Git submodules

Ben Klemens

23 May 2017

So you have two repositories and you want to treat one as a subpart of the other. Maybe you've segregated your project into the general-use library to distribute widely and the specific case that nobody else will care about. Maybe you have one person in your group who, for whatever reason, you want focused on only one subtask. Or your organization deems part of your project to be sensitive but you really want to work from home on the rest of it. I had one big project a few months ago, and thought it would be less unruly to start work on a new part in a separate repository—but then I had the problem of merging that part into the main while retaining the revision history.

By the way, the world (well, 11 people) wanted me to write this:

I found the abundance of the Web to be confusing for two reasons. First, there are ways to do it that are deemed archaic, whose instructions are still online, in reputable sources which haven't been updated yet. Second, there are two distinct threads, using different methods: do you want the subtree to remain its own repository with its own revision control, or do you want it to be tracked by the parent directory? The second option will be covered next time; here's discussion of the first.

1 Separate subdirectory

As a word of background, remember that which git repository you're in is determined by the first directory (self, parent, grandparent, ...) that has a hidden `.git` directory holding all the meta-junk. So if you have one git-controlled directory, and you make a subdirectory `subtree` and run `git init` in that subdirectory, then everything you do in `subtree` is tracked by its own git machinery, not that of the parent directory.

So here's the easiest way to turn a distinct repository into a submodule of your parent repository: leave a note in your parent repository, *Hey, thanks for cloning my repository. Now go run* `git clone http://XXX` *here in the parent repository, and check out revision* `abc123`*. Thx.* I would formalize this note into a makefile; here's an example with discussion to follow:

```
Repository=http://example.com/subrepo.git
Version=abc123

run: sub/somefile.c
	[compilation instructions here]
	[now run]
```

```
sub/somefile.c:
        make init

init:
        git clone $(Repository) sub
        cd sub; git checkout $(Version)
```

The `init` target clones the subrepository and checks out the appropriate version. Because the `run` target depends on a file in the sub, `make init` gets run the first time anybody tries to `make run`. The cloning gives a subdirectory with its own `.git` machinery, so we've basically achieved the goal.

If you're not a makefile user, you surely have your own way of running scripts. It would be nice to have the scripts run automatically; hooks don't really do it because you can't check a hook into the repository such that it'll run when you clone the repository. We'll get another partial automation below.

How would you maintain this? If you need exactly revision `abc123`, and are never going to modify the subtree, then you have few needs and few problems. If you are OK tracking a branch like `master`, whatever state it is in, you have even fewer needs.

But if you are modifying and checking in the sub, now you have two repositories to worry about. First, when you make a change in the sub, you may forget that it's a sub, and commit changes in the parent. You could maybe install a hook to check in the sub when you check in the parent (and you could maybe have `make init` install that hook). I wrote a script to check whether a repository is clean[1] (everything checked in, no detached head, no stashes, ...), which goes into submodules and checks their status as well; see this previous post (entry #194).

Second, once you've checked in the sub, do you need to change the ID in the makefile (or other script) from `Version=abc123` to `Version=def456`? If so, then now you have two check-ins to make: one for the sub you modified, then one to check in the revised makefile.

When you push to the origin repositories after updating the sub and parent's makefile, you'll have to do it in two separate steps. When your colleagues pull the sub, they may have to make sure that they are pulling the right version, or the parent has some way of correctly updating the sub after it gets pulled. If you pushed the parent, now referring to `def456`, and forgot to push that commit to your group's shared repository for the sub, your colleagues are going to to find you and talk to you.

submodule Moving on to formal git tools, there's `git submodule`. It's evidently an emulation of a feature of Subversion. One of the design principles for git was to do the opposite of whatever Subversion did, and it looks like this time they went with what Subversion did and people hated it.

You call the command with the name of a repository and a subdirectory to clone it into, and git clones the repository, and stores the metadata, about the origin repository and which subdirectory has the sub, in a file named `.gitmodules` in the parent's base directory. This file is part of the repository (after you commit), and when you clone a

[1] `https://github.com/b-k/git-isclean/blob/master/git-isclean`

copy with a `.gitmodule` file, you can run `git submodule update --init` to check out the sub as per the stored metadata.

But unlike the makefile, that `.gitmodules` directory only has the paths, not the commit you've checked out. The commit ID, like `abc123`, is stored in the metadata of `.git/modules/...`. That the commit ID is internally stored instead of transparently held in a makefile or `.gitmodules` has pros and cons: you can't directly modify it, you can't set it to an exotic alternative to a commit ID (like, say, `master`), but the submodule system knows when the sub changes commit names, and you can just run `git commit` in the parent to update the internal annotation of the current sub.

[As a digression, let me express one frustration from the submodule suite: `git submodules update` will "update the registered submodules to match what the superproject expects by cloning missing submodules and updating the working tree of the submodules." So it feels to me like it should've been named `git submodule reset`. If you think it updates the metadata the parent has, you're wrong (use `git commit` in the parent); if you think it takes an existing sub and updates it, you're wrong, because it ditches your last commit and pulls the one the parent knows. If you make this mistake (like I did a hundred times), use `git reflog` to look up the ID you need to get back to.]

I gave you the makefile example first because I think it's a useful mental model for how the two independent repositories will work, with a parent that tracks an independent sub using only metadata, which has to be kept up-to-date because there is a certain commit name the parent is tracking that needs to be updated, in plain text for the makefile approach and in the tracked metadata for the submodule system. The submodule version hides this behind some git chrome and a few commands that save you the trouble of typing `mkdir` and `git clone` and so on, but the model is similar enough that the pitfalls are similar: after you modify the sub, the parent is pointing to the wrong commit ID for the sub until you update/commit the parent, and it's up to you to make sure that colleagues can achieve the same sync.

It seems this is as good as it's going to get given the goods git gave us. We can leave some metadata notes in the parent, but it's still up to you and your colleagues to check that everything is in sync. We hate it when something isn't fully automated, but there it is.

Next time, I'll cover the case where you want the parent's commit history to include all the changes in the sub.

But meanwhile, here is a demo script for you to cut/paste onto the command line (probably piece by piece) to see how some work with these things might go. Or, this post[2] has further tricks and syntax notes—scroll down past the pages of caveats to get to the part where the author describes the commands and options.

```
mkdir mod_demo #everything will be in here. Clean up with rm −rf mod_demo
cd mod_demo

# Some admin junk; please ignore
bold_cyan="\033[1;36m"
no_color="\033[0m"
Divider="${bold_cyan}${no_color}"
```

[2]`https://medium.com/@porteneuve/mastering-git-submodules-34c65e940407`

```
alias Print='echo −e \\n$Divider $∗'
# Thanks for your patience

Print "This script demonstrates using git submodule to transfer some changes around
∗ create submod and larger_work repositories
∗ add the sub as a submodule to larger_work
∗ clone larger_work to lw2
∗ make changes to the sub in lw2, push those to the submod repository
∗ go back to larger_work and try to recover those changes from lw2."
Print


Print Generate a repository that will be the submodule, with two files.

mkdir submod; cd submod
git init
echo "This repository provides a system to analyze the contents of a directory on a POSIX
        filesystem" > Readme
echo "ls" > directory_analyze
chmod +x directory_analyze

git add ∗
git commit −a −m "Set up submodule"

# Others can't push to a branch you have checked out, so switch to a fake branch
git checkout −b xx


Print Now set up the parent module. No subs yet.
cd ..
mkdir larger_work; cd larger_work
git init
echo "Find information about a file. Usage: info yr_file" > Readme
echo 'fs_analyze/directory_analyze | grep $1' > info
chmod +x info

git add ∗
git commit −a −m "Set up parent"


Print "After 'git sumbodule add', .gitmodules now exists, and has:"
git submodule add ../submod fs_analyze −b master
cat .gitmodules

Print "But its status is tracked as new, but not yet committed:"
git status

Print "So, commit it."
git add .gitmodules; git commit −m "add .gitmodules"
git status
```

```
Print ”Let’s clone the parent and see what shows up”
cd ..
git clone larger_work lw2
cd lw2; ls −a

Print ”ls fs_analyze turns up a blank...”
cd fs_analyze; ls

Print ”...until we init/update.”
git submodule update −−init
ls −a

Print ”Detached HEAD.”
git branch
Print ”Change to master”
git checkout master
git branch

Print Make changes and push to origin
echo ”ls −l” > directory_analyze
git commit −a −m ”More powerful directory analysis technology”
git push origin master


Print The parent knows the sub has changed commits
cd ..
git status

Print Now tell the parent that the sub is on a new commit
git commit −a −m ”Updated sub”
git submodule status


Print Great, our cloned sub has moved forward. Back to the first, pull the changes:
cd ../larger_work
git remote add lw2 ../lw2
git pull lw2 master
git submodule status

Print No submodule movement. Do it manually via git pull:
cd fs_analyze
git checkout master
git pull
cd ..
git commit −a −m ”update recorded sub commit”
git submodule status
```