Лекция 2. Основы Git

Лекция 2. Основы Git

Основы Git

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

Создание репозитория Git

Для создания репозитория Git 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 создает в текущем каталоге новый подкаталог с именем .git содержащий все необходимые файлы репозитория — основу репозитория Git. На этом этапе ваш проект еще не находится под версионным контролем. (В Главе 9 приведено подробное описание файлов содержащихся в только что созданном вами каталоге «.git»)

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 репозиторий Git 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 правил (server-side hooks) и т.п., но все данные, помещённые под версионный контроль, будут сохранены, подробнее см. в Главе 4).

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 создает каталог с именем «grit», инициализирует в нем каталог .git, скачивает все данные для этого репозитория и создает (checks out) рабочую копию последней версии. Если вы зайдете в новый каталог grit, вы увидите в нем 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 все то же самое, что и предыдущая, только результирующий каталог будет назван mygrit.

Git реализует 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. В Главе 4 представлены все доступные варианты конфигурации сервера для доступа к вашему репозиторию Git, а также их достоинства и недостатки.

Запись изменений в репозиторий

Итак, у вас имеется настоящий репозиторий Git и рабочая копия файлов для некоторого проекта. Вам нужно делать некоторые изменения и фиксировать «снимки» состояния (snap- shots) 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 неизмененными, измененными 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 неизмененными, 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 измененные, 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 изображен на Рисунке 2-1.

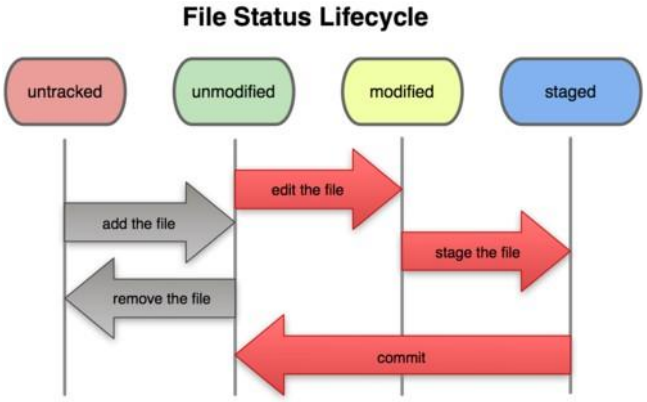

Рисунок 2.1: Жизненный цикл состояния ваших файлов.

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 нем нет отслеживаемых измененных 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. Пока что это всегда ветка master — это ветка по умолчанию; в этой главе это не важно. В следующей главе будет подробно рассказано про ветки и ссылки.

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

Вы можете видеть, что новый файл README неотслеживаемый, т.к. он находится в секции «Untracked files» в выводе команды status. Неотслеживаемый файл обычно означает, что Git нашел файл, отсутствующий в предыдущем снимке состояния (коммите); Git не станет добавлять его в ваши коммиты, пока вы явно ему это не укажете. Это предохраняет вас от случайного добавления в репозиторий сгенерированных двоичных файлов или каких-либо других, которые вы и не думали добавлять. Вы хотите добавить README, так что давайте сделаем это.

Отслеживание новых файлов

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

```
$ git add README
```

Если вы снова выполните команду status, то увидите, что файл README теперь отслеживаемый и 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 (files) — 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 измененных файлов

Давайте модифицируем файл, уже находящийся под версионным контролем. Если вы измените отслеживаемый файл benchmarks.rb 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
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
```

Файл benchmarks.rb находится в секции «Changed but not updated» — это означает, что отслеживаемый файл был изменен 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)
```

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. Но давайте-ка еще 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
#
# Changed but not updated:
```

Что за черт? 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 попадет 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 придется 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)
```

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 серьезно 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 упрощенные 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 еще один пример файла .gitignore:

комментарий — эта строка игнорируется

*.a # не обрабатывать файлы, имя которых заканчивается на .a

!lib.a # НО отслеживать файл lib.a, несмотря на то, что мы игнорируем все .a файлы с помощью предыдущего пра

/TODO # игнорировать только файл TODO находящийся в корневом каталоге, не относится к файлам вида subdir/ TODO

build/ # игнорировать все файлы в каталоге build/

doc/*.txt # игнорировать doc/notes.txt, но не doc/server/arch.txt

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 еще не проиндексировали, и что вы проиндексировали и собираетесь фиксировать. Если git status отвечает на эти вопросы слишком обобщенно, то git diff показывает вам непосредственно добавленные и удаленные 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
#
# Changed but not updated:
#   (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

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

Эта команда сравнивает содержимое вашего рабочего каталога с содержимым индекса.

Результат показывает еще не проиндексированные изменения.

Если вы хотите посмотреть, что вы проиндексировали и что войдет 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 еще не проиндексированы. Такое поведение может сбивать с толку, так как если вы проиндексируете все свои изменения, то git diff ничего не вернет.

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
```

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 ваши изменения. Запомните, всё, что до сих пор не проиндексировано — любые файлы, созданные или измененные вами, и для которых вы не выполнили git add после момента редактирования — не войдут в этот коммит. Они останутся измененными файлами на вашем диске. В нашем случае, когда вы в последний раз выполняли git status, вы видели что все проиндексировано, и вот, вы готовы к коммиту. Простейший способ зафиксировать ваши 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 отображен 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
#   (use "git reset HEAD <file>..." to unstage)
#
        new file:       README
#
#
".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 ваше сообщение или же оставить их для напоминания того, что вы фиксируете. (Для еще более подробного напоминания, что же вы именно меняли, вы можете передать аргумент -v в команду git commit. Это приведет к тому, что в комментарий будет помещена также разница/diff ваших изменений, таким образом вы сможете точно увидеть всё что сделано.) Когда вы выходите из редактора, Git создает ваш коммит с этим сообщением (удаляя комментарии и вывод diff-a).

Другой способ — вы можете набрать ваш 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/удаленным строкам в этом коммите.

Запомните, что коммит сохраняет снимок состояния вашего индекса. Все, что вы не проиндексировали, так и торчит в рабочем каталоге как измененное; вы можете сделать еще

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

                                  # Changed but not updated:

                                  # 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 add для файла benchmarks.rb.

Удаление файлов

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

Если вы просто удалите файл из вашего рабочего каталога, он будет показан в секции

«Changed but not updated» («Измененные но не обновленные» — читай не проиндексированные) вывода команды git status:

```
     $ rm grit.gemspec
       $ git status
    # On branch master
#    (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
#    (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.

Другая полезная штука, которую вы можете захотеть сделать — это удалить файл из индекса, оставив его при этом в вашем рабочем каталоге. Другими словами, вы можете захотеть оставить файл на вашем винчестере, и убрать его из-под бдительного ока Git-а. Это особенно полезно, если вы забыли добавить что-то в ваш 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 вы можете 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 *. Это обязательно, так как 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 непосредственно перемещение файла. Если вы переименуете файл в Git, то в Git не сохранится никаких метаданных

о том, что вы переименовали файл. Однако, Git довольно умён в плане обнаружения перемещений постфактум — мы рассмотрим обнаружение перемещения файлов чуть позже.

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

```
$ git mv file_from file_to
```

и это отлично сработает. На самом деле, если вы выполните что-то вроде этого и посмотрите на 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.
#   (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 было 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

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:

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. 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(-)

  README           |              6 ++++++
                          23 ++++++++++++++++++++++
  lib/simplegit.rb |
  3 files changed, 54 insertions(+), 0 deletions(-)
```

Как видно из лога, параметр --stat выводит под каждым коммитом список измененных файлов, количество измененных файлов, а также количество добавленных и удаленных 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 создаете отчеты для автоматического разбора(парсинга) — поскольку вы явно задаете формат и уверены в том, что он не будет изменяться при обновлениях Git:

```
$ 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`Сокращенный хеш коммита

`%T`Хеш дерева

`%t`Сокращенный хеш дерева

`%P`Хеши родительских коммитов

`%p`Сокращенные 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")

`%cn`Имя коммитера

`%ce`Электронная почта коммитера
```

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`Выводит патч (заплатку/diff) внесенный каждым коммитом.

`--stat`Выводит статистику по файлам измененным в каждом коммите.

`--shortstat`Отображает толькострокус changed/insertions/ deletions от вывода команды `--stat`.

`--name-only`Выводит список измененных файлов после каждого коммита.

`--name-status`Выводит список файлов вместе с информацией о добавлении/изменении/ удалении.

`--abbrev-commit`Выводит только первые несколько символов контрольной суммы SHA-1 вместо всех 40.

`--relative-date`Выводит дату в относительном формате (например, “2 недели назад”) вместо использования пол

`--graph`Выводит ASCII граф истории ветвлений и слияний рядом с выводом лога.

`--pretty`Выводит коммиты в альтернативном формате. Параметры включают oneline, short, full, fuller, и forma

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 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, 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 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 набрать в командной строке gitk, находясь в проекте, вы увидете что-то наподобие Рис. 2-2.

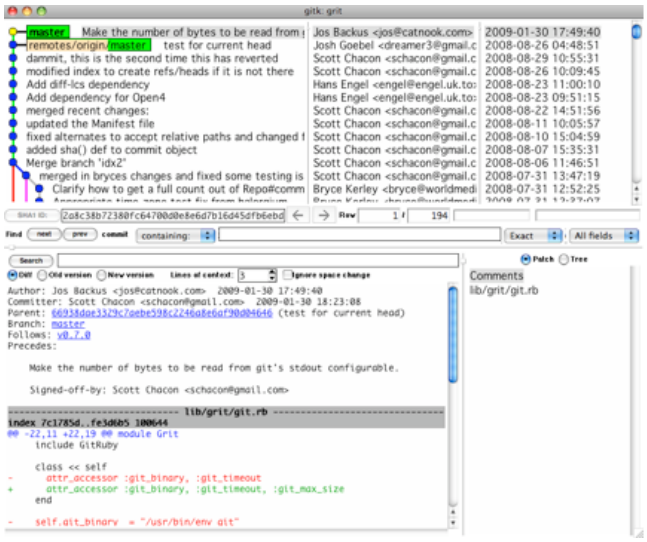

Рисунок 2.2: Визуализация истории с помощью gitk.

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, 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

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
#    (use "git reset HEAD <file>..." to unstage)
#        modified:     README.txt
#
```

Сразу после надписи «Changes to be committed», написано использовать git reset HEAD <file>... 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
#    (use "git reset HEAD <file>..." to unstage)
#        modified:    README.txt
# Changed but not updated:
#           (use "git add <file>..." to update what will be committed)
#
        modified:    benchmarks.rb
#
```

Эта команда немного странновата, но она работает. Файл benchmarks.rb изменён, но снова не в индексе.

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)? К счастью, git status говорит, как добиться и этого. В выводе для последнего примера, неиндексированная область выглядит следующим образом:

```
# Changed but not updated:
#           (use "git add <file>..." to update what will be committed)
#
#        modified:    benchmarks.rb
```

Здесь довольно ясно сказано, как отменить сделанные изменения (по крайней мере новые версии Git'a, начиная с 1.6.1, делают это; если у вас версия старее, мы настоятельно рекомендуем обновиться, чтобы получать такие подсказки и сделать свою работу удобней). Давайте сделаем то, что написано:

```
$ git checkout -- benchmarks.rb

          $ git status

       # On branch master
#    (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, 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 увидете снова.

Работа с удалёнными репозиторями

Чтобы иметь возможность совместной работы над каким-либо Git-проектом, необходимо знать, как управлять удалёнными репозиториями. Удалённые репозитории — это модификации проекта, которые хранятся в интернете или ещё где-то в сети. Их может быть несколько, каждый из которых, как правило, доступен для вас либо только на чтение, либо на чтение и запись. Совместная работа включает в себя управление удалёнными репозиториями и помещение (push) и получение (pull) данных в и из них тогда, когда нужно обменяться результатами работы. Управление удалёнными репозиториями включает умение добавлять удалённые репозитории, удалять те из них, которые больше не действуют, умение управлять различными удалёнными ветками и определять их как отслеживаемые (tracked)

или нет и прочее. Данный раздел охватывает все перечисленные навыки по управлению удалёнными репозиториями.

Отображение удалённых репозиториев

Чтобы просмотреть, какие удалённые серверы у вас уже настроены, следует выполнить команду git remote. Она перечисляет список имён-сокращений для всех уже указанных удалённых дескрипторов. Если вы склонировали ваш репозиторий, у вас должен отобразиться, по крайней мере, origin — это имя по умолчанию, которое Git присваивает серверу, с которого вы склонировали:

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

```
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.

remote: Compressing objects: 100% (269/269), done.
remote: Total 595 (delta 255), reused 589 (delta 253)

Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.

$ cd ticgit

$ git remote
origin
```

Чтобы посмотреть, какому URL соответствует сокращённое имя в Git, можно указать команде опцию -v:

```
$ git remote
origin

$ git remote add pb git://github.com/paulboone/ticgit.git

$ git remote -v

origin git://github.com/schacon/ticgit.git
pb git://github.com/paulboone/ticgit.git
```

```
$ git remote -v

origin git://github.com/schacon/ticgit.git
```

Если у вас больше одного удалённого репозитория, команда покажет их все. Например, мой репозиторий Grit выглядит следующим образом.

```
$ cd grit

$ git remote -v

bakkdoor  git://github.com/bakkdoor/grit.git
  cho45     git://github.com/cho45/grit.git
defunkt    git://github.com/defunkt/grit.git
  koke      git://github.com/koke/grit.git
  origin    git@github.com:mojombo/grit.git
```

Это означает, что мы легко можем получить изменения от любого из этих пользователей. Но, заметьте, что origin — это единственный удалённый сервер прописанный как SSH ссылка, поэтому он единственный, в который я могу помещать свои изменения (это будет рассмотрено в Главе 4).

Добавление удалённых репозиториев

В предыдущих разделах мы упомянули и немного продемонстрировали добавление удалённых репозиториев, сейчас мы рассмотрим это более детально. Чтобы добавить новый удалённый Git-репозиторий под именем-сокращением, к которому будет проще обращаться, выполните git remote add [сокращение] [url]:

Теперь вы можете использовать в командной строке имя pb вместо полного URL. Например, если вы хотите извлечь (fetch) всю информацию, которая есть в репозитории Павла, но нет в вашем, вы можете выполнить git fetch pb:

```
$ git fetch pb

remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
    Unpacking objects: 100% (44/44), done.

From git://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

Ветка master Павла теперь доступна локально как pb/master. Вы можете слить (merge) её в одну из своих веток или перейти на эту ветку, если хотите её проверить.

Fetch и Pull

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

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

Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта. Теперь эти ветки в любой момент могут быть просмотрены или слиты. (В 3 Главе мы перейдём к более детальному рассмотрению, что такое ветки и как их использовать.)

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

Если у вас есть ветка, настроенная на отслеживание удалённой ветки (для дополнительной информации смотри следующий раздел и Главу 3), то вы можете использовать команду git pull. Она автоматически извлекает и затем сливает данные из удалённой ветки в вашу текущую ветку. Этот способ может для вас оказаться более простым или более удобным. К тому же по умолчанию команда git clone автоматически настраивает вашу локальную ветку master на отслеживание удалённой ветки master на сервере, с которого вы клонировали (подразумевается, что на удалённом сервере есть ветка master). Выполнение git pull, как правило, извлекает (fetch) данные с сервера, с которого вы изначально склонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете.

Push

Когда вы хотите поделиться своими наработками, вам необходимо отправить (push) их в главный репозиторий. Команда для этого действия простая: git push [удал. сервер] [ветка]. Чтобы отправить вашу ветку master на сервер origin (повторимся, что клонирование, как правило, настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки наработок на сервер:

```
$ git push origin master
```

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

(pull) их изменения и объединить с вашими. Только после этого вам будет позволено выполнить push. Смотри Главу 3 для более подробного описания, как отправлять (push) данные на удалённый сервер.

Инспекция удалённого репозитория

Если хотите получить побольше информации об одном из удалённых репозиториев, вы можете использовать команду git remote show [удал. сервер]. Если вы выполните эту команду с некоторым именем, например, origin, вы получите что-то подобное:

```
$ git remote show origin
* remote origin
URL: git://github.com/schacon/ticgit.git
Remote branch merged with 'git pull' while on branch master
master
Tracked remote branches
master
ticgit
```

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

Это был пример для простой ситуации, и наверняка вы встретились с чем-то подобным.

Однако, если вы используете Git более интенсивно, вы можете увидеть гораздо большее количество информации от git remote show:

```
$ git remote show origin
* remote origin
URL: git@github.com:defunkt/github.git
Remote branch merged with 'git pull' while on branch issues
```

```
                                        issues

  Remote branch merged with 'git pull' while on branch master
                               master

  New remote branches (next fetch will store in remotes/origin)
                               caching

  Stale tracking branches (use 'git remote prune')
                       libwalker

                                        walker2

  Tracked remote branches
             acl

       apiv2
    dashboard2
      issues
      master
     postgres
```

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

Удаление и переименование удалённых репозиториев

Для переименования ссылок в новых версиях Git можно выполнить git remote rename, это изменит сокращённое имя, используемое для удалённого репозитория. Например, если вы хотите переименовать pb в paul, вы можете сделать это следующим образом:

```
                              $ git remote rename pb paul

$ git remote
   origin

                                        paul
```

Стоит упомянуть, что это также меняет для вас имена удалённых веток. То, к чему вы обращались как pb/master, стало paul/master.

Если по какой-то причине вы хотите удалить ссылку (вы сменили сервер или больше не используете определённое зеркало, или, возможно, контрибьютор перестал быть активным), вы можете использовать git remote rm:

```
                                 $ git remote rm paul

$ git remote
   origin
```

Работа с метками

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

Просмотр меток

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

```
$ git tag
   v0.1

                                          v1.3
```

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

Для меток вы также можете осуществлять поиск по шаблону. Например, репозиторий Git содержит более 240 меток. Если вас интересует просмотр только выпусков 1.4.2, вы можете выполнить следующее:

```
$ git tag -l 'v1.4.2.*'
        v1.4.2.1

                                       v1.4.2.2

                                       v1.4.2.3

                                       v1.4.2.4
```

Создание меток

Git использует два основных типа меток: легковесные и аннотированные. Легковесная метка — это что-то весьма похожее на ветку, которая не меняется — это просто указатель на определённый коммит. А вот аннотированные метки хранятся в базе данных Git’а как полноценные объекты. Они имеют контрольную сумму, содержат имя поставившего метку, e-mail и дату, имеют комментарий и могут быть подписаны и проверены с помощью GNU Privacy Guard (GPG). Обычно рекомендуется создавать аннотированные метки, чтобы иметь всю перечисленную информацию; но если вы хотите сделать временную метку или по какой-то причине не хотите сохранять остальную информацию, то для этого годятся и легковесные метки.

Аннотированные метки

Создание аннотированной метки в Git выполняется легко. Самый простой способ это указать -a при выполнении команды tag:

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

$ git tag
  v0.1

v1.3
v1.4
```

Опция -m задаёт меточное сообщение, которое будет храниться вместе с меткой. Если не указать сообщение для аннотированной метки, Git запустит редактор, чтоб вы смогли его ввести. Вы можете посмотреть данные метки вместе с коммитом, который был помечен, с помощью команды git show:

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

                    Tagger: Scott Chacon <schacon@gee-mail.com>

                       Date:    Mon Feb 9 14:45:11 2009 -0800



                                    my version 1.4

commit 15027957951b64cf874c3557a0f3547bd83b3ff6
         Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>
   Date:    Sun Feb 8 19:02:46 2009 -0800
```

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

Подписанные метки

Вы также можете подписывать свои метки с помощью GPG, конечно, если у вас есть ключ. Всё что нужно сделать, это использовать -s вместо -a:

```
                      $ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for
   user: "Scott Chacon <schacon@gee-mail.com>"

                1024-bit DSA key, ID F721C45A, created 2009-02-09
```

Если вы выполните git show на этой метке, то увидите прикреплённую к ней GPG-подпись:

```
$ git show v1.5
    tag v1.5

Tagger: Scott Chacon <schacon@gee-mail.com>
   Date:    Mon Feb 9 15:22:20 2009 -0800

                                my signed 1.5 tag

                            -----BEGIN PGP SIGNATURE-----

                            Version: GnuPG v1.4.8 (Darwin)

  iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
                 Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/

                                        =WryJ

                             -----END PGP SIGNATURE-----

commit 15027957951b64cf874c3557a0f3547bd83b3ff6
          Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>
   Date:    Sun Feb 8 19:02:46 2009 -0800
```

Чуть позже вы узнаете, как верифицировать метки с подписью.

Легковесные метки

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

```
                                $ git tag v1.4-lw

$ git tag
   v0.1

v1.3
v1.4

v1.4-lw
  v1.5
```

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

```
                              $ git show v1.4-lw

commit 15027957951b64cf874c3557a0f3547bd83b3ff6
          Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>
   Date:    Sun Feb 8 19:02:46 2009 -0800

                              Merge branch 'experiment'
```

Верификация меток

Для верификации подписанной метки, используйте git tag -v [имя метки]. Эта команда использует GPG для верификации подписи. Вам нужен открытый ключ автора подписи, чтобы команда работала правильно:

```
$ git tag -v v1.4.2.1

object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit

tag v1.4.2.1

tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1


 Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"

gpg:                 aka "[jpeg image of size 1513]"
```

Если у вас нет открытого ключа автора подписи, вы вместо этого получите что-то подобное:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found

error: could not verify the tag 'v1.4.2.1'
```

Выставление меток позже

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

```
$ git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

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

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

Можете проверить, что коммит теперь отмечен:

```
$ git tag
   v0.1

v1.2
v1.3
v1.4

v1.4-lw
  v1.5

$ git show v1.2
    tag v1.2

Tagger: Scott Chacon <schacon@gee-mail.com>
   Date:    Mon Feb 9 15:32:16 2009 -0800

version 1.2

commit 9fceb02d0ae598e95dc970b74767f19372d61af8
  Author: Magnus Chacon <mchacon@gee-mail.com>

Date:   Sun Apr 27 20:43:35 2008 -0700

                                       updated rakefile
```

Обмен метками

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

```
   $ git push origin v1.5
Counting objects: 50, done.

   Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
    Total 44 (delta 18), reused 8 (delta 1)

                        To git@github.com:schacon/simplegit.git

                           * [new tag]         v1.5 -> v1.5
```

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

```
  $ git push origin --tags
Counting objects: 50, done.

   Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
    Total 44 (delta 18), reused 8 (delta 1)

                        To git@github.com:schacon/simplegit.git

                            * [new tag]         v0.1 -> v0.1

                            * [new tag]         v1.2 -> v1.2
```

```
 * [new tag]             v1.4 -> v1.4
 * [new tag]
 * [new tag]          v1.4-lw -> v1.4-lw
```

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

Полезные советы

Перед тем как закончить данную главу об основах Git, дадим несколько полезных советов о том, как сделать ваш опыт работы с Git проще, удобнее или привычнее. Многие люди используют Git, не прибегая к этим советам, и мы дальше в книге не будем ссылаться на них или подразумевать, что вы ими пользуетесь, но вам всё же стоит знать о них.

Автоматическое дополнение

Если вы используете командую оболочку Bash, Git поставляется с замечательным сценарием (script), который вы можете активировать. Скачайте исходный код Git и посмотрите в каталоге contrib/completion; там должен быть файл git-completion.bash. Скопируйте этот файл в ваш домашний каталог и добавьте следующее в свой файл .bashrc:

```
source ~/.git-completion.bash
```

Если вы хотите настроить автоматическое дополнение в Bash для всех пользователей, скопируйте этот сценарий в каталог /opt/local/etc/bash_completion.d на Mac системах или в каталог /etc/bash_completion.d/ на Linux системах. Это каталог, из которого Bash автоматически загружает сценарии для автодополнения.

Если вы используете Git Bash на Windows, что является стандартным при установке Git на Windows с помощью msysGit, то автодополнение должно быть настроено заранее.

Нажав Tab во время ввода команды Git, вы должны получить набор вариантов на выбор:

```
$ git co<tab><tab>
  commit config
```

В данном случае, набрав git co и дважды нажав клавишу Tab, вы получите как варианты commit и config. Добавление m<tab> выполнит дополнение до git commit автоматически. То же самое работает и для опций, что, возможно, полезней. Например, если вы хотите выполнить команду git log и не помните какую-то опцию, вы можете начать её печатать и затем нажать Tab, чтобы увидеть, что подходит:

```
$ git log --s<tab>

--shortstat  --since=  --src-prefix=  --stat   --summary
```

Это довольно приятная уловка, и она может спасти вам немного времени от работы и чтения документации.

Псевдонимы в Git

Git не будет пытаться сделать вывод о том, какую команду вы хотели ввести, если вы ввели её не полностью. Если вы не хотите печатать каждую команду Git полностью, вы легко можете настроить псевдонимы (alias) для каждой команды с помощью git config. Вот пара примеров того, что вы, возможно, захотите настроить:

```
$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status
```

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

Такой способ может также быть полезен для создания команд, которые, вы думаете, должны существовать. Например, чтобы исправить неудобство, с которым вы столкнулись при исключении файла из индекса (unstage), вы можете добавить собственный псевдоним в Git:

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

Это делает следующие две команды эквивалентными:

```
$ git unstage fileA

$ git reset HEAD fileA
```

Так как будто немного понятней. Также обычно добавляют команду last следующим образом:

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

Так легко можно просмотреть последний коммит:

```
$ git last

commit 66938dae3329c7aebe598c2246a8e6af90d04646
  Author: Josh Goebel <dreamer3@example.com>

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

Можно сказать, что Git просто заменяет эти новые команды на то, для чего вы создавали псевдоним (alias). Однако, возможно, вы захотите выполнять внешнюю команду, а не подкоманду Git. В этом случае, следует начать команду с символа !. Такое полезно, если вы пишите свои утилиты для работы с Git-репозиторием. Продемонстрируем этот случай на примере создания псевдонима git visual для запуска gitk:

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

Итоги

К этому моменту вы умеете выполнять все базовые локальные операции с Git: создавать или клонировать репозиторий, вносить изменения, индексировать и фиксировать эти изменения, а также просматривать историю всех изменений в репозитории. Дальше мы рассмотрим самую убийственную особенность Git’а — его модель ветвления.