The 'git rebase -i' Command

Learn about the “git rebase -i” command and how it can help you to squash commits.

We'll cover the following

In order to squash a set of commits, you need:

- A reference to the last (**latest**) commit of the set you want to squash.
- A reference to the oldest (**first**) commit of the set you want to squash.

You already have a stable reference to the latest commit, i.e., where we are at the moment. Can you remember what it is?

You need a reference to the oldest commit in the history. Can you find a way to get it?

Reference to oldest commit

There are many ways to do it. The way we’re going to find it is to use the `git rev-list` command. You should be at the point where you’re comfortable reading up on a Git command, so go off and do that now.

OK? Now, run this command:

```
1   git rev-list --max-parents=0 HEAD
```

That gets you the original commit, and that's the method you're going to use here.

The following command will substitute the output of that command into the `git rebase` command below:

```
2   git rebase -i $(git rev-list --max-parents=0 HEAD) HEAD
```

Terminal 1

Terminal

Click to Connect...

Don't worry about the substitution part (the `$()` bit) of the above command if it doesn't make sense to you.

We have explicitly put `HEAD` in as the second argument. But technically, you don't need it, as Git assumes that `HEAD` was the second argument if none was given.

Follow the instructions and try to navigate your way to squashing the commits into a single one, and confirm you did it by using `git log`.

If you couldn't do it, then go to "Cleanup" below and start again.

If you still couldn't figure out what was going on, I'm going to explain a little more next.

The `git rebase -i` command explained

When you run the `git rebase -i` command, you are confronted with an editable file screen that shows the ten commits on ten lines. Each line looks like this:

```
pick c30a1bb commit:1
pick c30a1bb commit:2
```

You get a description of the rebase followed by a bunch of comment lines beginning with `#` that outline various commands that can be run on each commit:

```
    # Rebase 3922435..2816cf1 onto 3922435 (9 commands)
    #
    # Commands:
    # p, pick = use commit
    # r, reword = use commit, but edit the commit message
    # e, edit = use commit, but stop for amending
    # s, squash = use commit, but meld into previous commit
    # f, fixup = like "squash", but discard this commit's log message
    # x, exec = run command (the rest of the line) using shell
    # d, drop = remove commit
```

Finally, you get some helpful advice at the bottom:

```
    # These lines can be re-ordered; they are executed from top to bot
tom.
    #
    # 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
```

What's going on?

Well, rather than simply moving a set of commits from one base to another, you can now perform a specific action on each commit. The descriptions are fairly self-explanatory, but it's worth exploring and playing with them a little while you have the chance to do it without breaking anything.

The default is `pick` (or `p` for short), which means that the commit will be kept as it was before.

- We're concentrating on the most common task here, which is squashing. There are a few confusing things about this process that puzzled me at first glance.
- The first point is to note the time order of commits goes from top to bottom. It is fairly obvious here because you numbered the commits, but I often forget.
- The second point is that you can't squash *all* the commits. One must be picked in order that the rebase retains a single commit. Otherwise, there's nothing to squash to!

Other aspects worth noting here are that the `squash` and `fixup` options are the same, except that `fixup` does not give you the commit message in the following screen.

The `drop` option just removes the commit and all its changes from the history.

history.

Also, you still need the initial commit since you need a base commit to anchor your rebase to. This is why you end up with two commits in your repository after squashing into all but one of the offered commits.

Finally, when you have rebased (and no matter what you do with the commits), you always end up in a `detached HEAD` state. This is because the branch pointer for the local `master` has not been moved with the rebase. Remember that when you commit, the `HEAD` and branch pointer are automatically moved for you, but this is a convenience rebase does not provide for you.

Assuming you've squashed everything into the 10 commit, run these commands to make sure you're on the master branch and that the master branch is pointed to this new squashed commit:

```
3   git branch -f master
4   git checkout master
```

Terminal 1

Terminal

Note: Use the above terminal for all the commands below.

I use this pair of commands quite a lot, as this "move the branch pointer" pattern is a frequent requirement when people get into a mess with Git.

You will have seen output like this from the last command:

```
    Switched to branch 'master'
    Your branch and 'origin/master' have diverged,
    and have 1 and 10 different commits each, respectively.
        (use "git pull" to merge the remote branch into yours)
```

As always with Git, when it tells you something, it's really worth reading it carefully. Git is a person of few words, but you should listen to them!

Have a think about this now, and try to understand it before moving on.

How is this different from `git rebase`?

You may be wondering what the difference was between this `git rebase -i` command and the other one you used before without the `-i` flag.

Underneath it all, it is the same broader concept behind rebase: take a set of commits and do things with them. The interactive (`-i`) mode gives you more control over those individual commits.

If you get more advanced, you can both move *and* squash commits at the same time. But "one step at a time" is key when learning Git.

Pushing squashed commits

You may also be wondering why you did that clone of the `origin` repository before doing squashes.

What you're going to do now is try and push your squashed changes to the origin.

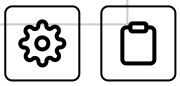
```
5   git push origin
```

That should have been rejected with an error message. Read the output carefully.

If you follow the instructions by issuing a `git pull` and merging the result:

```
6   git pull
```

```
6   git pull
```



And then repeat your `git push origin`:

```
7   git push origin
```

You will get a very long and probably quite hard-to-understand error message that tells you that the push was rejected.

If you understand what's going on, then that's great. If you didn't, we will explore it in more detail in the next chapter.

What's key to understanding is that you've tried to push a change to a repository that isn't configured to accept it. If you think about it, it makes sense: you've taken a Git repository that's being worked on, cloned it, and then tried to force a change on it. If the current user of that repository wasn't expecting changes, then that could be very disruptive!

So how do things get pushed normally to what people call "git servers" if all Git repositories are equal?

The answer is that there is a special type of Git repository designed for the purpose of being pushed to (rather than being worked on), and that's called a "bare repository." It is covered in the upcoming chapter.

At this point, you can try a "force" with the `-f` flag to try and override what is on the remote. But that won't work either.

```
8   git push -f origin
```

Why squash?

Finally, you might be wondering why you must squash when it causes all this bother.

Mostly, squashing is avoided when dealing with changes already seen on remote repositories. Instead, it is more commonly used when preparing changes made locally for submission to another repository.

changes made locally for submission to another repository.

If you do use it to squash and push to a remote repository (most likely a Git server repository), then you may need to force (`git -f`) a push if you have changed the history. If you have permission, it will go through. But you will have changed the history for others, which can cause much pain for them as they are trying to figure out why they are having to merge new changes into their cloned repositories.

In sum, squashing is dangerous. Think about how it might affect others before doing it!

Back

A Worked Example

Next

Introduction: Bare Repositories

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/the-git-rebase-i-command__squashing-commits__learn-git-the-hard-way)