Git subtrees

Ben Klemens

24 May 2017

This continues the last entry about submodules (entry #198). The two setups are not interchangeable: in the setup there, the repositories are tracked entirely separately, and you won't be able to read changes in the submodule in the parent's commit history. In the subtree setup here, your changes will appear in the parent's history, but we'll have a script that extricates the subtree-only changes should we need to push them back to the source of this subpart.

Prefixes Here's another feature of git you might not have known about: `git read-tree`. It is normally used in the background as part of the merge process to read the tree of files in a commit and merge them with what's in your directory at the moment. But the cool part is that you can specify a `--prefix` giving the name of a subdirectory to read into. If you have a side-branch in your extant repository that you want to compare with what you have checked out right now, you could use `mkdir tmp; git read-tree otherbranch --prefix=tmp -u` to put the contents of `otherbranch` in a subdirectory for your side-by-side comparison to the main project. [Without the `-u`, the subtree is put in the index but not the working dir, which makes sense for the original purpose of merging trees; with the `-u` it appears in the working directory as well. If you just want the tree in the working directory, do a `git reset` to clear out the index after the read-tree step.]

So you have another repository elsewhere. You could make it a side-branch by adding a remote (`git add remotesub http://xxx`), and then `git fetch`ing all the remote branches. Unlike typical branches, these fetched branches probably have nothing in common with `master` in your main project beyond the initial null commit. But now that the subproject lives in a side-branch, you can `git read-tree --prefix=...` the files into a subdirectory as above, and after progressing, merge changes from the subtree back to the side-branch. So the three-part flow of data is subtree ¡—¿ branch ¡—¿ origin repository.

You'll see the `--prefix` option in a lot of manual pages, if you look for it. For merge, specify the subtree via a *subtree merge strategy*, like `git merge -s subtree=thatsubdir mergetarget`, though calling it a merge strategy seems like a misnomer to me.

I got this from the git book[1], which takes you through the details and flags.

With this prefix business, your checkout is recorded by the parent repository itself—there's no subsidiary `.git` directory. Check-ins will continue to affect the parent, un-

[1] `https://git-scm.com/book/en/v1/Git-Tools-Subtree-Merging`

less you want to get really fancy about only pushing some check-ins to the side-branch and removing them from the parent's history.

Where's the metadata telling you that you built the subtree from a side-branch? It isn't there. If your side-branch is pulled from a remote origin, it knows where that is, as usual. But when you use `--prefix` to dump the side-branch's contents to a subdirectory, no annotation is made that this subdirectory is in any way special, and it's up to you to remember where it came from. Of course, you could make another script or makefile target to pull and push between the subdirectory and the repository branch.

git subtree On to the `git subtree` command. It does all of the above, with a little more intelligence. When you pull a distant repository into a subdirectory, all the commits are replayed as part of the main repository, as if you had been working in that subdirectory all along. You will check in new commits to the parent repository, as a unified whole. But if you want to push back to the subproject's origin repository, `git subtree` will go commit by commit and push only the subtree-relevant parts of each (skipping those commits that didn't touch the subtree). This is a nice bit of work none of us want to replicate, but at the core of the script, it's using `git read-tree --prefix=...` and `git merge -s subtree=...`, just as we did manually above.

Where's the metadata? Some of it is in the log. Here's the machine-oriented log entry I got when I first added a subtree to one of my projects via `git subtree add ...`:

```
Add 'pbox_proof/' from commit '7d2895ab...7f2a5e76'

    git-subtree-dir: pbox_proof
    git-subtree-mainline: 14e46e69...1037f3
    git-subtree-split: 7d2895ab0...77f2a5e76
```

The mainline and split references will be used by the subtree script internals, but you can see the name of the directory where the sub lives (twice, even), and you can easily grep for it. If you're going to be a heavy subtree user, the Internet recommends adding this to your `.gitconfig` in your home directory:

```
[alias]
    ls-subtrees = !"git log | grep git-subtree-dir | awk '{ print $2 }'"
```

With this, `git ls-subtrees` finds each marker in the log and prints the name. Of course, I have to know to do this, and if you gave this repository to a colleague, they will too. Also, we're missing the metadata about where the origin is that we cloned from, which you'll again have to have in a makefile or readme (or you can clone to a side-branch and then push/pull from the side branch as above, but that seems like more effort than it's worth).

For all the details, the usual man page at `man git-subtree` can give you the switches for `add`, `merge`, `push`, `pull`. [Split is cool, but I take it to be internals for merge/-push/pull.]

If we weren't the sort of people who use git, both the subtree and submodule setups would seem to be sufficient: you have all the tools you need to treat a subdirectory as a repository, either with its own `.git` branch or as a part of the parent's commit history, that could in either case be pushed to its origin as needed. But you have to know that the subtree is special and communicate how to handle it to colleagues, human to human. This sort of un-automation is incongruous to how we, the sort of people who take the trouble to read blogs about git, want things to work. We can partially solve the problem with post-tests like the `git isclean` script I mentioned last time or pre-commit hooks, but the efficient solution to the many problems discussed above and elsewhere may be to just add a readme file.

The demo script Similar to last time, here's a demo script to cut and paste (probably section by section) to your command line. It builds a parent and sub repository, grafts the sub into the parent as a subtree, and pushes some changes made in the consolidated project back to the sub. Most of it is setup, identical to last time; the subtree action happens after the commit with the message *"Set up parent"*.

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

# Some admin junk; please ignore
bold_cyan="\033[1;36m"
no_color="\033[0m"
Divider="${bold_cyan}${no_color}"
alias Print='echo −e \\n$Divider $∗'
# Thanks for your patience

Print "This script demonstrates using git subtree to transfer some changes around
∗ create submod and larger_work repositories. This matches the setup from last time with
    submodules.
∗ add the sub as a subtree to larger_work
∗ make changes to the subtree
∗ push those to the subtree repository"
Print


Print Generate a repository that will be the subtree, 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”

git subtree add -P fs_analyze ../submod master

Print Here is the log entry about the addition:
git log |head

Print Modify both the parent and child here; commit

echo ”ls -l” > fs_analyze/directory_analyze
echo ”Find information about a file. Usage: info yr_file.” > Readme


Print Here are the changes to commit:
git diff
git commit -a -m ”Changes made”

Print Push to the original submodule repository
git subtree push -P fs_analyze ../submod master

Print ”Go to the sub; see what change(s) got recorded in the sub”
cd ../submod
git checkout master
git diff HEAD~
```