Linux Fu: Deep Git Rebasing

Originalartikel

Backup

<html> If you spend much time helping people with word processor programs, you’ll find that many people don’t really use much of the product. They type, change fonts, save, and print. But cross-references? Indexing? Largely, those parts of the program go unused. I’ve noticed the same thing with Git. We all use it constantly. But do we? You clone a repo. Work on it. Maybe switch branches and create a pull request. That’s about 80% of what you want to do under normal circumstances. But what if you want to do something out of the ordinary? Git is very flexible, but you do have to know the magic incantations. For example, suppose you mess up a commit message — we never do that, of course, but just pretend. Or you accidentally added a file you didn’t want in the commit. Git has some very useful ways to deal with situations like this, especially the interactive rebase. <h2>Identify a Commit</h2> If you haven’t realized it, every version of your project in Git boils down to a commit. Branches and tags are just “pointers’ to commits. In addition, commits point to their parent commit. So, suppose you have the following sequence of commands: <pre>mkdir projectcd projectgit inittouch readme.mdgit add readme.mdgit commit -a -m „First Commit“</pre> So far, this is pretty standard stuff. Next, we are going to make our first change, and we’ll simulate an emacs backup file. This will be the first change we will commit. <pre>touch hackaday.txttouch hackaday.txt~git add hackaday*git commit -a -m „Add hackaday.text“</pre> Oops. We have two problems here, but for the sake of the example, suppose we only noticed the typo in the commit message (“text” instead of “txt”). If any of this doesn’t make sense, you might want to review the basics of Git before you keep going. The video below can help, although there are plenty of other options. If you’d rather read, there’s also the Pro Git book. <iframe class=„youtube-player c1“ width=„800“ height=„480“ src=„https://www.youtube.com/embed/RGOj5yH7evk?version=3&rel=1&showsearch=0&showinfo=1&iv_load_policy=1&fs=1&hl=en-US&autohide=2&wmode=transparent“ allowfullscreen=„allowfullscreen“ sandbox=„allow-scripts allow-same-origin allow-popups allow-presentation“>[embedded content]</iframe> <h2>Quick Fix</h2> If you need to fix the last commit, it is pretty easy. It helps to notice what the ID of each commit is. There’s a long ID string, but Git can show you the first few characters of it. Yours will be different, but when I did this, that last commit was 86d63c3. You can use these ids or tags like HEAD (the current commit) to go further up the git commit graph. If you ask git to show you what’s going on with “git log” you can see that the original first commit was 51a18eb. In addition, you can see that HEAD, the pointer to the tip of the commit graph, is pointing to 86d63c3. But since we’re just changing the last commit, you don’t need to know that. In this case, here’s what you can do: <pre>git commit –amend -m „Add hackaday.txt“</pre> If you do a

```
git log
```

now, you’ll see the commit message changed. However, since this is a new commit, the ID number changes also (for me, 9b0125c). That means if you have already pushed the commit to a remote repo, you shouldn’t do the amend. There are still only two commits, the original one and our new one with the corrected message. One problem solved, but we still have that backup file. Unfortunately, we don’t notice it until later. <h2>A New

Commit</h2> Next, we are going to get a Jolly Wrencher graphic in and commit again. <pre>cp ~/assets/wrencher.png .git add wrencher.pnggit commit -am 'Add logo'</pre> This is fine, and we now have three commits in our current branch. The problem is we need to fix the second one now. We now have three commits. The original initial commit was (for me) 51a18eb. The updated second commit is 9b0125c. The last commit with the logo is 3451549. To fix the commit, we will rebase the last commit. Remember that the last commit is “based” on the previous commit, which is based on the commit before that, and so on. To change our commit, we need to rebase to one commit beyond the one we want to change. One way to do that is to specify 51a183b. We can do that interactively with: <pre>git rebase -i 51a18eb</pre> This opens an editor with a “script”: <pre>pick 9b0125c Add Hackaday.txtpick 3451549 Add logo# Rebase 51a18eb..3451549 onto 51a18eb (2 commands)## 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 [-C | -c] <commit> = like „squash“ but keep only the previous# commit's log message, unless -C is used, in which case# keep only this commit's message; -c is same as -C but# opens the editor# 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.#</pre> The comments are helpful. By default, all the commits use the “pick” command which just keeps them. You can also do things like drop them or use reword to change a commit message. We want to do an edit, so you can change the first pick command to an edit command: <pre>edit 9b0125c Add Hackaday.txtpick 3451549 Add logo</pre> When you exit your editor, you get a helpful message: <pre>Stopped at 9b0125c... Add Hackaday.txt You can amend the commit now, with git commit –amend Once you are satisfied with your changes, run git rebase –continue</pre> Looking around now, you’ll see everything is as it was for that commit. That is, there’s no PNG file, and you have both hackaday.txt files. Let’s fix it: <pre>git rm hackaday.txt~git commit –amendgit rebase –continue</pre> You can always use a git status to see what git thinks you need to do next. After these commands, you still have three commits, but the accidental add of hackaday.txt~ has vanished. As you can see, there are other commands, too. You can merge multiple commits together. You can also squash them or fix them up. These essentially turn one commit into two. The difference is a squash gives you a chance to keep or change the commit message, while fixup keeps only one of the original messages. Remember that these are both different from a merge, which creates a new commit from multiple parent commits. A squash or a fixup converts multiple commits into a single commit. You can also hit the panic button with “git rebase –abort.” <h2>Picking Your Commit</h2> Rember that each commit has an ID, but you also have tags and even relative indexing available. You could also use the notation HEAD~2 (in this case). This tells git to start at the HEAD and go back two generations. If you are familiar with merge commits, this assumes you are going back in the same branch. When a merge commit gives you a choice of parents, you can also use the notation HEAD^ to pick one of the parents. You can even mix and match these. But you can also always get the ID from

```
git log
```

and use that. Or, you can use a tag if you’ve tagged a certain commit. Just make

sure you are working with the commit before the last commit you want to edit. In this case, we wanted to edit HEAD~ (or HEAD~1, if you prefer), so we had to rebase HEAD~2. If you really want to edit using the commit number you actually want to edit, just put a ~ after it. That selects the parent of the specified ID. <h2>Non-Interactive Mode</h2> You can, of course, do rebasing without the interactive mode, but it is a lot of work. The interactive mode is good, too, for things like splitting a commit into multiple commits. That’s because when you edit a commit, you can actually add multiple commits as part of the edit. You’ll find if you start rebasing, you’ll use

```
git log
```

a lot. There is a post that shows how to make better-looking output if you prefer. Turns out, you can use git for a lot of things. If you crave something simpler, try the gitless shell that runs over git. </html>