10 Git изнутри

Вы могли бы сразу перейти к этой главе, пропустив остальной материал, а может быть, вы добрались сюда в процессе вдумчивого чтения — как бы то ни было, именно тут мы рассмотрим внутреннюю работу и особенности реализации системы Git. Оказалось, что изучение именно этой информации позволяет понять, насколько полезным и мощным инструментом является Git, хотя некоторые считают, что попытка углубиться во все эти детали неоправданно сложна для новичков и только все запутывает. Поэтому эта глава и оказалась в самом конце книги, и вы можете самостоятельно выбрать, на каком этапе процесса обучения она вам потребуется.

Впрочем, приступим к делу. Сначала напомним, что Git — это в своей основе контентно-адресуемая файловая система, на которую наложен пользовательский интерфейс от VCS. Что это означает вы узнаете чуть позже.

В ранние годы своего существования (в основном до версии 1.5) пользовательский интерфейс системы Git был намного сложнее, так как был призван подчеркивать, что перед нами файловая система, а не законченная VCS. За последние годы интерфейс значительно улучшился и по удобству не уступает другим системам; но зачастую до сих пор можно столкнуться со стереотипным мнением, что пользовательский интерфейс системы Git запутан и труден в освоении.

Слой контентно-адресуемой файловой системы — это именно та замечательная вещь, которую мы рассмотрим в первую очередь; затем вы познакомитесь с транспортными механизмами и обслуживанием репозитория, с чем вам, возможно, придется сталкиваться в процессе использования Git.

Канализация и фарфор

В этой книге описывается, как работать с Git при помощи трех десятков команд, таких как `checkout`, `branch`, `remote` и т. п. Но так как система Git изначально представляла собой не VCS с удобным интерфейсом, а скорее инструментарий для создания VCS, Git предлагает множество команд для низкоуровневой работы, а также поддерживает использование конвейера в стиле UNIX и вызов из сценариев. Эти служебные команды иногда сравнивают с водопроводно-канализационными трубами (plumbing), а более удобные для пользователей команды — с фарфоровыми унитазами и раковинами (porcelain), облегчающими нам доступ к канализации.

В первых девяти главах мы рассматривали преимущественно «фарфоровые» команды. Теперь же пришло время более подробно познакомиться с низкоуровневыми командами, которые предоставляют контроль над внутренними процессами системы Git, помогая продемонстрировать, как функционирует Git и почему это происходит так, а не иначе. Многие из этих команд не предназначены для непосредственного вызова из командной строки, а являются строительными кирпичиками для новых инструментов и пользовательских сценариев.

При выполнении команды `git init` из новой или существующей папки система Git создает папку `.git`, в которой в дальнейшем сохраняются почти все используемые данные. При резервировании или клонировании репозитория достаточно скопировать эту папку, чтобы получить практически все необходимое. В этой главе мы в основном будем работать с содержимым этой папки. Вот как она выглядит:

```
$ ls -F1
HEAD
config*
description
hooks/
info/
objects/
refs/
```

Там могут находиться и другие файлы, но в данном случае перечислено только то, что по умолчанию оказывается в ней после выполнения команды `git init`. Файл `description` используется только программой GitWeb, поэтому на него можно не обращать внимания. Файл `config` содержит конфигурационные параметры, связанные с вашим проектом, а в папке `info` хранится глобальный файл исключений с игнорируемыми шаблонами, которые вы не хотите помещать в файл `.gitignore`. Папка `hooks` содержит сценарии хуков, работающих как на стороне клиента, так и на стороне сервера. Хуки подробно обсуждались в разделе «Git-хуки» главы 8.

Осталось четыре важных элемента: файл `HEAD`, еще не созданный файл `index` и папки `objects` и `refs`. Это ключевые элементы системы Git. В папке `objects` находится все содержимое вашей базы данных, папка `refs` хранит указатели на объекты-коммиты в этой базе (ветки), файл `HEAD` указывает, в какой ветке вы сейчас находитесь, а в файле `index` система Git хранит информацию из области индексирования. Далее мы детально рассмотрим все эти элементы, чтобы понять, как функционирует Git.

Объекты в Git

Система Git представляет собой контентно-адресуемую файловую систему. Но что это означает на практике? Это означает, что в своей основе Git является хранилищем данных вида «ключ-значение». В нее можно добавить любое содержимое, а в ответ вы получите ключ, по которому это содержимое позднее можно будет извлечь, когда оно вам понадобится. Для демонстрации этого принципа воспользуемся служебной командой `hash-object`, которая сохраняет данные в папке `.git` и возвращает ключ. Первым делом инициализируем новый репозиторий Git и удостоверимся, что в папке `objects` ничего нет:

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

Система Git инициализировала папку `objects`, создав внутри нее папки `pack` и `info`, но обычные файлы в ней пока отсутствуют. Добавим в базу данных системы Git какой-нибудь текст:

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

Параметр `-w` заставляет команду `hash-object` сохранить объект; без нее команда просто возвращает вам ключ. Параметр `--stdin` заставляет команду читать содержимое из стандартного потока ввода; без этого параметра команда `hash-object` будет искать путь к файлу с содержимым. Выводимые командой данные представляют собой контрольную хеш-сумму, состоящую из 40 символов. Это хеш SHA-1 — контрольная сумма сохраняемого вами содержимого плюс заголовок, о котором мы поговорим чуть позже. Теперь можно посмотреть, в каком виде Git сохраняет ваши данные:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

В папке `objects` появился файл. Именно так Git изначально сохраняет содержимое — один файл на единицу хранения с именем, которое представляет собой контрольную сумму SHA-1 содержимого и его заголовок. Подпапка именуется первыми двумя символами SHA, а имя файла формируют остальные 38 символов.

Извлечь содержимое обратно позволяет команда `cat-file`. Ее можно сравнить со швейцарским армейским ножом для вскрытия Git-объектов. Параметр `-p` заставляет эту команду определить тип содержимого и корректно его воспроизвести:

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

Теперь вы можете добавлять данные в систему Git и извлекать их оттуда. Эти операции доступны и для содержимого файлов. Рассмотрим пример управления версиями файла. Первым делом создадим новый файл и сохраним его в базе данных:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Запишем в этот файл какие-то новые данные и снова его сохраним:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Теперь в базе данных есть две новые версии этого файла, а также его исходное содержимое:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Мы можем вернуть файл к его первой версии:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

А можем вернуть его ко второй версии:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Однако запомнить ключ SHA-1 для каждой версии файла нереально; кроме того, в системе сохраняется не имя файла, а только его содержимое. Объекты такого типа называют массивами двоичных данных, или большими бинарными объектами (Binary Large Object, blob). Можно попросить систему Git по контрольной сумме SHA-1 определить тип любого объекта. Для этого используется команда `cat-file -t`:

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

Объекты-деревья

Рассмотрим другой тип объекта — дерево (tree). Объекты этого типа решают проблему хранения имен файлов, а также позволяют хранить группы файлов. Способ хранения содержимого в Git во многом такой же, как и в файловой системе UNIX, но несколько упрощенный. Вся информация хранится в виде объектов-деревьев

и blob-объектов, причем первые соответствуют записям UNIX-каталога, а вторые больше напоминают индексные дескрипторы, или содержимое файлов. Объект-дерево может содержать одну или несколько записей, каждая из которых включает в себя указатель SHA-1 на двоичный массив данных или на поддерево со связанными с ним правами доступа, типом и именем файла. Например, дерево последнего коммита в проекте может выглядеть так:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859 README
100644 blob 8f94139338f9404f26296befa88755fc2598c289 Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0 lib
```

Синтаксис `master^{tree}` определяет объект-дерево, на который указывает последний коммит в ветке `master`. Обратите внимание, что подпапка `lib` представляет собой не двоичный массив данных, а указатель на другое дерево:

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

По существу, данные, которые сохраняет Git, выглядят так, как показано на рис. 10.1.

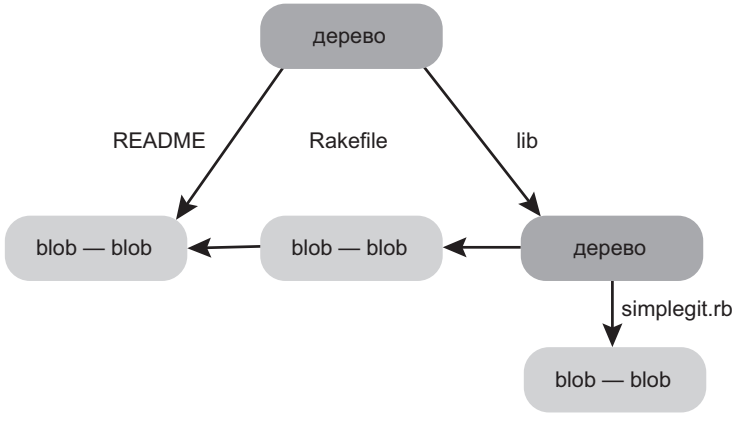

Рис. 10.1. Упрощенная версия модели данных в Git

Вы можете легко создать собственное дерево. Обычно Git берет состояние области индексирования и записывает оттуда набор объектов-деревьев. Поэтому для создания дерева первым делом нужно проиндексировать какие-либо файлы. Чтобы получить индекс с одной записью — первой версией файла `text.txt`, — воспользуемся служебной командой `update-index`. Она позволит нам принудительно поместить более раннюю версию файла `test.txt` в новую область индексации. Следует добавить к команде параметр `--add`, так как файл еще не индексирован (более того, у нас пока отсутствует настроенная область индексирования), а также параметр `--cacheinfo`, потому что добавляемый файл находится не в папке, а в базе данных. Затем мы указываем права доступа, контрольную сумму SHA-1 и имя файла:

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

В данном случае заданные права доступа 100644 означают обычный файл. Вариант 100755 соответствует исполняемому файлу, а 120000 — символической ссылке. Здесь мы имеем дело с аналогом прав доступа в операционной системе UNIX, но намного менее гибким — упомянутые три варианта являются единственными доступными для файлов (массивов двоичных данных) в Git (хотя для папок и подмодулей допустимы и другие варианты прав доступа).

Теперь с помощью команды `write-tree` добавим область индексирования к нашему объекту-дереву. Параметр `-w` в данном случае не требуется — вызов команды `write-tree` автоматически создает объект-дерево из состояния области индексирования, если такого дерева пока не существует:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Удостоверимся в том, что мы действительно получили дерево:

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

Создадим новое дерево со второй версией файла `test.txt` и новым файлом:

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

В области индексирования появилась новая версия файла `test.txt`, а также новый файл `new.txt`. Запишем это дерево (сохранив состояние области индексирования в объекте-дереве) и посмотрим, что из этого получилось:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
```

Обратите внимание, что наше дерево содержит записи для обоих файлов, а контрольная сумма SHA файла `test.txt` представляет собой SHA «версии 2», которая появлялась раньше (`1f7a7a`). Просто из интереса добавим первое дерево как подпапку для текущего. Чтение деревьев в область индексирования осуществляется командой `read-tree`. Так как в данном случае нам нужно прочитать существующее дерево как поддерево, потребуется параметр `--prefix`:

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
```

```
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579 bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
```

Если создать рабочую папку из только что записанного дерева, мы получим два файла в корне и подпапку `bak` с первой версией файла `test.txt`. Данные, которые система Git содержит для таких структур, можно представить так, как показано на рис. 10.2.

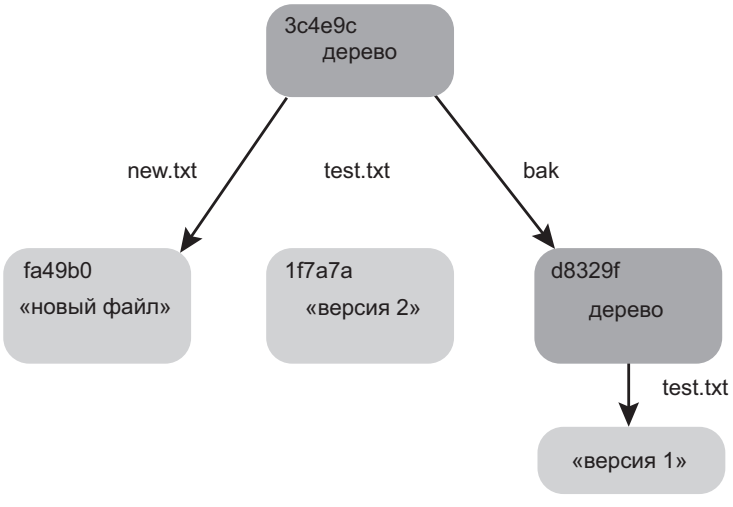

Рис. 10.2. Структура данных для текущего дерева в Git

Объекты-коммиты

У нас есть три дерева, соответствующие трем состояниям проекта, которые мы хотим отслеживать. Но это не решило стоящую перед нами проблему. Нам по-прежнему приходится запоминать все три значения SHA-1, чтобы иметь возможность доступа к снимкам состояний. К тому же у нас нет информации о том, кто сохранил эти состояния, когда это произошло и почему они были сохранены. Это именно та базовая информация, которую сохраняет для вас объект-коммит.

Создать объект-коммит позволяет команда `commit-tree`, которой мы сообщаем контрольную сумму SHA-1 нужного дерева и непосредственно предшествующие этому дереву объекты-коммиты, если таковые существуют. Начнем с первого записанного нами дерева:

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

Теперь посмотрим на полученный объект с помощью команды `cat-file`:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
```

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

first commit
```

Формат объекта-коммита прост: указываются дерево верхнего уровня для состояния проекта на определенный момент времени; сведения об авторе/создателе коммита (они берутся из конфигурационных настроек `user.name` и `user.email`) и временная метка; а также пустая строка, за которой следует сообщение фиксации.

Создадим еще два объекта-коммита, каждый из которых будет ссылаться на предыдущий коммит:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Каждый из этих трех объектов-коммитов указывает на одно из трех созданных нами деревьев для снимков состояний. Как ни странно это звучит, но теперь у нас есть полноценная Git-история, которую можно увидеть с помощью команды `git log`, указав хеш SHA-1 последнего коммита:

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:15:24 2009 -0700

    third commit

 bak/test.txt | 1 +
 1 file changed, 1 insertion(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:14:29 2009 -0700

    second commit

 new.txt  | 1 +
 test.txt | 2 +-
 2 files changed, 2 insertions(+), 1 deletion(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:09:34 2009 -0700

    first commit

 test.txt | 1 +
 1 file changed, 1 insertion(+)
```

Поразительно! Мы только что создали Git-историю с помощью исключительно низкоуровневых операций, не прибегая к командам пользовательского уровня. По сути, именно это проделывает система Git в ответ на команды `git add` и `git commit` — сохраняет двоичные массивы данных для измененных файлов, обновляет область индексирования, записывает деревья, объекты-коммиты, ссылающиеся на деревья верхнего уровня, а также коммиты, которые им предшествуют. Эти три основных вида Git-объектов — массивы двоичных данных, деревья и коммиты — изначально хранятся как отдельные файлы в папке `.git/objects`. Вот перечень объектов из рассматриваемой в данном примере папки с комментариями о том, что хранится внутри:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Проследив за внутренними указателями, получаем соответствующий граф объектов (рис. 10.3).

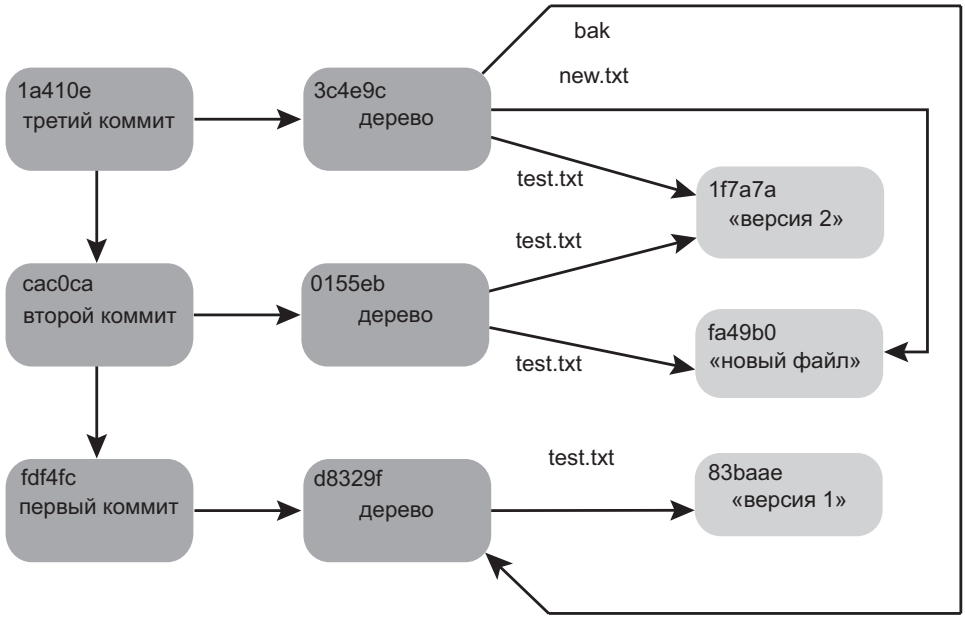

Рис. 10.3. Все объекты в вашей папке Git