CHAPTER 9

Git Internals

You may have skipped to this chapter from a previous chapter, or you may have gotten here after reading the rest of the book—in either case, this is where you'll go over the inner workings and implementation of Git. I found that learning this information was fundamentally important to understanding how useful and powerful Git is, but others have argued to me that it can be confusing and unnecessarily complex for beginners. Thus, I've made this discussion the last chapter in the book so you could read it early or later in your learning process. I leave it up to you to decide.

Now that you're here, let's get started. First, if it isn't yet clear, Git is fundamentally a content-addressable filesystem with a VCS user interface written on top of it. You'll learn more about what this means in a bit.

In the early days of Git (mostly pre 1.5), the user interface was much more complex because it emphasized this filesystem rather than a polished VCS. In the last few years, the UI has been refined until it's as clean and easy to use as any system out there; but often, the stereotype lingers about the early Git UI that was complex and difficult to learn.

The content-addressable filesystem layer is amazingly cool, so I'll cover that first in this chapter; then, you'll learn about the transport mechanisms and the repository maintenance tasks that you may eventually have to deal with.

Plumbing and Porcelain

This book covers how to use Git with 30 or so verbs such as `checkout`, `branch`, `remote`, and so on. But because Git was initially a toolkit for a VCS rather than a full user-friendly VCS, it has a bunch of verbs that do low-level work and were designed to be chained together UNIX style or called from scripts. These commands are generally referred to as *plumbing commands*, and the more user-friendly commands are called *porcelain commands*.

The book's first eight chapters deal almost exclusively with porcelain commands. But in this chapter, you'll be dealing mostly with the lower-level plumbing commands, because they give you access to the inner workings of Git and help demonstrate how and why Git does what it does. These commands aren't meant to be used manually on the command line, but rather to be used as building blocks for new tools and custom scripts.

When you run `git init` in a new or existing directory, Git creates the `.git` directory, which is where almost everything that Git stores and manipulates is located. If you want to back up or clone your repository, copying this single directory elsewhere gives you nearly

everything you need. This entire chapter basically deals with the stuff in this directory. Here's what it looks like:

```
$ ls
HEAD
branches/
config
description
hooks/
index
info/
objects/
refs/
```

You may see some other files in there, but this is a fresh `git init` repository—it's what you see by default. The `branches` directory isn't used by newer Git versions, and the `description` file is only used by the GitWeb program, so don't worry about those. The `config` file contains your project-specific configuration options, and the `info` directory keeps a global `exclude` file for ignored patterns that you don't want to track in a `.gitignore` file. The `hooks` directory contains your client- or server-side hook scripts, which are discussed in detail in Chapter 6.

This leaves four important entries: the `HEAD` and `index` files and the `objects` and `refs` directories. These are the core parts of Git. The `objects` directory stores all the content for your database, the `refs` directory stores pointers into commit objects in that data (branches), the `HEAD` file points to the branch you currently have checked out, and the `index` file is where Git stores your staging area information. You'll now look at each of these sections in detail to see how Git operates.

Git Objects

Git is a content-addressable filesystem. Great. What does that mean?

It means that at the core of Git is a simple key-value data store. You can insert any kind of content into it, and it will give you back a key that you can use to retrieve the content again at any time. To demonstrate, you can use the plumbing command `hash-object`, which takes some data, stores it in your `.git` directory, and gives you back the key the data is stored as. First, you initialize a new Git repository and verify that there is nothing in the `objects` directory:

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

Git has initialized the `objects` directory and created `pack` and `info` subdirectories in it, but there are no regular files. Now, store some text in your Git database:

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

The `-w` tells `hash-object` to store the object; otherwise, the command simply tells you what the key would be. `--stdin` tells the command to read the content from stdin; if you don't specify this, `hash-object` expects the path to a file. The output from the command is a 40-character checksum hash. This is the SHA-1 hash—a checksum of the content you're storing plus a header, which you'll learn about in a bit. Now you can see how Git has stored your data:

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

You can see a file in the `objects` directory. This is how Git stores the content initially—as a single file per piece of content, named with the SHA-1 checksum of the content and its header. The subdirectory is named with the first 2 characters of the SHA, and the filename is the remaining 38 characters.

You can pull the content back out of Git with the `cat-file` command. This command is sort of a Swiss army knife for inspecting Git objects. Passing `-p` to it instructs the `cat-file` command to figure out the type of content and display it nicely for you:

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

Now, you can add content to Git and pull it back out again. You can also do this with content in files. For example, you can do some simple version control on a file. First, create a new file and save its contents in your database:

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

Then, write some new content to the file, and save it again:

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

Your database contains the two new versions of the file as well as the first content you stored there:

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

Now you can revert the file back to the first version:

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

or the second version:

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

But remembering the SHA-1 key for each version of your file isn't practical; plus, you aren't storing the filename in your system—just the content. This object type is called a *blob*. You can have Git tell you the object type of any object in Git, given its SHA-1 key, with `cat-file -t`:

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

Tree Objects

The next type you'll look at is the tree object, which solves the problem of storing the filename and also allows you to store a group of files together. Git stores content in a manner similar to a UNIX filesystem, but a bit simplified. All the content is stored as tree and blob objects, with trees corresponding to UNIX directory entries and blobs corresponding more or less to inodes or file contents. A single tree object contains one or more tree entries, each of which contains an SHA-1 pointer to a blob or subtree with its associated mode, type, and filename. For example, the most recent tree in the simplegit project may look something like this:

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

The `master^{tree}` syntax specifies the tree object that is pointed to by the last commit on your `master` branch. Notice that the `lib` subdirectory isn't a blob but a pointer to another tree:

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

Conceptually, the data that Git is storing is something like Figure 9-1.

You can create your own tree. Git normally creates a tree by taking the state of your staging area or index and writing a tree object from it. So, to create a tree object, you first have to set up an index by staging some files. To create an index with a single entry—the first version of your `text.txt` file—you can use the plumbing command `update-index`. You use this command to artificially add the earlier version of the `test.txt` file to a new staging area. You must pass it the `--add` option because the file doesn't yet exist in your staging area (you don't even have a staging area set up yet) and `--cacheinfo` because the file you're adding isn't in your directory but is in your database. Then, you specify the mode, SHA-1, and filename:

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

In this case, you're specifying a mode of 100644, which means it's a normal file. Other options are 100755, which means it's an executable file; and 120000, which specifies a symbolic link. The mode is taken from normal UNIX modes but is much less flexible—these three

modes are the only ones that are valid for files in Git (although other modes are used for directories and submodules).

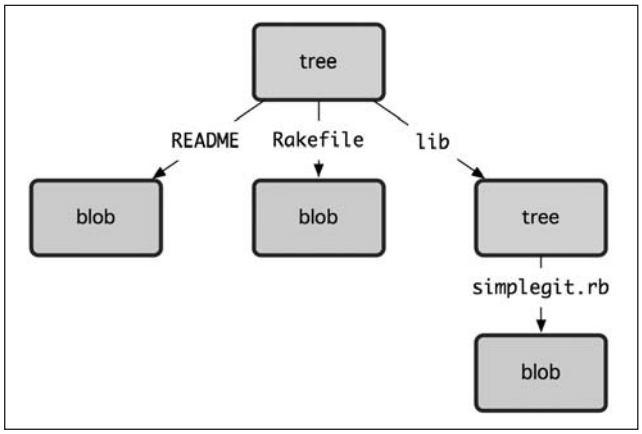

Figure 9-1. *Simple version of the Git data model*

Now, you can use the `write-tree` command to write the staging area out to a tree object. No `-w` option is needed—calling `write-tree` automatically creates a tree object from the state of the index if that tree doesn't yet exist:

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

You can also verify that this is a tree object:

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

You'll now create a new tree with the second version of `test.txt` and a new file as well:

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

Your staging area now has the new version of `test.txt` as well as the new file `new.txt`. Write out that tree (recording the state of the staging area or index to a tree object) and see what it looks like:

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

Notice that this tree has both file entries and also that the `test.txt` SHA is the version 2 SHA from earlier (1f7a7a). Just for fun, you'll add the first tree as a subdirectory into this one. You can read trees into your staging area by calling `read-tree`. In this case, you can read an existing tree into your staging area as a subtree by using the `--prefix` option to `read-tree`:

```
$ 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
```

If you created a working directory from the new tree you just wrote, you would get the two files in the top level of the working directory and a subdirectory named `bak` that contained the first version of the `test.txt` file. You can think of the data that Git contains for these structures as being like Figure 9-2.

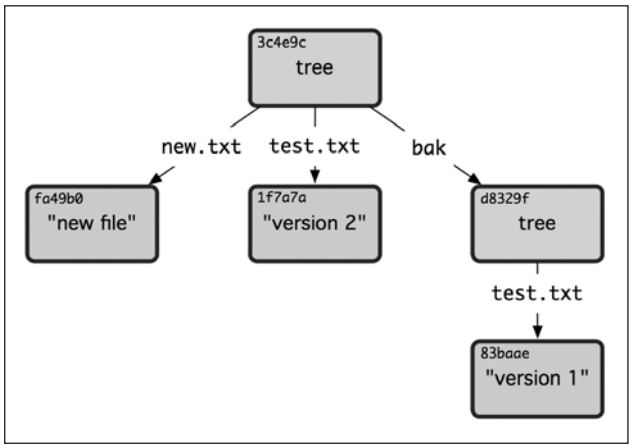

Figure 9-2. *The content structure of your current Git data*

Commit Objects

You have three trees that specify the different snapshots of your project that you want to track, but the earlier problem remains: you must remember all three SHA-1 values in order to recall the snapshots. You also don't have any information about who saved the snapshots, when they were saved, or why they were saved. This is the basic information that the commit object stores for you.

To create a commit object, you call `commit-tree` and specify a single tree SHA-1 and which commit objects, if any, directly preceded it. Start with the first tree you wrote:

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

Now you can look at your new commit object with `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
```

The format for a commit object is simple: it specifies the top-level tree for the snapshot of the project at that point; the author/committer information pulled from your `user.name` and `user.email` configuration settings, with the current timestamp; a blank line, and then the commit message.

Next, you'll write the other two commit objects, each referencing the commit that came directly before it:

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

Each of the three commit objects points to one of the three snapshot trees you created. Oddly enough, you have a real Git history now that you can view with the `git log` command, if you run it on the last commit SHA-1:

```
$ 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

 test.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Amazing. You've just done the low-level operations to build up a Git history without using any of the front ends. This is essentially what Git does when you run the `git add` and `git commit` commands—it stores blobs for the files that have changed, updates the index, writes out trees, and writes commit objects that reference the top-level trees and the commits that came immediately before them. These three main Git objects—the blob, the tree, and the commit—are initially stored as separate files in your `.git/objects` directory. Here are all the objects in the example directory now, commented with what they store:

```
$ 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
```

If you follow all the internal pointers, you get an object graph something like Figure 9-3.

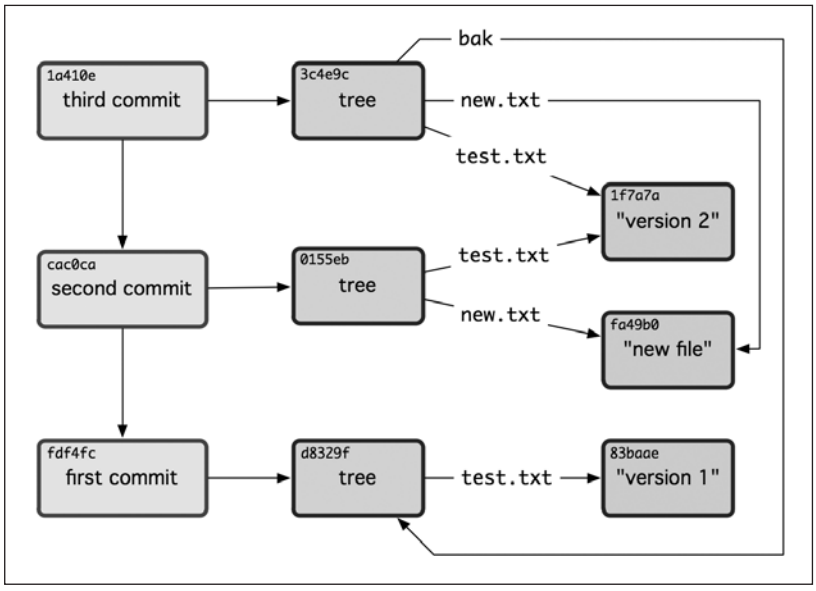

Figure 9-3. *All the objects in your Git directory*

Object Storage

I mentioned earlier that a header is stored with the content. Let's take a minute to look at how Git stores its objects. You'll see how to store a blob object—in this case, the string "what is up, doc?"—interactively in the Ruby scripting language. You can start up interactive Ruby mode with the `irb` command:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git constructs a header that starts with the type of the object, in this case a blob. Then, it adds a space followed by the size of the content and finally a null byte:

```
>> header = "blob #{content.length}\0"
=> "blob 16\000"
```

Git concatenates the header and the original content and then calculates the SHA-1 checksum of that new content. You can calculate the SHA-1 value of a string in Ruby by including the `SHA1` digest library with the `require` command and then calling `Digest::SHA1.hexdigest()` with the string:

```
>> store = header + content
=> "blob 16\000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git compresses the new content with zlib, which you can do in Ruby with the `zlib` library. First, you need to `require` the library and then run `Zlib::Deflate.deflate()` on the content:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\234K\312\311OR04c(\317H,Q\310,V(-\320QH\311O\266\a\000_\034\a\235"
```

Finally, you'll write your zlib-deflated content to an object on disk. You'll determine the path of the object you want to write out (the first two characters of the SHA-1 value being the subdirectory name, and the last 38 characters being the filename within that directory). In Ruby, you can use the `FileUtils.mkdir_p()` function to create the subdirectory if it doesn't exist. Then, open the file with `File.open()` and write out the previously zlib-compressed content to the file with a `write()` call on the resulting file handle:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

That's it—you've created a valid Git blob object. All Git objects are stored the same way, just with different types—instead of the string `blob`, the header will begin with `commit` or `tree`. Also, although the blob content can be nearly anything, the commit and tree content are very specifically formatted.

Git References

You can run something like `git log 1a410e` to look through your whole history, but you still have to remember that `1a410e` is the last commit in order to walk that history to find all those objects. You need a file in which you can store the SHA-1 value under a simple name so you can use that pointer rather than the raw SHA-1 value.

In Git, these are called *references* or *refs*; you can find the files that contain the SHA-1 values in the `.git/refs` directory. In the current project, this directory contains no files, but it does contain a simple structure:

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

To create a new reference that will help you remember where your latest commit is, you can technically do something as simple as this:

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

Now, you can use the head reference you just created instead of the SHA-1 value in your Git commands:

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

You aren't encouraged to directly edit the reference files. Git provides a safer command to do this if you want to update a reference called `update-ref`:

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

That's basically what a branch in Git is: a simple pointer or reference to the head of a line of work. To create a branch back at the second commit, you can do this:

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

Your branch will contain only work from that commit down:

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

Now, your Git database conceptually looks something like Figure 9-4.

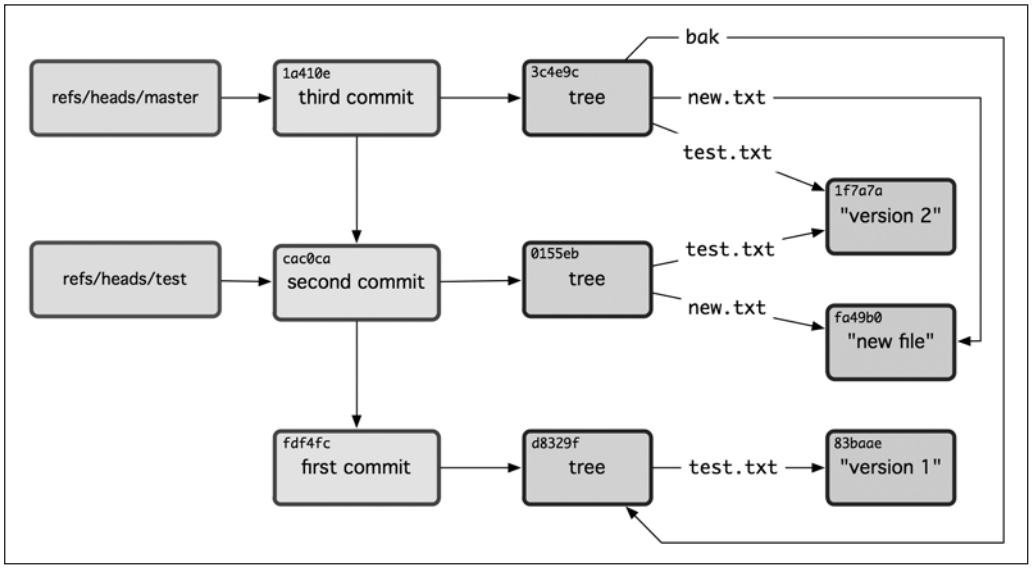

Figure 9-4. *Git directory objects with branch head references included*

When you run commands like `git branch (branchname)`, Git basically runs that `update-ref` command to add the SHA-1 of the last commit of the branch you're on into whatever new reference you want to create.

The HEAD

The question now is, when you run `git branch (branchname)`, how does Git know the SHA-1 of the last commit? The answer is the `HEAD` file. The `HEAD` file is a symbolic reference to the branch you're currently on. By *symbolic reference*, I mean that unlike a normal reference, it doesn't generally contain a SHA-1 value but rather a pointer to another reference. If you look at the file, you'll normally see something like this:

```
$ cat .git/HEAD
ref: refs/heads/master
```

If you run `git checkout test`, Git updates the file to look like this:

```
$ cat .git/HEAD
ref: refs/heads/test
```

When you run `git commit`, it creates the commit object, specifying the parent of that commit object to be whatever SHA-1 value the reference in `HEAD` points to.

You can also manually edit this file, but again a safer command exists to do so: `symbolic-ref`. You can read the value of your `HEAD` via this command:

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

You can also set the value of HEAD:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

You can't set a symbolic reference outside of the refs style:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

Tags

You've just gone over Git's three main object types, but there is a fourth. The tag object is very much like a commit object—it contains a tagger, a date, a message, and a pointer. The main difference is that a tag object points to a commit rather than a tree. It's like a branch reference, but it never moves—it always points to the same commit but gives it a friendlier name.

As discussed in Chapter 2, there are two types of tags: annotated and lightweight. You can make a lightweight tag by running something like this:

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

That is all a lightweight tag is—a branch that never moves. An annotated tag is more complex, however. If you create an annotated tag, Git creates a tag object and then writes a reference to point to it rather than directly to the commit. You can see this by creating an annotated tag (-a specifies that it's an annotated tag):

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

Here's the object SHA-1 value it created:

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

Now, run the cat-file command on that SHA-1 value:

```
$ 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
```

Notice that the object entry points to the commit SHA-1 value that you tagged. Also notice that it doesn't need to point to a commit; you can tag any Git object. In the Git source code, for example, the maintainer has added their GPG public key as a blob object and then tagged it. You can view the public key by running

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

in the Git source code. The Linux kernel also has a non-commit-pointing tag object—the first tag created points to the initial tree of the import of the source code.

Remotes

The third type of reference that you'll see is a remote reference. If you add a remote and push to it, Git stores the value you last pushed to that remote for each branch in the `refs/remotes` directory. For instance, you can add a remote called `origin` and push your `master` branch to it:

```
$ 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
```

Then, you can see what the `master` branch on the `origin` remote was the last time you communicated with the server, by checking the `refs/remotes/origin/master` file:

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

Remote references differ from branches (`refs/heads` references) mainly in that they can't be checked out. Git moves them around as bookmarks to the last known state of where those branches were on those servers.

Packfiles

Let's go back to the `objects` database for your test Git repository. At this point, you have 11 objects—4 blobs, 3 trees, 3 commits, and 1 tag:

```
$ 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/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git compresses the contents of these files with zlib, and you're not storing much, so all these files collectively take up only 925 bytes. You'll add some larger content to the repository to demonstrate an interesting feature of Git. Add the `repo.rb` file from the Grit library you worked with earlier—this is about a 12K source code file:

```
$ 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%)
```

If you look at the resulting tree, you can see the SHA-1 value your `repo.rb` file got for the blob object:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

You can then use `git cat-file` to see how big that object is:

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

Now, modify that file a little, and see what happens:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master ab1afef] modified repo a bit
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Check the tree created by that commit, and you see something interesting:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 05408d195263d853f09dca71d55116663690c27c      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

The blob is now a different blob, which means that although you added only a single line to the end of a 400-line file, Git stored that new content as a completely new object:

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

You have two nearly identical 12K objects on your disk. Wouldn't it be nice if Git could store one of them in full but then store the second object only as the delta between it and the first?

It turns out that it can. The initial format in which Git saves objects on disk is called a *loose object format*. However, occasionally Git packs up several of these objects into a single binary

file called a *packfile* in order to save space and be more efficient. Git does this if you have too many loose objects around, if you run the `git gc` command manually, or if you push to a remote server. To see what happens, you can manually ask Git to pack up the objects by calling the `git gc` command:

```
$ 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)
```

If you look in your `objects` directory, you'll find that most of your objects are gone, and a new pair of files has appeared:

```
$ 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
```

The objects that remain are the blobs that aren't pointed to by any commit—in this case, the "what is up, doc?" example and the "test content" example blobs you created earlier. Because you never added them to any commits, they're considered dangling and aren't packed up in your new packfile.

The other files are your new packfile and an index. The packfile is a single file containing the contents of all the objects that were removed from your filesystem. The index is a file that contains offsets into that packfile so you can quickly seek to a specific object. What is cool is that although the objects on disk before you ran the `gc` were collectively about 12K in size, the new packfile is only 6K. You've halved your disk usage by packing your objects.

How does Git do this? When Git packs objects, it looks for files that are named and sized similarly, and stores just the deltas from one version of the file to the next. You can look into the packfile and see what Git did to save space. The `git verify-pack` plumbing command allows you to see what was packed up:

```
$ git verify-pack -v pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
0155eb4229851634a0f03eb265b69f5a2d56f341 tree    71 76 5400
05408d195263d853f09dca71d55116663690c27c blob    12908 3478 874
09f01cea547666f58d6a8d809583841a7c6f0130 tree    106 107 5086
1a410efbd13591db07496601ebc7a059dd55cfe9 commit 225 151 322
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob    10 19 5381
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree    101 105 5211
484a59275031909e19aadb7c92262719cfcdf19a commit 226 153 169
83baae61804e65cc73a7201a7252750c76066a30 blob    10 19 5362
9585191f37f7b0fb9444f35a9bf50de191beadc2 tag     136 127 5476
9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e blob    7 18 5193 1➥
05408d195263d853f09dca71d55116663690c27c
ab1afef80fac8e34258ff41fc1b867c702daa24b commit 232 157 12
cac0cab538b970a37ea1e769cbbde608743bc96d commit 226 154 473
```

```
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree   36 46 5316
e3f094f522629ae358806b17daf78246c27c007b blob   1486 734 4352
f8f51d7d8a1760462eca26eebafde32087499533 tree   106 107 749
fa49b077972391ad58037050f2a75f74e3671e92 blob   9 18 856
fdf4fc3344e67ab068f836878b6c4951e3b15f3d commit 177 122 627
chain length = 1: 1 object
pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack: ok
```

Here, the `9bc1d` blob, which if you remember was the first version of your `repo.rb` file, is referencing the `05408` blob, which was the second version of the file. The third column in the output is the size of the object in the pack, so you can see that `05408` takes up 12K of the file but that `9bc1d` only takes up 7 bytes. What is also interesting is that the second version of the file is the one that is stored intact, whereas the original version is stored as a delta—this is because you're most likely to need faster access to the most recent version of the file.

The really nice thing about this is that it can be repacked at any time. Git will occasionally repack your database automatically, always trying to save more space. You can also manually repack at any time by running `git gc` by hand.

The Refspec

Throughout this book, you've used simple mappings from remote branches to local references; but they can be more complex.

Suppose you add a remote like this:

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

It adds a section to your `.git/config` file, specifying the name of the remote (`origin`), the URL of the remote repository, and the refspec for fetching:

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

The format of the refspec is an optional +, followed by *<src>*:*<dst>*, where *<src>* is the pattern for references on the remote side and *<dst>* is where those references will be written locally. The + tells Git to update the reference even if it isn't a fast-forward.

In the default case that is automatically written by a `git remote add` command, Git fetches all the references under `refs/heads/` on the server and writes them to `refs/remotes/origin/` locally. So, if there is a `master` branch on the server, you can access the log of that branch locally via

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

They're all equivalent, because Git expands each of them to `refs/remotes/origin/master`.

If you want Git instead to pull down only the `master` branch each time, and not every other branch on the remote server, you can change the `fetch` line to

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

This is just the default refspec for `git fetch` for that remote. If you want to do something one time, you can specify the refspec on the command line, too. To pull the `master` branch on the remote down to `origin/mymaster` locally, you can run

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

You can also specify multiple refspecs. On the command line, you can pull down several branches like so:

```
$ 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)
 * [new branch]      topic      -> origin/topic
```

In this case, the `master` branch pull was rejected because it wasn't a fast-forward reference. You can override that by specifying the + in front of the refspec.

You can also specify multiple refspecs for fetching in your configuration file. If you want to always fetch the `master` and `experiment` branches, add two lines:

```
[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
```

You can't use partial globs in the pattern, so this would be invalid:

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

However, you can use namespacing to accomplish something like that. If you have a QA team that pushes a series of branches, and you want to get the `master` branch and any of the QA team's branches but nothing else, you can use a config section like this:

```
[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/*
```

If you have a complex workflow process that has a QA team pushing branches, developers pushing branches, and integration teams pushing and collaborating on remote branches, you can namespace them easily this way.

Pushing Refspecs

It's nice that you can fetch namespaced references that way, but how does the QA team get their branches into a `qa/` namespace in the first place? You accomplish that by using refspecs to push.

If the QA team wants to push their `master` branch to `qa/master` on the remote server, they can run

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

If they want Git to do that automatically each time they run `git push origin`, they can add a `push` value to their config file:

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

Again, this will cause a `git push origin` to push the local `master` branch to the remote `qa/master` branch by default.

Deleting References

You can also use the refspec to delete references from the remote server by running something like this:

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

Because the refspec is *<src>*:*<dst>*, by leaving off the *<src>* part, this basically says to make the topic branch on the remote nothing, which deletes it.

Transfer Protocols

Git can transfer data between two repositories in two major ways: over HTTP and via the so-called smart protocols used in the `file://`, `ssh://`, and `git://` transports. This section will quickly cover how these two main protocols operate.

The Dumb Protocol

Git transport over HTTP is often referred to as the *dumb protocol* because it requires no Git-specific code on the server side during the transport process. The fetch process is a series of `GET` requests, where the client can assume the layout of the Git repository on the server. Let's follow the `http-fetch` process for the `simplegit` library:

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

The first thing this command does is pull down the `info/refs` file. This file is written by the `update-server-info` command, which is why you need to enable that as a post-receive hook in order for the HTTP transport to work properly:

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

Now you have a list of the remote references and SHAs. Next, you look for what the `HEAD` reference is so you know what to check out when you're finished:

```
=> GET HEAD
ref: refs/heads/master
```

You need to check out the `master` branch when you've completed the process.

At this point, you're ready to start the walking process. Because your starting point is the ca82a6 commit object you saw in the `info/refs` file, you start by fetching that:

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

You get an object back—that object is in loose format on the server, and you fetched it over a static HTTP `GET` request. You can zlib-uncompress it, strip off the header, and look at the commit content:

```
$ 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
```

Next, you have two more objects to retrieve—`cfda3b`, which is the tree of content that the commit you just retrieved points to, and `085bb3`, which is the parent commit:

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

That gives you your next commit object. Grab the tree object:

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

Oops—it looks like that tree object isn't in loose format on the server, so you get a 404 response back. There are a couple of reasons for this—the object could be in an alternate repository, or it could be in a packfile in this repository. Git checks for any listed alternates first:

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

If this comes back with a list of alternate URLs, Git checks for loose files and packfiles there—this is a nice mechanism for projects that are forks of one another to share objects on disk. However, because no alternates are listed in this case, your object must be in a packfile. To see what packfiles are available on this server, you need to get the `objects/info/packs` file, which contains a listing of them (also generated by `update-server-info`):

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

There is only one packfile on the server, so your object is obviously in there, but you'll check the `index` file to make sure. This is also useful if you have multiple packfiles on the server, so you can see which packfile contains the object you need:

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

Now that you have the packfile index, you can see if your object is in it—because the index lists the SHAs of the objects contained in the packfile and the offsets to those objects. Your object is there, so go ahead and get the whole packfile:

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

You have your tree object, so you continue walking your commits. They're all also within the packfile you just downloaded, so you don't have to do any more requests to your server. Git checks out a working copy of the `master` branch that was pointed to by the `HEAD` reference you downloaded at the beginning.

The entire output of this process looks like this:

```
$ 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
walk a11bef06a3f659402fe7563abf99ad00de2209e6
```

The Smart Protocol

The HTTP method is simple but a bit inefficient. Using smart protocols is a more common method of transferring data. These protocols have a process on the remote end that is intelligent about Git—it can read local data and figure out what the client has or needs and generate custom data for it. There are two sets of processes for transferring data: a pair for uploading data and a pair for downloading data.

Uploading Data

To upload data to a remote process, Git uses the `send-pack` and `receive-pack` processes. The `send-pack` process runs on the client and connects to a `receive-pack` process on the remote side.

For example, say you run `git push origin master` in your project, and `origin` is defined as a URL that uses the SSH protocol. Git fires up the `send-pack` process, which initiates a connection over SSH to your server. It tries to run a command on the remote server via an SSH call that looks something like this:

```
$ 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
```

The `git-receive-pack` command immediately responds with one line for each reference it currently has—in this case, just the `master` branch and its SHA. The first line also has a list of the server's capabilities (here, `report-status` and `delete-refs`).

Each line starts with a 4-byte hex value specifying how long the rest of the line is. Your first line starts with 005b, which is 91 in hex, meaning that 91 bytes remain on that line. The next line starts with 003e, which is 62, so you read the remaining 62 bytes. The next line is 0000, meaning the server is done with its references listing.

Now that it knows the server's state, your `send-pack` process determines what commits it has that the server doesn't. For each reference that this push will update, the `send-pack` process tells the `receive-pack` process that information. For instance, if you're updating the `master` branch and adding an `experiment` branch, the `send-pack` response may look something like this:

```
0085ca82a6dff817ec66f44342007202690a93763949 \
  15027957951b64cf874c3557a0f3547bd83b3ff6 refs/heads/master report-status
006700000000000000000000000000000000000000 \
  cdfdb42577e2506715f8cfeacdbabc092bf63e8d refs/heads/experiment
0000
```

The SHA-1 value of all 0s means that nothing was there before—because you're adding the `experiment` reference. If you were deleting a reference, you would see the opposite: all 0s on the right side.

Git sends a line for each reference you're updating with the old SHA, the new SHA, and the reference that is being updated. The first line also has the client's capabilities. Next, the client uploads a packfile of all the objects the server doesn't have yet. Finally, the server responds with a success (or failure) indication:

```
000Aunpack ok
```

Downloading Data

When you download data, the `fetch-pack` and `upload-pack` processes are involved. The client initiates a `fetch-pack` process that connects to an `upload-pack` process on the remote side to negotiate what data will be transferred down.

There are different ways to initiate the `upload-pack` process on the remote repository. You can run via SSH in the same manner as the `receive-pack` process. You can also initiate the process via the Git daemon, which listens on a server on port 9418 by default. The `fetch-pack` process sends data that looks like this to the daemon after connecting:

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

It starts with the 4 bytes specifying how much data is following, then the command to run followed by a null byte, and then the server's hostname followed by a final null byte. The Git daemon checks that the command can be run and that the repository exists and has public permissions. If everything is cool, it fires up the `upload-pack` process and hands off the request to it.

If you're doing the fetch over SSH, `fetch-pack` instead runs something like this:

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

In either case, after `fetch-pack` connects, `upload-pack` sends back something like this:

```
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
```

This is very similar to what `receive-pack` responds with, but the capabilities are different. In addition, it sends back the `HEAD` reference so the client knows what to check out if this is a clone.

At this point, the `fetch-pack` process looks at what objects it has and responds with the objects that it needs by sending "want" and then the SHA it wants. It sends all the objects it already has with "have" and then the SHA. At the end of this list, it writes "done" to initiate the `upload-pack` process to begin sending the packfile of the data it needs:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

That is a very basic case of the transfer protocols. In more complex cases, the client supports `multi_ack` or `side-band` capabilities; but this example shows you the basic back and forth used by the smart protocol processes.

Maintenance and Data Recovery

Occasionally, you may have to do some cleanup—make a repository more compact, clean up an imported repository, or recover lost work. This section will cover some of these scenarios.

Maintenance

Occasionally, Git automatically runs a command called `auto gc`. Most of the time, this command does nothing. However, if there are too many loose objects (objects not in a packfile) or too many packfiles, Git launches a full-fledged `git gc` command. The `gc` stands for *garbage collect*, and the command does a number of things: it gathers up all the loose objects and places them in packfiles, it consolidates packfiles into one big packfile, and it removes objects that aren't reachable from any commit and are a few months old.

You can run `auto gc` manually as follows:

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

Again, this generally does nothing. You must have around 7,000 loose objects or more than 50 packfiles for Git to fire up a real `gc` command. You can modify these limits with the `gc.auto` and `gc.autopacklimit` config settings, respectively.

The other thing `gc` will do is pack up your references into a single file. Suppose your repository contains the following branches and tags:

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

```
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

If you run `git gc`, you'll no longer have these files in the `refs` directory. Git will move them for the sake of efficiency into a file named `.git/packed-refs` that looks like this:

```
$ 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
```

If you update a reference, Git doesn't edit this file but instead writes a new file to `refs/heads`. To get the appropriate SHA for a given reference, Git checks for that reference in the `refs` directory and then checks the `packed-refs` file as a fallback. However, if you can't find a reference in the `refs` directory, it's probably in your `packed-refs` file.

Notice the last line of the file, which begins with a `^`. This means the tag directly above is an annotated tag and that line is the commit that the annotated tag points to.

Data Recovery

At some point in your Git journey, you may accidentally lose a commit. Generally, this happens because you force-delete a branch that had work on it, and it turns out you wanted the branch after all; or you hard-reset a branch, thus abandoning commits that you wanted something from. Assuming this happens, how can you get your commits back?

Here's an example that hard-resets the `master` branch in your test repository to an older commit and then recovers the lost commits. First, let's review where your repository is at this point:

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

Now, move the `master` branch back to the middle commit:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You've effectively lost the top two commits—you have no branch from which those commits are reachable. You need to find the latest commit SHA and then add a branch that points to it. The trick is finding that latest commit SHA—it's not like you've memorized it, right?

Often, the quickest way is to use a tool called `git reflog`. As you're working, Git silently records what your `HEAD` is every time you change it. Each time you commit or change branches, the reflog is updated. The reflog is also updated by the `git update-ref` command, which is another reason to use it instead of just writing the SHA value to your ref files, as you learned in the "Git References" section of this chapter. You can see where you've been at any time by running `git reflog`:

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

Here you can see the two commits you checked out, but there isn't much information. To see the same information in a much more useful way, you can run `git log -g`, which provides normal `log` output for your 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
```

It looks like the bottom commit is the one you lost, so you can recover it by creating a new branch at that commit. For example, you can start a branch named `recover-branch` at that commit (`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
```

Cool—now you have a branch named `recover-branch` that is where your `master` branch used to be, making the first two commits reachable again.

Next, suppose your loss was for some reason not in the reflog—you can simulate that by removing `recover-branch` and deleting the reflog. Now the first two commits aren't reachable by anything:

```
$ git branch –D recover-branch
$ rm -Rf .git/logs/
```

Because the reflog data is kept in the `.git/logs/` directory, you effectively have no reflog. How can you recover that commit at this point? One way is to use the `git fsck` utility, which checks your database for integrity. If you run it with the `--full` option, it shows you all objects that aren't pointed to by another object:

```
$ git fsck --full
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

In this case, you can see your missing commit after the dangling commit. You can recover it the same way, by adding a branch that points to that SHA.

Removing Objects

There are a lot of great things about Git, but one feature that can cause issues is the fact that a Git clone downloads the entire history of the project, including every version of every file. This is fine if the whole thing is source code, because Git is highly optimized to compress that data efficiently. However, if someone at any point in the history of your project added a single huge file, every clone for all time will be forced to download that large file, even if it was removed from the project in the very next commit. Because it's reachable from the history, it will always be there.

This can be a huge problem when you're converting Subversion or Perforce repositories into Git. Because you don't download the whole history in those systems, this type of addition carries few consequences. If you did an import from another system or otherwise find that your repository is much larger than it should be, here is how you can find and remove large objects.

Be warned: this technique is destructive to your commit history. It rewrites every commit object downstream from the earliest tree you have to modify to remove a large file reference. If you do this immediately after an import, before anyone has started to base work on the commit, you're fine—otherwise, you have to notify all contributors that they must rebase their work onto your new commits.

To demonstrate, you'll add a large file into your test repository, remove it in the next commit, find it, and remove it permanently from the repository. First, add a large object to your history:

```
$ 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
```

Oops—you didn't want to add a huge tarball to your project. Better get rid of it:

```
$ 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
```

Now, `gc` your database and see how much space you're using:

```
$ 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)
```

You can run the `count-objects` command to quickly see how much space you're using:

```
$ git count-objects -v
count: 4
size: 16
in-pack: 21
packs: 1
size-pack: 2016
prune-packable: 0
garbage: 0
```

The `size-pack` entry is the size of your packfiles in kilobytes, so you're using 2MB. Before the last commit, you were using closer to 2K—clearly, removing the file from the previous commit didn't remove it from your history. Every time anyone clones this repository, they will have to clone all 2MB just to get this tiny project, because you accidentally added a big file. Let's get rid of it.

First you have to find it. In this case, you already know what file it is. But suppose you didn't; how would you identify what file or files were taking up so much space? If you run `git gc`, all the objects are in a packfile; you can identify the big objects by running another plumbing command called `git verify-pack` and sorting on the third field in the output, which is file size. You can also pipe it through the `tail` command because you're only interested in the last few largest files:

```
$ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -n | tail -3
e3f094f522629ae358806b17daf78246c27c007b blob   1486 734 4667
05408d195263d853f09dca71d55116663690c27c blob   12908 3478 1189
7a9eb2fba2b1811321254ac360970fc169ba2330 blob   2056716 2056872 5401
```

The big object is at the bottom: 2MB. To find out what file it is, you'll use the `rev-list` command, which you used briefly in Chapter 7. If you pass `--objects` to `rev-list`, it lists all the commit SHAs and also the blob SHAs with the file paths associated with them. You can use this to find your blob's name:

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

Now, you need to remove this file from all trees in your past. You can easily see what commits modified this file:

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

You must rewrite all the commits downstream from `6df76` to fully remove this file from your Git history. To do so, you use `filter-branch`, which you used in Chapter 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
```

The `--index-filter` option is similar to the `--tree-filter` option used in Chapter 6, except that instead of passing a command that modifies files checked out on disk, you're modifying your staging area or index each time. Rather than remove a specific file with something like `rm file`, you have to remove it with `git rm --cached`—you must remove it from the index, not from disk. The reason to do it this way is speed—because Git doesn't have to check out each revision to disk before running your filter, the process can be much, much faster. You can accomplish the same task with `--tree-filter` if you want. The `--ignore-unmatch` option to `git rm` tells it not to error out if the pattern you're trying to remove isn't there. Finally, you ask `filter-branch` to rewrite your history only from the `6df7640` commit up, because you know that is where this problem started. Otherwise, it will start from the beginning and will unnecessarily take longer.

Your history no longer contains a reference to that file. However, your reflog and a new set of refs that Git added when you did the `filter-branch` under `.git/refs/original` still do, so you have to remove them and then repack the database. You need to get rid of anything that has a pointer to those old commits before you repack:

```
$ 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)
```

Let's see how much space you saved:

```
$ git count-objects -v
count: 8
size: 2040
in-pack: 19
packs: 1
```

```
size-pack: 7
prune-packable: 0
garbage: 0
```

The packed repository size is down to 7K, which is much better than 2MB. You can see from the `size` value that the big object is still in your loose objects, so it's not gone; but it won't be transferred on a push or subsequent clone, which is what's important. If you really wanted to, you could remove the object completely by running `git prune --expire`.

Summary

You should have a pretty good understanding of what Git does in the background and, to some degree, how it's implemented. This chapter has covered a number of plumbing commands—commands that are lower level and simpler than the porcelain commands you've learned about in the rest of the book. Understanding how Git works at a lower level should make it easier to understand why it's doing what it's doing and also to write your own tools and helping scripts to make your specific workflow work for you.

Git as a content-addressable filesystem is a very powerful tool that you can easily use as more than just a VCS. I hope you can use your newfound knowledge of Git internals to implement your own cool application of this technology and feel more comfortable using Git in more advanced ways.