Git from Scratch

Mark Bools

2023-10-23

Last Modified: 2023-10-23

Contents

Chapter 1

Setup

1.1 Required Tools

The simplest way to follow along with this book is to install the standardised environment.

- VirtualBox
- git
- vagrant

Chapter 2

Git Guts

Although commonly called a version control tool (and this is certainly the most common use) it is useful to think of Git as an object file system. That might not help much right now, but as we start to learn about Git you will hopefully appreciate why I say this.

In this book we take a bottom up approach to learning git, first learning the detail and then showing how the higher level git commands make many operations simpler.

Why not investigate git top down? I firmly believe that learning only the basic git operations is the cause of many problems people experience using git. The basic git operations commonly taught to new users leads to a misleading mental model of how git works, which in turn leads to confusion when git does not behave according to this faulty mental model. Although learning git 'bottom up' takes more effort you are rewarded with git superpowers.

2.1 Git Repository

Watch "Initial setup of Git repository" on Vimeo

When we use `git init`, Git will create a repository in a `.git` directory within the current working directory which become the Git workspace. You edit your files in the workspace just as you normally would and use `git` commands to manipulate objects and data stored in the `.git` database (known commonly as the repository).

```bash
mkdir class
cd class
ls -a
git init
tree -a
```

Within the `.git` directory are a number of files and sub-directories that constitute the `.git` database.

```tree -a
.git
├── branches
├── config
├── description
├── HEAD
├── hooks
│   ├── applypatch-msg.sample
│   ├── commit-msg.sample
│   ├── fsmonitor-watchman.sample
│   ├── post-update.sample
│   ├── pre-applypatch.sample
│   ├── pre-commit.sample
│   ├── prepare-commit-msg.sample
│   ├── pre-push.sample
│   ├── pre-rebase.sample
│   ├── pre-receive.sample
│   └── update.sample
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags

9 directories, 15 files
```

A few are of less interest to us at the moment:

- `config`—holds configuration to be used on this repository (more on `git` configuration later)
- `description`—used to provide a description of this repository to the web interface (not something we will look at for a while)

- `info`—contains the files to be ignored (`.gitignore`, to be investigated later) for this project's workspace.
- `hooks`—these are small scripts that can be triggered on certain actions. We will use these later but for now they can be ignored. These take up a lot of room on our output, we don't need these, so let's delete them (I'll leave the directory, even though it is not required, to remind us that it's typically there).

```bash
rm .git/hooks/*
```

The ones we are most interested in this chapter are:

- `HEAD`—Holds a special reference to the last object stored from the workspace
- `objects`—this holds the data
- `refs`—this holds `references` into the data in `objects`

We will see several other files and directories created as we use Git and we will discuss these as they occur.

There are several types of `object` held in `.git` repositories, the three we will encounter in this chapter are:

1. `blob`s—containing the data we want to store (typically files)
2. `tree`s—containing data about sets of `blob`s (and other `tree`s)
3. `commit`s—containing metadata about `tree`s

No need to worry about the details, all will become clear as we progress through this chapter.

2.2 `blobs`

Watch "Adding and displaying blobs" on Vimeo

We can use some low-level Git commands to create blobs directly[1]. The `git hash-object` sub-command creates and stores objects. Let's create an object:

[1] In day-to-day use we will use high-level commands to interact with our repository but in this chapter we're interested in learning what Git does under the hood.

```bash
echo 'version 1' > file1.txt
git hash-object file1.txt
tree .git
```

We created a simple text file and had `git hash-object` show us it's hash (a 40 character string, actually the SHA-1 hash of the file's content) but this object is not stored in the repository yet.

```tree -a
.git
├── branches
├── config
├── description
├── HEAD
├── hooks/
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags

9 directories, 15 files
```

To have `git hash-object` store the file we use the `-w` option.

```bash
git hash-object -w file1.txt
tree .git
```

tree -a

```
.git
├── branches
├── config
├── description
├── HEAD
├── hooks
├── info
│   └── exclude
├── objects
│   ├── 83
│   │   └── baae61804e65cc73a7201a7252750c76066a30
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags

10 directories, 16 files
```

The object is stored in the `objects` directory and the first two characters of the hash are used to create a directory (this is called 'sharding' and it is used to reduce the number of files stored in any one directory).

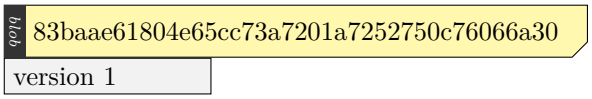

Figure 2.1: A Git 'blob'

It is important to note that Git has no idea what this blob is, it is just some data. No record is held about the original file name, for that matter Git doesn't even care that this blob came from a file.

bash

```
echo 'not a file' | git hash-object -w --stdin
tree .git
```

tree -a

```
.git
├── branches
├── config
├── description
├── HEAD
├── hooks
├── info
│   └── exclude
├── objects
│   ├── 7a
│   │   └── b4ff63b2ea4c2c3ff89ee972bc42988a4b8472
│   ├── 83
│   │   └── baae61804e65cc73a7201a7252750c76066a30
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags

11 directories, 17 files
```

Here the data for the blob is fed into Git straight from `stdin`, no file is involved this is 'raw data'.

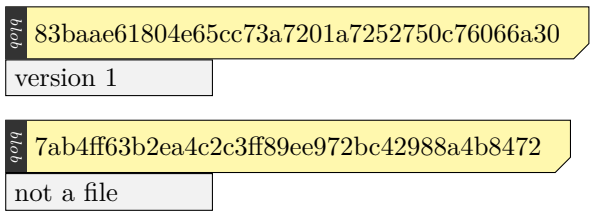

Figure 2.2: Two Git blobs

We can recall the blob from our repository using `git cat-file` (this is a bit misleading and would be better called `cat-object` because, as we shall see, we can use it to look inside various git objects).

bash

```
git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30
```

```
version 1
```

The `-p` option 'pretty prints' the content of the object to `stdout` so if we want to create a file from this object we need to redirect it ...

```bash
git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > new_file.txt
cat new_file.txt
```

```
version 1
```

Typing out those long hash identities quickly becomes tiresome. Fortunately Git allows us to specify shorter forms in many instances, specifically we can provide just enough of the start of an object's hash that is unambiguous.

```bash
git cat-file -p 83ba
```

In most circumstances 6 to 8 characters is sufficient, here we can use just 4 because our repository has so few entries this is all that is required to unambiguously reference each object. (We cannot go so far as reducing to just 2 as Git considers these too short—two characters will only identify the shard directory, not the object file.)

We can add another version of our `file1.txt` without any confusion (because Git does not care about the filename at this point).

```bash
echo 'version 2' > file1.txt
git hash-object -w file1.txt
```

Git adds the new object as a simple blob.

```bash
tree .git
git cat-file -p 1f7a
```

tree -a

```
.git
├── branches
├── config
├── description
├── HEAD
├── hooks
├── info
│   └── exclude
├── objects
│   ├── 1f
│   │   └── 7a7a472abf3dd9643fd615f6da379c4acb3e3a
│   ├── 7a
│   │   └── b4ff63b2ea4c2c3ff89ee972bc42988a4b8472
│   ├── 83
│   │   └── baae61804e65cc73a7201a7252750c76066a30
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags

 12 directories, 18 files
```

git cat-file -p 1f7a

```
version 2
```

So we can store blobs in our repository but this is of limited use as we normally deal with directories containing files and these tend to have human readable names (like `file1.txt`).

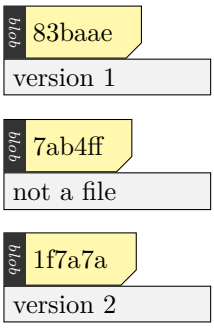

Figure 2.3: Three Git blobs

2.3 trees

Watch “Manipuate the index and create trees” on Vimeo

To get Git to track filenames and directories we have it create a different type of object called a ‘tree’ and to create tree objects we use the ‘index’. The index is a sort of holding area within our repository[2] (you will also see the ‘index’ called the ‘cache’ or ‘staging’ area). In the index we collect information about all of the objects we want to store in our repository, then we use a single command to create a tree entry using the entries in the index.

```bash
git update-index --add --cacheinfo 100644 83baae61804e65cc73a7201a7252750c76066a30 file1.txt
tree .git
```

```
.git
├── branches
├── config
├── description
├── HEAD
├── hooks
├── index
├── info
│   └── exclude
├── objects
│   ├── 1f
│   │   └── 7a7a472abf3dd9643fd615f6da379c4acb3e3a
│   ├── 7a
│   │   └── b4ff63b2ea4c2c3ff89ee972bc42988a4b8472
│   ├── 83
│   │   └── baae61804e65cc73a7201a7252750c76066a30
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags

12 directories, 19 files
```

[2]This is a lie! In Chapter 3 we will take a closer look at the index and learn why this lie is so often repeated.

`update-index` is used to manipulate our repository index. Initially a new repository has no index but after adding an object's information to the index we see a new file `index` (line 7 above). The `--cacheinfo` option specifies the object data to be added. The file's mode (`100644`) is stored, then the object hash (`83baae61804e65cc73a7201a7252750c76066a30`), and finally the filename we want to associated with the object (`file1.txt`). Note, these are entirely under our control in the `update-index` command and do not have to correspond with any real file. Even the object identity is not checked by the `update-index` command (you should always provide a real hash though, otherwise you will get an "invalid object" error when you attempt to write the tree—up next).

Having created our index we can examine its content using `git ls-files --stage`, the `--stage` option causes `ls-files` to display the mode and object hash.

bash

```
1 git ls-files --stage
2 git write-tree
3 git ls-files --stage
```

git ls-files --stage

```
1 100644 83baae61804e65cc73a7201a7252750c76066a30 0       file1.txt
```

git write-tree

```
1 b7e8fac7e3e35d93d39d2fa2260868f025a9efb4
```

git ls-files --stage

```
1 100644 83baae61804e65cc73a7201a7252750c76066a30 0       file1.txt
```

The `git write-tree` operation does not change the index file. The `ls-files` shows us that the `index` is the same before and after the `write-tree`.

bash

```
1 tree .git
```

```
.git
├── branches
├── config
├── description
├── HEAD
├── hooks
├── index
├── info
│   └── exclude
├── objects
│   ├── 1f
│   │   └── 7a7a472abf3dd9643fd615f6da379c4acb3e3a
│   ├── 7a
│   │   └── b4ff63b2ea4c2c3ff89ee972bc42988a4b8472
│   ├── 83
│   │   └── baae61804e65cc73a7201a7252750c76066a30
│   ├── b7
│   │   └── e8fac7e3e35d93d39d2fa2260868f025a9efb4
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags

13 directories, 20 files
```

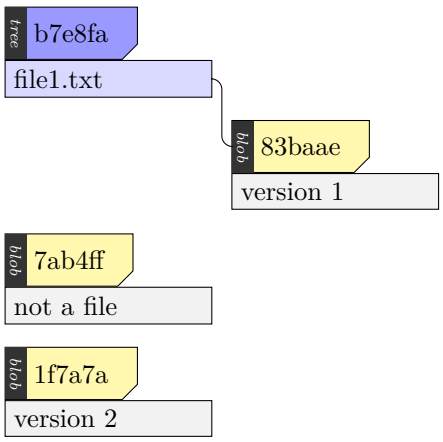

Figure 2.4: A Git 'tree' object

After the `write-tree` a new object has appeared in our repository. The hash for this object (`b7e8fac7e3e35d93d39d2fa2260868f025a9efb4`) is what

was returned from the `write-tree` command. You can check the type of this object, confirming it is a tree, and then look at its content to see that the `--cacheinfo` we used above has been captured.

bash

```
git cat-file -t b7e8
git cat-file -p b7e8
```

git cat-file -t b7e8

```
tree
```

git cat-file -p b7e8

```
100644 blob 83baae61804e65cc73a7201a7252750c76066a30    file1.txt
```

The second field of this tree record `blob` is telling us that the record refers to an object of type 'blob'. Why `blob` and not `object`? The `object` directory contains both file content (blob) and tree objects (which we will shortly see as analogous to directories in the workspace). In other words, blobs and trees are both objects. It is therefore fine to use the term 'object' when the context makes clear the type of object we are talking about (or we are talking collectively about any type of object). I will continue to use 'object' unless it is important to use a more specific type.

We can add multiple objects to our index and these can be a mix of existing repository objects and new files added from our working area.

bash

```
echo 'Another file' > another_file.txt
git update-index --add another_file.txt
git ls-files --stage
```

git ls-files --stage

```
100644 b0b9fc8f6cc2f8f110306ed7f6d1ce079541b41f 0    another_file.txt
100644 83baae61804e65cc73a7201a7252750c76066a30 0    file1.txt
```

Here we are using `update-index` directly on the file `another_file.txt`. This will create a new object in the repository holding the content of `another_file.txt` at the time this `update-index` is run and then create the entry in the index to relate the filename and the file mode to this object. We cannot use `--cacheinfo` here because the object does not exist within the repository un-

til we run the `update-index`. We need the `--add` option so that `update-index` will accept new files (files that have no existing index entry) into the index.

Some time back we created a new object containing the text '`version 2`'. This object was assigned the hash `1f7a7a472abf3dd9643fd615f6da379c4acb3e3a` when we created it with `hash-object -w`. We want to add this object to our index.

bash

```
git update-index --cacheinfo 100644
  1f7a7a472abf3dd9643fd615f6da379c4acb3e3a file1.txt
git ls-files --stage
```

git ls-files --stage

```
100644 b0b9fc8f6cc2f8f110306ed7f6d1ce079541b41f 0
  another_file.txt
100644 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a 0
  file1.txt
```

Notice that the index is modified so that the `file1.txt` entry now refers to object `1f7a7a472abf3dd9643fd615f6da379c4acb3e3a`.

Why was a new line not created in the index? Note the absence of the `--add` option. We are modifying the index entry associated with the name `file1.txt`, not adding a new entry. The index is a mapping between objects in the Git repository and files in the workspace and workspace files must be uniquely identified filename. There can only be a one to one mapping from filename to object in the index (a filename can only refer to one object).

It is fine for the index to have a one to many mapping from object to filename (one object can be referred to by many filenames). This can be illustrated by adding a second index entry referring to the object `1f7a7a472abf3dd9643fd615f6da379c4acb3e3a` but using a different filename.

bash

```
git update-index --add --cacheinfo 100644
  1f7a7a472abf3dd9643fd615f6da379c4acb3e3a filerX.txt
git ls-files --stage
```

git ls-files --stage

```
100644 b0b9fc8f6cc2f8f110306ed7f6d1ce079541b41f 0
  another_file.txt
100644 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a 0
  file1.txt
100644 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a 0
  fileX.txt
```

What does this represent?

Work through what we have learned so far. The object `1f7a7a472abf3dd9643fd615f6da379c4acb3e3` contains the data '`version` 2'. The index shows the mapping between the data and the files in the workspace. So both `file1.txt` and `fileX.txt` in the workspace are to have the same content (that from object `1f7a7a472abf3dd9643fd615f6da379c4acb3e3`

We don't really want this double mapping (interesting as it is), so we remove it from the index using the `--remove` option to the `update-index` command.

bash

```
git update-index --remove fileX.txt
git ls-files --stage
```

git ls-files --stage

```
100644 b0b9fc8f6cc2f8f110306ed7f6d1ce079541b41f 0       another_file.txt
100644 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a 0       file1.txt
```

We now create another tree object.

bash

```
git write-tree
```

So far we have created some basic blob and tree objects, but we have not yet dealt with directories. Or have we?

A directory is essentially a container holding files and other directories. Sounds familiar? The tree object we just created is a list of blobs related to file names. Can we similarly relate a directory name with a tree object and include it in another tree object?

Create a directory and a new file in that directory.

bash

```
mkdir dir1
echo 'version 1' > dir1/file11.txt
```

We now add this new file to the index.

bash

```
git update-index --add dir1/file11.txt
```

If we now look at our index we find that this has simply added an entry to the index with the path `dir1/file11.txt` rather than a simple filename. We have discovered that the index maps files by pathname rather than simply their file name. These pathnames are relative to the root of our working area.

```bash
git ls-files -s
```

```
100644 b0b9fc8f6cc2f8f110306ed7f6d1ce079541b41f 0 another_file.txt
100644 83baae61804e65cc73a7201a7252750c76066a30 0 dir1/file11.txt
100644 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a 0 file1.txt
```

2.3.1 Progress review: blobs and trees

Let's review the situation we now have.

We have some blobs in the `.git/objects` store holding various data. We have two tree objects in the `.git/objects` store (`b7e8fac7e3e35d93d39d2fa2260868f025a9efb4`) that relates `83baae` to the name `file1.txt` and `349fa0b7f3252dbe6989c2e8156803b3265a78e0` that relates `1f7a7a` to `file1.txt` and `b0b9fc` to `another_file.txt`). We have a `.git/index` file containing various mappings between blobs and filenames (which we just listed out above).

We can list all the objects in `.git/objects` using `cat-file` with the `--batch-all-objects` and `--batch-check` options.

```bash
git cat-file --batch-all-objects --batch-check
```

git cat-file --batch-all-objects --batch-check

```
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob 10
349fa0b7f3252dbe6989c2e8156803b3265a78e0 tree 81
7ab4ff63b2ea4c2c3ff89ee972bc42988a4b8472 blob 11
83baae61804e65cc73a7201a7252750c76066a30 blob 10
b0b9fc8f6cc2f8f110306ed7f6d1ce079541b41f blob 13
b7e8fac7e3e35d93d39d2fa2260868f025a9efb4 tree 37
```

We can now see what happens when we add sub-directories to our object store. Remember that our index has a new `dir1/file11.txt` path mapping so we are expecting `write-tree` to account for this in our repository.

```bash
git write-tree
git cat-file --batch-all-objects --batch-check
```

git cat-file --batch-all-objects --batch-check

```
0139f016af84acd889e2f707ef9eca2140e0222e tree 112
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob 10
337f3832b1bce2d8f364e99965c8519a3eb9dc6c tree 38
349fa0b7f3252dbe6989c2e8156803b3265a78e0 tree 81
7ab4ff63b2ea4c2c3ff89ee972bc42988a4b8472 blob 11
83baae61804e65cc73a7201a7252750c76066a30 blob 10
b0b9fc8f6cc2f8f110306ed7f6d1ce079541b41f blob 13
b7e8fac7e3e35d93d39d2fa2260868f025a9efb4 tree 37
```

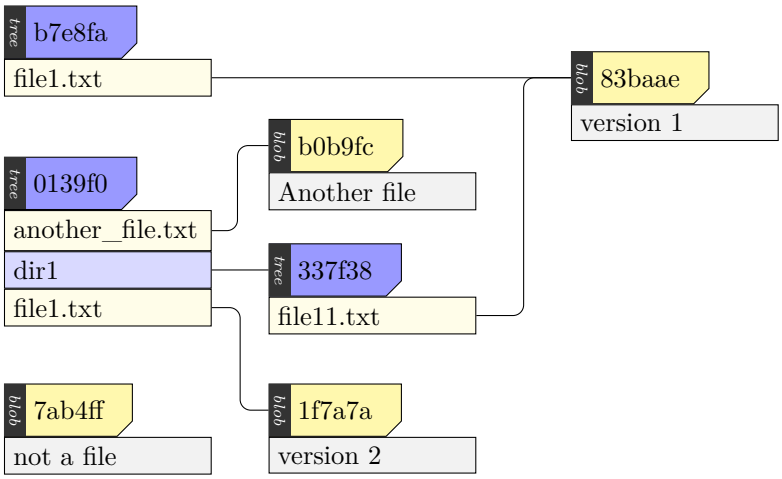

Figure 2.5: Hierarchy of tree objects

We have added two new tree objects, `337f38` and `0139f0`. Inspecting these we can see what has happened.

bash

```
git cat-file -p 337f38
git cat-file -p 0139f0
```

git cat-file -p 337f38

```
100644 blob 83baae61804e65cc73a7201a7252750c76066a30	file11.txt
```

git cat-file -p 0139f0

```
1 100644 blob b0b9fc8f6cc2f8f110306ed7f6d1ce079541b41f another_file.txt
2 040000 tree 337f3832b1bce2d8f364e99965c8519a3eb9dc6c dir1
3 100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a file1.txt
```

The first (`337f38`) represents the content of the `dir1` directory, in this instance just the mapping of `83baae` to the file name `file11.txt`.

The second (`0139f0`) represent the content of our root directory. The interesting entry being the tree object referenced on line 2 and mapped to the name `dir1`.

From this short exercise we can make a few observations.

1. The index maps blobs to file paths (not simply file names).
2. The index *does not* map tree objects.
3. Tree objects are created as required whenever a `write-tree` is executed.
4. Tree objects are mapped to names by other tree objects.
5. Tree objects form a directed graph representing a directory structure.
6. The root Tree object has no name (since names are mapped by tree objects and, by definition, the root tree object is not itself a part of a parent tree object).

We have now shown how Git stores data in blobs. Names are mapped to those blobs by tree objects. Tree objects can contain other tree objects and map them to names, allowing us to store directories[3].

Now that we can store a basic file structure it is time to consider how Git stores the history of changes to files.

2.4 commits

Watch “Creating a history with commits” on Vimeo

Tree objects effectively capture and freeze a hierarchical set of files and directories. Put another way, a tree object is a snapshot in time of a set of blob to file path mappings. This is useful to us when we want to capture a history, all we need do is capture tree objects representing the start and end of

[3]Note that we cannot create an empty tree object. This is the reason Git cannot store empty directories.

any operation and then somehow tell Git that the first snapshot precedes the second. We can now look at these two snapshots as a history of the files and directories captured by the tree objects.

We already have two snapshots we can use to start our history.

bash

```
git ls-tree -r b7e8fa
git ls-tree -r 0139f0
```

I've used the `-r` option to list the tree object recursively. This has no effect on the first tree but the second tree shows blob object `83baae` mapped to the file path `dir1/file11.txt` whereas without the `-r` option we would see only the tree object `337f38` mapped to directory `dir1` (as above).

git ls-tree -r b7e8fa

```
100644 blob 83baae61804e65cc73a7201a7252750c76066a30    file1.txt
```

git ls-tree -r 0139f0

```
100644 blob b0b9fc8f6cc2f8f110306ed7f6d1ce079541b41f    another_file.txt
100644 blob 83baae61804e65cc73a7201a7252750c76066a30    dir1/file11.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a    file1.txt
```

The first (`b7e8fa`) tree contains only `file1.txt`. In the second we have added file `another_file.txt`, the directory `dir1` and within that the file `file11.txt` (`file1.txt` has different content to that referred to in `b7e8fa`, we know this because it is mapped to a different blob (`1f7a7a`) rather than `83baae`).

So Git provides a simple mechanism for showing that tree `b7e8fa` is historically before tree `0139f0`? Yes ... and no. Although we know that we added and modified the files between creating tree objects `b7e8fa` and `0139f0` there is nothing reflecting this history. We could just as easily claim that `0139f0` was created first and then we modified `file1.txt` and removed `dir1` and it's content, the results would be the same.

To create a history we must first create a new type of object, the `commit` object. It will not surprise you that these objects are also stored under `.git/objects`.

Commit objects contain special metadata (that is data about data, in this case data about a tree object). To create a commit object we use the `commit-tree` command.

```bash
git commit-tree -m "First commit" b7e8fa
```

As with other commands that create new objects the `commit-tree` command returns the hash of the new commit object. This is the first time you will notice your commit will have a different hash to my commit object. Pause for a second and consider why this might be.

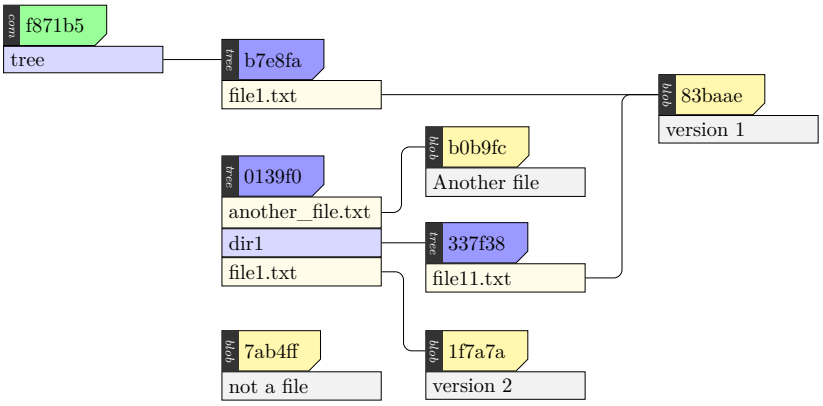

Figure 2.6: Single commit

We can now inspect this object, first confirming its type and then pretty printing it.

```bash
git cat-file -t f871b5
git cat-file -p f871b5
```

git cat-file -t f871b5

```
commit
```

git cat-file -p f871b5

```
tree b7e8fac7e3e35d93d39d2fa2260868f025a9efb4
author vagrant <vagrant@debian-10.7-amd64> 1615399633 +0000
committer vagrant <vagrant@debian-10.7-amd64> 1615399633 +0000

First commit
```

And here we see another difference between what you see and what I see. What causes this difference? After all we have, so far, started with the same setup and created the same objects in the repository. Compare closely the output of `cat-file`. At the end of the lines starting `author` and `committer` are two numbers, these are timestamps and since you and I created our commit objects at different times we have different timestamps and consequently these commit objects have different hashes.

We can demonstrate this clearly by repeating the `commit-tree` with no changes.

```bash
git commit-tree -m "First commit" b7e8fa
```

Git returns a different hash. If we compare the two commit objects (remembering that your commit objects' hashes will be different to mine!), we see they differ only in the timestamps recorded.

```bash
diff <(git cat-file -p f871b5) <(git cat-file -p e3004b)
```

We now have two commit objects, but they are not very interesting as they refer to the same tree object (and hence the same 'version 1' of `file1.txt`). Let's create some more interesting commit objects.

We previously created a tree object (`0139f0`) that captured the files `file1.txt` ('version 2'), `another_file.txt`, and `dir1/file11.txt`. We now what to create a history in which this configuration of files and directories follows from the 'version 1' `file1.txt`.

```bash
git commit-tree -m "Second commit" -p f871b5 0139f0
```

In this `commit-tree` we added the `-p` option to indicate that commit object `f871b5` is the parent of the commit object we are creating for tree object `0139f9`. As before we can examine the new commit object (in my case `0715e7`) with the `cat-file` command.

```bash
git cat-file -p 0715e7
```

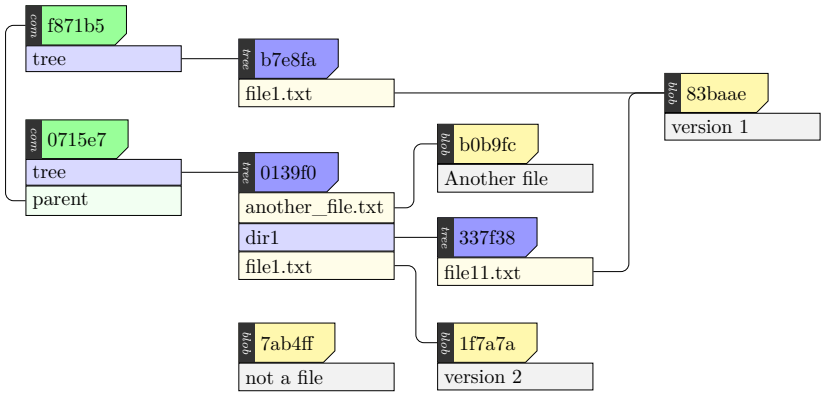

Figure 2.7: Commit with parent

```
1  tree 0139f016af84acd889e2f707ef9eca2140e0222e
2  parent f871b58596491e15ee1da91eaf0a4a6c1da3e573
3  author vagrant <vagrant@debian-10.7-amd64> 1615399872
   +0000
4  committer vagrant <vagrant@debian-10.7-amd64> 1615399872
   +0000
5
6  Second commit
```

On line 2 we see that this commit has a `parent` (`f871b5`).

Now let's quickly create one more commit. First we create a new version of `file1.txt`, then create a new tree object, and finally a new commit.

```bash
1  echo 'version 3' > file1.txt
2  git update-index file1.txt
3  git write-tree
4  git commit-tree -m "Third commit" -p 0715e7 fd97ab
```

2.4.1 Progress review: blobs, trees, and commits

Let us review the content of our objects store[4]. We have create three tree objects using the `write-tree` command. These were:

1. Version one of `file1.txt` on its own.

[4]If you want to be one of the cool kids you can point out that this structure (a tree in which each node hashes its children) is a Merkle tree.

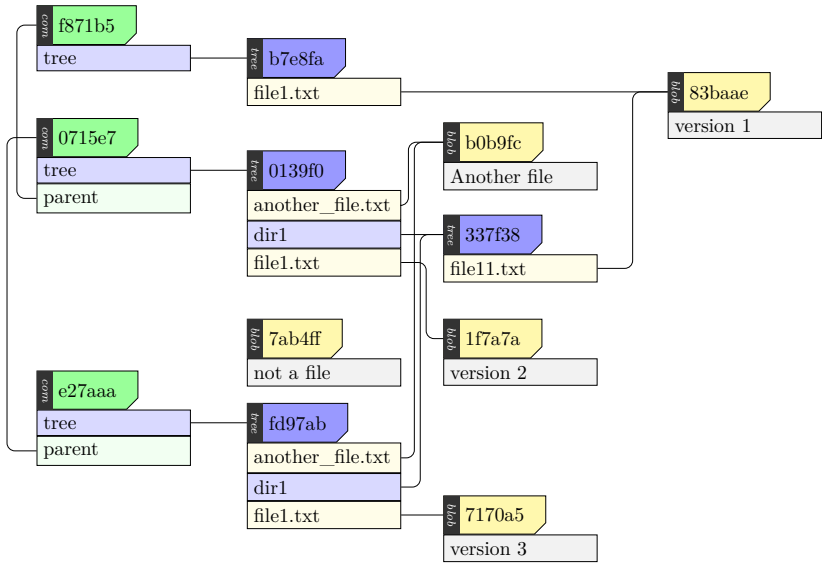

Figure 2.8: Three commit history

2. Adding `another_file.txt` alongside version one of `dir1/file11.txt` and updating `file1.txt` to version two.

3. Update `file1.txt` to version 3

To create a version chain of these three tree objects we use three `commit-tree` commands. The first commit object has no `parent` as it is the first entry, it contains the four pieces of data:

1. `tree`—the hash of the tree object to which this commit refers.

2. `author`—a record of the author's name and email (the person who write the changes in the tree object), along with the time the commit was authored

3. `committer`—a record of the committer (the user who actually executed the `commit-tree`)

4. A blank line, followed by the text of any comment we want to associate with the commit (in these example, supplied by the `-m` option to the `commit-tree` command).

Author versus Committer

Why the two entries 'author' and 'committer'?
The 'author' of a change is the individual who edited the files making up the change.
The 'committer' is the user who created the commit object.
In private use these two field normally contain the same information. The same user created the commit and makes the changes. However, suppose a user submits a change as a patch file using e-mail? That person is the author of the change but not the person who puts those changes into the Git repository. This is why there is a distinction between the 'author' and 'committer'.

The second commit specifies the first commit object (the hash returned by the first `commit-tree`). Looking at this commit object you can see one additional piece of data over the initial commit:

1. `parent`—the hash reference to parent commit object.

Finally we created a third commit object referencing the second as its parent.

The entire chain we just created can be displayed using the `log` command; the hash `e27aa` being the hash of the last (third) commit object we just created. The `--stat` option shows summary statistics of each commit and the `--patch` shows the changes to the files in each commit.

bash

```
1 git log --stat --patch e27aaa
```

```
commit e27aaa8c158e6f261f4c03aaaf173a149ad61d81
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:13:55 2021 +0000

    Third commit
---
 file1.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/file1.txt b/file1.txt
index 1f7a7a4..7170a52 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1 @@
-version 2
+version 3

commit 0715e707b906d30c9e395448ddc9e96acd89d5f7
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:11:12 2021 +0000

    Second commit
---
 another_file.txt | 1 +
 dir1/file11.txt  | 1 +
 file1.txt        | 2 +-
 3 files changed, 3 insertions(+), 1 deletion(-)

diff --git a/another_file.txt b/another_file.txt
new file mode 100644
index 0000000..b0b9fc8
--- /dev/null
+++ b/another_file.txt
@@ -0,0 +1 @@
+Another file
diff --git a/dir1/file11.txt b/dir1/file11.txt
new file mode 100644
index 0000000..83baae6
--- /dev/null
+++ b/dir1/file11.txt
@@ -0,0 +1 @@
+version 1
diff --git a/file1.txt b/file1.txt
index 83baae6..1f7a7a4 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1 @@
-version 1
+version 2

commit f871b58596491e15ee1da91eaf0a4a6c1da3e573
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:07:13 2021 +0000

    First commit
```

2.5 refs

Watch "Human readable references" on Vimeo

So far we have:

1. Created some hash objects.
2. Created some tree objects that associate file pathname and mode with one or more hash objects
3. Created some commit objects that associate metadata with tree objects and allows us to relate tree objects in a graph which is typically interpreted as a version graph where each parent is an earlier version (it should be noted that Git itself is completely unaware of this interpretation though).

So far, so good but it is still a bit cumbersome to use. For one thing we have to remember which commit object we last created so that we can use it as the parent for our next commit. We have seen this problem above, not only when using `commit-tree` but also using the `log` command where we needed to know the hash of the most recent commit object in our history.

`refs` (or 'references') to the rescue. A ref is a more human readable way to refer a commit object hash.

bash

```
tree -a .git/refs
```

tree -a .git/refs

```
.git/refs
├── heads
└── tags

2 directories, 0 files
```

The `refs` directory contains two sub-directories:

- `heads`—contains references to the head, or latest, commit object we want to name.
- `tags`—contains references to any object we want to give a human readable name.

We can set the head of our `master` branch (the default branch[5] on which Git works, more on branches later) to our latest commit object:

```bash
echo "e27aaa8c158e6f261f4c03aaaf173a149ad61d81" > .git/refs/heads/master
git log --stat
```

We have to use the full hash when writing to the `.git/refs/heads/master` file.

```git log --stat
commit e27aaa8c158e6f261f4c03aaaf173a149ad61d81 (HEAD -> master)
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:13:55 2021 +0000

    Third commit

file1.txt | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)

commit 0715e707b906d30c9e395448ddc9e96acd89d5f7
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:11:12 2021 +0000

    Second commit

 another_file.txt | 1 +
 dir1/file11.txt  | 1 +
 file1.txt        | 2 +-
 3 files changed, 3 insertions(+), 1 deletion(-)

commit f871b58596491e15ee1da91eaf0a4a6c1da3e573
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:07:13 2021 +0000

    First commit

 file1.txt | 1 +
 1 file changed, 1 insertion(+)
```

[5] I tend to freely use 'default branch' to mean the 'current default branch' and 'the branch Git will use absent any other branches'. This is lazy but I think context makes clear which is implied.

Issuing the `log` command without specifying the exact commit we are interested in causes Git to look up the `refs` entry of our current branch (actually it looks in `.git/HEAD` for the 'latest' commit and since we have not moved from the default branch this will be `master`, we look at `.git/HEAD` again in §2.6.1.)

bash

```
cat .git/HEAD
cat .git/refs/heads/master
```

cat .git/HEAD

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

cat .git/refs/head/master

```
e27aaa8c158e6f261f4c03aaaf173a149ad61d81
```

The `master` file in the `.git/refs/heads` directory contains the hash of the commit object we want to call the 'head' of our master branch.

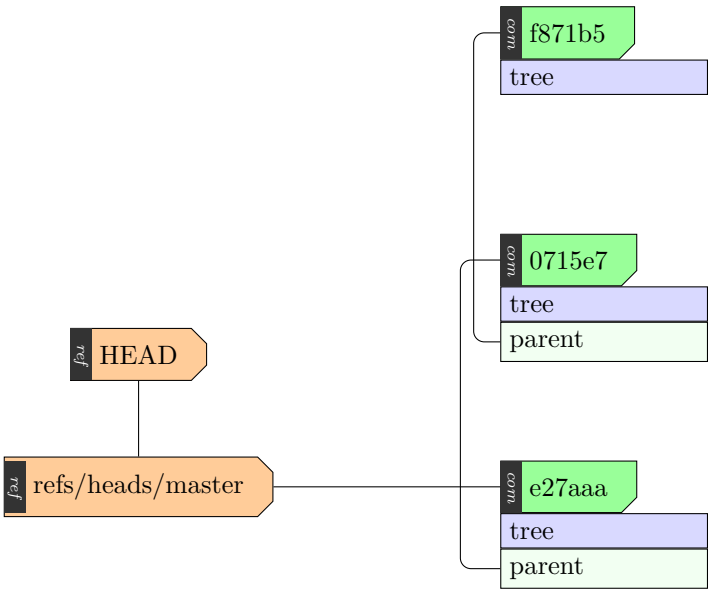

Figure 2.9: HEAD dereferencing

Editing `refs` files directly is not ideal (not least because we can't abbreviate the hash) so Git provides the `update-ref` command.

bash

```
git update-ref refs/heads/master 0715e7
git log --stat
```

git log --stat

```
commit 0715e707b906d30c9e395448ddc9e96acd89d5f7
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:11:12 2021 +0000

    Second commit

 another_file.txt | 1 +
 dir1/file11.txt  | 1 +
 file1.txt        | 2 +-
 3 files changed, 3 insertions(+), 1 deletion(-)

commit f871b58596491e15ee1da91eaf0a4a6c1da3e573
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:07:13 2021 +0000

    First commit

 file1.txt | 1 +
 1 file changed, 1 insertion(+)
```

After we update the reference to the penultimate commit in our repository (`0715e7`) we curtail the log at that point and Git will treat that commit as the latest on the master branch.

We can easily restore the head of master using the `update-ref` command.

bash

```
git update-ref refs/heads/master e27aaa
git log --stat
```

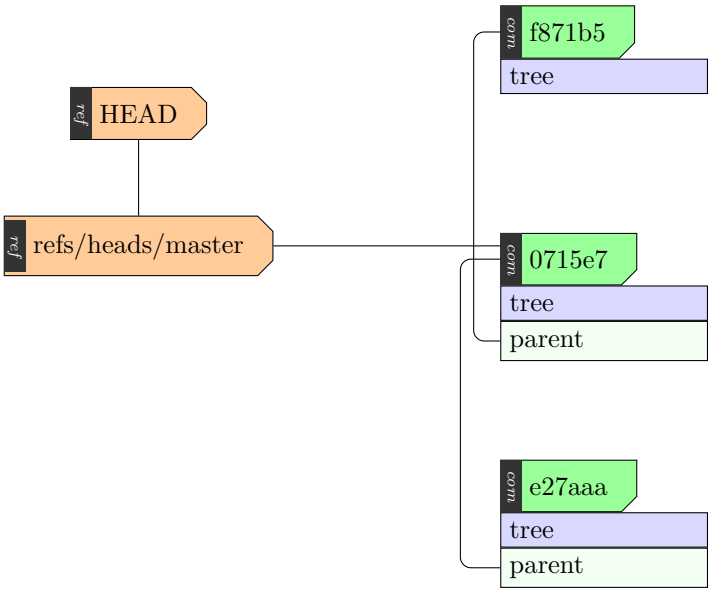

Figure 2.10: Penultimate change

git log --stat

```
commit e27aaa8c158e6f261f4c03aaaf173a149ad61d81 (HEAD -> master)
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:13:55 2021 +0000

    Third commit

file1.txt | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)

commit 0715e707b906d30c9e395448ddc9e96acd89d5f7
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:11:12 2021 +0000

    Second commit

 another_file.txt | 1 +
 dir1/file11.txt  | 1 +
 file1.txt        | 2 +-
 3 files changed, 3 insertions(+), 1 deletion(-)

commit f871b58596491e15ee1da91eaf0a4a6c1da3e573
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:07:13 2021 +0000

    First commit

 file1.txt | 1 +
 1 file changed, 1 insertion(+)
```

The `update-ref` command is doing several things for us. Firstly, it allows us to use short hashes (phew!), it checks that the hash we provide is valid too. Secondly you may have notices new directories and files appearing in the repository.

bash

```
tree -a
```

tree -a (truncated)

```
└── .git
    ├── branches
    ├── config
    ├── description
    ├── HEAD
    ├── hooks
    ├── index
    ├── info
    │   └── exclude
    ├── logs
    │   ├── HEAD
    │   └── refs
    │       └── heads
    │           └── master
```

The new `logs` directory contains two new files; `HEAD` and `refs/heads/master`. These contain a record of each time we modify a reference using `update-ref`. Each log entry records the old hash, the new hash, the user who made the change, and a time stamp for when the change was made. These 'logs of ref changes' can be viewed (and manipulated) using the `reflog` command.

bash

```
git reflog
```

git reflog

```
e27aaa8 (HEAD -> master) HEAD@{0}:
0715e70 HEAD@{1}:
```

This shows the history of changes we made to the master branch `refs/heads/master` using `update-ref`. As with the `log` command, by default, more recent changes are shown first (reverse chronological order).

The output may look like gibberish but it's actually simple enough. Let's break down the first line.

- `e27aaa8`—This is the new hash we set.

- (`HEAD -> master`)—This tells us that this entry is about the `HEAD` reference and this currently refers to the `master` branch. (Actually, `HEAD` is an indirect reference that points us to the 'latest commit on the active branch', more on this shortly.)
- `HEAD@\{0\}`—This is a commit reference, specifically is says that we are referring to the 'zeroth' (latest) change relative to the 'HEAD' reference.

Okay, we should now be able to read the simpler second line with ease.

- `0715e70`—The hash we set when this change was made.
- `HEAD@\{1\}`—This line refers to the 'first' change to `HEAD`, counting back from the current value (`HEAD@\{0\}`).

2.5.1 Remote refs

There is one more type of `ref` we need to discuss, the 'remote refs'. These are read only refs in the sense that one does not manipulate them directly but they are maintained through interaction with remote repositories. As these have such a specialised use I'm going to leave a complete discussion to §5.1, after we have discussed working with multiple repositories in Chapter 5.

2.6 References (branches and tags)

Watch "Branches and tags" on Vimeo

Using the head refs `.git/refs/heads` we can create named branches. In fact we have done so already, `.git/refs/heads/master` holds the reference to the latest commit (head) of the master branch.

There is nothing special about the master branch other than convention and that Git treats this as the default branch name in a new repository[6].

We can create a new branch very easily, we just create a new `.git/refs/heads` entry.

bash

```
git update-ref refs/heads/test_branch 0715e7
git log --stat test_branch
git log --stat
```

[6]In mid-2020 Git 2.8.0 provided the ability to change the default branch name (using the configuration setting `init.defaultBranch`). In October 2020 GitHub started using `main` rather than `master` in new repositories, GitLab announced a similar change in March 2021. This in response to sensitivity about the use of 'master' in all forms due to its tangential association with slavery. Etymology is not the strong suit of the over-sensitive. Thankfully the use of `master` is not forbidden so this change can be largely ignored.

```
git log --stat test\char `_branch
```

```
commit 0715e707b906d30c9e395448ddc9e96acd89d5f7
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:11:12 2021 +0000

    Second commit

 another_file.txt | 1 +
 dir1/file11.txt  | 1 +
 file1.txt        | 2 +-
 3 files changed, 3 insertions(+), 1 deletion(-)

commit f871b58596491e15ee1da91eaf0a4a6c1da3e573
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:07:13 2021 +0000

    First commit

 file1.txt | 1 +
 1 file changed, 1 insertion(+)
```

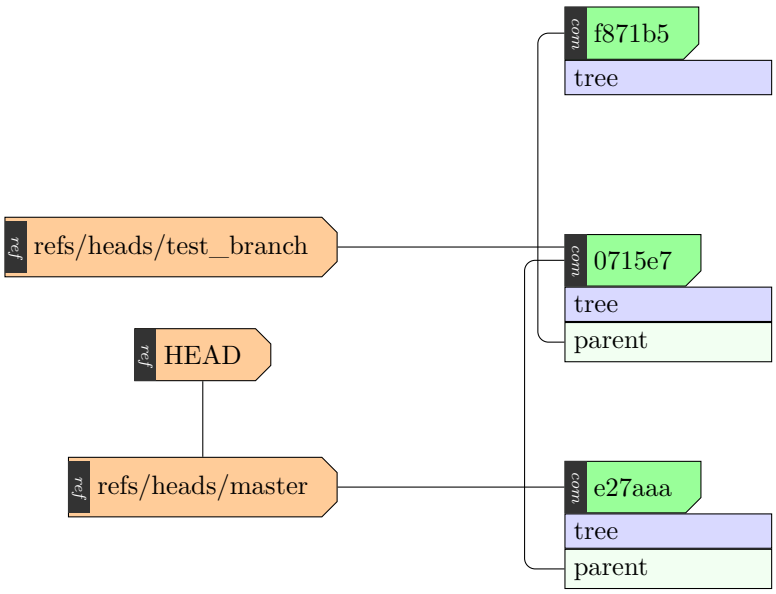

Figure 2.11: New `test_branch`

We create the new branch named `test_branch` from the second commit by creating the new `refs/heads/test_branch`. Now when we log that branch

we see only the first and second commit, while logging the current default ('master') we see all three commits.

git log --stat

```
commit e27aaa8c158e6f261f4c03aaaf173a149ad61d81 (HEAD -> master)
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:13:55 2021 +0000

    Third commit

file1.txt | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)

commit 0715e707b906d30c9e395448ddc9e96acd89d5f7
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:11:12 2021 +0000

    Second commit

 another_file.txt | 1 +
 dir1/file11.txt  | 1 +
 file1.txt        | 2 +-
 3 files changed, 3 insertions(+), 1 deletion(-)

commit f871b58596491e15ee1da91eaf0a4a6c1da3e573
Author: vagrant <vagrant@debian-10.7-amd64>
Date:   Wed Mar 10 18:07:13 2021 +0000

    First commit

 file1.txt | 1 +
 1 file changed, 1 insertion(+)
```

2.6.1 HEAD

How does Git know that our currently active branch is master? There is a special file in `.git` called `HEAD` (we saw this in §2.5) that tells Git where the current default head commit is located. The `HEAD` therefore indicates which commit object will be the parent to the next commit object created. In this way Git will add the next commit object to the end of the currently active branch.

bash

```
cat .git/HEAD
cat .git/refs/heads/master
```

cat .git/HEAD

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

cat .git/refs/heads/master

```
e27aaa8c158e6f261f4c03aaaf173a149ad61d81
```

Normally, as in this case, `HEAD` is an indirect reference to one of the `refs/heads` files, which is in turn a reference to the actual commit hash that we are to use as the current head (the current situation is illustrated in Figure 2.9).

We can change the branch to which `HEAD` refers (and consequently the branch on which we are working).

bash

```
git log --oneline
echo "ref: refs/heads/test_branch" > .git/HEAD
git log --oneline
```

I've switched to using the `--oneline` option on `log` to keep the output short (I don't think outputting the entire `--stat` output each time is adding any value here.).

git log --oneline

```
e27aaa8 (HEAD -> master) Third commit
0715e70 (test_branch) Second commit
f871b59 First commit
```

git log --oneline

```
0715e70 (HEAD -> test_branch) Second commit
f871b59 First commit
```

Before the change `log` outputs the master branch (`HEAD -> master`), after changing the content of `.git/HEAD` `log` outputs the `test_branch` (`HEAD -> test_branch`), we have effectively changed the default branch by changing the `ref` to which `HEAD` refers.

As with changing `refs/heads` files, manually editing the `HEAD` file is not ideal and Git provides the `symbolic-ref` command to make this safer.

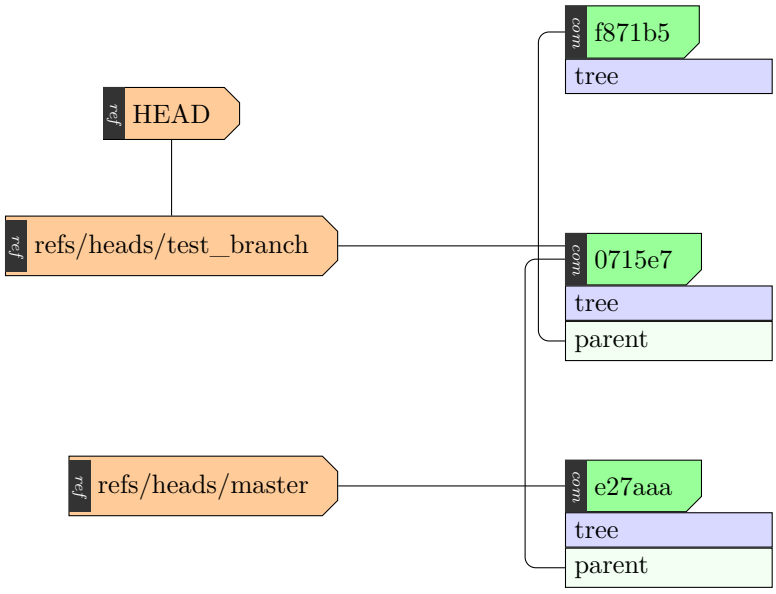

Figure 2.12: Shifting to the `test_branch`

bash

```
git symbolic-ref HEAD
git log --oneline
git symbolic-ref HEAD refs/heads/master
git symbolic-ref HEAD
git log --oneline
```

git symbolic-ref HEAD

```
refs/heads/test_branch
```

git log --oneline

```
0715e70 (HEAD -> test_branch) Second commit
f871b59 First commit
```

git symbolic-ref HEAD

```
refs/heads/master
```

git log --oneline

```
e27aaa8 (HEAD -> master) Third commit
0715e70 (test_branch) Second commit
f871b59 First commit
```

First we view the current value of the symbolic reference `HEAD`, then we change that reference; note that we specify the path of the actual ref file (`refs/head/master`).

2.6.1.1 Detached HEAD

You may occasionally encounter a 'detached HEAD' error. This seems to cause much confusion online but is actually a very simple issue.

In some circumstances the `HEAD` symbolic reference will contain a hash value directly (i.e. not a reference to one of the `refs/heads`). This can arise for a number of reasons, among which the most common are:

- `checkout` of a commit directly using its hash
- `checkout` of a `remote` (more on these later)
- `checkout` of a tag (which we look at next).

We examine `checkout` in detail in §**??**, in the following it is simply a way to ask Git to 'get' a commit object's content and, more importantly, update the `HEAD` file.

bash

```
git checkout f871b5
git symbolic-ref HEAD
cat .git/HEAD
```

git checkout f871b5

```
1  Note: checking out 'c1bf'.
2
3  You are in 'detached HEAD' state. You can look around, make experimental
4  changes and commit them, and you can discard any commits you make in this
5  state without impacting any branches by performing another checkout.
6
7  If you want to create a new branch to retain commits you create, you may
8  do so (now or later) by using -b with the checkout command again. Example:
9
10   git checkout -b <new-branch-name>
11
12 HEAD is now at f871b59 First commit
```

Line 3 announces that we are in the 'detached HEAD' state.

git symbolic-ref HEAD

```
1  fatal: ref HEAD is not a symbolic ref
```

We cannot look at the 'symbolic-ref' because it is no longer there.

cat .git/HEAD

```
1  f871b597ef5160ab19556e42e8a5264d092ad2bc
```

In fact the `.git/HEAD` file contains only the hash of the commit we checked out.

bash

```
1  git checkout 0715e70
2  git symbolic-ref HEAD
```

git checkout 0715e70

```
1  Previous HEAD position was f871b59 First commit
2  HEAD is now at 0715e70 Second commit
```

git symbolic-ref HEAD

```
1  fatal: ref HEAD is not a symbolic ref
```

If we checkout the second commit directly (using its hash) we are simply informed that we updated the hash and attempting to examine the `symbolic-ref` still results in an error.

bash

```
git checkout test_branch
git symbolic-ref HEAD
```

git checkout test\char `_branch

```
Switched to branch 'test_branch'
```

git symbolic-ref HEAD

```
refs/heads/test_branch
```

Checking out the `test_branch` (which is the same commit but now referred to by the `refs/heads/test_branch`) we are 'switched' to that branch's `HEAD` (the very same commit is being checked out, but the reference is now a `symbolic-ref`).

2.6.2 tags

It would be very useful to have a method of recalling a commit by name, for example when we release versions of our project it would be good to be able to say "this commit is version 1.0 of my project". Fortunately Git has tag references for just this purpose.

Tag references come in two types:

1. Lightweight—these tags are similar to the symbolic references used above, they are simple records in the `.git/refs/tags` directory that point to specific commit objects (much as we have just seen `.git/refs/heads` do). Lightweight tags are typically private temporary names assigned by the user.

2. Annotated—these are a new type of object, a `tag` object, that contains some metadata associated with the tag. An entry is then made in the `.git/refs/tags` directory referencing this tag object. Annotated tags are used for more public permanent tags, such as release commits.

To create a new lightweight tag we use `update-ref`.

bash

```
git update-ref refs/tags/v1.0 f871b5
cat .git/refs/tags/v1.0
```

cat .git/refs/tags/v1.0

```
f871b58596491e15ee1da91eaf0a4a6c1da3e573
```

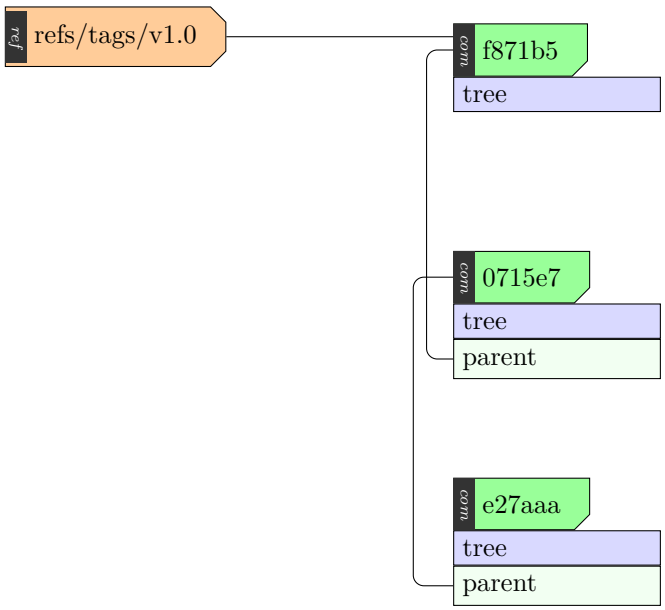

Figure 2.13: Lightweight tag

We can now refer to the commit object with hash `f871b5` using the tag name `v1.0`. These lightweight tags are useful to assign 'human readable' names to Git objects we may be interested in, but we can create an annotated tag that includes additional information.

bash

```
git tag -a v2.0 0715e7 -m "The second version"
cat .git/refs/tags/v2.0
git cat-file -t v2.0
git cat-file -p v2.0
```

cat .git/refs/tags/v2.0

```
a7edafabd57c0a3dc7788d021359083ae31d3826
```

git cat-file -t v2.0

```
tag
```

git cat-file -p v2.0

```
object 0715e707b906d30c9e395448ddc9e96acd89d5f7
type commit
tag v2.0
tagger vagrant <vagrant@debian-10.7-amd64> 1616259961 +0000

The second version
```

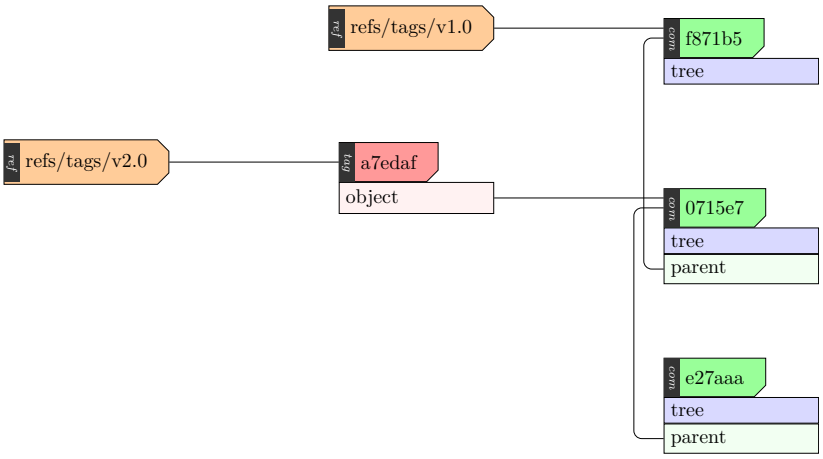

Figure 2.14: Annotated tag

Here we use the `git tag` command with the `-a` (for annotate) option to tag commit object `ac21f9` and add a comment with the `-m` option[7]. This creates a new object of type `tag`. Unlike other commands that we have seen that create objects, the `tag` command does not return the new object's hash. This is not a problem as the tag is now a proxy for that tag object's hash. We can specify either the hash or the new tag to the `cat-file` to examine the new tag object. Looking inside that tag object we see that it is referencing the object `ac21f9` (the one we tagged), this object is a commit object and the tag object is for tag `v2.0`. The last text block is the comment provided in the `-m` option.

The fact that the tag tracks the type of the object being tagged should be a clue that we can tag any object we like.

bash

```
git tag -a Meta v2.0 -m "Meta tagging dude"
git cat-file -p Meta
```

[7]The `-a` option is implied when `-m` is specified without `-a`,`-s` or `-u`, see `man git-tag(1)`.

git cat-file -p Meta

```
object a7edafabd57c0a3dc7788d021359083ae31d3826
type tag
tag Meta
tagger vagrant <vagrant@debian-10.7-amd64> 1616260030 +0000

Meta tagging dude
```

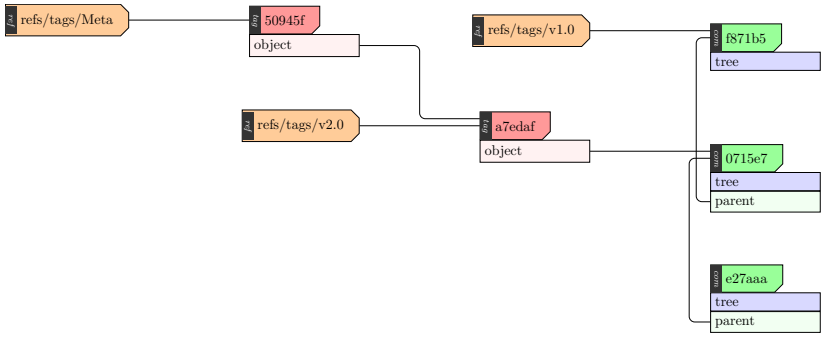

Figure 2.15: Annotated tag of a tag

Here we have created a tag (`Meta`) of a tag (`v2.0`). What is more Git will do what you might expect, it follow this meta-tag down until an object of the type expected by the command is found.

bash

```
git checkout master
git log --oneline
git log --oneline Meta
```

git checkout master

```
Switched to branch 'master'
```

git log --oneline

```
d17958c (HEAD -> master) Third commit
0715e70 (tag: v2.0, tag: Meta, test_branch) Second commit
f871b58 (tag: v1.0) First commit
```

Notice that the tags associate with each commit object are also shown.

git log --oneline Meta

```
0715e70 (tag: v2.0, tag: Meta, test_branch) Second commit
f871b58 (tag: v1.0) First commit
```

Switching back to the master branch we can see the log history from the master `HEAD` contains three commits. Specifying the `Meta` tag as the revision from which we want to log we see only the two commits from `Meta`—even though `Meta` is actually a tag object (`50945f`) that refers to a tag object (`a7edaf`) that finally refers to a commit object (`0715e7`).

Chapter 3

The Index

In Chapter 2 we introduced the index (held in the `.git/index` file[1]) and showed the basic steps for adding files to the Git repository. This chapter discusses the index in more detail.

3.1 The Git Triumvirate: the repo, the index, and the work dir

Figure 3.1 and Figure 3.2 show the relationships between the three main 'parts' of Git.

The index contains a list relating hash and mode to file paths.

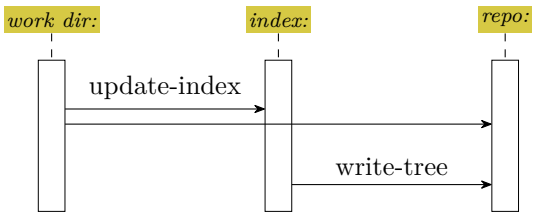

Figure 3.1: The Git Triumvirate: in-flow

[1] We shall see this, as many things in these early chapters, a partial truth. The index may be split by Git when dealing with large numbers of filename/objects mappings.

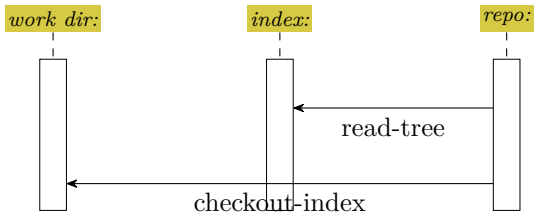

Figure 3.2: The Git Triumvirate: out-flow

Author Note

Use the diff command to illustrate how the index works.
the index 'marks' all files considered 'tracked' in the workspace.
files in the workspace with no entry in the index are 'untracked'
files in the index with hashes different to those in the workspace are 'modified'
the index refers to 'staged' files. These have hashes (and blobs in the objects store) that are NOT a part of the last commit tree (referened by HEAD).
It is therefore possible to have a single file both 'modified' (its workspace content differs from that referred to by the index) AND 'staged' (the ndex refers to an blob that is not the one referenced in the HEAD commit)

Chapter 4

Basic Local Workflow

We have investigated most of the key internal plumbing of Git in Chapter 2 and Chapter 3, in this chapter we look at a basic local Git workflow a developer might use and show how regular Git commands are interpreted into the plumbing actions we've seen so far.

4.1 Create a new repository

In Chapter 2 and Chapter 3 we looked at the details of local Git repositories and there interaction with a working area. In this chapter we look at the commands you are more likely to use day-to-day when using Git. The commands used in this chapter are so called 'porcelain' commands

4.2 Add a new file

4.2.1 How status leverages the index

l

4.3 Change a file

4.4 Add some directories and files

Chapter 5

Working With Other Repositories

So far we have focused on Git as a local tool but it's real power comes from being able to collaborate with others. In this chapter we look at how Git works with other repositories.

5.1 Remote refs