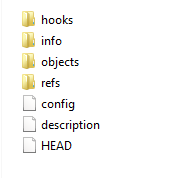
# Git Repository (.git folder structure)

First do a ***git init*** from any of the directory.



* The description file is only used by the GitWeb program, so don’t worry about it.
* The configfile contains your project-specific configuration options
* The info directory keeps a global exclude file for ignored patterns that you don’t want to track in a .gitignore file.
* The hooks directory contains your client- or server-side hook scripts

This leaves four important entries: the HEAD and (yet to be created) index files, and the objects and refs directories. These are the core parts of Git.

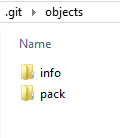
* The objects directory stores all the content for your database
* The refs directory stores pointers into commit objects in that data (branches)
* The HEAD file points to the branch you currently have checked out
* The index file is where Git stores your staging area information.

## Git Object

Git is a content-addressable filesystem, which means core of Git is a simple key-value data store. You can insert any kind of content into it, and it will give you back a key that you can use to retrieve the content again at any time.

For Eg:

Currently the object directory contains an empty info and pack directory

.

We can use the plumbing command hash-object, which takes some data, stores it in your .git directory, and gives you back the key the data is stored as.

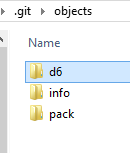
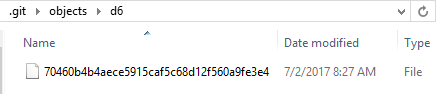


Here –w tells hash-object to store the object; otherwise, the command simply tells you what the key would be.

--stdin : to read from stdin, otherwise command will check for a file path at the end.

Here we can see it return a 40-character checksum hash. This is the **SHA-1 hash** – a checksum of the content you’re storing plus a header.

Now you can see how Git has stored your data:

Here it creates a new directory d6, which are the header and first two character of SHA-1Hash. Remaining 38 characters is used to create the file inside it.



You can pull the content back out of Git with the cat-file command

Passing -p to it instructs the cat-file command to figure out the type of content and display it nicely



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

$ echo 'version 1' > test.txt

$ git hash-object -w test.txt

83baae61804e65cc73a7201a7252750c76066a30

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

$ echo 'version 2' > test.txt

$ git hash-object -w test.txt

1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

Your database contains the two new versions of the file as well as the first content you stored there:

$ find .git/objects -type f

.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a

.git/objects/83/baae61804e65cc73a7201a7252750c76066a30

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4

Now you can revert the file back to the first version

$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt

$ cat test.txt

version 1

or the second version:

$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt

$ cat test.txt

version 2

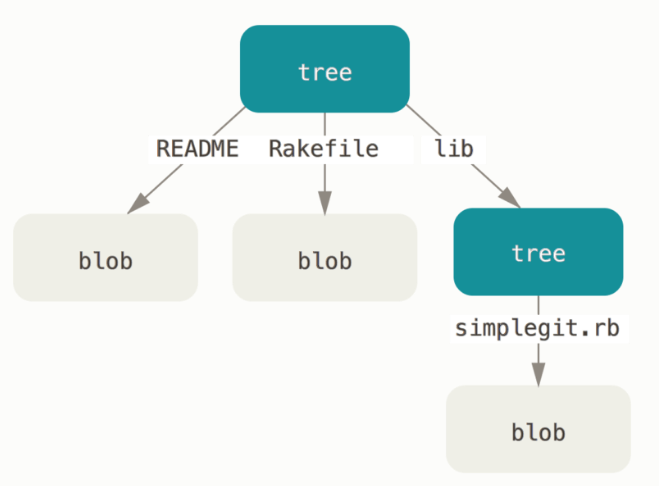
But remembering the SHA-1 key for each version of your file isn’t practical; plus, you aren’t storing the filename in your system – just the content. This object type is called a blob(binary large object)

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

$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

blob

### Tree Object



solves the problem of storing the filename and also allows you to store a group of files together of type tree, what ever content we staged it will store as tree. Which will contain blob.

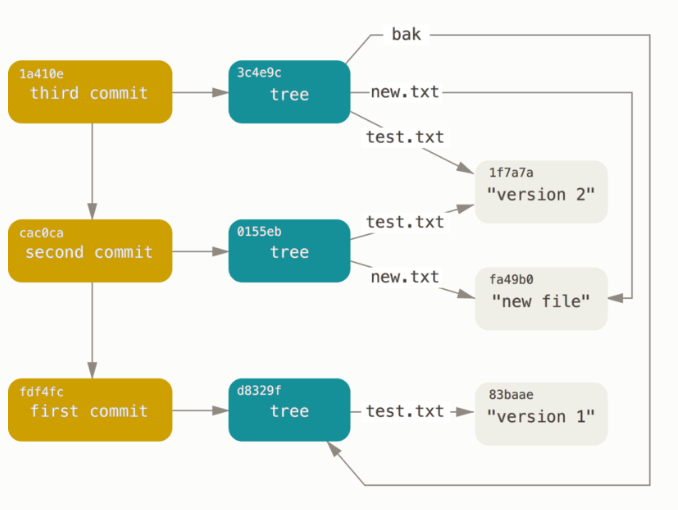
### Commit object

problem like: you must remember all SHA-1 values in order to recall the snapshots. You also don’t have any information about who saved the snapshots, when they were saved, or why they were saved. This is the basic information that the commit object stores for you.

There are 3 different object inside : **Blob, tree, commit**

Each will be store separately inside the object directory in .git

 it stores blobs for the files that have changed, updates the index, writes out trees, and writes commit objects that reference the top-level trees and the commits that came immediately before them



### Object Storage

You’ll see how to store a blob object – in this case, the string “what is up, doc?”

Git constructs a header that starts with the type of the object, in this case a blob. Then, it adds a space followed by the size of the content and finally a null byte:

>> header = "blob #{content.length}\0"

=> "blob 16\u0000"

Git concatenates the header and the original content and then calculates the SHA-1 checksum of that new content.

=> "blob 16\u0000what is up, doc?"

SHA-1 algorithm

=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"

Git compresses the new content with zlib

Finally, you’ll write your zlib-deflated content to an object on disk. You’ll determine the path of the object you want to write out (the first two characters of the SHA-1 value being the subdirectory name, and the last 38 characters being the filename within that directory.

That’s it – you’ve created a valid Git blob object. All Git objects are stored the same way, just with different types – instead of the string blob, the header will begin with commit or tree. Also, although the blob content can be nearly anything, the commit and tree content are very specifically formatted.

## Git Reference

You can run something like git log 1a410e to look through your whole history, but you still have to remember that 1a410e is the last commit in order to walk that history to find all those objects. You need a file in which you can store the SHA-1 value under a simple name so you can use that pointer rather than the raw SHA-1 value

Ref directory contain heads and tags sub directory

In that heads will have all branch name files and the commit object which it is pointing to.

In the tags will have different tags files which will point to the commit object which it is tagged at.

### The HEAD

The HEAD file is a symbolic reference to the branch you’re currently on.

### The Tags

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

### The remote

If you add a remote and push to it, Git stores the value you last pushed to that remote for each branch in the refs/remotes directory. For instance, you can add a remote called origin and push your master branch to it.

Then, you can see what the master branch on the origin remote was the last time you communicated with the server, by checking the refs/remotes/origin/master file

## Packfile

If you are having a large file and git store it. If you made any change it will a whole new copy of the file with the large size itself.

So what git does is it will pack the file, then it will take the delta add store it in that pack. It will store the current version and keep the original as delta.

This pack will be there in object folder. Git automatically does this, we can manually do this by ***git gc***