Version Control with Git

Table des matières

[1 Introduction 1](#_Toc529309427)

[3 Getting Started 1](#_Toc529309428)

[Viewing your commits 1](#_Toc529309429)

[Viewing Commit differences 1](#_Toc529309430)

[Removing and Renaming Files in your repository 1](#_Toc529309431)

[Configuration Files 2](#_Toc529309432)

[Configuring an Alias 2](#_Toc529309433)

[4 Basic Git Concepts 2](#_Toc529309434)

[Git Object Types 3](#_Toc529309435)

[Inside the .git directory 3](#_Toc529309436)

[Files and Trees 4](#_Toc529309437)

[Tags 4](#_Toc529309438)

[5 File management and the index 5](#_Toc529309439)

[21 Git and Github 5](#_Toc529309440)

# 1 Introduction

A tool that manages and tracks different versions of software or other content is referred to generically as a version control system (VCS), a source code manager (SCM), a revision control system (RCS).

# 3 Getting Started

Git commands understand both “short” and “long” options. For example, the git commit command treats the following examples as equivalents

$ **git commit -m "Fixed a typo."**

$ **git commit --message="Fixed a typo."**

The short form, -m, uses a single hyphen, whereas the long form, --message, uses two.(This is consistent with the GNU long options extension.) Some options exist only in one form.

Finally, you can separate options from a list of arguments via the “bare double dash” convention. For instance, use the double dash to contrast the control portion of the command line from a list of operands, such as filenames.

$ **git diff -w master origin -- tools/Makefile**

You may need to use the double dash to separate and explicitly identify filenames if they might otherwise be mistaken for another part of the command. For example, if you happened to have both a file and a tag named *main.c*, then you will get different behavior:

# Checkout the tag named "main.c"

$ **git checkout main.c**

# Checkout the file named "main.c"

$ **git checkout -- main.c**

Git places all its revision information in this one, top-level *.git* directory.

## Viewing your commits

Once you have one or more commits in the repository, you can inspect them in a variety of ways.

$ **git log**

To see more detail about a particular commit, use git show with a commit number

$ **git show 9da581d910c9c4ac93557ca4859e767f5caf5169**

If you run git show without an explicit commit number, it simply shows the details of the most recent commit.

## Viewing Commit differences

To see the differences between the two revisions of *index.html*, recall both full commit ID names and run git diff:

$ **git diff 9da581d910c9c4ac93557ca4859e767f5caf5169 \**

**ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6**

# Removing and Renaming Files in your repository

Removing a file from a repository is analogous to adding a file but uses git rm.

## Configuration Files

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'**

# 4 Basic Git Concepts

A Git *repository* is simply a database containing all the information needed to retain and manage the revisions and history of a project

Within a repository, Git maintains two primary data structures, the *object store* and the *index*. All of this repository data is stored at the root of your working directory in a hidden subdirectory named *.git*.

The object store is designed to be efficiently copied during a clone operation as part of the mechanism that supports a fully distributed VCS. The index is transitory information, is private to a repository, and can be created or modified on demand as needed.

### Git Object Types

At the heart of Git’s repository implementation is the object store. It contains your original data files and all the log messages, author information, dates, and other information required to rebuild any version or branch of the project. Git places only four types of objects in the object store: the *blobs*, *trees*, *commits*, and *tags*. These four atomic objects form the foundation of Git’s higher level data structures.

*Blobs*

* Each version of a file is represented as a *blob*. Blob, a contraction of “binary large object,” is a term that’s commonly used in computing to refer to some variable or file that can contain any data and whose internal structure is ignored by the program. A blob is treated as being opaque. A blob holds a file’s data but does not contain any metadata about the file or even its name.

*Trees*

* A *tree* object represents one level of directory information. It records blob identifiers, path names, and a bit of metadata for all the files in one directory. It can also recursively reference other (sub)tree objects and thus build a complete hierarchy of files and subdirectories.

*Commits*

* A *commit* object holds metadata for each change introduced into the repository, including the author, committer, commit date, and log message. Each commit points to a tree object that captures, in one complete snapshot, the state of the repository at the time the commit was performed. The initial commit, or *root commit*, has no parent. Most commits have one commit parent, although later in the book (Chapter 9) we explain how a commit can reference more than one parent.

*Tags*

* A *tag* object assigns an arbitrary yet presumably human readable name to a specific object, usually a commit. Although 9da581d910c9c4ac93557ca4859e767f5caf5169 refers to an exact and well-defined commit, a more familiar tag name like Ver-1.0-Alpha might make more sense!

Over time, all the information in the object store changes and grows, tracking and modeling your project edits, additions, and deletions. To use disk space and network bandwidth efficiently, Git compresses and stores the objects in *pack files*, which are also placed in the object store.

The Git object store is organized and implemented as a content-addressable storage system. Specifically, each object in the object store has a unique name produced by applying SHA1 to the contents of the object, yielding an SHA1 hash value

## Inside the .git directory

To begin, initialize an empty repository using git init and then run find to reveal what’s created.

$ **mkdir /tmp/hello**

$ **cd /tmp/hello**

$ **git init**

Initialized empty Git repository in /tmp/hello/.git/

# List all the files in the current directory

$ **find .**

As you can see, *.git* contains a lot of stuff.

Initially, the *.git/objects* directory (the directory for all of Git’s objects) is empty, except

for a few placeholders.

$ **find .git/objects**

Let’s now carefully create a simple object:

$ **echo "hello world" > hello.txt**

$ **git add hello.txt**

If you typed “hello world” exactly as it appears here (with no changes to spacing or

capitalization), then your objects directory should now look like this:

$ **find .git/objects**

The hash in this case is 3b18e512dba79e4c8300dd08aeb37f8e728b8dad. The 160 bits of an SHA1 hash correspond to 20 bytes, which takes 40 bytes of hexadecimal to display, so the content is stored as *.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad*. Git inserts a */* after the first two digits to improve filesystem efficiency.

To show that Git really hasn’t done very much with the content in the file (it’s still the

same comforting “hello world”), you can use the hash to pull it back out of the object

store any time you want:

$ **git cat-file -p 3b18e512dba79e4c8300dd08aeb37f8e728b8dad**

hello world

## Files and Trees

The index is found in *.git/index* and keeps track of file pathnames and corresponding blobs. Each time you run commands such as git add, git rm, or git mv, Git updates the index with the new pathname and blob information

Whenever you want, you can create a tree object from your current index by capturing

a snapshot of its current information with the low-level git write-tree command.

At the moment, the index contains exactly one file, *hello.txt*.

$ **git ls-files -s**

100644 3b18e512dba79e4c8300dd08aeb37f8e728b8dad 0 hello.txt

Here you can see the association of the file, *hello.txt*, and the 3b18e5... blob.

Next, let’s capture the index state and save it to a tree object:

$ **git write-tree**

68aba62e560c0ebc3396e8ae9335232cd93a3f60

$ **find .git/objects**

.git/objects

.git/objects/68

But what does a tree look like? Because it’s an object, just like the blob, you can use

the same low-level command to view it.

$ **git cat-file -p 68aba6**

It is now easy to see that the tree object has captured the information that was in the

index when you ran git ls-files -s.

## Tags

Finally, the last object Git manages is the tag. Although Git implements only one kind

of tag object, there are two basic tag types, usually called *lightweight* and *annotated*.

You create an annotated, unsigned tag with a message on a commit using the git tag

command:

$ **git tag -m "Tag version 1.0" V1.0 3ede462**

You can see the tag object via the git cat-file -p command

Git usually tags a commit object, which points to a tree object, which encompasses the

total state of the entire hierarchy of files and directories within your repository

# 5 File management and the index

When you manage your code with Git, you edit in your working directory, accumulate changes in your index, and commit whatever has amassed in the index as a single changeset.

You can query the state of the index at any time with the command git status. It explicitly calls out what files Git considers staged. You can also peer into the internal state of Git with “plumbing” commands such as git ls-files.

You’ll also likely find the git diff command useful during staging. (Diffs are discussed extensively in Chapter 8.) This command can display two different sets of changes: git diff displays the changes that remain in your working directory and are not staged;

git diff --cached shows changes that are staged and will therefore contribute to your next commit

You can use both variations of git diff to guide you through the process of staging changes. Initially, git diff is a large set of all modifications, and --cached is empty. As you stage, the former set will shrink and the latter set will grow. If all your working changes are staged and ready for a commit, the --cached will be full and git diff will show nothing.

# 21 Git and Github

Disk backup tools like SparkleShare.

*Pushing the Local Contents to GitHub*

Once one of the two options has been followed to connect the local repository to the remote repository, the contents of the local repo can be pushed to GitHub. This is done with the git push *remote branch* command. If the branch has never been published before, the more specific invocation git push -u origin master is appropriate, in which -u tells Git to track the pushed branch, push it to the origin remote, and to push just the master branch

Forks

The next idea that GitHub popularized, so much so that the phrase has spread to other

domains, is personal forks of projects (Figure 21-11). The term forking has commonly

carried a negative connotation. In the coding landscape of yesteryear, forking often

meant an aggressive parting of ways with the primary copy of the project with the intent

of taking the program in a different direction.

GitHub’s idea of forking is a positive one that enables a greater number of contributors

to make a greater number of contributions in a controlled and highly visible way. Forking

is the democratic ability of *any* potential contributor to get a personal copy of a

project’s code.