Git Tutorial

Contents

[1. Introduction 5](#_Toc158798828)

[2. Understanding Git Concepts 6](#_Toc158798829)

[A. What is Git? 6](#_Toc158798830)

[B. The Git Object Database. 6](#_Toc158798831)

[1) Git Objects 6](#_Toc158798832)

[2) Git References 9](#_Toc158798833)

[C. The Git promotion model 10](#_Toc158798834)

[1) The working directory 11](#_Toc158798835)

[2) The staging area 11](#_Toc158798836)

[3) The local repository 12](#_Toc158798837)

[4) The remote repository 12](#_Toc158798838)

[D. Commands in Git 12](#_Toc158798839)

[E. The Treeish 14](#_Toc158798840)

[3. Git Configuration 16](#_Toc158798841)

[A. Setting up a repository 16](#_Toc158798842)

[1) Initializing a new repository: git init 16](#_Toc158798843)

[2) Cloning an existing repository: git clone 16](#_Toc158798844)

[B. Configure git 17](#_Toc158798845)

[1) Git config 17](#_Toc158798846)

[2) Ignoring Files 19](#_Toc158798847)

[4. Committing 21](#_Toc158798848)

[A. Git Add 21](#_Toc158798849)

[Interactive Adding 23](#_Toc158798850)

[How to remove files no longer in the filesystem 31](#_Toc158798851)

[Git rm summary 32](#_Toc158798852)

[5. Branches 33](#_Toc158798853)

[Configuration & set up: git config 33](#_Toc158798854)

[Create a branch 33](#_Toc158798855)

[Merge an ancestor 33](#_Toc158798856)

[Merge a descendent 33](#_Toc158798857)

[Merge two commits from different lineages that both modify the same file 34](#_Toc158798858)

[Repo-to-repo collaboration: git push 34](#_Toc158798859)

[Bare vs. cloned repositories 35](#_Toc158798860)

[Repo-to-repo collaboration 35](#_Toc158798861)

[Interactive Rebasing 35](#_Toc158798862)

[6. Inspecting Repositories 37](#_Toc158798863)

[7. Git diff 38](#_Toc158798864)

[Comparing changes with git diff 38](#_Toc158798865)

[Reading diffs: outputs 38](#_Toc158798866)

[Raw output format 38](#_Toc158798867)

[1. Comparison input 39](#_Toc158798868)

[2. Meta data 39](#_Toc158798869)

[3. Markers for changes 39](#_Toc158798870)

[4. Diff chunks 39](#_Toc158798871)

[Highlighting changes 40](#_Toc158798872)

[1. git diff --color-words 40](#_Toc158798873)

[2. git diff-highlight 40](#_Toc158798874)

[Diffing binary files 40](#_Toc158798875)

[Comparing files: git diff file 41](#_Toc158798876)

[Comparing all changes 41](#_Toc158798877)

[Changes since last commit 41](#_Toc158798878)

[Comparing files between two different commits 41](#_Toc158798879)

[Comparing branches 42](#_Toc158798880)

[Comparing two branches 42](#_Toc158798881)

[Comparing files from two branches 42](#_Toc158798882)

[8. Inspecting a repository 42](#_Toc158798883)

[git log 42](#_Toc158798884)

[Usage 43](#_Toc158798885)

[Discussion 44](#_Toc158798886)

[Example 44](#_Toc158798887)

[9. Git blame 45](#_Toc158798888)

[How It Works 45](#_Toc158798889)

[Common Options 47](#_Toc158798890)

[Git Blame vs Git Log 47](#_Toc158798891)

[Summary 48](#_Toc158798892)

[Reviewing History - Git Log 48](#_Toc158798893)

[Comparing Commits - Git Diff 49](#_Toc158798894)

# Introduction

It is an introduction to Git that is currently the most widely used version control system in the world. The goals of this guide are:

* To shed some light on how Git works under the hood.
* To present the commands (basic and advanced) that you will need to know in order to use Git effectively.

# Understanding Git Concepts

## What is Git?

A version control system is a software designed to keep track of the changes made to files over time. There are a number of benefits to using VCS including the following:

* The ability to undo changes. You can recover an earlier version of you work
* A complete history of all the changes
* Documentation of why changes are made.
* Multiple streams of history.

One of the most popular VCS tools in use today is called Git. Git is a *Distributed* VCS, a category known as DVCS.



Centralized version control



Distributed version control

The essential difference between a Centralized Version Control System (CVCS) and a DVCS here is that users are performing the source management operations against a local copy of the server-side (remote) repository instead of making them against the actual server-side repository. Until users need to push the changes back to the remote, they do not even need to be connected to it. The connection between the local and the remote side is not constant. Rather, it is activated when updates need to be synchronized between the two repositories.

Git is an open source distributed version control system created in 2005 to manage the entire Linux kernel. The Git project spread rapidly, and quickly became used to manage a number of other projects. Git is the technology behind the enormously popular “social coding” website GitHub.

## The Git Object Database.

### Git Objects

The data model of Git is different from other common VCSs in the way Git handles its data. Traditionally, a VCS will store its data as an initial file, followed by a list of patches for each new version of the file. Git does not do this, it records a snapshot of all the files tracked and their paths relative to the repository root. Each commit in Git records the full tree state. If a file does not change between commits, Git will not store the file again.

Git is a version control system built on top of a key value object store. Git creates and stores a collection of objects when you commit. The object store is stored inside the Git repository. It exists entirely in a single .git directory in your project root. There is no central repository like in Subversion. The key is an SHA-1 hash of the object and the value is the object itself. The cryptographic hash function SHA-1 is an alphanumeric sequence of 40 characters representing a hexadecimal number. This hash identifies uniquely the object within the repository.

Git uses objects to track changes throughout the history of a repository. To achieve this tracking, Git uses four types of objects:

* **Blobs**. Git uses blobs to store the contents of a file. A blob is a **Binary Large OBject**. A blob is created when we commence the tracking of a file by using the git add command. 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.
* **Trees**. A tree object in Git can be thought of as a directory. It contains a list of blobs (files) and other tree objects (sub-directories).
* **Commits**:A commit object is essentially a pointer that contains a few pieces of important metadata. The commit itself has a hash, which is built from a combination of the metadata that it contains:
  + - The hash of the tree (the root tree object) at the time of the commit.
    - The hash of any parent commits.
    - The author’s name and email address, and the time that the changes were authored. The author is the name of the person responsible for this change.
    - The committer’s name and email address, and the time that the commit was made. The committer is 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
    - The commit message.
* **Annotated Tags** which point to a single commit object, and contain some metadata. It is a way to mark a specific commit as special in some way.

Almost all of Git is built around manipulating this simple structure of four different object types. In short, the Git data model can be summarized as shown in the following diagram:



So, the objects are tied together, blobs to trees, trees to other trees, and the root tree to the commit object, all connected by the SHA-1 identifier of the object. This implicitly forms a graph of commits known as the commit graph. Specifically, it's a directed acyclic graph (or DAG).



Git keeps all of these objects in the folder .git/objects. This is Git’s object database. Each object, regardless of type, is stored as a file, using its SHA-1 checksum as the filename. But, instead of storing all objects in a single folder, they are split up using the first two characters of their ID as a directory name, resulting in an object database that looks something like the following.

$ find .git/objects

.git/objects

.git/objects/00

.git/objects/00/11f080776acf2d04fb99b0d5c70f85747420a9

.git/objects/01

.git/objects/01/9da3ea8f032c4ebf7825cc13b5eeecc7cf017d

.git/objects/01/c3abfb09d4c4b2b306de4b20188574d4e02914

.git/objects/02

.git/objects/02/2d0352de4df1478f1f6571d0cf52ff22611f9f

.git/objects/03

.git/objects/03/1d9f1c82db42c05df688aed50bcea31bf7554b

For example, an object with the following ID:

022d0352de4df1478f1f6571d0cf52ff22611f9f

is stored in a folder called 02, using the remaining characters (2d0352...) as a filename.

This highlights once again the simplicity of Git: no metadata, no internal databases, or useless complexity, but simple files and folders are enough to make it possible to manage any repository.

### Git 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.

Git references come in various types, including branch references, tag references, and remote references:

* **Branch references** are the most common type of reference in Git. They are used to keep track of different branches of development in your repository. As you make changes and commit them, the branch reference is updated to point to the latest commit in the branch.
* **Tag references** are used to mark specific commits as important milestones or releases. Unlike branch references, tag references are static and do not change as new commits are added to the repository.
* **Remote references** are used to keep track of branches in remote repositories. When you clone a repository, Git creates remote references for the branches in the remote repository.

Git stores all of its references in the directory .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.

$ find .git/refs

.git/refs/heads/ContainerNR646

.git/refs/heads/develop

.git/refs/heads/master

.git/refs/heads/URS18\_Avril22

.git/refs/remotes/origin/AmendementJanv24

.git/refs/remotes/origin/Berthing

.git/refs/remotes/origin/CheckRatioPolar

.git/refs/remotes/origin/ComparaisonDouble2

.git/refs/remotes/origin/ContainerNR646

.git/refs/remotes/origin/Deflexion

.git/refs/remotes/origin/Dev4.2

.git/refs/remotes/origin/develop

.git/refs/remotes/origin/ExportExcel

.git/refs/remotes/origin/FactorisationNR467

.git/refs/remotes/origin/FloodingNR467

.git/refs/remotes/origin/Gaziers

.git/refs/remotes/origin/HEAD

.git/refs/remotes/origin/HeavyBallastDraught

.git/refs/remotes/origin/Mars\_4.2

.git/refs/remotes/origin/master

.git/refs/remotes/origin/MA\_1624

.git/refs/remotes/origin/Methanol

.git/refs/remotes/origin/MigrationNetStandard

.git/refs/remotes/origin/NeutralXmlDev

.git/refs/remotes/origin/OM9july

.git/refs/remotes/origin/OpenHatch

.git/refs/remotes/origin/Polar

.git/refs/remotes/origin/Release\_4.1

.git/refs/remotes/origin/SplitDredgers

.git/refs/remotes/origin/SteelMars

.git/refs/remotes/origin/TestDDV\_yieldingEpp

.git/refs/remotes/origin/URS18\_Avril22

.git/refs/remotes/origin/Version\_4.0.2

.git/refs/remotes/origin/YieldingEppLocale\_v10

.git/refs/remotes/origin/YieldingEppLocale\_v3

.git/refs/tags/Mars\_3.7

.git/refs/tags/Mars\_4.0

.git/refs/tags/Mars\_4.0.1

.git/refs/tags/Mars\_4.2

Git uses also a special reference HEAD to point to the current commit. You can see where it's currently pointing by checking the .git/HEAD file. Normally, HEAD points to another reference.

$ cat .git/HEAD

ref: refs/heads/develop

$ cat .git/refs/heads/develop

8eb85f55d7b4ff1c0f1205f5ee3a681ebf1a4188

## The Git promotion model

The Git model provides a local environment where you can work with a local copy of a remote repository. This local environment exists for users to create and update content and get it in the form they want before making it available or visible to others, in the remote repository.

The combination of the working directory, staging area, and local repository make up your local environment. The following diagram describes the tree stages and the commands used to move between the stages. Git manages and manipulates three stages in its normal operation.



The distinction between the working directory, the staged snapshot, and committed snapshots is at the core of Git version control. Most Git commands operate on one of the three main components of a Git repository:

### The working directory

Starting at the bottom is the working directory where content is created, edited, deleted, and so on. Any new content must exist here before it can be put into (tracked by) Git. Files in this directory are often removed or replaced by Git as you switch branches. The working directory is simply a temporary checkout place where you can modify the files until your next commit. All subdirectories are considered part of the working directory’s scope, unless Git is specifically told to ignore them via a .gitignore file or they are part of a Git *submodule*.

### The staging area

The staging area is an intermediate level between the working directory and the local repository. The staging area is a file, generally contained in your .git directory, that stores information about what will go into your next commit.

When a file is moved to the staging area, the SHA-1 hash of the file is created and the blob object is written to Git's database. At first glance, it may seem like an unnecessary intermediate level that gets in the way of trying to promote content from the working directory to the local repository. In fact, it plays a significant role in several parts of Git’s functionality. One part where the staging operation is required is when you need to complete a merge operation that had conflicts. Git stages files that merged successfully. In order to complete the merge, files that have conflicts manually resolved must be staged. This creates a complete set of content to be committed to complete the merge operation.

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. The primary function of the git add command, is to promote pending changes in the working directory, to the git staging area.

### The local repository

The .git 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 and that directory is in the root of your project.

### The remote repository

The remote repository is a separate Git repository intended to collect and host content pushed to it from one or more local repositories. Its main purpose is to be a place to share and access content from multiple users.

## Commands in Git

In Git the commands are divided up into the two categories:

* The *plumbing commands* function at a lower level and are not expected to be used by the average user. These commands are typically targeted at extracting or modifying content and information more directly from the repository. An example would be the git cat-file or git ls-files commands that provide a way to look at the contents of a file or directory within the repository.
* The *porcelain commands* are intended to be user-facing, more commonly used, and more convenient. The porcelain commands are based on the plumbing commands. They aggregate the functionality of plumbing commands and certain options and sequences in order to make things simpler for the typical Git user.

The general form of commands is a as follows:

$ git <command> <command-options> <operands>

Table 1 describes the different parts of this form.

|  |  |  |
| --- | --- | --- |
|  | Description | Examples |
| <command> | Git command to execute | $ git push |
| <command-options> | Options to the specified command | $ git commit -m “comment” |
| <operands> | Items for the command to operate on | $ git add \*.c |

Table1

The Git operand could be a commit | branch | tag or a filename. When both types are specified, if there is a possibility of Git not being able to tell the difference between a commit | branch | tag and one or more of the filenames or paths, then you can separate the two types using the special separation symbol “--”. Normally, this won’t be needed if a commit is expressed as a SHA1 value, but it may be needed if branch or tag names could be mistaken as names for files or paths.

As an example, the command git checkout a1b2c3d4 file1.txt might be clear enough, but git checkout my-tag-name -- my-file-name could be ambiguous enough when parsed to require the “--” separator symbol.

Table 2 shows a categorization of the porcelain (user-friendly) commands that are available in Git.

|  |  |
| --- | --- |
| Command | Purpose |
| add | Add files contents to the index |
| branch | List, create, or delete branches |
| checkout | Switch branches or restore working tree files. |
| cherry | Find commits yet to be applied to upstream (branch on the remote). |
| cherry-pick | Apply the changes introduced by some existing commits. |
| clean | Removes untracked files from the working directory. This is the logical counterpart to git reset, which (typically) only operates on tracked files. |
| clone | Clone a repository into a new directory. |
| commit | Record changes to the repository |
| config | Get and set repository or global options. |
| diff | Show changes between commits, commits and working tree, and so on |
| fetch | Download objects and refs from another repository |
| grep | Print lines matching a pattern |
| help | Display help information |
| init | Initializes a new Git repository |
| log | Show commit logs. |
| merge | A powerful way to integrate changes from divergent branches |
| mv | Move or rename a file, directory, or symlink. |
| pull | Runs a ‘git fetch’ then a ‘git merge’ |
| push | Update remote refs along with associated objects. |
| rebase | Forward-port local commits to the updated upstream head |
| rerere | Reuse recorded resolution for merged conflicts. |
| remote | Lists all the remote versions of your repository, or can be used to add and delete them |
| reset | Reset current HEAD to the specified state. |
| revert | Revert some existing commits. |
| rm | Remove files from the working tree and from the index. |
| show | Show various types of objects. |
| stash | Temporarily saves changes that you don’t want to commit immediately for later. |
| status | Show the working tree status. |
| submodule | Initialize, update, or inspect submodules. |
| subtree | Merge subtrees and split repositories into subtrees. |
| tag | Create, list, delete, or verify a tagged object. |
| worktree | Manage multiple working tree |

Table 2

Table 3 below shows the same categorization for the plumbing commands. These commands have names that indicate an action and an object to operate against as opposed to the simpler naming of the porcelain commands.

|  |  |
| --- | --- |
| cat-file | Provide content or type and size information for repository objects |
| commit-tree | Create a new commit object. |
| count-objects | Count an unpacked number of objects and their disk consumption. |
| diff-index | Compare a tree to the working tree or index. |
| for-each-ref | Output information on each ref. |
| hash-object | Compute object ID and optionally create a blob from a file. |
| ls-files | Show information about files in the index and the working tree. |
| merge-base | Find as good common ancestors as possible for a merge. |
| read-tree | Read tree information into the index. |
| rev-list | List commit objects in reverse chronological order. |
| rev-parse | Pick out and massage parameters. |
| show-ref | List references in a local repository. |
| symbolic-ref | Read, modify, and delete symbolic refs. |
| update-index | Register file contents in the working tree to the index. |
| update-ref | Update the object name stored in a ref safely. |
| verify-pack | Validate packed Git archive files. |
| write-tree | Create a tree object from the current index. |

Table 3

Arguments supplied to Git commands can be abbreviated as a single letter or spelled out as words. One important note here is that if the argument is spelled out, you must precede it with two hyphens, as in --global. If the argument is abbreviated, only one hyphen is required, as in -a. Abbreviated arguments may be passed together, as in -am instead of -a -m.

$ git help glossary

$ git help –a //list of over 150 commands

$ git help –g //list of common guides

$ git help config

$ git config –h

$ git config --help

## The Treeish

Besides Branch or tag names, 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

$ git show dae86e1950b1277e545cee180551750029cfe735

* Partial SHA-1.

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.

$ git show dae86e

* Date spec

$ git show master@{yesterday} # where master was yesterday

$ git show [master@{1.month.ago}](mailto:master@%7b1.month.ago%7d) # where master was a month ago

$ git show [master@{1.week.ago}](mailto:master@%7b1.week.ago%7d) # where master was a week ago

* Ordinal spec

$ git show 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

$ git show dae86e^N

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

* Tilde spec

$ git show dae86e~N

This refers to the Nth generation grandparent of that commit

dae86e~5 ⬄ dae86e^^^^^



* Tree pointer

$git cat-file -p e65s46^{tree}

Any time you add a ^{tree} to any commit, it resolves to its tree.

* Blob spec

$git cat-file -p master:/path/to/file

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

# Git Configuration

## Setting up a repository

### Initializing a new repository: git init

To create a new repo, you'll use the git init command. Executing this command will create a new .git subdirectory in your current working directory and will also create a new master branch.

*$ mkdir demo*

*$ cd demo*

*$ git init*

It can be used to convert an existing, unversioned project to a Git repository or initialize a new, empty repository. Most other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project.

This command can take a directory argument. If you provide the directory, the command is run inside it. If this directory does not exist, it will be created.

*$ git init <directory>*

Aside from the .git directory, in the root directory of the project, an existing project remains unaltered (unlike SVN, Git doesn't require a .git subdirectory in every subdirectory).

Git is designed to be as unobtrusive as possible. The .git folder is the only difference between a Git repository and an ordinary folder, so deleting it will turn your project back into an unversioned collection of files

The .git folder is created with various files and directories in the current directory.

$ find .git

.git/config //Contains the configuration of the local repository

.git/HEAD //Head pointer

.git/hooks

...

.git/objects // Object storage

.git/objects/info

.git/objects/pack

.git/refs

.git/refs/heads //Contains the branch pointers

.git/refs/tags //Contains the tag pointers

### Cloning an existing repository: git clone

git clone is a Git command line utility which is used to target an existing repository and create a clone of the target repository.

git clone <repo url>

$ git clone <https://mar-gitlab.bureauveritas.com/applications/dv/mars.git>

$cd mars

Git creates refs to the remote branch heads under refs/remotes/origin.

**Cloning to a specific folder:**

git clone <repo> <directory>

Clone the repository located at <repo> into the folder called ~<directory> on the local machine.

**Cloning a specific branch:**

git clone -branch <branch> --single-branch <repo>

You can limit the amount of the history you clone by specifying that you want to clone only a single branch

**Shallow clone**

git clone -depth=1 <repo>

Clone the repository located at <repo> and only clone the history of commits specified by the option depth=1. In this example a clone of <repo> is made and only the most recent commit is included in the new cloned Repo. Shallow cloning is most useful when working with repos that have an extensive commit history. An extensive commit history may cause scaling problems such as disk space usage limits and long wait times when cloning. A Shallow clone can help alleviate these scaling issues.

**Clone a git repository with submodules**

git clone <repo> --recurse-submodules

If you pass --recurse-submodules to the git clone command, it will automatically initialize and update each submodule in the repository

## Configure git

### Git config

In addition to configuring a remote repo URL, you may also need to set global Git configuration options such as username, or email. The git config command lets you configure your Git installation (or an individual repository) from the command line. This command can define everything from user info, to preferences, to the behavior of a repository.

The git config command can accept arguments to specify which configuration level to operate on. The following configuration levels are available:

* **Local**

By default, git config will write to a local level if no configuration option is passed. Local level configuration is applied to the context repository git config gets invoked in. Local configuration values are stored in a file that can be found in the repo's .git directory: .git/config

* **Global**

Global level configuration is user-specific, meaning it is applied to an operating system user. Global configuration values are stored in a file that is located in a user's home directory C:\Users\<username>\.gitconfig on windows

* **System**

System-level configuration is applied across an entire machine. This covers all users on an operating system and all repos. On windows this file can be found at C:\ProgramData\Git\config on Windows 10 and newer.

Thus the order of priority for configuration levels is: local, global, system

This is because in Git, every modification you make in a repository has to be signed with the name and email of the author. So, before doing anything else, we have to tell Git this information.

$ git config --global user.name "Your Name"

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

So, you prefer setting up usernames and emails per repository:

$ git config user.name "Your Name"

$ git config user.email [your.email@example.com](mailto:your.email@example.com)

Git’s command line relies on a text editor for most of its input. You can forece Git to use your editor of choice with the core.editor option (By default, Git uses Vim).

$ git config --global core.editor notepad

You could select your favorite diff-tool and merge-tool

git config --global diff.tool bc

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

To disable the "Launch 'bc3' [Y/n]?" prompt, run the command:

git config --global difftool.prompt false

git config --global merge.tool bc

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

Git aliases are a powerful workflow tool that create shortcuts to frequently used Git commands. Using Git aliases will make you a faster and more efficient developer. Aliases can be used to wrap a sequence of Git commands into new faux Git command. Git aliases are created through the use of the git config command which essentially modifies local or global Git config files. Learn more on the [git config](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) page

Git supports aliasing commands. This is a powerful utility to create custom shortcuts for commonly used git commands. A simplistic example would be

$ git config --global alias.graph "log --graph --oneline --all"

$ git graph -50

This creates a graph command that you can execute as a shortcut to git log –graph –oneline --all.

git config --global --edit

Open the global configuration file in a text editor for manual editing.

When options in these files conflict, local settings override user settings, which override system-wide. If you open any of these files, you’ll see something like the following:

The git config command is a convenience function that is used to set Git configuration values on a global or local project level. These configuration levels correspond to .gitconfig text files. Executing git config will modify a configuration text file.

Overall, git config is a helper tool that provides a shortcut to editing raw git config files on disk.

The most basic use case for git config is to invoke it with a configuration name, which will display the set value at that name. Configuration names are dot delimited strings composed of a 'section' and a 'key' based on their hierarchy.

$ git config user.email

### Ignoring Files

To tell Git to ignore specific files or directories (meaning not to track them), you just need to list them in a Git ignore file. This is a text file named .gitignore that is placed at the root (top level directory) of the local environment

Git sees every file in your working copy as one of three things:

* tracked - a file which has been previously staged or committed;
* untracked - a file which has not been staged or committed
* ignored - a file which Git has been explicitly told to ignore.

Ignored files are usually build artifacts and machine generated files that can be derived from your repository source or should otherwise not be committed. Some common examples are:

* compiled code, such as .obj
* build output directories, such as /bin, /debug or /release
* personal IDE config files, such as .vs/

Any files or directories that you'd like to ignore should be included on a separate line, and the \* symbol can be used as a wildcard.

$ cat .gitignore

# User-specific files

\*.aps

\*.eto

\*.user

\*.suo

\*.user

Build results

[Bb]in/

[Dd]ebug/

[Rr]elease/

x64/

x86/

bld/

[Bb]in/

[Oo]bj/

[Ll]og/

# Committing



The stage/commit process

cd /path/to/demo

echo "test content for git tutorial" >> CommitTest.txt

git add CommitTest.txt

git commit -m "added CommitTest.txt to the repo"

After executing this example, your repo will now have CommitTest.txt added to the history and will track future updates to the file.

The commands: git add, [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository), and [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) are all used in combination to save a snapshot of a Git project's current state. Theses commands compose the fundamental Git workflow

## Git Add

The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit).

In conjunction with these commands, you'll also need [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) to view the state of the working directory and the staging area.

|  |  |
| --- | --- |
| git add <file> | Stage all changes in <file> for the next commit. |
| git add <directory> | Stage all changes in <directory> for the next commit. |
| git add -p | Begin an interactive staging session that lets you choose portions of a file to add to the next commit. This will present you with a chunk of changes and prompt you for a command. Use y to stage the chunk, n to ignore the chunk, s to split it into smaller chunks, e to manually edit the chunk, and q to exit. |
| git add -i |  |

Instead of committing all of the changes you've made since the last commit, the stage lets you group related changes into highly focused snapshots before actually committing it to the project history. This means you can make all sorts of edits to unrelated files, then go back and split them up into logical commits by adding related changes to the stage and commit them piece-by-piece

Git’s staging area gives you a plcae to organize a commit before adding it to the project history. Staging is the process of moving changes from the working directory to the staged snapshot. It gives you the opportunity to pick and choose related changes from the working directory, instead of committing everything all at once

git add adds the file to the index. The index is a list that contains every file that Git has been told to keep track of. It is stored as a file at .git/index.

The [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) command is used to undo a git add.

If you need more detailed information about the changes in your working directory or staging area, you can generate a diff

$ git diff

This outputs a diff of every unstaged changes in your working directory. You can also generate a diff of all staged changes with the --cached flag

$ git diff --cached

## Git Status

The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven’t, and which files aren’t being tracked by Git.

While the git status output is pretty comprehensive, it’s also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run git status -s or git status --short you get a far more simplified output from the command:

$ git status –s

New files that aren’t tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M and so on. There are two columns to the output - the lefthand column indicates the status of the staging area and the right-hand column indicates the status of the working tree. So for example in that output, the README file is modified in the working directory but not yet staged, while the lib/simplegit.rb file is modified and staged. The Rakefile was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

To see what you’ve changed but not yet staged, type git diff with no other arguments. That command compares what is in your working directory with what is in your staging area. The result tells you the changes you’ve made that you haven’t yet staged.

If you want to see what you’ve staged that will go into your next commit, you can use git diff --staged. This command compares your staged changes to your last commit:

It’s important to note that git diff by itself doesn’t show all changes made since your last commit — only changes that are still unstaged. If you’ve staged all of your changes, git diff will give you no output.

Now you can use git diff to see what is still unstaged and git diff --cached to see what you’ve staged so far (--staged and --cached are synonyms):

We will continue to use the git diff command in various ways throughout the rest of the book. There is another way to look at these diffs if you prefer a graphical or external diff viewing program instead. If you run git difftool instead of git diff, you can view any of these diffs in software like emerge, vimdiff and many more (including commercial products). Run git difftool --tool-help to see what is available on your system

git status list which files are staged, unstaged, and untracked.

The git status command is a relatively straightforward command. It simply shows you what's been going on with git add and git commit. Status messages also include relevant instructions for staging/unstaging files. Sample output showing the three main categories of a git status call is included below:

It's good practice to check the state of your repository before committing changes so that you don't accidentally commit something you don't mean to.

It would be helpful to view the status of the new repository.

$ git status

$ git status -s

It outputs the state of the working directory and staging aea

It’s a good practice to run git status to see exactly what you’re committing before running git commit

To delete a file from project, you need to add it to the staging area like a new or modified file.The next command will stage the deletion and stop tracking the file but it won’t delete the file from he working directory

$ git rm --cached file1.txt

In addition to git add and git commit, a third command [git push](https://www.atlassian.com/git/tutorials/syncing) is essential for a complete collaborative Git workflow. git push is utilized to send the committed changes to remote repositories for collaboration. This enables other team members to access a set of saved changes.

. The [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) command is used to examine the current state of the repository and can be used to confirm a git add promotion

## Git Commit

The git commit command captures a snapshot of the project's currently staged changes. Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicitly ask it to. Prior to the execution of git commit, The [git add](https://www.atlassian.com/git/tutorials/saving-changes) command is used to promote or 'stage' changes to the project that will be stored in a commit. These two commands git commit and git add are two of the most frequently used.

In Git, repositories are distributed, Snapshots are committed to the local repository, and this requires absolutely no interaction with other Git repositories. Git commits can later be pushed to arbitrary remote repositories.

Aside from the practical distinctions between SVN and Git, their underlying implementation also follows entirely divergent design philosophies. Whereas SVN tracks differences of a file, Git’s version control model is based on snapshots. For example, a SVN commit consists of a diff compared to the original file added to the repository. Git, on the other hand, records the entire contents of each file in every commit.

This makes many Git operations much faster than SVN, since a particular version of a file doesn’t have to be “assembled” from its diffs—the complete revision of each file is immediately available from Git's internal database.

Git's snapshot model has a far-reaching impact on virtually every aspect of its version control model, affecting everything from its branching and merging tools to its collaboration work-flows.

|  |  |
| --- | --- |
| git commit | Commit the staged snapshot. This will launch a text editor prompting you for a commit message. After you’ve entered a message, save the file and close the editor to create the actual commit. |
| git commit -a | Commit a snapshot of all changes in the working directory. This only includes modifications to tracked files (those that have been added with git add at some point in their history). |
| git commit -m "commit message" | A shortcut command that immediately creates a commit with a passed commit message. By default, git commit will open up the locally configured text editor, and prompt for a commit message to be entered. Passing the -m option will forgo the text editor prompt in-favor of an inline message. |
| git commit -am "commit message" | A power user shortcut command that combines the -a and -m options. This combination immediately creates a commit of all the staged changes and takes an inline commit message. |
| git commit --amend | This option adds another level of functionality to the commit command. Passing this option will modify the last commit. Instead of creating a new commit, staged changes will be added to the previous commit. This command will open up the system's configured text editor and prompt to change the previously specified commit message.  Passing the --amend flag to git commit lets you amend the most recent commit. This is very useful when you forget to stage a file or omit important information from the commit message. |

Git doesn't require commit messages to follow any specific formatting constraints, but the canonical format is to summarize the entire commit on the first line in less than 50 characters, leave a blank line, then a detailed explanation of what’s been changed. For example:

Change the message displayed by hello.py

- Update the sayHello() function to output the user's name

- Change the sayGoodbye() function to a friendlier message

It is a common practice to use the first line of the commit message as a subject line, similar to an email. The rest of the log message is considered the body and used to communicate details of the commit change set. Note that many developers also like to use the present tense in their commit messages. This makes them read more like actions on the repository, which makes many of the history-rewriting operations more intuitive.

git add hello.py

git commit –amend

This will once again, open up the configured text editor. This time, however, it will be pre-filled with the commit message we previously entered. This indicates that we are not creating a new commit, but editing the last.

The git commit command is one of the core primary functions of Git. Prior use of the git add command is required to select the changes that will be staged for the next commit. Then git commit is used to create a snapshot of the staged changes along a timeline of a Git projects history. Learn more about [git add](https://www.atlassian.com/git/tutorials/saving-changes) usage on the accompanying page. The [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) command can be used to explore the state of the staging area and pending commit.

If you want to skip the staging area, Git provides a simple shortcut. Adding the -a option to the git commit command makes Git automatically stage every file that is already tracked before doing the commit,

$ git commit -am 'added new benchmarks'

$ 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

Git commits can be captured and built up locally, then pushed to a remote server as needed using the git push -u origin master command.

The commit command has three steps. It creates a tree graph to represent the content of the version of the project being committed. It creates a commit object. It points the current branch at the new commit object

* Git records the current state of the project by creating a tree graph from the index. This tree graph records the location and content of every file in the project.

The graph is composed of two types of object: blobs and trees.

Blobs are stored by git add. They represent the content of files.

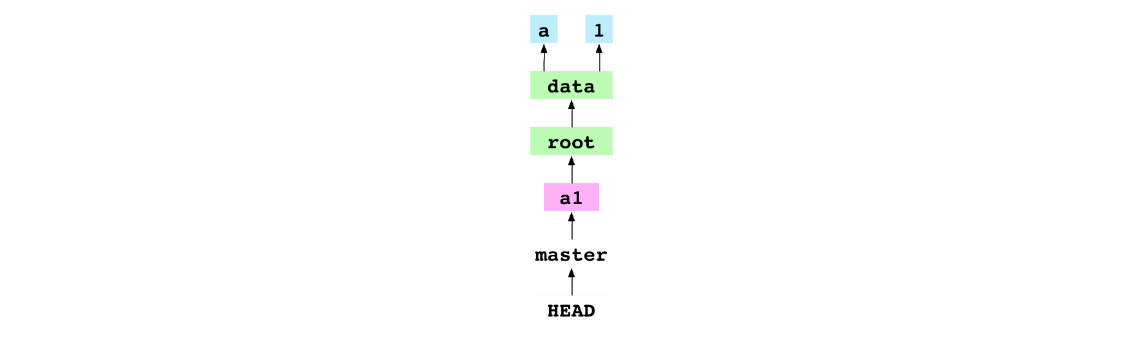
Trees are stored when a commit is made. A tree represents a directory in the working copy.

* git commit creates a commit object after creating the tree graph. The commit object is just another text file in .git/objects/:
* Finally, the commit command points the current branch at the new commit object. Which is the current branch? Git goes to the HEAD file at .git/HEAD and finds:

ref: refs/heads/master

This says that HEAD is pointing at master. master is the current branch.

HEAD and master are both refs. A ref is a label used by Git or the user to identify a specific commit.



HEAD pointing at master and master pointing at the a1 commit

Commits represent every saved version of a project which makes them the atomic unit of Git based version control.

The next command will open a text editor and prompt you to enter a message for the commit.

$ git commit

The -m option lets you specify a commit message on the command line instead of opening a text editor. This is just a convenient shortcut.

$ git commit -m "Add navigation links"

The --message flag for git commit can be abbreviated to -m (all abbreviations use a single -). If this flag is omitted, Git opens a text editor (specified by the EDITOR or GIT\_EDITOR environment variable) to prompt you for the commit message

It can also take the --all (or -a) flag to add all changes to files tracked in the repository into a new commit

Saving a version of your project is a two step process:

* **Staging.** Telling Git what files to include in the next commit.
* **Committing.** Recording the staged snapshot with a descriptive message.

Staging files with the git add command doesn’t actually affect the repository in any significant way—it just lets us get our files in order for the next commit. Only after executing git commit will our snapshot be recorded in the repository. Committed snapshots can be seen as “safe” versions of the project.

Our history can now be represented as the following. Note that the red circle, which represents the current commit, automatically moves forward every time we commit a new snapshot.



Current project history

Notice that we skipped the staging step this time around. Instead of using git add, we passed the -a flag to git commit. This convenient parameter tells Git to automatically include *all* tracked files in the staged snapshot. Combined with the -m flag, we can stage and commit snapshots with a single command

Rather than require all changes in the working tree to build up new commits, git allows files to be added incrementally to the index.

**Git rm**

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The git rm command does that, and also removes the file from your working directory so you don’t see it as an untracked file the next time around.

The next time you commit, the file will be gone and no longer tracked. If you modified the file and added it to the staging area already, you must force the removal with the -f option. This is a safety feature to prevent accidental removal of data that hasn’t yet been recorded in a snapshot and that can’t be recovered from Git.

Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your .gitignore file and accidentally staged it, like a large log file or a bunch of .a compiled files. To do this, use the --cached option:

$ git rm --cached README

That means you can do things such as:

$ git rm log/\*.log

$ git rm /\*~

**Moving Files**

Unlike many other VCS systems, Git doesn’t explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file. However, Git is pretty smart about figuring that out after the fact — we’ll deal with detecting file movement a bit later. Thus it’s a bit confusing that Git has a mv command. If you want to rename a file in Git, you can run something like:

$ git mv file\_from file\_to

However, this is equivalent to running something like this:

$ mv README.md README

$ git rm README.md

$ git add README

Git figures out that it’s a rename implicitly, so it doesn’t matter if you rename a file that way or with the mv command. The only real difference is that git mv is one command instead of three — it’s a convenience function. More importantly, you can use any tool you like to rename a file, and address

the add/rm later, before you commit.

A common question when getting started with Git is "How do I tell Git not to track a file (or files) any more?" The git rm command is used to remove files from a Git repository. It can be thought of as the inverse of the [git add](https://www.atlassian.com/git/tutorials/saving-changes) command.

**Git rm Overview**

The git rm command can be used to remove individual files or a collection of files. The primary function of git rm is to remove tracked files from the Git index. Additionally, git rm can be used to remove files from both the staging index and the working directory. There is no option to remove a file from only the working directory. The files being operated on must be identical to the files in the current HEAD. If there is a discrepancy between the HEAD version of a file and the staging index or working tree version, Git will block the removal. This block is a safety mechanism to prevent removal of in-progress changes.

Note that git rm does not remove branches. Learn more about [using git branches](https://www.atlassian.com/git/tutorials/using-branches)

**Usage**

<file>…​

Specifies the target files to remove. The option value can be an individual file, a space delimited list of files file1 file2 file3, or a wildcard file glob (~./directory/\*).

-f  
--force

The -f option is used to override the safety check that Git makes to ensure that the files in HEAD match the current content in the staging index and working directory.

-n  
--dry-run

The "dry run" option is a safeguard that will execute the git rm command but not actually delete the files. Instead it will output which files it would have removed.

-r

The -r option is shorthand for 'recursive'. When operating in recursive mode git rm will remove a target directory and all the contents of that directory.

--

The separator option is used to explicitly distinguish between a list of file names and the arguments being passed to git rm. This is useful if some of the file names have syntax that might be mistaken for other options.

--cached

The cached option specifies that the removal should happen only on the staging index. Working directory files will be left alone.

--ignore-unmatch

This causes the command to exit with a 0 sigterm status even if no files matched. This is a Unix level status code. The code 0 indicates a successful invocation of the command. The --ignore-unmatch option can be helpful when using git rm as part of a greater shell script that needs to fail gracefully.

-q  
--quiet

The quiet option hides the output of the git rm command. The command normally outputs one line for each file removed.

**How to undo git rm**

Executing git rm is not a permanent update. The command will update the staging index and the working directory. These changes will not be persisted until a new commit is created and the changes are added to the commit history. This means that the changes here can be "undone" using common Git commands.

git reset HEAD

A reset will revert the current staging index and working directory back to the HEAD commit. This will undo a git rm.

git checkout .

A checkout will have the same effect and restore the latest version of a file from HEAD.

In the event that git rm was executed and a new commit was created which persist the removal, git reflog can be used to find a ref that is before the git rm execution. Learn more about [using git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog).

**Discussion**

The <file> argument given to the command can be exact paths, wildcard file glob patterns, or exact directory names. The command removes only paths currently commited to the Git repository.

Wildcard file globbing matches across directories. It is important to be cautious when using wildcard globs. Consider the examples: directory/\* and directory\*. The first example will remove all sub files of directory/ whereas the second example will remove all sibling directories like directory1 directory2 directory\_whatever which may be an unexpected result.

**The scope of git rm**

The git rm command operates on the current branch only. The removal event is only applied to the working directory and staging index trees. The file removal is not persisted to the repository history until a new commit is created.

**Why use git rm instead of rm**

A Git repository will recognize when a regular shell rm command has been executed on a file it is tracking. It will update the working directory to reflect the removal. It will not update the staging index with the removal. An additional git add command will have to be executed on the removed file paths to add the changes to the staging index. The git rm command acts a shortcut in that it will update the working directory and the staging index with the removal.

**Examples**

git rm Documentation/\\*.txt

This example uses a wildcard file glob to remove all \*.txt files that are children of the Documentation directory and any of its subdirectories.

Note that the asterisk \* is escaped with slashes in this example; this is a guard that prevents the shell from expanding the wildcard. The wildcard then expands the pathnames of files and subdirectories under the Documentation/ directory.

git rm -f git-\*.sh

This example uses the force option and targets all wildcard git-\*.sh files. The force option explicitly removes the target files from both the working directory and staging index.

## How to remove files no longer in the filesystem

As stated above in "Why use git rm instead of rm" , git rm is actually a convenience command that combines the standard shell rm and git add to remove a file from the working directory and promote that removal to the staging index. A repository can get into a cumbersome state in the event that several files have been removed using only the standard shell rm command.

If intentions are to record all the explicitly removed files as part of the next commit, git commit -a will add all the removal events to the staging index in preparation of the next commit.

If however, intentions are to persistently remove the files that were removed with the shell rm, use the following command:

git diff --name-only --diff-filter=D -z | xargs -0 git rm --cached

This command will generate a list of the removed files from the working directory and pipe that list to git rm --cached which will update the staging index.

## Git rm summary

git rm is a command that operates on two of the primary Git [internal state management trees](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset): the working directory, and staging index. git rm is used to remove a file from a Git repository. It is a convenience method that combines the effect of the default shell rm command with git add. This means that it will first remove a target from the filesystem and then add that removal event to the staging index. The command is one of many that can be used for [undoing changes in Git.](https://www.atlassian.com/git/tutorials/undoing-changes)

# Branches

## Configuration & set up: git config

Once you have a remote repo setup, you will need to add a remote repo url to your local git config, and set an upstream branch for your local branches. The git remote command offers such utility.

git remote add <remote\_name> <remote\_repo\_url>

This command will map remote repository at <remote\_repo\_url> to a ref in your local repo under <remote\_name>. Once you have mapped the remote repo you can push local branches to it.

git push -u <remote\_name> <local\_branch\_name>

This command will push the local repo branch under <local\_branc\_name> to the remote repo at <remote\_name>.

* How to customize git colors

### Create a branch

git branch deputy

The user creates a new branch called deputy. This just creates a new file at .git/refs/heads/deputy that contains the hash that HEAD is pointing at: the hash of the a3 commit.

Merging two branches means merging two commits

### Merge an ancestor

For this merge, Git does nothing. It reports it is Already up-to-date.

### Merge a descendent

git checkout master

git merge deputy

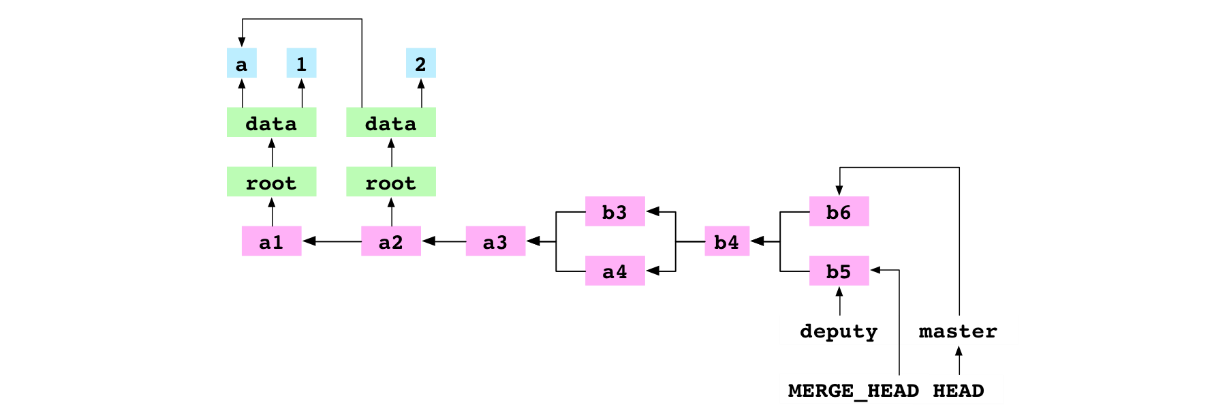
They merge deputy into master. Git discovers that the receiver commit, a2, is an ancestor of the giver commit, a3. It can do a fast-forward merge

### Merge two commits from different lineages that both modify the same file

$ git merge deputy

The user merges deputy into master. There is a conflict and the merge is paused.

First, Git writes the hash of the giver commit to a file at .git/MERGE\_HEAD



Adding a conflicted file tells Git that the conflict is resolved

It deletes the file at .git/MERGE\_HEAD. This completes the merge.

~/alpha $ cd ..

~ $ cp -R alpha bravo

The user copies the contents of the alpha/ repository to the bravo/ directory. This produces the following directory structure:

## Repo-to-repo collaboration: git push

It’s important to understand that Git’s idea of a “working copy” is very different from the working copy you get by checking out source code from an SVN repository. Unlike SVN, Git makes no distinction between the working copies and the central repository—they're all full-fledged [Git repositories](http://bitbucket-marketing.atlassian.com/product/code-repository).

This makes collaborating with Git fundamentally different than with SVN. Whereas SVN depends on the relationship between the central repository and the working copy, Git’s collaboration model is based on repository-to-repository interaction. Instead of checking a working copy into SVN’s central repository, you push or pull commits from one repository to another.

Of course, there’s nothing stopping you from giving certain Git repos special meaning. For example, by simply designating one Git repo as the “central” repository, it’s possible to replicate a centralized workflow using Git. This is accomplished through conventions rather than being hardwired into the VCS itself.

### Bare vs. cloned repositories

If you used git clone in the previous "Initializing a new Repository" section to set up your local repository, your repository is already configured for remote collaboration. git clone will automatically configure your repo with a remote pointed to the Git URL you cloned it from. This means that once you make changes to a file and commit them, you can git push those changes to the remote repository.

If you used git init to make a fresh repo, you'll have no remote repo to push changes to. A common pattern when initializing a new repo is to go to a hosted Git service like Bitbucket and create a repo there. The service will provide a Git URL that you can then add to your local Git repository and git push to the hosted repo. Once you have created a remote repo with your service of choice you will need to update your local repo with a mapping. We discuss this process in the Configuration & Set Up guide below.

If you prefer to host your own remote repo, you'll need to set up a "Bare Repository." Both git init and git clone accept a --bare argument. The most common use case for bare repo is to create a remote central Git repository

### Repo-to-repo collaboration

It’s important to understand that Git’s idea of a “working copy” is very different from the working copy you get by checking out code from an SVN repository. Unlike SVN, Git makes no distinction between the working copy and the central repository—they're all full-fledged Git repositories.

🡸=====

Checkout a local copy of the branch relating to the tick

* git branch -a # show a list of all available branches
* git fetch # downloads the branches
* git checkout --track -b NNNN-branch-name origin/NNNN-branch-name # NNNN should be the ticket number

(short version of above = git checkout --track origin NNNN-branch-name)

Delete the ticket branch as follows:

* git branch -d NNNN-branch-name # delete the local branch
* git push origin --delete NNNN-branch-name # delete the remote branch

Note that you can do git fetch -p origin. The -p option tells fetch to delete any tracking branches that no longer exist in the corresponding remotes; by default they are kept around.

============🡺

## 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.

# Inspecting Repositories

|  |  |
| --- | --- |
| git log | Shows a listing of commits on a branch or involving a specific file and optionally details about what changed between it and its par­ents. |
| git show | Shows information about a git object, normally used to view commit information. |
| git ls-tree | Shows a tree object, including the mode and name of each node and the SHA-1 value of the blob or tree that it points to. Can also be run recursively to see all subtrees as well. |
| git cat-file | Used to view the type of an object if you only have the SHA-1 value, or used to redirect contents of files or view raw information about any object. |
| git grep | Lets you search through your trees of content for words and phrases without having to actually check them out. |
| git diff | Generates patch files or statistics of differences between paths or files in your git repository, or your index or your working directory. |
| gitk | Graphical Tcl/Tk based interface to a local Git repository |
| git instaweb | Wrapper script to quickly run a web server with an interface into your repository and automatically directs a web browser to it. |

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 \( -e PATH -e MAX \)

# Git diff

## Comparing changes with git diff

Diffing is a function that takes two input data sets and outputs the changes between them. git diff is a multi-use Git command that when executed runs a diff function on Git data sources. These data sources can be commits, branches, files and more. This document will discuss common invocations of git diff and diffing work flow patterns. The git diff command is often used along with git status and git log to analyze the current state of a Git repo.

## Reading diffs: outputs

### Raw output format

The following examples will be executed in a simple repo. The repo is created with the commands below:

$:> mkdir diff\_test\_repo

$:> cd diff\_test\_repo

$:> touch diff\_test.txt

$:> echo "this is a git diff test example" > diff\_test.txt

$:> git init .

Initialized empty Git repository in /Users/kev/code/test/.git/

$:> git add diff\_test.txt

$:> git commit -am"add diff test file"

[master (root-commit) 6f77fc3] add diff test file

1 file changed, 1 insertion(+)

create mode 100644 diff\_test.txt

If we execute git diff at this point, there will be no output. This is expected behavior as there are no changes in the repo to diff. Once the repo is created and we've added the diff\_test.txt file, we can change the contents of the file to start experimenting with diff output.

$:> echo "this is a diff example" > diff\_test.txt

Executing this command will change the content of the diff\_test.txt file. Once modified, we can view a diff and analyze the output. Now executing git diff will produce the following output:

diff --git a/diff\_test.txt b/diff\_test.txt

index 6b0c6cf..b37e70a 100644

--- a/diff\_test.txt

+++ b/diff\_test.txt

@@ -1 +1 @@

-this is a git diff test example

+this is a diff example

Let us now examine a more detailed breakdown of the diff output.

### 1. Comparison input

diff --git a/diff\_test.txt b/diff\_test.txt

This line displays the input sources of the diff. We can see that a/diff\_test.txt and b/diff\_test.txt have been passed to the diff.

### 2. Meta data

index 6b0c6cf..b37e70a 100644

This line displays some internal Git metadata. You will most likely not need this information. The numbers in this output correspond to Git object version hash identifiers.

### 3. Markers for changes

--- a/diff\_test.txt

+++ b/diff\_test.txt

These lines are a legend that assigns symbols to each diff input source. In this case, changes from a/diff\_test.txt are marked with a --- and the changes from b/diff\_test.txt are marked with the +++ symbol.

### 4. Diff chunks

The remaining diff output is a list of diff 'chunks'. A diff only displays the sections of the file that have changes. In our current example, we only have one chunk as we are working with a simple scenario. Chunks have their own granular output semantics.

@@ -1 +1 @@

-this is a git diff test example

+this is a diff example

The first line is the chunk header. Each chunk is prepended by a header inclosed within @@ symbols. The content of the header is a summary of changes made to the file. In our simplified example, we have -1 +1 meaning line one had changes. In a more realistic diff, you would see a header like:

@@ -34,6 +34,8 @@

In this header example, 6 lines have been extracted starting from line number 34. Additionally, 8 lines have been added starting at line number 34.

The remaining content of the diff chunk displays the recent changes. Each changed line is prepended with a + or - symbol indicating which version of the diff input the changes come from. As we previously discussed, - indicates changes from the a/diff\_test.txt and + indicates changes from b/diff\_test.txt.

## Highlighting changes

### 1. git diff --color-words

git diff also has a special mode for highlighting changes with much better granularity: ‐‐color-words. This mode tokenizes added and removed lines by whitespace and then diffs those.

$:> git diff --color-words

diff --git a/diff\_test.txt b/diff\_test.txt

index 6b0c6cf..b37e70a 100644

--- a/diff\_test.txt

+++ b/diff\_test.txt

@@ -1 +1 @@

this is agit difftest example

Now the output displays only the color-coded words that have changed.

### 2. git diff-highlight

If you clone the git source, you’ll find a sub-directory called contrib. It contains a bunch of git-related tools and other interesting bits and pieces that haven’t yet been promoted to git core. One of these is a Perl script called diff-highlight. Diff-highlight pairs up matching lines of diff output and highlights sub-word fragments that have changed.

$:> git diff | /your/local/path/to/git-core/contrib/diff-highlight/diff-highlight

diff --git a/diff\_test.txt b/diff\_test.txt

index 6b0c6cf..b37e70a 100644

--- a/diff\_test.txt

+++ b/diff\_test.txt

@@ -1 +1 @@

-this is a git diff test example

+this is a diff example

Now we’ve pared down our diff to the smallest possible change.

## Diffing binary files

In addition to the text file utilities we have thus far demonstrated, git diff can be run on binary files. Unfortunately, the default output is not very helpful.

$:> git diff

Binary files a/script.pdf and b/script.pdf differ

Git does have a feature that allows you to specify a shell command to transform the content of your binary files into text prior to performing the diff. It does require a little set up though. First, you need to specify a textconv filter describing how to convert a certain type of binary to text. We're using a simple utility called pdftohtml (available via homebrew) to convert my PDFs into human readable HTML. You can set this up for a single repository by editing your .git/config file, or globally by editing ~ /.gitconfig

[diff "pdfconv"]

textconv=pdftohtml -stdout

Then all you need to do is associate one or more file patterns with our pdfconv filter. You can do this by creating a .gitattributes file in the root of your repository.

\*.pdf diff=pdfconv

Once configured, git diff will first run the binary file through the configured converter script and diff the converter output. The same technique can be applied to get useful diffs from all sorts of binary files, for example: zips, jars and other archives: using unzip -l (or similar) in place of pdf2html will show you paths that have been added or removed between commits images: exiv2 can be used to show metadata changes such as image dimensions documents: conversion tools exist for transforming .odf, .doc and other document formats to plain text. In a pinch, strings will often work for binary files where no formal converter exists.

## Comparing files: git diff file

The git diff command can be passed an explicit file path option. When a file path is passed to git diff the diff operation will be scoped to the specified file. The below examples demonstrate this usage.

git diff HEAD ./path/to/file

This example is scoped to ./path/to/file when invoked, it will compare the specific changes in the working directory, against the index, showing the changes that are not staged yet. By default git diff will execute the comparison against HEAD. Omitting HEAD in the example above git diff ./path/to/file has the same effect.

git diff --cached ./path/to/file

When git diff is invoked with the --cached option the diff will compare the staged changes with the local repository. The --cached option is synonymous with --staged.

## Comparing all changes

Invoking git diff without a file path will compare changes across the entire repository. The above, file specific examples, can be invoked without the ./path/to/file argument and have the same output results across all files in the local repo.

## Changes since last commit

By default git diff will show you any uncommitted changes since the last commit.

git diff

## Comparing files between two different commits

git diff can be passed Git refs to commits to diff. Some example refs are, HEAD, tags, and branch names. Every commit in Git has a commit ID which you can get when you execute GIT LOG. You can also pass this commit ID to git diff.

$:> git log --prety=oneline

957fbc92b123030c389bf8b4b874522bdf2db72c add feature

ce489262a1ee34340440e55a0b99ea6918e19e7a rename some classes

6b539f280d8b0ec4874671bae9c6bed80b788006 refactor some code for feature

646e7863348a427e1ed9163a9a96fa759112f102 add some copy to body

$:> git diff 957fbc92b123030c389bf8b4b874522bdf2db72c ce489262a1ee34340440e55a0b99ea6918e19e7a

## Comparing branches

### Comparing two branches

Branches are compared like all other ref inputs to git diff

git diff branch1..other-feature-branch

This example introduces the dot operator. The two dots in this example indicate the diff input is the tips of both branches. The same effect happens if the dots are omitted and a space is used between the branches. Additionally, there is a three dot operator:

git diff branch1...other-feature-branch

The three dot operator initiates the diff by changing the first input parameter branch1. It changes branch1 into a ref of the shared common ancestor commit between the two diff inputs, the shared ancestor of branch1 and other-feature-branch. The last parameter input parameter remains unchanged as the tip of other-feature-branch.

## Comparing files from two branches

To compare a specific file across branches, pass in the path of the file as the third argument to git diff

git diff master new\_branch ./diff\_test.txt

# Inspecting a repository

## git log

The git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes. While git status lets you inspect the working directory and the staging area, git log only operates on the committed history.

A diagram of different types of objects

Description automatically generated

Log output can be customized in several ways, from simply filtering commits to displaying them in a completely user-defined format. Some of the most common configurations of git log are presented below.

The git log command is Git's basic tool for exploring a repository’s history. It’s what you use when you need to find a specific version of a project or figure out what changes will be introduced by merging in a feature branch.

commit 3157ee3718e180a9476bf2e5cab8e3f1e78a73b7

Author: John Smith

Most of this is pretty straightforward; however, the first line warrants some explanation. The 40-character string after commit is an SHA-1 checksum of the commit’s contents. This serves two purposes. First, it ensures the integrity of the commit—if it was ever corrupted, the commit would generate a different checksum. Second, it serves as a unique ID for the commit.

This ID can be used in commands like git log <since>..<until> to refer to specific commits. For instance, git log 3157e..5ab91 will display everything between the commits with ID's 3157e and 5ab91. Aside from checksums, branch names (discussed in the [Branch Module](https://www.atlassian.com/git/tutorials/using-branches)) and the HEAD keyword are other common methods for referring to individual commits. HEAD always refers to the current commit, be it a branch or a specific commit.

The ~ character is useful for making relative references to the parent of a commit. For example, 3157e~1 refers to the commit before 3157e, and HEAD~3 is the great-grandparent of the current commit.

The idea behind all of these identification methods is to let you perform actions based on specific commits. The git log command is typically the starting point for these interactions, as it lets you find the commits you want to work with.

### Usage

$ git log

Display the entire commit history using the default formatting. If the output takes up more than one screen, you can use Space to scroll and q to exit.

$ git log -n <limit>

Limit the number of commits by <limit>. For example, git log -n 3 will display only 3 commits.

$ git log --oneline

Condense each commit to a single line. This is useful for getting a high-level overview of the project history.

$ git log --stat

Along with the ordinary git log information, include which files were altered and the relative number of lines that were added or deleted from each of them.

$ git log -p

Display the patch representing each commit. This shows the full diff of each commit, which is the most detailed view you can have of your project history.

$ git log --author="<pattern>"

Search for commits by a particular author. The <pattern> argument can be a plain string or a regular expression.

$ git log --grep="<pattern>"

Search for commits with a commit message that matches <pattern>, which can be a plain string or a regular expression.

$ git log <since>..<until>

Show only commits that occur between <since> and <until>. Both arguments can be either a commit ID, a branch name, HEAD, or any other kind of [revision reference](http://www.kernel.org/pub/software/scm/git/docs/gitrevisions.html).

$ git log <file>

Only display commits that include the specified file. This is an easy way to see the history of a particular file.

$ git log --graph --decorate --oneline

A few useful options to consider. The --graph flag that will draw a text based graph of the commits on the left hand side of the commit messages. --decorate adds the names of branches or tags of the commits that are shown. --oneline shows the commit information on a single line making it easier to browse through commits at-a-glance.

### Discussion

### Example

The Usage section provides many examples of git log, but keep in mind that several options can be combined into a single command:

git log --author="John Smith" -p hello.py

This will display a full diff of all the changes John Smith has made to the file hello.py.

The .. syntax is a very useful tool for comparing branches. The next example displays a brief overview of all the commits that are in some-feature that are not in master.

git log --oneline master..some-feature

# Git blame

* + The high-level function of git blame is the display of author metadata attached to specific committed lines in a file. This is used to explore the history of specific code and answer questions about what, how, and why the code was added to a repository.

The git blame command is a versatile troubleshooting utility that has extensive usage options. The high-level function of git blame is the display of author metadata attached to specific committed lines in a file. This is used to examine specific points of a file's history and get context as to who the last author was that modified the line. This is used to explore the history of specific code and answer questions about what, how, and why the code was added to a repository.

Git blame is often used with a GUI display. Online Git hosting sites like [Bitbucket](http://bitbucket-marketing.atlassian.com/product) offer blame views which are UI wrappers to git blame. These views are referenced in collaborative discussions around pull requests and commits. Additionally, most IDE's that have Git integration also have dynamic blame views.

## How It Works

In order to demonstrate git blame we need a repository with some history. We will use the open source project [git-blame-example](https://bitbucket.org/kevzettler/git-blame-example). This open source project is a simple repository that contains a README.md file which has a few commits from different authors. The first step of our git blame usage example is to git clone the example repository.

git clone https://kevzettler@bitbucket.org/kevzettler/git-blame-example.git && cd git-blame-example

Now that we have a copy of the example code we can start exploring it with git blame. The state of the example repo can be examined using [git log](https://www.atlassian.com/git/tutorials/git-log). The commit history should look like the following:

$ git log

commit 548dabed82e4e5f3734c219d5a742b1c259926b2

Author: Juni Mukherjee <jmukherjee@atlassian.com>

Date: Thu Mar 1 19:55:15 2018 +0000

Another commit to help git blame track the who, the what, and the when

commit eb06faedb1fdd159d62e4438fc8dbe9c9fe0728b

Author: Juni Mukherjee <jmukherjee@atlassian.com>

Date: Thu Mar 1 19:53:23 2018 +0000

Creating the third commit, along with Kev and Albert, so that Kev can get git blame docs.

commit 990c2b6a84464fee153253dbf02e845a4db372bb

Merge: 82496ea 89feb84

Author: Albert So <aso@atlassian.com>

Date: Thu Mar 1 05:33:01 2018 +0000

Merged in albert-so/git-blame-example/albert-so/readmemd-edited-online-with-bitbucket-1519865641474 (pull request #2)

README.md edited online with Bitbucket

commit 89feb84d885fe33d1182f2112885c2a64a4206ec

Author: Albert So <aso@atlassian.com>

Date: Thu Mar 1 00:54:03 2018 +0000

README.md edited online with Bitbucket

git blame only operates on individual files. A file-path is required for any useful output. The default execution of git blame will simply output the commands help menu. For this example, we will operate on the README.MD file. It is a common open source software practice to include a README file in the root of a git repository as documentation source for the project.

git blame README.MD

Executing the above command will give us our first sample of blame output. The following output is a subset of the full blame output of the README. Additionally, this output is static is reflective of the state of the repo at the time of this writing.

$ git blame README.md

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 1) # Git Blame example

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 2)

89feb84d (Albert So 2018-03-01 00:54:03 +0000 3) This repository is an example of a project with multiple contributors making commits.

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 4)

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 5) The repo use used elsewhere to demonstrate `git blame`

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 6)

89feb84d (Albert So 2018-03-01 00:54:03 +0000 7) Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod TEMPOR incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

89feb84d (Albert So 2018-03-01 00:54:03 +0000 8)

eb06faed (Juni Mukherjee 2018-03-01 19:53:23 +0000 9) Annotates each line in the given file with information from the revision which last modified the line. Optionally, start annotating from the given revision.

eb06faed (Juni Mukherjee 2018-03-01 19:53:23 +0000 10)

548dabed (Juni Mukherjee 2018-03-01 19:55:15 +0000 11) Creating a line to support documentation needs for git blame.

548dabed (Juni Mukherjee 2018-03-01 19:55:15 +0000 12)

548dabed (Juni Mukherjee 2018-03-01 19:55:15 +0000 13

This is a sample of the first 13 lines of the README.md file. To better understand this output lets break down a line. The following table displays the content of line 3 and the columns of the table indicate the column content.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | Author | Timestamp | Line Number | Line Content |
| 89feb84d | Albert So | 2018-03-01 00:54:03 +0000 | 3 | This repository is an example of a project with multiple contributors making commits. |

If we review the blame output list, we can make some observations. There are three authors listed. In addition to the project's maintainer Kev Zettler, Albert So, and Juni Mukherjee are also listed. Authors are generally the most valuable part of git blame output. The timestamp column is also primarily helpful. What the change was is indicated by line content column.

## Common Options

git blame -L 1,5 README.md

The -L option will restrict the output to the requested line range. Here we have restricted the output to lines 1 through 5.

git blame -e README.md

The -e option shows the authors email address instead of username.

git blame -w README.md

The -w option ignores whitespace changes. If a previous author has modified the spacing of a file by switching from tabs to spaces or adding new lines this, unfortunately, obscures the output of git blame by showing these changes.

git blame -M README.md

The -M option detects moved or copied lines within in the same file. This will report the original author of the lines instead of the last author that moved or copied the lines.

git blame -C README.md

The -C option detects lines that were moved or copied from other files. This will report the original author of the lines instead of the last author that moved or copied the lines.

## Git Blame vs Git Log

While git blame displays the last author that modified a line, often times you will want to know when a line was originally added. This can be cumbersome to achieve using git blame. It requires a combination of the -w, -C, and -M options. It can be far more convenient to use the [git log](https://www.atlassian.com/git/tutorials/git-log) command.

To list all original commits in-which a specific code piece was added or modified execute git log with the -S option. Append the -S option with the code you are looking for. Let's take one of the lines from the README output above to use as an example. Let us take the text "CSS3D and WebGL renderers" from Line 12 of the README output.

$ git log -S"CSS3D and WebGL renderers." --pretty=format:'%h %an %ad %s'

e339d3c85 Mario Schuettel Tue Oct 13 16:51:06 2015 +0200 reverted README.md to original content

509c2cc35 Daniel Tue Sep 8 13:56:14 2015 +0200 Updated README

cb20237cc Mr.doob Mon Dec 31 00:22:36 2012 +0100 Removed DOMRenderer. Now with the CSS3DRenderer it has become irrelevant.

This output shows us that content from the README was added or modified 3 times by 3 different authors. It was originally added in commit cb20237cc by Mr.doob. In this example, git log has also been prepended with the --pretty-format option. This option converts the default output format of git log into one that matches the format of git log. For more information on usage and configuration options visit the [git log](https://www.atlassian.com/git/tutorials/git-log) page.

## Summary

The git blame command is used to examine the contents of a file line by line and see when each line was last modified and who the author of the modifications was. The output format of git blame can be altered with various command line options. Online Git hosting solutions like Bitbucket offer blame views, which offer a superior user experience to command line git blame usage. git blame and git log can be used in combination to help discover the history of a file's contents. The git log command has some similar blame functionality, to learn more visit the [git log](https://www.atlassian.com/git/tutorials/git-log) overview page.

=======

## 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

**Viewing the Commit History**

The most basic and powerful tool to do this is the git log command.

One of the more helpful options is -p or --patch, which shows the difference (the *patch* output) introduced in each commit. You can also limit the number of log entries displayed, such as using -2 to show only the last two entries.

$ git log -p -2

if you want to see some abbreviated stats for each commit, you can use the --stat option

As you can see, the --stat option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end.

Another really useful option is --pretty. This option changes the log output to formats other than the default. A few prebuilt options are available for you to use. The oneline option prints each commit on a single line, which is useful if you’re looking at a lot of commits. In addition, the short, full, and fuller options show the output in roughly the same format but with less or more information, respectively:

$ git log --pretty=oneline

The most interesting option is format, which allows you to specify your own log output format. This is especially useful when you’re generating output for machine parsing — because you specify the format explicitly, you know it won’t change with updates to Git:

$ git log --pretty=format:"%h - %an, %ar : %s"

ca82a6d - Scott Chacon, 6 years ago : changed the version number

085bb3b - Scott Chacon, 6 years ago : removed unnecess

**Option Description of Output**

%H Commit hash

%h Abbreviated commit hash

%T Tree hash

%t Abbreviated tree hash

%P Parent hashes

%p Abbreviated parent hashes

%an Author name

%ae Author email

%ad Author date (format respects the --date=option)

%ar Author date, relative

%cn Committer name

%ce Committer email

%cd Committer date

%cr Committer date, relative

%s Subject

The oneline and format options are particularly useful with another log option called --graph. This option adds a nice little ASCII graph showing your branch and merge history:

*Table 2. Common options to* git log

**Option Description**

-p Show the patch introduced with each commit.

--stat Show statistics for files modified in each commit.

--shortstat Display only the changed/insertions/deletions line from the --stat command.

--name-only Show the list of files modified after the commit information.

--name-status Show the list of files affected with added/modified/deleted information as well.

--abbrev-commit Show only the first few characters of the SHA-1 checksum instead of all 40.

--relative-date Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format.

--graph Display an ASCII graph of the branch and merge history beside the log output.

--pretty Show commits in an alternate format. Options include oneline, short, full,fuller, and format (where you specify your own format).

--oneline Shorthand for --pretty=oneline --abbrev-commit used together.

**Limiting Log Output**

In addition to output-formatting options, git log takes a number of useful limiting options

* -<n>, where n is any integer to show the last n commits
* --since and --until the time-limiting options.

$ git log --since=2.weeks

This command works with lots of formats — you can specify a specific date like "2008-01-15", or a relative date such as "2 years 1 day 3 minutes ago"

* The --author option allows you to filter on a specific author,
* the --grep option lets you search for keywords in the commit messages

Another really helpful filter is the -S option (colloquially referred to as Git’s “pickaxe” option), which takes a string and shows only those commits that changed the number of occurrences of that string.

$ git log -S FindNextStructure

The last really useful option to pass to git log as a filter is a path. If you specify a directory or file name, you can limit the log output to commits that introduced a change to those files. This is always the last option and is generally preceded by double dashes (--) to separate the paths from the options.

**Option Description**

-<n> Show only the last n commits

--since, --after Limit the commits to those made after the specified date.

--until, --before Limit the commits to those made before the specified date.

--author Only show commits in which the author entry matches the specified string.

--committer Only show commits in which the committer entry matches the specified string.

--grep Only show commits with a commit message containing the string

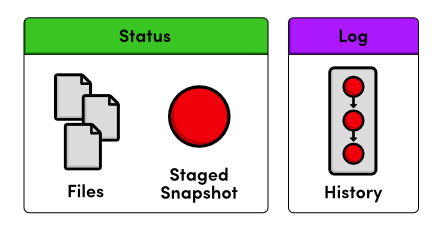
-S Only show commits adding or removing code matching the string

$ git log --pretty="%h - %s" --author='Junio C Hamano' --since="2008-10-01"

--before="2008-11-01" --no-merges -- testData/

To prevent the display of merge commits cluttering up your log history, simply add the log option --no-merges

The git status command will *only* show us *staged* changes. To view our project history (*committed* changes), we need a new command git log



Status output vs. Log output

The git log command comes with a lot of formatting options. For now, we’ll just use the convenient --oneline flag. Git outputs only the first 7 characters of the checksum. These first few characters effectively serve as a unique ID for each commit

$ git log --oneline

Condensing output to a single line is a great way to get a high-level overview of a repository. Another useful configuration is to pass a filename to git log:

$ git log --oneline file1.txt

This displays only the file1.txt history.

🡸==========

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.

Log subsets

$ git log branchA ^branchB

Show me commits reachable by branchA that are not reachable by branchB

=🡺