A brief look at `git` internals

Michael Hannon

2016-01-20

Contents

A different perspective on `git` **1**

Plumbing and porcelain 2

Objects in `git` 2

Set up and populate a repository **2**

New repository 2

Add some files 2

Move files to staging area and commit 3

Explore the objects **3**

The `HEAD` of the current branch 3

File corresponding to the `HEAD` commit 4

Found the file. Now what's in it? 5

The `HEAD` points to another object, a `tree` 5

Digression: other ways to look at `git` objects 6

Back to the `tree` object 8

Got the `tree` items – keep digging 9

Recap so far 11

One more thing – make the objects **11**

A different perspective on `git`

```
Git is fundamentally a content-addressable filesystem with a VCS
user interface written on top of it.
```

https://git-scm.com/book/en/v2/Git-Internals-Plumbing-and-Porcelain

Plumbing and porcelain

Two different types of commands:

- Plumbing (original and lower-level, oriented toward file system)
- Porcelain (newer and higher-level, oriented toward VCS)

Objects in `git`

There are four kinds of objects that are fundamental to the workings of `git`:

- blob
- tree
- commit
- tag

See, for instance:

http://www.gitguys.com/topics/all-git-object-types-blob-tree-commit-and-tag/

To explore the use of `git` at any level we need to have a repository to work with:

Set up and populate a repository

New repository

First, make a new git repository:

```
if [ -d ~/test ]; then
    \rm -rf ~/test
fi

mkdir ~/test
cd ~/test

git init
```

```
## Initialized empty Git repository in /home/mike/test/.git/
```

Add some files

Now add a couple of files to the new directory:

```
cd ~/test

cat <<EOF > firstFile

Friends, Romans, countrymen, lend me your ears;
I come to bury Caesar, not to praise him.

EOF

cat <<EOF > secondFile

The evil that men do lives after them;
The good is oft interred with their bones;

EOF
```

Move files to staging area and commit

```
cd ~/test
git add *
git commit -m "Initial commit -- two files about Julius"
```

```
## [master (root-commit) dce9268] Initial commit -- two files about Julius
##  2 files changed, 8 insertions(+)
##  create mode 100644 firstFile
##  create mode 100644 secondFile
```

Explore the objects

The `HEAD` of the current branch

We've done no branching, so we're currently on the `master` branch. Find the commit object that corresponds to the `HEAD` of this branch.

```
cd ~/test
head_commit=`cat .git/HEAD`
echo $head_commit
```

```
## ref: refs/heads/master
```

We have the HEAD commit, and it appears to contain a file name. What's in the file?

```
cd ~/test
cat .git/refs/heads/master
```

```
## dce92687b0106162951348244992d2bc6b42c0ec
```

The thing that's in the `.../master` file is a so-called `SHA`:

https://en.wikipedia.org/wiki/SHA-1

Taken at face value, the `SHA` is just an ID number. But it means more than that in `git`.

The `HEAD` of the branch is, in effect, a pointer to a file in a subdirectory of `.../objects/`.

The name of the subdirectory is just the first two characters of the `SHA`, and the name of the file is the remaining part of the `SHA`:

File corresponding to the `HEAD` commit

Given the information just above, we can find the file associated with the `HEAD` commit. Here's the `SHA` again:

```
cd ~/test
headSHA=`cat .git/refs/heads/master`
echo -n "SHA of HEAD commit on master branch: "
echo $headSHA
```

```
## SHA of HEAD commit on master branch: dce92687b0106162951348244992d2bc6b42c0ec
```

Here's the subdirectory for which we're looking:

```
subDir=${headSHA:0:2}
echo -n "Name of subdirectory: "
echo $subDir
```

```
## Name of subdirectory: dc
```

We've now found the subdirectory of `.../objects` for which we were looking. The next task is to find and examine the file in that subdirectory. As mentioned above, the name of the file is just the part of the `SHA` that remains after stripping off the first two characters.

```
lenSHA=`expr length $headSHA`

fileName=${headSHA:2:lenSHA}
echo -n "File name: "
echo $fileName
```

```
## File name: e92687b01061629513482449 92d2bc6b42c0ec
```

Let's do a directory listing of the file, just to confirm its existence.

```
echo -e "Directory listing for file:\n"
ls -l .git/objects/$subDir/$filename
```

```
## Directory listing for file:
##
## total 4
## -r--r----- 1 mike mike 155 Jan 20 14:14 e92687b0106162951348244992d2bc6b42c0ec
```

Found the file. Now what's in it?

We can now use one of the `git` "plumbing" commands to examine the file we've just tracked down. We first look at the `type` of the file we've found, then we look at the contents (both via the `git cat-file` command):

```
echo -e "File type:\n"
git cat-file -t $headSHA
```

```
## File type:
##
## commit
```

The type of the file is `commit`, which isn't much of a surprise, as we started out looking at the `SHA` of a commit. But what's in the file?

```
echo -e "File contents:\n"
git cat-file -p $headSHA
```

```
## File contents:
##
## tree d8d296e163cd7fa8cc1f3a9cc9290e61d73d38ae
## author Michael Hannon <jmhannon.ucdavis@gmail.com> 1453328066 -0800
## committer Michael Hannon <jmhannon.ucdavis@gmail.com> 1453328066 -0800
##
## Initial commit -- two files about Julius
```

The `HEAD` points to another object, a `tree`

Note from the output of `git cat-file -p ...` that the `commit` object for the `HEAD` of the `master` branch contains some meta information about the commit, but, more important for our purposes, it also contains a pointer to *another* object of type `tree`.

Digression: other ways to look at git objects

Before we examine that `tree` object, let's note that we can track down `git` objects using other utilities, including modules in `python` and the `zpipe` utility.

Note that `git` uses the `zlib` utility to compress and decompress objects:

https://en.wikipedia.org/wiki/Zlib

(This uses the same compression algorithm as the `gzip` utility.)

Looking at git objects from python

Hence, we can do the same compressing and decompressing in `python` (i.e., just to show that we *can* do it) using the `zlib` module:

https://docs.python.org/2/library/zlib.html

Let's have a look. The procedure is exactly analogous to the procedure we used above in the `bash` shell.

```
import os
import subprocess as sp


## headSHA = sp.check_output(["cat", "/home/mike/test/.git/refs/heads/master"])
headFile = open("/home/mike/test/.git/refs/heads/master", 'r')
headSHA  = headFile.read()
headSHA  = headSHA.strip('\n')

subDir = headSHA[0:2]

commitDir = "/home/mike/test/.git/objects/" + subDir + "/"
commitObj = headSHA[2: ]
commitFullPath = commitDir + commitObj

import zlib

commitFile     = open(commitFullPath, 'r')
zip_content    = commitFile.read()
commit_content = zlib.decompress(zip_content)

commit_content = commit_content.translate(None, '\0')  ## remove nulls

print(commit_content)
```

```
## commit 227tree d8d296e163cd7fa8cc1f3a9cc9290e61d73d38ae
## author Michael Hannon <jmhannon.ucdavis@gmail.com> 1453328066 -0800
## committer Michael Hannon <jmhannon.ucdavis@gmail.com> 1453328066 -0800
```

```
##
## Initial commit -- two files about Julius
```

(We have to eliminate the `null` character ('\0') in order to get the result to print correctly.)

Looking at `git` objects using `zpipe`

Just to belabor the point a bit, note that *another* way to examine a `git` object is to use the `zpipe` utility, as:

```
zpipe -d <git-object>  >  <decompressed version of object>
```

Here's an example and a comparison to the output of `git cat-file -p`:

```
zpipe -d < ${fileName} > zpipe.out
echo -e "output from zpipe:\n"

tr < zpipe.out -d '\000' > zpipe.out.no.null

cat zpipe.out.no.null

echo -e "\noutput from git cat-file:\n"

git cat-file -p ${headSHA}
```

```
## output from zpipe:
##
## commit 227tree d8d296e163cd7fa8cc1f3a9cc9290e61d73d38ae
## author Michael Hannon <jmhannon.ucdavis@gmail.com> 1453328066 -0800
## committer Michael Hannon <jmhannon.ucdavis@gmail.com> 1453328066 -0800
##
## Initial commit -- two files about Julius
##
## output from git cat-file:
##
## tree d8d296e163cd7fa8cc1f3a9cc9290e61d73d38ae
## author Michael Hannon <jmhannon.ucdavis@gmail.com> 1453328066 -0800
## committer Michael Hannon <jmhannon.ucdavis@gmail.com> 1453328066 -0800
##
## Initial commit -- two files about Julius
```

(Note that the output from `python` and the output from `zpipe` both contain an integer in the first line, immediately after the `commit` keyword. This is the number of bytes of actual content in the file. Evidently, `git cat-file` uses but does not display the information.)

Back to the tree object

We've examined the commit object in three different ways, all of which indicated the existence of another object, a `tree` object. Let's track that down and see what's in it.

```
print("commit_content....:\n")
print(commit_content)

tree_line = commit_content.split(' ')[2]

tree_commit = tree_line.split('\n')[0]
print("\ntree_commit....:\n")
print(tree_commit)

os.chdir("/home/mike/test")
tree_content = sp.check_output(["git", "cat-file", "-p", tree_commit])
print("\ntree_content (from 'git cat-file -p' ....:\n")
print(tree_content)

tree_content = tree_content.rstrip('\n')
tree_file_list = tree_content.split('\n')
print("\nList of tree_file items (from splitting lines in'tree_content')....:\n")
print(tree_file_list)

def extract_file_info(tree_file_list_item):


    file_name  = tree_file_list_item.split('\t')[1]
    other_info = tree_file_list_item.split('\t')[0]
    file_hash  =  other_info.split(' ')[2]

    return(file_name, file_hash)

tree_file_info = map(extract_file_info, tree_file_list)
print("\n List of (file_name, SHA_name) combinations in the tree....:\n")
print(tree_file_info)
```

```
## commit_content....:
##
## commit 227tree d8d296e163cd7fa8cc1f3a9cc9290e61d73d38ae
## author Michael Hannon <jmhannon.ucdavis@gmail.com> 1453328066 -0800
## committer Michael Hannon <jmhannon.ucdavis@gmail.com> 1453328066 -0800
##
## Initial commit -- two files about Julius
##
##
## tree_commit....:
```

```
##
## d8d296e163cd7fa8cc1f3a9cc9290e61d73d38ae
##
## tree_content (from 'git cat-file -p' ....:
##
## 100644 blob c876212bc93ee76bcaf240c271073c789b4ff664 firstFile
## 100644 blob 2de2a2fd44433be9c56c0e555af53dceb4077552 secondFile
##
##
## List of tree_file items (from splitting lines in'tree_content')....:
##
## ['100644 blob c876212bc93ee76bcaf240c271073c789b4ff664\tfirstFile', '100644 blob 2de2a2f
##
##  List of (file_name, SHA_name) combinations in the tree....:
##
## [('firstFile', 'c876212bc93ee76bcaf240c271073c789b4ff664'), ('secondFile', '2de2a2fd4443
```

Got the `tree` items – keep digging

We've now found a list of files that are associated with the `HEAD` of the current branch. We've got a list of `(SHA_name, file_system_name)` combinations for all files pointed to by the current `HEAD` commit.

As above, we can decode the contents of `SHA_name` by any of several ways:

- use the `git` "plumbing" command, `git cat-file -p SHA_name`
- use `zlib` in `python`
- use the `zpipe` utility

Let's have a look. We'll do this from within `python`, as we've already collected all the relevant information in `python`..

```
###### First with: git cat-file

for (file_name, SHA_name) in tree_file_info:
    print("File name (from git cat-file)....: " + file_name)
    file_contents = sp.check_output(["git", "cat-file", "-p", SHA_name])
    print("\nFile contents (from git cat-file)....:\n")
    print(file_contents)

###### Now with zlib

import zlib


for (file_name, SHA_name) in tree_file_info:
```

```python
    subDir = SHA_name[0:2]
    filesDir = "/home/mike/test/.git/objects/" + subDir + "/"
    fileObj = SHA_name[2: ]
    fileFullPath = filesDir + fileObj

    fin = open(fileFullPath, 'r')
    zip_content = fin.read()
    file_contents = zlib.decompress(zip_content)

    file_contents = file_contents.translate(None, '\0')

    print("\nFile contents (from python zlib)....:\n")
    print(file_contents)
```

```
## File name (from git cat-file)....: firstFile
##
## File contents (from git cat-file)....:
##
##
## Friends, Romans, countrymen, lend me your ears;
## I come to bury Caesar, not to praise him.
##
##
## File name (from git cat-file)....: secondFile
##
## File contents (from git cat-file)....:
##
##
## The evil that men do lives after them;
## The good is oft interred with their bones;
##
##
##
## File contents (from python zlib)....:
##
## blob 92
## Friends, Romans, countrymen, lend me your ears;
## I come to bury Caesar, not to praise him.
##
##
##
## File contents (from python zlib)....:
##
## blob 84
## The evil that men do lives after them;
## The good is oft interred with their bones;
```

The additional information shown in the `zlib` version shows:

- The object type, namely, `blob`
- The number of characters in the original file, 92 and 84

We can check that in the shell:

```
cd ~/test
ls

echo ""

wc *
```

```
## firstFile
## secondFile
##
##   4  16  92 firstFile
##   4  16  84 secondFile
##   8  32 176 total
```

And we see that the `ls` command does indeed report the same number of characters in the respective files as we have inferred from using `zlib`.

Recap so far

The steps we've taken in the above are:

- found the `HEAD` commit
- decoded the file object associated with the `HEAD` commit
- followed the link from that file object to a `tree` object
- decoded the file object associated with the `tree` object
- followed the link from that file object to some `blob` objects
- decoded the `blob` objects to reproduce the original files from our working directory

One more thing – make the objects

One final thing of interest is to explore how a `blob` file gets created in the first place. I.e., starting from a regular file, say `firstFile`, how does it get transformed into a `blob`.

We first have to create the `SHA1` "hash", then do the compressing. Both can be done with `python` modules, `hashlib` and `zlib`, respectively. Here's an example:

```
import os
os.chdir("/home/mike/test")

fin = open("firstFile", 'r')
content = fin.read()

print("Original file content....:")
print(content)
```

```
## Original file content....:
##
## Friends, Romans, countrymen, lend me your ears;
## I come to bury Caesar, not to praise him.
```

We now have the file content. Next we make the header.

```
header  = "blob {0}\0".format(len(content))
print("Header that we created....:\n")
print(repr(header))
```

```
## Header that we created....:
##
## 'blob 92\x00'
```

We now combine the header and the file content and “hash” the resulting object.

```
store = header + content

import hashlib
sha1 = hashlib.sha1()

sha1.update(store)

sha1_digest = sha1.hexdigest()
print("SHA hexdigest that we created....:\n")
print(sha1_digest)
```

```
## SHA hexdigest that we created....:
##
## c876212bc93ee76bcaf240c271073c789b4ff664
```

Almost there. We’ve hashed the “store”. Now we compress it and write it to a file.

```python
import zlib
zlib_content = zlib.compress(store)

print("Here's where we're storing our blob file....:\n")
print(os.getcwd())

fout = open("firstFile.blob.python", 'w')
fout.write(zlib_content)

fout.close()
```

```
## Here's where we're storing our blob file....:
##
## /home/mike/test
```

We deliberately put the `python` version of the "blob" in our working directory, a place outside the control of `git`.

Now let's examine the contents of the working directory.

```python
import subprocess as sp
pythonBlobList = sp.check_output(["ls", "-l"])

print("Examine the blob we just created in working directory....:\n")
print(pythonBlobList)
```

```
## Examine the blob we just created in working directory....:
##
## total 12
## -rw-rw---- 1 mike mike 92 Jan 20 14:14 firstFile
## -rw-rw---- 1 mike mike 97 Jan 20 14:14 firstFile.blob.python
## -rw-rw---- 1 mike mike 84 Jan 20 14:14 secondFile
```

Now we can go through the same process as above, picking off the first two digits of the `SHA` to find the `git` subdirectory and using the remaining digits as the file name.

```python
import subprocess as sp

subDir = sha1_digest[0:2]

blobDir = "/home/mike/test/.git/objects/" + subDir + "/"
blobObj = sha1_digest[2: ]
blobFullPath = blobDir + blobObj

gitBlobList    = sp.check_output(["ls", "-l", blobFullPath])

print("Examine the blob that git created in the .../objects dir....:\n")
print(gitBlobList)
```

```
## Examine the blob that git created in the .../objects dir....:
##
## -r--r----- 1 mike mike 97 Jan 20 14:14 /home/mike/test/.git/objects/c8/76212bc93ee76bcaf
```

The two `blob` files have the same number of bytes, although they *do* differ in the second byte of the file (and *only* in that byte). The `python` blob has the `ASCII` character `fs` ("file separator") at that position, while the `git` blob has the `ASCII` character `soh` ("start of heading") at that position.

I don't know the source of the discrepancy – maybe different versions of `zlib` or different options or . . . ? In any case, it's clear that the `python` procedure is essentially equivalent to the procedure use by `git`.