Git Internals - Plumbing and Porcelain

Using a bathroom metaphor, we daily use the porcelain items (sink and toilet). However, we rarely are in direct use of the plumbing of the bathroom. The porcelain is the layer between us and the plumbing. In the same manner, the commands that we will use below are generally referred to as Git’s “plumbing” commands, while the more user-friendly commands are called “porcelain” commands.

# Git Objects

## Prep

First, you initialize a new Git repository and verify that there is (predictably) nothing in the objects directory:

$ git init test

Initialized empty Git repository in /home/mark/test/.git/

$ cd test

$ find .git/objects

.git/objects

.git/objects/info

.git/objects/pack

$ find .git/objects -type f

Note that the last command is looking for file objects and did not find any

In a vertically split terminal pane, to show the .git directory updates, let's do the following:

$ cd test

$ while :

do

clear

tree .git

done

or do the following in the vertically split pane to show the .git directory updates:

$ cd test

$ watch -d 'ls -l | fgrep devisers'

I like the former in that the color display, at least in my terminal with the .bashrc prompt settings, is more attractive.

Back in the left terminal pane, let's clean up the hooks directory (Git populates the hooks directory with a bunch of example scripts) noise as we will not be using them:

$ rm -rf .git/hooks/

## blob object

Git has initialized the objects directory and created pack and info subdirectories in it, but there are no regular files. Now, let’s use git hash-object to create a new data object and manually store it in your new Git database:

$ echo "Once more unto the breach dear friends, once more." | git hash-object -w --stdin

24941f46f1b778a70a22424eb4c26fbefe629ea8

git hash-object takes the content you handed to it and merely return the unique key (40-character checksum hash. This is the SHA-1 hash--a checksum of the content you’re storing plus a header) that would be used to store it in your Git database. The -w option then tells the command to not simply return the key, but to write that object to the database. Finally, the --stdin option tells git hash-object to get the content to be processed from stdin; otherwise, the command would expect a filename argument at the end of the command containing the content to be used.

Let's look into the object with the git cat-file command:

$ git cat-file -p 24941f46f1b778a70a22424eb4c26fbefe629ea8

Once more unto the breach dear friends, once more.

Let's look at the type of git object for our new object:

$ git cat-file -t 24941f46f1b778a70a22424eb4c26fbefe629ea8

blob

Let's see if there are file objects:

$ find .git/objects -type f

.git/objects/24/941f46f1b778a70a22424eb4c26fbefe629ea8

Ahh, so the blob and file types are at least analogous.

Let's make a few more blob objects and give them the same file name:

$ echo 'version 1' > test.txt

$ git hash-object -w test.txt

83baae61804e65cc73a7201a7252750c76066a30

$ echo 'version 2' > test.txt

$ git hash-object -w test.txt

1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

So we see that while the file name is the same, the blob objects are different due the content.

$ find .git/objects -type f

.git/objects/24/941f46f1b778a70a22424eb4c26fbefe629ea8

.git/objects/83/baae61804e65cc73a7201a7252750c76066a30

.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a

$ git cat-file -p 83baa

version 1

Note that I only used the first 5 characters of the object's ID (40-character checksum hash. This is the SHA-1 hash--a checksum of the content you’re storing plus a header) to uniquely identify the blob.

Looking at the other text.txt blob:

$ git cat-file -p 1f7a7

version 2

While this may be interesting, the SHA-1 key for each version of your file isn’t practical. Do you want to memorize these IDs?

Now, let's get a look at the status of git in this repo:

$ git status

On branch master

No commits yet

Untracked files:

(use "git add <file>..." to include in what will be committed)

test.txt

nothing added to commit but untracked files present (use "git add" to track)

Now, let's look at what will hold blob objects--the tree object.

## tree object

Git normally creates a tree by taking the state of your staging area or index and writing a series of tree objects from it. So, to create a tree object, you first have to set up an index by staging some files. To create an index with a single entry — the first version of your test.txt file — you can use the plumbing command git update-index. You use this command to artificially add the earlier version of the test.txt file to a new staging area. You must pass it the --add option because the file doesn’t yet exist in your staging area (you don’t even have a staging area set up yet) and --cacheinfo because the file you’re adding isn’t in your directory but is in your database. Then, you specify the mode, SHA-1, and filename:

$ git update-index --add --cacheinfo 100644 83baae61804e65cc73a7201a7252750c76066a30 test.txt

In this case, you’re specifying a mode of 100644, which means it’s a normal file. Other options are 100755, which means it’s an executable file; and 120000, which specifies a symbolic link.

Now, we can look in the staging area (index) and see the blob there:

$ git ls-files --cached

test.txt

OK, so we have a blob in staging. Now, you can use git write-tree to write the staging area out to a tree object. No -w option is needed — calling this command automatically creates a tree object from the state of the index if that tree doesn’t yet exist:

$ git write-tree

d8329fc1cc938780ffdd9f94e0d364e0ea74f579

Let's look into the tree object:

$ git cat-file -p d8329

100644 blob 83baae61804e65cc73a7201a7252750c76066a30 test.txt

and its type:

$ git cat-file -t d8329

tree

We’ll now create a new tree with the second version of test.txt and a new file as well:

$ git update-index --add --cacheinfo 100644 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt

Let's add a new file as well:

$ echo 'new file' > new.txt

$ git update-index --add new.txt

The staging area now has the new version of test.txt as well as the new file new.txt. Write out that tree (recording the state of the staging area or index to a tree object) and see what it looks like:

$ git write-tree

0155eb4229851634a0f03eb265b69f5a2d56f341

Looking into the new tree:

$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341

100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt

100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt

Notice that this tree has both file entries and also that the test.txt SHA-1 is the “version 2” SHA-1 from earlier (1f7a7a). Just for fun, you’ll add the first tree as a subdirectory into this one. You can read trees into your staging area by calling git read-tree. In this case, you can read an existing tree into your staging area as a subtree by using the --prefix option with this command:

$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579

Note that d8329fc1cc938780ffdd9f94e0d364e0ea74f579 is the SHA-1 ID from the tree we created earlier.

Looking at staging now, we see an additional item, bak/test.txt:

$ git ls-files --cached

bak/test.txt

new.txt

test.txt

Let's add the bak tree:

$ git write-tree

3c4e9cd789d88d8d89c1073707c3585e41b0e614

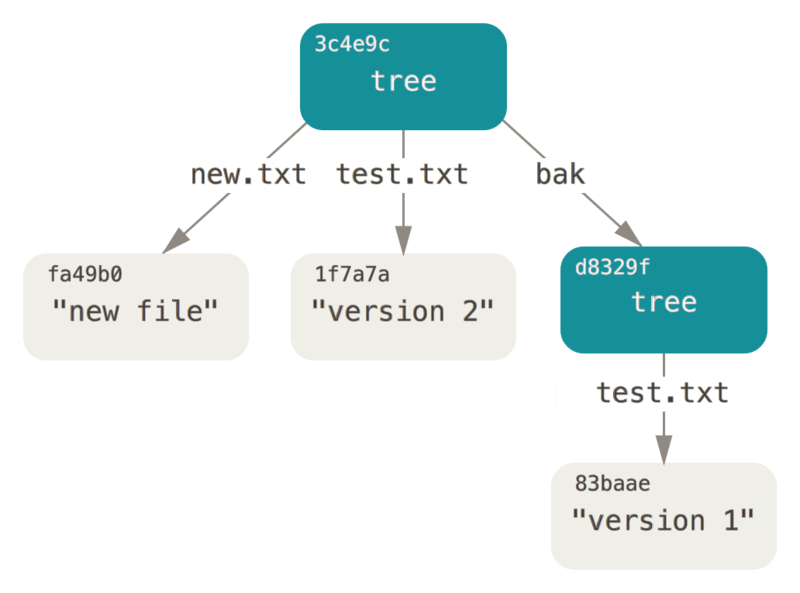
and let’s look into it:  
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614

040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579 bak

100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt

100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt

You can think of the data that the git contains for these structures as being like this:



You now have three trees (d8329fc1cc938780ffdd9f94e0d364e0ea74f579 with test.txt 'version 1', 0155eb4229851634a0f03eb265b69f5a2d56f341 that formally had new.txt and test.txt 'version 2', and the latest tree 3c4e9cd789d88d8d89c1073707c3585e41b0e614 with the d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree as well as new.txt and test.txt 'version 2' blobs) that represent the different snapshots of your project that you want to track. That’s the good news. Bad news; you still have to remember all three 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. Remedy...enter the commit object.

## commit object

To create a commit object, you call commit-tree and specify a single tree SHA-1 and which commit objects, if any, directly preceded it. Start with the first tree you wrote:

$ echo 'First commit' | git commit-tree d8329f

ba89b6b0fced65ccb90fe4ff27327a8c3b551e64

Now you can look at your new commit object:

$ git cat-file -p ba89b

tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579

author Mark McFadden <m2web@yahoo.com> 1592405875 -0400

committer Mark McFadden <m2web@yahoo.com> 1592405875 -0400

First commit

The format for a commit object is simple:

(1) it specifies the top-level tree for the snapshot of the project at that point.

(2) the parent commits if any (the commit object described above does not have any parents as it is the first commit)

(3) the author/committer information (which use  
(4) a blank line, and then the commit message

Next, we’ll write the other two commit objects, each referencing the commit that came directly before it:

$ echo 'Second commit' | git commit-tree 0155eb -p ba89b

fe4e9cf5fd9207848eeb9efedfb3e703290610b3

$ echo 'Third commit' | git commit-tree 3c4e9c -p fe4e9

9789529a6093286a847f440bbdde806c9778757a

Each of the three commit objects points to one of the three snapshot trees you created. Oddly enough, you have a real Git history now that you can view with the git log command, if you run it on the last commit SHA-1:

$ git log --stat 978952

commit 9789529a6093286a847f440bbdde806c9778757a

Author: Mark McFadden <m2web@yahoo.com>

Date: Wed Jun 17 11:31:20 2020 -0400

Third commit

bak/test.txt | 1 +

1 file changed, 1 insertion(+)

commit fe4e9cf5fd9207848eeb9efedfb3e703290610b3

Author: Mark McFadden <m2web@yahoo.com>

Date: Wed Jun 17 11:30:26 2020 -0400

Second commit

new.txt | 1 +

test.txt | 2 +-

2 files changed, 2 insertions(+), 1 deletion(-)

commit ba89b6b0fced65ccb90fe4ff27327a8c3b551e64

Author: Mark McFadden <m2web@yahoo.com>

Date: Wed Jun 17 10:57:55 2020 -0400

First commit

test.txt | 1 +

1 file changed, 1 insertion(+)

Awesome! We’ve just done the low-level operations (plumbing) to build up a Git history without using any of the front end commands (porcelain). This is essentially what Git does when you run the git add and git commit commands—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. These three main Git objects—the blob, the tree, and the commit—are initially stored as separate files in your .git/objects directory.

Here are all the objects in the example directory now, commented with what they store:

$ find .git/objects -type f

.git/objects/24/941f46f1b778a70a22424eb4c26fbefe629ea8 # blob - "Once more unto..."

.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # blob - test.txt v1

.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1

.git/objects/ba/89b6b0fced65ccb90fe4ff27327a8c3b551e64 # commit 1

.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3

.git/objects/97/89529a6093286a847f440bbdde806c9778757a # commit 3

.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # blob - new.txt

.git/objects/fe/4e9cf5fd9207848eeb9efedfb3e703290610b3 # commit 2

.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2

.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # blob - test.txt v2

If you follow all the internal pointers, you get an object graph something like this:  


# Git References

If you were interested in seeing the history of your repository reachable from commit, say, 97895, you could run something like git log 97895 to display that history, but you would still have to remember that 97895 is the commit you want to use as the starting point for that history. Instead, it would be easier if you had a file in which you could store that SHA-1 value under a simple name so you could use that simple name rather than the raw SHA-1 value.

In Git, these simple names are called “references” or “refs”; you can find the files that contain those SHA-1 values in the .git/refs directory. In the current project, this directory contains no files, but it does contain a simple structure:

$ find .git/refs

.git/refs

.git/refs/heads

.git/refs/tags

To create a new reference that will help you remember where your latest commit is, you can technically do something as simple as this:

$ echo 9789529a6093286a847f440bbdde806c9778757a > .git/refs/heads/master

Note that 9789529a6093286a847f440bbdde806c9778757a is the third commit created above.

Now, you can use the head reference you just created instead of the SHA-1 value in your Git commands:

$ git log --pretty=oneline master

9789529a6093286a847f440bbdde806c9778757a (HEAD -> master) Third commit

fe4e9cf5fd9207848eeb9efedfb3e703290610b3 Second commit

ba89b6b0fced65ccb90fe4ff27327a8c3b551e64 First commit

You aren’t encouraged to directly edit the reference files; instead, Git provides the safer command git update-ref to do this if you want to update a reference:

$ git update-ref refs/heads/master 9789529a6093286a847f440bbdde806c9778757a

That’s basically what a branch in Git is: a simple pointer or reference to the head of a line of work. To create a branch back at the second commit, you can do this:

$ git update-ref refs/heads/test fe4e9

Your branch will contain only work from that commit down:

$ git log --pretty=oneline test

fe4e9cf5fd9207848eeb9efedfb3e703290610b3 (test) Second commit

ba89b6b0fced65ccb90fe4ff27327a8c3b551e64 First commit

Now, your Git database conceptually looks something like this:



When you run commands like git branch <branch>, Git basically runs that update-ref command to add the SHA-1 of the last commit of the branch you’re on into whatever new reference you want to create.

## The HEAD

The question now is, when you run git branch <branch>, how does Git know the SHA-1 of the last commit? The answer is the HEAD file.

Usually the HEAD file is a symbolic reference to the branch you’re currently on. By symbolic reference, we mean that unlike a normal reference, it contains a pointer to another reference.

However in some rare cases the HEAD file may contain the SHA-1 value of a git object. This happens when you checkout a tag, commit, or remote branch, which puts your repository in "detached HEAD" state.

If you look at the file, you’ll normally see something like this:

$ cat .git/HEAD

ref: refs/heads/master

If you run git checkout test, Git updates the file to look like this:

$ cat .git/HEAD

ref: refs/heads/test

When you run git commit, it creates the commit object, specifying the parent of that commit object to be whatever SHA-1 value the reference in HEAD points to.

You can also manually edit this file, but again a safer command exists to do so: git symbolic-ref. You can read the value of your HEAD via this command:

$ git symbolic-ref HEAD

refs/heads/master

You can also set the value of HEAD using the same command:

$ git symbolic-ref HEAD refs/heads/test

$ cat .git/HEAD

ref: refs/heads/test

You can’t set a symbolic reference outside of the refs style:

$ git symbolic-ref HEAD test

fatal: Refusing to point HEAD outside of refs/

## Tags

We just finished discussing Git’s three main object types (blobs, trees and commits), 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.

You can make a lightweight tag by running something like this:

$ git update-ref refs/tags/v1.0 fe4e9cf5fd9207848eeb9efedfb3e703290610b3

That is all a lightweight tag is--a reference that never moves. An annotated tag is more complex, however. If you create an annotated tag, Git creates a tag object and then writes a reference to point to it rather than directly to the commit. You can see this by creating an annotated tag (using the -a option):

$ git tag -a v1.1 9789529a6093286a847f440bbdde806c9778757a -m 'Test tag'

Here’s the object SHA-1 value it created:

$ cat .git/refs/tags/v1.1

53ec66fd107435d4940f49cbffc42f4b2b4a84f5

Now, run git cat-file -p on that SHA-1 value:

$ git cat-file -p 53ec66

object 9789529a6093286a847f440bbdde806c9778757a

type commit

tag v1.1

tagger Mark McFadden <m2web@yahoo.com> 1592420299 -0400

Test tag

Notice that the object entry points to the commit SHA-1 value that you tagged (9789529a6093286a847f440bbdde806c9778757a). Also notice that it doesn’t need to point to a commit; **you can tag any Git object**.

What!?!? I can tag ANY Git object? Yes.

For example, let's say for some reason I wanted to tag my tree #3 above, 3c4e9cd789d88d8d89c1073707c3585e41b0e614, with the tag "treeV.3" and the tag message 'tree 3 tag':

$ git tag -a treeV.3 3c4e9 -m 'tree 3 tag'

Here’s the object SHA-1 value that the new tag created:

$ cat .git/refs/tags/treeV.3

b1a36d0c585be573ed2833e2b9cb748ab1d54327

Let's look into the tag object for the tree:

$ git cat-file -p b1a36d0c585be573ed2833e2b9cb748ab1d54327

object 3c4e9cd789d88d8d89c1073707c3585e41b0e614

type tree

tag treeV.3

tagger Mark McFadden <m2web@yahoo.com> 1592420846 -0400

tree 3 tag

## Remotes

The third type of reference that you’ll see is a remote reference. 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:

$ git remote add origin https://github.com/m2web/gitinternals.git

$ git push -u origin master

Enumerating objects: 9, done.

Counting objects: 100% (9/9), done.

Compressing objects: 100% (5/5), done.

Writing objects: 100% (9/9), 749 bytes | 749.00 KiB/s, done.

Total 9 (delta 0), reused 0 (delta 0)

To https://github.com/m2web/gitinternals.git

\* [new branch] master -> master

Branch 'master' set up to track remote branch 'master' from 'origin'.

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:

$ cat .git/refs/remotes/origin/master

9789529a6093286a847f440bbdde806c9778757a

Remote references differ from branches (refs/heads references) mainly in that they’re considered read-only. You can git checkout to one, but Git won’t point HEAD at one, so you’ll never update it with a commit command. Git manages them as bookmarks to the last known state of where those branches were on those servers.

# tree ./git output

