Mundane git commit-tree tricks, Part 6: Resetting by reusing an earlier tree

devblogs.microsoft.com/oldnewthing/20190513-00

May 13, 2019

Raymond Chen

Suppose you have a branch in which you have been doing a bunch of work, you've been merging regularly from the main branch to stay up to date, and you realize that your work should be abandoned, and the branch reset to a state as if it had been freshly-created from the main branch.

For most people, that would mean simply abandoning the branch and creating a new one.

Unfortunately, Windows hasn't quite reached the point where it can use trunk-based development. Thousands of developers working in a branch with three million files means that there would be commits going into the main branch pretty much continuously. Instead, Windows uses a variant of the dictator-lieutenant workflow. Setting up a new lieutenant involves a lot of paperwork,[1] and teams often want to avoid that paperwork by finding an existing no-longer-needed lieutenant and giving it a new purpose.

When you do that, you want to clean out any changes from the lieutenant that were part of its former purpose, so that the new purpose gets a fresh start, as if it were using a branch new lieutenant.

This is where `git commit-tree` once again comes in handy.

$$\begin{tikzcd}
M1 & M2 \arrow[l] & M3 \arrow[l] & M4 \arrow[l] & M5 \arrow[l] \\
 & L1 \arrow[ul] & L2 \arrow[l] & L3 \arrow[l] \arrow[ul] & L4 \arrow[l]
\end{tikzcd}$$

Suppose this is the state of the project at the time the team decides to repurpose its lieutenant. It had been doing some work in that branch (*L1* through *L4*) that it wants to abandon and pretend never happened.

Windows has a policy that official branches may not rewrite history,[2] so a hard-reset is not permitted.

Find the most recent commit in the main branch which has been merged into the lieutenant branch. You can look into the lieutenant's record keeping, or the `git merge-base` command will tell you. In our case, the commit is *M3*.

You can now wipe out all of the work done in commits *L1* through *L4* by committing the tree that matches the commit you most recently merged from the main branch.

```
git commit-tree M3^{tree} -p L4 -m "Reset to M3"
```

This prints a hash that you can fast-forward to.

```
git merge --ff-only ⟨hash⟩
```

This tree-based merge is the trick I referred to some time ago in the context of forcing a patch branch to look like a nop. In that diagram, we started with a commit *A* and cherry-picked a patch *P* so we could use it to patch the master branch. Meanwhile, we also want a nop to go into the feature branch. We did it with a `git revert`, but you can also do it in a no-touch way by committing trees.

```
git commit-tree A^{tree} -p P -m "Revert P"
```

By doing it this way, you are guaranteed that the trees *A* and *P2* are absolutely identical, because we created them that way.

Note that in both of these cases, if you are already checked out to the branch you want to roll back, you can use a different command sequence:

```
git read-tree ⟨hash⟩
git commit -m "Reset to ⟨hash⟩"
```

An alternative that uses more conventional commands (but which temporarily moves your `HEAD`):

```
git reset --hard ⟨hash⟩
git reset --soft @{1}
git commit -m "Reset to ⟨hash⟩"
```

However you manage to do it, once you get your branch reset, you can submit a pull request back to the main branch with your freshly-reset lieutenant. This should result in a nop change (no files changed), with a payload consisting of *L1* through *L4*, plus a massive commit of that wipes out all the custom changes.

Merging back to the main branch serves a few purposes:

- The emptiness of the pull request validates that your lieutenant branch has been properly reset.
- Payload tracking tools will report that your (now-abandoned) *L1* through *L4* payload has reached the main branch. Putting them all inside a “nothing happened” pull request makes it easier to prove that those changes were indeed abandoned.
- Your next pull request from the lieutenant to the main branch will consist solely of the new work.

[1] Part of the reason for the paperwork is that a lieutenant doesn’t get just a branch. A lieutenant also gets build resources, artifact retention policy, automated testing resources, a feature flag environment, and lots of other goodies. Besides, you need to know some basic information, like who to contact if there is a problem with their branch.

[2] I suspect many organizations have similar policies, You need to be able to recover the source code that produced any particular build. You also have to be able to identify all the changes that went into a particular build (say, when investigating a regression). Rewriting history undermines those principles.

Raymond Chen

Follow