Лекция 9. Git изнутри

Лекция 9. Git изнутри

Вы могли прочитать почти всю книгу перед тем, как приступить к этой главе, а могли только часть. Так или иначе, в данной главе рассматриваются внутренние процессы Git и особенности его реализации. На мой взгляд, изучение этих вещей это основа понимания того, насколько Git полезный и мощный инструмент. Хотя некоторые утверждают, что изложение этого материал может сбить новичков с толку и оказаться для них неоправданно сложным. Именно поэтому эта глава отнесена в конец, давая возможность заинтересованным освоить её раньше, а сомневающимся — позже.

Итак, приступим. Во-первых, напомню, что Git это по сути контентно-адресуемая файловая система с пользовательским СУВ-интерфейсом поверх неё. Довольно скоро станет понятнее, что это значит.

На заре развития Git (примерно до версии 1.5), интерфейс был значительно сложнее, поскольку был более похож на интерфейс доступа к файловой системе, чем на законченную СУВ. За последние годы, интерфейс значительно улучшился и по удобству не уступает аналогам; у некоторых, тем не менее, с тех пор сохранился стереотип о том, что интерфейс у Git чересчур сложный и труден для изучения.

Контентно-адресуемая файловая система — основа Git, очень интересна, именно её мы сначала рассмотрим в этой главе; далее будут рассмотрены транспортные механизмы и инструменты обслуживания репозитория, с которыми вам в своё время возможно придётся столкнуться.

Сантехника и фарфор

В этой книге было описано как пользоваться Git используя примерно три десятка команд, например, checkout, branch, remote и т.п. Но так как сначала Git был скорее инструментарием для создания СУВ, чем СУВ удобной для пользователей, в нём полно команд, выполняющих низкоуровневые операции, которые спроектированы так, чтобы их можно было использовать в цепочку в стиле UNIX, а также использовать в сценариях. Эти команды как правило называют служебными («plumbing» — трубопровод), а более ориентированные на пользователя называют пользовательскими («porcelain» — фарфор).

Первые восемь глав книги были посвящены практически только пользовательским командам.

В данной главе же рассматриваются именно низкоуровневые служебные команды, дающие контроль над внутренними процессами Git и показывающие, как он работает и почему он работает так, а не иначе. Предполагается, что данные команды не будут

использоваться напрямую из командной строки, а будут служить в качестве строительных блоков для новых команд и пользовательских сценариев.

Когда вы выполняете git init в новом или существовавшем ранее каталоге, Git создаёт подкаталог .git, в котором располагается почти всё, чем он заправляет. Если требуется выполнить резервное копирование или клонирование репозитория, достаточно скопировать всего лишь один этот каталог, чтобы получить почти всё необходимое. И данная глава почти полностью посвящена его содержимому. Вот так он выглядит:

```
$ ls
HEAD

 branches/
   config
description
   hooks/

  index
  info/
objects/
  refs/
```

Там могут быть и другие файлы, но непосредственно после git init вы увидите именно это. Каталог branchesне используется новыми версиями Git, а файл descriptionтребуется только программе GitWeb, на них не стоит обращать особого внимания. Файл config содержит настройки проекта, а каталог info — файл с глобальным фильтром, игнорирующим те файлы, которые вы не хотите поместить в .gitignore. В каталоге hooks располагаются клиентские и серверные хуки, подробно рассмотренные в главе 7.

Итак, осталось четыре важных элемента: файлы HEAD, index и каталоги objects, refs. Это ключевые элементы хранилища Git. В каталоге objects находится, собственно, база данных, в refs — ссылки на объекты коммитов в этой базе (ветки). Файл HEAD указывает на текущую ветку, и в файле index хранится информация индекса. В последующих разделах данные элементы будут рассмотрены более подробно.

Объекты в Git

Git — контентно-адресуемая файловая система. Здорово. Но что это означает? А означает это, что в своей основе Git — простое хранилище ключ-значение. Можно добавить туда любое содержимое, в ответ будет выдан ключ, по которому это содержимое можно извлечь. Для примера, можно воспользоваться служебной командой hash-object, которая добавляет данные в каталог .git и возвращает ключ. Для начала создадим новый Git-репозиторий и убедимся, что каталог objectsпуст:

```
$ mkdir test

$ cd test

$ git init

Initialized empty Git repository in /tmp/test/.git/

$ find .git/objects
```

```
.git/objects/info

.git/objects/pack

$ find .git/objects -type f

$
```

Git проинициализировал каталог objectsи создал в нём подкаталоги packи info, пока без файлов. Теперь добавим кое-какое текстовое содержимое в базу Git'а:

```
$ echo 'test content' | git hash-object -w --stdin
     d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Ключ -w команды hash-object указывает, что объект необходимо сохранить, иначе команда просто выведет ключ и всё. Флаг --stdinуказывает, что данные необходимо считать со стандартного ввода, в противном случае hash-objectожидает имя файла. Вывод команды — 40-символьная контрольная сумма. Это хеш SHA-1 — контрольная сумма содержимого и заголовка, который будет рассмотрен позднее. Теперь можно увидеть, в каком виде будут сохранены ваши данные:

```
$ find .git/objects -type f

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

В каталоге objects появился файл. Это и есть начальное внутреннее представление данных в Git — один файл на единицу хранения с именем, являющимся контрольной суммой содержимого и заголовка. Первые два символа SHA определяют подкаталог файла, остальные 38 — собственно, имя.

Получить обратно содержимое объекта можно командой cat-file. Это своеобразный швейцарский армейский нож для проверки объектов в Git. Ключ -pозначает автоматическое определение типа содержимого и вывод содержимого на печать в удобном виде:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
                        test content
```

Теперь вы умеете добавлять данные в Git и извлекать их обратно. То же самое можно делать и с файлами. Рассмотрим пример. Наиболее простой контроль версий файла можно осуществить, создав его и сохранив в базе:

```
$ echo 'version 1' > test.txt

     $ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Теперь изменим файл и сохраним его в базе ещё раз:

```
$ echo 'version 2' > test.txt

$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Теперь в базе содержатся две версии файла test.txt, а также самый первый сохранённый объект:

```
$ find .git/objects -type f

.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a

.git/objects/83/baae61804e65cc73a7201a7252750c76066a30

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Теперь можно откатить файл к его первой версии:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt

$ cat test.txt
  version 1
```

или второй:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt

$ cat test.txt
  version 2
```

Однако запоминать хеш для каждой версии неудобно, к тому же теряется само имя файла, сохраняется лишь содержимое. Объекты такого типа называют блобами (англ. binary large object). Имея SHA-1 объекта, можно попросить Git показать нам его тип с помощью команды cat-file -t:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Объекты-деревья

Рассмотрим другой тип объектов Git — деревья. Они решают проблему хранения имён файлов, а также позволяют хранить группы файлов вместе. Система хранения данных Git подобна файловым системам UNIX в упрощённом виде. Содержимое хранится в объектах- деревьях и блобах, дерево соответствует записи каталога в ФС, а блоб более или менее соответствует inode или содержимому файла. Объект-дерево может содержать одну и более записей, каждая из которых представляет собой набор из SHA-1 хеша, соответствующего блобу или поддереву, режима доступа к файлу, типа и имени файла. Например, в проекте simplegit дерево на момент написания выглядит так:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859    README
                                                        Rakefile
```

Запись masterˆ{tree} означает объект-дерево, на который указывает последний коммит ветки master. Заметьте, что подкаталог lib— не блоб, а указатель на другое дерево:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b    simplegit.rb
```

Схематически, данные, которые хранятся в Git, выглядят примерно так, как это изображено на рисунке 9-1.

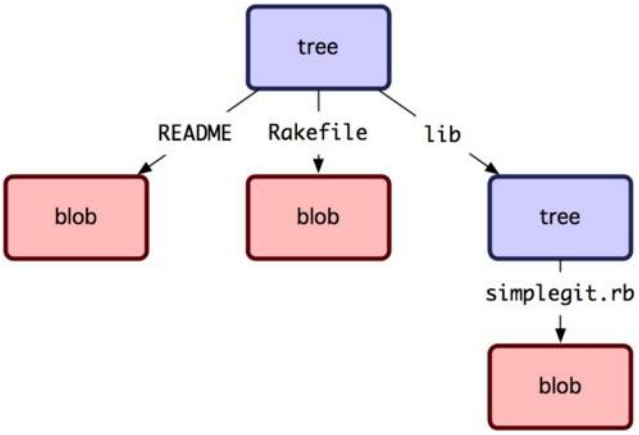

Рисунок 9.1: Упрощённая модель данных Git.

Вручную можно создавать не только блобы, но и деревья. Git обычно создаёт дерево исходя из состояния индекса и затем сохраняет соответствующий объект-дерево. Поэтому для создания объекта-дерева необходимо проиндексировать какие-нибудь файлы. Для создания индекса из одной записи — первой версии файла text.txt, воспользуемся командой update-index. Данная команда может искусственно добавить более раннюю версию test.txt в новый индекс. Необходимо передать опции --add, т.к. файл ещё не существует в индексе (да и самого индекса ещё нет), и --cacheinfo, т.к. добавляемого файла нет в рабочем каталоге, но он есть в базе данных. Также необходимо передать режим доступа, хеш и имя файла:

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

В данном случае режим доступа — 100644, что означает обычный файл. Другие возможные варианты: 100755— исполняемый файл, 120000— символическая ссылка. Режимы доступа в Git сделаны по аналогии с режимами доступа в UNIX, но они гораздо менее гибки: данные три режима — единственные доступные для файлов (блобов) в Git (хотя существуют и другие режимы используемые для каталогов и подмодулей).

Теперь можно воспользоваться командой write-tree для сохранения индекса в объект-дерево. Здесь опция -wne требуется — вызов write-treeавтоматически создаст объект- дерево по состоянию индекса, если такого дерева ещё не существует:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579

$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579

100644 blob 83baae61804e65cc73a7201a7252750c76066a30       test.txt
```

Также можно проверить, что мы действительно создали объект-дерево:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
                              tree
```

Создадим новое дерево со второй версией файла test.txt и ещё одним файлом:

```
$ echo 'new file' > new.txt

$ git update-index test.txt

$ git update-index --add new.txt
```

Теперь в индексе содержится новая версия файла test.txt и новый файл new.txt. Запишем это дерево (сохранив состояние индекса в объект-дерево) и посмотрим, что из этого получилось:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341

$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341

100644 blob fa49b077972391ad58037050f2a75f74e3671e92       new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a       test.txt
```

Заметьте, что в данном дереве находятся записи для обоих файлов, а также, что хеш файла test.txt это хеш «второй версии» этого файла (1f7a7a). Для интереса, добавим первое дерево как подкаталог для текущего. Зачитать дерево в индекс можно командой read-tree. В нашем случае, чтобы прочитать уже существующее дерево в индекс и сделать его поддеревом, необходимо использовать опцию --prefix:

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579

$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614

$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579       bak

100644 blob fa49b077972391ad58037050f2a75f74e3671e92       new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a       test.txt
```

Если бы вы создали рабочий каталог, соответствующий только что созданному дереву, вы бы получили два файла в корне и подкаталог bak со старой версией файла test.txt. Данные, которые хранит Git для такой структуры, представлены на рисунке 9-2.

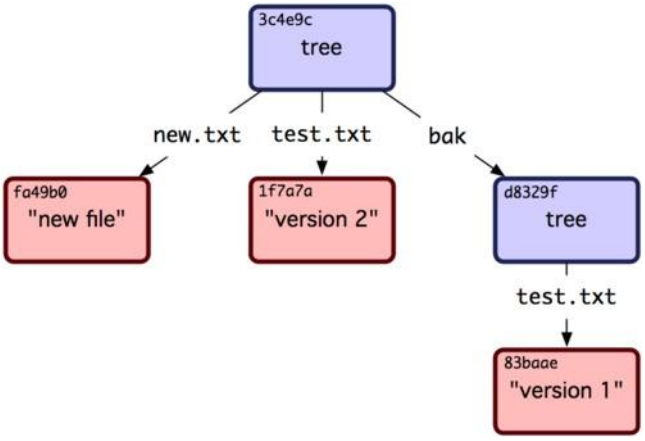

Рисунок 9.2: Структура данных Git для текущего дерева.

Объекты-коммиты

У нас есть три дерева, соответствующих разным состояниям проекта, но предыдущая проблема с необходимостью запоминать все три значения SHA-1, чтобы иметь возможность восстановить какое-либо из этих состояний, ещё не решена. К тому же у нас нет никакой информации о том, кто, когда и почему сохранил их. Такие данные — основная информация, которая хранится в объекте-коммите.

Для создания объекта-коммита необходимо вызвать commit-tree и задать SHA-1 нужного дерева и, если необходимо, родительские объекты-коммиты. Для начала создадим коммит для самого первого дерева:

```
$ echo 'first commit' | git commit-tree d8329f
   fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Просмотреть вновь созданный объект-коммит можно командой cat-file:

```
                              $ git cat-file -p fdf4fc3

                    tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579

                author Scott Chacon <schacon@gmail.com> 1243040974 -0700

committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

                          first commit
```

Формат объекта-коммита прост: в нём указано дерево верхнего уровня, соответствующее состоянию проекта на некоторый момент; имя автора и коммитера берутся из полей конфигурации user.name и user.email; также добавляется текущая временная метка, пустая строка и затем сообщение коммита.

Далее, создадим ещё два объекта-коммита, каждый из которых будет ссылаться на предыдущий коммит:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
         cac0cab538b970a37ea1e769cbbde608743bc96d

$ echo 'third commit'  | git commit-tree 3c4e9c -p cac0cab
         1a410efbd13591db07496601ebc7a059dd55cfe9
```

Каждый из трёх объектов-коммитов указывает на одно из состояний проекта. Может показаться странным, но теперь у нас есть полноценная Git-история, которую можно посмотреть командой git log, указав хеш последнего коммита:

```
                                $ git log --stat 1a410e

commit 1a410efbd13591db07496601ebc7a059dd55cfe9
    Author: Scott Chacon <schacon@gmail.com>

                         Date:    Fri May 22 18:15:24 2009 -0700


        third commit

 bak/test.txt |    1 +

 1 files changed, 1 insertions(+), 0 deletions(-)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
    Author: Scott Chacon <schacon@gmail.com>

                         Date:    Fri May 22 18:14:29 2009 -0700


     second commit

 new.txt  |    1 +

                                    test.txt |    2 +-

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

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
    Author: Scott Chacon <schacon@gmail.com>

                         Date:    Fri May 22 18:09:34 2009 -0700


      first commit
```

Поразительно. Мы только что выполнили низкоуровневые операции для построения истории без использования высокоуровневых интерфейсов. По существу, именно это делает Git, когда выполняются команды git add и git commit — сохраняет блобы для изменённых файлов, обновляет индекс, записывает объекты-деревья и коммит-объекты, ссылающиеся на объекты- деревья верхнего уровня и предшествующие коммиты. Эти три основных вида объектов в Git: блоб, дерево и коммит — сначала сохраняются как отдельные файлы в каталоге .git/ objects. Вот все объекты, которые сейчас лежат в каталоге с примером (в комментариях написано чему объекты соответствует):

```
$ find .git/objects -type f

.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2

.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3

.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2

.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3

.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1

.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'

.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
```

Если пройти по всем внутренним ссылкам, получится граф объектов такой, как на рисунке 9-3.

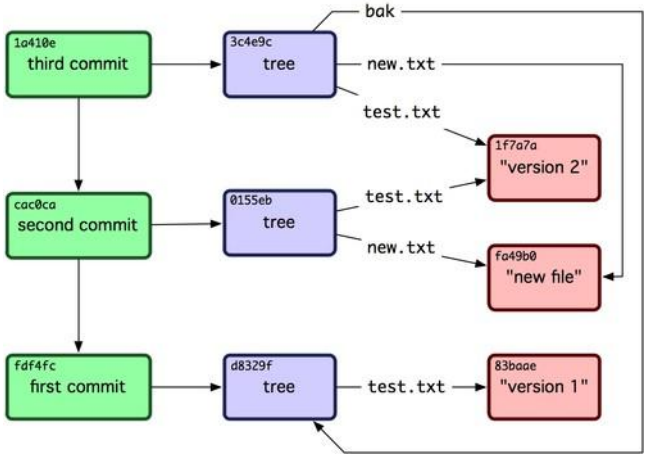

Рисунок 9.3: Все объекты в репозитории Git.

Хранение объектов

Ранее я упоминал, что заголовок сохраняется вместе с содержимым. Давайте посмотрим, как сохраняются объекты Git на диске. Мы рассмотрим сохранение блоб-объекта, в данном случае это будет строка «есть проблемы, шеф?». Пример будет выполнен на языке Ruby. Для запуска интерактивного интерпретатора воспользуйтесь командой irb:

```
$ irb

>> content = "есть проблемы, шеф?"

=> "есть проблемы, шеф?"
```

Git создаёт заголовок, начинающийся с типа объекта, в данном случае это блоб. Далее добавляется пробел, размер содержимого и в конце нулевой байт:

```
>> header = "blob #{content.length}\0"

=> "blob 34\000"
```

Git дописывает содержимое после заголовка и вычисляет SHA-1 сумму для полученного результата. В Ruby значение SHA-1 для строки можно получить, подключив соответствующую библиотеку командой require и затем воспользовавшись вызовом Digest::SHA1.hexdigest():

```
>> store = header + content
=> "blob 34\000\320\225\321\201\321\202\321\214
\320\277\321\200\320\276\320\261\320\273\320\265\320\274\
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=>"d8a734f44240bdf766c8df342664fde23d421d64"
```

Git сжимает новые данные при помощи zlib, что решается в Ruby соответствующей библиотекой.

Сперва, необходимо подключить её, а после вызвать Zlib::Deflate.deflate() с данными в качестве параметра:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\234\001*\000\325\377blob 34\000\320\225\321\201\321\202\321\214
\320\277\321\200\320\276\320\261\32
\3453\030S"
```

После этого, запишем сжатую zlib’ом строку в объект на диск. Определим путь к файлу, который будет записан (первые два символа хеша используются в качестве названия подкаталога, оставшиеся 38 — в качестве имени файла в этом каталоге). В Ruby для этой задачи можно использовать функцию FileUtils.mkdir_p() для создания подкаталога, если он не существует. Далее, откроем файл вызовом File.open() и запишем наши сжатые данные вызовом write() для полученного файлового дескриптора:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/d8/a734f44240bdf766c8df342664fde23d421d64"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
```

Вот и всё, мы создали корректный объект-блоб для Git. Все другие объекты создаются аналогично, меняется только запись о типе в заголовке (blob, commit, tree). Стоит добавить, что хотя в блобе может храниться почти любое содержимое, содержимое объектов-деревьев и объектов-коммитов записывается в очень строгом формате.

Ссылки в Git

Для просмотра всей истории можно выполнить команду вроде git log 1a410e, но, опять же, требуется помнить, что именно коммит 1a410e является последним, чтобы иметь возможность найти все наши объекты. Нам нужен файл-указатель с простым именем, который бы содержал это значение хеша SHA-1, чтобы можно было пользоваться этим файлом вместо хеша.

В Git такие файлы, содержащие SHA-1, называются ссылками («refs») и располагаются в каталоге .git/refs. В нашем проекте этот каталог пока пуст, но в нём уже существует некоторая структура каталогов:

```
$ find .git/refs

.git/refs

.git/refs/heads

.git/refs/tags

$ find .git/refs -type f

$
```

Чтобы создать новую ссылку, которая поможет вам вспомнить, какой коммит последний, по сути, необходимо сделать всего лишь следующее:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Теперь можно использовать только что созданную ссылку из каталога heads вместо хеша в командах Git:

```
$ git log --pretty=oneline  master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Тем не менее, редактировать данные файлы напрямую не рекомендуется. Git предоставляет безопасную команду update-refдля изменения ссылок:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Вот что такое по сути ветка в Git — простой указатель или ссылка на последнюю версию в работе. Для создания ветки, соответствующей состоянию второго коммита, можно выполнить следующее:

```
$ git update-ref refs/heads/test cac0ca
```

Данная ветка будет содержать только коммиты, предшествующие выбранному:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Теперь наша база данных Git схематично выглядит так, как показано на рисунке 9.4.

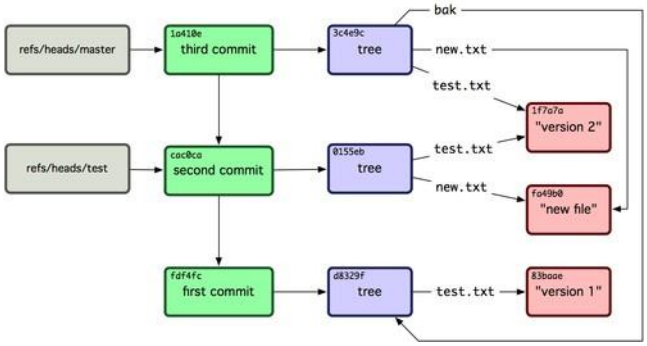

Рисунок 9.4: Объекты в каталоге .git, а также указатели на вершины веток.

Когда выполняется команда git branch (имя ветки), Git, по сути, выполняет update-ref для добавления хеша последнего коммита текущей ветки под указанным именем в виде новой ссылки.

HEAD

Вопрос в том, как же Git получает хеш последнего коммита при выполнении git branch (имя ветки)? Ответ содержится в файле HEAD. Данный файл является символической ссылкой на текущую ветку. Символическая ссылка отличается от обычной тем, что она содержит не сам хеш SHA-1, а указатель на другую ссылку. Если вы загляните в этот файл, то увидите что-то такое:

```
$ cat .git/HEAD

ref: refs/heads/master
```

Если выполнить git checkout test, то содержимое файла изменится:

```
$ cat .git/HEAD

ref: refs/heads/test
```

При выполнении git commit, Git создаёт объект-коммит, указывая его родителем тот объект, SHA-1 которого содержится в файле, на который ссылается HEAD.

Данный файл, конечно, можно редактировать вручную, но безопаснее использовать команду symbolic-ref. Получить значение HEAD данной командой можно так:

```
$ git symbolic-ref HEAD
   refs/heads/master
```

Изменить значение HEAD можно так:

```
$ git symbolic-ref HEAD refs/heads/test

$ cat .git/HEAD

ref: refs/heads/test
```

Символическую ссылку на файл вне refs поставить нельзя:

```
$ git symbolic-ref HEAD test

fatal: Refusing to point HEAD outside of refs/
```

Метки

Мы рассмотрели три основных типа объектов в Git, но есть и четвёртый. Объект-метка очень похож на объект-коммит: он содержит имя поставившего метку, дату, сообщение и указатель. Разница же в том, что метка указывает на коммит, а не на дерево. Она похожа на ветку, которая никогда не перемещается — она всегда указывает на один и тот же коммит, она просто даёт ему понятное имя.

Как было сказано в главе 2, метки бывают двух типов: аннотированные и легковесные.

Легковесную метку можно сделать следующей командой:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Вот и всё! Легковесная метка — это ветка, которая никогда не перемещается. Аннотированная метка имеет более сложную структуру. При создании аннотированной метки, Git создаёт специальный объект, на который будет указывать ссылка, а не просто указатель на коммит.

Мы можем увидеть это создав аннотированную метку (-a задаёт аннотированные метки):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Вот значение SHA-1 созданного объекта:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Теперь выполним cat-filedля этого хеша:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
      object 1a410efbd13591db07496601ebc7a059dd55cfe9

type commit
  tag v1.1

tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

                                  test tag
```

Заметьте, в поле object записан SHA-1 коммита, для которого мы делали метку. Также стоит отметить, что это поле не обязательно указывает на коммит, но на любой объект в Git. Например, в исходный код Git мейнтейнер добавил свой открытый GPG-ключ в качестве блоба и поставил для него метку. Увидеть этот ключ можно, выполнив команду

```
$ git cat-file blob junio-gpg-pub
```

в репозитории с исходным кодом Git. В репозитории ядра Linux также есть метка, указывающая не на коммит — первая метка указывает на дерево первичного импорта.

Ссылки на удалённые ветки

Третий тип ссылок, который мы рассмотрим — ссылка на удалённую ветку. Если вы добавили удалённый репозиторий и отправили (push) на него изменения, Git сохранит последнее отправленное значение SHA-1 в каталоге refs/remotes для всех отправленных веток. Например, можно добавить удалённый репозиторий origin и отправить туда ветку master:

```
          $ git remote add origin git@github.com:schacon/simplegit-progit.git

  $ git push origin master
Counting objects: 11, done.

    Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
     Total 7 (delta 2), reused 4 (delta 1)

To git@github.com:schacon/simplegit-progit.git
        a11bef0..ca82a6d  master -> master
```

Позже, вы сможете посмотреть где находилась ветка master с сервера origin во время последнего соединения с сервером заглянув в файл refs/remotes/origin/master:

```
  $ cat .git/refs/remotes/origin/master
 ca82a6dff817ec66f44342007202690a93763949
```

Ссылки на удалённые ветки отличаются от обычных веток (ссылки в refs/heads) тем, что на них нельзя переключиться с помощью git checkout. Git работает с ними как с закладками, указывающими на последнее состояние соответствующих веток на ваших серверах.

Pack-файлы

Вернёмся к базе объектов в нашем тестовом репозитории. К этому моменту их должно быть 11 штук: 4 блоба, 3 дерева, 3 коммита и одна метка:

```
$ find .git/objects -type f

.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2

.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3

.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2

.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3

.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1

.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag

.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
```

Git сжал содержимое этих файлов при помощи zlib, к тому же мы не записывали много данных, поэтому все эти файлы вместе занимают всего 925 байт. Для того, чтобы продемонстрировать одну интересную возможность Git, добавим файл побольше. Добавим файл repo.rb из библиотеки Grit, с которой мы работали ранее, он занимает примерно 12 Кбайт:

```
$ curl http://github.com/mojombo/grit/raw/master/lib/grit/repo.rb > repo.rb

$ git add repo.rb

$ git commit -m 'added repo.rb'
 [master 484a592] added repo.rb

 3 files changed, 459 insertions(+), 2 deletions(-)
          delete mode 100644 bak/test.txt

 create mode 100644 repo.rb
  rewrite test.txt (100%)
```

Если мы посмотрим на полученное дерево, мы увидим значение SHA-1, которое получил блоб для файла repo.rb:

```
$ git cat-file -p master^{tree}

100644 blob fa49b077972391ad58037050f2a75f74e3671e92          new.txt

100644 blob 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e          repo.rb
```

Для определения размера этого объекта воспользуемся командой git cat-file:

```
$ git cat-file -s 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e
12898
```

Теперь изменим немного данный файл и посмотрим на результат:

```
                              $ echo '# testing' >> repo.rb

$ git commit -am 'modified repo a bit'
 [master ab1afef] modified repo a bit

                      1 files changed, 1 insertions(+), 0 deletions(-)
```

Взглянув на дерево полученное в результате коммита, мы увидим любопытную вещь:

```
           $ git cat-file -p master^{tree}
                                                                  new.txt
100644 blob fa49b077972391ad58037050f2a75f74e3671e92
                                                                  repo.rb
```

Теперь файлу repo.rb соответствует другой объект-блоб. Это означает, что даже одна единственная строка добавленная в конец 400-строчного файла требует создания абсолютно нового объекта:

```
$ git cat-file -s 05408d195263d853f09dca71d55116663690c27c
                              12908
```

Итак, мы имеем два почти одинаковых объекта по 12 Кбайт занимающих место на диске. Было бы неплохо, если бы Git сохранял только один объект целиком, а другой как разницу между ними.

Оказывается, что Git так и делает. Первоначальный формат для сохранения объектов в Git называется рыхлым форматом (англ. loose format) объектов. Однако, время от времени Git упаковывает несколько таких объектов в один pack-файл (pack в пер. с англ. — упаковывать, уплотнять) для сохранения места на диске и повышения эффективности. Это происходит, когда «рыхлых» объектов становится слишком много, а также при вызове git gc вручную, и при отправке изменений на удалённый сервер. Чтобы посмотреть, как происходит упаковка, можно выполнить команду git gc:

```
                                    $ git gc

                               Counting objects: 17, done.

   Delta compression using 2 threads.
Compressing objects: 100% (13/13), done.
  Writing objects: 100% (17/17), done.

                          Total 17 (delta 1), reused 10 (delta 0)
```

Если вы загляните в каталог с объектами, вы обнаружите, что большая часть объектов исчезла, зато появились два новых файла:

```
                               $ find .git/objects -type f

                  .git/objects/71/08f7ecb345ee9d0084193f147cdad4d2998293
```

```
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack
```

Оставшиеся объекты — блобы, на которые не указывает ни один коммит. В нашем случае это созданные ранее объекты: содержащий строку «есть проблемы, шеф?», и блоб содержащий «test content». В силу того, что ни в одном коммите данные файлы не присутствуют, они считаются «висячими» и не упаковываются.

Остальные файлы — это pack-файл и его индекс. Pack-файл — это файл, который теперь содержит все объекты, которые были удалены. А индекс — это файл, в котором записаны их смещения в pack-файле, что даёт возможность быстро найти нужный объект. Упаковка данных положительно повлияла на общий размер файлов, если до вызова gc они занимали примерно 12 Кбайт, то pack-файл занимает всего 6 Кбайт. Упаковкой объектов мы смогли сократить место занятое на диске в два раза.

Как Git это делает? При упаковке Git ищет файлы, которые похожи по имени и размеру и сохраняет только разницу между двумя версиями файла. Можно рассмотреть pack-файл подробнее и понять, какие действия были выполнены для сжатия. Для просмотра содержимого упакованного файла существует служебная команда git verify-pack:

Здесь блоб 9bc1d, который, как мы помним, был первой версией файла repo.rb, ссылается на блоб 05408, который был второй его версией. Третья колонка в выводе — это размер объекта. Как видите, 05408 занимает в файле 12 Кбайт, при этом 9bc1d занимает всего лишь 7 байт. Что интересно, вторая версия сохраняется «как есть», а исходная — в виде дельты. Это из-за того, что необходимость получения доступа к последней версии файла является более вероятной.

Также здорово, что переупаковку можно выполнять в любое время. Время от времени Git будет выполнять её автоматически, чтобы сэкономить место на диске, если вдруг этого недостаточно, всегда можно выполнить git gc вручную.

Спецификации ссылок

Во всей книге использовались простые связи между ветками в удалённых репозиториях и локальными ветками, но они могут быть и более сложными. Предположим, мы добавили следующий удалённый репозиторий:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
```

Данный вызов добавляет секцию в файл .git/config, в которой заданы имя удалённого репозитория (origin), его URL и спецификация ссылок для извлечения данных:

```
[remote "origin"]

url = git@github.com:schacon/simplegit-progit.git
   fetch = +refs/heads/*:refs/remotes/origin/*
```

Формат спецификации следующий: опциональный +, далее пара <src>:<dst>, где <src>— шаблон ссылок в удалённом репозитории, а <dst> — соответствующий шаблон локальных ссылок. Символ + сообщает Git, что обновление необходимо выполнять даже в том случае, если оно не является перемоткой.

В случае настроек по умолчанию, которые записываются во время выполнения git remote add, Git выбирает все ссылки из refs/heads/ на стороне сервера, и записывает их в локальный каталог refs/remotes/origin/. Таким образом, если на сервере есть ветка master, журнал данной ветки можно получить, вызвав:

```
$ git log origin/master

$ git log remotes/origin/master

$ git log refs/remotes/origin/master
```

Все эти команды эквивалентны, так как Git развернёт каждую запись до refs/remotes/origin/master.

Если хочется, чтобы Git забирал при обновлении только ветку master, а не все доступные на сервере, можно изменить соответствующую строку в файле конфигурации на следующее:

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Данный refspec будет использоваться по умолчанию при вызове git fetch для данного удалённого репозитория. Если же вам нужно изменить спецификацию всего раз, можно задать refspec в командной строке. Например, чтобы получить данные из ветки master из удалённого репозитория в локальную origin/mymaster, можно выполнить

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Конечно, можно задать несколько спецификаций. Получить данные нескольких веток из командной строки можно так:

```
$ git fetch origin master:refs/remotes/origin/mymaster \
             topic:refs/remotes/origin/topic

                        From git@github.com:schacon/simplegit

        ! [rejected]        master     -> origin/mymaster  (non fast forward)
```

В данном случае, слияние ветки master выполнить не удалось, поскольку слияние не было просто перемоткой. Такое поведение можно изменить, добавив перед спецификацией знак +.

В конфигурационном файле также можно задавать несколько спецификаций для получения обновлений. Чтобы каждый раз получать обновления веток master и experiment, добавьте две такие строки:

```
[remote "origin"]

  url = git@github.com:schacon/simplegit-progit.git
fetch = +refs/heads/master:refs/remotes/origin/master

        fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Задавать частичные регулярные выражения в спецификации нельзя, следующая запись неверна:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Тем не менее, можно использовать пространства имён для получения похожего результата. Если имеется команда QA (сокр. от quality assurance — контроль качества), которая использует свои несколько веток, и вы хотите получать только ветку master и все ветки команды QA, а остальные — нет, то можно добавить в конфигурацию следующее:

```
[remote "origin"]

  url = git@github.com:schacon/simplegit-progit.git
fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

Если ваш рабочий процесс является сложным, и разные команды: разработчики, тестеры, внедренцы — коммитят в разные ветки одного и того же проекта, то так вы с лёгкостью можете разделить их по разным пространствам имён.

Спецификации ссылок для команды push

Это хорошо, что мы научились получать данные по ссылкам в отдельных пространствах имён, но нам же ещё надо сделать так, чтобы команда QA сначала смогла отправить свои ветки в пространство имён qa/. Мы решим эту задачу используя спецификации ссылок для команды push.

Если разработчик из команды QA хочет отправить изменения из локальной ветки master в qa/masterна удалённом сервере, он может выполнить команду

```
$ git push origin master:refs/heads/qa/master
```

Если хочется, чтобы Git автоматически делал так при вызове git push origin, можно добавить в конфигурационный файл значение для push:

```
                                [remote "origin"]

        url = git@github.com:schacon/simplegit-progit.git
           fetch = +refs/heads/*:refs/remotes/origin/*

                          push = refs/heads/master:refs/heads/qa/master
```

Опять же, это приведёт к тому, что при вызове git push originлокальная ветка master будет по умолчанию отправляться в удалённую ветку qa/master.

Удаление ссылок

Кроме всего прочего, спецификации ссылок можно использовать следующим образом для удаления ссылок на удалённом сервере:

```
                                $ git push origin :topic
```

Так как спецификация ссылки задаётся в виде <src>:<dst>, опускание <src>означает, что указанную ветку на удалённом сервере надо сделать пустой, что приводит к её удалению.

Протоколы передачи

Git может передавать данные между репозиториями одним из двух основных способов: через HTTP или через «умные» протоколы для транспортов file://, ssh://и git://. В данном разделе мы кратко рассмотрим как эти два протокола работают.

Тупой протокол

Git-транспорт работающий по HTTP часто называют «тупым» протоколом, потому что для его работы во время передачи данных не требуется исполнения никакого Git-специфичного кода на стороне сервера. Процесс извлечения данных представляет собой последовательность GET-запросов, клиент обращается к стандартной структуре каталогов Git. Давайте рассмотрим процесс получения данных по HTTP на примере библиотеки simplegit:

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

Первое действие, выполняемое данной командой — загрузка файла info/refs. Данный файл записывается командой update-server-info, поэтому для использования HTTP-транспорта необходимо запускать эту команду в post-receivехуке:

```
            => GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

Теперь у нас имеется список удалённых веток и их хеши. Далее, нам надо посмотреть куда ссылается HEAD, чтобы знать на какую версию переключиться после завершения работы команды.

```
=> GET HEAD

ref: refs/heads/master
```

Нам надо переключиться на ветку master после завершения процесса. На данном этапе можно начать обход дерева. Начальной точкой является объект-коммит ca82a6, о чём мы узнали из файла info/refs, и мы начинаем с его загрузки:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

Объект получен, он был в рыхлом формате на сервере, и мы получили его по HTTP используя статический GET-запрос. Теперь можно его разархивировать, отрезать заголовок и посмотреть на его содержимое:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf

parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

author Scott Chacon <schacon@gmail.com> 1205815931 -0700

committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Далее, необходимо загрузить ещё два объекта: cfda3b — объект-дерево, который обозначен как содержимое только что загруженного коммита, и 085bb3 — родительский коммит:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

Так, мы получили следующий объект-коммит. Прихватим и наш объект-дерево:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Ой! Похоже, этого объекта-дерева нет на сервере в рыхлом формате, поэтому мы получили ответ 404. У этого могут быть разные причины: объект в другом репозитории, или в упакованном файле текущего репозитория. Сперва Git проверяет список альтернативных репозиториев:

```
=> GET objects/info/http-alternates
(empty file)
```

Если бы этот запрос вернул нам список альтернативных URL, Git обратился по ним в поиске «рыхлых» и pack-файлов — это такой механизм, позволяющий не дублировать данные проектам, являющимися форками друг для друга. Так как в данном случае альтернативных адресов нет, объект должен быть в pack-файле. Для того, чтобы узнать, какие упакованные файлы есть на сервере, необходимо загрузить файл со списком pack-файлов: objects/info/ packs(который также генерируется update-server-info):

```
=> GET objects/info/packs

P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

На сервере имеется только один pack-файл, поэтому объект точно там, но необходимо проверить индексный файл, чтобы в этом убедиться. Если бы на сервере было несколько pack-файлов, загрузив сначала индексы, мы смогли бы определить в каком именно pack-файле находится нужный нам объект:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
                           (4k of binary data)
```

Теперь, когда мы получили индекс упакованного файла, можно проверить, тут ли наш объект. Это возможно благодаря тому, что в индексе хранятся SHA-1 объектов содержащихся в pack-файле, а также их смещения. Необходимый объект там присутствует, так что продолжим и получим весь pack-файл:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
                           (13k of binary data)
```

Итак, мы получили наш объект-дерево, можно продолжить обход списка коммитов. Все они лежат внутри упакованного файла, который мы только что скачали, так что снова обращаться к серверу не надо. Git извлекает рабочую копию ветки master, на которую ссылается HEAD.

Полный вывод этого процесса выглядит так:

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

Initialized empty Git repository in /private/tmp/simplegit-progit/.git/
              got ca82a6dff817ec66f44342007202690a93763949

walk ca82a6dff817ec66f44342007202690a93763949
 got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Getting alternates list for http://github.com/schacon/simplegit-progit.git
   Getting pack list for http://github.com/schacon/simplegit-progit.git
      Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835

  Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835
 which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf

walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
```

Умный протокол

Методика работы HTTP проста, но неэффективна, поэтому чаще используются «умные» протоколы. Эти протоколы обслуживаются процессом на стороне сервера, который учитывает особенности работы Git — он считывает локальные данные, выясняет что есть и чего не хватает на клиенте и генерирует для него соответствующие данные. Существует два набора процессов передачи данных: процессы для загрузки данных и процессы для скачивания.

Загрузка данных Для загрузки данных на удалённый сервер используются процессы send- pack и receive-pack. Процесс send-pack запускается на стороне клиента и подключается к receive-pack на стороне сервера.

Например, выполняется команда git push origin master и origin определён как URL использующий протокол SSH. Git запускает процесс send-pack, который устанавливает соединение с сервером по протоколу SSH. Он пытается запустить команду на удалённом сервере через вызов команды ssh, который выглядит следующим образом:

```
$ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic

0000
```

Команда git-receive-pack тут же посылает в ответ по одной строке на каждую из имеющихся в наличии ссылок — в данном случае только ветку master и её SHA. Первая строка также содержит список возможностей сервера (здесь это report-status и delete- refs).

Каждая строка начинается с 4-байтового шестнадцатеричного значения, содержащего длину оставшейся строки. Первая строка начинается с 005b, это 91 в 16-ричном виде, значит в этой строке ещё 91 байт. Следующая строка начинается с 003e, что означает 62, то есть надо прочитать 62 байта. Далее следует строка 0000, которая означает, что сервер закончил листинг своих ссылок.

Теперь, когда процесс send-pack выяснил состояние сервера, он определяет коммиты, которые есть локально, но которых нет на сервере. Для каждой ссылки, которая будет обновлена текущей командой push, процесс send-pack передаёт процессу receive-pack эти данные. Например, если мы обновляем ветку master, и добавляем ветку experiment, ответ send- pack будет выглядеть следующим образом:

0085ca82a6dff817ec66f44342007202690a93763949
15027957951b64cf874c3557a0f3547bd83b3ff6refs/
heads/master report-status
006700000000000000000000000000000000000000
cdfdb42577e2506715f8cfeacdbabc092bf63e8d refs/

heads/experiment 0000

Значение SHA-1 из одних нулей означает, что раньше здесь ничего не было — так получилось из-за того, что мы добавили новую ссылку experiment. Если бы мы удаляли ссылку, было бы на оборот: одни нули были бы справа. Git отправляет строку для каждой ссылки, для которой производится обновление. В строке содержится старый хеш, новый хеш и имя обновляемой ссылки. Первая строка также содержит возможности клиента. Далее, клиент загружает упакованный файл со всеми объектами, которых ещё нет на сервере. В конце, сервер отвечает статусным сообщением сообщающем об успехе (или ошибке):

```
000Aunpack ok
```

Скачивание данных Если выполняется скачивание данных, используются процессы fetch- pack и upload-pack. Клиент запускает процесс fetch-pack, который подключающийся к процессу upload-packна удалённой машине для определения, какие данные будут переданы.

Существуют разные способы запуска upload-pack на удалённом репозитории. Можно запустить его по SSH так же, как и receive-pack. Ещё можно вызвать процесс через Git-демон, по умолчанию принимающий соединения на порте 9418. Процесс fetch-pack после подключения отправляет демону данные примерно следующего вида:

```
003fgit-upload-pack    schacon/simplegit-progit.git\0host=myserver.com\0
```

Начальные 4 байта задают размер последующих данных, далее следует команда, которую следует запустить, завершаемая нулевым байтом, а потом имя сервера и последний нулевой байт. Git-демон проверяет возможность выполнения команды, а также, что репозиторий существует и имеет необходимые права доступа. Если всё хорошо, демон запускает процесс upload- packи передаёт запрос ему.

Если извлечение данных производится по SSH, fetch-packвыполняет другие действия:

```
$ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

В обоих случаях, после того как fetch-pack подключится, upload-pack передаст обратно следующее:

```
0088ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag

003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
                            0000
```

Это очень похоже на ответ receive-pack, но только возможности другие. В добавок

upload-pack отсылает обратно ссылку HEAD, чтобы клиент понимал, на какую ветку переключиться, если выполняется клонирование.

На данном этапе процесс fetch-pack смотрит на объекты, имеющиеся в наличии и для недостающих объектов отвечает словом «want» и за ним SHA объекта. Для уже имеющихся объектов процесс отправляет их хеши со словом «have». В конце списка он пишет «done», и это даёт понять процессу upload-pack, что пора начинать отправлять упакованный файл с необходимыми данными:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
     0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

                                        0000

                                      0009done
```

Это самый основной случай передачи данных. В более сложных случаях, клиент поддерживает функции multi_ack или side-band, но этот пример иллюстрирует основные взаимодействия используемые процессами умного протокола.

Обслуживание и восстановление данных

Иногда, требуется выполнить очистку — сделать репозиторий более компактным, почистить импортированный репозиторий, или восстановить потерянную работу. Данный раздел охватывает некоторые из этих сценариев.

Обслуживание

Иногда Git сам выполняет команду запускающую автоматический сборщик мусора. Чаще всего, эта команда ничего не делает. Однако, если неупакованных объектов слишком много,

или у вас слишком много pack-файлов, Git запускает полноценную команду git gc. Здесь gc это сокращение от «garbage collect», что означает «сборка мусора». Эта команда выполняет несколько действий: собирает все объекты в рыхлом формате и упаковывает их в pack-файлы, объединяет несколько упакованных файлов в один большой, удаляет объекты недостижимые ни из одного коммита и те, которые хранятся дольше нескольких месяцев.

Вы также можете запустить сборку мусора вручную:

```
$ git gc --auto
```

Опять же, как правило, эта команда ничего не делает. Необходимо иметь 7000 несжатых объектов или более 50 упакованных файлов, чтобы запустился настоящий gc. Данные пределы можно изменить с помощью параметров gc.auto и gc.autopacklimit в конфигурационном файле.

Другое действие, выполняемое gc— упаковка ссылок в единый файл. Предположим, репозиторий содержит следующие ветки и теги:

```
$ find .git/refs -type f

.git/refs/heads/experiment

.git/refs/heads/master

.git/refs/tags/v1.0

.git/refs/tags/v1.1
```

Если выполнить git gc, данные файлы в каталоге refs перестанут существовать. Git перенесёт их в файл .git/packed-refsв угоду эффективности. Файл будет иметь следующий вид:

```
$ cat .git/packed-refs

# pack-refs with: peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1

^1a410efbd13591db07496601ebc7a059dd55cfe9
```

При обновлении ссылки, Git не будет редактировать этот файл, а добавит новый файл в refs/heads. Для получения хеша для нужной ссылки, Git сначала проверит наличие ссылки в каталоге refs, а к файлу packed-refsобратится только в случае неудачи. Однако, если в каталоге refsфайла нет, скорее всего, он в packed-refs.

Заметьте, последняя строка файла начинается с ^. Это означает, что метка непосредственно над ней является аннотированной и данная строка это коммит, на который аннотированная метка указывает.

Восстановление данных

В какой-то момент при работе с Git, вы нечаянно можете потерять коммит. Как правило, такое случается, когда вы удаляете ветку, в которой находились некоторые наработки, а потом оказывается, что они всё-таки были нужными. Либо вы жёстко сбросили ветку, тем самым отказавшись от коммитов, которые теперь понадобились. Как же в таком случае заполучить свои коммиты обратно?

Рассмотрим пример, в котором жёстко сбросим ветку master в тестовом репозитории на какой-нибудь более ранний коммит и затем восстановим потерянные коммиты. Для начала, рассмотрим в каком состоянии находится репозиторий на данном этапе:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Теперь сдвинем ветку masterна несколько коммитов назад:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
            HEAD is now at 1a410ef third commit

              $ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Итак, теперь два последних коммита по-настоящему потеряны — они не достижимы ни из одной ветки. Необходимо найти SHA последнего коммита и создать ветку, указывающую на него. Сложность в том, чтобы найти этот самый SHA последнего коммита, ведь вряд ли вы его запомнили, да?

Зачастую, самый быстрый способ — использовать инструмент под названием git reflog. Во время вашей работы, Git записывает все изменения HEAD. Каждый раз при переключении веток и коммите, добавляется запись в reflog. Также обновление производится при вызове git update-ref, это, в частности, является причиной необходимости использования этой команды вместо прямой записи значения хеша в ref-файл, как было рассмотрено в разделе «Ссылки в Git» в этой главе. Итак, изменения HEAD в хронологическом порядке можно увидеть, вызвав git reflog:

```
                                    $ git reflog

1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
ab1afef HEAD@{1}: ab1afef80fac8e34258ff41fc1b867c702daa24b: updating HEAD
```

Здесь мы видим два коммита, на которых мы когда-то находились, однако информации не так много. Более интересный вывод можно получить, используя git log -g, что даст стандартный вывод лога для записей из reflog:

```
                                    $ git log -g

  commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
           Reflog message: updating HEAD

                       Author: Scott Chacon <schacon@gmail.com>

                        Date:   Fri May 22 18:22:37 2009 -0700

                                      third commit

  commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
           Reflog message: updating HEAD

                       Author: Scott Chacon <schacon@gmail.com>

                        Date:   Fri May 22 18:15:24 2009 -0700

                                   modified repo a bit
```

Похоже, что нижний коммит это тот, который мы потеряли, и он может быть восстановлен созданием ветки, указывающей на него. Например, создадим ветку с именем recover-branch, указывающую на этот коммит (ab1afef):

```
$ git branch recover-branch ab1afef

$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Здорово, теперь у нас есть ветка recover-branch, указывающая туда, куда ранее указывала master, и потерянные коммиты вновь доступны. Теперь, положим, потерянная ветка по какой-то причине не попала в reflog, для этого удалим восстановленную ветку и весь reflog.

Теперь два первых коммита недоступны ниоткуда:

```
$ git branch -D recover-branch

$ rm -Rf .git/logs/
```

Теперь данные из .git/logs/ удалены, а значит и reflog больше нет, так как все его данные находились там. Как восстановить коммиты теперь? Один способ — использовать утилиту git fsck, проверяющую базу на целостность. Если выполнить её с ключом -- full, будут показаны все объекты недостижимые из других объектов:

```
$ git fsck --full

dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

В данном случае потерянный коммит указан после слов «dangling commit» (dangling com- mit в пер. с англ. — «висячий» коммит). Его можно восстановить аналогичным образом, добавив ветку, указывающую на данный хеш.

Удаление объектов

У Git есть много замечательных особенностей, но одна из них способна вызвать проблемы — команда git clone загружает проект вместе со всей историей включая все версии всех файлов. Это нормально, если в репозитории хранится только исходный код, так как Git хорошо оптимизирован под такой тип данных и может эффективно сжимать их. Однако, если когда- либо в проект был добавлен большой файл, каждый кто потом захочет клонировать проект будет вынужден скачивать этот большой файл, даже если он был

удалён в следующем же коммите. Он будет в базе всегда просто потому, что он доступен в истории.

Это может стать огромной проблемой при конвертации репозиториев Subversion или Per- force в Git. В данных системах вам не нужно загружать всю историю, поэтому добавление больших бинарных файлов не имеет там особых последствий. Если при импорте из другой системы или при каких-либо других обстоятельствах стало ясно, что ваш репозиторий намного больше, чем он должен быть, то как раз сейчас мы расскажем как можно найти и удалить большие объекты.

Будьте внимательны, данный способ разрушителен по отношению к истории коммитов. Каждый коммит будет переписан начиная с самого раннего, из которого вы удалите ссылку на большой файл. Если сделать это непосредственно после импорта, когда никто ещё не работал с репозиторием, всё хорошо, иначе придётся сообщать всем участникам разработки о необходимости перемещения их правок на новые коммиты.

Для примера, добавим большой файл в свой тестовый репозиторий, удалим его в следующем коммите, а потом найдём и удалим его полностью из базы. Для начала добавим большой файл в нашу историю:

```
$ curl http://kernel.org/pub/software/scm/git/git-1.6.3.1.tar.bz2 > git.tbz2

$ git add git.tbz2

$ git commit -am 'added git tarball'
 [master 6df7640] added git tarball

 1 files changed, 0 insertions(+), 0 deletions(-)
           create mode 100644 git.tbz2
```

Упс, кажется, этот огромный архив нам в проекте не нужен. Избавимся от него:

```
$ git rm git.tbz2
  rm 'git.tbz2'

$ git commit -m 'oops - removed large tarball'
 [master da3f30d] oops - removed large tarball

 1 files changed, 0 insertions(+), 0 deletions(-)
           delete mode 100644 git.tbz2
```

Теперь «соберём мусор» в базе и узнаем её размер:

```
$ git gc

Counting objects: 21, done.

   Delta compression using 2 threads.
Compressing objects: 100% (16/16), done.
  Writing objects: 100% (21/21), done.

Total 21 (delta 3), reused 15 (delta 1)
```

Чтобы быстро узнать сколько у нас занято места, можно воспользоваться командой count- objects:

```
$ git count-objects -v
       count: 4

                                        size: 16

                                      in-pack: 21

                                        packs: 1

                                    size-pack: 2016

                                   prune-packable: 0
```

Запись size-pack— это размер упакованных файлов в килобайтах, то есть всего занято 2 МБ. Перед последним коммитом, использовалось около 2 КБ, то есть, удаление файла не удалило его из истории. Из-за того, что мы однажды случайно добавили большой файл, при каждом клонировании этого репозитория каждому человеку придётся скачивать все эти 2 МБ, только для того, чтобы получить этот крошечный проект. Попробуем избавиться от этого файла.

Сперва найдём его. В данном случае, мы знаем, что это за файл. Но если бы не знали, как можно было бы определить, какие файлы занимают много места? При вызове git gc все объекты упаковываются в один файл, несмотря на это определить самые крупные файлы можно запустив служебную команду git verify-pack, и отсортировав её вывод по третьей колонке, в которой записан размер файла. К тому же, так как нас интересуют только самые крупные файлы, оставим только последние несколько строк, направив вывод команде tail:

```
$ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -n | tail -3
```

Большой объект в самом внизу, его размер — 2 МБ. Для того, чтобы узнать, что это за файл, воспользуемся командой rev-list, которая уже упоминалась в главе 7. Если передать ей ключ --objects, то она выдаст хеши всех коммитов, а также хеши объектов и соответствующие им имена файлов. Воспользуемся этим для определения имени выбранного объекта:

```
$ git rev-list --objects --all | grep 7a9eb2fb
7a9eb2fba2b1811321254ac360970fc169ba2330 git.tbz2
```

Теперь необходимо удалить данный файл из всех деревьев в прошлом по истории. Легко получить все коммиты, которые меняли данный файл:

```
$ git log --pretty=oneline -- git.tbz2
da3f30d019005479c99eb4c3406225613985a1db oops - removed large tarball
6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 added git tarball
```

Необходимо переписать все коммиты, начиная с 6df76для полного удаления данного файла. Для этого воспользуемся командой filter-branch, которая приводилась в главе 6:

```
$ git filter-branch --index-filter \

'git rm --cached --ignore-unmatch git.tbz2' -- 6df7640^..

Rewrite 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 (1/2)rm 'git.tbz2'
       Rewrite da3f30d019005479c99eb4c3406225613985a1db (2/2)

Ref 'refs/heads/master' was rewritten
```

Опция --index-filterпохожа на --tree-filterиспользовавшуюся в главе 6, за исключением того, что вместо передачи команды, модифицирующей файлы на диске, мы используем команду, изменяющую файлы в индексе. Вместо удаления файла чем-то вроде rm file, стоит сделать это командой git rm --cached, так как нам надо удалить файл из индекса, а не с диска. Причина, по которой мы делаем именно так — скорость. Нет необходимости извлекать каждую ревизию на диск, чтобы применить фильтр, а это может очень сильно ускорить процесс.

Можете использовать и tree-filterдля получения аналогичного результата, если хотите. Опция --ignore-unmatch команды git rm отключает вывод сообщения об ошибке в случае отсутствия файлов, соответствующих шаблону. И последнее, команда filter-branch переписывает историю начиная с коммита 6df7640, потому что мы знаем, что именно с этого коммита появилась проблема. По умолчанию перезапись начинается с самого первого коммита, что потребовало бы гораздо больше времени.

Теперь наша история не содержит ссылок на данный файл. Однако, в reflog и в новом наборе ссылок, добавленном Git'ом в .git/refs/originalпосле выполнения filter- branch, ссылки на него всё ещё присутствуют. Поэтому необходимо их удалить, а потом переупаковать базу. Необходимо избавиться от всех возможных ссылок на старые коммиты перед переупаковкой:

```
$ rm -Rf .git/refs/original

$ rm -Rf .git/logs/

$ git gc

Counting objects: 19, done.

   Delta compression using 2 threads.
Compressing objects: 100% (14/14), done.
  Writing objects: 100% (19/19), done.

Total 19 (delta 3), reused 16 (delta 1)
```

```
$ git count-objects -v
        count: 8
                                        size: 2040
                                      in-pack: 19
                                       packs: 1
                                     size-pack: 7
                                   prune-packable: 0
                                      garbage: 0
```

Размер упакованного репозитория сократился до 7 КБ, что намного лучше, чем 2 МБ. Из значения поля size видно, что большой объект всё ещё хранится в одном из ваших «рыхлых» объектов, но, что самое важное, при любой последующей отправке данных наружу и в том числе при клонировании он передаваться не будет. Если очень хочется, можно удалить его навсегда локально, выполнив git prune --expire.

Итоги

Теперь вы довольно хорошо понимаете, что Git делает в фоне и, в некоторой степени, как он написан. В данной главе мы рассмотрели несколько служебных команд — простых команд работающих на более низком уровень, чем обычные пользовательские команды описанные в остальной части книги. Понимание принципов работы Git на низком уровне упрощает понимание работы Git в целом и даёт возможность написания собственных утилит и сценариев для организации специфического процесса работы с Git.

Git как контентно-адресуемая файловая система, это очень мощный инструмент, который можно использовать не только как систему управления версиями. Надеюсь, полученное знание внутренней реализации Git поможет вам в написании ваших собственных интересных приложений использующих данные технологии и сделает вашу работу с Git более продвинутой и комфортной.