## Основы Git - Создание Git-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

Чтобы начать использовать Git для существующего проекта, 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'
```

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.

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'е 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.

## Основы Git - Запись изменений в репозиторий

### Запись изменений в репозиторий

Вам нужно делать некоторые изменения и фиксировать “снимки” состояния (snapshots) этих изменений в вашем репозитории каждый раз, когда проект достигает состояния, которое вам хотелось бы сохранить.

Запомните, каждый файл в вашем рабочем каталоге может находиться в одном из двух

состояний: под 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 вашем рабочем каталоге, которые не входили в ваш последний слепок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что вы только взяли их из хранилища (checked them out) и ничего пока не редактировали.

Как только вы отредактируете файлы, Git будет рассматривать их как изменённые, т.к. вы изменили их 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 2-1.

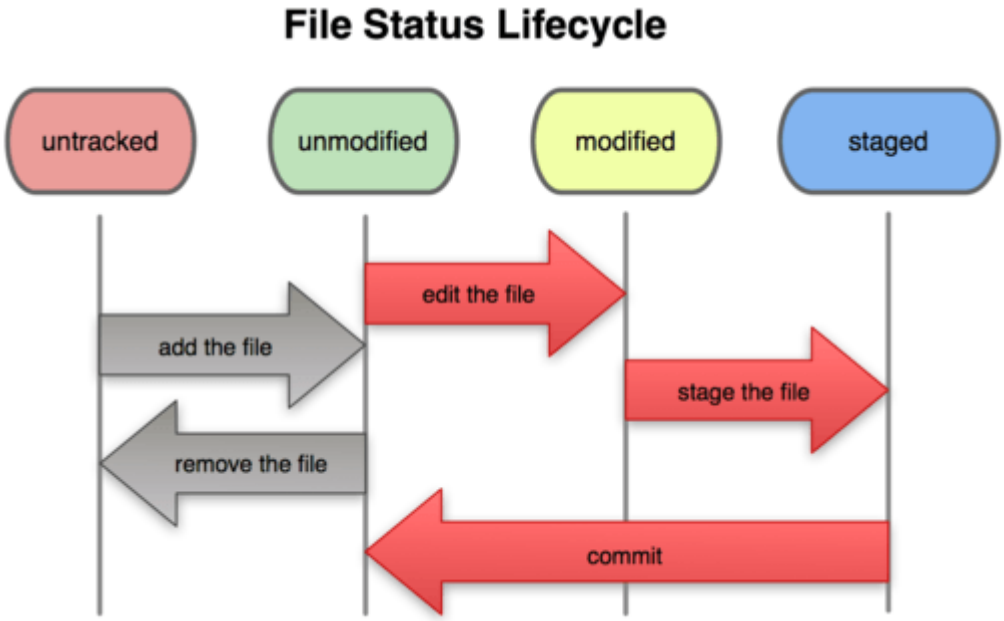


Рисунок 2-1. Жизненный цикл состояний файлов.

**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 здесь. И наконец, команда сообщает вам на какой ветке (branch) вы сейчас находитесь (ветка master — это ветка по умолчанию).

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

```
$ 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 вас 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. Как вы помните, когда вы ранее выполнили git init, вы затем выполнили git add (файлы) — это было сделано для того, чтобы добавить файлы в вашем каталоге под версионный контроль. Команда git add принимает параметром путь к файлу или каталогу, если это каталог, команда рекурсивно добавляет (индексирует) все файлы в данном каталоге.

**Индексация изменённых файлов**

Давайте модифицируем файл, уже находящийся под версионным контролем. Если вы измените отслеживаемый файл 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:   benchmarks.rb
#
```

Файл benchmarks.rb 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 benchmarks.rb, а затем снова выполним 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. В этот момент вы, предположим, вспомнили одно небольшое изменение, которое вы хотите сделать в benchmarks.rb до фиксации. Вы открываете файл, вносите и сохраняете необходимые изменения и вроде бы готовы к коммиту. Ещё раз выполним 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)
#
#   modified:   benchmarks.rb
#
```

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

**Игнорирование файлов**

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

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

*~

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

К шаблонам в файле .gitignore применяются следующие правила:

- Пустые строки, а также строки, начинающиеся с #, игнорируются.
- 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. Символ * соответствует 0 или более символам; последовательность [abc] — любому символу из указанных в скобках (в данном примере a, b или c); знак вопроса (?) соответствует одному символу; [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 для вас — вам хочется знать, что конкретно поменялось, а не только какие файлы были изменены — вы можете использовать команду git diff. Позже мы рассмотрим команду git diff подробнее; вы, скорее всего, будете использовать эту команду для получения ответов на два вопроса: что вы изменили, но ещё не проиндексировали, и что вы проиндексировали и собираетесь фиксировать. Если git status отвечает на эти вопросы слишком обобщённо, то git diff показывает вам непосредственно добавленные и удалённые строки — собственно заплатку (patch).

Допустим, вы снова изменили и проиндексировали файл README, а затем изменили файл benchmarks.rb 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:   benchmarks.rb
#
```

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

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 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 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'e 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 benchmarks.rb 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 benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
# On branch master
#
# Changes to be committed:
#
#   modified:   benchmarks.rb
#
# Changes not staged for commit:
#
#   modified:   benchmarks.rb
#
```

Теперь вы можете используя git diff посмотреть непроиндексированные изменения

```
$ 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 также уже проиндексированные, используя git diff --cached:

```
$ 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 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 этот коммит. Они останутся изменёнными файлами на вашем диске. В нашем случае, когда вы в последний раз выполняли git status, вы видели что всё проиндексировано, и вот, вы готовы к коммиту. Простейший способ зафиксировать изменения — это набрать git commit:

```
$ git commit
```

Эта команда откроет выбранный вами текстовый редактор. (Редактор устанавливается

системной переменной $EDITOR — обычно это vim или emacs, хотя вы можете установить ваш любимый с помощью команды git config --global core.editor, как было показано в главе 1).

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:   benchmarks.rb
~
~
~
".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 торчит в рабочем каталоге как изменённое; вы можете сделать ещё один коммит, чтобы добавить эти изменения в репозиторий. Каждый раз, когда вы делаете коммит, вы сохраняете снимок состояния вашего проекта, который позже вы можете восстановить или с которым можно сравнить текущее состояние.

**Игнорирование индексации**

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

```
$ git status
# On branch master
#
# Changes not staged for commit:
#
#   modified:   benchmarks.rb
```

```
#
$ 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 addдля файла benchmarks.rb.

**Удаление файлов**

Для того чтобы удалить файл из Git'a, вам необходимо удалить его из отслеживаемых файлов (точнее, удалить его из вашего индекса) а затем выполнить коммит. Это позволяет сделать команда git rm, которая также удаляет файл из вашего рабочего каталога, так что 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 вы выполните 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'a.

Другая полезная штука, которую вы можете захотеть сделать — это удалить файл из индекса, оставив его при этом в рабочем каталоге. Другими словами, вы можете захотеть оставить файл на винчестере, и убрать его из-под бдительного ока Git'a. Это особенно полезно, если вы забыли добавить что-то в файл .gitignore и по ошибке проиндексировали, например, большой файл с логами, или кучу промежуточных файлов компиляции. Чтобы сделать это, используйте опцию --cached:

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

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

```
$ 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 каталогelog/. 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'e, в нём не сохраняется никаких метаданных, говорящих о том, что файл был переименован. Однако, Git довольно умён в плане обнаружения перемещений постфактум — мы рассмотрим обнаружение перемещения файлов чуть позже.

Таким образом, наличие в Git'e команды mv выглядит несколько странным. Если вам хочется переименовать файл в Git'e, вы можете сделать что-то вроде:
$ git mv file_from file_to
и это отлично сработает. На самом деле, если вы выполните что-то вроде этого и посмотрите на 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 — вы можете использовать любой удобный способ, чтобы переименовать файл, и затем воспользоваться add/rm перед коммитом.

Просмотр истории коммитов

**Просмотр истории коммитов**

После того как вы создадите несколько коммитов, или же вы склонируете репозиторий с уже существующей историей коммитов, вы, вероятно, захотите оглянуться назад и узнать, что же происходило с этим репозиторием. Наиболее простой и в то же время мощный инструмент для этого — команда git log.

Данные примеры используют очень простой проект, названный simplegit, который я часто использую для демонстраций. Чтобы получить этот проект, выполните:
git clone git://github.com/schacon/simplegit-progit.git
В результате выполнения git log в данном проекте, вы должны получить что-то вроде этого:

```
$ 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 code

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

    first commit
```

По умолчанию, без аргументов, git log выводит список коммитов созданных в данном репозитории в обратном хронологическом порядке. То есть самые последние коммиты показываются первыми. Как вы можете видеть, эта команда отображает каждый коммит вместе с его контрольной суммой SHA-1, именем и электронной почтой автора, датой создания и комментарием.

Существует превеликое множество параметров команды git log и их комбинаций, для того чтобы показать вам именно то, что вы ищете. Здесь мы покажем вам несколько наиболее часто применяемых.

Один из наиболее полезных параметров — это -p, который показывает дельту (разницу/diff), привнесенную каждым коммитом. Вы также можете использовать -2, что ограничит вывод до 2-х последних записей:

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

    changed the version number

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

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

    removed unnecessary test code

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
```

Этот параметр показывает ту же самую информацию плюс внесённые изменения, отображаемые непосредственно после каждого коммита. Это очень удобно для инспекций кода или для того, чтобы быстро посмотреть, что происходило в результате последовательности коммитов, добавленных коллегой.

В некоторых ситуациях гораздо удобней просматривать внесённые изменения на уровне слов, а не на уровне строк. Чтобы получить дельту по словам вместо обычной дельты по строкам, нужно дописать после команды git log -p опцию --word-diff. Дельты на уровне слов практически бесполезны при работе над программным кодом, но они буду очень кстати при работе над длинным текстом, таким как книга или диссертация. Рассмотрим пример:

```
$ git log -U1 --word-diff
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|
    s.name     =   "simplegit"
    s.version  =   [-"0.1.0"-]{+"0.1.1"+}
    s.author   =   "Scott Chacon"
```

Как видите, в этом выводе нет ни добавленных ни удалённых строк, как для обычного diff'a. Вместо этого изменения показаны внутри строки. Добавленное слово заключено в {+ +}, а удалённое в [- -]. Также может быть полезно сократить обычные три строки контекста в выводе команды diff до одной строки, так как контекстом в данном случае являются слова, а не строки. Сделать это можно с помощью опции -U1 как было показано в примере выше.

С командой git log вы также можете использовать группы суммирующих параметров. Например, если вы хотите получить некоторую краткую статистику по каждому коммиту, вы можете использовать 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 version number

 Rakefile |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
```

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

    removed unnecessary test code

 lib/simplegit.rb |    5 -----
 1 files changed, 0 insertions(+), 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(+), 0 deletions(-)
```

Как видно из лога, параметр --stat выводит под каждым коммитом список изменённых файлов, количество изменённых файлов, а также количество добавленных и удалённых строк в этих файлах. Он также выводит сводную информацию в конце. Другой действительно полезный параметр — это --pretty. Он позволяет изменить формат вывода лога. Для вас доступны несколько предустановленных вариантов. Параметр oneline выводит каждый коммит в одну строку, что удобно если вы просматриваете большое количество коммитов. В дополнение к этому, параметры short, full, и fuller, практически не меняя формат вывода, позволяют выводить меньше или больше деталей соответственно:

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

Наиболее интересный параметр — это format, который позволяет вам полностью создать собственный формат вывода лога. Это особенно полезно, когда вы создаёте отчёты для автоматического разбора (парсинга) — поскольку вы явно задаёте формат и уверены в том, что он не будет изменяться при обновлениях Git'a:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit
```

Таблица 2-1 содержит список наиболее полезных параметров формата.

| **Параметр** | **Описание выводимых данных** |
|---|---|
| %H | Хеш коммита |
| %h | Сокращённый хеш коммита |
| %T | Хеш дерева |
| %t | Сокращённый хеш дерева |
| %P | Хеши родительских коммитов |
| %p | Сокращённые хеши родительских коммитов |
| %an | Имя автора |
| %ae | Электронная почта автора |
| %ad | Дата автора (формат соответствует параметру --date=) |
| %ar | Дата автора, относительная (пр. "2 мес. назад") |

| Параметр | Описание выводимых данных |
|---|---|
| %cn | Имя коммитера |
| %ce | Электронная почта коммитера |
| %cd | Дата коммитера |
| %cr | Дата коммитера, относительная |
| %s | Комментарий |

Вас может заинтересовать, в чём же разница между *автором* и *коммитером*. Автор — это человек, изначально сделавший работу, тогда как коммитер — это человек, который последним применил эту работу. Так что если вы послали патч (заплатку) в проект и один из основных разработчиков применил этот патч, вы оба не будете забыты — вы как автор, а разработчик как коммитер. Мы чуть подробнее рассмотрим это различие в главе 5.

Параметры oneline и format также полезны с другим параметром команды log — --graph. Этот параметр добавляет миленький ASCII-граф, показывающий историю ветвлений и слияний. Один из таких можно увидеть для нашей копии репозитория проекта Grit:

```
$ 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
```

Мы рассмотрели только самые простые параметры форматирования вывода для git log — их гораздо больше. Таблица 2-2 содержит как уже рассмотренные нами параметры, так и другие полезные параметры вместе с описанием того, как они влияют на вывод команды log.

| Параметр | Описание |
|---|---|
| -p | Для каждого коммита показывать дельту внесённых им изменений. |
| --word-diff | Показывать изменения на уровне слов. |
| --stat | Для каждого коммита дополнительно выводить статистику по изменённым файлам. |
| --shortstat | Показывать только строку changed/insertions/deletions от вывода с опцией --stat. |
| --name-only | Показывать список изменённых файлов после информации о коммите. |
| --name-status | Выводить список изменённых файлов вместе с информацией о добавлении/изменении/удалении. |
| --abbrev-commit | Выводить только первые несколько символов контрольной суммы SHA-1 вместо всех 40. |
| --relative-date | Выводить дату в относительном формате (например, "2 weeks ago") вместо полной даты. |
| --graph | Показывать ASCII-граф истории ветвлений и слияний рядом с выводом лога. |
| --pretty | Отображать коммиты в альтернативном формате. Возможные параметры: oneline, short, full, fuller и format (где вы можете указать свой собственный формат). |

**Ограничение вывода команды log**

Кроме опций для форматирования вывода, git log имеет ряд полезных ограничительных параметров, то есть параметров, которые дают возможность отобразить часть коммитов. Вы

уже видели один из таких параметров — параметр -2, который отображает только два последних коммита. На самом деле, вы можете задать -<n>, где n это количество отображаемых коммитов. На практике вам вряд ли придётся часто этим пользоваться потому, что по умолчанию Git через канал (pipe) отправляет весь вывод на pager, так что вы всегда будете видеть только одну страницу.

А вот параметры, ограничивающие по времени, такие как --since и --until, весьма полезны. Например, следующая команда выдаёт список коммитов, сделанных за последние две недели:

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

Такая команда может работать с множеством форматов — вы можете указать точную дату (“2008-01-15”) или относительную дату, такую как “2 years 1 day 3 minutes ago”.

Вы также можете отфильтровать список коммитов по какому-либо критерию поиска. Опция --author позволяет фильтровать по автору, опция --grep позволяет искать по ключевым словам в сообщении. (Заметим, что, если вы укажете и опцию author, и опцию grep, то будут найдены все коммиты, которые удовлетворяют первому ИЛИ второму критерию. Чтобы найти коммиты, которые удовлетворяют первому И второму критерию, следует добавить опцию --all-match.)

Последняя действительно полезная опция-фильтр для git log — это путь. Указав имя каталога или файла, вы ограничите вывод log теми коммитами, которые вносят изменения в указанные файлы. Эта опция всегда указывается последней и обычно предваряется двумя минусами (--), чтобы отделить пути от остальных опций.

В таблице 2-3 для справки приведён список часто употребляемых опций.

| **Опция** | **Описание** |
|---|---|
| -(n) | Показать последние n коммитов |
| --since, --after | Ограничить коммиты теми, которые сделаны после указанной даты. |
| --until, --before | Ограничить коммиты теми, которые сделаны до указанной даты. |
| --author | Показать только те коммиты, автор которых соответствует указанной строке. |
| --committer | Показать только те коммиты, коммитер которых соответствует указанной строке. |

Например, если вы хотите посмотреть из истории Git'a такие коммиты, которые вносят изменения в тестовые файлы, были сделаны Junio Hamano, не являются слияниями и были сделаны в октябре 2008го, вы можете выполнить что-то вроде такого:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attribute
acd3b9e - Enhance hold_lock_file_for_{update,append}()
f563754 - demonstrate breakage of detached checkout wi
d1a43f2 - reset --hard/read-tree --reset -u: remove un
51a94af - Fix "checkout --track -b newbranch" on detac
b0ad11e - pull: allow "git pull origin $something:$cur
```

Из примерно 20 000 коммитов в истории Git'a, данная команда выбрала всего 6 коммитов, соответствующих заданным критериям.

**Использование графического интерфейса для визуализации истории**

Если у вас есть желание использовать какой-нибудь графический инструмент для визуализации истории коммитов, можно попробовать распространяемую вместе с Git'ом программу gitk, написанную на Tcl/Tk. В сущности gitk — это наглядный вариант git log, к тому же он принимает почти те же фильтрующие опции, что и git log. Если наберёте в командной строке gitk, находясь в проекте, то увидите что-то наподобие рис. 2-2.

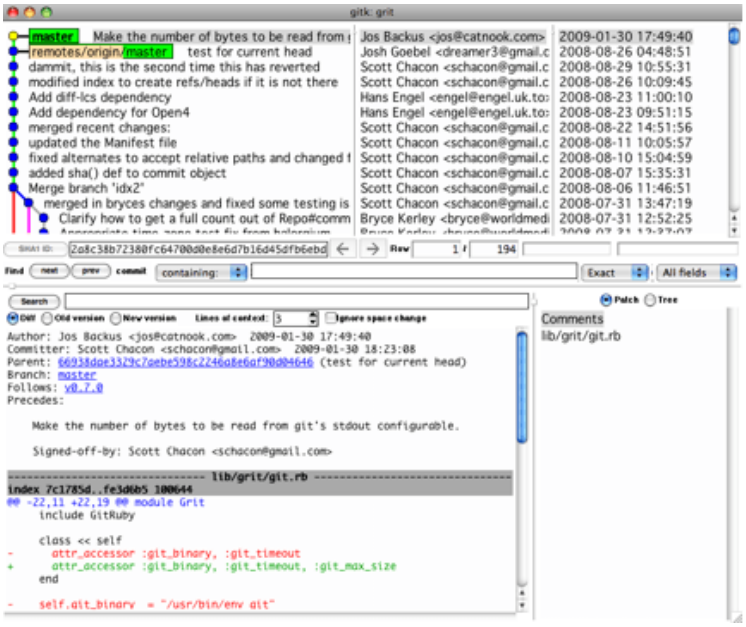


Рисунок 2-2. Визуализация истории с помощью gitk.

В верхней части окна располагается история коммитов вместе с подробным графом наследников. Просмотрщик дельт в нижней половине окна отображает изменения, сделанные выбранным коммитом. Указать коммит можно с помощью щелчка мышью.

**Отмена изменений**

На любой стадии может возникнуть необходимость что-либо отменить. Здесь мы рассмотрим несколько основных инструментов для отмены произведённых изменений. Будьте осторожны, ибо не всегда можно отменить сами отмены. Это одно из немногих мест в Git'e, 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 напутали с комментарием к коммиту. Если вам хотелось бы сделать этот коммит ещё раз, вы можете 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. Вы можете отредактировать это сообщение так же, как обычно, и оно перепишет предыдущее.

Для примера, если после совершения коммита вы осознали, что забыли проиндексировать изменения в файле, которые хотели добавить в этот коммит, вы можете сделать что-то подобное:

$ 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**

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. Так что давайте последуем совету и 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
#
```

Файл benchmarks.rb изменён, но снова не в индексе.

**Отмена изменений файла**

Как быстро отменить изменения, вернуть то состояние, в котором файл находился во время последнего коммита (или первоначального клонирования, или какого-то другого действия, после которого файл попал в рабочий каталог)? git status говорит, как добиться и этого. 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 ясно сказано, как отменить сделанные изменения:

```
$ 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 вами 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 мешался, мы рассмотрим прятание (stash) и ветвление в следующей главе; эти способы обычно более

предпочтительны.

Помните, что всё, что является частью коммита в Git'e, почти всегда может быть восстановлено. Даже коммиты, которые находятся на ветках, которые были удалены, и коммиты переписанные с помощью --amend могут быть восстановлены (см. главу 9 для восстановления данных). Несмотря на это, всё, что никогда не попадало в коммит, вы скорее всего уже не увидите снова.

**Основы ветвления**

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

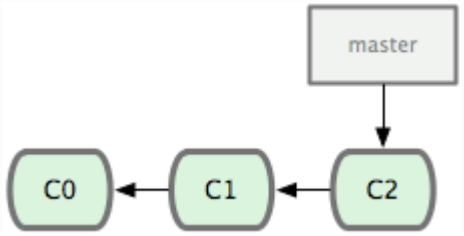


Рисунок 3-10. Короткая и простая история коммитов.

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.

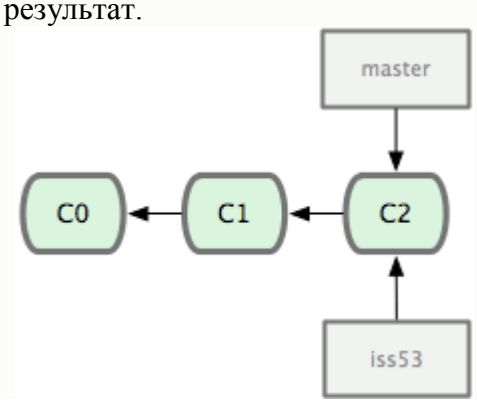


Рисунок 3-11. Создание новой ветки / указателя.

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; см. рис. 3-12):

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

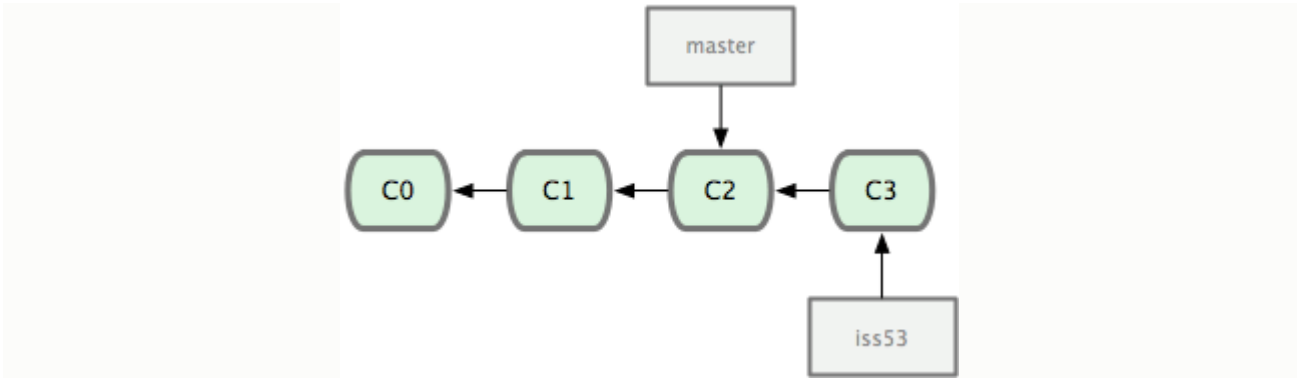


Рисунок 3-12. Ветка iss53 передвинулась вперёд во время работы.

Теперь вы получаете звонок о том, что есть проблема с веб-сайтом, которую необходимо немедленно устранить. С Git'ом 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), которые мы рассмотрим позже. 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 (см. 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(-)
```

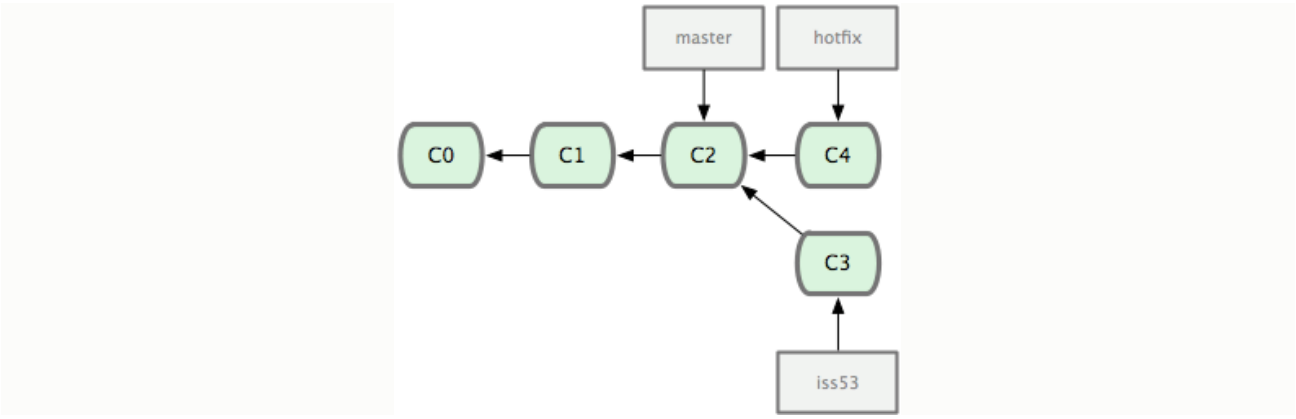


Рисунок 3-13. Ветка для решения срочной проблемы базируется на ветке master.

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 (см. рис. 3-14).

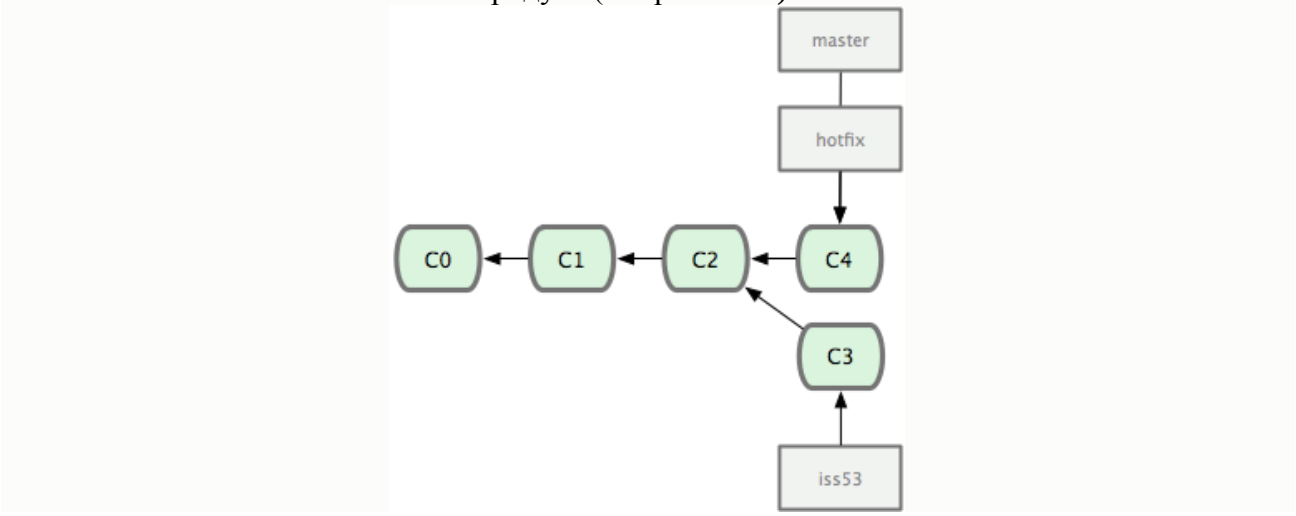


Рисунок 3-14. После слияния ветка master указывает туда же, куда и ветка hotfix.

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 (см. 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(-)
```

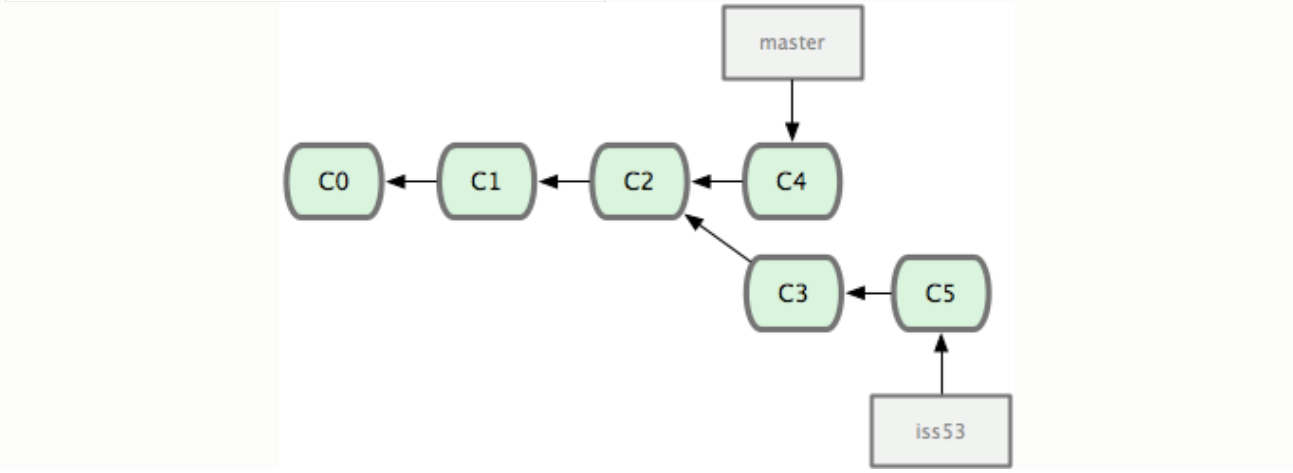


Рисунок 3-15. Ветка iss53 может двигаться вперёд независимо.

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'y 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.

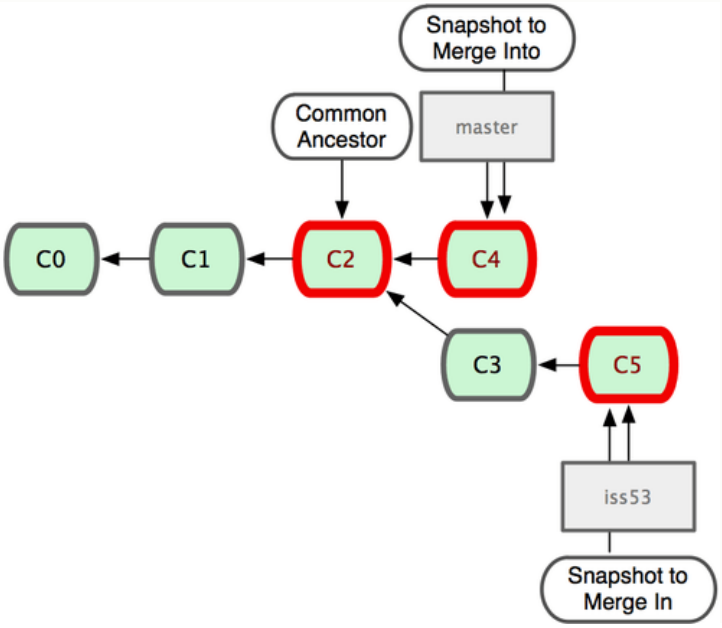


Рисунок 3-16. Git автоматически определяет наилучшего общего предка для слияния веток.

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'e гораздо более простым занятием, чем в других системах.

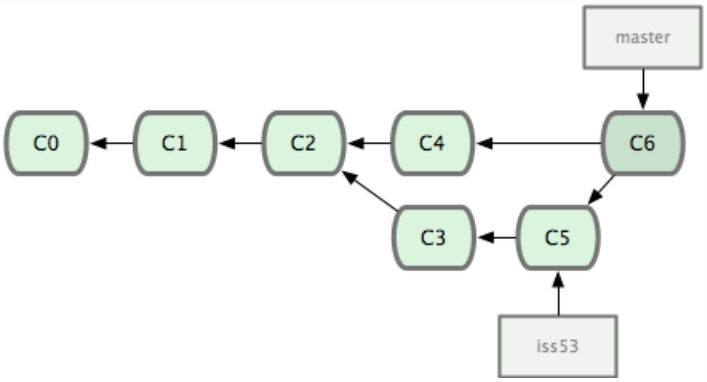


Рисунок 3-17. Git автоматически создаёт новый коммит, содержащий результаты слияния.

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
```

**Основы конфликтов при слиянии**

Иногда процесс слияния не идёт гладко. Если вы изменили одну и ту же часть файла по-разному в двух ветках, которые собираетесь слить, Git не сможет сделать это чисто. Если ваше решение проблемы №53 изменяет ту же часть файла, что и `hotfix`, вы получите конфликт слияния, и выглядеть он будет примерно так:

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

Git не создал новый коммит для слияния. Он приостановил этот процесс до тех пор, пока вы не разрешите конфликт. Если вы хотите посмотреть, какие файлы не прошли слияние (на любом этапе после возникновения конфликта), выполните команду `git status`:

```
[master*]$ git status
index.html: needs merge
# 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)
#
#   unmerged:   index.html
#
```

Всё, что имеет отношение к конфликту слияния и что не было разрешено, отмечено как unmerged. Git добавляет стандартные маркеры к файлам, которые имеют конфликт, так что вы можете открыть их вручную и разрешить эти конфликты. Ваш файл содержит секцию, которая выглядит примерно так:

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

В верхней части блока (всё что выше `=======`) это версия из HEAD (вашей ветки master, так как именно на неё вы перешли перед выполнением команды merge), всё, что находится в нижней части — версия в `iss53`. Чтобы разрешить конфликт, вы должны либо выбрать одну из этих частей, либо как-то объединить содержимое по своему усмотрению. Например, вы можете разрешить этот конфликт заменой всего блока, показанного выше, следующим блоком:

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

Это решение содержит понемногу из каждой части, и я полностью удалил строки `<<<<<<<`, `=======` и `>>>>>>>`. После того как вы разобрались с каждой из таких секций в каждом из конфликтных файлов, выполните `git add` для каждого конфликтного файла. Индексирование будет означать для Git'a, что все конфликты в файле теперь разрешены. Если вы хотите использовать графические инструменты для разрешения конфликтов, можете выполнить команду `git mergetool`, которая запустит соответствующий графический инструмент и покажет конфликтные ситуации:

```
$ git mergetool
```

```
merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff
Merging the files: index.html

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

Если вы хотите использовать другой инструмент для слияния, нежели выбираемый по умолчанию. Вы можете увидеть все поддерживаемые инструменты, указанные выше после “merge tool candidates”. Укажите название предпочтительного для вас инструмента. После того как вы выйдете из инструмента для выполнения слияния, Git спросит вас, было ли оно успешным. Если вы отвечаете, что да — файл индексируется (добавляется в область для коммита), чтобы дать вам понять, что конфликт разрешён.

Можете выполнить `git status` ещё раз, чтобы убедиться, что все конфликты были разрешены:

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

Если вы довольны тем, что получили, и удостоверились, что всё, имевшее конфликты, было проиндексировано, можете выполнить `git commit` для завершения слияния. По умолчанию сообщение коммита будет выглядеть примерно так:

```
Merge branch 'iss53'

Conflicts:
  index.html
#
# It looks like you may be committing a MERGE.
# If this is not correct, please remove the file
# .git/MERGE_HEAD
# and try again.
#
```

Вы можете дополнить это сообщение информацией о том, как вы разрешили конфликт, если считаете, что это может быть полезно для других в будущем. Например, можете указать почему вы сделали то, что сделали, если это не очевидно, конечно.

**Управление ветками**

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

Команда `git branch` делает несколько больше, чем просто создаёт и удаляет ветки. 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'e 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.