Git community Book (scott Charon)

Introduction

This book is meant to be a starting point for people new to Git to learn it as quickly and easily as possible.

This book will start out by introducing you to the way Git stores data, to give you the context for why it is different than other VCS tools. This is meant to take you about 20 minutes.

Next we will cover **Basic Git Usage** - the commands you will be using 90% of the time. These should give you a good basis to use Git comfortably for most of what you're going to use it for. This section should take you about 30 minutes to read through.

Next we will go over **Intermediate Git Usage** - things that are slightly more complex, but may replace some of the basic commands you learned in the first section. This will mostly be tricks and commands that will feel more comfortable after you know the basic commands.

After you have all of that mastered, we will cover **Advanced Git** - commands that most people probably don't use very often, but can be very helpful in certain situations. Learning these commands should round out your day-to-day git knowledge; you will be a master of the Git!

Now that you know Git, we will then cover **Working with Git**. Here we will go over how to use Git in scripts, with deployment tools, with editors and more. These sections are meant to help you integrate Git into your environment.

Lastly, we will have a series of articles on **low-level documentation** that may help the Git hackers who want to learn how the actual internals and protocols work in Git.

## The Git Object Model

### The SHA

All the information needed to represent the history of a project is stored in files referenced by a 40-digit "object name" that looks something like this:

6ff87c4664981e4397625791c8ea3bbb5f2279a3

The SHA1 hash is a cryptographic hash function. What that means to us is that it is virtually impossible to find two different objects with the same name. This has a number of advantages; among others:

* Git can quickly determine whether two objects are identical or not, just by comparing names.
* Since object names are computed the same way in every repository, the same content stored in two repositories will always be stored under the same name.
* Git can detect errors when it reads an object, by checking that the object's name is still the SHA1 hash of its contents

### The Objects

Every object consists of three things - a **type**, a **size** and **content**. The *size* is simply the size of the contents, the contents depend on what type of object it is, and there are four different types of objects: "blob", "tree", "commit", and "tag".

* A **"blob"** is used to store file data - it is generally a file.
* A **"tree"** is basically like a directory - it references a bunch of other trees and/or blobs (i.e. files and sub-directories)
* A **"commit"** points to a single tree, marking it as what the project looked like at a certain point in time. It contains meta-information about that point in time, such as a timestamp, the author of the changes since the last commit, a pointer to the previous commit(s), etc.
* A **"tag"** is a way to mark a specific commit as special in some way. It is normally used to tag certain commits as specific releases or something along those lines.

Almost all of Git is built around manipulating this simple structure of four different object types. It is sort of its own little filesystem that sits on top of your machine's filesystem.

It is important to note that this is very different from most SCM systems that you may be familiar with. Subversion, CVS, Perforce, Mercurial and the like all use Delta Storage systems - they store the differences between one commit and the next. Git does not do this - it stores a snapshot of what all the files in your project look like in this tree structure each time you commit. This is a very important concept to understand when using Git.

### Blob Object

A blob generally stores the contents of a file.

You can use git-show to examine the contents of any blob. Since the blob is entirely defined by its data, if two files in a directory tree (or in multiple different versions of the repository) have the same contents, they will share the same blob object. The object is totally independent of its location in the directory tree, and renaming a file does not change the object that file is associated with

### Tree Object

A tree is a simple object that has a bunch of pointers to blobs and other trees - it generally represents the contents of a directory or subdirectory.

The ever-versatile git-show command can also be used to examine tree objects, but :git-ls-tree will give you more details.

### Commit Object

The "commit" object links a physical state of a tree with a description of how we got there and why.

You can use the --pretty=raw option to git-show or git-log to examine your favorite commit.

A commit is defined by:

* a **tree**: The SHA1 name of a tree object (as defined below), representing the contents of a directory at a certain point in time.
* **parent(s)**: The SHA1 name of some number of commits which represent the immediately previous step(s) in the history of the project. The example above has one parent; merge commits may have more than one. A commit with no parents is called a "root" commit, and represents the initial revision of a project. Each project must have at least one root. A project can also have multiple roots, though that isn't common (or necessarily a good idea).
* an **author**: The name of the person responsible for this change, together with its date.
* a **committer**: The name of the person who actually created the commit, with the date it was done. This may be different from the author; for example, if the author wrote a patch and emailed it to another person who used the patch to create the commit.
* a **comment** describing this commit.

A commit is usually created by git-commit, which creates a commit whose parent is normally the current HEAD, and whose tree is taken from the content currently stored in the index.

### Tag Object

A tag object contains an object name (called simply 'object'), object type, tag name, the name of the person ("tagger") who created the tag, and a message, as can be seen using git-cat-file.

$ git cat-file tag v1.5.0

git-tag can also be used to create "lightweight tags", which are not tag objects at all, but just simple references whose names begin with "refs/tags/"

## Git Directory and Working Directory

### The Git Directory

The 'git directory' is the directory that stores all Git's history and meta information for your project - including all of the objects (commits, trees, blobs, tags), all of the pointers to where different branches are and more.

There is only one Git Directory per project (as opposed to one per subdirectory like with SVN or CVS), and that directory is (by default, though not necessarily) '.git' in the root of your project. If you look at the contents of that directory, you can see all of your important files:

$>tree -L 1

.

|-- HEAD # pointer to your current branch

|-- config # your configuration preferences

|-- description # description of your project

|-- hooks/ # pre/post action hooks

|-- index # index file (see next section)

|-- logs/ # a history of where your branches have been

|-- objects/ # your objects (commits, trees, blobs, tags)

`-- refs/ # pointers to your branches

### The Working Directory

The Git 'working directory' is the directory that holds the current checkout of the files you are working on. Files in this directory are often removed or replaced by Git as you switch branches - this is normal. All your history is stored in the Git Directory; the working directory is simply a temporary checkout place where you can modify the files until your next commit.

## The Git Index

The Git index is used as a staging area between your working directory and your repository. You can use the index to build up a set of changes that you want to commit together. When you create a commit, what is committed is what is currently in the index, not what is in your working directory.

The easiest way to see what is in the index is with the git-status command. When you run git status, you can see which files are staged (currently in your index), which are modified but not yet staged, and which are completely untracked.

## How Git Stores Objects

All objects are stored as compressed contents by their sha values.

If the sha of your object is ab04d884140f7b0cf8bbf86d6883869f16a46f65, then the file will be stored in the following path:

.git/objects/ab/04d884140f7b0cf8bbf86d6883869f16a46f65

It pulls the first two characters off and uses that as the subdirectory, so that there are never too many objects in one directory. The actual file name is the remaining 38 characters.

## Browsing Git Objects

We can ask git about particular objects with the cat-file command. Note that you can shorten the shas to only a few characters to save yourself typing all 40 hex digits:

$ git cat-file -t 54196cc2

$ git cat-file -s 54196cc2

$ git cat-file -p 54196cc2

You can examine the contents of any tree using ls-tree

$ git ls-tree 92b8b694

All of these objects are stored under their SHA1 names inside the git directory and the contents of these files is just the compressed data plus a header identifying their length and their type. The type is either a blob, a tree, a commit, or a tag.

$ find .git/objects/

The simplest commit to find is the HEAD commit, which we can find from .git/HEAD:

$ cat .git/HEAD

ref: refs/heads/master

$ cat .git/refs/heads/master

c4d59f390b9cfd4318117afde11d601c1085f241

$ git cat-file -t c4d59f39

commit

The "tree" object here refers to the new state of the tree and the "parent" object refers to the previous commit:

## Git References

Branches, remote-tracking branches, and tags are all references to commits. All references are named with a slash-separated path name starting with "refs"; the names we've been using so far are actually shorthand:

- The branch "test" is short for "refs/heads/test".

- The tag "v2.6.18" is short for "refs/tags/v2.6.18".

- "origin/master" is short for "refs/remotes/origin/master

We can list all the heads in this repository with linkgit:git-show-ref

$ git show-ref --heads

$ git show-ref --tags

## The Git Index

The index is a binary file (generally kept in .git/index) containing a sorted list of path names, each with permissions and the SHA1 of a blob object; linkgit:git-ls-files[1] can show you the contents of the index:

$ git ls-files --stage

Computes the object ID value for an object with specified type with the contents of the named file

$ git hash-objec

## Customizing Git

### Git Config

The first thing you're going to want to do is set up your name and email address for Git to use to sign your commits.

$ git config --global user.name "Scott Chacon"

$ git config --global user.email "schacon@gmail.com"

That will set up a file in your home directory which may be used by any of your projects. By default that file is ~/.gitconfig and the contents will look like this:

If you want to override those values for a specific project (to use a work email address, for example), you can run the git config command without the --global option while in that project. This will add a [user] section like the one shown above to the .git/config file in your project's root directory.

At a Windows command prompt enter the commands:  
git config --global diff.tool bc  
git config --global difftool.bc.path "c:/Program Files/Beyond Compare 4/bcomp.exe"

git config --global merge.tool bc  
git config --global mergetool.bc.path "c:/Program Files/Beyond Compare 4/bcomp.exe"

### Changing your Editor

$ git config --global core.editor emacs

### Adding Aliases

$ git config --global alias.last 'cat-file commit HEAD'

$ git last

### Commit Template

$ git config commit.template '/etc/git-commit-template'

### Log Format

$ git config format.pretty oneline

## Getting a Git Repository

### Cloning a Repository

git clone git://git.kernel.org/pub/scm/git/git.git

or over http:

git clone http://www.kernel.org/pub/scm/git/git.git

### Initializing a New Repository

$ cd project

$ git init

## Normal Workflow

add files to the index:

$ git add file1 file2 file3

You are now ready to commit. You can see what is about to be committed using linkgit:git-diff[1] with the --cached option:

$ git diff --cached

You can also get a brief summary of the situation with git-status

$ git commit

This will again prompt you for a message describing the change, and then record a new version of the project.

Alternatively, instead of running git add beforehand, you can use

$ git commit -a

which will automatically notice any modified (but not new) files, add them to the index, and commit, all in one step

Git's "add" command does something simpler and more powerful: git add is used both for new and newly modified files, and in both cases it takes a snapshot of the given files and stages that content in the index, ready for inclusion in the next commit.

## Reviewing History - Git Log

The linkgit:git-log[1] command can show lists of commits. On its own, it shows all commits reachable from the parent commit;

$ git log v2.5.. # commits since (not reachable from) v2.5

$ git log test..master # commits reachable from master but not test

$ git log master..test # commits reachable from test but not master

$ git log master...test # commits reachable from either test or

# master, but not both

$ git log --since="2 weeks ago" # commits from the last 2 weeks

$ git log Makefile # commits that modify Makefile

$ git log fs/ # commits that modify any file under fs/

$ git log -S'foo()' # commits that add or remove any file data

# matching the string 'foo()'

$ git log --no-merges # dont show merge commits

And of course you can combine all of these; the following finds commits since v2.5 which touch the Makefile or any file under fs:

$ git log v2.5.. Makefile fs/

You can also ask git log to show patches:

$ git log -p

If you pass the --stat option to 'git log', it will show you which files have changed in that commit and how many lines were added and removed from each.

You can also format the log output almost however you want. The '--pretty' option can take a number of preset formats, such as 'oneline':

$ git log --pretty=oneline

$ git log --pretty=short

$ git log --pretty=full

$ git log --pretty=fuller

If those formats aren't exactly what you need, you can also create your own format with the '--pretty=format' option (see the git-log[1] docs for all the formatting options).

$ git log --pretty=format:'%h was %an, %ar, message: %s'

Another interesting thing you can do is visualize the commit graph with the '--graph' option,

you can reverse the order of the log with the '--reverse' option

## Comparing Commits - Git Diff

You can generate diffs between any two versions of your project using

$ git diff master..test

That will produce the diff between the tips of the two branches. If you'd prefer to find the diff from their common ancestor to test, you can use three dots instead of two:

$ git diff master...test

You will commonly use linkgit:git-diff[1] for figuring out differences between your last commit, your index, and your current working directory. A common use is to simply run

$ git diff

which will show you changes in the working directory that are not yet staged for the next commit. If you want to see what isstaged for the next commit, you can run

$ git diff --cached

$ git diff HEAD

which shows changes in the working directory since your last commit; what you would be committing if you run "git commit -a".

If you want to see how your current working directory differs from the state of the project in another branch, you can run something like

$ git diff test

This will show you what is different between your current working directory and the snapshot on the 'test' branch ou can also limit the comparison to a specific file or subdirectory by adding a *path limiter*:

$ git diff HEAD -- ./lib

That command will show the changes between your current working directory and the last commit (or, more accurately, the tip of the current branch), limiting the comparison to files in the 'lib' subdirectory.

If you don't want to see the whole patch, you can add the '--stat' option

## Git Tag

### Lightweight Tags

$ git tag stable-1 1b2e1d63ff

This creates a "lightweight" tag, basically a branch that never moves

### Tag Objects

If one of **-a** the command creates a tag object, and requires the tag message.

When this happens, a new object is added to the Git object database and the tag ref points to that tag object, rather than the commit itself.

## Ignoring files

You can tell git to ignore certain files by creating a file called .gitignore in the top level of your working directory, with contents such as:

## Interactive Rebasing

If you have a number of commits that you would like to somehow modify during the rebase, you can invoke interactive mode by passing a '-i' or '--interactive' to the 'git rebase' command.

$ git rebase -i origin/master

If 'pick' is specified, it will simply try to apply the patch and save the commit with the same message as before.

If 'squash' is specified, it will combine that commit with the previous one to create a new commit.

## Interactive Adding

$>git add -i

If you type '5' or 'p' in the menu, git will show you your diff patch by patch (or hunk by hunk) and ask if you want to stage each one. That way you can actually stage for a commit a part of a file edit. If you've edited a file and want to only commit part of it and not an unfinished part, or commit documentation or whitespace changes seperate from substantive changes, you can use 'git add -i' to do so relatively easily.

## Stashing

$ git stash

$>git stash list

## Git Treeishes

^ will give you the Nth parent of a particular commit. This format is only useful on merge commits - commit objects that have more than one direct parent.

master^2

The tilde spec will give you the Nth grandparent of a commit object. For example,

master~2

Finding files with words or phrases in Git is really easy with the linkgit:git-grep command.

$ git grep adel

If I wanted to see the line number of each match as well, I can add the '-n' option:

$ git grep –n adel

If we're only interested in the filename, we can pass the '--name-only' option:

$>git grep --name-only adel

We could also see how many line matches we have in each file with the '-c' option:

$>git grep -c adel

Now, if I wanted to see where that was used in a specific version of git, I could add the tag reference to the end, like this:

$ git grep xmmap v1.5.0

We can also combine search terms in grep.

$ git grep -e 'adel' --and -e saki

We can also search for lines that have one term and either of two other terms, for example, if we wanted to see where we defined constants that had either PATH or MAX in the name:

$ git grep -e '#define' --and \( -e PATH -e MAX \)

## Advanced Branching And Merging

### Getting conflict-resolution help during a merge

All of the changes that git was able to merge automatically are already added to the index file, so git-diff shows only the conflicts.

Recall that the commit which will be committed after we resolve this conflict will have two parents instead of the usual one: one parent will be HEAD, the tip of the current branch; the other will be the tip of the other branch, which is stored temporarily in MERGE\_HEAD.

During the merge, the index holds three versions of each file. Each of these three "file stages" represents a different version of the file:

$ git show :1:file.txt # the file in a common ancestor of both branches

$ git show :2:file.txt # the version from HEAD.

$ git show :3:file.txt # the version from MERGE\_HEAD.

When you ask linkgit:git-diff[1] to show the conflicts, it runs a three-way diff between the conflicted merge results in the work tree with stages 2 and 3 to show only hunks whose contents come from both sides, mixed (in other words, when a hunk's merge results come only from stage 2, that part is not conflicting and is not shown. Same for stage 3).

Some special diff options allow diffing the working directory against any of these stages:

$ git diff -1 file.txt # diff against stage 1

$ git diff --base file.txt # same as the above

$ git diff -2 file.txt # diff against stage 2

$ git diff --ours file.txt # same as the above

$ git diff -3 file.txt # diff against stage 3

$ git diff --theirs file.txt # same as the above.

## Finding Issues - Git Blame

The git-blame command is really helpful for figuring out who changed which sections of a file. If you simple run 'git blame [filename]' you'll get an output of the entire file with the last commit sha, date and author for every line in the file.

$> git blame sha1\_file.c

This is often helpful if a file had a line reverted or a mistake that broke the build to help you see who changed that line last

$> git blame -L 160,+10 sha1\_file.c

## SCM Migration

So you've made the decision to move away from your existing system and convert your whole project to Git. How can you do that easily?

### Importing Subversion

Git comes with a script called git-svn that has a clone command that will import a subversion repository into a new git repository. There is also a free tool on the GitHub service that will do this for you.

$ git-svn clone http://my-project.googlecode.com/svn/trunk new-project

This will give you a new Git repository with all the history of the original Subversion repo. This takes a pretty good amount of time, generally, since it starts with version 1 and checks out and commits locally every single revision one by one.

## Submodules

To see how submodule support works, create (for example) four example repositories that can be used later as a submodule:

$ mkdir ~/git

$ cd ~/git

$ for i in a b c d

do

mkdir $i

cd $i

git init

echo "module $i" > $i.txt

git add $i.txt

git commit -m "Initial commit, submodule $i"

cd ..

done

Now create the superproject and add all the submodules:

$ mkdir super

$ cd super

$ git init

$ for i in a b c d

do

git submodule add ~/git/$i $i

done

See what files git-submodule created:

$ ls -a

. .. .git .gitmodules a b c d

The git-submodule add command does a couple of things:

* It clones the submodule under the current directory and by default checks out the master branch.
* It adds the submodule's clone path to the linkgit:gitmodules[5] file and adds this file to the index, ready to be committed.
* It adds the submodule's current commit ID to the index, ready to be committed.

Commit the superproject:

$ git commit -m "Add submodules a, b, c and d."

Now clone the superproject:

$ cd ..

$ git clone super cloned

$ cd cloned

The submodule directories are there, but they're empty:

Pulling down the submodules is a two-step process. First run git submodule init to add the submodule repository URLs to .git/config:

$ git submodule init

Now use git-submodule update to clone the repositories and check out the commits specified in the superproject:

$ git submodule update