# Git Internals

## Browsing Git

Git also gives you access to a number of lower level tools that can be used to browse the repository, inspect the status and contents of any of the objects, and are generally helpful for inspection and debug­ging.

The git show command is really useful for presenting any of the objects in a very human readable format. Running this command on a file will simply output the contents of the file. Running it on a tree will just give you the filenames of the contents of that tree, but none of its subtrees.

If you call it on a tree-ish that is a commit object, you will get simple information about the commit (the author, message, date, etc) and a diff of what changed between that commit and its parents.

$ git show master^

Instead of the git show command, it’s generally more useful to use the lower level git ls-tree command to view trees, because it gives you the SHA-1s of all the blobs and trees that it points to.

$ git ls-tree master^{tree}

You can also run this command recursively, so you can see all the subtrees as well. This is a great way to get the SHA-1 of any blob anywhere in the tree without having to walk it one node at a time.

$ git ls-tree -r -t master^{tree}

* git ls-tree master^
* git ls-tree –r –t master^ (run it recursively)

The -t makes it also show the SHA-1s of the subtrees themselves, rather than just all the blobs

you may want to extract the contents of individual blobs. The cat-file command is an easy way to do that, and can also serve to let you know what type of object a SHA-1 is, if you don’t know. It is sort of a catch-all command that you can use to inspect objects.

$ git cat-file -t ae850bd698b2b5dfbac

$ git cat-file -p ae850bd698b2b5dfbac

With those basic commands, you should be able to explore and inspect any object in any git repository relatively easily.

There are two major graphical interfaces that come with Git as tools to browse the repository.

gitk

A very popular choice for browsing Git repositories is the Tcl/Tk based browser called gitk. If you want to see a simple visualization of your repository, gitk is a great tool.

One of the most interesting visualizations that I regularly use is gitk—all, which will show all of your branches next to each other

instaweb

If you don’t want to fire up Tk, you can also browse your repository quickly via the git instaweb command. This will basically fire up a web server running the gitweb (http://git.or.cz/gitwiki/Gitweb) CGIscript using lighttpd, apache or webrick. It then tries to automatically fire up your default web browser and points it at the new

## Branching

There are two ways we can do this, one is to create the branch and then switch to it:

$ git branch newfunc; git checkout newfunc

The other way is to checkout a branch that doesn’t exist yet and tell git you want to create it by passing the ‘-b’ flag:

$ git checkout -b newfunc

### Undoing a Merge

So you try to git merge old\_branch it and you get conflict after conflict and it is just too much trouble to deal with and you just want to undo it all.

$ git reset --hard HEAD

The —hard makes sure both your index file and working directory are changed to match what it used it be. By default it will only reset your index, leaving the partially merged files in your working directory.

If you happen to have worked through it all and committed, then decided that it was a mistake because all of your tests break or something, you can still go back (and throw away that commit) by running:

$ git reset --hard ORIG\_HEAD

Command to see the differences between the branches

Git diff –stat master newfunc

git merge newfunc

git branch –d newfunc

## Undoing a merge

git reset

By default it will only reset your index, leaving the partially merged files in your directory.

git reset –hard

--hard makes sure both your index file and working directory are changed

## Rebasing

Much like Git provides a nicer way to work with your index before committing with ‘git add—interactive’, there is an interactive rebasing

git rebase master

if there is a conflict, yout have three things you can do here

* fix the file runs git add on it and run git rebase –continue
* run git rebase --abort will reset us to what our repo looks like before tried the rebase
* run git rebase –skip skips this patch , abandoning the change forever

When you clone a repository, it in essence copies all the git objects to a new directory, checks you out a single local branch named the same as the HEAD branch on the cloned repo (normally master) and stores all the other branches under a remote reference by default named ‘origin’

git checkout –track newfunction origin/newfunction

--track indicates that you may want to pull from or push to the origin of this branch later

Bare repository Is a repository without a working directory

# Stashing

Normally I will just use ‘git stash’ to save something, go work elsewhere, then come back and run ‘git stash apply’ to get back to where I was.

You can run ‘git stash’, which will basically take the changes from your last commit to the current state of your working directory and store it temporarily.

$ git status

$ git stash

$ git status

Now I can see that my working directory is clean, as if I had committed, but I did not

I can see my stashes by running git stash list

git stash show stash@{1}

git stash show stash@{0}

git stash show stash@{2}

I can also use any normal git tools that will take a tree on it, for instance, ‘git diff’:

git diff stash@{1}

And finally I can apply it

git stash apply stash@{1}

git stash apply without the actual stash reference it will just apply the last trash you saved on that branch

### Tagging

it is simply an arbitrary string that points to a specific commit. A tag in Git serves is basically a signed branch that never moves.

$ git tag -a v0.1 -m ‘this is my v0.1 tag’

that command will create a git object and will store that in the ‘.git/objects/’ directory and then will create a permanent reference to it in ‘.git/refs/tags/v0.1’ that contains the SHA-1 of that tag

Then you can use that as a reference to that commit at any time in commands like ‘diff’ or ‘archive

#### Lightweight Tags

You can also create a tag that doesn’t actually add a Tag object to the database, but just creates a reference to it in the ‘.git/refs/tags’ directory. If you run the following command:

$ git tag v0.1

Git will create the same file as before, ‘.git/refs/tags/v0.1’, but it will contain the SHA-1 of the current HEAD commit itself, not the SHA-1 of a Tag object pointing to that commit. Unlike object Tags, these can be moved around easily, which is generally undesirable.

### Multiple remotes

In Git, there is really no special repository. You can add as many remote repositories that are related to your codebase in some way as you want. You can add each of your co-workers repositories as read-only repositories.

Pushing to and pulling from multiple sources is easy and straightfor­ward

git remote add mycap https://

git remote add official https://

You can also remove remotes at any time, which simply removes the lines that contain the URL in your .git/config file and the references to their remote branches in .git/refs/[remote\_name] directory. It will not remove any of the git objects, so if you decide to add it again and fetch, very little will be transferred.

You can also view useful information about a remote branch by using the remote show command.

useful information about a remote branch

git remote show origin

## Commands Overview