Contents

[1. Introduction 3](#_Toc158044336)

[2. Understanding Git Concepts 4](#_Toc158044337)

[1. What is Git? 4](#_Toc158044338)

[2. The Git Object Database. 5](#_Toc158044339)

[a. Git Objects 5](#_Toc158044340)

[b. Git references 7](#_Toc158044341)

[3. The Git promotion model 9](#_Toc158044342)

[a. The working directory 10](#_Toc158044343)

[b. The staging area 10](#_Toc158044344)

[4. The local repository 11](#_Toc158044345)

[The Working Directory sandbox, scratch area 14](#_Toc158044346)

[5. Exploring the Object Database 16](#_Toc158044347)

# 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.

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 gets their own local repository and 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 projects. Git is the technology behind the enormously popular “social coding” website GitHub,.

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.

## The Git Object Database.

### Git Objects

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 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 SHA1 hash is a cryptographic hash function. The key **SHA-1** ([https://en.wikipedia.org/wiki/SHA-1)](https://en.wikipedia.org/wiki/SHA-1)s) is an alphanumeric sequence of 40 characters representing a hexadecimal number.

This **hash**, as it is usually called, uniquely identifies the commit within the repository. What that means to us is that it is virtually impossible to find two different objects with the same name.

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 (BLOB)**. 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.
* T**rees**. 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. This is what gives a repository its history: every commit has a parent commit, all the way back to the very first commit. This implicitly forms a graph of commits known as the commit graph. Specifically, it's a directed acyclic graph (or DAG).
    - 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

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. 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:



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 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 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.

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.

**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.

This is why we use the git cat-file –p command, which decompresses them on the fly for us

### Git references

Another important directory are is .refs, where Git stores all of its references. .git stores:

* **References**, which are pointers to a single object (usually a commit or tag object).
* 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.

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

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

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

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.

The HEAD in Git is the pointer to the current branch reference, which is in turn a pointer to the last commit you made or the last commit that was checked out into your working directory. That also means it will be the parent of the next commit you do. It's generally simplest to think of it as **HEAD is the snapshot of your last commit**.

**$ ls –al**

**$ ls -al .git/**

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

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

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

git cat-file -p 4bb6f98

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

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

$ cat .git/HEAD

ref: refs/heads/master

$ cat .git/refs/heads/master

e9a570524b63d2a2b3a7c3325acf5b89bbeb131e

$ git cat-file -p e9a570524b63d2a2b3a7c3325acf5b89bbeb131e

tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf

author Scott Chacon 1301511835 -0700

committer Scott Chacon 1301511835 -0700

initial commit

$ git ls-tree -r cfda3bf379e4f8dba8717dee55aab78aef7f4daf

100644 blob a906cb2a4a904a152... README

100644 blob 8f94139338f9404f2... Rakefile

040000 tree 99f1a6d12cb4b6f19... lib

## The Git promotion model

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 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 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*.

This is where the content of files are placed into actual files on your filesystem so they're easily edited by you. **The Working Directory is your scratch space, used to easily modify file content.**

When you checkout a branch, it changes **HEAD** to point to the new commit, populates your **Index** with the snapshot of that commit, then checks out the contents of the files in your **Index** into your **Working Directory**.

### 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. 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.

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.

It's simplest to think of it as the Index is the snapshot of your next commit.

Git populates it with a list of all the file contents that were last checked out into your working directory and what they looked like when they were originally checked out

$ git ls-files -s

100644 a906cb2a4a904a152e80877d4088654daad0c859 0 README

100644 8f94139338f9404f26296befa88755fc2598c289 0 Rakefile

100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0 lib/simplegit.rb

Again, here we’re using git ls-files, which is more of a behind the scenes command that shows you what your index currently looks like

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.

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.

### 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 (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:

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

MERGING AND THE STAGING AREA

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.

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.

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

### The Working Directory sandbox, scratch area

The other two trees store their content in an efficient but inconvenient manner, inside the .git folder. The working directory unpacks them into actual files, which makes it much easier for you to edit them. Think of the working directory as a **sandbox**, where you can try changes out before committing them to your staging area (index) and then to history.

\*\*\*\*\*

* **repository** — A **repository** is a collection of commits, each of which is an archive of what the project’s working tree looked like at a past date, whether on your machine or someone else’s. It also defines HEAD (see below), which identifies the branch or commit the current working tree stemmed from. Lastly, it contains a set of branches and tags, to identify certain commits by name
* **the index** — Unlike other, similar tools you may have used, Git does not commit changes directly from the working tree into the repository. Instead, changes are first registered in something called **the index**. Think of it as a way of “confirming” your changes, one by one, before doing a commit (which records all your approved changes at once). Some find it helpful to call it instead as the “staging area”, instead of the index.

 **working tree** — A **working tree** is any directory on your filesystem which has a *repository* associated with it (typically indicated by the presence of a sub-directory within it named .git.). It includes all the files and sub-directories in that directory.

 **commit** — A **commit** is a snapshot of your working tree at some point in time. The state of HEAD (see below) at the time your commit is made becomes that commit’s parent. This is what creates the notion of a “revision history”.

 **branch** — A **branch** is just a name for a commit (and much more will be said about commits in a moment), also called a reference. It’s the parentage of a commit which defines its history, and thus the typical notion of a “branch of development”.

 **tag** — A **tag** is also a name for a commit, similar to a *branch*, except that it always names the same commit, and can have its own description text.

 **master** — The mainline of development in most repositories is done on a branch called “\*\*master\*\*”. Although this is a typical default, it is in no way special.

 **HEAD** — **HEAD** is used by your repository to define what is currently checked out:

* If you checkout a branch, HEAD symbolically refers to that branch, indicating that the branch name should be updated after the next commit operation.
* If you checkout a specific commit, HEAD refers to that commit only. This is referred to as a detached *HEAD*, and occurs, for example, if you check out a tag name.

## Exploring the Object Database

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.

**[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.

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

**banana**

You can see the contents of any commit like this:

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

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

$ 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

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

$ git show-ref --heads

$ git show-ref --tags

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

$ git hash-object

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

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 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.

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.

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/"

$ ls -al

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

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

**$ 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

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

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

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:

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.

**$ 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

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",

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/

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

* 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:

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.