Git pour les breles

# 1 Introduction

# 2 Installation

If you ever need help while using Git, there are two equivalent ways to get the comprehensive manual page (manpage) help for any of the Git commands:

$ git help <verb>

In addition, if you don’t need the full-blown manpage help, but just need a quick refresher on the available options for a Git command, you can ask for the more concise “help” output with the -h or --help options, as in:

$ git add –h

The git help command is used to show you all the documentation shipped with Git about any command

$ git help <command>

# 3 Git Configuration

Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places

1. /etc/gitconfig file: Contains values applied to every user on the system and all their repositories. If you pass the option --system to git config, it reads and writes from this file specifically. (Because this is a system configuration file, you would need administrative or superuser privilege to make changes to it.)

2. ~/.gitconfig or ~/.config/git/config file: Values specific personally to you, the user. You can make Git read and write to this file specifically by passing the --global option, and this affects *all* of the repositories you work with on your system.

3. config file in the Git directory (that is, .git/config) of whatever repository you’re currently using: Specific to that single repository. You can force Git to read from and write to this file with the --local option, but that is in fact the default. (Unsurprisingly, you need to be located somewhere in a Git repository for this option to work properly.)

*On Windows systems, Git looks for the .gitconfig file in the $HOME directory (C:Users\$USER for most people). It also still looks for /etc/gitconfig, although it’s relative to the MSys root, which is wherever you decide to install Git on your Windows system when you run the installer. If you are using version 2.x or later of Git for Windows, there is also a system-level config file at C:\Documents and Settings\All Users\Application Data\Git\config on Windows XP, and in C:\ProgramData\Git\config on Windows Vista and newer. This config file can only be changed by git config -f <file> as an admin.*

Each level overrides values in the previous level

The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it’s immutably baked into the commits you start creating:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

**Your Editor**

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message. If not configured, Git uses your system’s default editor.If you want to use a different text editor, such as Emacs, you can do the following:

$ git config --global core.editor emacs

If you want to check your configuration settings, you can use the git config --list command to list all the settings Git can find at that point:

You can also check what Git thinks a specific key’s value is by typing git config <key>:

$ git config user.name

Since Git might read the same configuration variable value from more than one file, it’s possible that you have an unexpected value for one of these values and you don’t know why. In cases like that, you can query Git as to the *origin* for that value, and it will tell you which configuration file had the final say in setting that

value:

$ git config --show-origin rerere.autoUpdate

file:/home/johndoe/.gitconfig false

As you read briefly in Getting Started, you can specify Git configuration settings with the git config command. One of the first things you did was set up your name and email address:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

First, a quick review: Git uses a series of configuration files to determine non-default behavior that you may want. The first place Git looks for these values is in the system-wide /etc/gitconfig file, which contains settings that are applied to every user on the system and all of their repositories. If you pass the option --system to git config, it reads and writes from this file specifically.

The next place Git looks is the ~/.gitconfig (or ~/.config/git/config) file, which is specific to each user. You can make Git read and write to this file by passing the --global option.

Finally, Git looks for configuration values in the configuration file in the Git directory (.git/config) of whatever repository you’re currently using. These values are specific to that single repository, and represent passing the --local option to git config. (If you don’t specify which level you want to work with, this is the default.)

Each of these “levels” (system, global, local) overwrites values in the previous level, so values in .git/config trump those in /etc/gitconfig, for instance

core.editor:

By default, Git uses whatever you’ve set as your default text editor via one of the shell environment variables VISUAL or EDITOR, or else falls back to the vi editor to create and edit your commit and tag messages. To change that default to something else, you can use the core.editor setting:

$ git config --global core.editor emacs

commit.template:

If you set this to the path of a file on your system, Git will use that file as the default initial message when you commit.

$ git config --global commit.template ~/.gitmessage.txt

core.autocrlf

$ git config --global core.autocrlf true

$ git config --global core.autocrlf input

If you’re a Windows programmer doing a Windows-only project, then you can turn off this functionality, recording the carriage returns in the repository by setting the config value to false:

$ git config --global core.autocrlf false

**git config**

Git has a default way of doing hundreds of things. For a lot of these things, you can tell Git to default to doing them a different way, or set your preferences.

Git’s configuration files are all simple text files in the style of *.ini* files. They record various choices and settings used by many Git commands. Some settings represent purely personal preferences (should a color.pager be used?); others are vital to a repository functioning correctly (core.repositoryformatversion); and still others tweak command behavior a bit (gc.auto).

Like many tools, Git supports a hierarchy of configuration files. In decreasing precedence they are:

*git/config*

* Repository-specific configuration settings manipulated with the --file option or by default. These settings have the highest precedence.

*~/.gitconfig*

* User-specific configuration settings manipulated with the --global option.

*/etc/gitconfig*

* System-wide configuration settings manipulated with the --system option if you have proper Unix file write permissions on it. These settings have the lowest precedence. Depending on your actual installation, the system settings file might be somewhere else (perhaps in */usr/local/etc/gitconfig*), or may be entirely absent.

Or, to set a repository-specific name and email address that would override a

--global setting, simply omit the --global flag:

$ **git config user.name "Jon Loeliger"**

$ **git config user.email "jdl@special-project.example.org"**

Use git config -l to list the settings of all the variables collectively found in the complete set of configuration files:

$ **git config –l**

Because the configuration files are simple text files, you can view their contents with cat and edit them with your favorite text editor, too.

$ **cat .git/config**

Use the --unset option to remove a setting:

$ **git config --unset --global user.email**

## Configuring an Alias

For starters, here is a tip for setting up command aliases. If there is a common but

complex Git command that you type frequently, consider setting up a simple Git alias

for it.

$ **git config --global alias.show-graph 'log --graph --abbrev-commit --pretty=oneline'**

## 3. 1 Ignoring Files

Often, you’ll have a class of files that you don’t want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named .gitignore. Here is an example .gitignore file:

$ cat .gitignore

\*.[oa]

\*~

The first line tells Git to ignore any files ending in “.o” or “.a” — object and archive files that may be the product of building your code. The second line tells Git to ignore all files whose names end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files. You may also include a log, tmp, or pid directory; automatically generated documentation; and so on. Setting up a .gitignore file for your new repository before you get going is generally a good idea so you don’t accidentally commit files that you really don’t want in your Git repository. The rules for the patterns you can put in the .gitignore file are as follows:

* Blank lines or lines starting with # are ignored.
* Standard glob patterns work, and will be applied recursively throughout the entire working tree.
* You can start patterns with a forward slash (/) to avoid recursivity.
* You can end patterns with a forward slash (/) to specify a directory.
* You can negate a pattern by starting it with an exclamation point (!).

Glob patterns are like simplified regular expressions that shells use. An asterisk (\*) matches zero or more characters; [abc] matches any character inside the brackets (in this case a, b, or c); a question mark (?) matches a single character; and brackets enclosing characters separated by a hyphen ([0-9]) matches any character between them (in this case 0 through 9). You can also use two asterisks to match nested directories; a/\*\*/z would match a/z, a/b/z, a/b/c/z, and so on. Here is another example .gitignore file:

# ignore all .a files

\*.a

# but do track lib.a, even though you're ignoring .a files above

!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO

/TODO

# ignore all files in any directory named build

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

doc/\*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories

doc/\*\*/\*.pdf

# Inspecting code

## Code History

Perhaps you’re looking not for *where* a term exists, but *when* it existed or was introduced. The git log command has a number of powerful tools for finding specific commits by the content of their messages or even the content of the diff they introduce.

If, for example, we want to find out when the ZLIB\_BUF\_MAX constant was originally introduced, we can use the -S option (colloquially referred to as the Git “pickaxe” option) to tell Git to show us only those commits that changed the number of occurrences of that string.

$ git log -S ZLIB\_BUF\_MAX –oneline

If you need to be more specific, you can provide a regular expression to search for with the –G option.

**Line Log Search**

Another fairly advanced log search that is insanely useful is the line history search. Simply run git log with the -L option, and it will show you the history of a function or line of code in your codebase.

For example, if we wanted to see every change made to the function git\_deflate\_bound in the zlib.c file, we could run git log -L :git\_deflate\_bound:zlib.c. This will try to figure out what the bounds of that function are and then look through the history and show us every change that was made to the function as a series of patches back to when the function was first created.

$ git log -L :git\_deflate\_bound:zlib.c

Git can’t figure out how to match a function or method in your programming language, you can also provide it with a regular expression (or *regex*). For example, this would

have done the same thing as the example above:

$ git log -L '/unsigned long git\_deflate\_bound/',/^}/:zlib.c.

You could also give it a range of lines or a single line number and you’ll get the same sort of output

gitk is a graphical history viewer. Think of it like a powerful GUI shell over git log and git grep

Gitk is easiest to invoke from the command-line. Just cd into a Git repository, and type:

$ gitk [git log options]

Gitk accepts many command-line options, most of which are passed through to the underlying git log action. Probably one of the most useful is the --all flag, which tells gitk to show commits reachable from *any* ref, not just HEAD.

**File Annotation**

you can annotate the file with git blame to determine which commit was responsible for the introduction of that line.

The following example uses git blame to determine which commit and committer was responsible for lines in the top-level Linux kernel Makefile and, further, uses the -L option to restrict the output of the annotation to lines 69 through 82 of that file:

$ git blame -L 69,82 Makefile

Notice that the first field is the partial SHA-1 of the commit that last modified that line. The next two fields are values extracted from that commit — the author name and the authored date of that commit — so you can easily see who modified that line and when. After that come the line number and the content of the file. Also note the ^1da177e4c3f4 commit lines, where the ^ prefix designates lines that were introduced in the repository’s initial commit and have remained unchanged ever since. This is a tad confusing, because now you’ve seen at least three different ways that Git uses the ^ to modify a commit SHA-1, but that is what it means here.

Another cool thing about Git is that it doesn’t track file renames explicitly. It records the snapshots and then tries to figure out what was renamed implicitly, after the fact. One of the interesting features of this is that you can ask it to figure out all sorts of code movement as well. If you pass –C to git blame, Git analyzes the file you’re annotating and tries to figure out where snippets of code within it originally came from if they were copied from elsewhere. For example, say you are refactoring a file named GITServerHandler.m into multiple files, one of which is GITPackUpload.m. By blaming GITPackUpload.m with the -C option, you can see where sections of the code originally came from:

$ git blame -C -L 141,153 GITPackUpload.m

## Searching

Git provides a couple of useful tools for looking through the code and commits stored in its database quickly and easily. We’ll go through a few of them

**Git Grep**

Git ships with a command called grep that allows you to easily search through any committed tree,the working directory, or even the index for a string or regular expression.

By default, git grep will look through the files in your working directory. As a first variation, you can use either of the -n or --line-number options to print out the line numbers where Git has found matches:

$ git grep -n gmtime\_r

In addition to the basic search shown above, git grep supports a plethora of other interesting options.

For instance, instead of printing all of the matches, you can ask git grep to summarize the output by showing you only which files contained the search string and how many matches there were in each file with the -c or --count option:

$ git grep --count gmtime\_r

If you’re interested in the *context* of a search string, you can display the enclosing method or function for each matching string with either of the -p or --show-function options

$ git grep -p gmtime\_r \*.c

You can also search for complex combinations of strings with the --and flag, which ensures that multiple matches must occur in the same line of text. For instance, let’s look for any lines that define a constant whose name contains *either* of the substrings “LINK” or “BUF\_MAX”, specifically in an older version of the Git codebase represented by the tag v1.8.0 (we’ll throw in the --break and --heading options which help split up the output into a more readable format):

$ git grep --break --heading -n -e '#define' --and ( -e LINK -e BUF\_MAX ) v1.8.0

The git grep command has a few advantages over normal searching commands like grep and ack. The first is that it’s really fast, the second is that you can search through any tree in Git, not just the working directory. As we saw in the above example, we looked for terms in an older version of the Git source code, not the version that was currently checked out.

# Altering history

Many times, when working with Git, you may want to revise your local commit history. This can involve changing the order of the commits, changing messages or modifying files in a commit, squashing together or splitting apart commits, or removing commits entirely — all before you share your

work with others.

One of the cardinal rules of Git is that, since so much work is local within your clone, you have a great deal of freedom to rewrite your history *locally*. However, once you push your work, it is a different story entirely, and you should consider pushed work as final unless you have good reason to change it. In short, you should avoid pushing your work until you’re happy with it and ready to share it with the rest of the world.

**Changing the Last Commit**

$ git commit –amend

The command above loads the previous commit message into an editor session, where you can make changes to the message, save those changes and exit. When you save and close the editor, the editor writes a new commit containing that updated commit message and makes it your new last commit.

If, on the other hand, you want to change the actual *content* of your last commit, the process works basically the same way — first make the changes you think you forgot, stage those changes, and the subsequent git commit --amend *replaces* that last commit with your new, improved commit.

You need to be careful with this technique because amending changes the SHA-1 of the commit. It’s like a very small rebase — don’t amend your last commit if you’ve already pushed it.

you can simply make the changes, stage them, and avoid the unnecessary editor

session entirely with

$ git commit --amend --no-edit

**Changing Multiple Commit Messages**

For example, if you want to change the last three commit messages, or any of the commit messages in that group, you supply as an argument to git rebase -i the parent of the last commit you want to edit, which is HEAD~2^ or HEAD~3. It may be easier to remember the ~3 because you’re trying to edit the last three commits, but keep in mind that you’re actually designating four commits ago, the parent of the last commit you want to edit:

$ git rebase -i HEAD~3

**Reordering Commits**

You can also use interactive rebases to reorder or remove commits entirely. If you want to remove the “added cat-file” commit and change the order in which the other two commits are introduced, you can change the rebase script from this

**Squashing Commits**

It’s also possible to take a series of commits and squash them down into a single commit with the interactive rebasing tool.

**Splitting a Commit**

Splitting a commit undoes a commit and then partially stages and commits as many times as commits you want to end up with.

There, you can do a mixed reset of that commit with git reset HEAD^, which effectively undoes that commit and leaves the modified files unstaged. Now you can stage and commit files until you have several commits, and run git rebase –continue when you’re done:

$ git reset HEAD^

$ git add README

$ git commit -m 'updated README formatting'

$ git add lib/simplegit.rb

$ git commit -m 'added blame'

$ git rebase --continue

**The Nuclear Option: filter-branch**

There is another history-rewriting option that you can use if you need to rewrite a larger number of commits in some scriptable way – for instance, changing your email address globally or removing a file from every commit. The command is filter-branch, and it can rewrite huge swaths of your history, so you probably shouldn’t use it unless your project isn’t yet public and other people haven’t based work off the commits you’re about to rewrite. However, it can be very useful.

**Removing a File from Every Commit**

To remove a file named passwords.txt from your entire history, you can use the --tree-filter option to filter-branch:

$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD

Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)

Ref 'refs/heads/master' was rewritten

The --tree-filter option runs the specified command after each checkout of the project and then recommits the results. In this case, you remove a file called passwords.txt from every snapshot, whether it exists or not. If you want to remove all accidentally committed editor backup files, you can run something like

$ git filter-branch --tree-filter 'rm -f \*~'HEAD.

To run filter-branch on all your branches, you can pass --all to the command

# Mainteance and archivage

**Cleaning your Working Directory**

You’ll want to be pretty careful with this git clean command, since it’s designed to remove files from your working directory that are not tracked. If you change your mind, there is often no retrieving the content of those files. A safer option is to run git stash --all to remove everything but save it in a stash.

Assuming you do want to remove cruft files or clean your working directory, you can do so with git clean. To remove all the untracked files in your working directory, you can run git clean -f -d, which removes any files and also any subdirectories that become empty as a result. The -f means *force* or "really do this".

If you ever want to see what it would do, you can run the command with the -n option, which means “do a dry run and tell me what you *would* have removed”

$ git clean -d -n

Si vous souhaitez visualiser ce qui serait fait, vous pouvez lancer la commande avec l’option -n qui signifie ≪ fais-le a blanc et montre-moi ce qui *serait* supprime ≫.

$ git clean -d –n

By default, the git clean command will only remove untracked files that are not ignored. Any file that matches a pattern in your .gitignore or other ignore files will not be removed. If you want to remove those files too, such as to remove all .o files generated from a build so you can do a fully clean build, you can add a -x to the clean command.

If you don’t know what the git clean command is going to do, always run it with a -n first to double check before changing the -n to a -f and doing it for real. The other way you can be careful about the process is to run it with the -i or “interactive” flag.

This will run the clean command in an interactive mode.

$ git clean -x –i

# Git Internals

When you run git init in a new or existing directory, Git creates the .git directory, which is where almost everything that Git stores and manipulates is located.

a newlyinitialized .git directory typically looks like:

$ ll -la .git

total 11

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 ./

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 ../

-rw-r--r-- 1 asaki 1049089 130 Sep 27 18:37 config

-rw-r--r-- 1 asaki 1049089 73 Sep 27 18:37 description

-rw-r--r-- 1 asaki 1049089 23 Sep 27 18:37 HEAD

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 hooks/

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 info/

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 objects/

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 refs/

$ ls -F1

config

description

HEAD

hooks/

info/

objects/

refs/

Depending on your version of Git, you may see some additional content there, but this is a fresh git init repository — it’s what you see by default. The description file is used only by the GitWeb program, so don’t worry about it. The config file contains your project-specific configuration options, and the info directory keeps a global exclude file for ignored patterns that you don’t want to track in a .gitignore file. The hooks directory contains your client- or server-side hook scripts, which are discussed in detail in Git Hooks.

This leaves four important entries: the HEAD and (yet to be created) index files, and the objects and refs directories. These are the core parts of Git. The objects directory stores all the content for your database, the refs directory stores pointers into commit objects in that data (branches, tags,

remotes and more), the HEAD file points to the branch you currently have checked out, and the index file is where Git stores your staging area information. You’ll now look at each of these sections in detail to see how Git operates.

**Git Objects**

Git is a content-addressable filesystem. Great. What does that mean? It means that at the core of Git is a simple key-value data store. What this means is that you can insert any kind of content into a Git repository, for which Git will hand you back a unique key you can use later to retrieve that content.

$ find .git/objects

.git/objects

.git/objects/info

.git/objects/pack

$ find .git/objects -type f

Git has initialized the objects directory and created pack and info subdirectories in it, but there are no regular files. Now, let’s use git hash-object to create a new data object and manually store it in your new Git database

$ echo 'test content' | git hash-object -w –stdin

In its simplest form, git hash-object would take the content you handed to it and merely return the unique key that *would* be used to store it in your Git database. The -w option then tells the command to not simply return the key, but to write that object to the database. Finally, the --stdin option tells git hash-object to get the content to be processed from stdin; otherwise, the command would expect a filename argument at the end of the command containing the content to be used.

The output from the above command is a 40-character checksum hash. This is the SHA-1 hash — a checksum of the content you’re storing plus a header, which you’ll learn about in a bit. Now you can see how Git has stored your data:

$ find .git/objects -type f

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4

This is how Git stores the content initially — as a single file per piece of content, named with the SHA-1 checksum of the content and its header. The subdirectory is named with the first 2 characters of the SHA-1, and the filename is the remaining 38 characters.

Once you have content in your object database, you can examine that content with the git cat-file command. This command is sort of a Swiss army knife for inspecting Git objects. Passing -p to catfile instructs the command to first figure out the type of content, then display it appropriately:

$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4

test content

Now, you can add content to Git and pull it back out again. You can also do this with content in files. For example, you can do some simple version control on a file. First, create a new file and save its contents in your database:

$ echo 'version 1' > test.txt

$ git hash-object -w test.txt

83baae61804e65cc73a7201a7252750c76066a30

Then, write some new content to the file, and save it again:

$ echo 'version 2' > test.txt

$ git hash-object -w test.txt

1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

Your object database now contains both versions of this new file (as well as the first content you stored there):

$ find .git/objects -type f

.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a

.git/objects/83/baae61804e65cc73a7201a7252750c76066a30

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4

You can have Git tell you the object type of any object in Git, given its SHA-1 key, with git cat-file -t:

$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

Blob

**Tree Objects**

The next type of Git object we’ll examine is the *tree*, which solves the problem of storing the filename and also allows you to store a group of files together. Git stores content in a manner similar to a UNIX filesystem, but a bit simplified. All the content is stored as tree and blob objects, with trees corresponding to UNIX directory entries and blobs corresponding more or less to inodes or file contents. A single tree object contains one or more entries, each of which is the SHA-1 hash of a blob or subtree with its associated mode, type, and filename. For example, the most recent tree in a project may look something like this:

$ git cat-file -p master^{tree}

100644 blob a906cb2a4a904a152e80877d4088654daad0c859 README

100644 blob 8f94139338f9404f26296befa88755fc2598c289 Rakefile

040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0 lib

The master^{tree} syntax specifies the tree object that is pointed to by the last commit on your master branch. Notice that the lib subdirectory isn’t a blob but a pointer to another tree:

**Commit Objects**

This is essentially what Git does when you run the git add and git commit

commands — it stores blobs for the files that have changed, updates the index, writes out trees, and writes commit objects that reference the top-level trees and the commits that came immediately before them. These three main Git objects — the blob, the tree, and the commit — are initially stored as separate files in your .git/objects directory. Here are all the objects in the example directory now, commented with what they store:

**Object Storage**

**Git References**

Instead, itwould be easier if you had a file in which you could store that SHA-1 value under a simple name so you could use that simple name rather than the raw SHA-1 value.

In Git, these simple names are called “references” or “refs”; you can find the files that contain those SHA-1 values in the .git/refs directory. In the current project, this directory contains no files, but it does contain a simple structure:

$ find .git/refs

.git/refs

.git/refs/heads

.git/refs/tags

$ find .git/refs -type f

That’s basically what a branch in Git is: a simple pointer or reference to the head of a line of work.

**The HEAD**

The HEAD file is a symbolic reference to the branch you’re currently on. By symbolic reference, we mean that unlike a normal reference, it doesn’t generally contain a SHA-1 value but rather a pointer to another reference.

$ cat .git/HEAD

ref: refs/heads/master

When you run git commit, it creates the commit object, specifying the parent of that commit object to be whatever SHA-1 value the reference in HEAD points to.

**Tags**

We just finished discussing Git’s three main object types (*blobs*, *trees* and *commits*), but there is a fourth. The *tag* object is very much like a commit object — it contains a tagger, a date, a message, and a pointer. The main difference is that a tag object generally points to a commit rather than a tree. It’s like a branch reference, but it never moves — it always points to the same commit but gives it a friendlier name

That is all a lightweight tag is — a reference that never moves. An annotated tag is more complex, however. If you create an annotated tag, Git creates a tag object and then writes a reference to point to it rather than directly to the commit. You can see this by creating an annotated tag (using the -a option):

$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'

Here’s the object SHA-1 value it created:

$ cat .git/refs/tags/v1.1

9585191f37f7b0fb9444f35a9bf50de191beadc2

**Remotes**

The third type of reference that you’ll see is a remote reference. If you add a remote and push to it, Git stores the value you last pushed to that remote for each branch in the refs/remotes directory

Then, you can see what the master branch on the origin remote was the last time you

communicated with the server, by checking the refs/remotes/origin/master file:

$ cat .git/refs/remotes/origin/master

ca82a6dff817ec66f44342007202690a93763949

Remote references differ from branches (refs/heads references) mainly in that they’re considered read-only. You can git checkout to one, but Git won’t point HEAD at one, so you’ll never update it with a commit command. Git manages them as bookmarks to the last known state of where those branches were on those servers.

**Packfiles**

The initial format in which Git saves objects on disk is called a “loose” object

format. However, occasionally Git packs up several of these objects into a single binary file called a “packfile” in order to save space and be more efficient. Git does this if you have too many loose objects around, if you run the git gc command manually

If you look in your objects directory, you’ll find that most of your objects are gone, and a new pair of files has appeared:

$ find .git/objects -type f

.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4

.git/objects/info/packs

.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx

.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack

The objects that remain are the blobs that aren’t pointed to by any commit — in this case, the “what is up, doc?” example and the “test content” example blobs you created earlier. Because you never added them to any commits, they’re considered dangling and aren’t packed up in your new packfile.

The other files are your new packfile and an index. The packfile is a single file containing the contents of all the objects that were removed from your filesystem. The index is a file that contains offsets into that packfile so you can quickly seek to a specific object. What is cool is that although the objects on disk before you ran the gc command were collectively about 15K in size, the new packfile is only 7K. You’ve cut your disk usage by half by packing your objects.

How does Git do this? When Git packs objects, it looks for files that are named and sized similarly, and stores just the deltas from one version of the file to the next. You can look into the packfile and see what Git did to save space. The git verify-pack plumbing command allows you to see what was packed up:

$ git verify-pack -v .git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx

**The Refspec**

$ git remote add origin https://github.com/schacon/simplegit-progit

Running the command above adds a section to your repository’s .git/config file, specifying the name of the remote (origin), the URL of the remote repository, and the *refspec* to be used for fetching:

[remote "origin"]

url = https://github.com/schacon/simplegit-progit

fetch = +refs/heads/\*:refs/remotes/origin/\*

The format of the refspec is, first, an optional +, followed by <src>:<dst>, where <src> is the pattern for references on the remote side and <dst> is where those references will be tracked locally. The + tells Git to update the reference even if it isn’t a fast-forward.In the default case that is automatically written by a git remote add command, Git fetches all the references under refs/heads/ on the server and writes them to refs/remotes/origin/ locally. So, if there is a master branch on the server, you can access the log of that branch locally via any of the following:

$ git log origin/master

$ git log remotes/origin/master

$ git log refs/remotes/origin/master

They’re all equivalent, because Git expands each of them to refs/remotes/origin/master.

If you want Git instead to pull down only the master branch each time, and not every other branch on the remote server, you can change the fetch line to refer to that branch only:

fetch = +refs/heads/master:refs/remotes/origin/master

This is just the default refspec for git fetch for that remote. If you want to do a one-time only fetch, you can specify the specific refspec on the command line, too. To pull the master branch on the remote down to origin/mymaster locally, you can run:

$ git fetch origin master:refs/remotes/origin/mymaster

You can also specify multiple refspecs for fetching in your configuration file. If you want to always fetch the master and experiment branches from the origin remote, add two lines:

[remote "origin"]

url = https://github.com/schacon/simplegit-progit

fetch = +refs/heads/master:refs/remotes/origin/master

fetch = +refs/heads/experiment:refs/remotes/origin/experiment

However, you can use namespaces (or directories) to accomplish something like that. If you have a QA team that pushes a series of branches, and you want to get the master branch and any of the QA team’s branches but nothing else, you can use a config section like this:

[remote "origin"]

url = https://github.com/schacon/simplegit-progit

fetch = +refs/heads/master:refs/remotes/origin/master

fetch = +refs/heads/qa/\*:refs/remotes/origin/qa/\*

If you have a complex workflow process that has a QA team pushing branches, developers pushing branches, and integration teams pushing and collaborating on remote branches, you can namespace them easily this way.

**Pushing Refspecs**

It’s nice that you can fetch namespaced references that way, but how does the QA team get their branches into a qa/ namespace in the first place? You accomplish that by using refspecs to push. If the QA team wants to push their master branch to qa/master on the remote server, they can run

$ git push origin master:refs/heads/qa/master

If they want Git to do that automatically each time they run git push origin, they can add a push value to their config file:

[remote "origin"]

url = https://github.com/schacon/simplegit-progit

fetch = +refs/heads/\*:refs/remotes/origin/\*

push = refs/heads/master:refs/heads/qa/master

Again, this will cause a git push origin to push the local master branch to the remote qa/master branch by default.

**Deleting References**

You can also use the refspec to delete references from the remote server by running something like this:

$ git push origin :topic

Because the refspec is <src>:<dst>, by leaving off the <src> part, this basically says to make the topic branch on the remote nothing, which deletes it.

Or you can use the newer syntax (available since Git v1.7.0):

$ git push origin --delete topic

# Submodules

Submodules allow you to keep a Git repository as a subdirectory of another Git repository. This lets you clone another repository into your project and keep your commits separate.

**Starting with Submodules**

To add a new submodule you use the git submodule add command with the absolute or relative URL of the project you would like to start tracking

$ git submodule add <https://github.com/chaconinc/DbConnector>

If you run git status at this point, you’ll notice a few things. First you should notice the new .gitmodules file. This is a configuration file that stores the mapping between the project’s URL and the local subdirectory you’ve pulled it into

$ cat .gitmodules

[submodule "DbConnector"]

path = DbConnector

url = <https://github.com/chaconinc/DbConnector>

If you have multiple submodules, you’ll have multiple entries in this file. It’s important to note that this file is version-controlled with your other files, like your .gitignore file. It’s pushed and pulled with the rest of your project. This is how other people who clone this project know where to get the submodule projects from.

$ git diff DbConnector

Although DbConnector is a subdirectory in your working directory, Git sees it as a submodule and doesn’t track its contents when you’re not in that directory. Instead, Git sees it as a particular commit from that repository. If you want a little nicer diff output, you can pass the --submodule option to git diff.

$ git diff –submodule

**Cloning a Project with Submodules**

When you clone such a project, by default you get the directories that contain submodules, but none of the files within them yet.

The DbConnector directory is there, but empty. You must run two commands: git submodule init to initialize your local configuration file, and git submodule update to fetch all the data from that project and check out the appropriate commit listed in your superproject.

$ git submodule init

$ git submodule update

$ git submodule init

Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbConnector'

$ git submodule update

There is another way to do this which is a little simpler, however. If you pass --recurse-submodules to the git clone command, it will automatically initialize and update each submodule in the repository.

**Working on a Project with Submodules**

**Pulling in Upstream Changes**

If you want to check for new work in a submodule, you can go into the directory and run git fetch and git merge the upstream branch to update the local code.

$ git fetch

$ git merge origin/master

Now if you go back into the main project and run git diff --submodule you can see that the submodule was updated and get a list of commits that were added to it. If you don’t want to type --submodule every time you run git diff, you can set it as the default format by setting the diff.submodule config value to “log”.

$ git config --global diff.submodule log

$ git diff

There is an easier way to do this as well, if you prefer to not manually fetch and merge in the subdirectory. If you run git submodule update --remote, Git will go into your submodules and fetch and update for you.

$ git submodule update --remote DbConnector

This command will by default assume that you want to update the checkout to the master branch of the submodule repository. You can, however, set this to something different if you want. For example, if you want to have the DbConnector submodule track that repository’s “stable” branch, you can set it in either your .gitmodules file (so everyone else also tracks it), or just in your local .git/config file. Let’s set it in the .gitmodules file:

$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote

If you leave off the -f .gitmodules it will only make the change for you, but it probably makes more sense to track that information with the repository so everyone else does as well.

If you set the configuration setting status.submodulesummary, Git will also show you a short summary of changes to your submodules:

$ git config status.submodulesummary 1

C’est une information interessante car vous pouvez voir le journal des modifications

que vous vous appretez a valider dans votre sous-module. Une fois validees, vous pouvez encore visualiser cette information en lancant git log -p.

$ git log -p –submodule

Git will by default try to update **all** of your submodules when you run git submodule update --remote so if you have a lot of them, you may want to pass the name of just the submodule you want to try to update.

$ git submodule update --remote DbConnector

**Working on a Submodule**

So far, when we’ve run the git submodule update command to fetch changes from the submodule repositories, Git would get the changes and update the files in the subdirectory but will leave the sub-repository in what’s called a “detached HEAD” state. This means that there is no local working branch (like “master”, for example) tracking changes

First of all, let’s go into our submodule directory and check out a branch.

$ git checkout stable

Let’s try it with the “merge” option. To specify it manually, we can just add the --merge or --rebase option to our update call. Here we’ll see that there was a change on the server for this submodule and it gets merged in.

$ git submodule update --remote --merge

If we go into the DbConnector directory, we have the new changes already merged into our local stable branch. Now let’s see what happens when we make our own local change to the library and someone else pushes another change upstream at the same time.

Now if we update our submodule we can see what happens when we have made a local change and upstream also has a change we need to incorporate.

$ git submodule update --remote --rebase

If you forget the --rebase or --merge, Git will just update the submodule to whatever is on the server and reset your project to a detached HEAD state.

If you haven’t committed your changes in your submodule and you run a submodule update that would cause issues, Git will fetch the changes but not overwrite unsaved work in your submodule directory.

If you made changes that conflict with something changed upstream, Git will let you know when you run the update. You can go into the submodule directory and fix the conflict just as you normally would.

**Publishing Submodule Changes**

In order to make sure this doesn’t happen, you can ask Git to check that all your submodules have been pushed properly before pushing the main project. The git push command takes the –recurse -submodules argument which can be set to either “check” or “on-demand”. The “check” option will make push simply fail if any of the committed submodule changes haven’t been pushed.

If you want the check behavior to happen for all pushes, you can make this behavior the default by doing git config push.recurseSubmodules check

The other option is to use the “on-demand” value, which will try to do this for you.

$ git push --recurse-submodules=on-demand

As you can see there, Git went into the DbConnector module and pushed it before pushing the main project. If that submodule push fails for some reason, the main project push will also fail. You can make this behavior the default by doing git config push.recurseSubmodules on-demand.

**Submodule Tips**

There are a few things you can do to make working with submodules a little easier.

**Submodule foreach**

There is a foreach submodule command to run some arbitrary command in each submodule. This can be really helpful if you have a number of submodules in the same project.

We can easily stash all the work in all our submodules

$ git submodule foreach git stash

Then we can create a new branch and switch to it in all our submodules.

$ git submodule foreach 'git checkout -b featureA'

$ git diff; git submodule foreach 'git diff'

**Useful Aliases**

You may want to set up some aliases for some of these commands as they can be quite long and you can’t set configuration options for most of them to make them defaults.

$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"

$ git config alias.spush 'push --recurse-submodules=on-demand'

$ git config alias.supdate 'submodule update --remote --merge'

This way you can simply run git supdate when you want to update your submodules, or git spush to push with submodule dependency checking.