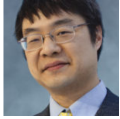
Stop cherry-picking, start merging, Part 5: Exploiting the three-way merge

devblogs.microsoft.com/oldnewthing/20180316-00

March 16, 2018



Raymond Chen

Last time, we answered some questions based on what we know about the recursive merge algorithm. Today, we'll answer questions based on what we know about the three-way merge algorithm.

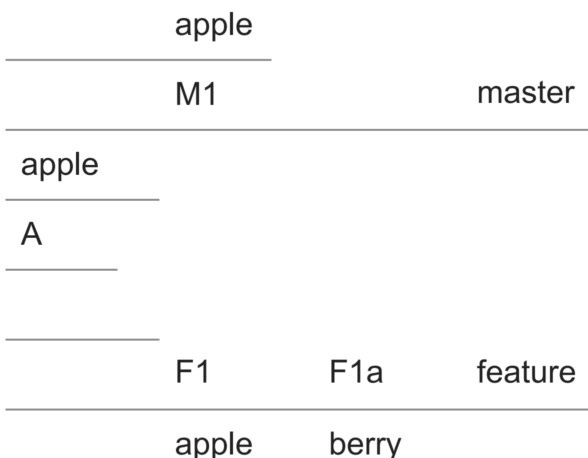
After choosing a merge base (possibly by manufacturing one via the recursive merge algorithm), the three-way merge algorithm takes the three versions identified by the merge base, the source head commit, and the destination head commit. It then identifies the changes in the two head commits relative to the merge base and tries to reconcile them.

The important detail here is what *doesn't* participate in the merge: Everything else.

In particular, any commits leading up to the head commits have no effect. And you can take advantage of this when answering the next few questions.

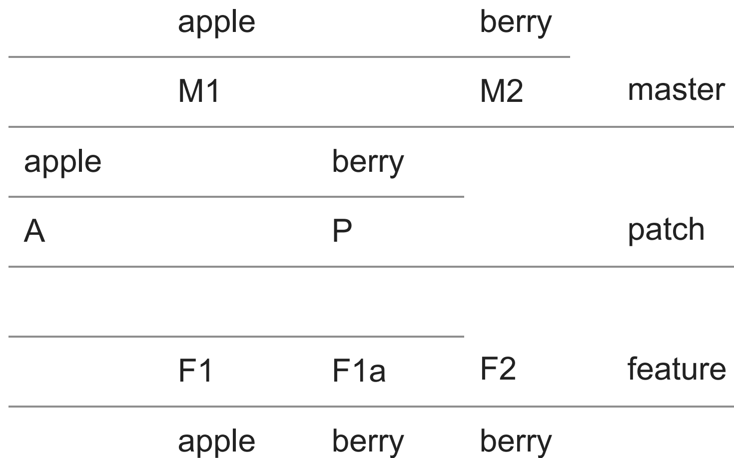
What if I already made the fix in my feature branch by committing directly to it, rather than creating a patch branch? Can I create a patch branch retroactively?

Yes, you can create a patch branch retroactively. Suppose you are in this situation:



Starting from a common commit A, you fork off a feature branch and commit a change F1. Meanwhile, the master branch commits a change M1. You then discover a terrible problem in the feature branch and apply an emergency fix F1a to the feature branch. Further investigation reveals that this terrible problem also exists in the master branch. How do you get the fix into the master branch without running the risk of a cherry-pick disaster?

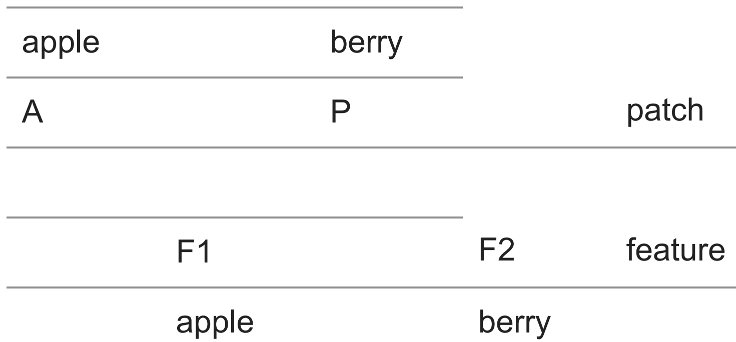
Go ahead and create your patch branch like before, and merge it into both the master and feature branches.



We created a new branch called patch based on the common ancestor commit A, and cherry-picked our fix F1a to the patch branch as commit P. We then merged commit P into the master branch, and also into the feature branch, producing commits M2 and F2, respectively. The merge into the master branch as M2 propagates the fix to the master branch, and the merge into the feature branch as F2 has no code effect because the fix is already in the feature branch. However, the merge into the feature branch is a crucial step, because it establishes commit P as the new common ancestor.

Observe that as far as the three commits involved in the merge are concerned, everything look the same as if you had made the fix in the patch branch originally. The fix is in the patch branch and in the heads of the master and feature branches. The feature branch can continue making changes, possibly to the same file, and that will be correctly detected as a change in the feature branch.

From a merge-theoretical point of view, you can use your thumb and cover up commit F1a, because that commit doesn’t participate in the three-way merge:



And then you see that this diagram is the same as the diagram we had when the change originated in the patch branch.

How can I verify that a merge carried no code change?

If you have committed the merge locally, then you can run local git commands to get your answer. If you just want a yes/no answer as to whether the most recent commit carried no code change, you can see whether the trees are the same.

```
git diff-tree HEAD
```

If there is no output, then the trees are the same, which means that there was no code change.

If you don't trust `git diff-tree`, you can compare the trees manually:

```
git rev-parse HEAD^{tree} HEAD~^{tree}
```

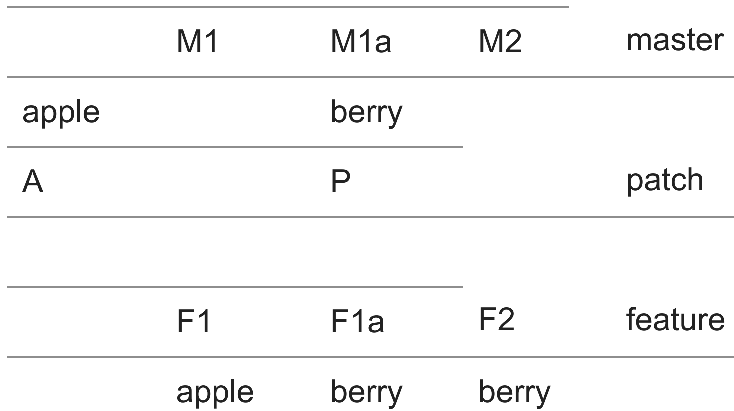
(If you are using `cmd.exe`, then you'll have to double the `^` characters because ^ is the command prompt's escape character.)

If you want to see the differences, you can use `git diff HEAD~ HEAD` to view them.

If you use an online service to manage pull requests, then you'll have to consult your online service's documentation to see if there's a way to preview the merge commit and diff it against the parent. (We'll pick up this topic in a future installment.)

What if I already made the fix in my feature branch by committing directly to it, and then I cherry-picked the change into the master branch? Can I create a patch branch retroactively?

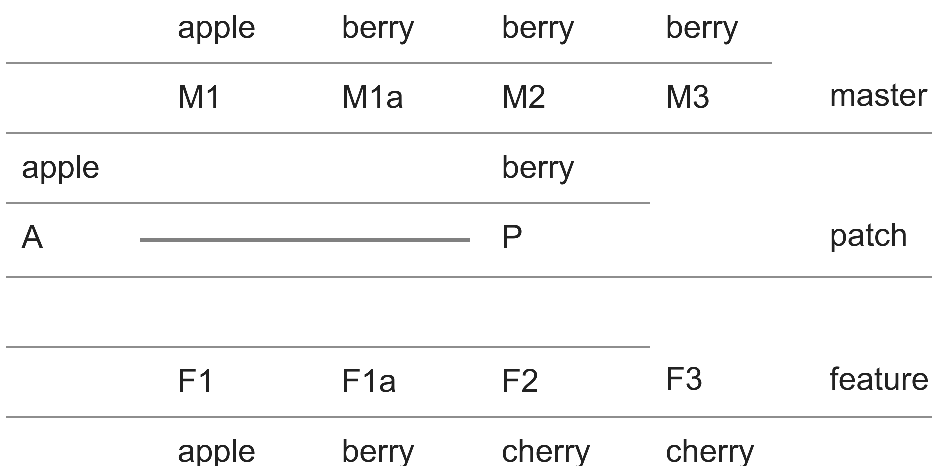
Yes, you can still create a patch branch retroactively. This is just an extension of the case where you want to retroactively pull the commit back from the feature branch, except this time you're retroactively pulling the commit back from *both* branches:



The analysis is the same: The only commits that participate in the three-way merge are the common merge base P and the heads of the master and feature branches.

What if I already made the fix in my feature branch by committing directly to it, and then I cherry-picked the change into the master branch, and I already made further changes in both branches, including a conflicting change in my feature branch? Can I create a patch branch retroactively?

Yes, you can still create the patch branch retroactively, but you have to be a bit careful because you want the merge into the feature branch to contain no code changes; the merge is for bookkeeping purposes only.



From the initial common commit A, the feature branch makes an unrelated commit F1, then makes the fix F1a, and then makes a second commit F2 that alters the fix from berry to cherry. Meanwhile, the main branch makes an unrelated commit M1, then cherry-picks the fix M1a, and then makes another unrelated commit M2.

How do you connect the fix in the feature branch with its cherry-picked doppelgänger?

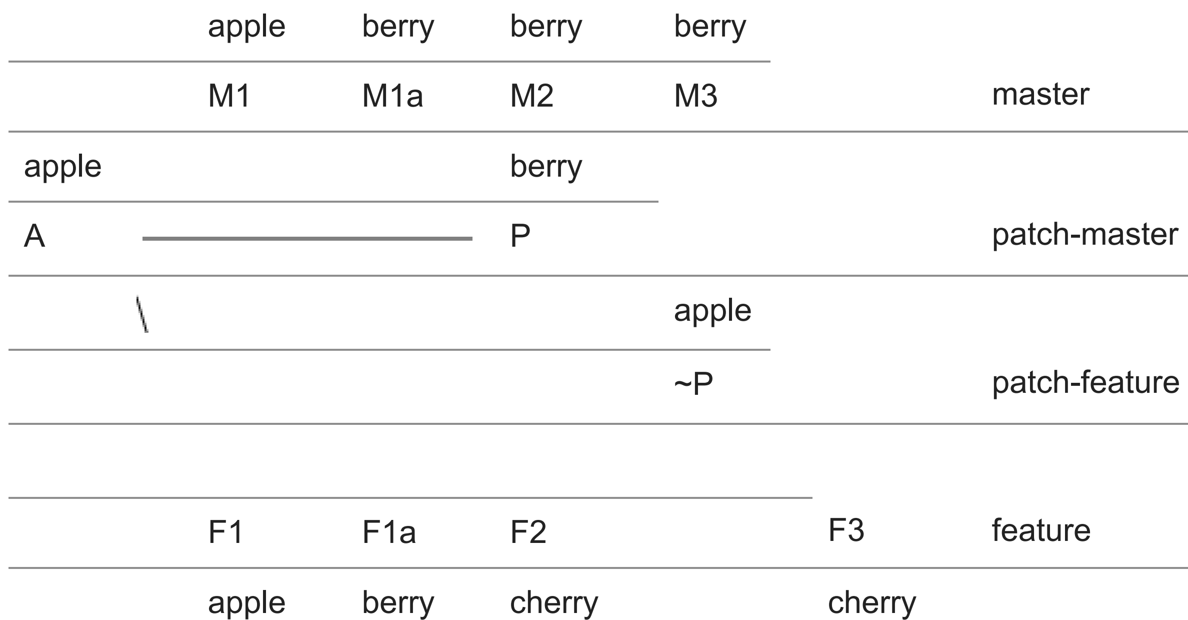
As before, create a patch branch from the common commit A and cherry-pick F1a into it. This is the fix that you want to be considered as existing in both the master and feature branches. Merge this branch into the master and feature branches, as usual. The merge into the master branch will go cleanly because the master branch hasn't made any changes that conflict with the fix. However, the merge into the feature branch will encounter a merge conflict because the feature branch continued and made a subsequent conflicting change F2.

When you get that merge conflict, specify that you want to keep the changes in the feature branch and ignore the changes in the patch branch. In other words, you want this to be a no-code-change merge. You can use the `-s ours` option to `git merge` to indicate that you want no code changes from the merge; you are doing this only for bookkeeping purposes.

I use an online service to manage pull requests. How can I force the online service to use the `-s ours` merge algorithm?

This is really a question for your online service. But let's suppose that your online service doesn't let you customize the merge algorithm. How can you force it anyway?

You can do it by pre-merging the result in your pull request. Note that this means that you will need two patch branches, one for each of the merge destinations.



As is customary, we start with a common ancestor commit A. The feature branch makes an unrelated commit F1, and then applies an important bug fix as commit F1a. The master branch makes an unrelated change M1, and then cherry-picks the fix as commit M1a. Both branches make additional changes: In the master branch, an unrelated commit M2, and in the feature branch, a conflicting commit F2.

Now you want to retroactively connect the commit F1a with its cherry-pick commit M1a so that when the master and feature branches merge, you don't get a conflict (or worse, a silent revert).

We start as before and create a patch branch from the common ancestor commit A, and create a commit P that describes the commit that got cherry-picked. This branch merges cleanly into the master branch with the cherry-picked version M1a. However, this branch doesn't merge cleanly into the feature branch made a conflicting commit F2, and your online service service rejects the pull request due to the conflict.

To fix this, you need to make sure that the branch submitted to your online service has all the conflicts pre-resolved. Create a new patch-feature branch from the patch branch you used for the master branch, and in that patch-feature branch, revert commit P, producing commit ~P, so that the patch-feature branch shows no net code change relative to the common ancestor commit A.[1]

Now that the patch-feature branch has no net change, it should merge cleanly into the feature branch. There was no code change in the payload, but the reason for the merge wasn't to pick up a code change; it was to connect the master and feature branches via the shared commit P, which becomes the new common ancestor for the future merge of the master and feature branches.

Conclusion

Okay, we saw the sorts of problems that cherry-picks can create, from merge conflicts (sometimes in unrelated branches) to silent reverts. In practice, people cherry-pick only because they don't have a better choice available. They would rather perform a partial merge but git doesn't support partial merges, so people feel that they have to cherry-pick. But I showed that partial merges are possible after all! You just have to think about the graph the right way: Instead of merging directly between branches, you create a helper branch that contains the partial content and merge the helper branch into the desired destinations.

As we saw when we explored the recursive merge algorithm, if you expect that your change will need to be cherry-picked to many other branches, you can stage a helper branch that is based on a commit far back enough in time that everybody who would be interested in cherry-picking the change will also have the commit your branch is based on. (In practice, this means going back to the commit that introduced the change that you are trying to patch.) If everybody merges from that helper branch rather than cherry-picking, then when all the branches merge together, the helper branch will contribute to the merge base, and that avoids the conflicts and other bad things.

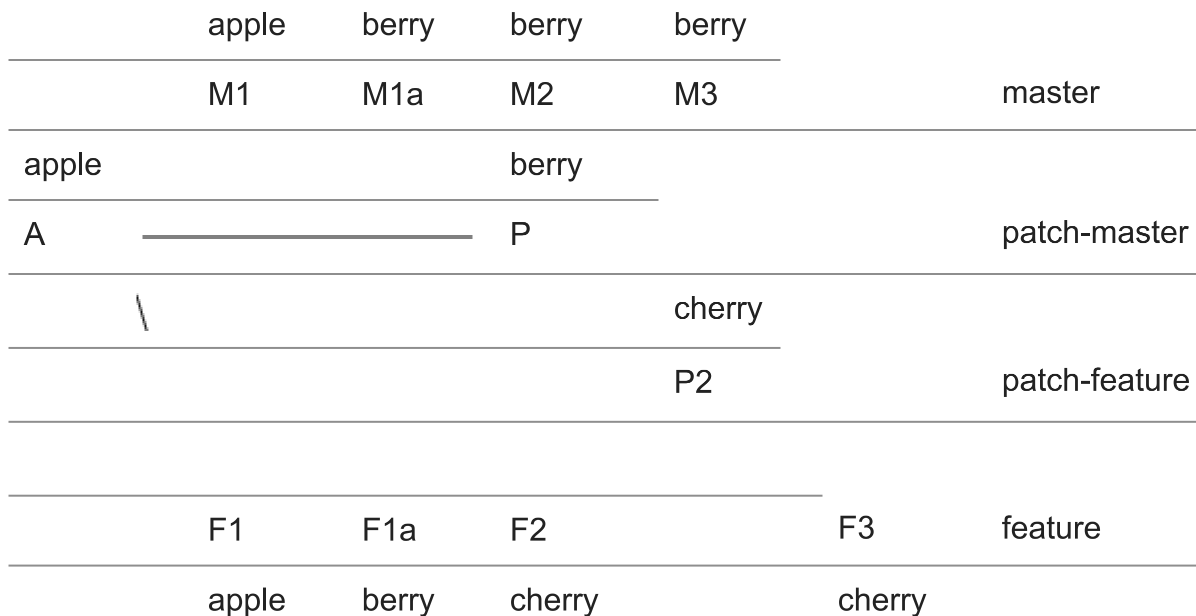
My team applied the techniques in this series, and following the guidance herein reduced the number of conflicts in a single merge from over 1500 files to only 20. This changed an unmanageable merge to one that could be handled by contacting the person responsible for

each conflict and asking them to resolve it.

(Note: This series is only half-over, even though I wrote a Conclusion. So don't worry: There's plenty of agony still to come.)

Footnote

[1] Another way to do this is to create a new branch named patch-feature from commit F2, and then perform a `git merge -s ours patch-master` to create a no-code-change merge from the patch-master branch. This results in a line from P2 to F2, which is harmless:



If you want to get rid of the superfluous line, you could use the `--squash` option, but I would leave it because it makes it clearer what happened. (Otherwise, it will look like the patch branch made a huge commit.)

Personally, I would use `git commit-tree` to construct commit P2. I'll talk about the magical powers of `git commit-tree` at some unspecified future point.

However you created the patch-feature branch, you can then create a pull request from the patch-feature branch to the feature branch.

Raymond Chen

Follow