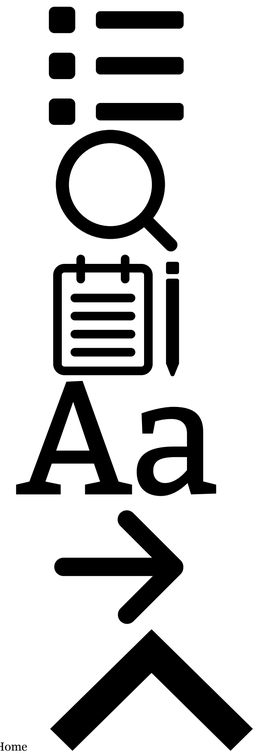

1

How Does Git Actually Work? Written by Jawwad Ahmad & Chris Belanger

Git is one of those wonderful, elegant tools that does an amazing job of abstracting the underlying mechanism from the front-end workings. To pull changes from the remote down to the local, you execute `git pull`. To commit your changes in your local repository, you execute `git commit`. To push commits from your local repository to the remote repository, you execute `git push`. The front end does an excellent job of mirroring the mental model of what's happening to your code.

But as you would expect, a lot is going on underneath. The nice thing about Git is that you could spend your entire career not knowing how the Git internals work, and you'd get along quite well. But being aware of how Git manages your repository will help cement that mental model and give a little more insight into why Git does what it does.

To follow along, you can start with any repository. If you don't have one handy, you can use one of the repos provided with the materials for this book.

Everything is a hash

Well, not *everything* is a hash, to be honest. But it's a useful point to start when you want to know how Git works.

Git refers to all commits by their SHA-1 hashes. You've seen that many times over, both in this book and in your personal and professional work with Git. The hash is the key that points to a particular commit in the repository, and it's pretty clear to see that it's just a type of unique ID. One ID references one commit. There's no ambiguity there.

But if you dig down a little bit, the commit hash doesn't reference *everything* that has to do with a commit. In fact, a lot of what Git does is create references to references in a tree-like structure to store and retrieve your data, and its metadata, as quickly and efficiently as possible.

To see this in action, you'll dissect the "secret" files underneath the **.git** directory and see what's inside of each.

Dissecting the commit

Since the atomic particle of Git workflow is the commit, it makes sense to start there. You'll start walking down the tree to see how Git stores and tracks your work.

Note: The commit hashes I'll use will be different than the ones in your repository. Simply follow the steps below, substituting in your hashes for the ones I have in my repository.

I'm going to pick one of my most recent commits that has a change that I made, as opposed to a merge, just to narrow down the set of changes I want to look at.

To get the list of the most recent five commits, navigate to the main directory of your repository and execute the `git log` command as below:

```
git log -5 --oneline
```

My log result looks like the following:

```
f8098fa (HEAD -> main, origin/main, origin/HEAD) Merge branch 'clickbait' wi...
d83ab2b (crispy8888/clickbait, clickbait) Ticked off the last item added
5415c13 More clickbait ideas
fed347d (from-crispy8888) Merge branch 'main' of https://www.github.com/bela...
ace7251 Adding debugging book idea
```

I'll select the commit with the short hash `d83ab2b` to start stepping through the tree structure. First, though, you'll need to get the long hash for this, instead of the short one. You'll see why this is in a moment.

You *could* simply run `git log` again without the `--oneline` option to get the long hash, but there's an easier way.

Converting short hash into long

Execute the command below to convert a short hash into its long equivalent, substituting your own short hash:

```
git rev-parse d83ab2b
```

Git responds with the long hash equivalent: `d83ab2b104e4addd03947ed3b1ca57b2e68dfc85`.

Now, you need to start crawling through the Git tree to find out what this commit *look*s like on disk.

The inner workings of Git

Next, navigate into the **.git** directory of your repository:

```
cd .git
```

Now, pull up a directory listing of what's in the **.git** directory, and have a look at the directories there. You should, at a minimum, see the following directories:

```
info/
objects/
hooks/
logs/
refs/
```

The directory you're interested in is the **objects** directory. In Git, the most common objects are:

Commits: Structures that hold metadata about your commit, as well as the pointers to the parent commit and the files underneath.

Trees: Tree structures of all the files contained in a commit.

Blobs: Compressed collections of files in the tree.

Next, navigate into the **objects** directory:

```
cd objects
```

Pull up a directory listing to see what's inside, and you'll be greeted with the following puzzling list of directories:

Note: You may not see subfolders other than **info** and **pack** if you are running this on a newly cloned repository. This is because in a newly cloned repository, the objects are already compressed, or "packed". You can also run `git gc` to manually compress these objects which will make the extra subfolders disappear.

```
02    14    39    55    6e    84    ad    c5    db    f8
05    19    3a    56    72    88    b4    c8    e0    f9
06    1a    3b    57    73    8b    b5    ca    e6    fb
0a    1c    3d    59    75    99    b8    ce    e7    fe
0b    24    3e    5d    76    9d    b9    cf    eb    ff
0c    29    43    5f    78    9f    ba    d2    ec    info
0d    2c    45    62    7a    a0    bb    d3    ed    pack
0e    33    47    65    7d    a1    be    d7    ee
0f    35    4e    67    7f    a4    bf    d8    f1
11    36    50    69    81    ab    c0    d9    f4
12    37    54    6c    83    ac    c4    da    f5
```

It's clear that this is a lookup system of some sort, but what does that two-character directory name mean?

The Git object repository structure

When Git stores objects, instead of dumping them all into a single directory, which would get unwieldy in rather short order, it structures them neatly into a tree. Git takes the first two characters of your object's hash, uses that as the directory name, and then uses the remaining 38 characters as the object identifier.
Here's an example of the Git **object** directory structure, from my repository, that shows this hierarchy:

```
objects
├── 02
│   ├── 1f10a861cb8a8b904aac751226c67e42fadbf5
│   └── 8f2d5e0a0f99902638039794149dfa0126bede
├── 05
│   └── 66b505b18787bbc710aeef2c8981b0e13810f9
├── 06
│   └── f468e662b25687de078df86cbc9b67654d938b
├── 0a
│   └── 795bccdec0f85ebd9411e176a90b1b4dfe2002
├── 0b
│   └── 2d0890591a57393dc40e2155bff8901acafbb6
├── 0c
│   └── 66fedfeb176b467885ccd1a1ec70849299eeac
├── 0d
│   └── dfac290832b19d1cf78284226179a596bf5825
├── 0e
│   └── 066e61ce93bf5dfaa9a6eba812aa62038d7875
├── 0f
│   └── a80ee6442e459c501c6da30bf99a07c0f5624e
├── 11
│   ├── 06774ed5ad653594a848631f1f2786a76a776f
│   ├── 92339da7c0831ba4448cb46d40e1b8c2bed12c
│   └── c1a7373df5a0fbea20fa8611f41b4a032b846f
.
.
.
```

To find the object associated with a commit, simply take the commit hash you found above:

```
d83ab2b104e4addd03947ed3b1ca57b2e68dfc85
```

Decompose that into a directory name and an object identifier:
Directory: `d8`
Object identifier: `3ab2b104e4addd03947ed3b1ca57b2e68dfc85`
Now you know that the object you want to look at is inside the **d8** directory. Navigate into that directory and pull up another listing to see the files inside:

```
.
.
.
d7
├── c33fdd7d35372cba78386dfe5928f1ba8dfb70
└── e92f9daeec6cd217fda01c6b726cb07866728c
d8
└── 3ab2b104e4addd03947ed3b1ca57b2e68dfc85
d9
└── 809bc1dafdec03f0d60f41f6c7f6cfc3228c80
da
├── 967ae1f60e59d2a223e37301f63050dca0cf6f
└── fe823560ecc5694151c37187f978b5cf3d5cf1
.
.
.
```

In my case, I only see one file: **3ab2b104e4addd03947ed3b1ca57b2e68dfc85**. You may see other files in there, and that's to be expected in a moderately busy repository.
You can't take a look at this object directly, though, as objects in Git are compressed. If you tried to look at it using `cat 3ab2b104e4addd03947ed3b1ca57b2e68dfc85` or similar, you'll probably see a pile of gibberish like so, along with a few chirps from your computer as it tries to read control characters from the binary object:

```
xu?Ko?0??51??Ⅎ
yB
    ??f?y?cBwo?{¿?|♭FL?:?@??_?0Td5?D2Br?D$??f?B??b?5W?HÅ?H*?&??(fb↺
                                                                dC!DV%?????D@?(???u0??8{?w????0?IULC1????@(<?s '
mO????????ze?S????>?K8                  89_vxm(#?jxOs?u?b?5m????=w\l?
%?O??[V?t]?^??????G6.n?Mu?%
                                  ?̇?X??Ẋv??x?EX???:sys???G2?y??={X?Ⅎe?X?4u???????4o'G??^"q␤???$?Ccu?ml???vB_)?I?6?$?(?E9?z??nUmV?Em]?p???3?`??????q?Ţqjw????VR?O? q?.r???e|lN?p??Gq?)?????#???85V?W6?????
)|Wc*??8?1a?b?=?f*??pSvx3??;???3??^??O?S}???Z4?/?%J?
^??*ɹF?of??O
```

Viewing Git objects

Git provides a way to look at the contents of a compressed Git object: `git cat-file`. This decompresses the object and writes it out to your console in a human-readable form. You can simply pass it a short or long hash, and Git will write out the contents of that object in a human-readable form.
So take a look at the uncompressed form of the object file with the following command, substituting in the short or long hash from the commit that you want to look at:

```
git cat-file -p d83ab2b
```

The `-p` option here tells Git to figure out what type of object it's dealing with and to provide appropriately formatted output.

The commit object

In my case, Git tells me the details about my chosen commit object:

```
tree c0425d3b2aa2bfbbc0a08efda69ed00286dec6e4
parent 5415c13d2449f9719a8a8e84ee25105a1a587c5f
author cripsy8888 <chris@razeware.com> 1549849076 -0400
committer GitHub <noreply@github.com> 1549849076 -0400
gpgsig -----BEGIN PGP SIGNATURE-----

 wsBcBAABCAAQBQJcYNH0CRBK7hj4Ov3rIwAAdHIIABLgrn6UmK0fzh/jqaIg7ax2
 kie1Grd4EqLA+kuNT0jR+qTbc6x+OwlYt2PWZXOzfyOwY3UNKByHWhJDrhgzjLjB
 65CT7GGmMOKlGi7gis3W6jZetka+Lnauoeg9e/VnAu6q/9JOv6ZyRN4j13wYpnK1
 9wyooTbV2ipKMRFBs56DjL+6LkJcuIdD98rqluUzugGIvjFnGmIUCKF48511bN3Q
 eZ+PsFGeqqIFHdWnXOyvBhzjVogoumR8K7WtQ8tGMXnAnwlBo0s+sikJa4tTmO/o
 feVt0ln+frS+j6zhnC1RHRPkucPDBV9DuVdrSiA4w1xmXCXmVZ26bCEHQkaf1Z0=
 =QrF9
 -----END PGP SIGNATURE-----


Ticked off last item added
```

```
No one would believe you could skew election results...
```

There's a wealth of information here, but what you're interested in is the **`tree`** hash.

The tree object

The tree object is a pointer to another object that holds the collection of files for this commit.
So execute `git cat-file` again to see what's inside *that* object, substituting your particular hash:

```
git cat-file -p c0425d3b2aa2bfbbc0a08efda69ed00286dec6e4
```

I get the following information about the tree object:

```
100644 blob 8b23445f4a55ae5f9e38055dec94b27ef2b14150    LICENSE
100644 blob f5c651739ff232f6226d686724f3c9618dd9f840    README.md
040000 tree d27f2eb006fff5b83fdc5d6639c7cfabdcf9fc37    articles
040000 tree 0b2d0890591a57393dc40e2155bff8901acafbb6    books
040000 tree 028f2d5e0a0f9990263039794149dfa0126bede    videos
```

Ah — that looks a *lot* like the working tree of your project, doesn't it? That's because that's precisely what this *is*: a compressed representation of your file structure inside the repository.
Now, again, this object is simply a pointer to other objects. But you can keep unwrapping objects as you go.

The blob object

For instance, you can see the state of the **LICENSE** file in this commit with `git cat-file`:

```
git cat-file -p 8b23445f4a55ae5f9e38055dec94b27ef2b14150
```

I see all that glorious legalese of the MIT license I added to my repository so many months ago:

```
MIT License

Copyright (c) 2019

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell...
<snip>
```

You can dig further into the tree by following the references down. What's inside the **articles** directory in this commit? The following command will tell you that:

```
git cat-file -p d27f2eb006fff5b83fdc5d6639c7cfabdcf9fc37
```

I see the following files inside that directory:

```
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    .keep
100644 blob f8a69b62146eceef1b9078fed8788fbb6089f14f    clickbait_ideas.md
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    ios_article_ideas.md
```

Looking inside **clickbait_ideas.md** with `git cat-file` again, I'll see the full contents of that file as I committed it:

```
# Clickbait Article Ideas

These articles shouldn't really have any content but need irresistible titles.

- [ ] Top 10 iOS interview questions
- [ ] 8 hottest rumors about Swift 5 - EXPOSED
- [ ] Try these five weird Xcode tips to reduce app bloat
- [ ] Apple to skip iOS 13, eyes a piece of Android's pie
- [ ] 15 ways Android beats iOS into the ground and 7 ways it doesn't
- [ ] I migrated my entire IT department back to Windows XP - and then this happened
- [ ] The Apple announcement that should worry Swift developers
- [ ] iOS 13 to bring back skeuomorphism amidst falling iPhone sales
- [x] Machine Learning to blame for skewed election results
```

You could keep digging further, but I'm sure you've seen enough to get an understanding of how Git stores commits, trees and the objects that represent the files in your project. It's turtles all the way down, man.
So you can see how easily Git can reconstruct a branch, based on a single commit hash:

. You switch to a named branch, which is a label that references a commit hash.

. Git finds that commit object by its hash, then it gets the tree hash from the commit object.

. Git then recurses down the tree object, uncompressing file objects as it goes.

. Your working directory now represents the state of that branch as it is stored in the repo.

That's enough mucking about under the hood of Git; navigate back up to the root directory of your project and let Git take care of its own business. You have more important things to attend to.

Key points

Git uses the SHA-1 hash of content to create references to commits, trees and blobs.
A commit object stores the metadata about a commit, such as the parent, the author, timestamps and references to the file tree of this commit.
A tree object is a collection of references to either child trees or blob objects.
Blob objects are compressed collections of files; usually, the set of files in a particular directory inside the tree.
`git rev-parse`, among other things, will translate a short hash into a long hash.
`git cat-file`, among other things, will show you the pertinent metadata about an object.

Where to go from here?

Git has quite an elegant and powerful design when you think about it. And the wonderful thing is that all of this is abstracted away from you at the command line, so you don't need to know *anything* about the mechanisms underneath if you're the type who thinks ignorance is bliss.
But for those of you who *do* want to know how things work, and for those people who find that development is messy and unpredictable, you're likely to want to master merge conflicts.

2. Merge Conflicts v. Introduction

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.