Mundane git commit-tree tricks, Part 3: Building a throwaway commit in order to perform a combined cherry-pick-squash

devblogs.microsoft.com/oldnewthing/20190508-00

May 8, 2019

Raymond Chen

Suppose you have a series of commits you want to cherry-pick and squash onto another branch.

The traditional way of doing this is to cherry-pick the series of commits with the `-n` option so that they all stack on top of each other, and then perform a big `git commit` when you're done. However, this mechanism churns the local hard drive with copies of each of the intermediate commits, and if there are merge conflicts, you may end up having to resolve the conflict in the same block of code over and over again.

For example, suppose that you want to cherry-pick a series of commits that look like this:

base ← A ← B ← revert A ← B2 ← B3 ← C

Suppose you are cherry-picking it onto a branch which contains changes that conflict with A and B. The conflict may not be serious, but it could be tedious.

For example, maybe the target branch changed one line of code, and A changed the indentation of a large section of code that encompasses that one line. This results in a large merge conflict. It's not hard to resolve, but it can be tedious verifying that the only change from A was indentation and therefore the correct merge is to take the one modified line from the target branch and fix its indentation.

And then the cherry-pick reverts A and you have to go through the same exercise.

Similarly, if B conflicts with a change in the target branch, you are going to have to deal with that conflict three times. Once when B is cherry-picked, and once again each for the follow-up commits.

What you want to do is squash all the commits together and then cherry-pick that one commit. That way, you have to deal with the conflicts only once. (Or, in the case of a conflict that was later reverted, you won't have to deal with it at all!)

So let's do that. Squash all the commits from A to C (inclusive) into a single commit:

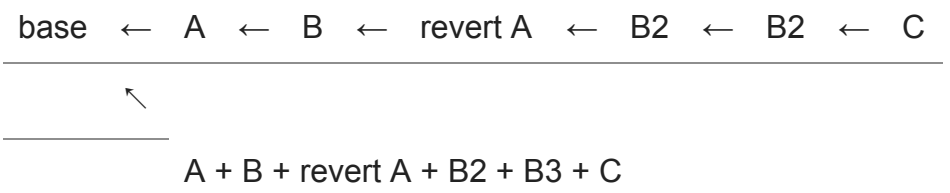
```
git commit-tree C^{tree} -p A~ -m "squasheroo!"
(this prints a hash)
```

Note: If using the Windows `CMD` command prompt, you need to type

```
git commit-tree C^^{tree} -p A~ -m "squasheroo!"
```

for reasons discussed earlier.

The hash that got printed out is a dangling commit that is the squash of everything from A to C inclusive.



Note that you don't have to be checked out to the branch that contains the series of commits you want to cherry-pick. You are creating a commit with an explicit parent, so there's no reference to `HEAD` and therefore no need to be on a particular branch. (And in practice, you are already checked out to the branch that you want to cherry-pick *into*.)

Now you can cherry-pick the dangling commit and deal with the conflicts only once.

```
git checkout target-branch [but you are probably there already]
git cherry-pick ⟨that hash that was printed by git commit-tree⟩
```

This is similar to `git diff A~ C | git apply`, but has the following advantages:

- The `git cherry-pick` can take advantage of the full contents of the base version to help resolve the conflict.
- The output of `git diff` may be problematic for binary content.
- If there are any merge conflicts, the `git apply` will just throw away the conflicting chunks, rather than leaving merge markers in the output. You'll have to go dig into the patch to find the pieces that were not successfully applied and then try to merge them yourself.
- The `git cherry-pick` keeps track of the conflicting files and will remind you of them when you do a `git status`.

- You can use your regular merge conflict workflow (such as `git mergetool`) to resolve the conflicts.
- If you decide that the merge conflicts are too nasty, you can abandon the `git cherry-pick`. A `git apply` cannot be abandoned.

Raymond Chen

Follow