10.1 Git Internals - Plumbing and Porcelain

You may have skipped to this chapter from a much earlier chapter, or you may have gotten here after sequentially reading the entire book up to this point — in either case, this is where we'll go over the inner workings and implementation of Git. We found that understanding this information was fundamentally important to appreciating how useful and powerful Git is, but others have argued to us that it can be confusing and unnecessarily complex for beginners. Thus, we've made this discussion the last chapter in the book so you could read it early or later in your learning process. We 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; however, the stereotype lingers about the early Git UI that was complex and difficult to learn.

The content-addressable filesystem layer is amazingly cool, so we'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 primarily how to use Git with 30 or so subcommands such as `checkout`, `branch`, `remote`, and so on. But because Git was initially a toolkit for a version control system rather than a full user-friendly VCS, it has a number of subcommands 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 Git's "plumbing" commands, while the more user-friendly commands are called "porcelain" commands.

As you will have noticed by now, this book's first nine 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. Many of 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 what you can see in this directory. Here's what a newly-initialized `.git` directory typically looks like:

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

Depending on your version of Git, you may see some additional content there, but this is a fresh `git init`repository — it's what you see by default. The `description` file is used only by the GitWeb program, so don't worry about it. 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 Git Hooks.

This leaves four important entries: the `HEAD` and (yet to be created) `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, tags, remotes and more), 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.

prev | next