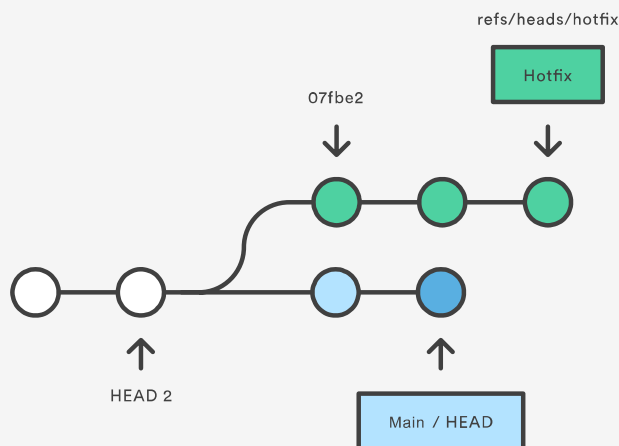


Git Refs: An Overview

Hashes Refs Packed Refs Special Refs Refspecs Relative Refs The Reflog Summary.

Git is all about commits: you stage commits, create commits, view old commits, and transfer commits between repositories using many different Git commands. The majority of these commands operate on a commit in some form or another, and many of them accept a commit reference as a parameter. For example, you can use `git checkout` to view an old commit by passing in a commit hash, or you can use it to switch branches by passing in a branch name.

Many ways of referring to a commit



By understanding the many ways to refer to a commit, you make all of these commands that much more powerful. In this chapter, we'll shed some light on the internal workings of common commands like `git checkout`, `git branch`, and `git push` by exploring the many methods of referring to a commit.

We'll also learn how to revive seemingly "lost" commits by accessing them through Git's reflog mechanism.

Hashes

The most direct way to reference a commit is via its SHA-1 hash. This acts as the unique ID for each commit. You can find the hash of all your commits in the `git log` output.

```
commit 0c708fdec272bc4446c6cabea4f0022c2b616eba
```

When passing the commit to other Git commands, you only need to specify enough characters to uniquely identify the commit. For example, you can inspect the above commit with `git show` by running the following command:

```
git show 0c708f
```

It's sometimes necessary to resolve a branch, tag, or another indirect reference into the corresponding commit hash. For this, you can use the `git rev-parse` command. The following returns the hash of the commit pointed to by the `main` branch:

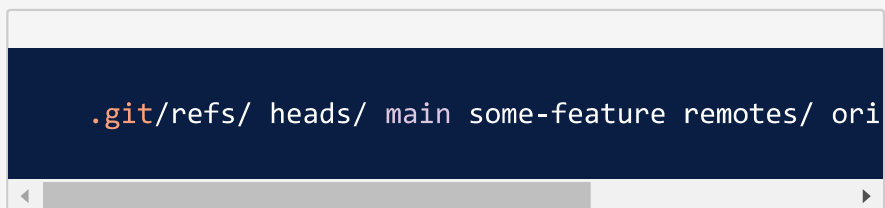
```
git rev-parse main
```

This is particularly useful when writing custom scripts that accept a commit reference. Instead of parsing the commit reference manually, you can let `git rev-parse` normalize the input for you.

Refs

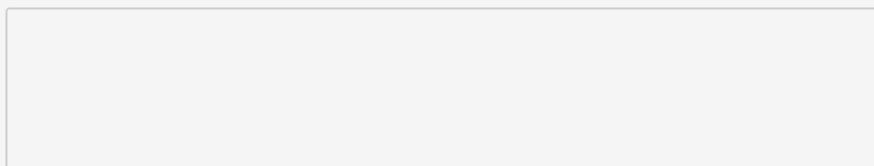
A **ref** is an indirect way of referring to a commit. You can think of it as a user-friendly alias for a commit hash. This is Git's internal mechanism of representing branches and tags.

Refs are stored as normal text files in the `.git/refs` directory, where `.git` is usually called `.git`. To explore the refs in one of your repositories, navigate to `.git/refs`. You should see the following structure, but it will contain different files depending on what branches, tags, and remotes you have in your repo:

A terminal window with a dark blue background and light blue text. The text shows the directory structure of a Git repository's refs: `.git/refs/ heads/ main some-feature remotes/ ori`. A horizontal scrollbar is visible at the bottom of the terminal window, indicating that the text is longer than the visible width.

```
.git/refs/ heads/ main some-feature remotes/ ori
```

The `heads` directory defines all of the local branches in your repository. Each filename matches the name of the corresponding branch, and inside the file you'll find a commit hash. This commit hash is the location of the tip of the branch. To verify this, try running the following two commands from the root of the Git repository:

An empty rectangular box with a thin gray border, intended for the user to enter the commands mentioned in the text above.

```
# Output the contents of `refs/heads/main` file:
```

The commit hash returned by the `cat` command should match the commit ID displayed by `git log`.

To change the location of the `main` branch, all Git has to do is change the contents of the `refs/heads/main` file. Similarly, creating a new branch is simply a matter of writing a commit hash to a new file. This is part of the reason why Git branches are so lightweight compared to SVN.

The `tags` directory works the exact same way, but it contains tags instead of branches. The `remotes` directory lists all remote repositories that you created with `git remote` as separate subdirectories. Inside each one, you'll find all the remote branches that have been fetched into your repository.

Specifying Refs

When passing a ref to a Git command, you can either define the full name of the ref, or use a short name and let Git search for a matching ref. You should already be familiar with short names for refs, as this is what you're using each time you refer to a branch by name.

```
git show some-feature
```

The `some-feature` argument in the above command is actually a short name for the branch. Git resolves this

to `refs/heads/some-feature` before using it. You can also specify the full ref on the command line, like so:

```
git show refs/heads/some-feature
```

This avoids any ambiguity regarding the location of the ref. This is necessary, for instance, if you had both a tag and a branch called `some-feature`. However, if you're using proper naming conventions, ambiguity between tags and branches shouldn't generally be a problem.

We'll see more full ref names in the Refspecs section.

Packed Refs

For large repositories, Git will periodically perform a garbage collection to remove unnecessary objects and compress refs into a single file for more efficient performance. You can force this compression with the garbage collection command:

```
git gc
```

This moves all of the individual branch and tag files in the `refs` folder into a single file called `packed-refs` located in the top of the `.git` directory. If you open up this file, you'll find a mapping of commit hashes to refs:

```
00f54250cf4e549fdfcafe2cf9a2c90bc3800285 refs/he
```

On the outside, normal Git functionality won't be affected in any way. But, if you're wondering why your `.git/refs` folder is empty, this is where the refs went.

Special Refs

In addition to the `refs` directory, there are a few special refs that reside in the top-level `.git` directory. They are listed below:

- `HEAD` – The currently checked-out commit/branch.
- `FETCH_HEAD` – The most recently fetched branch from a remote repo.
- `ORIG_HEAD` – A backup reference to `HEAD` before drastic changes to it.
- `MERGE_HEAD` – The commit(s) that you're merging into the current branch with `git merge`.
- `CHERRY_PICK_HEAD` – The commit that you're cherry-picking.

These refs are all created and updated by Git when necessary. For example, The `git pull` command first runs `git fetch`, which updates the `FETCH_HEAD` reference. Then, it runs `git merge FETCH_HEAD` to finish pulling the fetched branches into the repository. Of course, you can use all of these like any other ref, as I'm sure you've done with `HEAD`.

These files contain different content depending on their type and the state of your repository. The `HEAD` ref can contain either a **symbolic ref**, which is simply a reference to another ref instead of a commit hash, or a commit hash. For example, take a look at the contents of `HEAD` when you're on the `main` branch:

```
git checkout main cat .git/HEAD
```

This will output `ref: refs/heads/main`, which means that `HEAD` points to the `refs/heads/main` ref. This is how Git knows that the `main` branch is currently checked out. If you were to switch to another branch, the contents of `HEAD` would be updated to reflect the new branch. But, if you were to check out a commit instead of a branch, `HEAD` would contain a commit hash instead of a symbolic ref. This is how Git knows that it's in a detached HEAD state.

For the most part, `HEAD` is the only reference that you'll be using directly. The others are generally only useful when writing lower-level scripts that need to hook into Git's internal workings.

Refspecs

A refspec maps a branch in the local repository to a branch in a remote repository. This makes it possible to manage remote branches using local Git commands and to configure some advanced `git push` and `git fetch` behavior.

A refspec is specified as `[+] <src> : <dst>`. The `<src>` parameter is the source branch in the local repository, and the `<dst>` parameter is the destination branch in the remote repository. The optional `+` sign is for forcing the remote repository to perform a non-fast-forward update.

Refspecs can be used with the `git push` command to give a different name to the remote branch. For example, the following command pushes the `main` branch to the `origin` remote repo like an ordinary `git push`, but it uses `qa-main` as the name for the branch in the `origin` repo. This is useful for QA teams that need to push their own branches to a remote repo.

```
git push origin main:refs/heads/qa-main
```

You can also use refspecs for deleting remote branches. This is a common situation for feature-branch workflows that push the feature branches to a remote repo (e.g., for backup purposes). The remote feature branches still reside in the remote repo after they are deleted from the local repo, so you get a build-up of dead feature branches as your project progresses. You can delete them by pushing a refspec that has an empty `-` parameter, like so:

```
git push origin :some-feature
```

This is very convenient, since you don't need to log in to your remote repository and manually delete the

remote branch. Note that as of Git v1.7.0 you can use the `--delete` flag instead of the above method. The following will have the same effect as the above command:

```
git push origin --delete some-feature
```

By adding a few lines to the Git configuration file, you can use refsspecs to alter the behavior of `git fetch`. By default, `git fetch` fetches *all* of the branches in the remote repository. The reason for this is the following section of the `.git/config` file:

```
[remote "origin"] url = https://git@github.com:m
```

The `fetch` line tells `git fetch` to download all of the branches from the `origin` repo. But, some workflows don't need all of them. For example, many continuous integration workflows only care about the `main` branch. To fetch only the `main` branch, change the `fetch` line to match the following:

```
[remote "origin"] url = https://git@github.com:m
```

You can also configure `git push` in a similar manner. For instance, if you want to always push the `main` branch to `qa-main` in the `origin` remote (as we did above), you would change the config file to:

```
[remote "origin"] url = https://git@github.com:m
```

Refspecs give you complete control over how various Git commands transfer branches between repositories. They let you rename and delete branches from your local repository, fetch/push to branches with different names, and configure `git push` and `git fetch` to work with only the branches that you want.

Relative Refs

You can also refer to commits relative to another commit. The `~` character lets you reach parent commits. For example, the following displays the grandparent of `HEAD`:

```
git show HEAD~2
```

But, when working with merge commits, things get a little more complicated. Since merge commits have more than one parent, there is more than one path that you can follow. For 3-way merges, the first parent is from the branch that you were on when you performed the merge, and the second parent is from the branch that you passed to the `git merge` command.

The `~` character will always follow the *first* parent of a merge commit. If you want to follow a different parent, you need to specify which one with the `^` character.

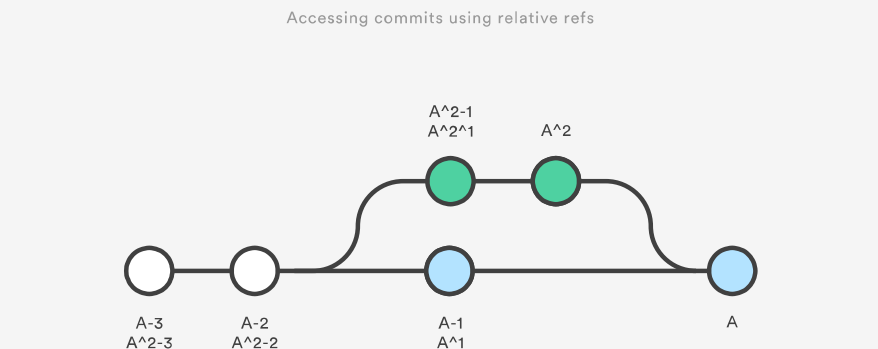
For example, if `HEAD` is a merge commit, the following returns the second parent of `HEAD`.

```
git show HEAD^2
```

You can use more than one `^` character to move more than one generation. For instance, this displays the grandparent of `HEAD` (assuming it's a merge commit) that rests on the *second* parent.

```
git show HEAD^2^1
```

To clarify how `~` and `^` work, the following figure shows you how to reach any commit from `A` using relative references. In some cases, there are multiple ways to reach a commit.



Relative refs can be used with the same commands that a normal ref can be used. For example, all of the following commands use a relative reference:

```
# Only list commits that are parent of the second
```

The Reflog

The reflog is Git's safety net. It records almost every change you make in your repository, regardless of whether you committed a snapshot or not. You can think of it as a chronological history of everything you've done in your local repo. To view the reflog, run the `git reflog` command. It should output something that looks like the following:

A terminal window with a dark blue background. The text displayed is "400e4b7 HEAD@{0}: checkout: moving from main to". Below the text is a horizontal scrollbar with a grey track and a white slider.

This can be translated as follows:

- You just checked out `HEAD~2`
- Before that you amended a commit message
- Before that you merged the `feature` branch into `main`
- Before that you committed a snapshot

The `HEAD{}` syntax lets you reference commits stored in the reflog. It works a lot like the `HEAD~` references from the previous section, but the `{}` refers to an entry in the reflog instead of the commit history.

You can use this to revert to a state that would otherwise be lost. For example, let's say you just scrapped a new feature with `git reset`. Your reflog might look something like this:

```
ad8621a HEAD@{0}: reset: moving to HEAD~3 298eb9
```

The three commits before the `git reset` are now dangling, which means that there is no way to reference them—except through the reflog. Now, let's say you realize that you shouldn't have thrown away all of your work. All you have to do is check out the `HEAD@{1}` commit to get back to the state of your repository before you ran `git reset`.

```
git checkout HEAD@{1}
```

This puts you in a detached `HEAD` state. From here, you can create a new branch and continue working on your feature.

Summary

You should now be quite comfortable referring to commits in a Git repository. We learned how branches and tags were stored as refs in the `.git` subdirectory, how to read a `packed-refs` file, how `HEAD` is represented, how to use refsspecs for advanced pushing and fetching, and how to use the relative `~` and `^` operators to traverse a branch hierarchy.

We also took a look at the reflog, which is a way to reference commits that are not available through any other means. This is a great way to recover from those little "Oops, I shouldn't have done that" situations.

The point of all this was to be able to pick out exactly the commit that you need in any given development scenario. It's very easy to leverage the skills you learned in this article against your existing Git knowledge, as some of the most common commands accept refs as arguments, including `git log`, `git show`, `git checkout`, `git reset`, `git revert`, `git rebase`, and many others.