# Git Internals

In this section, we will go over what Git was built for and how it works, hopefully laying the groundwork to properly understand what it is doing when we run the commands.

Git is a stupid content tracker. Git tracks content – files and directories. It is at its heart a collection of simple tools that implement a tree history storage and directory content management system. It is simply used as an SCM, not really designed as one.

When most SCMs store a new version of a project, they store the code delta or diff. When Git stores a new version of a project, it stores a new *tree* – a bunch of blobs of content and a collection of point­ers that can be expanded back out into a full directory of files and subdirectories. If you want a diff between two versions, it doesn’t add up all the deltas, it simply looks at the two trees and runs a new diff on them.

This is what fundamentally allows the system to be easily distributed – it doesn’t have issues figuring out how to apply a complex series of deltas, it simply transfers all the directories and content that one user has and another does not have but is requesting. It is efficient about it – it only stores identical files and directories once and it can com­press and transfer its content using delta-compressed packfiles – but in concept, is a very simple beast. Git is at it’s heart very stupid-simple.

* Non-Linear Development

Git is optimized for cheap and efficient branching and merging

* Distributed Development

Git is built to make distributed development simple. No repository is special or central in Git – each clone is basically equal and could generally replace any other one at any time. It works completely offline or with hundreds of remote repositories

* Efficiency

Most operations are local, which reduces unnecessary network overhead

* A Toolkit Design

Git is not really a single binary, but a collection of dozens of small specialized programs, which is sometimes annoying to people trying to learn Git, but is pretty cool when you want to do anything non-standard with it. Git is less a program and more a toolkit that can be combined and chained to do new and interesting things.

The tools can be more or less divided into two major camps, often referred to as the *porcelain* and the *plumbing*. The plumbing is not really meant to be used by people on the command line, but rather to do simple things flexibly and are combined by programs and scripts into porcelain programs.

## Git Object Types

Git objects are the actual data of Git, the main thing that the repository is made up of.

All of these types of objects are stored in the Git Object Database, which is kept in the Git Directory. Each object is compressed (with Zlib) and referenced by the SHA-1 value of its contents plus a small header (SHA stands for Secure Hash Algorithm)

In Git, the contents of files are stored as blobs. It is important to note that it is the contents that are stored, not the files. The names and modes of the files are not stored with the blob, just the contents.

Directories in Git basically correspond to **trees**. A tree is a simple list of trees and blobs that the tree contains, along with the names and modes of those trees and blobs.

### The Commit

The commit is very simple, much like the tree. It simply points to a tree and keeps an *author*, *committer*, *message* and any *parent* com­mits that directly preceded it.

The Tag

The final type of object you will find in a Git database is the **tag**. This is an object that provides a permanent shorthand name for a par­ticular commit.



The Git Data Model

the Git object data is a *directed acyclic graph.*

The cheap references I’ve represented as the grey boxes, the immutable objects are the colored round cornered boxes.



References

In addition to the Git objects, which are immutable – that is, they cannot ever be changed, there are references also stored in Git. Unlike the objects, references can constantly change. They are simple pointers to a particular commit, something like a tag, but eas­ily moveable.

A branch in Git is nothing more than a file in the .git/refs/heads/ directory that con­tains the SHA-1 of the most recent commit of that branch

In fact, in Git the act of creating a new branch is simply writing a file in the .git/refs/heads directory that has the SHA-1 of the last commit for that branch.

How does Git actually retrieve these objects in practice?

Well, it gets the initial SHA-1 of the starting commit object by looking in the .git/refs directory for the branch, tag or remote you specify. Then it tra­verses the objects by walking the trees one by one, checking out the blobs under the names listed.

In fact, in Git the act of creating a new branch is simply writing a file in the .git/refs/heads directory that has the SHA-1 of the last commit for that branch.

Switching to that branch simply means having Git make your work­ing directory look like the tree that SHA-1 points to and updating the HEAD file so each commit from that point on moves that branch pointer forward (

## The Treeish

Besides branch heads, there are a number of shorthand ways to refer to particular objects in the Git data store. These are often referred to as a *treeish*. Any Git command that takes an object – be it a commit, tree or blob – as an argument can take one of these shorthand versions as well.

* Full SHA-1

dae86e1950b1277e545cee180551750029cfe735

* PARTIAL SHA-1

dae86e

the full SHA-1 can be referenced fine with the first 6 or 7 characters. Git is smart enough to figure out a partial SHA-1 as long as it’s unique.

* Branch or tag name

Anything in *.git/refs/heads* or *.git/refs/tags* can be used to refer to the commit it points to.

* date spec

master@{yesterday}

master@{1 month ago}

* ordinal spec

master@{5}

This indicates the 5th prior value of the master branch. Like the *Date Spec*, this depends on special files in the *.git/log* directory that are written during commits, and is specific to *your* repository

* Carrot parent

dae86e^N

this refers to the Nth parent of that commit. Only really helpful for commits that merged two or more commits

* Tilde spec

dae86e~N

refers to the Nth generation grandparent of that commit

dae86e~5 ⬄ dae86e^^^^^

* tree pointer

e65s46^{tree}

This points to the tree of that commit. Any time you add a ^{tree} to any commit-ish, it resolves to its tree.



* Tree pointer

dae86e^{tree}

This points to th tree of that commit

blob spec

master:/path/to/file

This is very helpful for referring to a blob under a particular commit or tree.

#### Git repository

When you initialize a Git repository, either by cloning an existing one or creating a new one, the first thing Git does is create a Git directory. This is the directory that stores all the object data, tags, branches, hooks and more. Everything that Git permanently stores goes in this single directory. When you clone someone else’s reposi­tory, it basically just copies the contents of this directory to your computer.

When you run git init to initialize your repository, the Git directory is by default installed in the directory you are currently in as .git. The Git directory for our little project looks something like this:

For now, let’s go over some of the more important contents of this directory.

* .git/config

This is the main Git configuration file. It keeps your project specific Git options, such as your remotes, push configurations, tracking branches and more.

* .git/index

This is the default location of the index file for your Git project.

* .git/objects/

This is the main directory that holds the data of your Git objects – that is, all the contents of the files you have ever checked in, plus your commit, tree and tag objects.

The files are stored by their SHA-1 values. The first two characters make up the subdirectory and the last 38 is the filename

* .git/refs/

This directory normally has three subdirectories in it – *heads*, *remotes* and *tags*. Each of these directories will hold files that correspond to your local branches, remote branches and tags, respectively

* ..git/HEAD

This file holds a reference to the branch you are currently on. This tells Git what to use as the parent of your next commit

* .git/hooks

Contains shell scripts that are invoked after the git command

### Working directory

Your working directory is temporary – everything is stored permanently in your git repository. Your working directory is a just a copy of a tree so you can edit it and commit changes

The Index

The index was called the cache for a while, because that’s largely what it does. It is a staging area for changes that are made to files or trees that are not committed to your repository yet. It acts as sort of a middle ground between your working directory and your repository. When you run git commit, the resulting tree and commit object will be built based on the contents of the index.

Now that you *hopefully* understand what Git is designed to do at a fundamental level – how it tracks and stores content, how it stores branches and merges and tracks remote copies of the repository,

# Using Git

This next section presents some of the basic commands that you will need to know in order to use Git effectively.

## Setting Up Your Profile

For every commit you do, Git will try to associate a name and email address. One of the first things you’ll want to do in Git is to set these values. You can set them as global configuration values with the git config command:

$ git config --global user.name “Scott Chacon”

$ git config --global user.email [schacon@gmail.com](mailto:schacon@gmail.com)

If you want to set different values for a specific project, just leave out the —global and it will write the same snippet into your .git/config file in that repository, which will overwrite your global values

You can change those variables at any time either by editing that file, or running the git config commands again

## Getting a Git Repository

There are two major ways you will get a Git repository – you will either clone an existing project, or you will initialize a new one

To create a new Git repository somewhere, simply go to the directory you want to add version control to and type:

$ git init

$ git add .

$ git commit -m ‘my first commit’

This will by default create a new directory called simplegit and do an initial checkout of the master branch. If you want to put it in a different directory than the name of the project, you can specify that on the command line, too

$ git clone git://github.com/schacon/simplegit.git my\_directory

## Normal Workflow Examples

### Ignoring

First off, we will often want Git to automatically ignore certain files – often ones that are automatically generated during our development. To do this, we can add patterns into the .gitignore file to tell Git that we don’t want it to track them.

## Adding and Committing



If we want to commit all our changes, we can use this shorthand, which will automatically run a git add on every modified file to our index, then commit the whole thing:

$ git commit -a -m ‘committing all changes’



there is another way of adding files that makes for a more controlled and thematic set of commits. This is called *interactive* adding, and it is a very powerful tool to controlling what goes into each commit.

The interactive shell is pretty simple and very powerful – playing with it instead of running git add commands directly may help in under­standing what’s happening, since you can see the status of your files in the index versus the working directory more clearly. It helps visualize that what is in your index (the staged column) is what will be committed when you run git commit.

For removing files from your tree, you can simply run:

git rm <filename>

This will remove that file from the index (and thus from the next com­mit) as well as from your working directory. On your next commit, the tree that commit points to will simply not contain that file anymore.

## Log – the Commit History

The log com­mand can show you nearly anything you want to know about your commit history. Also, since the entire history is stored locally, it’s really fast compared with most other SCM systems,

$ git log

This will show you the SHA-1 of each commit, the committer and date of the commit, and the full message, starting from the last com­mit on your current branch and going backward in reverse chrono­logical order (so if there are multiple parents, it just squishes them together, interleaving the commits ordered by date)

### Formatting Log Output

The default format takes up a lot of space though, so there are ways to limit and format this output differently. —pretty is a useful option for formatting the output in different ways.

For example, we can list the commit SHA-1s and the first line of the message with —pretty=oneline:

With —pretty, you can choose between *oneline*, *short*, *medium*, *full*, *fuller*, *email*, *raw* and *format:(string)*, where (string) is a format you specify with variables (ex: —format:”%an added %h on %ar” will give you a bunch of lines like “Scott Chacon added f1cc9df 4 days ago”).

### Filtering Log Output

There are also a number of options for filtering the log output. You can specify the maximum number of commits you want to see with -n, you can limit the range of dates you want to see commits for with —since and —until, you can filter it on the author or committer, text in the commit message and more. Here is an example showing at most 30 commits between yesterday and a month ago by me :

### git log -n 30 -- since=”1 month ago” --until=yesterday --author=”schacon”

## 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

## Searching Git

Git has an easy way for searching through trees in your repository whitout having to check them out into your working directory. It is called ‘git-grep’ and works very much like the tradi­tional UNIX ‘grep’ command, with the difference that instead of listing the files you want to search as an argument, you list the trees you want to search

For example, if we wanted to search for the string ‘log\_syslog’ in ver­sions 1.0 and 1.5.3.8 of the Git source code in the C files only, we can find that very easily.

$ git grep -n ‘log\_syslog’ v1.5.3.8 v1.0.0 -- \*.c

$ git grep -c ‘log\_syslog’ v1.5.3.8 v1.0.0 -- \*.c

## Git diff

Git has a great diff utility built in that can give you statistics or a patch file given any combination of tree objects, working directory and index.

If you simply run ‘git diff’ with no arguments, it will show you the dif­ferences between your current working directory and your index, that is, the last time you ran ‘git add’ on your files.

You can also use ‘git diff’ to show you some spiffy stats for a diff, rather than a patch file

$ git diff --numstat a11bef06a3f65..cf25cc3bfb0

$ git diff --stat 0576fac35..

If you want to see what the specific difference is in one of those files, you can just add a path limiter to the diff command.

$ git diff a11bef06a3f65..cf25cc3bfb0 – Rakefile

You can use this command to detect changes between your index and any tree, or your working directory and any tree, your working directory and your index, etc.

The default output of the ‘git diff’ command is a valid patch file.

### Branching

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

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

# Statshing

git stash

take the changes from the last commit to the current state of your working directory and store it manually

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}

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

Creating a tag in git is much like making abranch. A tag is basically a signed branch that never moves

git tag –a v0.1 -m

### Multiple remotes

Decentralized part in git. You can add multiple repositories

git remote add mycap https://

git remote add official https://

useful information about a remote branch

git remote show origin

## Extra tools

* git gc: runs the garbage collector for your repository
* git fsck: does an integrity check of the git filesystem
* git prune: Removes objects that are no longer pointed to by any object in any reachable branch

git prune –n : to see what it will do