Yet another Git tutorial

Ben Klemens

8 November 2009

Git is a revision control system, meaning that it is designed to keep track of the many different versions of a project as it develops, such as the stages in the development of a book, a tortured love letter, or a program.

Here's a typical story: you begin chapter one, and commit it to the repository. Your coauthor is working on chapter two, and does the same. Tomorrow, you pull out the current version of the repository, and both chapters are now on hand for you to revise. Later, your coauthor calls and tells you that she was robbed and used her laptop to block a bullet, and you reassure her that it's no problem, because the draft is safely stored in the repository. She gets a new laptop, checks out the current state of the project, and is back to revising your and her work so far.

I typically put even small solo projects under revision control, because it makes me a better writer/coder: I'm more confident deleting things when I know that they're safely stored should I want them back. Git makes this easy, as you'll see below.

Git's history is relevant to its use. It was written by the guy who originally wrote Linux, Linus Torvalds, with the intent of supporting Linux development. Linus is a communist in the best possible way, and thus pushes Git at a means of easy collaboration among equals. So if you like the idea of collaborative development, and especially if you're a computer geek, then Git will enthuse you.

To me, the most interesting thing about Git is just how many tutorials there are about it. It's a complex system, and people have interesting reactions to it. Some tutorials break through the complexity by just giving *to do this, type this* instructions; others get enthusiastic and effuse about the clever structure of the system more than showing you how to use it. For my purposes, I need something to explain the underlying concepts, and not condescend to the reader, but not confuse the story with all the details that are basically irrelevant to those who don't need to create clean patches (or know what a patch is) and don't see a need to cryptograpically sign our book chapters.

I will assume that you are familiar with the basics of POSIX directories, files, text editing, and the usual basic commands like `cp`, `mv`, `rm`, and so on.

The structure A revision control system does two difficult tasks: it has to organize a pile of slightly different versions of a project, and it has to merge together revisions, say when you and a pal are separately working on the same project and finally come together.

That first part is already enough to get lost: you have the version of the file before you, the fifteen earlier versions you wrote, and the twenty earlier versions that your colleague wrote. As noted above, the modern trend is toward distributed version control, meaning that you may have a repository in your home directory, and there may be a remote repository, and all those repositories have some number of versions. All this multiplicity is great because every little change is tracked, history can be interrogated and compared, and completely screwing up your local repository just means you have to start over with a new copy.

But all this multiplicity means that you need to know the address of where you are and where you want to go—a typical simple setup, like the one in Figure **??**, is already overwhelming if you don't know how to get around. So, we will begin with an overview of the repository-branch-version-file hierarchy that you will be navigating.

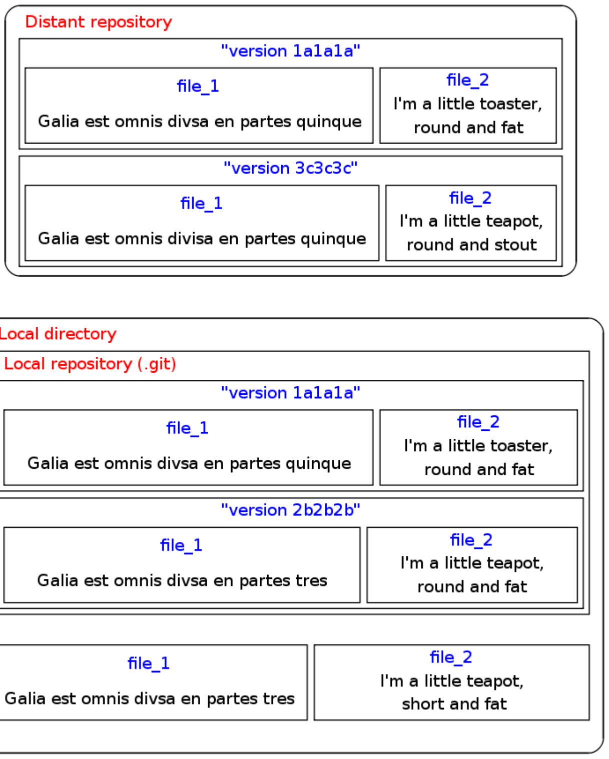
Distant repository
"version 1a1a1a"
file_1
Galia est omnis divsa en partes quinque
file_2
I'm a little toaster, round and fat
"version 3c3c3c"
file_1
Galia est omnis divisa en partes quinque
file_2
I'm a little teapot, round and stout
Local directory
Local repository (.git)
"version 1a1a1a"
file_1
Galia est omnis divsa en partes quinque
file_2
I'm a little toaster, round and fat
"version 2b2b2b"
file_1
Galia est omnis divsa en partes tres
file_2
I'm a little teapot, round and fat
file_1
Galia est omnis divsa en partes tres
file_2
I'm a little teapot, short and fat

Figure 1: ¡Multiplicity! Here, there are three places where a version of the project could be: a distant repository, in your local directory in a repository named `.git`, or checked out in your local working directory. In the story so far, there are four versions: `1a1a1a`, which is present in both the local and remote repositories; `2b2b2b`, where you reallocated Gaul and became a teapot; `3c3c3c`, where someone else fixed a typo in *divsa* and checked it in to the distant repository; and the version you are looking at in your current directory.

- There may be several *repositories*.
- Each repository holds several *branches.*
- Each branch holds several *versions* (aka *commit objects*).
- At any one time, you are looking at exactly one version of the project, and the several files that comprise that version.

The repository The repository is a pile of versions, in a binary format that Git can read and you can't. Here is how you would create a new one, in a given directory that will be the repository from then on:

```
mkdir new_project
cd new_project
git init
```

You now have a blank repository. [If you are putting an existing project under revision control, you will need to add existing files to the index with `git add .`; see below.] Git stores all its files in a directory named `.git`, where the dot means that all the usual utilities like `ls` will take it to be hidden; after the init step, you can look for it via, e.g., `ls -a`.

This is the first embodiment of Linus's egalitarian communism: he wanted to make it easy to create lots of repositories on lots of machines. The key to this is that the `.git` subdirectory holds all the information, so copying your project directory (including the `.git` subdirectory) generates a new, entire respository. If you want to back up your project and repository, just recursively copy your project directory:

```
cp -r new_project project_bkup
```

Versions Now create a new file for yourself, using your favorite text editor or what-have-you.[1] Git doesn't know about this file yet; you have to tell the machine about it using the index (discussed below). For now, try

```
git add .
```

to add to Git's index everything in the current working directory (in UNIX-speak, `.`).

Now you can save your work to the repository in your working directory, or in revision control jargon, commit your changes:

```
git commit -a -m "What I've been doing up until this check-in."
```

You can make more changes to your text file, and then re-run `git commit -a -m "..."` as often as you wish, thus creating a history of commits that you'll be able

[1]Git, like every revision control system I know, works best around the POSIX paradigm of relatively short lines of text. Computer code naturally looks like this, as does typical human-written plain text (with linewrap enabled so you don't have one line per paragraph). If you're using a binary format like a word processor document, then Git will have trouble making meaningful merges, and you're basically stuck using whatever revision control the word processor vendor gave you (if any).

to refer to below. Notice that once you've put a file into the index, you don't need to re-run `git add ...`.[2]

I've been avoiding the term *commit objects* as being a little too jargon, but the jargon does get across the idea of a single committed blob, which is treated as a unit for our purposes. You can use `git log` to get the list of commits in your history. The log shows two relevant pieces of information: the 40-digit SHA1 cryptographic hash, and the human-language message you wrote when you did the commit. The SHA1 hash is a computer-scientist clever means of solving several problems, and is the name you will use for the commit. Fortunately, you need only as much of it as will uniquely identify your commit. So if you want to check out revision #fe9c49cddac5150dc974de1f7248a1c5e3b33e89, you can do so with

```
git checkout fe9c
```

With that command, you've gone back in time, and have in your current directory whatever you had back then. Take notes, copy off that paragraph you wish you hadn't deleted, then

```
git checkout master
```

to return to the head of the master branch (which is where you started off, being that we haven't discussed creating new branches yet).

[I suggested that you take notes and make observations, but not that you change anything. ¿What would happen if you were to build a time machine, go back to before you were born, and kill your parents? If we learned anything from science fiction, it's that if we change history, the present doesn't change, but a new alternate history splinters off. So if you check out an old version, make changes, and check in the changed version, then you've created your first branch off of the master branch.[3]]

Git is designed to make it as easy as possible to bounce back to an alternate version and bounce back to where you were, as often as you need. But this may not be ideal, because you may want to have both versions living side-by-side. The easiest way to do this is to just make yourself a second repository.

```
cp -R /path/to/maindir new_tempdir
cd new_tempdir
git checkout fe9c
```

Now you can do side-by-side comparisons between the main version in the main repository and your disposable copy. [There's also a `git clone` command, see below, that does about the same as this in a slightly slicker manner.]

[2] Much of git's advanced technique is about rewriting the history to produce a smoother course of events. This document makes a point of not worrying about the history, but I will mention one nice feature to keep your log from filling up with small commits. After you commit, you will almost certainly slap your forehead and realize somthing you forgot. Instead of just doing another commit, you can do `git commit --amend -a` to add to your last commit.

[3] You haven't named your branch, which can create problems. Getting ahead of the story, if you ever call `git branch` and find that you are on `(no branch)` then you can run `git branch -m new_branch_name` to name the branch you've just splintered off to.

I've found the ability to quickly jump around in time to be immense fun, and has made me a more confident editor. When in doubt, I make the cut, and know that the worst punishment for an error is the small bother of checking out an old version.[4]

The index versus your files We've looked at prior check-ins; now ¿what will the *next* check-in look like? Git maintains what is called the index, which is the nascent list of files that will become a commit object when you next call `git commit`. That index is not identical to the files you see when you do a directory listing, for a few reasons. First, many systems produce annoying files like log files, object code, and other mid-processing cruft, and you don't want those taking up space in the repository. There are also advanced commit strategies wherein you may change several files, but want to save only the changes you'd made in one or two.

Regardless of the rationale, bear in mind that the working directory you are looking at is probably a mix of files Git is tracking and files Git doesn't care about. If you want to add a new file to the repository, remove an old one, or fix the name of a file, you'll need to do one of:

```
git add newfile
git rm oldfile
git mv flie file
```

so that the change is evident both in the working directory and in the index. As for files that you are modifying but are not shuffling around in the filesystem, you technically have to add those one by one as well by running `git add modified_file` with every single commit, but the `-a` in the command `git commit -a` tells git to save all modifications on known files (including removal), thus saving you all that tedium. In my own work, I have never encountered a reason to commit without the `-a` flag, but perhaps you may one day run across something.

But do remember that `git add, rm, mv` only change the index of what the next commit will look like. The commit won't actually happen until you say so with `git commit -a`.

Branches To this point, you have been alternating between doing work and saving it via `git commit -a`, thus producing a sequence of committed versions of your program. That's a branch.

Perhaps *thread* would be a better term, being that this represents a single thread of your work conversation. By default, you are on a branch named *master*. Other threads come up in two manners: your own work may digress, or you may have colleagues who are following their own threads. The typical story in your own work would be that you are trying something speculative, which may or may not work. By creating a new branch, you are ensuring that you have something stable in the master branch at all times; you can merge the experimental branch back into the master thread later if all works out (where merging will be covered below).

¿What branch are you on right now? Find out with

[4]See also, e.g., `git show fe9c:oldfile` to just display a single file from an old version without doing a full checkout.

```
git branch
```

which will list all branches and put a * by the one that is currently active.

Now create a new branch. There are two ways to do it:

```
git branch new_leaf
git checkout new_leaf

#or equivalently:

git checkout -b new_leaf
```

There are really two steps here: establishing a new branch in the repository, then changing your working version to make use of that branch. The two-command version makes that explicit; the single `checkout -b` form is provided because it's so common to want to immediately use the branch that you are creating.

Having built a new branch, you can switch between branches easily. E.g., to switch back to the master branch:

```
git checkout master
```

You can see that the branch checkouts, like `git checkout newleaf` or `git checkout master`, have the same syntax as that infernal SHA1 syntax like `git checkout fe9c`. The reason is that the name of the branch is really just a synonym for the SHA1 hash that is the last item on that branch (aka the *branch head*). Use them interchangeably, though I'm guessing you'll lean toward using the branch name.

Merging To this point, everything has been about creating new versions, and jumping around between versions. Now for the hard part: you have a version, your colleague has a version, and they differ.

The command is simple enough. You have on your screen a current version, and you want to fold in the revisions from version fe9f. Then

```
git merge fe9f
```

will do the work. Of course, you can use a branch name as well, like `git merge new_leaf`.

For some things, the system will have no problem merging together the two threads: if your coauthor was working only on the intro to chapter three and you were working only on chapter three's conclusion, that's easy to merge.

But if you were both wrestling with the same paragraph, then the computer will be confused. It will tell you that there are conflicts, and write both into the file in your current directory. You will then have to open the file(s) in your text editor, and find the place where git wrote both versions for you to compare and choose from.

If you were to merge revision `3c3c3c` from the remote repository into the current working revision (i.e. the head), then `file2` would probably wind up looking something like this:

```
I'm a little teapot,
<<<<<<< HEAD
short and stout
=======
round and fat
>>>>>>> 3c3c3c
```

Here, Git finds a single line with two different versions, and it can't rely on timing or other heuristics to pick one. So, it shows you the two versions, and it is up to you as a human to decide what to do. The solution will often require human contact with another author (IM is a perfect medium for this). Git can't call your coauthor, but it can at least point you to the exact line where differences exist.

The other type of conflict, which is just annoying, is when your colleague has renamed a file or moved it from one directory to another. Git typically won't just move the darn file for you, but will instead list it as a conflict for you to deal with. Moving files can create other awkward issues; for example, if you are doing `git pull` from a subdirectory that your coauthor has deleted, you'll get entirely confused errors.

Here is the procedure for committing merges:

1. `git merge a_branch`.
2. Get told that there are conflicts you have to resolve.
3. Check the list of unmerged files using `git status`.
4. Pick a file to manually check on. Open it in a text editor and find the merge-me marks if it is a content conflict; move it into place if it's a file name or file position conflict.
5. `git add your_now_fixed_file`.
6. Repeat steps 3–5 until all unmerged files are checked in.
7. `git commit` to finalize the merge.

I have always found merging to be unnerving. There is a computer modifying your files, without even telling you what it is modifying. Unlike the long list of versions and your endless power to shunt branches, the merge algorithm is more-or-less a black box, and you just have to trust it. In that context, it's a somewhat good thing that the machine sometimes refuses to auto-merge and demands human attention. If the computer does go too far and makes a total mess of things, you can take recourse knowing that you have the previous version safely stowed.

The stash It doesn't take long working with Git to discover that it doesn't like doing anything when there are uncommited changes in the current working directory. It typically asks you to commit your work, and then do the checkout or such that you had intended.

One thing you can do in this case is a variant of the merge routine above: ask `git status` which files are tracked but modified; `git add` those files; then `git` `commit` your changes. Once your working tree, the index, and the latest commit are all in harmony, you can go back to your original plan.

Another sometimes-appropriate alternative is `git reset --hard`, which takes the working directory back to where you had last checked out. If the command sounds severe, it is because you are about to throw away all work you had done since the last checkout.[5]

The other option is the *stash*, a quick-to-use branch, with a few special features, like retaining all the junk in your working directory. Here is the typical procedure:

```
git stash
git checkout newleaf #or another commit, or what-have-you
#do work here
git checkout master #or the branch you had stashed from
git stash pop
```

So this is the above procedure of checking out, doing work, and then returning to the current version, but you stash your in-progress working directory beforehand and pop it back into place afterward.

Popping the stash works by merging the stash's semi-branch back to whatever is currently checked out, which is why you have to check out the commit you had been on before going exploring in the history: doing the merge is trivial if you have checked out the commit that you had been on when you started, and could be a mess if you are elsewhere. [The ability to apply the stash onto a separate commit allows for creative merging strategies which you may find use for if you are feeling clever.]

Remote repositories So far, I've talked only of checking out versions and branches that are in the respository of the directory you are in right now. But you can copy branches across repositories, which is how sharing happens. As alluded above, there can be amusing reasons for cloning a directory to another directory, and then merging changes between them.

To do all this, you need to be able to copy a branch from another repository. And to do that, you will need to name the other repository. Do this via

```
  #for an on-disk remote repository:
git remote add my_copy /path/to/copy

  #and for a distant remote repository:
git remote add distant_version http://...
```

That is, you will give a nickname for the repository, then a locator. There are many options for locators; the odds are good that the maintainer of any given repository handed you a locator to use, so I won't bore you with a list of options here.

[5]By the way, you can reset individual files by just checking them out. I recommend this syntax: `git checkout -- one_file`, after which `one_file` has reverted to its state as of the last checkin, and all changes lost.

If you are joining a project that already has a repository, running `git clone /path/to/copy` will set up a local copy and add a `remote` label of `origin`, set to the location of that parent repository.

A plain `git remote` will give you a list of remote repositories your repository knows about. You will probably just assign one remote and be done with it, but you have the power to live Linus's dream of concurrently passing files among several of your peers' several repositories.

Having established a remote, you also have more branches to choose from: try `git branch -r` to list remote branches (or `git branch -a` to list all branches, local and remote).

There are a few ways to get a remote branch:

```
git checkout -b new_local distant_version/master
 #or
git pull distant_version master
```

The first version uses the plain checkout mechanism using the remote tag you got via `git branch -r`, and uses the `-b` flag to create a name for the new branch you are about to create (otherwise you'd be stuck on `(no branch)`). The `pull` version merges into whatever you are working on now. You are probably getting things from the repository to bring your own work back up-to-date and in sync, so you probably want to use `pull` instead of `checkout`.

The converse is `push`, which you'll use to update the repository with your last commit (not the state of your index or working directory).

```
git push distant_version
```

When somebody had made another commit to the repository while you were working, then you will first need to do another `git pull`, slog through the merging procedure, and then push back the cleaned-up final version. This is common in team projects, and the error you get (about fast-forward merging) is entirely unhelpful.

git help That's all I'm giving you, and it should be enough for you to keep versions of your work, confidently delete things, merge in your colleagues' work, and be able to keep your bearings in Git's repository/branch/version/index system. From there, `git help` and your favorite search engine will teach you a whole lot of ways to doing these things more smoothly, and many of the tricks that I didn't cover here.