Stop cherry-picking, start merging, Part 3: Avoiding problems by creating a new merge base

devblogs.microsoft.com/oldnewthing/20180314-00

March 14, 2018

Raymond Chen

The first two parts of the series discussed the bad things that can happen if you cherry-pick a change that is subsequently modified. If you're lucky, you get a merge conflict. If you're not lucky, your modification is simply ignored. If only there were a way to do a partial merge instead of a cherry-pick, the problems could have been avoided.

It turns out that git does support partial merges. It's just that nobody talks about it that way. You create a partial merge by doing full merge with a custom merge base.

At the start of our saga, we have a commit tree like this:

apple	apple	
A	M1	master

	F1	feature
	apple	

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. You don't want to merge the entire feature branch into the master branch, because that would also pick up commit F1.

Here's the trick: Create a third branch and merge it into both the master and feature branches.

	apple	berry	
	M1	M2	master

apple	berry		
A	P		patch

	F1	F2	feature
	apple	berry	

We created a new branch called patch based on the common ancestor commit A, and committed our fix 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.

As before, work continues on both the master and feature branches, and eventually the root cause of the problem is determined, and the patch is reverted in the feature branch and a proper fix applied.

	apple	berry	berry	
	M1	M2	M3	master

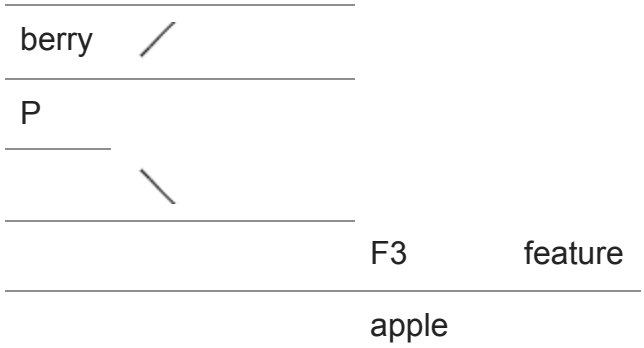
apple	berry	
A	P	patch

	F1	F2	F3	feature
	apple	berry	apple	

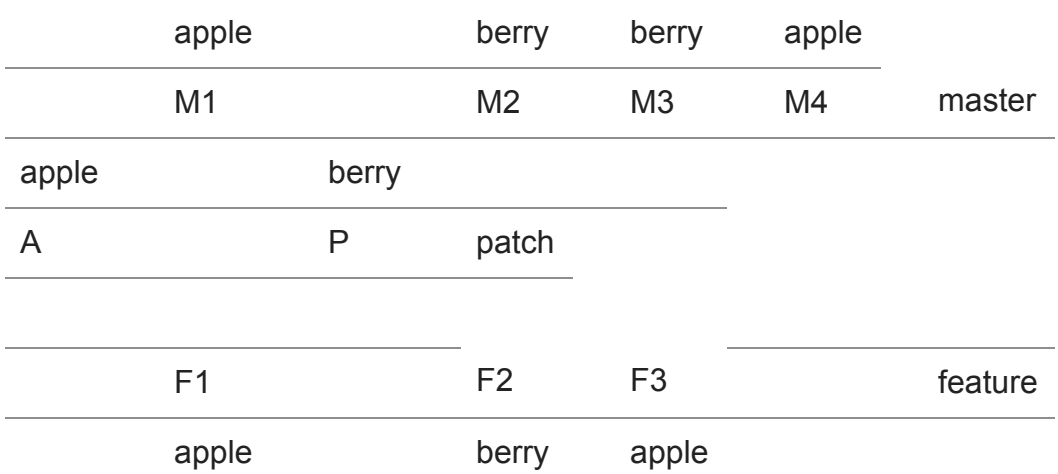
On the master branch, commit M3 does additional work unrelated to our patch. Meanwhile, in our feature branch, we figure out the proper fix and commit it as F3. Commit F3 changes the line back to `apple` (undoing our patch) as well as containing the proper fix.

Eventually, it comes time to merge the feature branch to the master branch. The merge chooses commit P as the merge base, since it is the most recent common ancestor. The commits involved in the three-way merge are P (the base), M3 (the head of the master branch) and F3 (the head of the feature branch). Let's erase all the other commits, since they don't participate in the merge.

	berry	
	M3	master

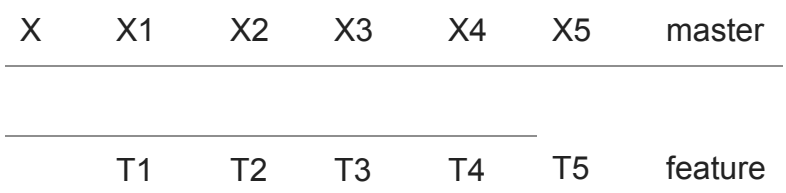


There is no change to the line in question in the master branch relative to the merge base, but in the feature branch, `berry` changed to `apple`. Therefore, the merge result will have `apple`.



But wait, what about the changes in commits M1 and F1? They were bypassed by commit P, weren't they? Are those changes going to be lost?

Nope, those changes will be merged in just fine because they are also present in M3 and F3. This is the same situation you run into in normal day-to-day operation when you merge from the master to the feature branch periodically while you work on your feature:



In the above diagram (a brand new diagram unrelated to the previous diagrams), you created a feature branch from the master branch at some commit X. Work continues in the master branch as commits X1, X2, and so on. Simultaneously, work continues in the feature branch

as commits T1, T2, and so on. But every so often, the feature branch takes a merge from the master branch, so that the two don't drift too far out of sync.

Suppose you are now ready to merge the feature branch back to the master branch. The last time the feature branch merged from the master branch was when it merged commit X4, resulting in commit T5 on the feature branch. This makes commit X4 the merge base. Are you worried that this upcoming merge will throw away the changes in commits T1 through T4, since the merge base commit X4 post-dates them? No, you aren't, because you know that the changes in T1 through T4 are also present in T5, and they will go into the master branch as part of the merge.

Okay, back to our original story. Creating the patch branch and merging it into both the master and feature branches preserves the connection between the two commits in the respective branches, and in particular identifies them as being two manifestations of the same underlying change (namely, commit P). The resulting merge of the two branches recognizes this relationship and doesn't double-apply the change.

Basically, the patch branch converts what was originally a cherry-pick into a merge. It was the cherry-pick that was the source of all the problems, and the fix is to get rid of the cherry-pick and use merges instead. The temporary patch branch gives us our partial merge.

That's the basic idea. There are still a lot of questions to answer, such as "How do I find the correct merge base?", "What if I pick the wrong merge base?", "What if I need to perform two cherry-picks?", or "What if I already did the cherry-pick; can I somehow repair the damage and prevent the future merge conflict or ABA problem?" We'll start delving into them next time.

Raymond Chen

Follow