ПЕРВОНАЧАЛЬНАЯ НАСТРОЙКА GIT

Необходимо настроить среду для работы с Git. Это нужно сделать только один раз — при обновлении версии Git настройки сохранятся. Но, при необходимости, вы можете поменять их в любой момент, выполнив те же команды снова. В состав Git входит утилита *git config*, которая позволяет просматривать и настраивать параметры, контролирующие все аспекты работы Git, а также его внешний вид. Эти параметры могут быть сохранены в трёх местах:

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

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

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

Имя пользователя

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

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

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

Выбор редактора

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

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

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

Проверка настроек

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

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

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

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

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

КАК ПОЛУЧИТЬ ПОМОЩЬ?

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

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

Например, так можно открыть руководство по команде config:

```
$ git help config
```

Эти команды хороши тем, что ими можно пользоваться всегда, даже без подключения к сети. Если данного руководства недостаточно и вам нужна персональная помощь, вы можете попытаться поискать её на каналах #git и #github IRC-сервера Freenode (irc.freenode.net).

Теперь у вас должно быть общее понимание, что такое Git, и в чём его отличие от тех ЦСКВ, которыми вы, вероятно, пользовались раньше. Также у вас должна быть установлена рабочая версия Git с вашими личными настройками.

ОСНОВЫ GIT

Рассмотрим все базовые команды, необходимые вам для решения подавляющего большинства задач возникающих при работе с Git. Изучим как настраивать и инициализировать репозиторий, начинать и прекращать версионный контроль файлов, а также подготавливать и фиксировать изменения, изучим как настроить в Git игнорирование отдельных файлов или их групп, как быстро и просто отменить ошибочные изменения, как просмотреть историю проекта и изменения между отдельными коммитами (commit), а также как отправлять (push) и получать (pull) изменения в/из удалённого (remote) репозитория.

СОЗДАНИЕ GIT-РЕПОЗИТОРИЯ

Для создания Git-репозитория существуют два основных подхода. Первый подход — импорт в Git уже существующего проекта или каталога. Второй — клонирование уже существующего репозитория с сервера.

СОЗДАНИЕ РЕПОЗИТОРИЯ В СУЩЕСТВУЮЩЕМ КАТАЛОГЕ

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

```
$ git init
```

Эта команда создаёт в текущем каталоге новый подкаталог с именем .git содержащий все необходимые файлы репозитория — основу Git-репозитория. **На этом этапе ваш проект ещё не находится под версионным контролем.**

Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит проиндексировать эти файлы и осуществить первую фиксацию изменений. Осуществить это вы можете с помощью нескольких команд git add указывающих индексируемые файлы, а затем commit:

```
$ git add *.c
$ git add README
$ git commit -m “initial project version”
```

Мы разберём, что делают эти команды чуть позже. На данном этапе, у вас есть Git-репозиторий с добавленными файлами и начальным коммитом.

КЛОНИРОВАНИЕ СУЩЕСТВУЮЩЕГО РЕПОЗИТОРИЯ

Если вы желаете получить копию существующего репозитория Git, например, проекта, в котором вы хотите поучаствовать, то вам нужна команда *git clone*. Если вы знакомы с другими системами контроля версий, такими как Subversion, то заметите, что команда называется *clone*, а не *checkout*. Это важное отличие — Git получает копию практически всех данных, что есть на сервере. **Каждая версия каждого файла из истории проекта забирается (*pulled*) с сервера, когда вы выполняете *git clone*.** Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов, для того чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования (вы можете потерять часть серверных перехватчиков (server-side hooks) и т.п., но все данные, помещённые под версионный контроль, будут сохранены).

Клонирование репозитория осуществляется командой git clone [url]. Например, если вы хотите клонировать библиотеку Ruby Git, известную как Grit, вы можете сделать это следующим образом:

```
$ git clone git://github.com/schacon/grit.git
```

Эта команда создаёт каталог с именем *grit*, инициализирует в нём каталог *.git*, скачивает все данные для этого репозитория и создаёт (*checks out*) рабочую копию последней версии. Если вы зайдёте в новый каталог *grit*, вы увидите в нём проектные файлы, пригодные для работы и использования. Если вы хотите клонировать репозиторий в каталог, отличный от grit, можно это указать в следующем параметре командной строки:

```
$ git clone git://github.com/schacon/grit.git mygrit
```

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

В Git реализовано несколько транспортных протоколов, которые вы можете использовать. В предыдущем примере использовался протокол git://, вы также можете встретить http(s):// или user@server:/path.git, использующий протокол передачи SSH. Далее будут рассмотрены все доступные варианты конфигурации сервера для обеспечения доступа к вашему Git-репозиторию, а также рассмотрим их достоинства и недостатки.

ЗАПИСЬ ИЗМЕНЕНИЙ В РЕПОЗИТОРИЙ

Итак, имеется настоящий Git-репозиторий и рабочая копия файлов для некоторого проекта. Необходимо делать некоторые изменения и фиксировать “снимки” состояния (snapshots) этих изменений в репозитории каждый раз, когда проект достигает состояния, которое хотелось бы сохранить.

Запомните, каждый файл в вашем рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые). Отслеживаемые файлы — это те файлы, которые были в последнем слепке состояния проекта (snapshot); они могут быть неизменёнными, изменёнными или подготовленными к коммиту (staged). Неотслеживаемые файлы — это всё остальное, любые файлы в рабочем каталоге, которые не входили в последний слепок состояния и не подготовлены к коммиту. Когда впервые клонируется репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что они только взяты их из хранилища (*checked them out*) и не отредактированы.

Как только файлы отредактированы, Git будет рассматривать их как изменённые, т.к. они были изменены с момента последнего коммита. Вы индексируете (*stage*) эти изменения и затем фиксируете все индексированные изменения, а затем цикл повторяется. Этот жизненный цикл изображён на рисунке 1.

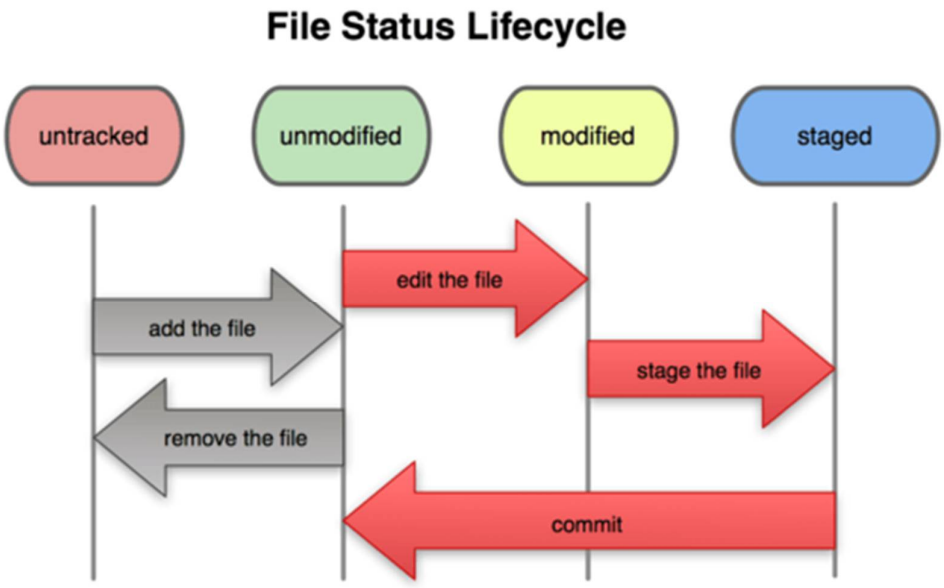

Рисунок 1 – Жизненный цикл состояний файлов

ОПРЕДЕЛЕНИЕ СОСТОЯНИЯ ФАЙЛОВ

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

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

Это означает, что у вас чистый рабочий каталог, другими словами — в нём нет отслеживаемых изменённых файлов. Git также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены. Данная команда сообщает вам на какой ветке (branch) вы сейчас находитесь. Сейчас это ветка master —ветка по умолчанию.

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

```
$ vim 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 неотслеживаемый можно по тому, что он находится в секции "Untracked files" в выводе команды *status*. Статус "неотслеживаемый файл", по сути, означает, что Git видит файл, отсутствующий в предыдущем снимке состояния (коммите); Git не станет добавлять его в ваши коммиты, пока вы его явно об этом не попросите. Это предохранит от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые вы и не думали добавлять.

ОТСЛЕЖИВАНИЕ НОВЫХ ФАЙЛОВ

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

```
$ git add README
```

Если вы снова выполните команду status, то увидите, что файл README теперь отслеживаемый и индексированный:

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

Вы можете видеть, что файл проиндексирован по тому, что он находится в секции “Changes to be committed”. Если вы выполните коммит в этот момент, то версия файла, существовавшая на момент выполнения вами команды *git add*, будет добавлена в историю снимков состояния. Когда вы ранее выполнили *git init*, вы затем выполнили *git add (файлы)* — это было сделано для того, чтобы добавить файлы в каталоге под версионный контроль. Команда *git add* принимает параметром путь к файлу или каталогу, если это каталог, то команда рекурсивно добавляет (индексирует) все файлы в данном каталоге.

ИНДЕКСАЦИЯ ИЗМЕНЁННЫХ ФАЙЛОВ

Модифицируем файл, уже находящийся под версионным контролем. Если вы измените отслеживаемый файл Table.java и после этого снова выполните команду status, то результат будет примерно следующим:

```
$ 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)
#
#   modified:   Table.java
#
```

Файл Table.java находится в секции “Changes not staged for commit” — это означает, что отслеживаемый файл был изменён в рабочем каталоге, но пока не проиндексирован. Чтобы проиндексировать его, необходимо выполнить команду *git add* (это многофункциональная команда, она используется для добавления под версионный контроль новых файлов, для индексации изменений, а также для других целей, например для указания файлов с исправленным конфликтом слияния). Выполним *git add*, чтобы проиндексировать Table.java, а затем снова выполним *git status*:

```
$ git add Table.java
$ git status
# On branch master
```

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

Теперь оба файла проиндексированы и войдут в следующий коммит. В этот момент вы, предположим, вспомнили одно небольшое изменение, которое вы хотите сделать в Table.java до фиксации. Вы открываете файл, вносите и сохраняете необходимые изменения и готовы к коммиту. Выполним ещё раз команду *git status*:

```
$ vim Table.java
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   Table.java
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   Table.java
#
```

Теперь Table.java отображается как проиндексированный и непроиндексированный одновременно. Как такое возможно? Такая ситуация наглядно демонстрирует, что Git индексирует файл в точности в том состоянии, в котором он находился, когда вы выполнили команду *git add*. Если вы выполните коммит сейчас, то файл Table.java попадёт в коммит в том состоянии, в котором он находился, когда вы последний раз выполняли команду *git add*, а не в том, в котором он находится в вашем рабочем каталоге в момент выполнения *git commit*. Если вы изменили файл после выполнения *git add*, вам придётся снова выполнить *git add*, чтобы проиндексировать последнюю версию файла:

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

ИГНОРИРОВАНИЕ ФАЙЛОВ

Зачастую, у вас имеется группа файлов, которые вы не только не хотите автоматически добавлять в репозиторий, но и видеть в списках неотслеживаемых. К таким файлам обычно относятся автоматически генерируемые файлы (различные лог-файлы, результаты сборки программ и

т.п.). В таком случае, вы можете создать файл .gitignore с перечислением шаблонов соответствующих таким файлам. Вот пример файла .gitignore:

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

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

Файлы .gitignore как и любые другие файлы могут быть добавлены в репозиторий (выполните git add .gitignore и git commit).

К шаблонам в файле .gitignore применяются следующие правила:

- Пустые строки, а также строки, начинающиеся с #, игнорируются.
- Можно использовать стандартные glob шаблоны.
- Можно заканчивать шаблон символом слэша (/) для указания каталога.
- Можно инвертировать шаблон, использовав восклицательный знак (!) в качестве первого символа.

Glob-шаблоны представляют собой упрощённые регулярные выражения используемые командными интерпретаторами, где:

- символ * соответствует 0 или более символам;
- последовательность [abc] — соответствует любому символу из указанных в скобках (в данном примере a, b или c);
- знак вопроса (?) соответствует одному символу;
- [0-9] соответствует любому символу из интервала (в данном случае от 0 до 9).

Вот ещё один пример файла .gitignore:

```
# комментарий — эта строка игнорируется
# не обрабатывать файлы, имя которых заканчивается на .a
*.a
# НО отслеживать файл lib.a, несмотря на то, что мы игнорируем все .a
файлы с помощью предыдущего правила
!lib.a
# игнорировать только файл TODO находящийся в корневом каталоге, не
#относится к файлам вида subdir/TODO
/TODO
# игнорировать все файлы в каталоге build/
build/
# игнорировать doc/notes.txt, но не doc/server/arch.txt
doc/*.txt
# игнорировать все .txt файлы в каталоге doc/
```

*doc/**/*.txt*

Шаблон **/ доступен в Git, начиная с версии 1.8.2.

ПРОСМОТР ИНДЕКСИРОВАННЫХ И НЕИНДЕКСИРОВАННЫХ ИЗМЕНЕНИЙ

Если результат работы команды *git status* недостаточно информативен и вам хочется знать, что конкретно поменялось, а не только какие файлы были изменены — вы можете использовать команду git diff. Если команда *git status* отвечает на вопросы о том, что вы изменили, но ещё не проиндексировали, и что вы проиндексировали и собираетесь фиксировать слишком обобщённо, то *git diff* показывает непосредственно добавленные и удалённые строки — собственно заплатку (*patch*).

Допустим, вы снова изменили и проиндексировали файл README, а затем изменили файл Table.java без индексирования. Если вы выполните команду status, вы опять увидите что-то вроде:

```
$ 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)
#
#   modified:   Table.java
#
```

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

```
$ git diff
diff --git a/Table.java b/Table.java
index 3cb747f..da65585 100644
--- a/Table.java
+++ b/Table.java
@@ -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 вашего рабочего каталога с содержимым индекса. Результат показывает ещё не проиндексированные изменения.

Если вы хотите посмотреть, что вы проиндексировали и что войдёт в следующий коммит, вы можете выполнить git diff --cached. (В Git версии 1.6.1 и выше, вы также можете использовать git diff --staged, которая легче

запоминается.) Эта команда сравнивает ваши индексированные изменения с последним коммитом:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

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

Другой пример: вы проиндексировали файл Table.java и затем изменили его, вы можете использовать git diff для просмотра как индексированных изменений в этом файле, так и тех, что пока не проиндексированы:

```
$ git add Table.java
$ echo '# test line' >> Table.java
$ git status
# On branch master
#
# Changes to be committed:
#
#   modified:   Table.java
#
# Changes not staged for commit:
#
#   modified:   Table.java
#
```

Теперь вы можете используя git diff посмотреть непроиндексированные изменения

```
$ git diff
diff --git a/Table.java b/Table.java
index e445e28..86b2f7c 100644
--- a/Table.java
+++ b/Table.java
@@ -127,3 +127,4 @@ end
 main()

 ##pp Grit::GitRuby.cache_client.stats
+# test line
```

а также уже проиндексированные, используя git diff --cached:

```
$ git diff --cached
diff --git a/Table.java b/Table.java
index 3cb747f..e445e28 100644
--- a/Table.java
```

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

$ git commit

Эта команда откроет выбранный вами текстовый редактор. (Редактор устанавливается системной переменной $EDITOR — обычно это vim или emacs, хотя вы можете установить ваш любимый с помощью команды git config --global core.editor). В редакторе будет отображён следующий текст (это пример окна Vim'a):

```
#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)
#
#       new file:   README
#       modified:   Table.java
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

Вы можете видеть, что комментарий по умолчанию для коммита содержит закомментированный результат работы ("выхлоп") команды *git status* и ещё одну пустую строку сверху. Вы можете удалить эти комментарии и набрать своё сообщение или же оставить их для напоминания о том, что вы фиксируете. (Для ещё более подробного напоминания, что же именно вы поменяли, можете передать аргумент -v в команду git commit. Это приведёт к тому, что в комментарий будет также помещена дельта/diff изменений, таким образом вы сможете точно увидеть всё, что сделано.) Когда вы выходите из редактора, Git создаёт для вас коммит с этим сообщением (удаляя комментарии и вывод diff'a).

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

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"
 2 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 README
```

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

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

ИГНОРИРОВАНИЕ ИНДЕКСАЦИИ

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

```
$ git status
# On branch master
#
# Changes not staged for commit:
#
#   modified:   Table.java
#
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Обратите внимание на то, что в данном случае перед коммитом вам не нужно выполнять git add для файла Table.java.

УДАЛЕНИЕ ФАЙЛОВ

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

Если вы просто удалите файл из своего рабочего каталога, он будет показан в секции “Changes not staged for commit” (“Изменённые но не обновлённые” — читай не проиндексированные) вывода команды git status:

```
$ rm grit.gemspec
$ git status
# On branch master
#
# Changes not staged for commit:
#      (use "git add/rm <file>..." to update what will be
committed)
#
#       deleted:    grit.gemspec
#
```

Затем, если выполнить команду git rm, удаление файла попадёт в индекс:

```
$ 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
#
```

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

Можно также удалить файл из индекса, оставив его при этом в рабочем каталоге. Другими словами, вы можете захотеть оставить файл на винчестере, и убрать его из-под наблюдения Git. Это полезно, если вы забыли добавить что-то в файл .gitignore и по ошибке проиндексировали, например, большой файл с лог-данными, или кучу промежуточных файлов компиляции. Чтобы сделать это, используйте опцию --cached:

```
$ git rm --cached readme.txt
```

В команду git rm можно передавать файлы, каталоги или glob-шаблоны. Это означает, что вы можете написать:

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

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

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

Эта команда удаляет все файлы, чьи имена заканчиваются на ~.

ПЕРЕМЕЩЕНИЕ ФАЙЛОВ

В отличие от многих других систем версионного контроля, Git не отслеживает перемещение файлов явно. Когда вы переименовываете файл в Git, в нём не сохраняется никаких метаданных, говорящих о том, что файл был переименован. Однако, Git довольно умён в плане обнаружения

перемещений постфактум. Таким образом, наличие в Git команды mv выглядит несколько странным. Если вам хочется переместить файл в Git, вы можете написать:

```
$ git mv file_from file_to
```

и это сработает. На самом деле, если вы выполните эту команду и посмотрите на статус, вы увидите, что Git считает, что произошло переименование файла:

```
$ git mv README.txt README
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    README.txt -> README
#
```

Однако, это эквивалентно выполнению следующих команд:

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

Git неявно определяет, что произошло переименование, поэтому неважно, переименуете вы файл так или используя команду mv. Единственное отличие состоит лишь в том, что mv — это одна команда вместо трёх — это функция для удобства. Важнее другое — можно использовать любой удобный способ, чтобы переименовать файл, и затем воспользоваться add/rm перед коммитом.

Отмена изменений

На любой стадии может возникнуть необходимость что-либо отменить. Рассмотрим несколько основных инструментов для отмены произведённых изменений. **Не всегда можно отменить сами отмены!** Это одно из немногих мест в Git, где вы можете потерять свою работу, если сделаете что-то неправильно.

Изменение последнего коммита

Одна из типичных отмен происходит тогда, когда вы делаете коммит слишком рано, забыв добавить какие-то файлы, или ошиблись с комментарием к коммиту. Если необходимо сделать этот коммит ещё раз, то можно выполнить commit с опцией --amend:

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

Эта команда берёт индекс и использует его для коммита. Если после последнего коммита не было никаких изменений (например, вы запустили приведённую команду сразу после предыдущего коммита), то состояние проекта будет абсолютно таким же и всё, что вы измените, это комментарий к коммиту.

Появится всё тот же редактор для комментариев к коммитам, но уже с введённым комментарием к последнему коммиту. Можно отредактировать это сообщение так же, как обычно, и оно перепишет предыдущее.

Для примера, если после совершения коммита необходимо проиндексировать изменения в файле, которые надо добавить в этот коммит, то можно сделать выполнить следующие команды:

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

Все три команды вместе дают один коммит — второй коммит заменяет результат первого.

Отмена индексации файла

В следующих двух разделах мы продемонстрируем, как переделать изменения в индексе и в рабочем каталоге. Приятно то, что команда, используемая для определения состояния этих двух вещей, дополнительно напоминает о том, как отменить изменения в них. Приведём пример. Допустим, вы внесли изменения в два файла и хотите записать их как два отдельных коммита, но случайно набрали git add * и проиндексировали оба файла. Как теперь отменить индексацию одного из двух файлов? Команда git status напомнит вам об этом:

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

Сразу после надписи “Changes to be committed”, написано использовать git reset HEAD <файл>... для исключения из индекса. Отменим индексацию файла benchmarks.rb:

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
# 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 немного странновата, но она работает. Файл benchmarks.rb изменён, но не проиндексирован.

Отмена изменений файла

Что, если вы поняли, что не хотите оставлять изменения, внесённые в файл benchmarks.rb? Как быстро отменить изменения, вернуть то состояние, в котором он находился во время последнего коммита (или первоначального клонирования, или какого-то другого действия, после которого файл попал в рабочий каталог)? В выводе для последнего примера, неиндексированная область выглядит следующим образом:

```
# 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 довольно сказано, как отменить сделанные изменения (новые версии Git, начиная с 1.6.1, делают это; если у версия старее, то её рекомендуем обновить).

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

Как вы видите, изменения были отменены. Вы должны понимать, что это опасная команда: все сделанные изменения в этом файле пропали — вы просто скопировали поверх него другой файл. Никогда не используйте эту команду, если вы не полностью уверены, что этот файл вам не нужен. Если вам нужно просто сделать, чтобы он не мешал, мы рассмотрим прятание (stash) и ветвление далее.

Помните, что всё, что является частью коммита в Git, почти всегда может быть восстановлено. Даже коммиты, которые находятся на ветках, которые были удалены, и коммиты переписанные с помощью --amend могут быть восстановлены. Несмотря на это, всё, что никогда не попадало в коммит, скорее всего уже нельзя увидите снова.

ВЕТВЛЕНИЕ В GIT

Почти каждая СКВ имеет в какой-то форме поддержку ветвления. Ветвление означает, что вы отклоняетесь от основной линии разработки и продолжаете работу, не вмешиваясь в основную линию. Во многих СКВ это в некотором роде дорогостоящий процесс, зачастую требующий от вас создания новой копии каталога с исходным кодом, что может занять продолжительное время для больших проектов.

Некоторые говорят, что модель ветвления Git это его “killer feature“ (преимущество) и она безусловно выделяет Git в СКВ-сообществе.

Способ ветвления в Git чрезвычайно легковесен, что делает операции ветвления практически мгновенными и переключение между ветками обычно так же быстрым. В отличие от многих других СКВ, Git поощряет процесс

работы, при котором ветвление и слияние осуществляется часто, даже по несколько раз в день. Понимание и владение этой функциональностью даёт уникальный мощный инструмент и может буквально изменить то, как вы ведёте разработку.

ЧТО ТАКОЕ ВЕТКА?

Git хранит данные не как последовательность изменений или дельт, а как последовательность снимков состояния (snapshot).

Когда вы создаёте коммит в Git, Git записывает в базу объект-коммит, который содержит указатель на снимок состояния, записанный ранее в индекс, метаданные автора и комментария и ноль и более указателей на коммиты, являющиеся прямыми предками этого коммита: ноль предков для первого коммита, один — для обычного коммита и несколько — для коммита, полученного в результате слияния двух или более веток.

Для наглядности предположим, что у вас есть каталог, содержащий три файла, и вы хотите добавить их все в индекс и сделать коммит. При добавлении файлов в индекс для каждого из них вычислится контрольная сумма (SHA-1 хеш), затем эти версии файлов будут сохранены в Git-репозиторий (Git обращается к ним как к двоичным данным), а их контрольные суммы добавятся в индекс:

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

Когда создаётся коммит, выполняя команду git commit, Git вычисляет контрольную сумму каждого подкаталога (в нашем случае только корневого каталога) и сохраняет эти объекты-деревья в Git-репозиторий. Затем Git создаёт объект для коммита, в котором есть метаданные и указатель на объект-дерево для корня проекта. Таким образом, Git сможет воссоздать текущее состояние, когда будет нужно.

Git-репозиторий теперь содержит пять объектов: по одному блобу для содержимого каждого из трёх файлов, одно дерево, в котором перечислено содержимое каталога и определено соответствие имён файлов и блобов, и один коммит с указателем на объект-дерево для корня и со всеми метаданными коммита. Схематично данные в этом Git-репозитории выглядят так, как показано на рисунке 3-1.

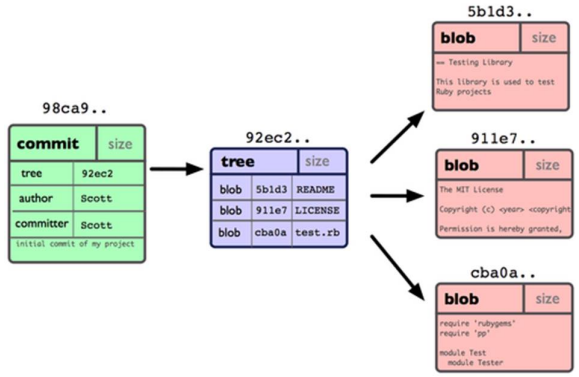

Рисунок 3-1. Данные репозитория с единственным коммитом

Если вы сделаете некоторые изменения и создадите новый коммит, то следующий коммит сохранит указатель на коммит, который шёл

непосредственно перед ним. После следующих двух коммитов история может выглядеть, как на рисунке 3-2.

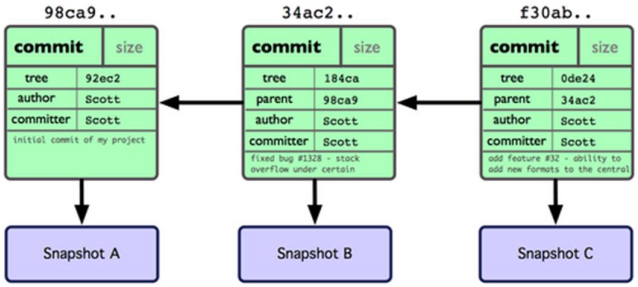

Рисунок 3-2. Данные объектов Git для нескольких коммитов

Ветка в Git — это просто легковесный подвижный указатель на один из этих коммитов. Ветка по умолчанию в Git называется master. Когда вы создаёте коммиты на начальном этапе, вам дана ветка master, указывающая на последний сделанный коммит. При каждом новом коммите она сдвигается вперёд автоматически.

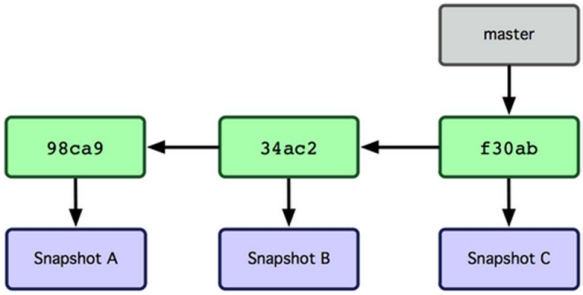

Рисунок 3-3. Ветка указывает на историю коммитов

Что произойдёт, если вы создадите новую ветку? Этим вы создадите новый указатель, который можно будет перемещать. Скажем, создадим новую ветку под названием testing. Это делается командой git branch:

$ git branch testing

Эта команда создаст новый указатель на тот самый коммит, на котором вы сейчас находитесь (см. рис. 3-4).

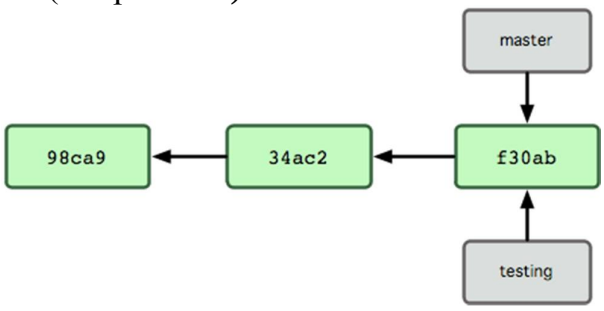

Рисунок 3-4. Несколько веток, указывающих на историю коммитов

Откуда Git узнает, на какой ветке вы находитесь в данный момент? Он хранит специальный указатель, который называется HEAD (верхушка). Учтите, что это сильно отличается от концепции HEAD в других СКВ, таких как Subversion или CVS. В Git это указатель на локальную ветку, на которой вы находитесь. В данный момент вы всё ещё на ветке master. Команда git

branch только создала новую ветку, она не переключила вас на неё (см. рис. 3-5).

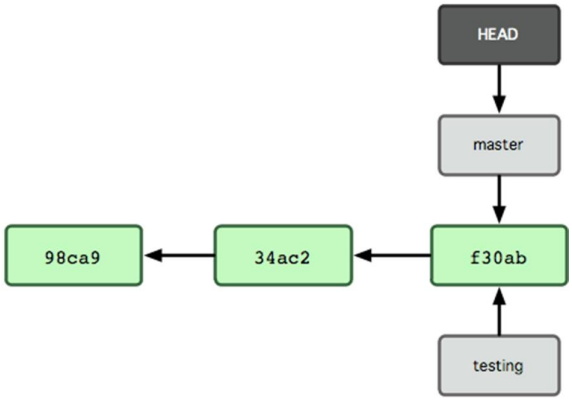

Рисунок 3-5. Файл HEAD указывает на текущую ветку

Чтобы перейти на существующую ветку, вам надо выполнить команду git checkout. Давайте перейдём на новую ветку testing:

```
$ git checkout testing
```

Это действие передвинет HEAD так, чтобы тот указывал на ветку testing (см. рис. 3-6).

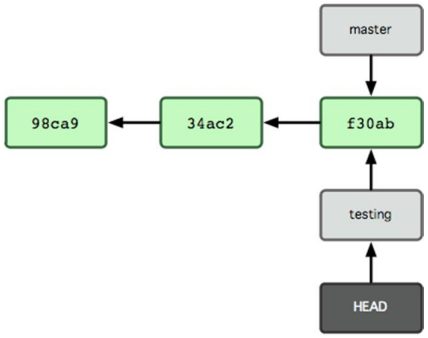

Рисунок 3-6. HEAD указывает на другую ветку после переключения веток

В чём же важность этого? Давайте сделаем ещё один коммит:

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

На рисунке 3-7 показан результат.

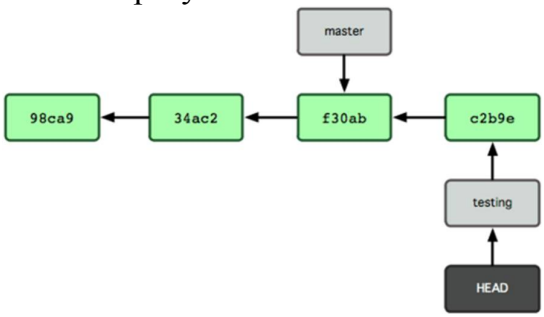

Рисунок 3-7. Ветка, на которую указывает HEAD, движется вперёд с каждым коммитом

Это интересно, потому что теперь ваша ветка testing передвинулась вперёд, но ветка master всё ещё указывает на коммит, на котором вы были, когда выполняли git checkout, чтобы переключить ветки. Давайте перейдём обратно на ветку master:

```
$ git checkout master
```

На рисунке 3-8 можно увидеть результат.

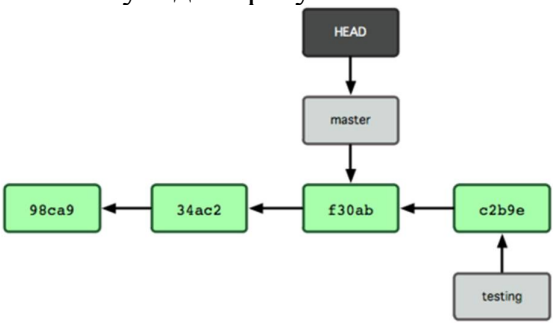

Рисунок 3-8. HEAD перемещается на другую ветку при выполнении команды checkout

Эта команда выполнила два действия. Она передвинула указатель HEAD назад на ветку master и вернула файлы в рабочем каталоге назад, в соответствие со снимком состояния, на который указывает master. Это также означает, что изменения, которые вы делаете, начиная с этого момента, будут ответвляться от старой версии проекта. Это, по сути, откатывает изменения, которые вы временно делали на ветке testing, так что дальше вы можете двигаться в другом направлении.

Давайте снова внесём немного изменений и сделаем коммит:

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

Теперь история вашего проекта разветвилась (рис. 3-9). Вы создали новую ветку, перешли на неё, поработали на ней немного, переключились обратно на основную ветку и выполнили другую работу. Оба эти изменения изолированы в отдельных ветках: вы можете переключаться туда и обратно между ветками и слить их, когда будете готовы. И всё это было сделано простыми командами branch и checkout.

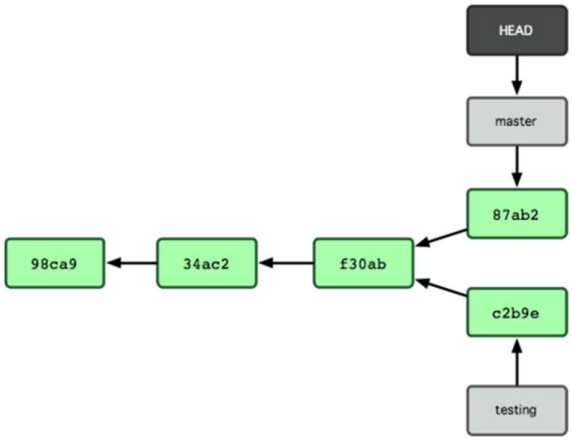

Рисунок 3-9. История с разошедшимися ветками

Из-за того, что ветка в Git на самом деле является простым файлом, который содержит 40 символов контрольной суммы SHA-1 коммита, на который он указывает, создание и удаление веток практически беззатратно. Создание новой ветки настолько же быстрое и простое, как запись 41 байта в файл (40 символов + символ новой строки).

Это разительно отличается от того, как в большинстве СКВ делается ветвление. Там это приводит к копированию всех файлов проекта в другой каталог. Это может занять несколько секунд или даже минут, в зависимости от размера проекта, тогда как в Git это всегда происходит моментально. Также благодаря тому, что мы запоминаем предков для каждого коммита, поиск нужной базовой версии для слияния уже автоматически выполнен, и в общем случае слияние делается легко. Эти особенности помогают поощрять разработчиков к частому созданию и использованию веток.

ОСНОВЫ ВЕТВЛЕНИЯ И СЛИЯНИЯ

Давайте рассмотрим ветвление и слияние на простом примере с таким процессом работы, который вы могли бы использовать в настоящей разработке. Мы выполним следующие шаги:

1. Поработаем над веб-сайтом.
2. Создадим ветку для работы над новой задачей.
3. Выполним некоторую работу на этой ветке.

На этом этапе вам поступит звонок о том, что сейчас критична другая проблема, и её надо срочно решить. Мы сделаем следующее:

1. Вернёмся на ветку для версии в производстве.
2. Создадим ветку для исправления ошибки.
3. После тестирования ветки с исправлением сольём её обратно и отправим в продакшн.
4. Вернёмся к своей исходной задаче и продолжим работать над ней.

Основы ветвления

Для начала представим, что вы работаете над своим проектом и уже имеете пару коммитов (см. рис. 3-10).

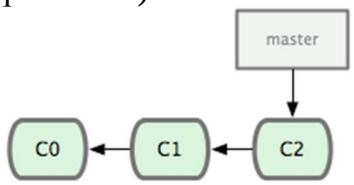

Рисунок 3-10. Короткая и простая история коммитов

Вы решили, что вы будете работать над проблемой №53 из системы отслеживания ошибок, используемой вашей компанией. Разумеется, Git не привязан к какой-то определенной системе отслеживания ошибок. Так как проблема №53 является обособленной задачей, над которой вы собираетесь работать, мы создадим новую ветку и будем работать на ней. Чтобы создать ветку и сразу же перейти на неё, вы можете выполнить команду git checkout с ключом -b:

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

Это сокращение для:

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

Рисунок 3-11 демонстрирует результат.

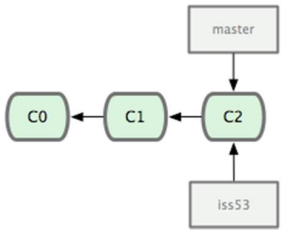

Рисунок 3-11. Создание новой ветки / указателя

Во время работы над своим веб-сайтом вы делаете несколько коммитов. Эти действия сдвигают ветку iss53 вперёд потому, что вы на неё перешли (то есть ваш HEAD указывает на неё; рис. 3-12):

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

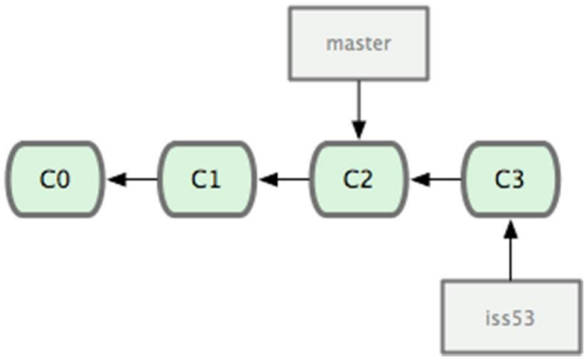

Рисунок 3-12. Ветка iss53 передвинулась вперёд во время работы

Теперь вы получаете звонок о том, что есть проблема с веб-сайтом, которую необходимо немедленно устранить. С Git вам нет нужды делать исправления для неё поверх тех изменений, которые вы уже сделали в iss53, и нет необходимости прикладывать много усилий для отмены этих изменений перед тем, как вы сможете начать работать над решением срочной проблемы. Всё, что вам нужно сделать, это перейти на ветку master.

Однако, прежде чем сделать это, учтите, что если в вашем рабочем каталоге или индексе имеются незафиксированные изменения, которые конфликтуют с веткой, на которую вы переходите, Git не позволит переключить ветки. Лучше всего при переключении веток иметь чистое рабочее состояние. Существует несколько способов добиться этого (а именно, прятанье (stash) работы и правка (amend) коммита). А на данный момент представим, что все изменения были добавлены в коммит, и теперь вы можете переключиться обратно на ветку master:

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

Теперь рабочий каталог проекта находится точно в таком же состоянии, что и в момент начала работы над проблемой №53, так что вы можете сконцентрироваться на исправлении срочной проблемы. Очень важно запомнить: Git возвращает ваш рабочий каталог к снимку состояния того коммита, на который указывает ветка, на которую вы переходите. Он добавляет, удаляет и изменяет файлы автоматически, чтобы гарантировать,

что состояние вашей рабочей копии идентично последнему коммиту на ветке.

Итак, вам надо срочно исправить ошибку. Давайте создадим для этого ветку, на которой вы будете работать (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]: created 3a0874c: "fixed the broken email address"
 1 files changed, 0 insertions(+), 1 deletions(-)
```

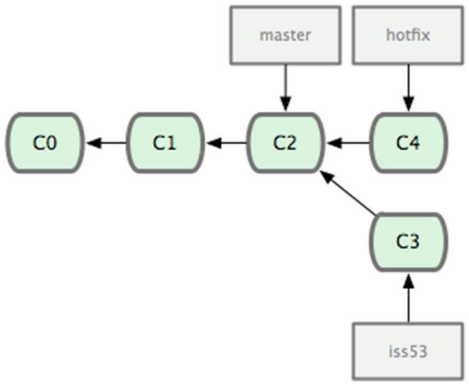

Рисунок 3-13. Ветка для решения срочной проблемы базируется на ветке master

Вы можете запустить тесты, убедиться, что решение работает, и слить (merge) изменения назад в ветку master, чтобы включить их в продукт. Это делается с помощью команды git merge:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
 README |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```

Наверное, вы заметили фразу "Fast forward" в этом слиянии. Так как ветка, которую мы слили, указывала на коммит, являющийся прямым родителем коммита, на котором мы сейчас находимся, Git просто сдвинул её указатель вперёд. Иными словами, когда вы пытаетесь слить один коммит с другим таким, которого можно достигнуть, проследовав по истории первого коммита, Git поступает проще, перемещая указатель вперёд, так как нет расходящихся изменений, которые нужно было бы сливать воедино. Это называется "перемотка" (fast forward).

Ваши изменения теперь в снимке состояния коммита, на который указывает ветка master, и вы можете включить изменения в продукт (рис. 3-14).

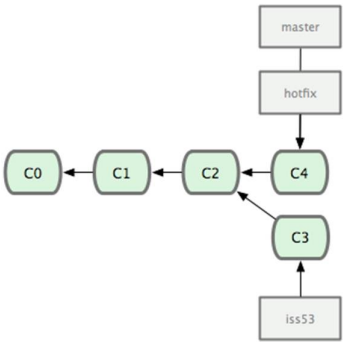

Рисунок 3-14. После слияния ветка master указывает туда же, куда и ветка hotfix

После того как очень важная проблема решена, вы готовы вернуться обратно к тому, над чем вы работали перед тем, как вас прервали. Однако, сначала удалите ветку hotfix, так как она больше не нужна — ветка master уже указывает на то же место. Вы можете удалить ветку с помощью опции -d к git branch:

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

Теперь вы можете вернуться обратно к рабочей ветке для проблемы №53 и продолжить работать над ней (рис. 3-15):

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53]: created ad82d7a: "finished the new footer [issue 53]"
 1 files changed, 1 insertions(+), 0 deletions(-)
```

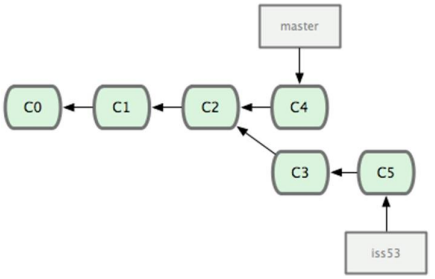

Рисунок 3-15. Ветка iss53 может двигаться вперёд независимо

Стоит напомнить, что работа, сделанная на ветке hotfix, не включена в файлы на ветке iss53. Если вам это необходимо, вы можете слить ветку master в ветку iss53 посредством команды git merge master. Или же вы можете подождать с интеграцией изменений до тех пор, пока не решите включить изменения на iss53 в продуктовую ветку master.

ОСНОВЫ СЛИЯНИЯ

Допустим, вы разобрались с проблемой №53 и готовы объединить эту ветку и свой master. Чтобы сделать это, мы сольём ветку iss53 в ветку master точно так же, как мы делали это ранее с веткой hotfix. Всё, что вам нужно

сделать, — перейти на ту ветку, в которую вы хотите слить свои изменения, и выполнить команду git merge:

```
$ git checkout master
$ git merge iss53
Merge made by recursive.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Это слияние немного отличается от слияния, сделанного ранее для ветки hotfix. В данном случае история разработки разделилась в некоторой точке. Так как коммит на той ветке, на которой вы находитесь, не является прямым предком для ветки, которую вы сливаете, Git придётся проделать кое-какую работу. В этом случае Git делает простое трёхходовое слияние, используя при этом те два снимка состояния репозитория, на которые указывают вершины веток, и общий для этих двух веток снимок-прародитель. На рисунке 3-16 выделены три снимка состояния, которые Git будет использовать для слияния в данном случае.

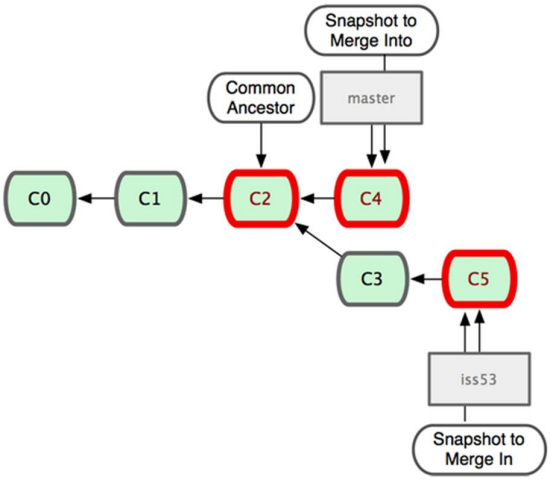

Рисунок 3-16. Git автоматически определяет наилучшего общего предка для слияния веток

Вместо того чтобы просто передвинуть указатель ветки вперёд, Git создаёт новый снимок состояния, который является результатом трёхходового слияния, и автоматически создаёт новый коммит, который указывает на этот новый снимок состояния (см. рис. 3-17). Такой коммит называют коммит-слияние, так как он является особенным из-за того, что имеет больше одного предка.

Стоит отметить, что Git сам определяет наилучшего общего предка для слияния веток; в CVS или Subversion (версии ранее 1.5) этого не происходит. Разработчик должен сам указать основу для слияния. Это делает слияние в Git гораздо более простым занятием, чем в других системах.

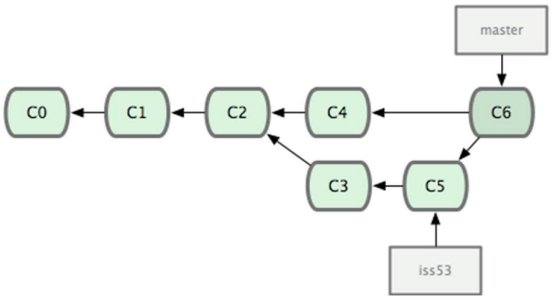

Рисунок 3-17. Git автоматически создаёт новый коммит, содержащий результаты слияния

Теперь, когда вы осуществили слияние ваших наработок, ветка iss53 вам больше не нужна. Можете удалить её и затем вручную закрыть карточку (ticket) в своей системе:

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

УПРАВЛЕНИЕ ВЕТКАМИ

Команда git branch делает несколько больше, чем просто создаёт и удаляет ветки. Если вы выполните её без аргументов, то получите простой список имеющихся у вас веток:

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

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

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

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

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

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

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

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

```
  testing
```

Вы увидите оставшуюся ветку. Так как она содержит ещё не слитые наработки, попытка удалить её командой git branch -d не увенчается успехом:

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

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

УДАЛЁННЫЕ ВЕТКИ

Удалённые ветки — это ссылки на состояние веток в ваших удалённых репозиториях. Это локальные ветки, которые нельзя перемещать; они двигаются автоматически всякий раз, когда вы осуществляете связь по сети. Удалённые ветки действуют как закладки для напоминания о том, где ветки в удалённых репозиториях находились во время последнего подключения к ним.

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

Всё это, возможно, сбивает с толку, поэтому давайте рассмотрим пример. Скажем, у вас в сети есть свой Git-сервер на git.ourcompany.com. Если вы с него что-то склонируете (clone), Git автоматически назовёт его origin, заберёт оттуда все данные, создаст указатель на то, на что там указывает ветка master, и назовёт его локально origin/master (но вы не можете его двигать). Git также сделает вам вашу собственную ветку master, которая будет начинаться там же, где и ветка master в origin, так что вам будет с чем работать (см. рис. 3-22).

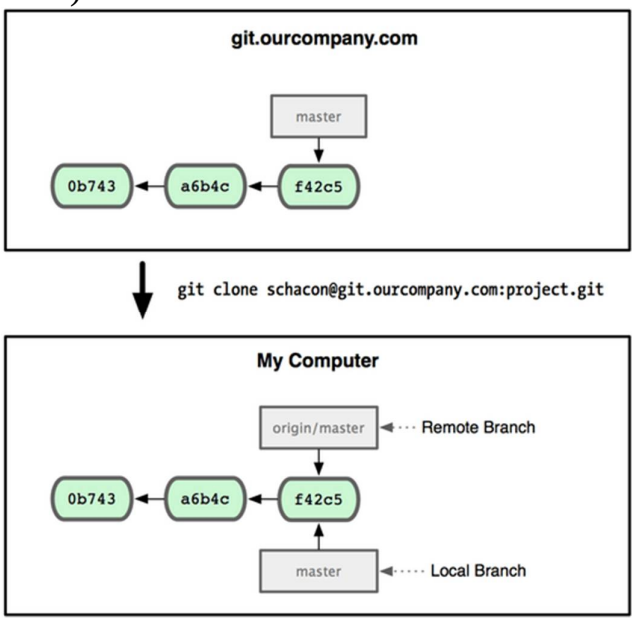

Рисунок 3-22. Клонирование Git-проекта даёт вам собственную ветку master и origin/master, указывающий на ветку master в origin

Если вы сделаете что-то в своей локальной ветке master, а тем временем кто-то ещё отправит (push) изменения на git.ourcompany.com и обновит там ветку master, то ваши истории продолжатся по-разному. Ещё, до тех пор, пока вы не свяжетесь с сервером origin, ваш указатель origin/master не будет сдвигаться (рис. 3-23).

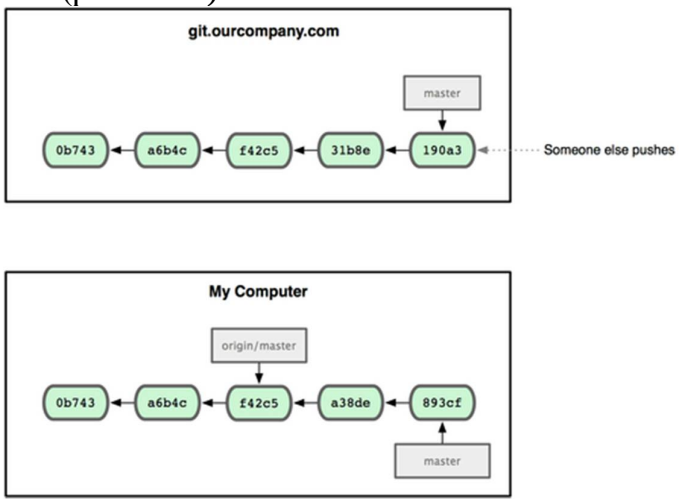

Рисунок 3-23. При выполнении локальной работы и отправке кем-то изменений на удалённый сервер каждая история продолжается по-разному

Для синхронизации вашей работы выполняется команда git fetch origin. Эта команда ищет, какому серверу соответствует origin (в нашем случае это git.ourcompany.com); извлекает оттуда все данные, которых у вас ещё нет, и обновляет ваше локальное хранилище данных; сдвигает указатель origin/master на новую позицию (см. рис. 3-24).

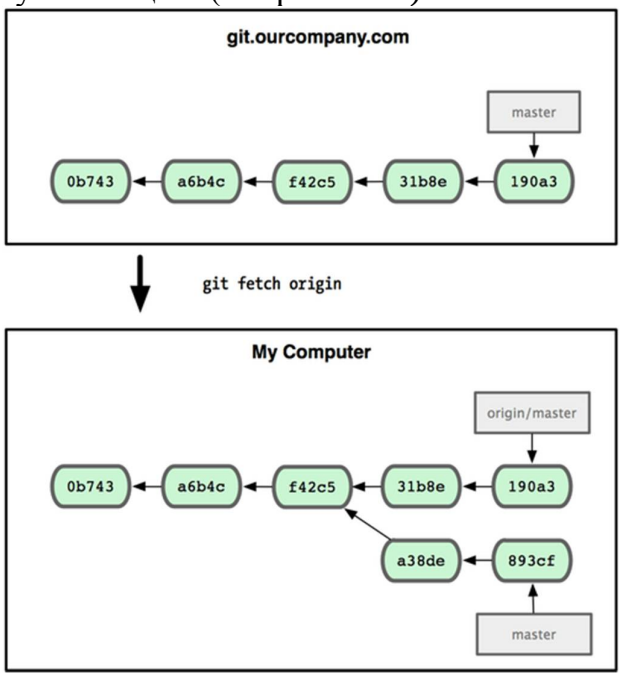

Рисунок 3-24. Команда git fetch обновляет ваши удалённые ссылки

Чтобы продемонстрировать то, как будут выглядеть удалённые ветки в ситуации с несколькими удалёнными серверами, предположим, что у вас есть

ещё один внутренний Git-сервер, который используется для разработки только одной из ваших команд разработчиков. Этот сервер находится на git.team1.ourcompany.com. Вы можете добавить его в качестве новой удалённой ссылки на проект, над которым вы сейчас работаете с помощью команды git remote add так же, как было описано в главе 2. Дайте этому удалённому серверу имя teamone, которое будет сокращением для полного URL (рис. 3-25).

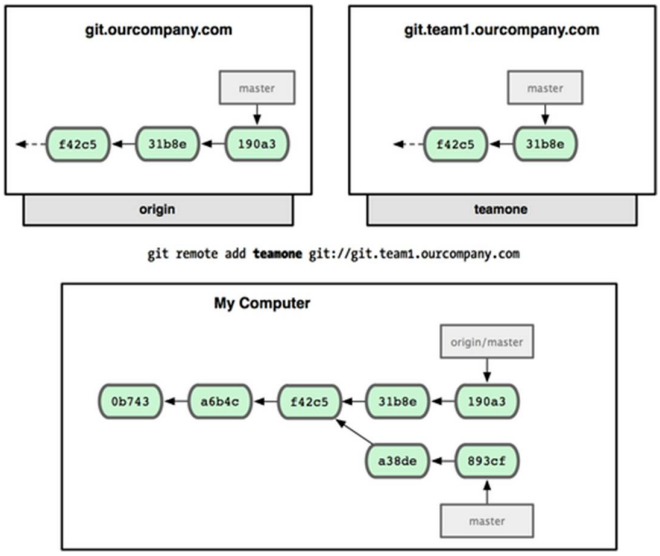

Рисунок 3-25. Добавление дополнительного удалённого сервера

Теперь можете выполнить git fetch teamone, чтобы извлечь всё, что есть на сервере и нет у вас. Так как в данный момент на этом сервере есть только часть данных, которые есть на сервере origin, Git не получает никаких данных, но выставляет удалённую ветку с именем teamone/master, которая указывает на тот же коммит, что и ветка master на сервере teamone (рис. 3-26).

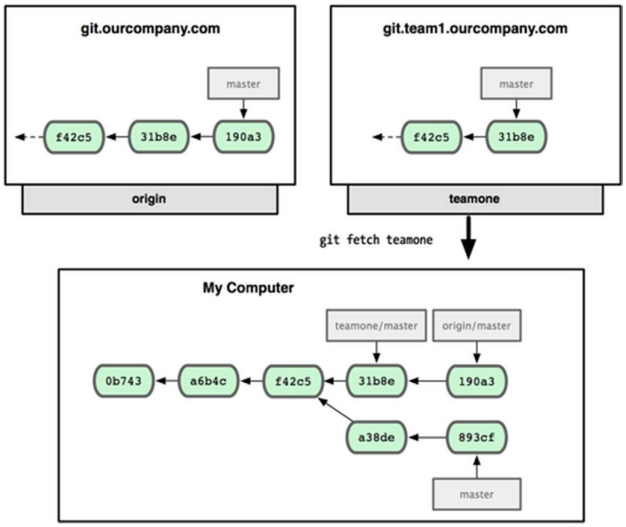

Рисунок 3-26. У вас появилась локальная ссылка на ветку master на teamone-е

Отправка изменений

Когда вы хотите поделиться веткой с окружающими, вам необходимо отправить (push) её на удалённый сервер, на котором у вас есть права на

запись. Ваши локальные ветки автоматически не синхронизируются с удалёнными серверами — вам нужно явно отправить те ветки, которыми вы хотите поделиться. Таким образом, вы можете использовать свои личные ветки для работы, которую вы не хотите показывать, и отправлять только те тематические ветки, над которыми вы хотите работать с кем-то совместно.

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

```
$ git push origin serverfix
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      serverfix -> serverfix
```

Это в некотором роде сокращение. Git автоматически разворачивает имя ветки serverfix до refs/heads/serverfix:refs/heads/serverfix, что означает “возьми мою локальную ветку serverfix и обнови из неё удалённую ветку serverfix”. Мы подробно обсудим часть с refs/heads/ в главе 9, но обычно её можно опустить. Вы также можете выполнить git push origin serverfix:serverfix — произойдёт то же самое — здесь говорится “возьми мой serverfix и сделай его удалённым serverfix”. Можно использовать этот формат для отправки локальной ветки в удалённую ветку с другим именем. Если вы не хотите, чтобы ветка называлась serverfix на удалённом сервере, то вместо предыдущей команды выполните git push origin serverfix:awesomebranch. Так ваша локальная ветка serverfix отправится в ветку awesomebranch удалённого проекта.

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

```
$ git fetch origin
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

Важно отметить, что когда при получении данных у вас появляются новые удалённые ветки, вы не получаете автоматически для них локальных редактируемых копий. Другими словами, в нашем случае вы не получите новую ветку serverfix — только указатель origin/serverfix, который вы не можете менять.

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

```
$ git checkout -b serverfix origin/serverfix
```

```
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

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

ОТСЛЕЖИВАНИЕ ВЕТОК

Получение локальной ветки с помощью git checkout из удалённой ветки автоматически создаёт то, что называется *отслеживаемой веткой*. Отслеживаемые ветки — это локальные ветки, которые напрямую связаны с удалённой веткой. Если, находясь на отслеживаемой ветке, вы наберёте git push, Git уже будет знать, на какой сервер и в какую ветку отправлять изменения. Аналогично выполнение git pull на одной из таких веток сначала получает все удалённые ссылки, а затем автоматически делает слияние с соответствующей удалённой веткой.

При клонировании репозитория, как правило, автоматически создаётся ветка master, которая отслеживает origin/master, поэтому git push и git pull работают для этой ветки "из коробки" и не требуют дополнительных аргументов. Однако, вы можете настроить отслеживание и других веток удалённого репозитория. Простой пример, как это сделать, вы увидели только что — git checkout -b [ветка] [удал. сервер]/[ветка]. Если вы используете Git версии 1.6.2 или более позднюю, можете также воспользоваться сокращением --track:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

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

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

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

УДАЛЕНИЕ ВЕТОК НА УДАЛЁННОМ СЕРВЕРЕ

Скажем, вы и ваши соавторы закончили с нововведением и слили его в ветку master на удалённом сервере (или в какую-то другую ветку, где хранится стабильный код). Вы можете удалить ветку на удалённом сервере, используя несколько бестолковый синтаксис git push [удал. сервер] :[ветка]. Чтобы удалить ветку serverfix на сервере, выполните следующее:

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

Хлоп. Нет больше ветки на вашем сервере. Вам может захотеться сделать закладку на текущей странице, так как эта команда вам понадобится, а синтаксис вы, скорее всего, забудете. Можно запомнить эту команду вернувшись к синтаксису git push [удал. сервер] [лок. ветка]:[удал. ветка],

который мы рассматривали немного раньше. Опуская часть [лок. ветка], вы, по сути, говорите “возьми ничто в моём репозитории и сделай так, чтобы в [удал. ветка] было то же самое”.

ПЕРЕМЕЩЕНИЕ

В Git есть два способа включить изменения из одной ветки в другую: merge (слияние) и rebase (перемещение). В этом разделе вы узнаете, что такое перемещение, как его осуществлять, почему это удивительный инструмент и в каких случаях вам не следует его использовать.

Основы перемещения

Если мы вернёмся назад к одному из ранних примеров из раздела про слияние (рис. 3-27), увидим, что мы разделили свою работу на два направления и сделали коммиты на двух разных ветках.

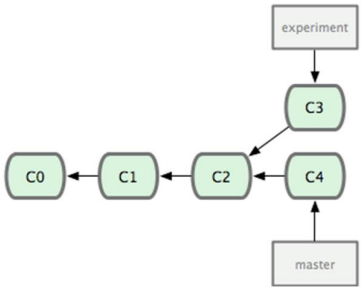

Рисунок 3-27. Впервые разделенная история коммитов

Наиболее простое решение для объединения веток, как мы уже выяснили, команда merge. Эта команда выполняет трёхходовое слияние между двумя последними снимками состояний из веток (C3 и C4) и последним общим предком этих двух веток (C2), создавая новый снимок состояния (и коммит), как показано на рисунке 3-28.

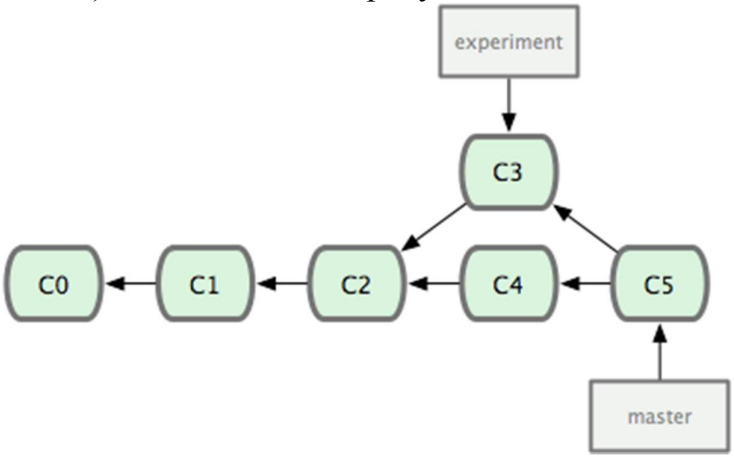

Рисунок 3-28. Слияние ветки для объединения разделившейся истории разработки

Однако, есть и другой путь: вы можете взять изменения, представленные в C3, и применить их поверх C4. В Git это называется *перемещение* (rebasing). При помощи команды rebase вы можете взять все

изменения, которые попали в коммиты на одной из веток, и повторить их на другой.

Для этого примера надо выполнить следующее:

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

Перемещение работает следующим образом: находится общий предок для двух веток (на которой вы находитесь сейчас и на которую вы выполняете перемещение); для каждого из коммитов в текущей ветке берётся его дельта и сохраняется во временный файл; текущая ветка устанавливается на тот же коммит, что и ветка, на которую выполняется перемещение; и, наконец, одно за другим применяются все изменения. Рисунок 3-29 иллюстрирует этот процесс.

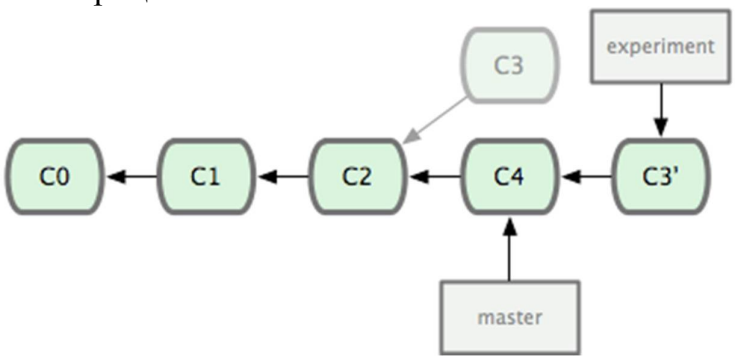

Рисунок 3-29. Перемещение изменений, сделанных в C3, на C4

На этом этапе можно переключиться на ветку master и выполнить слияние-перемотку (fast-forward merge) (рис. 3-30).

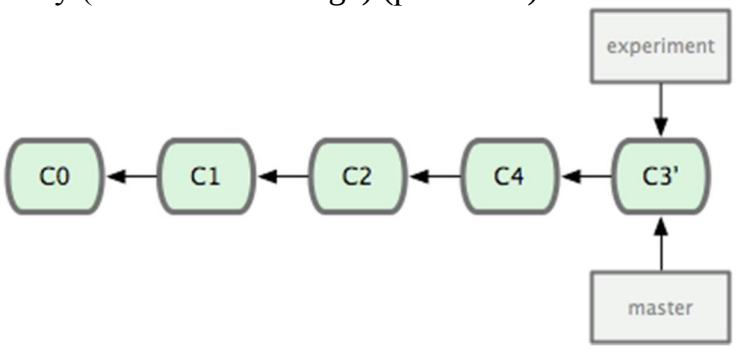

Рисунок 3-30. Перемотка ветки master

Теперь снимок состояния, на который указывает C3', точно такой же, как тот, на который указывал C5 в примере со слиянием. Нет никакой разницы в конечном результате объединения, но перемещение выполняется для того, чтобы история была более аккуратной. Если вы посмотрите лог для перемещённой ветки, то увидите, что он выглядит как линейная история работы: выходит, что вся работа выполнялась последовательно, когда в действительности она выполнялась параллельно.

Часто вы будете делать это, чтобы удостовериться, что ваши коммиты правильно применяются для удалённых веток — возможно для проекта, владельцем которого вы не являетесь, но в который вы хотите внести свой

вклад. В этом случае вы будете выполнять работу в какой-нибудь ветке, а затем, когда будете готовы внести свои изменения в основной проект, выполните перемещение вашей работы на origin/master. Таким образом, владельцу проекта не придётся делать никаких действий по объединению — просто перемотка (fast-forward) или чистое применение патчей.

Заметьте, что снимок состояния, на который указывает последний коммит, который у вас получился, является ли этот коммит последним перемещённым коммитом (для случая выполнения перемещения) или итоговым коммитом слияния (для случая выполнения слияния), есть один и тот же снимок — разной будет только история. Перемещение применяет изменения из одной линии разработки в другую в том порядке, в котором они были представлены, тогда как слияние объединяет вместе конечные точки двух веток.

Более интересные перемещения

Можно также сделать так, чтобы при перемещении воспроизведение коммитов начиналось не от той ветки, на которую делается перемещение. Возьмём, например, историю разработки как на рис. 3-31. Вы создали тематическую ветку (server), чтобы добавить в проект некоторый функционал для серверной части, и сделали коммит. Затем вы выполнили ответвление, чтобы сделать изменения для клиентской части, и несколько раз выполнили коммиты. Наконец, вы вернулись на ветку server и сделали ещё несколько коммитов.

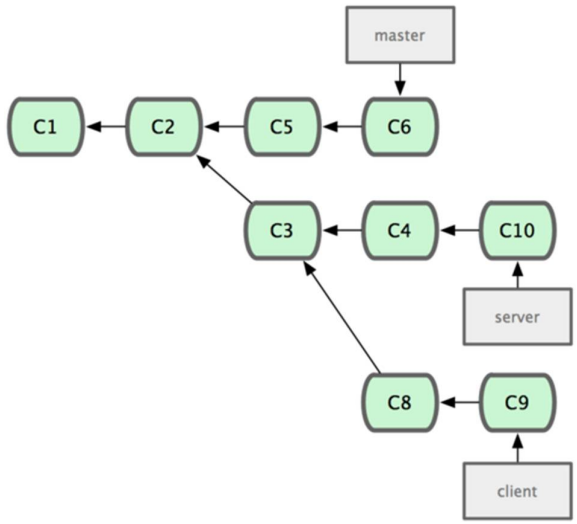

Рисунок 3-31. История разработки с тематической веткой, ответвлённой от другой тематической ветки

Предположим, вы решили, что хотите внести свои изменения для клиентской части в основную линию разработки для релиза, но при этом хотите оставить в стороне изменения для серверной части, пока они не будут полностью протестированы. Вы можете взять изменения из ветки client, которых нет в server (C8 и C9), и применить их на ветке master при помощи опции --onto команды git rebase:

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

По сути, это указание “переключиться на ветку client, взять изменения от общего предка веток client и server и повторить их на master”. Это немного сложно; но результат, показанный на рисунке 3-32, довольно классный.

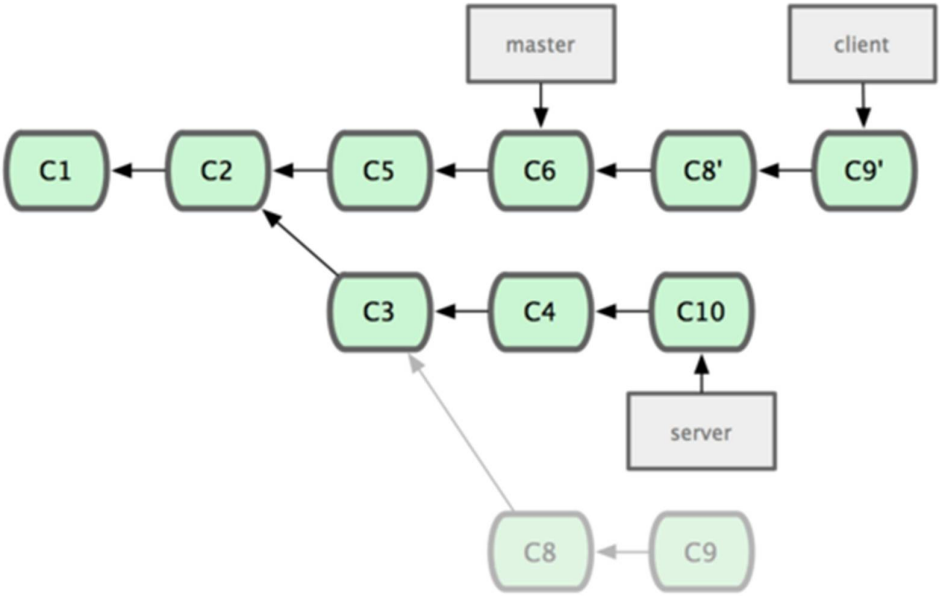

Рисунок 3-32. Перемещение тематической ветки, ответвлённой от другой тематической ветки

Теперь вы можете выполнить перемотку (fast-forward) для ветки master (рис. 3-33):

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

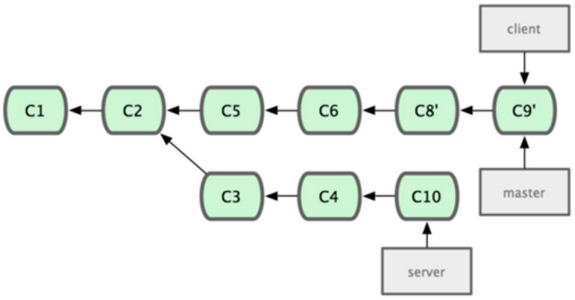

Рисунок 3-33. Перемотка ветки master для добавления изменений из ветки client

Представим, что вы решили включить работу и из ветки server тоже. Вы можете выполнить перемещение ветки server на ветку master без предварительного переключения на эту ветку при помощи команды git rebase [осн. ветка] [тем. ветка] — которая устанавливает тематическую ветку (в данном случае server) как текущую и применяет её изменения на основной ветке (master):

```
$ git rebase master server
```

Эта команда применит изменения из вашей работы над веткой server на вершину ветки master, как показано на рисунке 3-34.

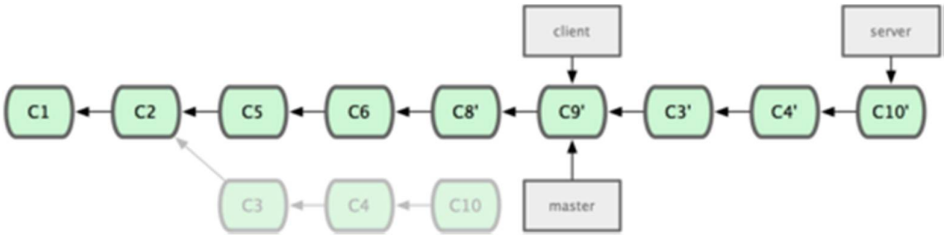

Рисунок 3-34. Перемещение ветки server на вершину ветки master

Затем вы можете выполнить перемотку основной ветки (master):

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

Вы можете удалить ветки client и server, так как вся работа из них включена в основную линию разработки и они вам больше не нужны. При этом полная история вашего рабочего процесса выглядит как на рисунке 3-35:

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

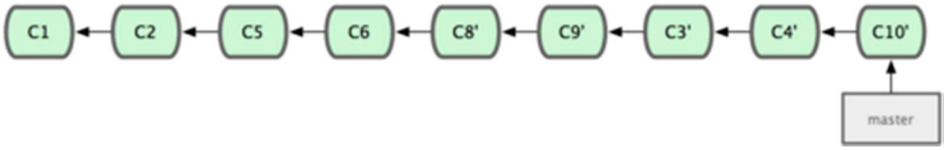

Рисунок 3-35. Финальная история коммитов

ВОЗМОЖНЫЕ РИСКИ ПЕРЕМЕЩЕНИЯ

Всё бы хорошо, но кое-что омрачает всю прелесть использования перемещения. Это выражается одной строчкой:

Не перемещайте коммиты, которые вы уже отправили в публичный репозиторий.

Если вы будете следовать этому указанию, всё будет хорошо. Если нет — люди возненавидят вас, вас будут презирать ваши друзья и семья.

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

Давайте рассмотрим пример того, как перемещение публично доступных наработок может вызвать проблемы. Представьте себе, что вы склонировали себе репозиторий с центрального сервера и поработали в нём. И ваша история коммитов выглядит как на рисунке 3-36.

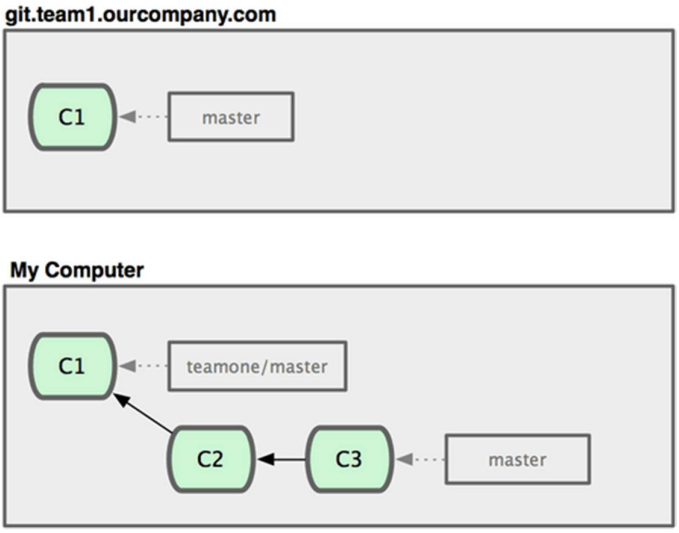

Рисунок 3-36. Клонирование репозитория и выполнение в нём какой-то работы

Теперь кто-то ещё выполняет работу, причём работа включает в себя и слияние, и отправляет свои изменения на центральный сервер. Вы извлекаете их и сливаете новую удалённую ветку со своей работой. Тогда ваша история выглядит как на рисунке 3-37.

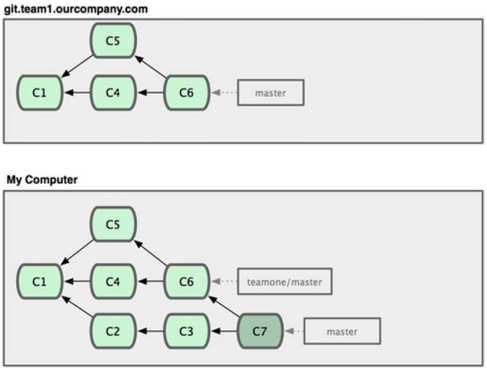

Рисунок 3-37. Извлечение коммитов и слияние их со своей работой

Далее, человек, выложивший коммит, содержащий слияние, решает вернуться и вместо слияния (merge) переместить (rebase) свою работу; он выполняет git push --force, чтобы переписать историю на сервере. Затем вы извлекаете изменения с этого сервера, включая и новые коммиты.

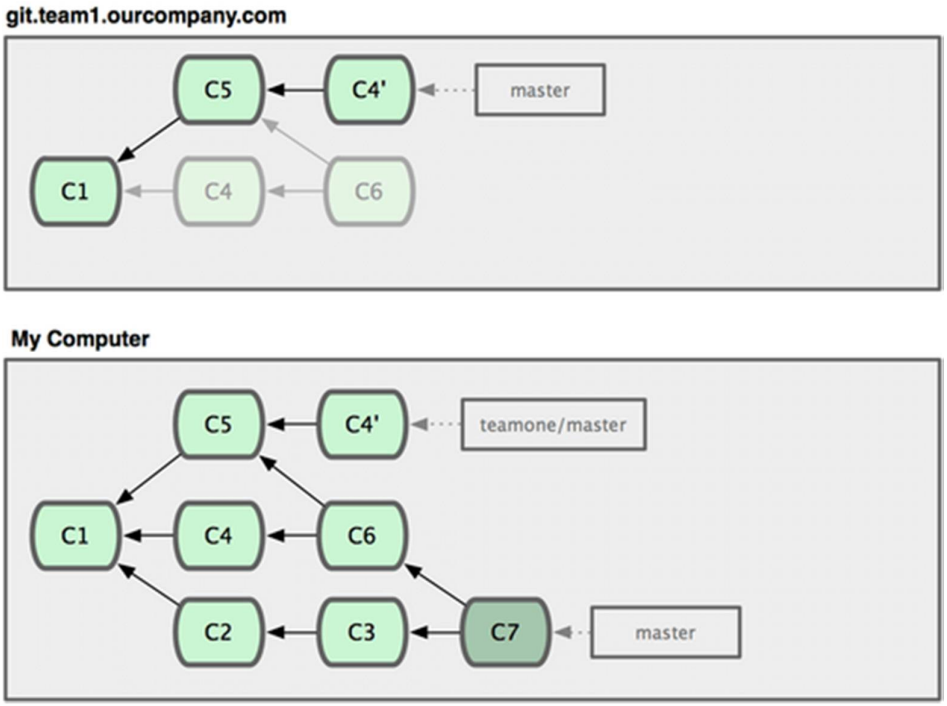

Рисунок 3-38. Кто-то выложил перемещённые коммиты, отменяя коммиты, на которых вы основывали свою работу

На этом этапе вы вынуждены объединить эту работу со своей снова, даже если вы уже сделали это ранее. Перемещение изменяет у этих коммитов SHA-1 хеши, так что для Git они выглядят как новые коммиты, тогда как на самом деле вы уже располагаете наработками из C4 в своей истории (рис. 3-39).

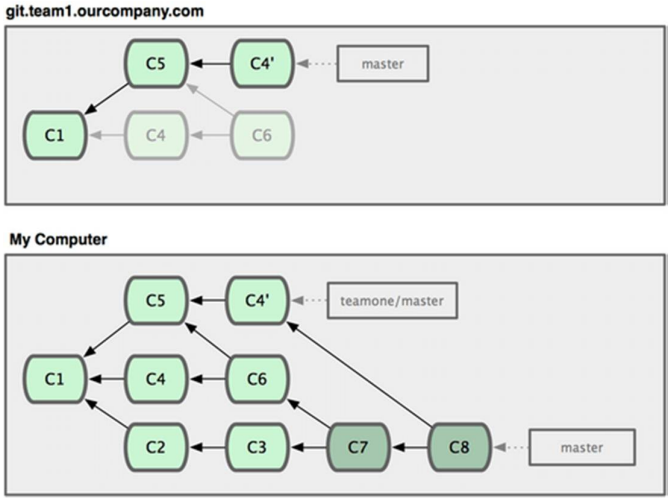

Рисунок 3-39. Вы снова выполняете слияние для той же самой работы в новый коммит слияния

Вы вынуждены объединить эту работу со своей на каком-либо этапе, чтобы иметь возможность продолжать работать с другими разработчиками в будущем. После того, как вы сделаете это, ваша история коммитов будет содержать оба коммита — C4 и C4', которые имеют разные SHA-1 хеши, но представляют собой одинаковые изменения и имеют одинаковые сообщения. Если вы выполните команду git log, когда ваша история выглядит таким

образом, вы увидите два коммита, которые имеют одинакового автора и одни и те же сообщения. Это сбивает с толку. Более того, если вы отправите такую историю обратно на сервер, вы добавите все эти перемещенные коммиты в репозиторий центрального сервера, что может ещё больше запутать людей.

Если вы рассматриваете перемещение как возможность наведения порядка и работы с коммитами до того, как выложили их, и если вы перемещаете только коммиты, которые никогда не находились в публичном доступе — всё нормально. Если вы перемещаете коммиты, которые уже были представлены для общего доступа, и люди, возможно, основывали свою работу на этих коммитах, тогда вы можете получить наказание за разные неприятные проблемы.