We select and review products independently. When you purchase through our links we may earn a commission. Learn more.

How-To Geek

DevOps 〉 Programming

How Does Git Reset Actually Work? Soft, Hard, and Mixed Resets Explained

ANTHONY HEDDINGS
JUL 27, 2021, 7:00 AM EST | 3 MIN READ

`git reset` is a powerful command that can modify the history of your Git repository and fix mistakes you made. However, while the name makes it seem scary to use, it's actually pretty easy, so let's talk about how it works.

What Is The Git HEAD?

Before understanding how resets work, we need to talk about the Git HEAD.

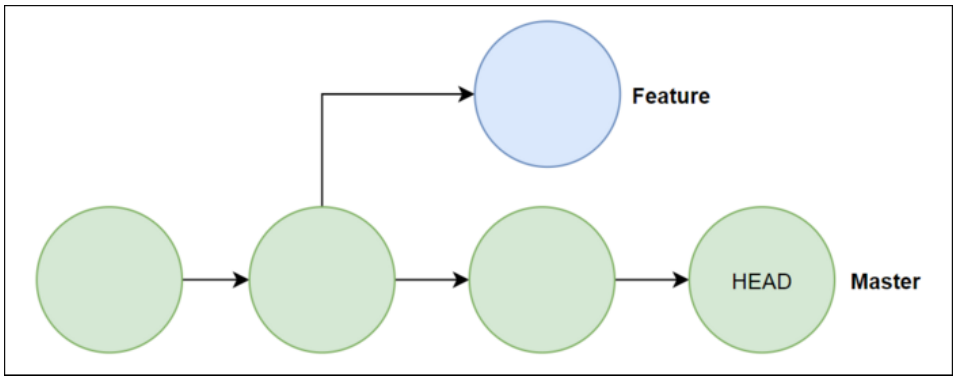
"HEAD" is simply an alias for your current working commit. You can think of it like a record player head, where the position of it determines what data is being used. If you're currently using the `master` branch, the HEAD will be at the latest commit in that branch.

Despite being an alias for a commit, the HEAD doesn't actually point directly to a commit most of the time. It points towards a branch, and automatically uses the latest commit. It can also point towards a commit directly, which is known as "detached HEAD," though that doesn't matter for `git reset`.

How Does Git Reset Work?

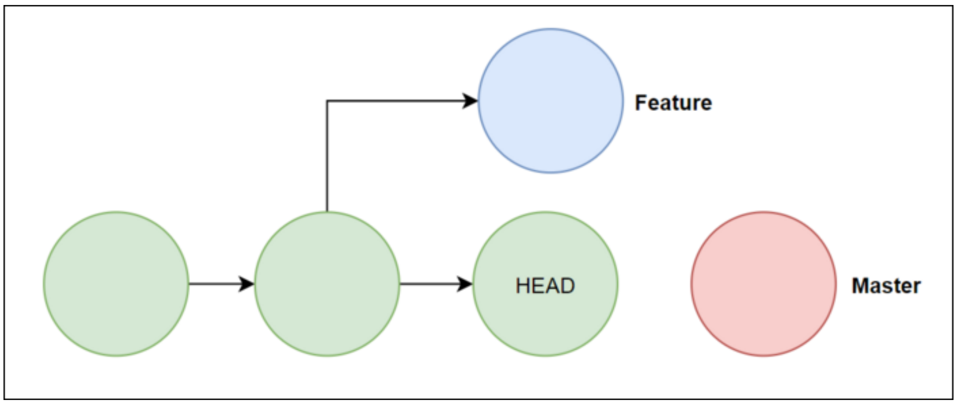
Git's commit history is stored as a tree of commits, and is intended to be immutable for the most part. However, sometimes it's necessary to modify this history, and that's where `git reset` comes into play.

Each commit links to the commit before it, and can branch off into different limbs that will eventually get merged back into the `master` branch. In either case, the HEAD will usually point towards the latest commit in whatever branch you're working on.



So what happens when you make a commit you want to revert?

Well, running `git reset` basically moves the HEAD back, and leaves all the commits in front of it hanging. This rewrites the usually immutable Git history to get rid of the commits in front of the HEAD.



This can be quite useful for a lot of reasons. Perhaps you made a commit, then made some additional changes, and wanted to push the whole thing as one commit. You could `git reset` back to the previous commit, and then re-commit correctly.

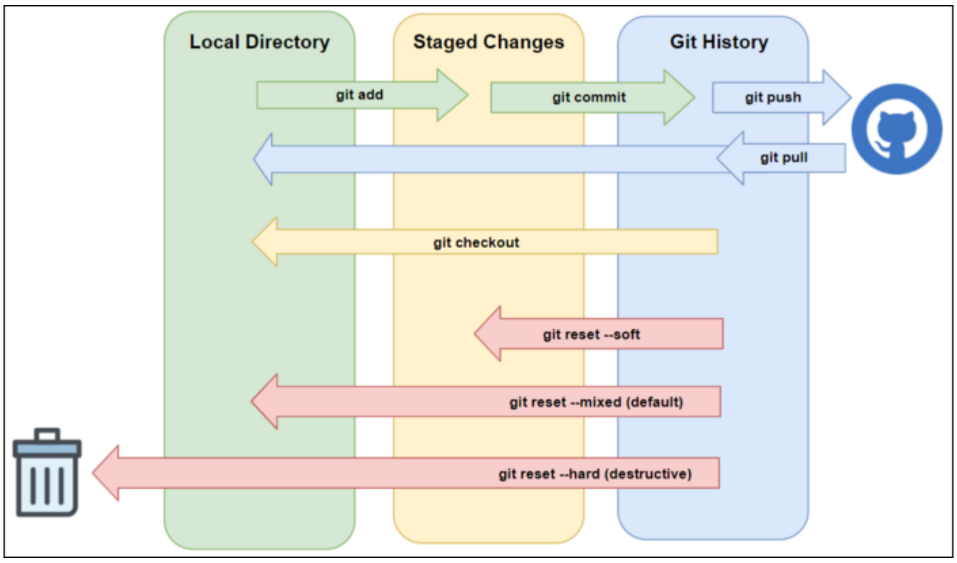
Or perhaps you accidentally made a commit that included some changes you didn't want to track. This can be very hard to figure out if you don't know how `git reset` works, but resetting the HEAD and then only staging the correct changes will achieve this. Note that this is different from `git revert`, which reverses commits.

RELATED: ***How To Recover Reverted Commits In a Git Repository***

There are three different kinds of resets in Git, and they all differ based on how they handle the commits that get left hanging. They all rewrite Git history, and they all move the HEAD back, but they deal with the changes differently:

- `git reset --soft`, which will keep your files, and stage all changes back automatically.
- `git reset --hard`, which will *completely destroy any changes and remove them from the local directory*. Only use this if you know what you're doing.
- `git reset --mixed`, which is the default, and keeps all files the same but unstages the changes. This is the most flexible option, but despite the name, it doesn't modify files.

The difference between soft and mixed is whether or not the changes are staged. Staged is basically an in-between zone between the local directory and Git history. `git add` stages files, and `git commit` writes them to the history. In either case, the local directory is unaffected, it just changes the state of Git's tracking of those changes.

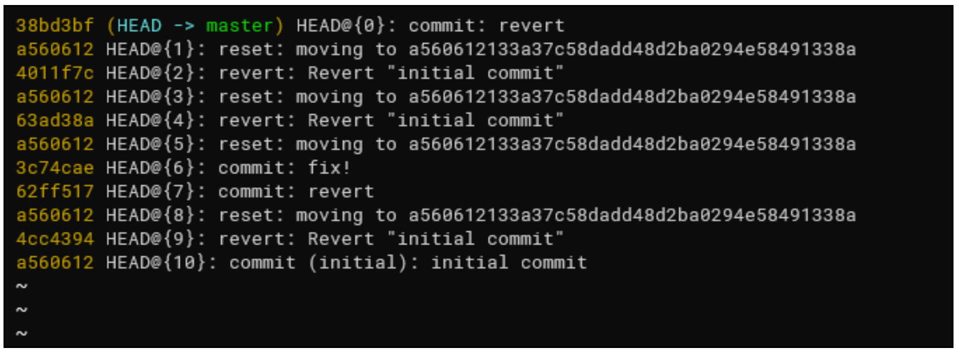


Basically, Soft and Mixed resets are mostly the same and allow you to keep the changes, and Hard resets will completely set your local directory back to where it was at the time of the commit.

Using Git Reset

Once you understand what's going on, actually using `git reset` is incredibly easy. To reset, you'll need a reference to the commit you want to move back to. You can get this by running `reflog`:

```
git reflog
```



```
38bd3bf (HEAD -> master) HEAD@{0}: commit: revert
a560612 HEAD@{1}: reset: moving to a560612133a37c58dadd48d2ba0294e58491338a
4011f7c HEAD@{2}: revert: Revert "initial commit"
a560612 HEAD@{3}: reset: moving to a560612133a37c58dadd48d2ba0294e58491338a
63ad38a HEAD@{4}: revert: Revert "initial commit"
a560612 HEAD@{5}: reset: moving to a560612133a37c58dadd48d2ba0294e58491338a
3c74cae HEAD@{6}: commit: fix!
62ff517 HEAD@{7}: commit: revert
a560612 HEAD@{8}: reset: moving to a560612133a37c58dadd48d2ba0294e58491338a
4cc4394 HEAD@{9}: revert: Revert "initial commit"
a560612 HEAD@{10}: commit (initial): initial commit
~
~
~
```

Copy the seven digit code on the right. If you just got stuck in `vim`, press Q, and maybe run `git config --global core.editor "nano"`.

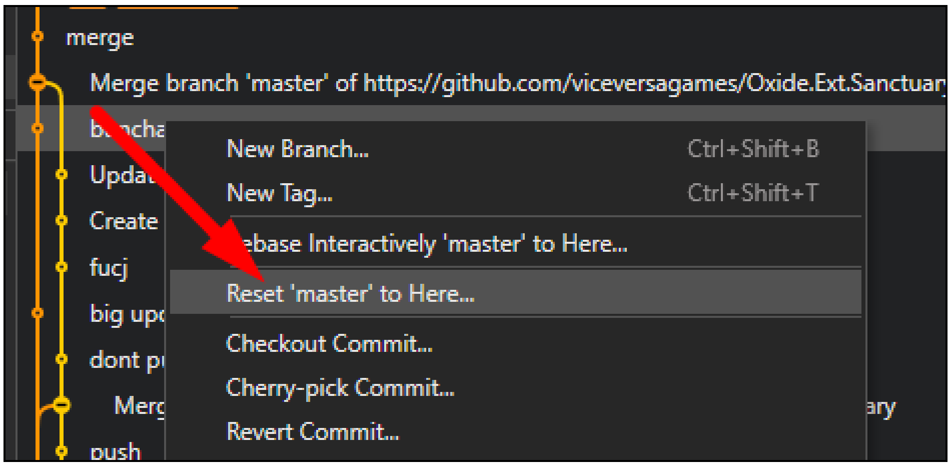
Then, you can reset back to the target commit:

```
git reset --mixed a560612
```

Or, you can target commits based on their position relative to the HEAD. The following command targets the commit right before the HEAD, which is useful shorthand if you need to reset the latest commit:

```
git reset --mixed HEAD~
```

If you're using a Git client like Fork, resetting is as easy as right clicking the target commit and selecting "Reset":


merge
Merge branch 'master' of https://github.com/viceversagames/Oxide.Ext.Sanctuar
New Branch... Ctrl+Shift+B
New Tag... Ctrl+Shift+T
ebase Interactively 'master' to Here...
Reset 'master' to Here...
Checkout Commit...
Cherry-pick Commit...
Revert Commit...

Should You Ever Hard Reset?

You should really only be using soft or mixed resets, but, if you messed up your repository so bad that you need to completely reset it, you can use the following sequence of commands to fully set it back to normal.

```
git fetch origin
git checkout master
git reset --hard origin/master
git clean -d --force
```

Of course, you could always pull the classic "delete your repository and re-clone from Github," but this way is at least Git approved, and only resets a single branch.

ANTHONY HEDDINGS

Anthony Heddings is the resident cloud engineer for LifeSavvy Media, a technical writer, programmer, and an expert at Amazon's AWS platform. He's written hundreds of articles for How-To Geek and CloudSavvy IT that have been read millions of times. **READ FULL BIO »**

How-To Geek is where you turn when you want experts to explain technology. Since we launched in 2006, our articles have been read more than 1 billion times. Want to know more?