккуратной сводки коммитов.

git describe

Команда **git describe** работает со всем, что можно трактовать как коммит, выводя удобную для чтения неизменяемую строку. Она создает описание коммита, что является хотя и не такой однозначной, зато более понятной характеристикой, чем контрольная сумма SHA.

Мы использовали команду **git describe** в разделах «Генерация номера сборки» и «Подготовка устойчивой версии» главы 5 для генерации названия нашего файла с новой версией.

Отладка

В системе Git есть команды, помогающие в процессе отладки искать проблемные места в коде. Они позволяют не только отследить момент возникновения проблемы, но и найти ее причину.

git bisect

Команда **git bisect** представляет собой чрезвычайно полезный инструмент отладки, позволяя путем автоматизированного двоичного поиска найти коммит, в котором в первый раз возникла проблема. Она полностью рассмотрена в разделе «Двоичный поиск» главы 7.

git blame

Команда **git blame** открывает примечания к файлу, указывающие, когда и кем в последний раз редактировалась каждая строка. Она помогает найти человека, которому можно задать вопросы по поводу конкретного фрагмента кода. Детально эта команда рассматривается в разделе «Примечания к файлам» главы 7.

git grep

Команда **git grep** позволяет легко найти строку или регулярное выражение в любом файле вашего кода, в том числе в более старых версиях проекта. Она исчерпывающе описана в разделе «Команда Git Grep» главы 7.

Исправления

В Git есть группа команд, рассматривающих коммиты с точки зрения вносимых ими изменений. Фактически цепочка коммитов воспринимается как цепочка исправлений. Именно в таком стиле эти команды помогают нам управлять ветками.

git cherry-pick

Команда **git cherry-pick** берет изменения, вносимые одним коммитом, и пытается повторно применить их в виде нового коммита в текущей ветке. Эта возможность полезна в ситуации, когда нужно взять из ветки один или два коммита, а не сливать ветку целиком со всеми внесенными в нее изменениями. Этот процесс описывается и демонстрируется в разделе «Схема с перемещением и отбором» главы 5.

git rebase

Команда **git rebase** представляет собой автоматизированную версию команды **cherry-pick**. Она определяет набор коммитов и по очереди выбирает из них данные, как бы перенося ветку на новое место.

- ❑ Процедура перемещения детально рассмотрена в разделе «Перемещение данных» главы 3. Коснулись мы и проблем, возникающих в процессе совместной работы при переносе уже опубликованных веток.
- ❑ Практический пример применения этой команды рассмотрен в разделе «Замена» главы 7, где мы разбили историю проекта на два репозитория, попутно воспользовавшись флагом `--onto`.
- ❑ Мы задействовали эту команду при разрешении конфликта слияния, возникшего в процессе переноса в разделе «Команда `rebase`» главы 7.
- ❑ Кроме того, в разделе «Редактирование нескольких сообщений фиксации» главы 7 мы использовали эту команду в интерактивном сценарии, иницилируемом параметром `-i`.

git revert

Команда `git revert`, по сути, является антиподом команды `git cherry-pick`. Она создает новый коммит, который действует диаметрально противоположно целевому коммиту, отменяя все внесенные последним изменения. Мы воспользовались ею в разделе «Отмена коммита» главы 7 для отмены результатов слияния.

Электронная почта

Множество использующих Git проектов, включая саму систему Git, активно поддерживаются через списки рассылки. В систему Git встроено несколько инструментов, облегчающих этот процесс, от генерации исправлений, которые можно легко отправить по электронной почте, до применения этих изменений непосредственно из папки «Входящие».

git apply

Команда `git apply` применяет исправления, созданные командой `git diff` или даже командой `GNU diff`. Ее действие практически аналогично действию команды `patch`, хотя есть и ряд мелких отличий. Процесс применения этой команды и обстоятельства, в которых она может потребоваться, мы описали в разделе «Исправления, присланные по почте» главы 5.

git am

Команда `git am` используется для применения исправлений из ящика входящих сообщений, особенно если он находится в формате mbox. Она требуется, когда вы получаете исправления по электронной почте и хотите применить их к проекту.

- ❑ Применение команды `git am`, в том числе с параметрами `--resolved`, `-i` и `-3`, рассмотрено в разделе «Команда `am`» главы 5.

- ❑ Существует набор хуков, которые могут оказаться полезными в рабочих схемах с использованием команды `git am`. Все они рассмотрены в разделе «Хуки для работы с электронной почтой» главы 8.
- ❑ Мы также воспользовались этой командой, чтобы применить исправления из запроса на включения в формате GitHub в разделе «Уведомления по электронной почте» главы 6.

git format-patch

Команда `git format-patch` генерирует набор исправлений в формате mbox, которые можно отправить в список рассылки. Пример содействия проекту посредством данной команды был рассмотрен в разделе «Открытый проект, электронная почта» главы 5.

git send-email

Команда `git send-email` применяется для отправки по электронной почте исправлений, сгенерированных командой `git format-patch`. Пример вклада в проект с использованием этой команды вы найдете в разделе «Открытый проект, электронная почта» главы 5.

git request-pull

Команда `git request-pull` генерирует пример тела сообщения для отправки. Если у вас есть ветка на открытом сервере и вы хотите поделиться с кем-то результатами своего труда, не отправляя исправления по электронной почте, запустите эту команду и пошлите получателю выводимые ею данные.

Процесс применения команды `git request-pull` мы продемонстрировали в разделе «Открытый проект, ветвление» главы 5.

Внешние системы

В системе Git есть ряд команд для интеграции с другими системами контроля версий.

git svn

Команда `git svn` используется для работы с Subversion-сервером. Это означает, что вы можете использовать систему Git в качестве клиента для получения изменений с Subversion-сервера и отправки туда собственных коммитов. Эта команда подробно рассматривается в разделе «Git и Subversion» главы 9.

git fast-import

Для других систем контроля версий и для импорта произвольным образом отформатированных данных применяется команда **git fast-import**. Она быстро преобразует данные в формат, понятный системе Git. Подробно она рассмотрена в разделе «Другие варианты импорта» главы 9.

Администрирование

Для тех, кто занимается администрированием Git-репозитория или хочет внести какие-то глобальные исправления в проект, система Git предоставляет набор административных команд.

git gc

Команда **git gc** запускает в репозитории сборщик мусора, удаляя из базы данных ненужные файлы, а все остальное упаковывая в более компактный формат. Обычно эта команда запускается автоматически, но при желании ее можно вызвать и вручную. Примеры ее применения были рассмотрены в разделе «Обслуживание репозитория» главы 10.

git fsck

Команда **git fsck** служит для проверки внутренней базы данных на наличие ошибок и утрату целостности. Мы воспользовались ею всего один раз в разделе «Восстановление данных» главы 10 для поиска недостижимых ни по одной ссылке объектов.

git reflog

Команда **git reflog** просматривает во время вашей работы журнал положения вершин веток с целью поиска коммитов, которые могут быть потеряны при переписи истории.

- ❑ В основном эта команда рассматривалась в разделе «Сокращения журнала ссылок» главы 7, где мы показали стандартный вариант ее применения, а также воспользовались вариантом **git log -g** для просмотра тех же самых данных, которые предоставляет команда **git log**.
- ❑ На практике мы применили эту команду для восстановления утраченной ветки в разделе «Восстановление данных» главы 10.

git filter-branch

Команда **git filter-branch** по заданному шаблону переписывает содержимое коммитов, позволяя, к примеру, полностью удалить файл из истории или извлечь во вложенную папку результаты фильтрации целого репозитория.

- ❑ В разделе «Удаление файла из всех коммитов» главы 7 мы объяснили механизм работы этой команды и познакомились с такими ее параметрами, как `--commit-filter`, `--subdirectory-filter` и `--tree-filter`.
- ❑ В разделах «Мост Git-p4» и «TFS» главы 9 мы воспользовались этой командой для приведения в нужный вид импортированных внешних репозиториях.

Служебные команды

В этой книге вы найдете и ряд низкоуровневых служебных команд.

- ❑ Во-первых, это команда `ls-remote`, с помощью которой в разделе «Обращения к запросам на включение» главы 6 мы просматривали ссылки на сервере.
- ❑ Мы применяли команду `ls-files` в таких разделах главы 7, как «Слияние файлов вручную», «Команда `rerere`» и «Индекс», чтобы понять, как на самом деле выглядит область индексирования.
- ❑ Кроме того, в разделе «Ссылки из веток» главы 7 упоминается команда `rev-parse`, позволяющая практически любую строку превратить в SHA-объект.
- ❑ Подавляющее большинство низкоуровневых служебных команд рассмотрено в посвященной им главе 10. В других главах мы старались по возможности избегать применения этих команд.

Об авторах

Скотт Чакон — соучредитель и директор по инвестициям проекта GitHub. Также он является администратором официального сайта Git (git-scm.com). Скотт Чакон принимал участие в десятках конференций, посвященных Git, GitHub и будущему этих систем.



Бен Страуб — разработчик, долгое время участвовавший в проекте Libgit2, лектор и преподаватель по Git международного уровня, книголюб, вечный исследователь, человек, осваивающий искусство создания прекрасного программного обеспечения. С женой и двумя детьми он проживает в Портланде, штат Орегон.



С. Чакон, Б. Штрауб
Git для профессионального программиста
Перевела на русский И. Рузмайкина

Заведующая редакцией
Ведущий редактор
Литературный редактор
Художник
Корректоры
Верстка

Ю. Сергиенко
Н. Римицан
А. Жданов
В. Шимкевич
Н. Викторова, В. Сайко
Л. Панич