Git in Ten Minutes

Spencer Tipping

October 16, 2010

Contents

1 Introduction

Git is a widely misunderstood and therefore oft-maligned tool. It does have some major flaws, but I believe most of the negative feelings towards Git arise because people don't understand how it works. This guide is similar in purpose to "JavaScript in Ten Minutes" – the idea is to explain the confusing parts of Git for anyone familiar with some of its basic use cases.

A disclaimer: I don't know Git as well as I know JavaScript. There are lots of longer guides online written by people with more knowledge of Git than I have, so if there are discrepancies between this guide and someone else's, chances are they're right.

2 Commits and the Index

Git has two things that you use to build history (which means branches, tags, etc). One is the index, where you can incrementally construct a commit that you're going to make; and the other is a commit. Let's talk about commits.

2.1 Commits

Git doesn't care about branches, tags, or labels like `HEAD`. At the end of the day, all that exist are commits, and each commit has a name like `08c183fa68d1.....787f`. The name of a commit is derived from its contents via some hash function (SHA-1, I believe), and the contents of a commit consist of:

1. The parent commit ID(s) (there are multiple if you're merging)
2. Changes to files
3. The commit message and other commit metadata

So suppose you create an empty repository, add a file, and commit it:

```
$ mkdir tmp && cd tmp
$ git init
Initialized empty Git repository in /home/spencertipping/tmp/.git/
$ echo test > foo
$ git add foo
$ git commit -m "Added a new file called 'foo'"
[master (root-commit) a8c7d33] Added a file called 'foo'
 1 files changed, 1 insertions(+), 0 deletions(-)
  create mode 100644 foo
$ cat .git/refs/heads/master
a8c7d33649e32471493cebdaa1246a9281be2622
$
```

At this point you'll have a single commit in your repo and the branch `master` will point to that commit. Just for fun, we can temporarily bork our repo:

```
$ cp .git/refs/heads/master old-master
$ echo 'foo' > .git/refs/heads/master
$ git log
fatal: bad default revision 'HEAD'
$
```

Git complains about this because we've changed the commit that `master` points to. Git changes `refs/heads/master` all the time, but each of those changes points it to a valid commit ID, not a string like `foo`. To get our repo back in working order, just `mv old-master .git/refs/heads/master`.

OK, so why go to the trouble to break our repo? Perhaps surprisingly, that wasn't just because I like to live dangerously. The point of that exercise is to illustrate that commits are a fundamental idea and branches exist purely on the surface. Branches, tags, etc. are all *pointers* into the immutable, shared tree of commits. Important things to know about this:

1. Each commit is immutable (rebasing and using `--amend` and such create new commits, but don't modify existing ones)
2. Because of this, separate branches can share commits (which is why Git's branching is so fast)
3. The only time Git removes commit objects are when:
   (a) No tags or branches refer to them (we had this going on after we borked the repo), and
   (b) A `git gc` is run

2.2 The Index

It's hard to say whether the index is a feature or a problem. It could probably be done better,[1] but it's still useful the way it is. Basically, the index is what Git thinks the world (i.e. your working directory) should look like. You can modify the index with these commands (among others):

1. `git add` — copies a file from the working directory into the index. (Used with `-p`, you can copy pieces of a file into the index, which can be useful when you're debugging stuff)
2. `git reset` — copies a file from some commit into the index without updating the working directory. (Used with `--hard`, it also copies that file into the working directory, which can be useful for clobbering changes)
3. `git commit -a` — copies all tracked files into the index and commits.
4. `git checkout` — copies a commit into both the index and the working directory.

It's important to realize that `git diff` with no arguments always compares your working directory to the index, not to the most recent commit.[2] Also, `git commit` always commits from the index (though some variants, such as `-a` or when given a filename, update the index from the working directory before committing).

[1] It certainly could have a more meaningful name.
[2] This is why once you've `git add`ed something you won't see a diff for it anymore.

3 Branching

Let's suppose you've got a repo with a few commits on it, so that the history is entirely linear:

```
empty repo -> commit A -> commit B -> [master] commit C
```

In this case, `.git/refs/heads/master` contains the commit ID of commit C, and the commit log (stored in `.git/logs/refs/heads/master`) contains three log entries.[3] Let's create a new branch:

```
$ git checkout -b other-branch
Switched to a new branch 'other-branch'
$
```

The notation looks confusing; why are we running `git checkout` to create a new history path? It actually makes some sense. The deal is that we're pulling the latest commit (the default unless you specify one) into a branch called `other-branch`. (A shorthand for this command is `git branch other-branch`.) This does a few things:

1. Copies `.git/refs/heads/master` into `.git/refs/heads/other-branch` – now `master` and `other-branch` point to the same commit
2. Creates a log entry in the new file `.git/logs/refs/heads/other-branch` indicating that we created the branch from `HEAD`
3. Updates `.git/HEAD` to indicate that we're following `refs/heads/other-branch` (the effect of this is that when you commit, `refs/heads/other-branch` gets updated instead of `refs/heads/master`)

You can also branch from a previous point in history using the same interface. To do that, you say `git checkout -b <branch> <commit-id>`. This starts a branch at the specified commit rather than branching off of the current `HEAD`.

3.1 Tagging

A tag is just a human-friendly name for a commit. For example, you can say `git tag foo` to make `foo` a reference to the current `HEAD`. Then later on you can refer to `foo` instead of the commit ID that it points to. `git tag -l` lists all of the tags you've defined.

[3] This commit log is really handy in case you lose a commit somewhere. You can pull the commit ID out of the log and copy it into `.git/refs/heads/X`, restoring or moving a branch.

3.2 Rebasing

You may have used `git rebase` before. This command rewrites your commit history in a way that appears to be destructive. A common use case (and the only one that I'm familiar with) is to squash a bunch of commits into a single one. So, for example, let's suppose we have this history:

```
empty repo -> commit A -> [tag:foo] commit B -> commit C -> [master] commit D
```

You can run `git rebase -i foo` from the `master` branch, change all but the first line to begin with `squash`, and now your history will look like this:

```
empty repo -> commit A -> [tag:foo] commit B -> [master] squashed commit of C and D
```

What just happened? Git didn't really rewrite your history; rather, it created a new history for you and moved `master` to it. The actual commit map now has a fork in it:

```
empty repo -> commit A -> [tag:foo] commit B -> [master] squashed commit of C and D
                                             \
                                               -> commit C -> commit D
```

You can run `git checkout <commit D's ID>` to get back to where you were before the rebase, and run `git checkout master` to return.[4]

3.2.1 Errata

The previous version of this guide contained the following untrue statement:

> "Now before you go and assume that rebasing is safe, it isn't exactly – if you were to run a `git gc` without either a branch or a tag on commit D, then commits C and D would get deleted."

Thanks to Daniel Brockman (@dbrock) for pointing out that this is in fact not the case; commits with no named references can be kept around because they appear in the reflog (viewable with the `git reflog` command). I actually had no idea that such a thing existed, but it looks very useful. It's basically a record of every change to `HEAD`, which makes it easy to undo nearly any operation.

4 Remotes

Before I talk about this, I want to clear up a common point of confusion. The word `origin`, which appears frequently when discussing remotes, is very much like the word `master` – it is the default name for your remote, but it is by no

[4] This is one reason it's a good idea to create a tag of `HEAD` before rebasing, and why you shouldn't rebase over commits that you've pushed to other people.

means special to Git, nor is it any kind of Git terminology. This had me confused for a very long time before I managed to figure out how this stuff worked.

Anyway, here's the deal with remotes. Git lets you define shorthands to remote repositories, whether they're accessed over SSH, HTTP, or are on the local filesystem.[5] Let's talk about what `git push` and `git pull` do, and then I'll go into how to make it easy with remotes and tracking.

4.1 `git push`

`git push` copies commits and branches from your repo to someone else's. The syntax is unintuitive but workable. Here are some examples:[6]

- `git push /path/to/a/repo master:master`

  Merge our `master` branch into another local repo. `master:master` specifies that we're pushing from our `master` branch (the first `master`) into the repo's `master` branch (the second one). Note that this will probably complain, because it's likely that the destination repo has a checked-out working directory. Git tries to avoid the case where the working directory mysteriously becomes out-of-sync with the `HEAD` commit because other people are pushing to your branch.

- `git push . commit-id:refs/heads/branch-name`

  Create a new branch from an old commit, all on this repo. This is exactly the same as `git checkout -b branch-name commit-id`, except that the new branch won't be checked out. We need the `refs/heads/` on the beginning of the remote branch to indicate that we're trying to create a new branch, not mistakenly referring to a nonexistent branch.

- `git push username@host:path/to/repo master:master`

  Merge our `master` branch into someone else's repo over SSH. This works just like the first example; the only difference is that the remote repo is accessed via SSH rather than through the filesystem.

In each of these cases, the first argument to `git push` is called a *remote*. Generally you don't actually specify a remote when you push; you configure the remote for each branch and let Git fill it in. Section 4.2 describes how this works.

4.2 Adding a remote

Git has an interface for dealing with the very common case that you've got two repos and will sometimes synchronize changes between them. The easiest way

[5] Each of these has a protocol wrapper to make sure things get transferred correctly, but ultimately it treats the push and pull operations the same regardless of how you're connecting to a repo.

[6] You probably won't end up using these examples for most things. I'm just going over the low-level stuff to illustrate how these commands work.

to get this set up is to use `git clone` from one to another, but that covers up what's actually going on. Let's use the very common Github configuration as an example for how to configure a remote.

When you create a repo on Github, it contains instructions for setting up your local repo to push to it. They advise that you do the following:

```
$ git remote add origin git@github.com:yourname/reponame.git
$ git push origin master
```

The first command just establishes an alias for the remote `git@github.com:yourname/reponame.git` so that you don't have to type it out each time. The following `git push` says to push our `master` branch into this remote. (Notice that we didn't specify a destination branch; Git's default is to use the same branch on both ends.) I like to add a couple of configuration options to the local repo to make `git push` and `git pull` easier:

```
$ git config branch.master.remote origin
$ git config branch.master.merge refs/heads/master
```

These are the two parameters that we were specifying earlier. Basically, `branch.master.remote` is the remote to use by default when using `git push`.[7] `branch.master.merge` specifies the remote branch that the local `master` corresponds to. Once you have these options in place, you can just run `git push` and `git pull` without specifying either the remote or the branch and Git will do the right thing.

4.3 Using your local repo as a remote

This sounds weird, but it's actually really handy sometimes. Let's suppose you're working on a project hosted on Github, and you want to fork the project in a fairly major way without subjecting your work-in-progress to the scrutiny of your extensive Github fan base. At the same time, though, you want to keep your fork up-to-date with any fixes you make on `master`. The easiest way to do all of this is to create a local branch whose remote is your local repo, and that merges into your `master` branch:

```
$ git checkout -b my-fork
Switched to a new branch 'my-fork'
$ git config branch.my-fork.remote .
$ git config branch.my-fork.merge refs/heads/master
$ git pull        # merges from local master
$ git push        # merges into local master
```

[7] Notice that it's a per-branch thing; this is useful sometimes, especially when you want to create a local fork that stays in sync with your local `master` or some such. Then `git push` and `git pull` can use your local repo, called "`.`", as the remote. See section 4.3 for more about this.

As you're making small fixes on `master` you can merge them into `my-fork` by running `git pull` from `my-fork`. When you've got `my-fork` working properly, you can `git push` to merge back into `master`. Then you can checkout `master` and `git push` to push all of those changes to Github.[8]

[8] If you're going to be doing much of this type of thing, it's really helpful to have a shell prompt that shows you which branch you're on. I've merged into `master` by accident more than once by forgetting that I had left the repository on my fork branch.