How to split off an older copy of a file while preserving git line history

devblogs.microsoft.com/oldnewthing/20230728-00

July 28, 2023

Raymond Chen

Some time ago, I showed how to duplicate a file while preserving git line history. But what if you want to revive and split off an older version of a file while preserving git line history?

You can follow the same cookbook as splitting off a copy of a file. Just do the copy when you are checked out to the point when the file contains the contents you want to copy.

```
git checkout -b split old-commit
git mv file file-copy
git commit -m "Copy old copy of file at old-commit"
git checkout HEAD~ file
git commit -m "Restore mainline of file"
git checkout main
git merge split
```

Once you realize that you just have to end up with an unchanged original and a new file that was moved from an (earlier copy of) that original file, you can reduce the number of commits required to set things up in the desired final state, although it's also a bit more cumbersome.

```
git checkout -b split old-commit
git mv file file-copy
git commit -m "Copy old copy of file at old-commit"
git checkout main
git merge --no-commit --no-ff split
git checkout HEAD file
git checkout split file-copy
git commit -m "Split out old copy of file at old-commit"
```

In the `split` branch, we move the file (as of `old-commit`) to its copy. We then merge that branch into the main branch, but without committig yet. The normal result of the merge is that both the effects in the main branch and in the split branch take effect:

- In the split branch, the file moved, so the result of the merge is that the file moves.
- In the main branch, the file changed, so the result of the merge is that the file changed.

The merged result is therefore that the file moved *and* changed. Which is not what we want. We want the file to be copied, not moved.

We fix the merge result by restoring the original file, and we also send the copy back to what it held at the time it split off from the original.

We then commit this edited result and explain what we did.

The `file`'s history follows its path through the `main` branch. Since there was no net change to the file from the merge, git by default ignores the merge when studying the file history.

The `file-copy`'s history follows its path through the `split` branch, since it didn't exist in the main branch. That branch tells git that `file-copy` was moved from `file`, which causes the old history to be attached to the copy.

The rule of thumb is that when git is following the history of a file, and it finds the commit that created the file, it will look in that same commit for a deleted file whose former contents are identical to the newly-created file. If there is a match, then it will treat the operation as a "move/rename" and consider the deleted file to be the predecessor of the newly-created file.[1]

All of the machinations above are to create a commit that allows git to treat the new file as a move/rename of the file you want its history to be connected to.

It is perfectly legitimate for a single file to be considered the predecessor for multiple files. Indeed, all of the "splitting a file" tricks rely on it!

[1] If git cannot find a perfect match, it will also look for a near-match. Although the near-match algorithm is tunable, it cannot be configured in the repo, so different people with different tunings will get different results. I prefer to use a method that is guaranteed to work rather than one that relies on everybody remembering to use the same tuning settings.