Up to date for Git 2.21

Mastering Git by Tutorials

FIRST EDITION

Understanding Git Internals and Commands

By the raywenderlich Tutorial Team

Chris Belanger

Table of Contents: Overview

Table of Contents: Extended

Chances are if you're involved with software development you've heard of and have used Git at some point in your life. Version control systems are critical for any successful collaborative software project. Git is both simple to start using and accommodating for the most complex tasks with version control. Even seasoned Git users hit roadblocks on how to handle common situations.

Mastering Git is here to help! This book is the easiest and fastest way to get hands-on experience with using Git for version control of your projects.

We are pleased to offer you this sample chapter from *Mastering Git*:

How Does Git Actually Work?: If you've been using Git for a while, you might be wondering how it actually works. Discover how Git is built on top of a simple key-value store-based file system, and what power this provides to you.

We hope this hands-on look inside the book will give you a good idea of what's to come in the full version and show you why this book is a must-have for any app developer.

The full book is now available for purchase:

- https://store.raywenderlich.com/products/mastering-git.

Enjoy!

The *Mastering Git* Team

Mastering Git

Chris Belanger

Notice of Rights

Notice of Liability

Trademarks

Book License

By purchasing *Mastering Git*, you have the following license:

- You are allowed to use and/or modify the source code in *Mastering Git* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Mastering Git* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Mastering Git*, available at www.raywenderlich.com".
- The source code included in *Mastering Git* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Mastering Git* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

What You Need

To follow along with this book, you'll need the following:

- **Git 2.21.0 or later.** Git is the software package that you'll use for all of the work in this book. There are installers for macOS, Windows, and Linux available for free from the official Git page here: https://git-scm.com/downloads.

B Book Source Code & Forums

If you bought the digital edition

The digital edition of this book comes with the source code for the starter and completed projects for each chapter. These resources are included with the digital edition you downloaded from store.raywenderlich.com.

If you bought the print version

You can get the source code for the print edition of the book here:

https://store.raywenderlich.com/products/mastering-git

Forums

We've also set up an official forum for the book at forums.raywenderlich.com. This is a great place to ask questions about the book or to submit any errors you may find.

Digital book editions

We have a digital edition of this book available in both ePUB and PDF, which can be handy if you want a soft copy to take with you, or you want to quickly search for a specific term within the book.

Buying the digital edition version of the book also has a few extra benefits: free updates each time we update the book, access to older versions of the book, and you can download the digital editions from anywhere, at anytime.

Visit our *Mastering Git* store page here:

- https://store.raywenderlich.com/products/mastering-git.

And if you purchased the print version of this book, you're eligible to upgrade to the digital editions at a significant discount! Simply email support@razeware.com with your receipt for the physical copy and we'll get you set up with the discounted digital edition version of the book.

Chapter 12: How Does Git Actually Work?

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.

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 it's 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, execute the `git log` command as below:

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

My log result looks like the following:

```
f8098fa (HEAD -> master, origin/master, origin/HEAD) Merge
branch 'clickbait' with changes from crispy8888/clickbait
d83ab2b (crispy8888/clickbait, clickbait) Ticked off the last
item added
5415c13 More clickbait ideas
fed347d (from-crispy8888) Merge branch 'master' of https://
www.github.com/belangerc/ideas
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:

```
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 *looks* like on disk.

The inner workings of Git

Change to your terminal program and navigate to the main directory of your repository. Once you're there, 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.

Start by navigating 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:

```
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??Л
yB
    ??f?y?cBшo?{ɛ?|ЬFL?:?@??_?0Td5?D2Br?D$??f?B??b?5W?HÁ?H*?&??
(fbd

dC!DV%?????D@?(???u0??8{?w????0?IULC1????@(<?s '
mO????????ƶe?S????>?K8                    89_vxm(#?jxOs?u?b?5m????
=w\l?
%?O??[V?t]?^??????G6.n?Mu?%
                              ??X??Xv??x?EX???:sys???G2?y??={X?ᑎe?
X?4u???????4o'G??^"qِ???$?Ccu?ml???vB_)?I?6?$?(?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?
                                                    F?of??Oڑ*??`
```

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+6LkJcuIdD98rqluUzugGIvjFnGmIUCKF485l1bN3Q
 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 028f2d5e0a0f99902638039794149dfa0126bede    videos
```

Ah — that looks a *lot* like the working tree of the project from the first part of this book, 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 chapters 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:

1. You switch to a named branch, which is a label that references a commit hash.
2. Git finds that commit object by its hash, then it gets the tree hash from the commit object.
3. Git then recurses down the tree object, uncompressing file objects as it goes.
4. 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 who want to be able to fix things when they go awry (and in Git, they often do), then this entire section will be a treat for you. The next chapter deals with a very common scenario that will (and should) occur with some regularity if you're doing any level of distributed development: merge conflicts.

Where to Go From Here?

We hope you enjoyed this sample of *Mastering Git*!

If you enjoyed this sample, be sure to check out the full book, which contains the following chapters:

1. **A Crash Course in Git**: Learn how to get started with Git, the differences between platforms, and a quick overview of the typical Git workflow.
2. **Cloning a Repo**: It's quite common to start by creating a copy of somebody else's repository. Discover how to clone a remote repo to your local machine, and what constitutes "forking" a repository.
3. **Committing Your Changes**: A Git repo is made up of a sequence of commits—each representing the state of your code at a point in time. Discover how to create these commits to track the changes you make in your code.
4. **The Staging Area**: Before you can create a Git commit, you have to use the "add" command. What does it do? Discover how to use the staging area to great effect through the interactive git add command.
5. **Ignoring Files**: Sometimes, there are things that you really don't want to store in your source code repository.
6. **Git log and Viewing History**: There's very little point in creating a nice history of your source code if you can't explore it. You'll discover the versatility of the git log command—displaying branches, graphs and even filtering the history.
7. **Branching**: The real power in Git comes from its branching and merging model. This allows you to work on multiple things simultaneously. Discover how to manage branches, and exactly what they are in this chapter.

8. **Syncing with a Remote**: You've been working hard on your local copy of the Git repository, and now you want to share this with your friends. See how you can share through using remotes, and how you can use multiple remotes at the same time.
9. **Creating a Repo**: If you are starting a new project, and want to use Git for source control, you first need to create a new repository.
10. **Merging**: Branches in Git without merging would be like basketball without the hoop—fun, sure, but with very little point. In this chapter you'll learn how you can use merging to combine the work on multiple branches back into one.
11. **Stashes**: Git stashes offer a great way for you to create a temporary snapshot of what you're working on, without having to create a full-blown commit. Discover when that might be useful, and how to go about it.
12. **How Does Git Actually Work?**: If you've been using Git for a while, you might be wondering how it actually works. Discover how Git is built on top of a simple key-value store-based file system, and what power this provides to you.
13. **Merge Conflicts**: Merging isn't always as simple as it might first appear. In this chapter you will learn how to handle merge conflicts — which occur when Git cannot work out how to automatically combine changes.
14. **What is Rebasing?**: Rebasing is poorly understood, although it can be an incredibly powerful tool. In this chapter, we'll cover what happens behind the scenes when you rebase and set you up for some useful applications of rebasing in the coming chapters.
15. **Rebasing to Rewrite History**: Rebase is a whole lot more powerful than just as a replacement for merge. It offers the ability to completely rewrite the history of your git repo.
16. **Gitignore After the Fact**: Gitignore is easy right? If you've been using it for a while you'll know that isn't always true. Discover how you can fix problems with gitignore such as handling files that have been accidentally committed to the repository.
17. **Cherry Picking**: Cherry picking provides a way to grab single commits from other branches, and apply them to your own branch.
18. **Many Faces of Undo**: One of the common questions associated with git is "how can I get out of this mess?". In this chapter you'll learn about the different "undo" commands that git provides — what they are and when to use them.

19. **Centralized Workflow**: This model means you work in master all the time. Although this might seem terrifying, it actually works rather well for small teams with infrequent commits.
20. **Feature Branch Workflow**: Feature branches are used to create new features in your code and then merged to master when they're done.
21. **Gitflow Workflow**: Gitflow uses feature branches for development which are merged to develop. When develop is ready for release, a branch is made. After the release is done, the release branch is merged to master.
22. **Forking Workflow**: Forking is a cloned copy of a public repository. Contributing back changes from your fork (or copy) requires a pull request to be created.
23. **Best Practices**: There are lots of good practices, regardless of which workflow you use. You'll learn some of the most common and helpful ones in this chapter.

You can find the book on the raywenderlich.com store here: https://store.raywenderlich.com/products/mastering-git

We hope you enjoy the book!

-Chris and the *Mastering Git* team