Stop cherry-picking, start merging, Part 4: Exploiting the recursive merge algorithm

devblogs.microsoft.com/oldnewthing/20180315-00

March 15, 2018

Raymond Chen

The last few days have looked at the dangers of cherry-picking, both in terms of latent merge conflicts and (even worse) *missing* merge conflicts, and last time, I proposed the alternative to cherry-picking: Merging from a common branch.

Before we can explore further, we need to understand the recursive merge algorithm.

Instead of trying to explain it, I will defer to this explanation of the recursive merge algorithm. Go read it, and then we can talk about its consequences.

Hi, thanks for coming back. Our simple example last time did not require the full power of the recursive merge because there is still a single best common ancestor. But knowing how the recursive merge works helps you answer some common follow-up questions.

How do I find the correct merge base?

The `git merge-base master feature` command will find a merge base. You can use that as the basis for your patch branch. You can also say `git merge-base -a master feature` to show *all* merge bases, and you can choose the one that best describes your intent.

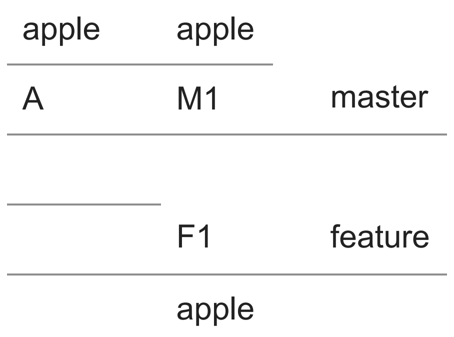
How do I know which best describes my intent?

That's really up to you to decide. For example, maybe one of the merge bases is a patch branch, and the other is a regularly-scheduled merge between the master and feature branch. If your patch is intended to build on top of the previous patch, then using the previous patch branch describes your intent better. But if your patch is independent of the previous patch, then using the regularly-scheduled merge describes your intent better.

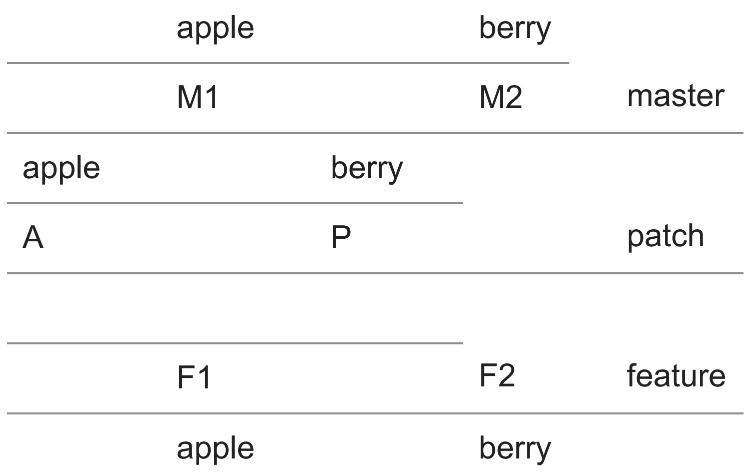
What if I pick the wrong merge base, and instead pick a merge base that is not a common ancestor?

If you choose a merge base that isn't actually a common ancestor, then when you prepare the merges from the patch branch into the master and feature branches, one or the other merge will encompass more than just the single commit you are trying to patch.

Let's go back to the diagram we had at the start of yesterday's discussion:



From a common ancestor A, commit F1 happens on the feature branch, and commit M1 happens on the master branch. Now you realize that you need to apply a fix to both branches. But instead of creating your patch branch from commit A (the common ancestor), you mistakenly create it from F1.



From commit F1, you create a patch branch and apply a commit P to it, which contains the desired fix. This branch is then merged into the master branch (creating commit M2) and into the feature branch (creating commit F2).

This diagram is identical to the second diagram from last time, except that the patch commit P is based off commit F1 rather than commit A.

What happens?

What happens is that the merge into the master branch includes commit F1, which is not what you intended.

If you're using a pull request, then the list of encompassed commits in the pull request will be longer than just one commit, and the diff of the pull request will show unwanted changes. That is your signal that something funny is going on. If you're doing straight merges from the command line, you'll find that the history for the merge into the master branch pulled in more than just one change, and the diff of the merge shows that the merge into the master branch contained both the desired changes from commit P as well as some unwanted changes from commit F1.

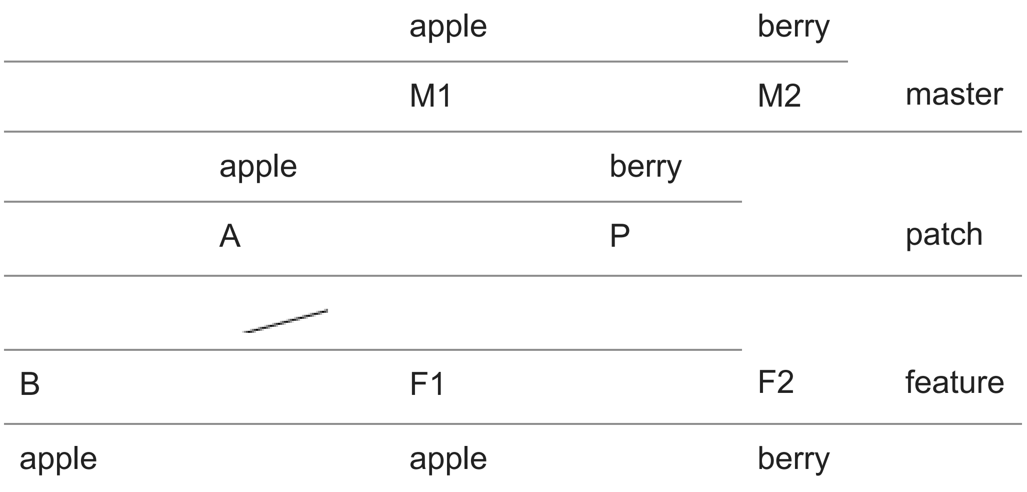
You can protect against this by doing

```
git log master..patch
git log feature..patch
```

and verifying that only one commit (namely, your patch) comes out of each log query.

What if I pick the wrong merge base, and instead pick a merge base that is a common ancestor, but not the most recent one?

Suppose that instead of choosing the most recent common ancestor, you choose an older one.



From a common ancestor A, commit F1 happens on the feature branch, and commit M1 happens on the master branch. We create a patch branch not from commit A, but from an even older commit B that is an ancestor of commit A. We then merge that patch branch into the master and feature branches.

What happens is that the eventual merge of the master and feature branch will have multiple best common ancestors. One is the merge base that would have been selected if you had never created a patch branch (A). The other is the patch branch that you created (P). The recursive merge algorithm will merge these two branches together, the result of which is… surprise! the version of the patch branch you would have gotten if you had created it from the correct merge base in the first place.

In other words, it doesn't matter which common ancestor you pick, as long as you don't pick one so far back that the merge with the most recent common ancestor will encounter a merge conflict. But you're unlikely to do that because that would mean that the merges into the current heads of the master and feature branches would also encounter merge conflicts, and that would tell you that something is wrong.

The above result is an important one, because it means that you could choose as your common ancestor not the most recent common ancestor, but in fact the *oldest* common ancestor that the patch still applies to. In other words, go back to the commit that introduced the code you want to change. That commit is in both the master and feature branches by virtue of the fact that the problem exists in both branches. Apply your patch to that commit, and then merge the patch into the master and feature branches. From the graph, it looks like you had a side branch that immediately fixed the problem, but you merely took a long time before deciding to merge that fix back into the master and feature branches.

Having a patch branch ready with the fix is handy when we get to the next question.

Note that you might choose an older common ancestor *on purpose*, if it better describes your intent. For example, in the above diagram, commit A might be a commit from a patch branch, whereas commit B is a regularly-scheduled merge between the master and feature branches. As with the case of multiple merge bases, you can choose the commit that best expresses what you're trying to do with your patch. If your patch builds on top of the work in commit A, then creating your patch branch from commit A describes your intent best. On the other hand, if your patch is independent of the previous patch, then creating your patch branch from commit B makes it clearer that your patch is unrelated to the previous one.

What if I have multiple branches that I need to fix?

As we discovered in the previous question, it doesn't matter which common ancestor you use, as long as it's a common ancestor. So create a patch branch that is based on an old common ancestor, old enough to be in all the branches you want to apply the fix to. Say, the commit that introduced the line of code you want to modify with the patch. Tell anyone who wants to pick up the fix, "Merge the patch branch into your branch."

Instead of telling everybody to cherry-pick the fix, tell them to merge the patch branch. This is a branch specially crafted so that merging it picks up exactly one commit, namely the fix.

And if everybody merges the same patch branch, then they won't encounter conflicts when they merge with each other.

What if I need to share multiple changes between the branches?

Maybe you need multiple changes rather than a single change. For example, you created a patch branch with the fix, then discovered a problem with the fix, so you have another commit that fixes the fix.

No problem. In your patch branch, put all the changes you want to share. Once you have your entire payload sitting in the patch branch, you can merge it into the master and feature branches.

What if I later realize I need to merge another fix?

What if you're having a really bad day, and after you merge the patch branch into the master and feature branches, you discover another problem that forces you to disable a different part of the feature. Is it safe to create a second patch branch and follow the same exercise? Does the second patch branch have to be based on the first patch branch?

Again, through the magic of the recursive merge algorithm, it doesn't matter which way you do it. Whether your second patch branch is based on the first patch branch or whether it's an independent branch turns out to be irrelevant, because the recursive merge algorithm will merge all the patch branches together anyway. The decision to base it on the previous patch branch should be based on what is easier for others to understand.

Okay, those are the follow-up questions that can be answered by applying your understanding of the recursive merge algorithm. Next time, we'll look at follow-up questions that can be answered by applying your understanding of the three-way merge algorithm.

Raymond Chen

Follow