Contents

[1. Introduction 3](#_Toc157442160)

[2. Understanding Git Concepts 4](#_Toc157442161)

[1. What is Git? 4](#_Toc157442162)

[a) What is Version Control ? 4](#_Toc157442163)

[b) History of Git 5](#_Toc157442164)

[2. The Git Object Database 7](#_Toc157442165)

[The branch object 12](#_Toc157442166)

[The tag object 12](#_Toc157442167)

[Digging into Git internals 13](#_Toc157442168)

[Branches are movable labels 14](#_Toc157442169)

[Explore the Object Database 15](#_Toc157442170)

[The SHA 18](#_Toc157442171)

[The Objects 18](#_Toc157442172)

[Blob Object 18](#_Toc157442173)

[Tree Object 18](#_Toc157442174)

[Commit Object 19](#_Toc157442175)

[Tag Object 19](#_Toc157442176)

[Git Directory and Working Directory 19](#_Toc157442177)

[The Git Directory 19](#_Toc157442178)

[The Working Directory 20](#_Toc157442179)

[The Git Index 20](#_Toc157442180)

[How Git Stores Objects 20](#_Toc157442181)

[Browsing Git Objects 20](#_Toc157442182)

[Git References 21](#_Toc157442183)

[The Git Index 21](#_Toc157442184)

[3. The Git Promotion Model 23](#_Toc157442185)

[4 Commands in Git. 28](#_Toc157442186)

[4. Configuration 31](#_Toc157442187)

[**Initialize the Git Repository** 31](#_Toc157442188)

[**Configure Git** 31](#_Toc157442189)

[1. Maintenance 34](#_Toc157442190)

# Introduction

It is an introduction to Git that is currently the most widely used version control system in the world, mostly thanks to GitHub. The goal 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.

Working on a team, VCS provides a number of additional benefits

* The ability to resolve conflicts
* Independent 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 is the notion of a *repository instance.* With a DVCS there was no longer a central repository, everyone could develop at their own pace, store the updates locally, and put off merging conflicts until their convenience. The local nature of DVCSs also made development much faster, since you no longer had to perform actions over a network. The only time networking code gets involved is when the repository instances are being synchronized.

And, since each developer had a complete copy of the project, the risk of a server crash, a corrupted repository, or any other type of data loss was much lower than that of their CVCS predecessors.

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 Linux projects

Git is the technology behind the enormously popular “social coding” website GitHub, which includes many wellknown open source projects.

By far, Git is the most used modern version control system in the world. Git is a mature, actively maintained open source project originally developed in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel

The data model of Git is different from other common **version control systems** (**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 is a tool for tracking changes made to a set of files over time, a task traditionally known as “version control.” Although it is most often used by programmers to coordinate changes to software source code, and it is especially good at that, you can use Git to track any kind of content at all. Any body of related files evolving over time, which we’ll call a “project,” is a candidate for using Git. With Git, you can:

* Examine the state of your project at earlier points in time
* Show the differences among various states of the project
* Split the project development into multiple independent lines, called “branches,” which can evolve separately
* Periodically recombine branches in a process called “merging,”

reconciling the changes made in two or more branches

* Allow many people to work on a project simultaneously,

sharing and combining their work as needed

Having a distributed architecture, Git is an example of a DVCS (hence Distributed Version Control System). Rather than have only one single place for the full version history of the software as is common in once-popular version control systems like CVS or Subversion (also known as SVN), in Git, every developer's working copy of the code is also a repository that can contain the full history of all changes.

Git, however, is a distributed version control system. Instead of a working copy, each developer gets their own local repository, complete with a full history of commits

Unlike some version control software, Git is not fooled by the names of the files when determining what the storage and version history of the file tree should be, instead, Git focuses on the file content itself.

Git has been designed with the integrity of managed source code as a top priority. The content of the files as well as the true relationships between files and directories, versions, tags and commits, all of these objects in the Git repository are secured with a cryptographically secure hashing algorithm called SHA1. This protects the code and the change history against both accidental and malicious change and ensures that the history is fully traceable.

With Git, you can be sure you have an authentic content history of your source code.

Some other version control systems have no protections against secret alteration at a later date. This can be a serious information security vulnerability for any organization that relies on software development.

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

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

For Windows users, Git installation will install a special command shell called *Git Bash*. To test your installation, open a new command prompt and run

$ git --version.

## The Git Object Database

Git is a version control system built on top of an 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.

Git is different: Instead of the regular file and patches list, Git records a snapshot of all the files tracked by Git and their paths relative to the repository root—that is, the files tracked by Git in the filesystem tree. 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's object storage is a key-value storage, the key being the ID of the object and the value being the object itself. The key is an SHA-1 hash of the object. 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 (sort of). 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 gives us an object ID, but before we can inspect items in the object database, we need to know what type of object it is. Again, we can use the -t flag:

$ git cat-file -t 022d0352de4

tree

Of course, change the object ID to an object from your database (don’t forget to combine the folder name with the filename to get the full ID). This will output the type of commit, which we can then pass to a normal call to git cat-file.

git cat-file blob 7a52bb8

My object was a blob, but yours may be different. If it’s a tree, remember to use git ls-tree to turn that ugly binary data into a pretty directory listing.

$ git ls-tree 022d0352de4

100644 blob 1d09ca3ac33e045ccde753b47f81a9e980c90774 .gitconfig-template

100644 blob 139597f9cb07c5d48bed18984ec4747f4b4f3438 .gitignore

100644 blob 812e4df6163374ffb1ffbd1dac2cf8ec5460684e Basic-Tutorials-master.zip

040000 tree b9e065e8380804fcc424dc08128b3163ecf3ae6b Chapters

100644 blob 42ca1cef8e65effe2ad5bae228bcfc406c058f85 Links.txt

100644 blob 3aa637263b63eaaa57a2f34419374cfe7fc0701a Separating Collated Code with Branching Strategies.docx

040000 tree ba6aabea140dfdd02d0b4d84844dbe95410a78dd book-svg

100644 blob 2c6500fd68bb673ba4a5f571ced1e9cbdee31bf9 borderl.txt

100644 blob 6f6305ebd02adcfc9967d396246d5b0bc0183e37 instaLL.rtf

040000 tree e950201e1a88e43226e5d497656125936a514286 ppt

100644 blob 87701586a37307d58549e39dcdfc70bd8db51936 script.sh

100644 blob 0fe633e30c461e9e5e08d545fe06d909243955dd txt1.txt

Any file is compressed and transformed into a blob before archiving it into a Git repository. Each file is marked with a *hash*; this hash uniquely identifies the file within our repository, and it is thanks to this ID that Git can then retrieve it when needed, and detect any changes when the same file is altered (files with different content will have different hashes).

SHA-1 hashes are unique

**$ echo "banana" | git hash-object --stdin**

**637a09b86af61897fb72f26bfb874f2ae726db82**

The git hash-object command is the plumbing command to calculate the hash of any object; in this example, we used the --stdin option to pass as a command argument the result of the preceding command, echo "banana"; in a few words, we calculated the hash of the string "banana",

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

6ff87c4664981e4397625791c8ea3bbb5f2279a3

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

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

The first line contains the commit's **SHA-1** (https://en.wikipedia.org/wiki/SHA-1), an alphanumeric sequence of 40 characters representing a hexadecimal number. This *code*, or **hash**, as it is usually called, uniquely identifies the commit within the repository

Git uses four different types of **objects**, and *commit* is one of these. Then there are *tree*, *blob*, and *annotated tag*.

You can initialize a Git repository anywhere with the git init command. Take a look inside the .git folder to get a glimpse of what a repository looks like.

$ git init

Initialized empty Git repository in C:/temp/demo/.git/

$ ls -la .git

total 11

drwxr-xr-x 1 asaki 1049089 0 Jun 22 13:50 ./

drwxr-xr-x 1 asaki 1049089 0 Jun 22 13:49 ../

-rw-r--r-- 1 asaki 1049089 130 Jun 22 13:50 config

-rw-r--r-- 1 asaki 1049089 73 Jun 22 13:49 description

-rw-r--r-- 1 asaki 1049089 23 Jun 22 13:49 HEAD

drwxr-xr-x 1 asaki 1049089 0 Jun 22 13:49 hooks/

drwxr-xr-x 1 asaki 1049089 0 Jun 22 13:49 info/

drwxr-xr-x 1 asaki 1049089 0 Jun 22 13:50 objects/

drwxr-xr-x 1 asaki 1049089 0 Jun 22 13:49 refs/

***.git subdirectory***

Under the new Git repository directory, a .git subdirectory at /Users/mike/GitIn-PracticeRedux/.git/ (for example) is created with various files and directories under it.

running the find command.

$ find .git

$ find .git

.git

.git/config

.git/description

.git/HEAD

.git/hooks

.git/hooks/applypatch-msg.sample

.git/hooks/commit-msg.sample

.git/hooks/fsmonitor-watchman.sample

.git/hooks/post-update.sample

.git/hooks/pre-applypatch.sample

.git/hooks/pre-commit.sample

.git/hooks/pre-merge-commit.sample

.git/hooks/pre-push.sample

.git/hooks/pre-rebase.sample

.git/hooks/pre-receive.sample

.git/hooks/prepare-commit-msg.sample

.git/hooks/push-to-checkout.sample

.git/hooks/sendemail-validate.sample

.git/hooks/update.sample

.git/info

.git/info/exclude

.git/objects

.git/objects/info

.git/objects/pack

.git/refs

.git/refs/heads

.git/refs/tags

Git uses objects to track changes throughout the history of a repository. To achieve this tracking, Git uses four types of objects. The objects are **commits**, **trees**, **blobs,** and **tags**. These objects are stored in **.git/objects**.

There are four types of objects in Git

* Files, or blobs as they are also called in the Git context
* Directories, or trees in the Git context
* Commits
* Tags

The important directories are objects, where Git stores all of its objects; and .refs, where Git stores all of its references. .git stores:

* **Blobs**, which are the most basic data type in Git. Essentially, a blob is just a bunch of bytes; usually a binary representation of a file.
* **Tree objects**, which are a bit like directories. Tree objects can contain pointers to blobs and other tree objects. 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)
* **Commit objects**, which point to a single tree object, and contain some metadata including the commit author and any parent 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. This is what gives a repository its history: every commit has a parent commit, all the way back to the very first commit.

The author’s name and email address, and the time that the changes were authored.

The committer’s name and email address, and the time that the commit was made.

The commit message.

* **Tag objects**, which point to a single commit object, and contain some metadata.
* **References**, which are pointers to a single object (usually a commit or tag object).
* .

The file-contents reference is actually a reference to a *tree object*. A tree object stores a reference to all the *blob objects* at a particular point in time and other tree objects if there are any subdirectories. A blob object stores the contents of a particular version of a particular single file in the Git repository

Git uses objects to track changes throughout the history of a repository. To achieve this tracking, Git uses four types of objects. The objects are **commits**, **trees**, **blobs,** and **tags**. These objects are stored in **.git/objects**.

In short, the Git data model can be summarized as shown in the following diagram:





* The commit object points to the root tree. The root tree points to subtrees and files.
* Branches and tags point to a commit object and the HEAD object points to the **branch** that is currently checked out. So, for every commit, the full tree state and snapshot are identified by the root tree.

The special HEAD pointer that refers to the branch/commit currently being checked out

* The commit object:

The Git command git cat-file -p will print the object given as an input. Normally, it is not used in everyday Git commands, but it is quite useful to investigate how it ties the objects together

**$ git cat-file -p HEAD**

We can now see the commit object, consisting of the root tree (tree), the parent commit object's ID (parent), the author and timestamp information (author), the committer and timestamp information (committer), and the commit message.

* The tree object:

There are many ways to see the objects in the Git database. The git ls-tree command can easily show the content of trees and subtrees, and git show can show the Git objects, but in a different way.

We can also specify that we want the tree object from the commit pointed to by HEAD by specifying:

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

The special notation HEAD^{tree} means that from the reference given, HEAD recursively dereferences the object at the reference until a tree object is found. The first tree object is the root tree object found from the commit pointed to by the master branch, which is pointed to by HEAD.

A generic form of the notation is <rev>^<type>, and will return the first object of <type>, searching recursively from <rev>.

* The blob object:

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.

### The branch object

we can take a look at the branch inside the .git folder where the whole Git repository is stored. If we open the text file .git/refs/heads/master, we can actually see the commit ID that the master branch points to. We can do this using cat, as follows:

**$ cat .git/refs/heads/master**

**13dcada077e446d3a05ea9cdbc8ecc261a94e42d**

We can also see that HEAD is pointing to the active branch by using cat with the .git/HEAD file:

**$ cat .git/HEAD**

The branch object is simply a pointer to a commit, identified by its SHA-1 hash.

### The tag object

There are three different kinds of tag: a lightweight (just a label) tag, an annotated tag, and a signed tag. In the example repository, there are two annotated tags:

**$ git cat-file -p v1.0**

As you can see, the tag consists of an object—which, in this case, is the latest commit on the master branch—the object's type (commits, blobs, and trees can be tagged), the tag name, the tagger and timestamp, and finally the tag message.

There are many ways to see the objects in the Git database. The git ls-tree command can easily show the content of trees and subtrees, and git show can show the Git objects, but in a different way.

### Digging into Git internals

**[15] ~/grocery (master)**

**$ git cat-file -p a57d7**

**tree a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40**

this plumbing command lets you peek into the Git objects; with the -p option (which means *pretty-print* here), we ask Git to show an easier way to read what the contents of the object are.

* The **tree** is a **container** for blobs and other trees.
* Git blobs represent the files. Blobs are binary files, nothing more.

**$ git cat-file -p 637a0**

**banana**

Wow! Its content is exactly the content of our shoppingFile.txt file

**an object**, whatever it is, **will always have the same hash in any repository**, in any computer on the face of the Earth.

**Git calculates the hash on the content of the file, not in the file itself.** This teaches us an important lesson: if you have two different files with the same content, even if they have different names and paths, in Git you will end up having only one blob.

**the Git storage object model**

**$ ls –al**

**$ ls -al .git/**

**$ ls -al .git/objects**

**$ ls -al .git/objects/63**

Git is amazingly smart and simple: to be quicker while searching through the filesystem, Git creates a set of folders where the name is two characters long, and those two characters represent the first two characters of a hash code; inside those folders, Git writes all the objects using as a name the other 38 characters of the hash, regardless of the kind of Git object.

Git compresses them using the zlib library to reserve space on your disk. This is why we use the git cat-file –p command, which decompresses them on the fly for us

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.

So, every commit has a parent, and following these relations between commits, we can always navigate from a random one down to the first one, the already mentioned **root commit**

Now it's time to investigate another well-known difference between Git and other versioning systems. Take Subversion as an example: when you do a new commit, Subversion creates a new numbered revision that only contains deltas between the previous one; this is a smart way to archive changes to files, especially among big text files, because if only a line of text changes, the size of the new commit will be much smaller.

Instead, in Git even if you change only a char in a big text file, it always stores a new version of the file: **Git doesn't do deltas** (at least not in this case), and **every commit is actually a snapshot of the entire repository**.

Git repository can be imagined as a tree that, starting from a root (the root-commit), grows upward through one or more branches. These branches are generally distinguished by a name. *Master* is precisely the name of the *default branch* of a Git repository, somewhat like trunk is for Subversion

In Git, **a branch is nothing more than a label**, a *mobile label* placed on a commit

In fact, every leaf on a Git branch has to be labeled with a meaningful name to allow us to reach it and then move around, go back, merge, rebase, or discard some commits when needed.

I made a commit without first making git add; the *trick* is in the -a (--add) option added to the git commit command, which means *add to this commit all the modified files that I have already committed at least one time before*. In our case, this option allowed us to go faster and skip the git add command.

## Branches are movable labels

the commits are linked to each other by a parent-and-son relationship: each commit contains a reference to the previous commit.

So, branches are nothing but labels that are on the tip commit, the last one. This commit, our leaf, must always be identified by a label

Every time we make a commit to a branch, the **reference** that identifies that branch will move accordingly to always stay associated with the tip commit

**$ ls -al .git/**

**$ ls -al .git/refs**

**$ ls -al .git/refs/heads**

**$ cat .git/refs/heads/master**

**0e8b5cf1c1b44110dd36dea5ce0ae29ce22ad4b8**

Git manages all this articulated reference system... with a trivial text file! It contains the hash of the last commit made on the branch

As branches are, HEAD is a **reference**. It represents a pointer to the place on where we are right now, nothing more, nothing less. In practice instead, it is just another plain text file:

**$ cat .git/HEAD**

**ref: refs/heads/berries**

The difference between the HEAD file and branches text file is that the HEAD file usually refers to a branch, and not directly to a commit as branches do. The ref: part is the convention Git uses internally to declare a pointer to another branch, while refs/heads/berries is of course the relative path to the berries branch text file

*In Subversion, we usually have different folders for each different branch.* When you switch a branch, Git goes to the commit the branch is pointing to, and following the parent relationship and analyzing trees and blobs, rebuilds the content on the **working directory** accordingly, getting hold of that files and folders

**$ git checkout -**

**Switched to branch 'berries**

New trick: using the dash (-), you actually are saying to Git: "*Move me to the branch I was before switching*"; and Git obeys, moving us to the berries branch

🡸====================================🡺

## Explore the Object Database

**git** is fundamentally a key-value store. When you add data to **git**, it builds an object and uses the SHA-1 hash of the object's contents as a key.

A **git** repository is an on-disk data structure which stores metadata for a set of files and directories. It lives in your project's .git**/** folder. Every time you commit data to git, it gets stored here.

It's basic structure is like this:

.git**/**

objects**/**

refs**/**

Therefore, any content in **git** can be looked up by it's hash:

**git cat-file** -p 4bb6f98

There are 4 types of Object:

* Blob
* **Tree**
* Commit
* Tag

HEAD is a special ref. It always points to the current object. You can see where it's currently pointing by checking the .git**/**HEAD file. Normally, HEAD points to another ref:

$cat .git**/**HEAD

ref: refs**/**heads**/**mainline

This is what's known as a "detached head" - because HEAD is not attached to (pointing at) any ref, but rather points directly to an object.

A ref is essentially a pointer. It's a name that points to an object. For example,

"master" --**>** 1a410e...

They are stored in `.git/refs/heads/ in plain text files.

$ **cat** .git**/**refs**/**heads**/**mainline

4bb6f98a223abc9345a0cef9200562333

Now, it's possible to navigate **git** purely by jumping around to different objects directly by their hashes. But this would be terribly inconvenient. A ref gives you a convenient name to refer to objects by. It's much easier to ask **git** to go to a specific place by name rather than by hash.

A commit is probably the object type most familiar to **git** users, as it's what they are used to creating with the **git commit** commands.

However, the commit does not directly contain any changed files or data. Rather, it contains mostly metadata and pointers to other objects which contain the actual contents of the commit.

A commit contains a few things:

* hash of a **tree**
* hash of a parent commit
* author name/email, commiter name/email
* commit message

You can see the contents of any commit like this:

$ **git cat-file** commit 5bac93

**Tree**

A very important note is that the **tree** objects stores EVERY file in your project, and it stores whole files not diffs. This means that each commit contains a snapshot of the entire project\*.

\**Technically, only changed files are stored. But this is more an implementation detail for efficiency. From a design perspective, a commit should be considered as containing a complete copy of the project*.

**Parent**

The parent line contains a hash of another commit object, and can be thought of as a "parent pointer" that points to the "previous commit". This implicitly forms a graph of commits known as the **commit graph**. Specifically, it's a directed acyclic graph (or DAG).

A **tree** basically represents a folder in a traditional filesystem: nested containers for files or other folders.

A **tree** contains:

* 0 or more blob objects
* 0 or more **tree** objects

A blob contains arbitrary binary file contents. Commonly, it will be raw text such as source code or a blog article. But it could just as easily be the bytes of a PNG file or anything else.

If you have the hash of a blob, you can look at it's contents.

The **git commit** command does a few things:

1. Create blobs and trees to represent your project directory - stored in .git**/**objects

2. Creates a new commit object with your author information, commit message, and the root **tree** from step 1 - also stored in .git**/**objects

3. Updates the HEAD ref in .git**/**HEAD to the hash of the newly-created commit

This results in a new snapshot of your project being added to **git** that is connected to the previous state.

>ls -al

A commit object stores the hash of the directory tree object that the commit corresponds to, the parent commit hash, the author, the committer date and time, and the commit message:

**git cat-file -t 11b8b15**

**git cat-file -p 11b8b15**

**BLOBs**

Git uses blobs to store the contents of a file at a given point in time. A blob is a **Binary Large OBject (BLOB)**. It's Git's methodology of storing the contents of a file at a given point in its lifetime. A blob is created when we commence the tracking of a file by using the **git add** command:

### The SHA

### The Objects

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

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

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

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

### Blob Object

A blob generally stores the contents of a file.

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

### Tree Object

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

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

### Commit Object

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

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

A commit is defined by:

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

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

### Tag Object

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

$ git cat-file tag v1.5.0

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

## Git Directory and Working Directory

### The Git Directory

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

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

$>tree -L 1

.

|-- HEAD # pointer to your current branch

|-- config # your configuration preferences

|-- description # description of your project

|-- hooks/ # pre/post action hooks

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

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

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

`-- refs/ # pointers to your branches

### The Working Directory

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

## The Git Index

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

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

## How Git Stores Objects

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

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

.git/objects/ab/04d884140f7b0cf8bbf86d6883869f16a46f65

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

## Browsing Git Objects

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

$ git cat-file -t 54196cc2

$ git cat-file -s 54196cc2

$ git cat-file -p 54196cc2

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

$ git ls-tree 92b8b694

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

$ find .git/objects/

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

$ cat .git/HEAD

ref: refs/heads/master

$ cat .git/refs/heads/master

c4d59f390b9cfd4318117afde11d601c1085f241

$ git cat-file -t c4d59f39

commit

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

## Git References

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

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

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

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

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

$ git show-ref --heads

$ git show-ref --tags

## The Git Index

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

$ git ls-files --stage

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

$ git hash-objec

## The Git Promotion Model



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.

The combination of the working directory, staging area, and local repository make up your local environment. These are the parts of the Git system that exist on your local machine—actually, within a special subdirectory of the root (top-level) directory of your working directory. 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 remote repository is a separate Git repository intended to collect and host content pushed to it from one or more local repositories. Like the Public level in the dev-test-prod model, its main purpose is to be a place to share and access content from multiple users. There are various forms of hosting and protocols

The Working Directory

Any directory or directory tree on your local system can be a working directory for a Git repository. A working directory can have any number of subdirectories that form an overall *workspace*. (You might also hear this referred to by similar names such as “working tree” or “worktree.” In a tree structure, the higher-level directory where you initiated work with Git becomes the top level or root of your workspace. All subdirectories are considered part of the working directory’s scope, unless Git is specifically told to ignore them via a .gitignore file (discussed in Chapter 10) or they are part of a Git *submodule* (discussed in Chapter 14).

When you connect Git to a local directory tree, by default Git creates a repository skeleton in a special subdirectory at the top level of the tree. That repository skeleton is the local repository. The physical subdirectory is named *.git* by default. This is a similar convention that many open source projects use, storing metadata in a directory starting with a period (.) followed by the name of the tool or application.

When developing code, a workspace should most likely consist of the structure needed to create a single deliverable—a JAR file or DLL, and so on. For other kinds of content, consider what makes sense as a logical unit that can be managed separately and maintained by a small number of users to reduce the occurrence of merge conflicts

The Staging Area

The staging area is one of the concepts in Git that many new users have difficulty understanding and appreciating. 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.

What’s the Point of the Staging Area?

As its name implies, the staging area provides a place to *stage* changes before they are committed (promoted) into the local repository. The staging area can hold any set of content that has been promoted from the working directory and is a candidate for going into the local repository—from a single file to all of the eligible files. The staging area provides a place to collect or assemble individual changes into the set of things that will be committed. It allows finer-grained control over the set of things that make up a change. Now let’s look at the common use cases for it.

However, in a case where there are merge conflicts that Git cannot automatically resolve, Git puts those files in your working directory for you to fix, and stages any files that merged cleanly. What it is doing is starting to create a set of merged content to be committed once everything is resolved.

There is another side benefit of this arrangement. After the merge has been attempted, if there are conflicts, the merged files are grouped together in the staging area.

MERGING AND THE STAGING AREA

One other area where the staging operation is required is when you need to complete a merge operation that had conflicts. As discussed in the previous section, 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.

As mentioned earlier, this repository is physically stored inside a separate (normally hidden) subdirectory normally within the root of the working directory. It is created in one of two ways: via a clone (copy) of a repository from a remote, or through telling Git to initialize a new environment locally.

Local Repository to Working Directory

The *checkout* command is used to retrieve content (as flat files) from the local repository into the

working directory. This is usually done by supplying a branch name and telling Git to get the latest

copy of content from that branch. Checkout also tells Git to switch the branch that you are currently

working with.

Remote Repository to Local Environment

When moving content from the remote repository to the local environment, there are several ways the local repository and the working directory can receive content from the remote repository.

The *clone* command is used to create a new local environment from an existing remote repository. Essentially, it makes a local copy of the specified remote repository onto the local disk and checks out a flat copy of the files from a branch (typically master, although this is configurable) into the working directory.

The *fetch* command is used to update the local repository from the remote repository. More specifically, it is updating reference copies of the remote branches (*reference branches*) that are maintained in the local repository.

 

The following diagram describes the tree stages and the commands used to move between the stages:



The Git model provides a local environment where you can work with a local copy of a server-side repository (this server-side repository is known as the *remote* in Git terminology). This copy resides within your workspace.

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 is where the developer edit files and compile code.
* 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. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.

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. This happens every time a file is added, but if nothing changes for a file, it means that it is already stored in the database.

A Git *repository* is the local collection of all the files related to a particular Git version control system and contains a .git subdirectory in its root. Git keeps track of the state of the files in the repository’s directory on disk.

Git repositories store all their data on your local machine. Making commits, viewing history, and requesting differences between commits are all local operations that don’t require a network connection. This makes all these operations much faster in Git than with centralized version control systems such as Subversion.

Under the new Git repository directory, a .git subdirectory is created with various files and directories under it.

The key difference here is that, in a DVCS such as Git, 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.

🡸==============================================================🡺

A **repository** is a container for your entire project; every file or subfolder within it belongs to that repository, in a consistent manner. Physically, a repository is nothing other than a folder that contains a special .git folder, the folder where the magic happens

The result of the git init command is the creation of a .git folder, where Git stores all the files it needs to manage our repository

**$ git add .**

With this trick (the dot after the git add command), you can add all the new or modified files in one shot.

So, we can move this grocery folder wherever we want, and no data will be lost. Another important thing to highlight is that we don't need any server: we can create a repository locally and work with it whenever we want, even with no LAN or internet Connection

==================

## 4 Commands in Git.

Git has a myriad of commands, some of which are practically never used by the average user; as by example, the previous git cat-file. These commands are called *plumbing commands*, while others, such as git add, git commit, are among the so-called *porcelain commands*.

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 if you know how to reference those elements.

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>

|  |  |  |
| --- | --- | --- |
|  | 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 |

The primary reason to specify both commit references and paths would be to select certain paths that are part of the tree associated with the commit. Because Git operates at the granularity of a tree, you may not always want to do the operation against all items in the tree. To indicate that the operation should only be done against certain files or paths in the scope of the snapshot, you need to add specific filenames or paths.

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.

|  |  |
| --- | --- |
| 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. |
| 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 |
| log | Show commit logs. |
| merge | Join two or more development histories together |
| mv | Move or rename a file, directory, or symlink. |
| pull | Fetch from, or integrate with, another repository or a local branch |
| 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. |
| 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. |
| 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 |

Porcelain Commands in Git

Table 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. |

Plumbing commands

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. When arguments are combined in this way, the ordering is important. If the first argument requires a value, then the second argument may be taken as the required value instead of an additional argument.

# Configuration

**Initialize the Git Repository**

$ mkdir myrepo

$ cd myrepo

$ git init

Git is designed to be as unobtrusive as possible. Notice that there is now a .git directory in myrepo that stores all the tracking data for our repository. 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

git init can be run without any arguments to create the local Git repository in the current directory. Under the myrepo repository directory, a .git 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

**Configure Git**

Git comes with a long list of configurations options from your name to your favorite merge tool. You can set options with the git config or by manually editing a file called .gitconfig in your home directory.

The first thing you do with any Git installation is set you name an email.

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

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

Once you’ve installed Git, the first thing you need to do is to tell Git your name and email (particularly before creating any commits). Rather than usernames, Git uses a name and an email address to identify the author of a commit.

We can do this with the git config command:

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

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

The --global option tells Git to use this configuration as a default for all of your repositories. Omitting it lets you specify different user information for individual repositories, which will come in handy later on.

First off, Mary needs to configure her repository so that we know who contributed what to the project.

git config user.name "Mary"

git config user.email mary.example@rypress.com

You may recall from the first module that we used a --global flag to set the configuration for the entire Git installation. But since Mary’s repository is on the local filesystem, she needs a local configuration.

Use a text editor to open up the file called config in the .git folder of Mary’s project (you may need to enable hidden files to see .git). This is where local configurations are stored, and we see Mary’s information at the bottom of the file. Note that this overrides the global configuration that we set in [The Basics](http://rypress.com/tutorials/git/the-basics.html).

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

$ git config --global core.editor notepad

Git supports aliasing commands

$ git config --global alias.st status

To tell Git to ignore certain files (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

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.

You can configure Git to use your own preferred editor, but if you don't do it, this is what you have to deal with. Vim is powerful, but for newcomers, it can be a pain to use. It has a strange way of dealing with text. To start typing, you have to press *I* for inserting text, as shown in the following

Once you have typed your commit message, you can press *Esc* to get out of editing mode. Then, you can type the :w command to write changes and the :q command to quit. You can also type the command in pairs as :wq,

So, I prefer setting up usernames and emails per repository; in Git, you can set up your config variables at three levels: *repository* (with the --local option, the default one), *user* (with the --global option), and *system-wide* (with the --system option).

**$ git config user.name "Ferdinando Santacroce"**

**$ git config user.email** [**ferdinando.santacroce@gmail.com**](mailto:ferdinando.santacroce@gmail.com)

**$ git log --format=fuller**

**Other than the author, a commit preserves even the committer, and the committing date;**

**99% of commits in your repository will have the same values for the author and committer, and the same dates.** **In some situations, such as the cherry-pick, you carry an existing commit on top of another branch, making a brand-new commit that applies the same changes of the**

**previous. In this case, the author and author date will remain the same, while the**

**committer and the committing date will be related to the person who performed this**

**operation and the date they did it.**

**Using the git log command again, we can enable x-ray vision using the --format=raw option:**

**$ git log --format=raw**

**$ git cat-file -p a57d7**

**the output is the same of git log --format=raw**

# Maintenance

We can use the git fsck command to check for dangling objects—that is, objects that are not referred to by other objects or references:

$ git fsck --dangling

=====