Mundane git commit-tree tricks, Part 5: Squashing without git rebase

devblogs.microsoft.com/oldnewthing/20190510-00

May 10, 2019

Raymond Chen

Suppose you've made a bunch of changes.

		X WIP		almost		rename method		silly typo		X works		update Y		upc Z
base	←	A	←	B	←	C	←	D	←	E	←	F	←	G

You started by trying to get the *X* component working. I subscribe to the theory of *commit early and commit often*. I don't wait until all of *X* is done before committing. I'll commit every time I reach a point where I have built up enough work that I don't want to lose it, especially if I might wind up breaking it in the next stage of work. Think of it as *save game* for source code.

After four tries, you finally got component *X* working. Next step is to update components *Y* and *Z* to use the new component.

Okay, you're ready to create your pull request. Now, a pull request is a story, so you need to decide how you want to tell the story of your work to others, so that they can review it. For this story, we want to say "First, I wrote this awesome bug-free *X* component. Then I updated the *Y* component to use the *X* component. Finally, I did the same with *Z*." To tell this story, we want to do some internal squashing.

		X WIP		almost		rename method		silly typo		X works		update Y		upc Z
base	←	A	←	B	←	C	←	D	←	E	←	F	←	G
	↖													
		AE′									←	F′	←	G′

The conventional way to do this is to check out the branch and perform an interactive rebase, squashing together commits *A* through *E* to form a new commit *AE′*, and then picking commits *F* and *G*, producing *F′* and *G′*.

However, the conventional way may not be the convenient way. You may have moved on and checked out a different branch to to some other work, and returning to this branch for some squashing action would churn your working directory, forcing unwanted rebuilds.

Or you might still be on that branch, but rewinding back to *base* is going to churn so many files that it will invalidate all the build collateral you've created, forcing you to rebuild them pointlessly. For example, part of the work in adding the *X* component may have involved changing a centrally-used header file, which means that your entire project will have to rebuild.

Since all of the commits we want to squash are consecutive, we can do all this squashing by simply committing trees.

```
git commit-tree E^{tree} -p base -m "Write component X"
```

Note: As before, if using the Windows `CMD` command prompt, you need to double the `^` character because it is the `CMD` escape character.

This command prints out a hash, which is our *AE′*.

Now we can stack *F* and *G* on top of it:

```
git commit-tree F^{tree} -p AE' -m "Update Y"
```

This prints a hash, which is our *F′*.

```
git commit-tree G^{tree} -p F' -m "Update Z"
```

This prints a hash, which is our *G′*.

We can now reset the local branch to that commit, and then push it.

If the branch you are "virtually rebasing" is the current branch, you can reset to it.

```
git reset --soft G'
```

Since the trees for *G* and *G′* are identical, this has no effect on your index. Any files that were staged remain staged, with exactly the same changes.

If you are virtually rebasing a non-checked-out branch, then you can update it, and even push it, without checking it out:

```
git branch -f that-branch G'
git push -f origin that-branch
```

Or we could bypass our local branch and push directly to the remote.

```
git push -f origin G':that-branch
```

The point is that we were able to rewrite a branch without touching any files in the working directory.

Raymond Chen

Follow