Git in Ten Minutes

Spencer Tipping

February 17, 2012

Contents

1 Commits, branches, and tags

The working directory is probably the single most confusing thing about Git. In fact, it's so confusing that I'm not even going to talk about Git in this section. Instead, I'm going to talk about files and text editors because we all understand how those work.

Suppose you open a file in Vim.[1] Under the hood, Vim loads the file's contents into memory and displays them to you. You then have the freedom to do whatever the heck you want to do to the buffer and nothing changes on disk.

So you're editing along and you think, "it would be nice to keep track of intermediate versions of this file so that I can track regressions and stuff." To solve this problem, you devise the following scheme: Each time you write the file, you copy it to a unique file. (Naturally, you do this with a Vim `autocmd`, not by hand.) To make absolutely sure the files are unique, you name each one after the md5sum of its contents. Here's what your directory ends up looking like:[2]

```
$ ls -A
.23971fdc08e2cc4d24464fc1c99844b4
.a6be517f0e6cacb0978fd7361c5e1cf4
.aa73751a6010b18d41bf67db1eb4d425
my-file
$
```

[1] Or Emacs if you insist.

[2] The files are hidden here because otherwise we'd lose track of the ones we actually care about.

And before too long, you realize that `my-file` could really just be a symlink, since its contents are always identical to the thing you've just copied it to. So every time you save the file, you re-link it to the md5-named snapshot that has the right contents. Taking this idea to the max, you then start bookmarking things like releases by making more symlinks to the md5-named files:

```
$ ls -lA
... .23971fdc08e2cc4d24464fc1c99844b4
... .a6be517f0e6cacb0978fd7361c5e1cf4
... .aa73751a6010b18d41bf67db1eb4d425
... my-file -> .aa73751a6010b18d41bf67db1eb4d425
... version1 -> .23971fdc08e2cc4d24464fc1c99844b4
$
```

Later on you decide to fork `my-file` to try a new idea. Rather than copying anything, you just create a new link called `my-file-2` that points to the same thing that `my-file` does. Every time you save `my-file-2`, it will be re-linked to a new unique md5-file, so you won't change the state of the original `my-file`.

2 Using Git

The previous section set up the mental model for using Git. Here's how it works in practice. Below are two representations of the same workflow. On the left I'm using the file representation I described in the last section, and on the right I'm using the analogous Git commands.

```
$ mkdir project && cd project       $ git init

$ vim master                        $ vim contents          # hack hack hack
  insert contents                   $ vim other-contents    # hack hack hack
  insert other-contents             $ git add contents other-contents
  :w   (save)                       $ git commit

$ ls                                $ git branch
master                              * master

$ cp -a master version-1            $ git tag version-1

$ cp -a master experiment           $ git branch experiment

$ ls                                $ git branch
experiment                            experiment
master                              * master

$ vim master                        $ vim contents          # hack hack hack
  do stuff                          $ vim other-contents    # hack hack hack
```

```
  :e   (reload, discard changes)    $ git reset --hard master

$ vim experiment                    $ git checkout experiment
```

Hopefully this makes a little bit of sense. Here's what each Git command does:

`git init` Creates a new empty repository without any commits. This is just like the state we were in with an empty directory, but having already started `vim master`.

`git add [files]` Adds files to Git's index. The index is kind of like vim's swap file, but the analogy starts to break down here. It's basically an intermediate step between the working directory and a commit.

`git commit` Turns the index into a commit object, writes the commit (whose name is a SHA of its contents), and updates the `master` branch to point to the new commit. Just like in our file representation, a branch is a pointer to a commit but does not itself contain any data.

`git branch` Lists the branches in the repository.

`git tag [name]` Creates a tag. This is exactly like a branch, except that the tag doesn't change when you make commits. The similarity is more obvious in the filesystem representation; branching and tagging are done the same way. The difference between the two is how they are treated after they're created.

`git branch [name]` Creates a new branch, but doesn't switch to it. The notion of a "current branch" is sort of like Vim's notion of "the file I'm editing," though I didn't model it precisely in the example above.

`git reset --hard [thing]` Resets the working directory and the index to a previously committed state. This is exactly like blowing away the edit buffer and reloading a file. There are other ways to reset that don't impact the working directory; I recommend reading the man page for `git-reset`.

`git checkout [thing]` Resets the working directory and the index to a previously committed state. The difference between this and `git reset` is that `git checkout` is analogous to exiting the editor and loading another file, whereas `git reset` is like clearing the buffer, reading another file's contents into it, but not telling your editor that you've reloaded.

The `[thing]` notation is probably a bit off-putting. The reason I wrote it that way was to emphasize that you can use different kinds of objects with most Git commands. When you run Vim on a file, it will follow any symlinks to get to the actual data. Git does something similar; as a consequence, you can tell it to `checkout` or `reset` to any commit (as a commit ID), branch name, or tag name.