7.6 Git Tools - Rewriting History

Rewriting History

Many times, when working with Git, you may want to revise your local commit history. One of the great things about Git is that it allows you to make decisions at the last possible moment. You can decide what files go into which commits right before you commit with the staging area, you can decide that you didn't mean to be working on something yet with `git stash`, and you can rewrite commits that already happened so they look like they happened in a different way. This can involve changing the order of the commits, changing messages or modifying files in a commit, squashing together or splitting apart commits, or removing commits entirely — all before you share your work with others.

In this section, you'll see how to accomplish these tasks so that you can make your commit history look the way you want before you share it with others.

Don't push your work until you're happy with it

Note One of the cardinal rules of Git is that, since so much work is local within your clone, you have a great deal of freedom to rewrite your history *locally*. However, once you push your work, it is a different story entirely, and you should consider pushed work as final unless you have good reason to change it. In short, you should avoid pushing your work until you're happy with it and ready to share it with the rest of the world.

Changing the Last Commit

Changing your most recent commit is probably the most common rewriting of history that you'll do. You'll often want to do two basic things to your last commit: simply change the commit message, or change the actual content of the commit by adding, removing and modifying files.

If you simply want to modify your last commit message, that's easy:

```
$ git commit --amend
```

The command above loads the previous commit message into an editor session, where you can make changes to the message, save those changes and exit. When you save and close the editor, the editor writes a new commit containing that updated commit message and makes it your new last commit.

If, on the other hand, you want to change the actual *content* of your last commit, the process works basically the same way — first make the changes you think you forgot, stage those changes, and the subsequent `git commit --amend` *replaces* that last commit with your new, improved commit.

You need to be careful with this technique because amending changes the SHA-1 of the commit. It's like a very small rebase — don't amend your last commit if you've already pushed it.

An amended commit may (or may not) need an amended commit message

When you amend a commit, you have the opportunity to change both the commit message and the content of the commit. If you amend the content of the commit substantially, you should almost certainly update the commit message to reflect that amended content.

Tip

On the other hand, if your amendments are suitably trivial (fixing a silly typo or adding a file you forgot to stage) such that the earlier commit message is just fine, you can simply make the changes, stage them, and avoid the unnecessary editor session entirely with:

```
$ git commit --amend --no-edit
```

Changing Multiple Commit Messages

To modify a commit that is farther back in your history, you must move to more complex tools. Git doesn't have a modify-history tool, but you can use the rebase tool to rebase a series of commits onto the HEAD they were originally based on instead of moving them to another one. With the interactive rebase tool, you can then stop after each commit you want to modify and change the message, add files, or do whatever you wish. You can run rebase interactively by adding the `-i` option to `git rebase`. You must indicate how far back you want to rewrite commits by telling the command which commit to rebase onto.

For example, if you want to change the last three commit messages, or any of the commit messages in that group, you supply as an argument to `git rebase -i` the parent of the last commit you want to edit, which is `HEAD~2^` or `HEAD~3`. It may be easier to remember the `~3` because you're trying to edit the last three commits, but keep in mind that you're actually designating four commits ago, the parent of the last commit you want to edit:

```
$ git rebase -i HEAD~3
```

Remember again that this is a rebasing command — every commit in the range `HEAD~3..HEAD` with a changed message *and all of its descendants* will be rewritten. Don't include any commit you've already pushed to a central server — doing so will confuse other developers by providing an alternate version of the same change.

Running this command gives you a list of commits in your text editor that looks something like this:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

It's important to note that these commits are listed in the opposite order than you normally see them using the `log` command. If you run a `log`, you see something like this:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d Add cat-file
310154e Update README formatting and add blame
f7f3f6d Change my name a bit
```

Notice the reverse order. The interactive rebase gives you a script that it's going to run. It will start at the commit you specify on the command line (`HEAD~3`) and replay the changes introduced in each of these

commits from top to bottom. It lists the oldest at the top, rather than the newest, because that's the first one it will replay.

You need to edit the script so that it stops at the commit you want to edit. To do so, change the word `pick' to the word `edit' for each of the commits you want the script to stop after. For example, to modify only the third commit message, you change the file to look like this:

```
edit f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

When you save and exit the editor, Git rewinds you back to the last commit in that list and drops you on the command line with the following message:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... Change my name a bit
You can amend the commit now, with

       git commit --amend

Once you're satisfied with your changes, run

       git rebase --continue
```

These instructions tell you exactly what to do. Type:

```
$ git commit --amend
```

Change the commit message, and exit the editor. Then, run:

```
$ git rebase --continue
```

This command will apply the other two commits automatically, and then you're done. If you change pick to edit on more lines, you can repeat these steps for each commit you change to edit. Each time, Git will stop, let you amend the commit, and continue when you're finished.

Reordering Commits

You can also use interactive rebases to reorder or remove commits entirely. If you want to remove the "Add cat-file" commit and change the order in which the other two commits are introduced, you can change the rebase script from this:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

to this:

```
pick 310154e Update README formatting and add blame
pick f7f3f6d Change my name a bit
```

When you save and exit the editor, Git rewinds your branch to the parent of these commits, applies `310154e`and then `f7f3f6d`, and then stops. You effectively change the order of those commits and remove the "Add cat-file" commit completely.

Squashing Commits

It's also possible to take a series of commits and squash them down into a single commit with the interactive rebasing tool. The script puts helpful instructions in the rebase message:

```
#
# Commands:
```

```
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

If, instead of “pick” or “edit”, you specify “squash”, Git applies both that change and the change directly before it and makes you merge the commit messages together. So, if you want to make a single commit from these three commits, you make the script look like this:

```
pick f7f3f6d Change my name a bit
squash 310154e Update README formatting and add blame
squash a5f4a0d Add cat-file
```

When you save and exit the editor, Git applies all three changes and then puts you back into the editor to merge the three commit messages:

```
# This is a combination of 3 commits.
# The first commit's message is:
Change my name a bit

# This is the 2nd commit message:

Update README formatting and add blame

# This is the 3rd commit message:

Add cat-file
```

When you save that, you have a single commit that introduces the changes of all three previous commits.

Splitting a Commit

Splitting a commit undoes a commit and then partially stages and commits as many times as commits you want to end up with. For example, suppose you want to split the middle commit of your three commits. Instead of “Update README formatting and add blame”, you want to split it into two commits: “Update README formatting” for the first, and “Add blame” for the second. You can do that in the `rebase -i` script by changing the instruction on the commit you want to split to “edit”:

```
pick f7f3f6d Change my name a bit
edit 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

Then, when the script drops you to the command line, you reset that commit, take the changes that have been reset, and create multiple commits out of them. When you save and exit the editor, Git rewinds to the parent of the first commit in your list, applies the first commit (`f7f3f6d`), applies the second (`310154e`), and drops you to the console. There, you can do a mixed reset of that commit with `git reset HEAD^`, which effectively

undoes that commit and leaves the modified files unstaged. Now you can stage and commit files until you have several commits, and run `git rebase --continue` when you're done:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'Update README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'Add blame'
$ git rebase --continue
```

Git applies the last commit (`a5f4a0d`) in the script, and your history looks like this:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd Add cat-file
9b29157 Add blame
35cfb2b Update README formatting
f7f3f6d Change my name a bit
```

This changes the SHA-1s of the three most recent commits in your list, so make sure no changed commit shows up in that list that you've already pushed to a shared repository. Notice that the last commit (`f7f3f6d`) in the list is unchanged. Despite this commit being shown in the script, because it was marked as "pick" and was applied prior to any rebase changes, Git leaves the commit unmodified.

Deleting a commit

If you want to get rid of a commit, you can delete it using the `rebase -i` script. In the list of commits, put the word "drop" before the commit you want to delete (or just delete that line from the rebase script):

```
pick 461cb2a This commit is OK
drop 5aecc10 This commit is broken
```

Because of the way Git builds commit objects, deleting or altering a commit will cause the rewriting of all the commits that follow it. The further back in your repo's history you go, the more commits will need to be recreated. This can cause lots of merge conflicts if you have many commits later in the sequence that depend on the one you just deleted.

If you get partway through a rebase like this and decide it's not a good idea, you can always stop. Type `git rebase --abort`, and your repo will be returned to the state it was in before you started the rebase.

If you finish a rebase and decide it's not what you want, you can use `git reflog` to recover an earlier version of your branch. See Data Recovery for more information on the `reflog` command.

Note Drew DeVault made a practical hands-on guide with exercises to learn how to use `git rebase`. You can find it at: https://git-rebase.io/

The Nuclear Option: filter-branch

There is another history-rewriting option that you can use if you need to rewrite a larger number of commits in some scriptable way — for instance, changing your email address globally or removing a file from every commit. The command is `filter-branch`, and it can rewrite huge swaths of your history, so you probably shouldn't use it unless your project isn't yet public and other people haven't based work off the commits you're about to rewrite. However, it can be very useful. You'll learn a few of the common uses so you can get an idea of some of the things it's capable of.

Caution `git filter-branch` has many pitfalls, and is no longer the recommended way to rewrite history. Instead, consider using `git-filter-repo`, which is a Python script that does a better job for most applications where you would normally turn to `filter-branch`. Its documentation and source code can be found at https://github.com/newren/git-filter-repo.

Removing a File from Every Commit

This occurs fairly commonly. Someone accidentally commits a huge binary file with a thoughtless `git add .`, and you want to remove it everywhere. Perhaps you accidentally committed a file that contained a password, and you want to make your project open source. `filter-branch` is the tool you probably want to use to scrub your entire history. To remove a file named `passwords.txt` from your entire history, you can use the `--tree-filter` option to `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

The `--tree-filter` option runs the specified command after each checkout of the project and then recommits the results. In this case, you remove a file called `passwords.txt` from every snapshot, whether it exists or not. If you want to remove all accidentally committed editor backup files, you can run something like `git filter-branch --tree-filter 'rm -f *~' HEAD`.

You'll be able to watch Git rewriting trees and commits and then move the branch pointer at the end. It's generally a good idea to do this in a testing branch and then hard-reset your `master` branch after you've determined the outcome is what you really want. To run `filter-branch` on all your branches, you can pass `--all` to the command.

Making a Subdirectory the New Root

Suppose you've done an import from another source control system and have subdirectories that make no sense (`trunk`, `tags`, and so on). If you want to make the `trunk` subdirectory be the new project root for every commit, `filter-branch` can help you do that, too:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Now your new project root is what was in the `trunk` subdirectory each time. Git will also automatically remove commits that did not affect the subdirectory.

Changing Email Addresses Globally

Another common case is that you forgot to run `git config` to set your name and email address before you started working, or perhaps you want to open-source a project at work and change all your work email addresses to your personal address. In any case, you can change email addresses in multiple commits in a batch with `filter-branch` as well. You need to be careful to change only the email addresses that are yours, so you use `--commit-filter`:

```
$ git filter-branch --commit-filter '
        if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
        then
                GIT_AUTHOR_NAME="Scott Chacon";
                GIT_AUTHOR_EMAIL="schacon@example.com";
                git commit-tree "$@";
        else
                git commit-tree "$@";
        fi' HEAD
```

This goes through and rewrites every commit to have your new address. Because commits contain the SHA-1 values of their parents, this command changes every commit SHA-1 in your history, not just those that have the matching email address.

prev | next