Stop cherry-picking, start merging, Part 1: The merge conflict

devblogs.microsoft.com/oldnewthing/20180312-00

March 12, 2018

Raymond Chen

Cherry-picking is a common operation in git, and it's not a good idea. Sometimes it's a neutral idea, but I haven't yet found a case where it's actually good.

This is the start of a series that will begin by explaining why cherry-picking is bad, continue by explaining why cherry-picking is worse, then try to talk you down from the ledge by showing how to get the same effect as a cherry-pick but with merging, showing how to apply that technique to the case where you need to do a retroactive merge, and wrap up by showing how to apply that technique to the case where you already made the mistake of cherry-picking and want to fix it before something bad or worse happens.

It's a tall order, but I've been meaning to write this up for a while, and what's gotta get done gotta get done.

In order to cherry-pick, you need two branches, one to be the donor and one to be the recipient. Let's call them the master branch and the feature branch. And for simplicity's sake, let's say that the commit being cherry-picked is a one-line change to a single file. Each commit will be annotated with the contents of that one line.

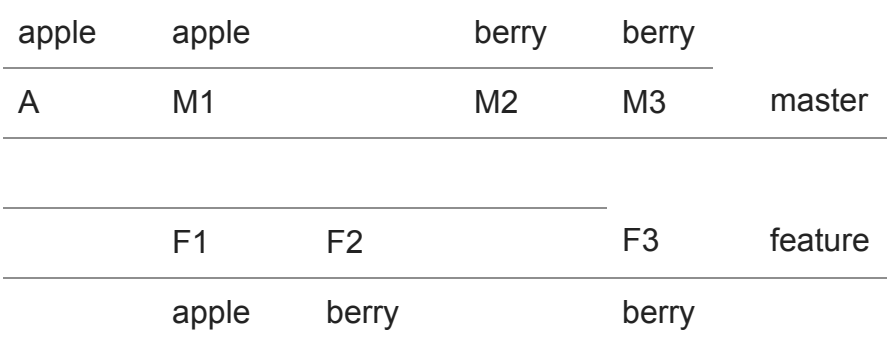
apple	apple		berry	
A	M1		M2	master
	F1	F2		feature
	apple	berry		

For the purpose of illustration, I'm using a dotted line to denote cherry-picks. This dotted line doesn't really exist in the repo, but I'm drawing it to help express the chronology. (Eventually, I'll stop drawing dotted lines, too.)

You have some common ancestor A, and in the commit, the line in question is the word `apple`. From that common ancestor, the two branches diverge: Commit F1 happens on the feature branch, and commit M1 happens on the master branch. These changes don't affect the line in question, so it still says `apple`. You then make some commit F2 in the feature branch that changes the line in question from `apple` to `berry`, and you cherry-pick commit F2 into the master branch as M2.

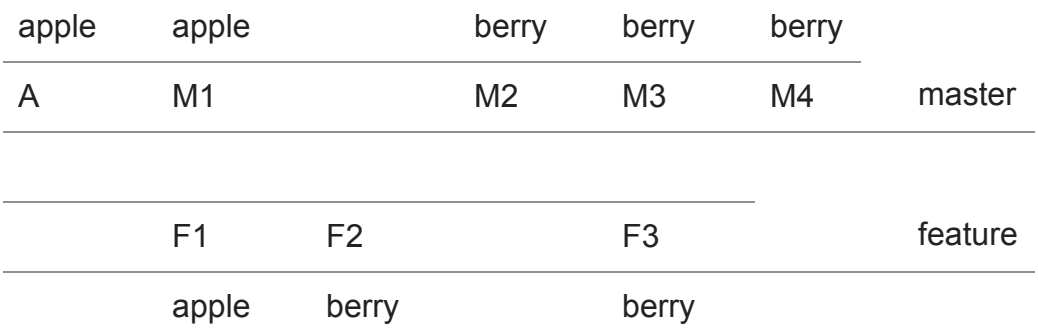
So far, nothing exciting is happening.

Time passes, more commits occur, and your commit graph looks like this:



You made another commit M3 to the master branch and another commit F3 to the feature branch. Neither of these commits affected the line in question, so the line is still the word `berry`.

It's time to merge back, and since the line in question is the same in both branches, the merge is trivial, and the result in the final merged result is `berry`.



This is the ideal case.

It is also relatively uncommon in an active code base.

Consider this alternate timeline: After the cherry-pick, additional commits M3 to the master branch and F3 to the feature branch are made, but this time commit F3 changes the line in question to `cherry`. This could be because the person who made the original commit F2

found an improvement (cherries are on sale right now), or maybe they made a larger change that happened to require switching from berries to cherries.

Whatever the reason, the commit graph now looks like this:

apple	apple		berry	berry	
A	M1		M2	M3	master
	F1	F2		F3	feature
	apple	berry		cherry	

This time, when it's time to merge the feature branch back into the master branch, there is a merge conflict. The base of the three-way merge contains `apple`, the incoming feature branch has `cherry` and the existing master branch has `berry`.

```
<<<<<<< HEAD (master)
berry
||||||| merged common ancestors
apple
=======
cherry
>>>>>>> feature
```

The conflict occurred because the cherry-picked changes were subsequently changed again by one of the branches. We've been using dotted lines in our diagrams to emphasize that the cherry-pick relationship is all in our heads, and not actually recorded anywhere in the commit graph.

You have to hope that whoever resolves this merge conflict remembers the history of this line, or can access the team's knowledge of this line of code to understand that the correct resolution it to accept the changes in the feature branch rather than the one in the master branch.

In this case, there haven't been many changes, and there are only two branches involved, and hopefully there aren't too many other conflicts in the merge (so that the person resolving the merge hasn't gotten tired and burnt out), so the chance of a correct resolution are pretty good. But consider this three-branch scenario:

apple	apple	berry	berry	
A	M1	M2	M3	master

	apple		apple	apple	cherry	
	V1		V2	V3	V4	victim

	F1	F2	F3	feature
	apple	berry	cherry	

Start with a commit A, where the line in question is `apple`. We create a branch based on commit A, ominously named `victim`, and add a commit called V1, which doesn't affect the line in question, so it still is `apple`. From the victim branch we create our feature branch from commit V1, and then the story is the same: To the feature branch, we add the same commit F1 from before, which doesn't affect the line in question, so it continues to be `apple`. Meanwhile, the master branch added a commit M1 which doesn't affect the line in question.

We continue as before: The feature branch adds a commit F2 which changes the line in question to `berry`, and the master branch cherry-picks that commit as `M2`. The feature branch makes another change F3 which happens to update the line in question from `berry` to `cherry`, while the master branch adds a commit M3 that doesn't change the line in question, so it remains `berry`.

All through this, the victim branch is blithely unaware of the cherry-picking disaster being created by the feature and master branches. It commits changes V2 and V3 which have nothing to do with the line in question, so the line is still `apple`.

Eventually, the feature branch merges its changes back into the victim branch, producing commit V4, where the line in question is now `cherry`, thanks to the changes that were made in the feature branch.

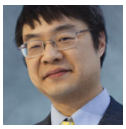
The time bomb has now moved into the victim branch.

The victim branch decides to take a merge from the master branch, and that is where the conflict is detected, because this is the first time the original change F2 encounters its cherry-picked doppelgänger M2. The poor person stuck with this merge conflict has no idea of the deal with the devil struck by the feature and master branches behind his back. Furthermore, the person stuck with this merge conflict may be exhausted from dealing with all the other (valid) conflicts caused by the merge from the master branch and may not have the mental energy to reverse-engineer how the two branches ended up the way they did and figure out which side is right.

Basically, when you cherry-pick a commit, you now have two copies of the commit sitting in the graph. Any lines of code affected by that commit must remain untouched in both branches until the two copies of the commit finally merge. If either branch modifies any line touched by the cherry-pick, then that creates a powderkeg that can sit quietly indefinitely. It is at the time somebody tries to merge the two commits together that the explosion occurs, and that point could be in a faraway place not immediately related to the branches involved in the cherry-pick. This means that the person trying to resolve the merge was never part of the cherry-pick madness and may not know who to talk to in order to figure out what happened.

Okay, that was a long story, and you probably knew most of it already, but believe it or not, as bad as this is, it could get even *worse*: The explosion might not happen.

Wait, why is it worse that an explosion doesn't happen? We'll pick this up next time.



Raymond Chen

Follow