

10.2 Git Internals - Git Objects

Git Objects

Git is a content-addressable filesystem. Great. What does that mean? It means that at the core of Git is a simple key-value data store. What this means is that you can insert any kind of content into a Git repository, for which Git will hand you back a unique key you can use later to retrieve that content.

As a demonstration, let's look at the plumbing command `git hash-object`, which takes some data, stores it in your `.git/objects` directory (the *object database*), and gives you back the unique key that now refers to that data object.

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 /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

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 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

In its simplest form, `git hash-object` would take the content you handed to it and merely return the unique key 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.

The output from the above command is a 40-character checksum hash. This is the SHA-1 hash — a checksum of the content you're storing plus a header, which you'll learn about in a bit. Now you can see how Git has stored your data:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

If you again examine your objects directory, you can see that it now contains a file for that new content. This is how Git stores the content initially — as a single file per piece of content, named with the SHA-1 checksum of the content and its header. The subdirectory is named with the first 2 characters of the SHA-1, and the filename is the remaining 38 characters.

Once you have content in your object database, you can examine that content with the `git cat-file` command. This command is sort of a Swiss army knife for inspecting Git objects. Passing `-p` to `cat-file` instructs the command to first figure out the type of content, then display it appropriately:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

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 object database now contains both versions of this new 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
```

At this point, you can delete your local copy of that `test.txt` file, then use Git to retrieve, from the object database, either the first version you saved:

```
$ 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*. You can have Git tell you the object type of any object in Git, given its SHA-1 key, with `git cat-file -t`:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Tree Objects

The next type of Git object we'll examine is the *tree*, which solves the problem of storing the filename and also allows you to store a group of files together. Git stores content in a manner similar to a UNIX filesystem, but a bit simplified. All the content is stored as tree and blob objects, with trees corresponding to UNIX directory entries and blobs corresponding more or less to inodes or file contents. A single tree object contains one or more entries, each of which is the SHA-1 hash of a blob or subtree with its associated mode, type, and filename. For example, the most recent tree in a project may look something like this:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859    README
100644 blob 8f94139338f9404f26296befa88755fc2598c289    Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0    lib
```

The `master^{tree}` syntax specifies the tree object that is pointed to by the last commit on your `master` branch. Notice that the `lib` subdirectory isn't a blob but a pointer to another tree:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b    simplegit.rb
```

Depending on what shell you use, you may encounter errors when using the `master^{tree}` syntax.

Note In CMD on Windows, the `^` character is used for escaping, so you have to double it to avoid this: `git cat-file -p master^^{tree}`. When using PowerShell, parameters using `{}` characters have to be quoted to avoid the parameter being parsed incorrectly: `git cat-file -p 'master^{tree}'`.

If you're using ZSH, the `^` character is used for globbing, so you have to enclose the whole expression in quotes: `git cat-file -p "master^{tree}"`.

Conceptually, the data that Git is storing looks something like this:

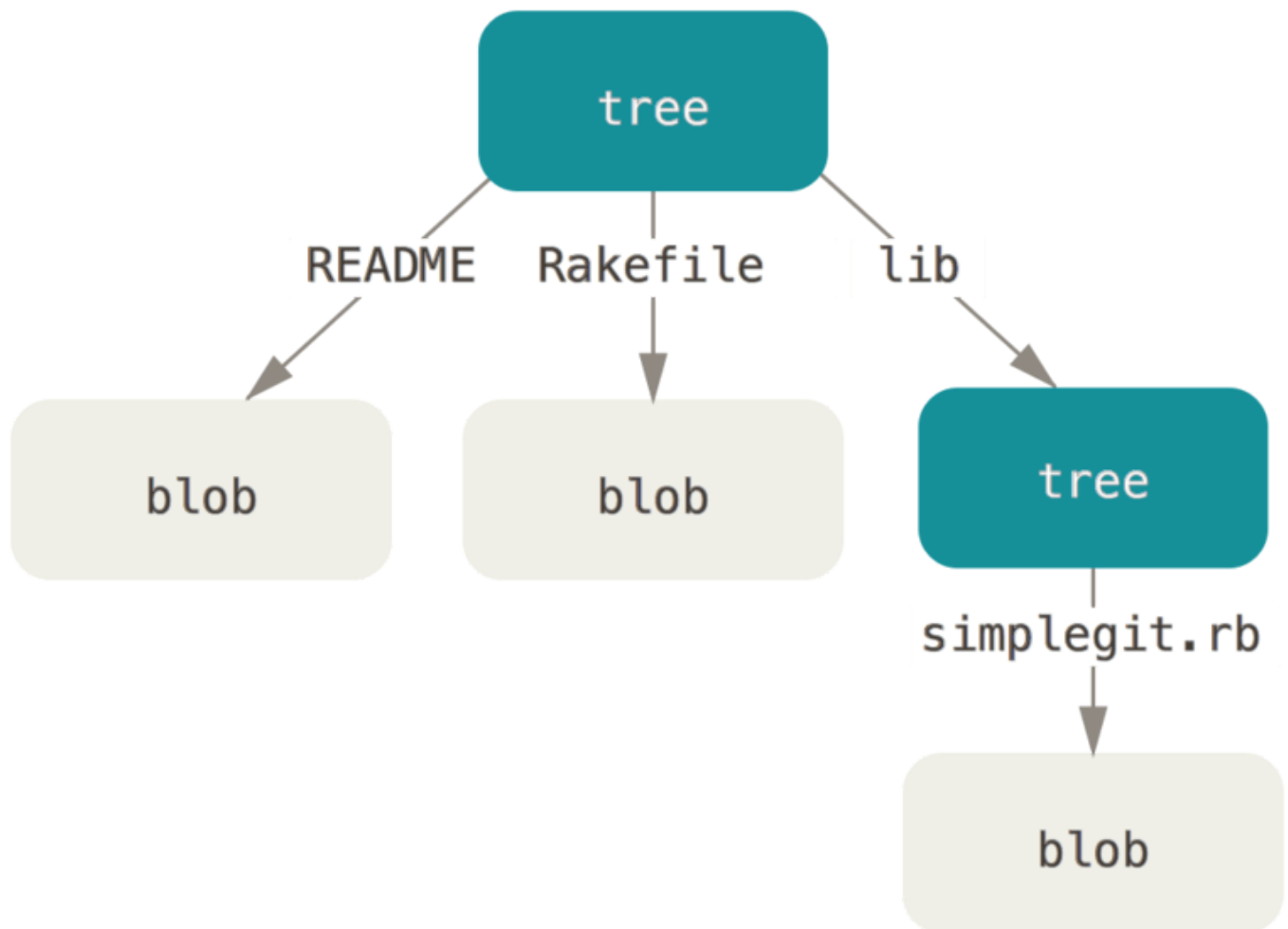


Figure 147. Simple version of the Git data model

You can fairly easily create your own tree. 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. The mode is taken from normal UNIX modes but is much less flexible — these three modes are the only ones that are valid for files (blobs) in Git (although other modes are used for directories and submodules).

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
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

You can also verify that this is a tree object using the same `git cat-file` command you saw earlier:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

You'll now create a new tree with the second version of `test.txt` and a new file as well:

```
$ echo 'new file' > new.txt
$ git update-index --add --cacheinfo 100644 \
```

```
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
$ git update-index --add new.txt
```

Your 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
$ 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
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579    bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a    test.txt
```

If you created a working directory from the new tree you just wrote, you would get the two files in the top level of the working directory and a subdirectory named `bak` that contained the first version of the `test.txt` file. You can think of the data that Git contains for these structures as being like this:

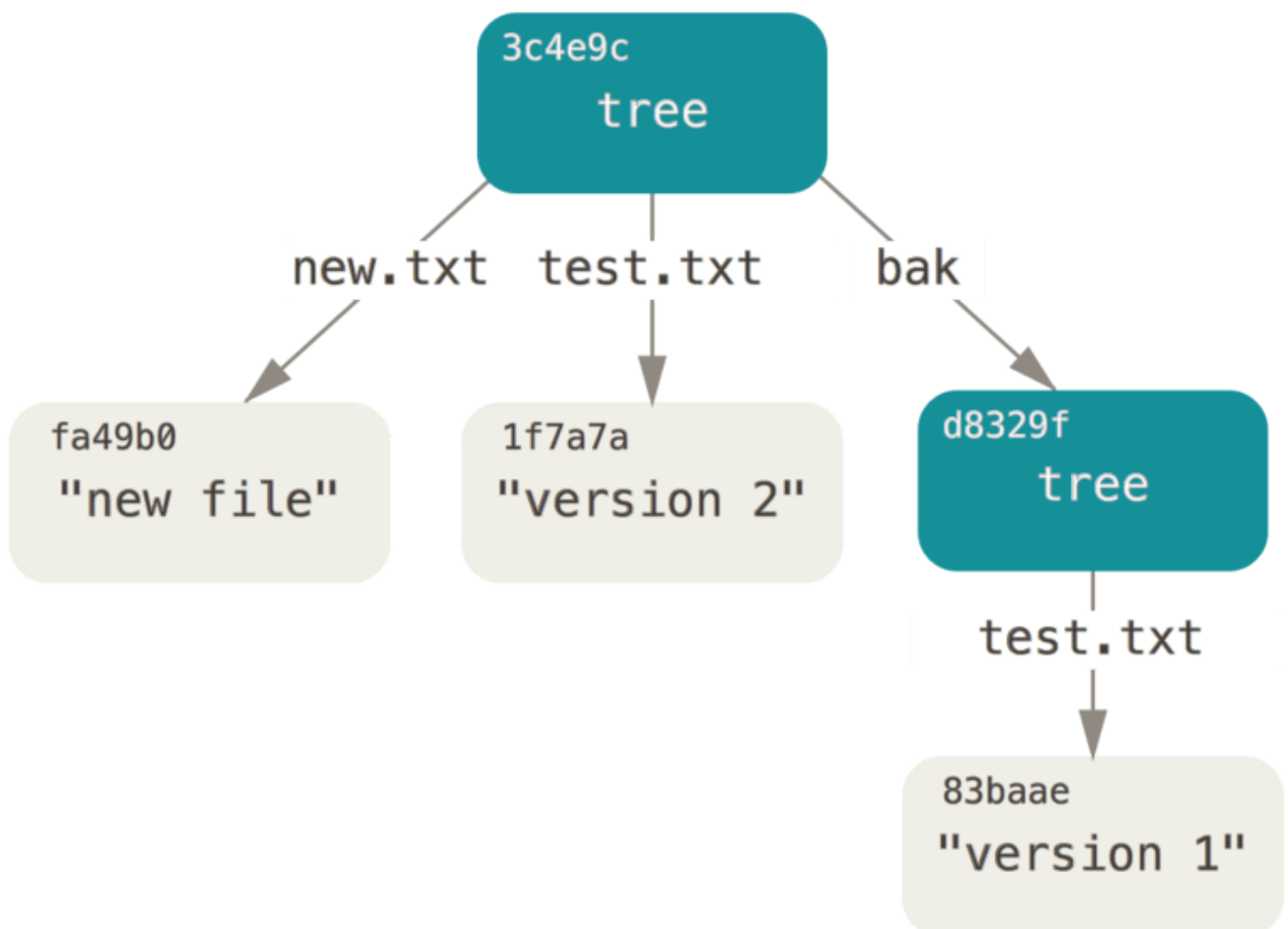


Figure 148. The content structure of your current Git data

Commit Objects

If you’ve done all of the above, you now have three trees that represent the different snapshots of your project that you want to track, but the earlier problem remains: you must 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. This is the basic information that the commit object stores for you.

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
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Note You will get a different hash value because of different creation time and author data. Moreover, while in principle any commit object can be reproduced precisely given that data, historical details of this book's construction mean that the printed commit hashes might not correspond to the given commits. Replace commit and tag hashes with your own checksums further in this chapter.

Now you can look at your new commit object with `git cat-file`:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700
```

First commit

The format for a commit object is simple: it specifies the top-level tree for the snapshot of the project at that point; the parent commits if any (the commit object described above does not have any parents); the author/committer information (which uses your `user.name` and `user.email` configuration settings and a timestamp); a blank line, and then the commit message.

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

```
$ echo 'Second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'Third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

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 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700
```

Third commit

```
bak/test.txt | 1 +
1 file changed, 1 insertion(+)
```

```
commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700
```

Second commit

```
new.txt | 1 +
test.txt | 2 +-
2 files changed, 2 insertions(+), 1 deletion(-)
```

```
commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700
```

First commit

```
test.txt | 1 +
1 file changed, 1 insertion(+)
```

Amazing. You've just done the low-level operations to build up a Git history without using any of the front end commands. 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/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

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

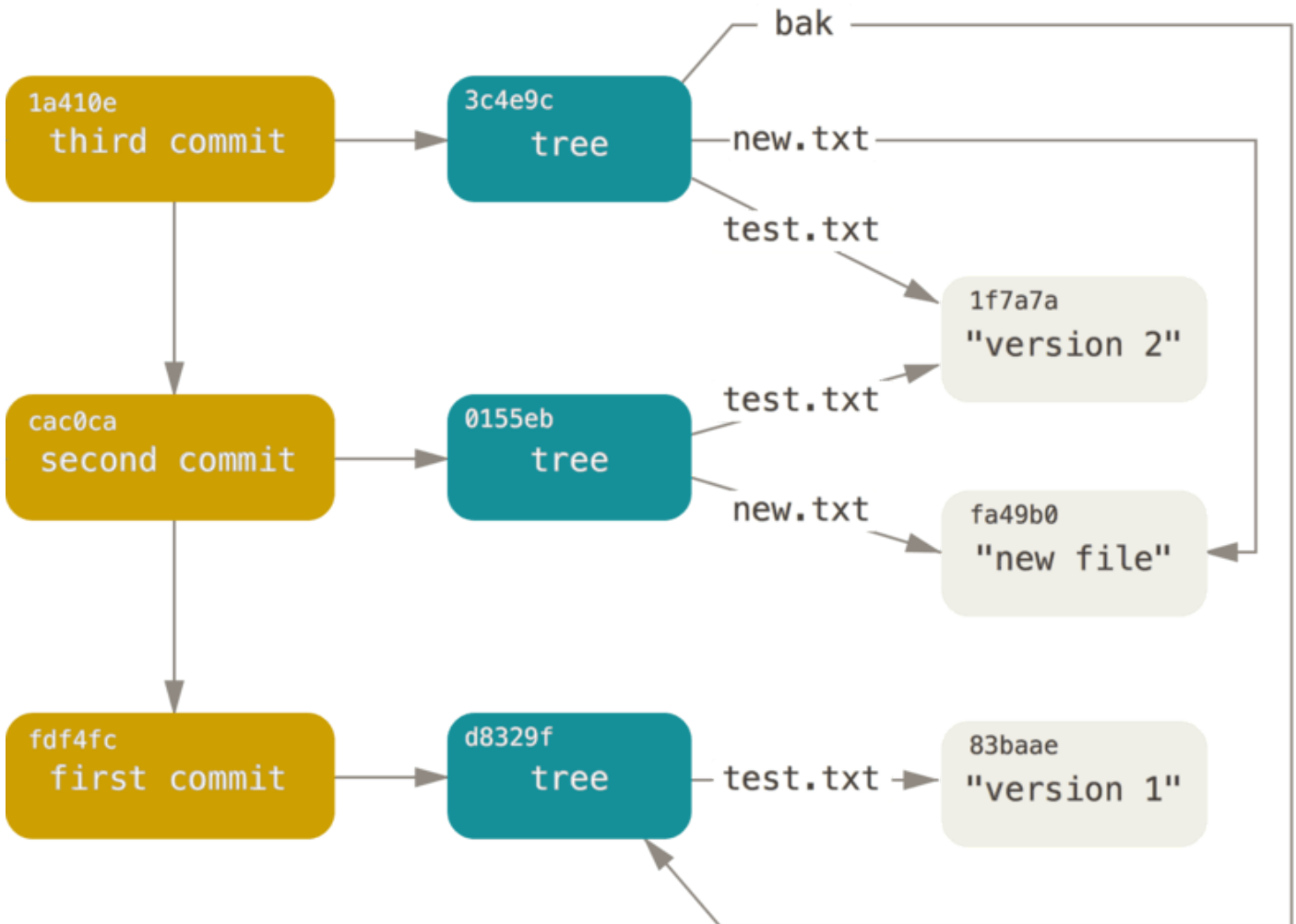


Figure 149. All the reachable objects in your Git directory

Object Storage

We mentioned earlier that there is a header stored with every object you commit to your Git object database. Let's take a minute to see how Git stores its objects. You'll see how to store a blob object — in this case, the string "what is up, doc?" — interactively in the Ruby scripting language.

You can start up interactive Ruby mode with the `irb` command:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git first constructs a header which starts by identifying the type of object — in this case, a blob. To that first part of the header, Git adds a space followed by the size in bytes of the content, and adding a final null byte:

```
>> header = "blob #{content.bytesize}\0"
=> "blob 16\u0000"
```

Git concatenates the header and the original content and then calculates the SHA-1 checksum of that new content. You can calculate the SHA-1 value of a string in Ruby by including the SHA1 digest library with the `require` command and then calling `Digest::SHA1.hexdigest()` with the string:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Let's compare that to the output of `git hash-object`. Here we use `echo -n` to prevent adding a newline to the input.

```
$ echo -n "what is up, doc?" | git hash-object --stdin
bd9dbf5aae1a3862dd1526723246b20206e5fc37
```

Git compresses the new content with `zlib`, which you can do in Ruby with the `zlib` library. First, you need to require the library and then run `Zlib::Deflate.deflate()` on the content:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\x9CK\xCA\xC90R04c(\xCFH,Q\xC8,V(-\xD0QH\xC90\xB6\xa\x00_\x1C\xa\x9D"
```

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). In Ruby, you can use the `FileUtils.mkdir_p()` function to create the subdirectory if it doesn't exist. Then, open the file with `File.open()` and write out the previously `zlib`-compressed content to the file with a `write()` call on the resulting file handle:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

Let's check the content of the object using `git cat-file`:

```
---
$ git cat-file -p bd9dbf5aae1a3862dd1526723246b20206e5fc37
what is up, doc?
---
```

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.

[prev](#) | [next](#)