**How to reset, revert, and return to previous states in Git**

**Reset**

Let's start with the Git command **reset**. Practically, you can think of it as a "rollback"—it points your local environment back to a previous commit. By "local environment," we mean your local repository, staging area, and working directory.

Take a look at Figure 1. Here we have a representation of a series of commits in Git. A branch in Git is simply a named, movable pointer to a specific commit. In this case, our branch *master* is a pointer to the latest commit in the chain.

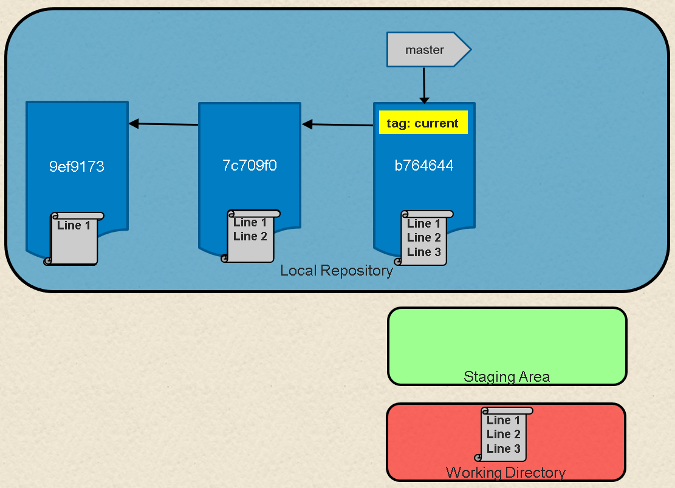


Fig. 1: Local Git environment with repository, staging area, and working directory

If we look at what's in our *master* branch now, we can see the chain of commits made so far.

**$ git log --oneline  
b764644 File with three lines  
7c709f0 File with two lines  
9ef9173 File with one line**

Programming and development

* [New Python content](https://opensource.com/tags/python?src=programming_resource_menu1)
* [Our latest JavaScript articles](https://opensource.com/tags/javascript?src=programming_resource_menu2)
* [Recent Perl posts](https://opensource.com/tags/perl?src=programming_resource_menu3)
* [Red Hat Developers Blog](https://developers.redhat.com/?intcmp=7016000000127cYAAQ&src=programming_resource_menu4)

What happens if we want to roll back to a previous commit. Simple—we can just move the branch pointer. Git supplies the **reset** command to do this for us. For example, if we want to reset *master* to point to the commit two back from the current commit, we could use either of the following methods:

**$ git reset 9ef9173** (using an absolute commit SHA1 value 9ef9173)

or

**$ git reset current~2** (using a relative value -2 before the "current" tag)

Figure 2 shows the results of this operation. After this, if we execute a **git log** command on the current branch (*master*), we'll see just the one commit.

**$ git log --oneline  
9ef9173 File with one line**

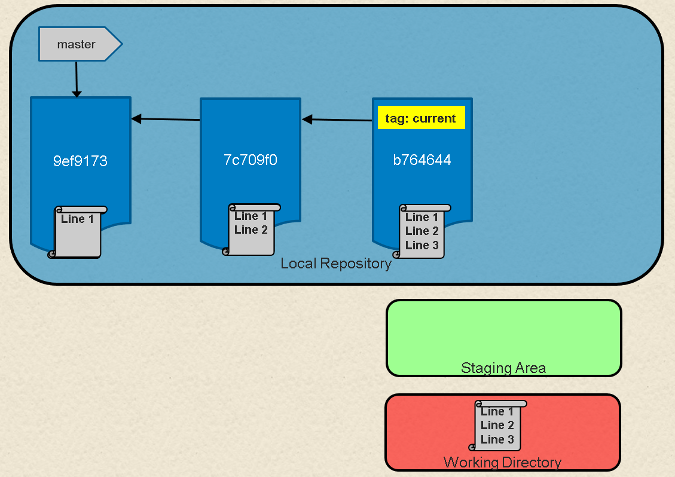


Fig. 2: After**reset**

The **git reset** command also includes options to update the other parts of your local environment with the contents of the commit where you end up. These options include: **hard** to reset the commit being pointed to in the repository, populate the working directory with the contents of the commit, and reset the staging area; **soft** to only reset the pointer in the repository; and **mixed** (the default) to reset the pointer and the staging area.

Using these options can be useful in targeted circumstances such as **git reset --hard <commit sha1 | reference>.** This overwrites any local changes you haven't committed. In effect, it resets (clears out) the staging area and overwrites content in the working directory with the content from the commit you reset to. Before you use the **hard**option, be sure that's what you really want to do, since the command overwrites any uncommitted changes.

**Revert**

The net effect of the **git revert** command is similar to reset, but its approach is different. Where the **reset** command moves the branch pointer back in the chain (typically) to "undo" changes, the **revert** command adds a new commit at the end of the chain to "cancel" changes. The effect is most easily seen by looking at Figure 1 again. If we add a line to a file in each commit in the chain, one way to get back to the version with only two lines is to reset to that commit, i.e., **git reset HEAD~1**.

Another way to end up with the two-line version is to add a new commit that has the third line removed—effectively canceling out that change. This can be done with a **git revert** command, such as:

**$ git revert HEAD**

Because this adds a new commit, Git will prompt for the commit message:

**Revert "File with three lines"  
  
This reverts commit b764644bad524b804577684bf74e7bca3117f554.  
  
# Please enter the commit message for your changes. Lines starting  
# with '#' will be ignored, and an empty message aborts the commit.  
# On branch master  
# Changes to be committed:  
#       modified:   file1.txt  
#**

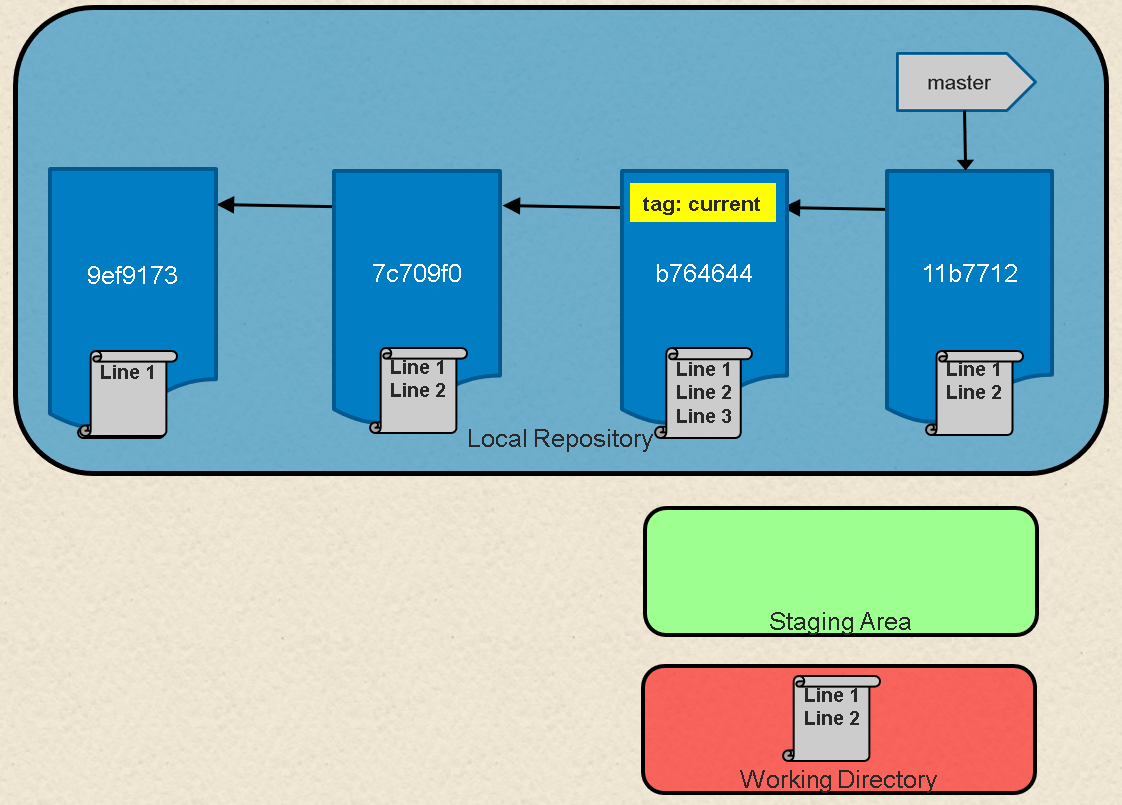
Figure 3 (below) shows the result after the **revert** operation is completed.

If we do a **git log** now, we'll see a new commit that reflects the contents before the previous commit.

**$ git log --oneline  
11b7712 Revert "File with three lines"  
b764644 File with three lines  
7c709f0 File with two lines  
9ef9173 File with one line**

Here are the current contents of the file in the working directory:

**$ cat <filename>  
Line 1  
Line 2**



**Revert or reset?**

Why would you choose to do a **revert** over a **reset** operation? If you have already pushed your chain of commits to the remote repository (where others may have pulled your code and started working with it), a revert is a nicer way to cancel out changes for them. This is because the Git workflow works well for picking up additional commits at the end of a branch, but it can be challenging if a set of commits is no longer seen in the chain when someone resets the branch pointer back.

This brings us to one of the fundamental rules when working with Git in this manner: Making these kinds of changes in your *local repository* to code you haven't pushed yet is fine. But avoid making changes that rewrite history if the commits have already been pushed to the remote repository and others may be working with them.

In short, if you rollback, undo, or rewrite the history of a commit chain that others are working with, your colleagues may have a lot more work when they try to merge in changes based on the original chain they pulled. If you must make changes against code that has already been pushed and is being used by others, consider communicating before you make the changes and give people the chance to merge their changes first. Then they can pull a fresh copy after the infringing operation without needing to merge.

You may have noticed that the original chain of commits was still there after we did the reset. We moved the pointer and reset the code back to a previous commit, but it did not delete any commits. This means that, as long as we know the original commit we were pointing to, we can "restore" back to the previous point by simply resetting back to the original head of the branch:

**git reset <sha1 of commit>**

A similar thing happens in most other operations we do in Git when commits are replaced. New commits are created, and the appropriate pointer is moved to the new chain. But the old chain of commits still exists.

**Rebase**

Now let's look at a branch rebase. Consider that we have two branches—*master* and *feature*—with the chain of commits shown in Figure 4 below. *Master* has the chain **C4->C2->C1->C0** and *feature* has the chain **C5->C3->C2->C1->C0**.

[**gitcommands4\_commits-branches.png**](https://opensource.com/file/401141)

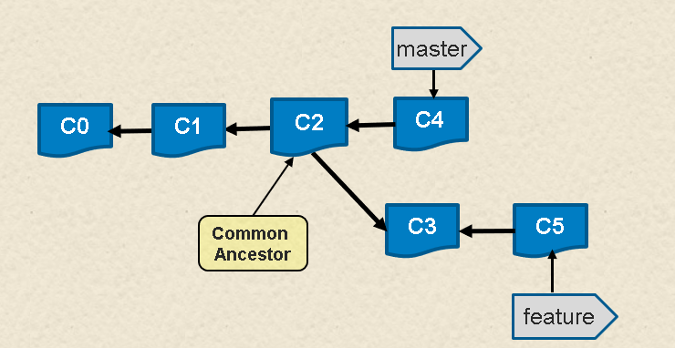


Fig. 4: Chain of commits for branches*master*and*feature*

If we look at the log of commits in the branches, they might look like the following. (The **C**designators for the commit messages are used to make this easier to understand.)

**$ git log --oneline master  
6a92e7a C4  
259bf36 C2  
f33ae68 C1  
5043e79 C0  
  
$ git log --oneline feature  
79768b8 C5  
000f9ae C3  
259bf36 C2  
f33ae68 C1  
5043e79 C0**

I tell people to think of a rebase as a "merge with history" in Git. Essentially what Git does is take each different commit in one branch and attempt to "replay" the differences onto the other branch.

So, we can rebase a feature onto *master* to pick up **C4** (e.g., insert it into feature's chain). Using the basic Git commands, it might look like this:

**$ git checkout feature  
$ git rebase master  
  
First, rewinding head to replay your work on top of it...  
Applying: C3  
Applying: C5**

Afterward, our chain of commits would look like Figure 5.

[**gitcommands5\_commits-rebase.png**](https://opensource.com/file/401146)

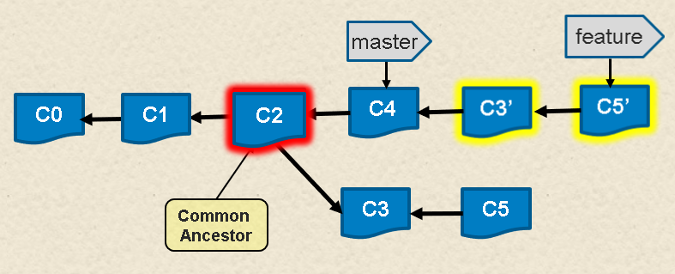


Fig. 5: Chain of commits after the**rebase**command

Again, looking at the log of commits, we can see the changes.

**$ git log --oneline master  
6a92e7a C4  
259bf36 C2  
f33ae68 C1  
5043e79 C0  
  
$ git log --oneline feature  
c4533a5 C5  
64f2047 C3  
6a92e7a C4  
259bf36 C2  
f33ae68 C1  
5043e79 C0**

Notice that we have **C3'** and **C5'**—new commits created as a result of making the changes from the originals "on top of" the existing chain in *master*. But also notice that the "original" **C3** and **C5** are still there—they just don't have a branch pointing to them anymore.

If we did this rebase, then decided we didn't like the results and wanted to undo it, it would be as simple as:

**$ git reset 79768b8**

With this simple change, our branch would now point back to the same set of commits as before the **rebase** operation—effectively undoing it (Figure 6).

[**gitcommands6\_rebase-undo.png**](https://opensource.com/file/401151)

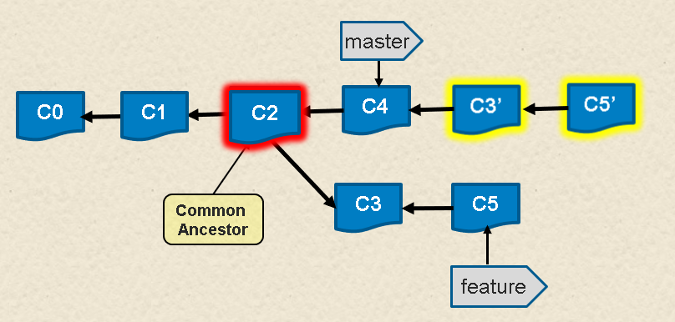


Fig. 6: After undoing the**rebase**operation

What happens if you can't recall what commit a branch pointed to before an operation? Fortunately, Git again helps us out. For most operations that modify pointers in this way, Git remembers the original commit for you. In fact, it stores it in a special reference named **ORIG\_HEAD**within the **.git** repository directory. That path is a file containing the most recent reference before it was modified. If we **cat** the file, we can see its contents.

**$ cat .git/ORIG\_HEAD  
79768b891f47ce06f13456a7e222536ee47ad2fe**

We could use the **reset** command, as before, to point back to the original chain. Then the log would show this:

**$ git log --oneline feature  
79768b8 C5  
000f9ae C3  
259bf36 C2  
f33ae68 C1  
5043e79 C0**

Another place to get this information is in the reflog. The reflog is a play-by-play listing of switches or changes to references in your local repository. To see it, you can use the **git reflog** command:

**$ git reflog  
79768b8 HEAD@{0}: reset: moving to 79768b  
c4533a5 HEAD@{1}: rebase finished: returning to refs/heads/feature  
c4533a5 HEAD@{2}: rebase: C5  
64f2047 HEAD@{3}: rebase: C3  
6a92e7a HEAD@{4}: rebase: checkout master  
79768b8 HEAD@{5}: checkout: moving from feature to feature  
79768b8 HEAD@{6}: commit: C5  
000f9ae HEAD@{7}: checkout: moving from master to feature  
6a92e7a HEAD@{8}: commit: C4  
259bf36 HEAD@{9}: checkout: moving from feature to master  
000f9ae HEAD@{10}: commit: C3  
259bf36 HEAD@{11}: checkout: moving from master to feature  
259bf36 HEAD@{12}: commit: C2  
f33ae68 HEAD@{13}: commit: C1  
5043e79 HEAD@{14}: commit (initial): C0**

You can then reset to any of the items in that list using the special relative naming format you see in the log:

**$ git reset HEAD@{1}**

Once you understand that Git keeps the original chain of commits around when operations "modify" the chain, making changes in Git becomes much less scary. This is one of Git's core strengths: being able to quickly and easily try things out and undo them if they don't work.