7.1 Git Tools - Revision Selection

By now, you've learned most of the day-to-day commands and workflows that you need to manage or maintain a Git repository for your source code control. You've accomplished the basic tasks of tracking and committing files, and you've harnessed the power of the staging area and lightweight topic branching and merging.

Now you'll explore a number of very powerful things that Git can do that you may not necessarily use on a day-to-day basis but that you may need at some point.

Revision Selection

Git allows you to refer to a single commit, set of commits, or range of commits in a number of ways. They aren't necessarily obvious but are helpful to know.

Single Revisions

You can obviously refer to any single commit by its full, 40-character SHA-1 hash, but there are more human-friendly ways to refer to commits as well. This section outlines the various ways you can refer to any commit.

Short SHA-1

Git is smart enough to figure out what commit you're referring to if you provide the first few characters of the SHA-1 hash, as long as that partial hash is at least four characters long and unambiguous; that is, no other object in the object database can have a hash that begins with the same prefix.

For example, to examine a specific commit where you know you added certain functionality, you might first run the `git log` command to locate the commit:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    Fix refs handling, add gc auto, update tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    Add some blame and merge stuff
```

In this case, say you're interested in the commit whose hash begins with `1c002dd…`. You can inspect that commit with any of the following variations of `git show` (assuming the shorter versions are unambiguous):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git can figure out a short, unique abbreviation for your SHA-1 values. If you pass `--abbrev-commit` to the `git log`command, the output will use shorter values but keep them unique; it defaults to using seven characters but makes them longer if necessary to keep the SHA-1 unambiguous:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d Change the version number
085bb3b Remove unnecessary test code
a11bef0 Initial commit
```

Generally, eight to ten characters are more than enough to be unique within a project. For example, as of February 2019, the Linux kernel (which is a fairly sizable project) has over 875,000 commits and almost seven million objects in its object database, with no two objects whose SHA-1s are identical in the first 12 characters.

A SHORT NOTE ABOUT SHA-1

A lot of people become concerned at some point that they will, by random happenstance, have two distinct objects in their repository that hash to the same SHA-1 value. What then?

If you do happen to commit an object that hashes to the same SHA-1 value as a previous *different* object in your repository, Git will see the previous object already in your Git database, assume it was already written and simply reuse it. If you try to check out that object again at some point, you'll always get the data of the first object.

Note

However, you should be aware of how ridiculously unlikely this scenario is. The SHA-1 digest is 20 bytes or 160 bits. The number of randomly hashed objects needed to ensure a 50% probability of a single collision is about 2^{80}(the formula for determining collision probability is `p = (n(n-1)/2) * (1/2^160))`. 2^{80} is 1.2 x 10^{24} or 1 million billion billion. That's 1,200 times the number of grains of sand on the earth.

Here's an example to give you an idea of what it would take to get a SHA-1 collision. If all 6.5 billion humans on Earth were programming, and every second, each one was producing code that was the equivalent of the entire Linux kernel history (6.5 million Git objects) and pushing it into one enormous Git repository, it would take roughly 2 years until that repository contained enough objects to have a 50% probability of a single SHA-1 object collision. Thus, an organic SHA-1 collision is less likely than every member of your programming team being attacked and killed by wolves in unrelated incidents on the same night.

If you dedicate several thousands of dollars' worth of computing power to it, it is possible to synthesize two files with the same hash, as proven on https://shattered.io/ in February 2017. Git is moving towards using SHA256 as the default hashing algorithm, which is much more resilient to collision attacks, and has code in place to help mitigate this attack (although it cannot completely eliminate it).

Branch References

One straightforward way to refer to a particular commit is if it's the commit at the tip of a branch; in that case, you can simply use the branch name in any Git command that expects a reference to a commit. For instance, if you want to examine the last commit object on a branch, the following commands are equivalent, assuming that the `topic1` branch points to commit `ca82a6d…`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

If you want to see which specific SHA-1 a branch points to, or if you want to see what any of these examples boils down to in terms of SHA-1s, you can use a Git plumbing tool called `rev-parse`. You can see Git Internals for more information about plumbing tools; basically, `rev-parse` exists for lower-level operations and isn't designed to be used in day-to-day operations. However, it can be helpful sometimes when you need to see what's really going on. Here you can run `rev-parse`on your branch.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

RefLog Shortnames

One of the things Git does in the background while you're working away is keep a "reflog" — a log of where your HEAD and branch references have been for the last few months.

You can see your reflog by using `git reflog`:

```
$ git reflog
734713b HEAD@{0}: commit: Fix refs handling, add gc auto, update tests
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by the 'recursive' strategy.
1c002dd HEAD@{2}: commit: Add some blame and merge stuff
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
95df984 HEAD@{4}: commit: # This is a combination of two commits.
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

Every time your branch tip is updated for any reason, Git stores that information for you in this temporary history. You can use your reflog data to refer to older commits as well. For example, if you want to see the fifth prior value of the HEAD of your repository, you can use the `@{5}` reference that you see in the reflog output:

```
$ git show HEAD@{5}
```

You can also use this syntax to see where a branch was some specific amount of time ago. For instance, to see where your `master` branch was yesterday, you can type:

```
$ git show master@{yesterday}
```

That would show you where tip of your `master` branch was yesterday. This technique only works for data that's still in your reflog, so you can't use it to look for commits older than a few months.

To see reflog information formatted like the `git log` output, you can run `git log -g`:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: Fix refs handling, add gc auto, update tests
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    Fix refs handling, add gc auto, update tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

It's important to note that reflog information is strictly local — it's a log only of what *you've* done in *your* repository. The references won't be the same on someone else's copy of the repository; also, right after you initially clone a repository, you'll have an empty reflog, as no activity has occurred yet in your repository. Running `git show HEAD@{2.months.ago}`will show you the matching commit only if you cloned the project at least two months ago — if you cloned it any more recently than that, you'll see only your first local commit.

Think of the reflog as Git's version of shell history

Tip If you have a UNIX or Linux background, you can think of the reflog as Git's version of shell history, which emphasizes that what's there is clearly relevant only for you and your "session", and has nothing to do with anyone else who might be working on the same machine.

Escaping braces in PowerShell

Note When using PowerShell, braces like `{` and `}` are special characters and must be escaped. You can escape them with a backtick `` ` `` or put the commit reference in quotes:

```
$ git show HEAD@{0}     # will NOT work
$ git show HEAD@`{0`}   # OK
$ git show "HEAD@{0}"   # OK
```

Ancestry References

The other main way to specify a commit is via its ancestry. If you place a `^` (caret) at the end of a reference, Git resolves it to mean the parent of that commit. Suppose you look at the history of your project:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b Fix refs handling, add gc auto, update tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd Add some blame and merge stuff
|/
* 1c36188 Ignore *.gem
* 9b29157 Add open3_detach to gemspec file list
```

Then, you can see the previous commit by specifying `HEAD^`, which means "the parent of HEAD":

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

Escaping the caret on Windows

Note

On Windows in `cmd.exe`, `^` is a special character and needs to be treated differently. You can either double it or put the commit reference in quotes:

```
$ git show HEAD^     # will NOT work on Windows
$ git show HEAD^^    # OK
$ git show "HEAD^"   # OK
```

You can also specify a number after the `^` to identify *which* parent you want; for example, `d921970^2` means "the second parent of d921970." This syntax is useful only for merge commits, which have more than one parent — the *first* parent of a merge commit is from the branch you were on when you merged (frequently `master`), while the *second* parent of a merge commit is from the branch that was merged (say, `topic`):

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    Add some blame and merge stuff

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

The other main ancestry specification is the ~ (tilde). This also refers to the first parent, so `HEAD~` and `HEAD^` are equivalent. The difference becomes apparent when you specify a number. `HEAD~2` means "the first parent of the first parent," or "the grandparent" — it traverses the first parents the number of times you specify. For example, in the history listed earlier, `HEAD~3` would be:

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    Ignore *.gem
```

This can also be written `HEAD~~~`, which again is the first parent of the first parent of the first parent:

```
$ git show HEAD~~~
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    Ignore *.gem
```

You can also combine these syntaxes — you can get the second parent of the previous reference (assuming it was a merge commit) by using `HEAD~3^2`, and so on.

Commit Ranges

Now that you can specify individual commits, let's see how to specify ranges of commits. This is particularly useful for managing your branches — if you have a lot of branches, you can use range specifications to answer questions such as, "What work is on this branch that I haven't yet merged into my main branch?"

Double Dot

The most common range specification is the double-dot syntax. This basically asks Git to resolve a range of commits that are reachable from one commit but aren't reachable from another. For example, say you have a commit history that looks like Example history for range selection.

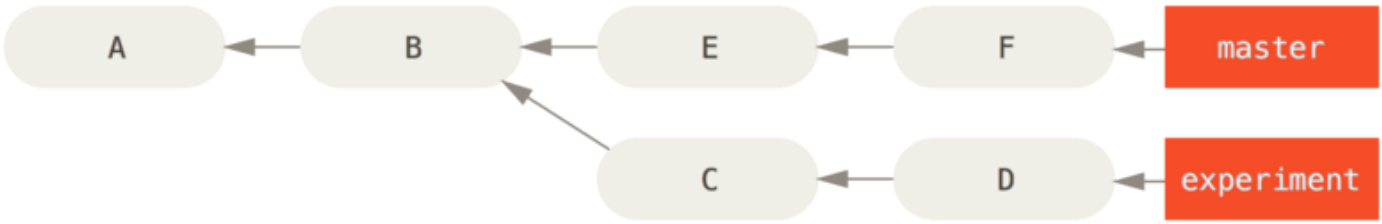

Figure 136. Example history for range selection

Say you want to see what is in your `experiment` branch that hasn't yet been merged into your `master` branch. You can ask Git to show you a log of just those commits with `master..experiment` — that means "all commits reachable from `experiment` that aren't reachable from `master`." For the sake of brevity and clarity in these examples, the letters of the commit objects from the diagram are used in place of the actual log output in the order that they would display:

```
$ git log master..experiment
D
C
```

If, on the other hand, you want to see the opposite — all commits in `master` that aren't in `experiment` — you can reverse the branch names. `experiment..master` shows you everything in `master` not reachable from `experiment`:

```
$ git log experiment..master
F
E
```

This is useful if you want to keep the `experiment` branch up to date and preview what you're about to merge. Another frequent use of this syntax is to see what you're about to push to a remote:

```
$ git log origin/master..HEAD
```

This command shows you any commits in your current branch that aren't in the `master` branch on your `origin` remote. If you run a `git push` and your current branch is tracking `origin/master`, the commits listed by `git log origin/master..HEAD`are the commits that will be transferred to the server. You can also leave off one side of the syntax to have Git assume `HEAD`. For example, you can get the same results as in the previous example by typing `git log origin/master..` — Git substitutes `HEAD` if one side is missing.

Multiple Points

The double-dot syntax is useful as a shorthand, but perhaps you want to specify more than two branches to indicate your revision, such as seeing what commits are in any of several branches that aren't in the branch you're currently on. Git allows you to do this by using either the `^` character or `--not` before any reference from which you don't want to see reachable commits. Thus, the following three commands are equivalent:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

This is nice because with this syntax you can specify more than two references in your query, which you cannot do with the double-dot syntax. For instance, if you want to see all commits that are reachable from `refA` or `refB` but not from `refC`, you can use either of:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

This makes for a very powerful revision query system that should help you figure out what is in your branches.

Triple Dot

The last major range-selection syntax is the triple-dot syntax, which specifies all the commits that are reachable by *either*of two references but not by both of them. Look back at the example commit history in Example history for range selection. If you want to see what is in `master` or `experiment` but not any common references, you can run:

```
$ git log master...experiment
F
E
D
C
```

Again, this gives you normal `log` output but shows you only the commit information for those four commits, appearing in the traditional commit date ordering.

A common switch to use with the `log` command in this case is `--left-right`, which shows you which side of the range each commit is in. This helps make the output more useful:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

With these tools, you can much more easily let Git know what commit or commits you want to inspect.

prev | next