# Stop cherry-picking, start merging, Part 2: The merge conflict that never happened (but should have)

devblogs.microsoft.com/oldnewthing/20180313-00

March 13, 2018

Raymond Chen

Last time, we saw how editing the code affected by a cherry-pick creates a potential merge conflict that doesn't become realized until the original commit and its cherry-picked doppelgänger meet in a merge somewhere, which could be far away from the branches that contained the original commit and its cherry-pick.

But you know what's worse than a merge conflict?

No merge conflict.

Let's set up the same situation as last time:

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

Suppose this feature branch has been around for a while, merging its changes back into the master branch when it reaches a stability milestone, Our diagram begins with the point just after the most recent merge back to the master branch, where the feature branch has started its work on the next milestone's worth of features.

Let's suppose that the line that contains the word `apple` is in a configuration file that controls the feature. Both the master branch and feature branch make commits (M1 and F1, respectively) which are unrelated to the configuration file.

Suppose you now discover a serious problem in the feature that is causing it to go haywire. To stop the immediate problem, you make a commit F2 to the feature branch which sets the configuration file to `berry`, which has the effect of shutting off the feature.

(In real life, the change would be more like changing

```
#define IS_FEATURE_ENABLED 1
```

to

```
#define IS_FEATURE_ENABLED 0
```

but I'm sticking with `apple` and `berry` so that it lines up better with yesterday's examples.)

Okay, you disable the feature in the feature branch, verify that it doesn't have any unexpected side effects, and cherry-pick the fix into the master branch. Phew, this stops the bleeding and buys you time to figure out what went wrong and come up with a fix.

(If your workflow is to apply the fix to the master branch and then cherry-pick it into the feature branch, then great, do it that way. The story is the same.)

Work continues in the master branch while you investigate the problem. Later, you come up with the real fix in the feature branch, which involves re-enabling the feature (by setting the line to `apple`) and fixing the root cause in some other place. The commit graph now looks like this:

| | | | | |
|---|---|---|---|---|
| apple | apple | | berry | berry |
| A | M1 | | M2 | M3 |

master

| | | | |
|---|---|---|---|
| | F1 | F2 | F3 |
| | apple | berry | apple |

feature

In the master branch, an additional unrelated commit M3 was made on top of M2. In the feature branch, an additional commit F3 was made on top of F2, and F3 changes `berry` back to `apple`, as well as fixing the root cause of the issue.

Okay, now you want to merge the feature branch into the master branch so that the temporary fix can be replaced by the real fix. But when you do the merge, this happens:

| | | | | |
|---|---|---|---|---|
| apple | apple | | berry | berry |

| | | | | | |
|---|---|---|---|---|---|
| A | M1 | | M2 | M3 | M4 | master |

| | | | | |
|---|---|---|---|---|
| | F1 | F2 | F3 | feature |
| | apple | berry | apple | |

The master branch merged from the feature branch, producing commit M4, but in commit M4, the line still says `berry`! The temporary fix is still in place in the master branch. Actually, it's worse than that. The `berry` part of the temporary fix is in place in the master branch, but so too is the permanent fix in the other part of commit F3! It's possible that these two partial fixes don't interact well with each other, in which case you're in the even worse position that the feature is broken in the master branch but works in your feature branch.

Today, we'll investigate what happened. Next time, we'll investigate how to prevent this from happening in the future.

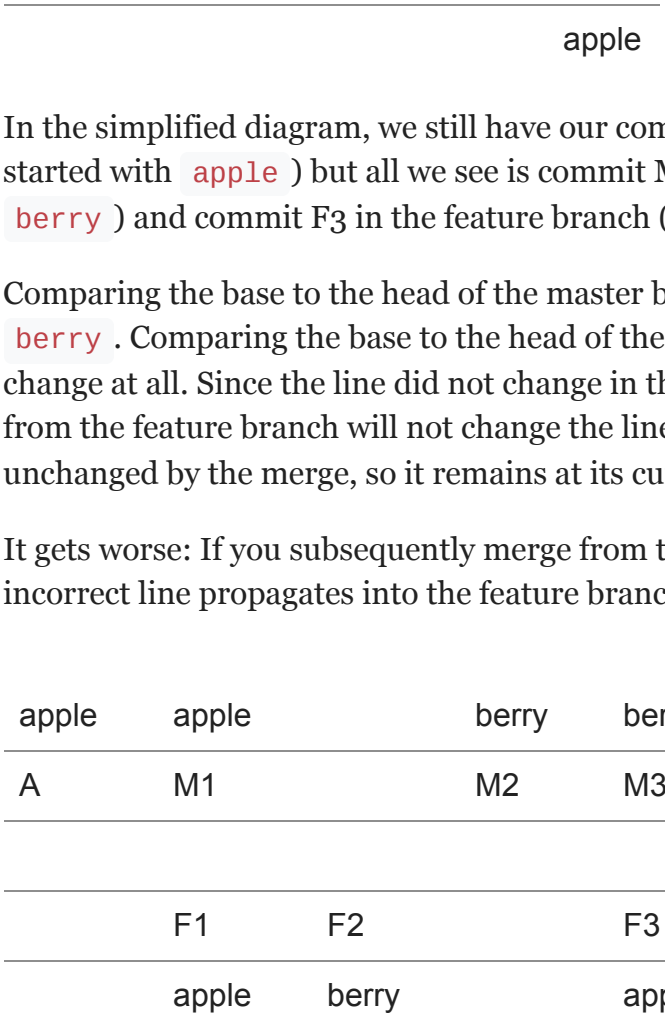
Let's go back to the state of the repo before we tried to merge the feature branch into the master branch:

| apple | apple | | berry | berry | |
|---|---|---|---|---|---|
| A | M1 | | M2 | M3 | master |

| | | | | |
|---|---|---|---|---|
| | F1 | F2 | F3 | feature |
| | apple | berry | apple | |

Now we perform the merge. Git looks for a merge base, which is commit A, the most recent common ancestor between the two branches. Git then performs a three-way merge using A as the base, M3 as HEAD, and F3 as the inbound change. All that matters now is the delta between the base and the two terminal commits, so let's remove the irrelevant commits from the diagram.

| apple | | berry | |
|---|---|---|---|
| A | | M3 | master |
| | \ | | |
| | | F3 | feature |




In the simplified diagram, we still have our common merge base at commit A (where we started with `apple`) but all we see is commit M3 in the master branch (where we have `berry`) and commit F3 in the feature branch (where we have `apple`).

Comparing the base to the head of the master branch, we see that `apple` changed to `berry`. Comparing the base to the head of the feature branch, we see that `apple` didn't change at all. Since the line did not change in the feature branch, it means that the merge from the feature branch will not change the line either. The result is that the line remains unchanged by the merge, so it remains at its current value in the master branch of `berry`.

It gets worse: If you subsequently merge from the master branch into the feature branch, the incorrect line propagates into the feature branch.

For the merge from the master branch to the feature branch, the common merge base is commit F3, which is also the head of the feature branch. In commit F3, the line is `apple`. In the head of the master branch, it is `berry`, and that change propagates to the feature branch. As a result, in the new commit F4 in the feature branch, the line is now `berry`. (I chose to use a non-fast-forward merge, but you would see the same thing if it were a fast-forward merge.)

Most people think of cherry-picks as "anticipatory partial merges", where you want to merge part of a source branch into your destination branch. The expectation is that if you later decide to merge the rest of the source branch into the destination branch, it will merge in only the new parts.

And if you are careful not to touch the lines affected by the cherry-pick until the two sides of the cherry-pick finally merge, that's what happens, because the merge will see that both sides modified the file in the same way, and the two commits are coalesced.

But if you make additional changes to the affected line in either of the branches, then instead of coalescing, the two changes are added together. And if your additional changes to the affected line have the effect of canceling out the cherry-picked change, then you don't even

get a merge conflict to inform you that something weird happened. (Internally on our team, we call this the ABA problem because the line started with A, changed to B, the B got cherry-picked away, and then the line changed back to A prior to the merge back to the master branch.)

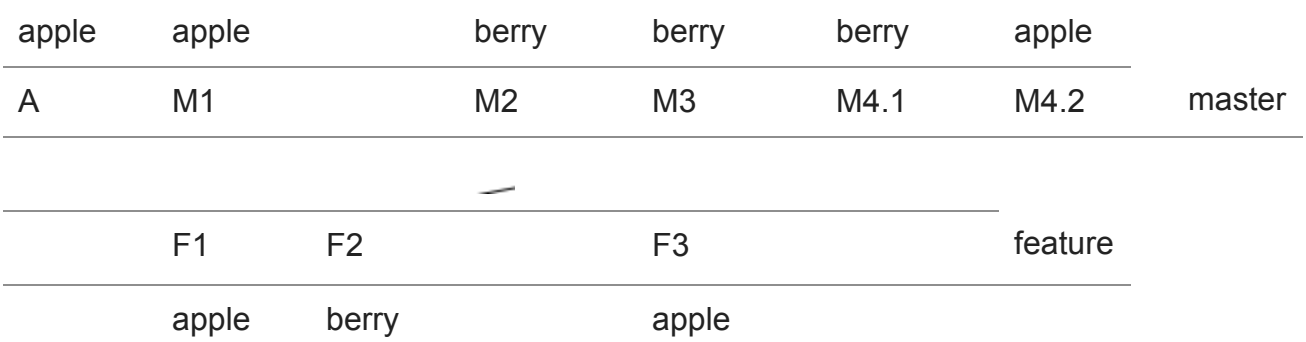
The master branch applied a change, and the feature branch applied the change, and the feature branch reverted the change. Mathematically, you performed two changes and one revert, so the net effect is still a +1 in favor of the change.

Okay, so the problem is that we wanted to do a partial merge from the feature branch back into the master branch. Too bad there's no such thing as a partial merge.

Or is there?

Next time, we'll show how to perform a partial merge.

**Bonus chatter**: Normally, merging twice produces the same result as merging once, just with more merge conflicts (because you have to resolve the conflict twice, once at each merge). But in this scenario, we get different results, neither of which raise merge conflicts. If we had performed *two* merges from the feature branch into the master branch, first by merging commit F2, then again by merging commit F3, then we would have had two clean merges, but the result would have been different:




This is troubling because it means that changing your policy on how often you merge can result in different final results, without any warnings from git.

**More bonus chatter**: Note that the "revert" need not be an actual revert. It might merely happen to resemble a revert. For example, suppose you start with

```
char* predefined_items[4] = {
 "armoire",
 "bed",
 "credenza",
 "desk",
};
```

You decide that you need a fifth item, so you add the fifth item and bump the array size:

```
char* predefined_items[5] = {
 "armoire",
 "bed",
 "credenza",
 "desk",
 "end table",
};
```

Another branch cherry-picks this because it needs the end table. Meanwhile, you realize that you don't need the bed any more, so you remove it and drop the array size to four.

```
char* predefined_items[4] = {
 "armoire",
 "credenza",
 "desk",
 "end table",
};
```

When these two changes merge, the result will be

```
char* predefined_items[5] = {
 "armoire",
 "credenza",
 "desk",
 "end table",
};
```

Notice that the length of the `predefined_items` array is five, even though there are only four entries in it.

Raymond Chen

**Follow**