Mundane git tricks: Combining two files into one while preserving line history

devblogs.microsoft.com/oldnewthing/20190514-00

May 14, 2019

Raymond Chen

Suppose you have two files that you want to combine into one. Let’s set up a scratch repo to demonstrate. I’ve omitted the command prompts so you can copy-paste this into your shell of choice and play along at home. (The timestamps and commit hashes will naturally be different.)

```
git init

>fruits echo apple
git add fruits
git commit --author="Alice <alice>" -m "create fruits"
>>fruits echo grape
git commit --author="Bob <bob>"     -am "add grape"
>>fruits echo orange
git commit --author="Carol <carol>" -am "add orange"

>veggies echo celery
git add veggies
git commit --author="David <david>" -m "create veggies"
>>veggies echo lettuce
git commit --author="Eve <eve>"     -am "add lettuce"
>>veggies echo peas
git commit --author="Frank <frank>" -am "add peas"

git tag ready
```

We now have two files, one with fruits and one with vegetables. Each has its own history, and the `git blame` command can attribute each line to the commit that introduced it.

```
git blame fruits

^adbef3a (Alice 2019-05-14 07:00:00 -0700 1) apple
8312990f (Bob   2019-05-14 07:00:01 -0700 2) grape
2259ff53 (Carol 2019-05-14 07:00:02 -0700 3) orange

git blame veggies

2f11bacc (David 2019-05-14 07:00:03 -0700 1) celery
2d7b11e8 (Eve   2019-05-14 07:00:04 -0700 2) lettuce
8c8cf113 (Frank 2019-05-14 07:00:05 -0700 3) peas
```

Now you decide that `fruits` and `veggies` should be combined into a single file called `produce`. How do you do this while still preserving the commit and histories of the contributing files?

The naïve way of combining the files would be to do it in a single commit:

```
cat fruits veggies > produce
git rm fruits veggies
git add produce
git commit --author="Greg <greg>" -m "combine"
```

The resulting file gets blamed like this:

```
eefddfb1 produce (Greg  2019-05-14 07:01:00 -0700 1) apple
eefddfb1 produce (Greg  2019-05-14 07:01:00 -0700 2) grape
eefddfb1 produce (Greg  2019-05-14 07:01:00 -0700 3) orange
7a542f13 veggies (David 2019-05-14 07:00:03 -0700 4) celery
2c258db0 veggies (Eve   2019-05-14 07:00:04 -0700 5) lettuce
87296161 veggies (Frank 2019-05-14 07:00:05 -0700 6) peas
```

The history from `veggies` was preserved, but the history from `fruits` was not. What git saw in the commit was that one file appeared and two files vanished. The rename detection machinery kicked in and decided that since the majority of the `produce` file matches the `veggies` file, it infers that what you did was delete the `fruits` file, renamed the `veggies` file to `produce`, and then added three new lines to the top of `produce`.

You can tweak the `git blame` algorithms with options like `-M` and `-C` to get it to try harder, but in practice, you don't often have control over those options: The `git blame` may be performed on a server, and the results reported back to you on a web page. Or the `git blame` is performed by a developer sitting at another desk (whose command line options you don't get to control), and poor Greg has to deal with all the tickets that get assigned to him from people who used the `git blame` output to figure out who introduced the line that's causing problems.

What we want is a way to get `git blame` to report the correct histories for both the fruits and the vegetables.

The trick is to use a merge. Let's reset back to the original state.

```
git reset --hard ready
```

We set up two branches. In one branch, we rename `veggies` to `produce`. In the other branch, we rename `fruits` to `produce`.

```
git checkout -b rename-veggies
git mv veggies produce
git commit --author="Greg <greg>" -m "rename veggies to produce"

git checkout -
git mv fruits produce
git commit --author="Greg <greg>" -m "rename fruits to produce"

git merge -m "combine fruits and veggies" rename-veggies
```

The merge fails with a rename-rename conflict:

```
CONFLICT (rename/rename):
Rename fruits->produce in HEAD.
Rename veggies->produce in rename-veggies

Renaming fruits to produce~HEAD
and veggies to produce~rename-veggies instead

Automatic merge failed; fix conflicts and then commit the result.
```

Update: Version 2.25.1 changed what happens in the case of a rename/rename conflict.

```
CONFLICT (rename/rename):
Rename fruits->produce in HEAD.
Rename veggies->produce in rename-veggies

Auto-merging produce

Automatic merge failed; fix conflicts and then commit the result.
```

At this point, you create the resulting `produce` file by combining the two originals.

If running pre-2.25.1:

```
cat "produce~HEAD" "produce~rename-veggies" >produce
```

If running post-2.25.1:

```
git cat-file --filters HEAD:produce >produce
git cat-file --filters rename-veggies:produce >>produce
```

Once you've generated the combined file, you can treat the file as resolved.

```
git add produce
git merge --continue
```

The resulting `produce` file was created by a merge, so git knows to look in both parents of the merge to learn what happened. And that's where it sees that each parent contributed half of the file, and it also sees that the files in each branch were themselves created via renames of other files, so it can chase the history back into both of the original files.

```
^fa19403 fruits  (Alice 2019-05-14 07:00:00 -0700 1) apple
00ef7240 fruits  (Bob   2019-05-14 07:00:01 -0700 2) grape
10e90730 fruits  (Carol 2019-05-14 07:00:02 -0700 3) orange
7a542f13 veggies (David 2019-05-14 07:00:03 -0700 4) celery
2c258db0 veggies (Eve   2019-05-14 07:00:04 -0700 5) lettuce
87296161 veggies (Frank 2019-05-14 07:00:05 -0700 6) peas
```

Magic! Greg is nowhere to be found in the blame history. Each line is correctly attributed to the person who introduced it in the original file, whether it's `fruits` or `veggies`. People investigating the `produce` file get a more accurate history of who last touched each line of the file.

Greg might need to do some editing to the two files before committing. Maybe the results need to be sorted, and maybe Greg figures he should add a header to remind people to keep it sorted.

```
>produce echo # keep sorted
git cat-file --filters HEAD:produce >>produce
git cat-file --filters rename-veggies:produce >>produce
sort -o produce produce
git add produce
git merge --continue

git blame produce

057507c7 produce (Greg  2019-05-14 07:01:00 -0700 1) # keep sorted
^943c65d fruits  (Alice 2019-05-14 07:00:00 -0700 2) apple
cfce62ae veggies (David 2019-05-14 07:00:03 -0700 3) celery
43c9aeb6 fruits  (Bob   2019-05-14 07:00:01 -0700 4) grape
5f60490e veggies (Eve   2019-05-14 07:00:04 -0700 5) lettuce
143eb20f fruits  (Carol 2019-05-14 07:00:02 -0700 6) orange
75a1ad0c veggies (Frank 2019-05-14 07:00:05 -0700 7) peas
```

For best results, your rename commit should be a pure rename. Resist the tempotation to edit the file's contents at the same time you rename it. A pure rename ensure that git's rename detection will find the match. If you edit the file in the same commit as the rename, then whether the rename is detected as such will depend on git's "similar files" heuristic.[1] If you need to edit the file as well as rename it, do it in two separate commits: One for the rename, another for the edit.

Wait, we didn't use `git commit-tree` yet. What's this doing in the *Mundane git commit-tree tricks* series?

We'll add `commit-tree` to the mix next time. Today was groundwork, but this is a handy technique to keep in your bag of tricks, even if you never get around to the `commit-tree` part.

[1] If you cross the `merge.renameLimit`, then git won't look for similar files; it requires exact matches. The Windows repo is so large that the rename limit is easily exceeded. The "similar files" detector is $O(m \times n)$ in the number of files changed in the two parents, and when your repo has 3 million files, that quadratic growth becomes a problem.

Raymond Chen

Follow