Git Internals

Git's design philosophy

Plumbing

Plumbing in Git are the low-level commands that perform the fundamental tasks.
These commands are not intended to be user-friendly or to be used directly in day-to-day version control operations by typical end users.
However, Plumbing commands give users the ability to manipulate the internals of Git's data structures and processes directly, such as objects, refs, index files, and more.

e.g. `git hash-object`, `git update-index`, and `git cat-file`.

Porcelain

Porcelain are the high-level commands that provide a user-friendly interface for the most common version control tasks.
These commands are designed to be used directly by end users and hide the complexity of Git's internal mechanisms.
They provide convenient and intuitive ways to achieve tasks such as committing changes, merging branches, and viewing history.

e.g. `git commit`, `git merge`, and `git log`.

Git actually violates a common software engineering rule, which is "Just show the porcelain to the outside world. Don't expose your plumbing."
Instead, Git wants to have two levels. Even at the low-level, it wants to expose its basic model for how Git works if the user is willing to understand how it works.

Git's modification principle

1. `.git/objects`

Inside a Git repository, the `.git/objects` directory contains the actual files and changes stored in your repository. Each file and change is stored as an object with a unique SHA-1 hash. These objects are read-only in the sense that they are not meant to be manually edited. Git manages these objects itself.

Git objects in the `.git/objects` directory are immutable, meaning once they are created, they do not change. If you were to somehow change them, it could corrupt the repository. This immutability is a core feature of Git's design, ensuring data integrity and history consistency.

2. Hard Links

Recall: Hard links are filesystem features that allow two or more filenames to refer to the same underlying inode (the filesystem's internal representation of a file). Unlike a regular file copy that duplicates the file's content, a hard link creates another entry for the same content without using additional disk space for the file's data.

While Git's internal objects are not meant to be changed, and Git uses certain optimizations like hard links when cloning locally.

> **Why is cloning so fast on a local machine?**
>
> If you explore your files with `ls -l`, you'll notice that the link count of some files that were cloned locally are greater than expected.
> When clones a repository locally, Git creates **hard links** to the objects in the `.git/objects` directory (the object database) of the source repository instead of copying them.
> More specifically, it makes hard links to read-only files. It knows it's not going to cause a problem because people cannot modify them, so they're safe to share.

A consequence is that most of the stuff in `.git` is stuff you cannot change. If you could, you can no longer clone it. If you try to be a troublemaker and use `chmod` and try editing such files, it could mess up other repositories using it (if that's even allowed at all).

3. Working Directory

When you clone a repository, Git checks out a working directory. This is where you can see and edit your files normally. When you make changes, add, and commit, Git creates new objects to represent these changes. Your modifications do not directly alter the original objects. Instead, Git will creates new objects and updates references, like branches and HEAD, to point to the new commits.

The `.git` Directory

The entire state of the repository is encoded in the special `.git` directory.

Note that there's an issue of compatibility. When a new version of Git comes out, it needs to be able to work with old repositories (to maintain **backwards compatibility**). The converse is not always true; repositories created by recent versions of Git may not necessarily work with older versions of Git. Thus, there are some files in the `.git` folder that are archaic.

Some important files and directories in the `.git` subdirectory:

- `branches` - all the branches
- `config` - repository-specific configuration
- `description` - used for GitWeb, an attempt to put Git on the web
- `HEAD` - where the current branch is
- `hooks/*` - executable scripts that Git will invoke at certain "pressure points" (important triggers, like making a commit). By default, there are no working hooks; default ones all end with `.sample`, which illustrate what you might want to put in such hooks
- `index` - a list of planned changes for the next commit. This is in binary data
- `info/exclude` - addition to `.gitignore`
- `logs` - keeps track of where the branches have been (histories of branch tip locations)
- `objects` - where the actual "repository" is, where the object database is stored
- `refs` - where the branch tips and tags are (where all the "pointers" in the repository are)
- `packed-refs` - optimized version of `refs`

> **Emacs Hooks ASIDE:** Because `git clone` does NOT copy hooks, fresh local copies of a repository always have no functioning hooks. Emacs maintainers went around this by preparing a script in the

> Emacs source code called `autogen.sh`. This script creates a bunch of Git hooks that tailor the repository to be the way the Emacs developers want it to be tailored. This is a nice "gatekeeper" approach that ensures your development is relatively clean.

Comparison to Filesystems

It seems that Git uses a collection of files to represent a repository (and the index). In reality, Git actually uses a combination of secondary storage and RAM (cache).

A local Git repository and the index are made up of **objects** and some other auxiliary files. Git objects are like a tree of files in the filesystem.

REMINDER: Every file has a unique index, namely the **inode number**, which you can see with `ls -i`.

Analogously, SHA-1 checksums for Git objects have the role that inode numbers have in filesystems. They are comparable to pointers in C/C++, values that uniquely identify the actual objects they reference.

DIFFERENCE: Files in the filesystem can be mutable. inode numbers thus exist *independently* of the contents of their files. However, checksums uniquely identify objects *by their content*. Therefore, you **cannot** change objects' contents.

SIMILARITY: Both are directed acyclic graphs (DAGs). In the filesystem, it's guaranteed by the OS that you cannot have cycles. For Git objects, you cannot create a cycle because you can only *add* to history, not change it.

Git Objects

Git is a content-addressable filesystem, which means that at the core of Git is a simple key-value data store(kind of a dictionary). 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.

Git Object Types

1. Blob

   Represents any bytes sequence, like the content of a regular file in a filesystem. (only the content of a file, but not filename)
   It does not contain any metadata about the file itself (like its name, mode, or timestamp). Metadata is stored in tree objects.

2. Tree

   Represents a directory.
   It contains a list of filenames and their corresponding metadata, including file modes, types (blob or tree), and SHA-1 hashes.
   A tree object can reference other tree objects (subdirectories) and blob objects (files).

3. Commit

   Represents a snapshot of the repository at a certain point in time.
   It points to a tree object that represents the top-level directory of the repository for that snapshot. Additionally, it contains metadata such as the author, committer, commit message, and one or more parent commits (the commit(s) that directly preceded this commit).

4. Tag

   Used to mark specific points in a repository's history as important, typically used for releases. There are two types of tags in Git:

   - Annotated(注释)
   - Lightweight

Common Commands for Git Objects

- `git hash-object`
  - `-t <type>` specifies the type of object, default type is blob (represents the file content).
  - `-w` stands for write, which tells the command to not simply return the key, but to write that object to the database
  - `--stdin` tells the command to get the content from the stdin
  - `file_path` path to the file for which you want to calculate the SHA-1 hash
- `git cat-file -p hash_code` - cat(output) the content associated with passed in hash_code
  - `-p` makes it recognizes the type of passed in argument first
  - `-t` return the type of passed in hash_code
- `git cat-file -p master^{tree}` - print the tree that the **master branch** is pointing to, with each nodes' associated mode, type and filename.
- `git update-index --add --cacheinfo mode SHA-1 file_name`
- `git write-tree`
- `git read-tree --prefix=bak SHA-1`

1. `Blob` Object

Creating Objects

Generate a checksum from string content:

```
$ echo 'Arma virumque cano.' | git hash-object -w --stdin
24b390b0e3489b71977f5c7242a4679287349242
```

git hash-object takes the content you handed to it and merely return the unique key (its SHA-1 hash) that would be used to store it in your Git database(`.git/objects`).

`SHA-1 hash` is a 40-character checksum hash standed for a unique identifier for a Git object, which represents a key to the database of the content you just store.

You can also supply a filename as a positional argument instead of using `--stdin`.

```
$ echo 'Hello, World!' > example.txt
$ git hash-object -w example.txt
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

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

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. Such hash_id is stored in the `.git/objects` directory.
The subdirectory is named with the first 2 characters of the SHA-1, and the filename is the remaining 38 characters.

Computing the checksum *and* writing it to the repository:

```
$ echo 'Arma virumque cano.' | git hash-object -w --stdin
24b390b0e3489b71977f5c7242a4679287349242
$ # This object now exists in the filesystem
$ ls -l .git/objects/24/b390b0e3489b71977f5c7242a4679287349242
-r--r--r-- 1 vinlin 197609 36 Nov 22 03:46
.git/objects/24/b390b0e3489b71977f5c7242a4679287349242
```

Notice that the file has 444 permissions. *No one* is allowed to write and exexute to this file.

But there's always a troublemaker!

```
chmod u+w .git/objects/24/b390b0e3489b71977f5c7242a4679287349242
```

You're technically *allowed* to do this, but in doing so, you're violating an invariant that Git trusts in order to properly function, so live you with your consequences I guess.

Examining Objects

Once you have content in your object database, you can examine that content by decoding the hash of the created Git object:

```
$ git cat-file -p 24b390b0e3489b71977f5c7242a4679287349242
Arma virumque cano.
```

You can check the *type* of the file with:

```
$ # !$ is shorthand for the last arg of prev cmd!
$ git cat-file -t !$
blob
```

We can add some contents with different versions to Git and pull them back out again (version control).

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
```

The 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
```

Now, you can use Git to retrieve, from the object database, either the first version or the second version you saved:

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

p.s. in the Blob object, you aren't storing the filename in your system—just the content.

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

```
$ git cat-file -t 83baae61804e65cc73a7201a7252750c76066a30
blob
```

2. Tree Object

Tree, which solves the problem of storing the filename and also allows us to store a group of files together. 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, there's a project where the most-recent tree looks something like:

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

`master^{tree}` specifies the tree object that is pointed to by the last commit on your **master** branch (recall `HEAD^`).
Notice that the `lib` subdirectory isn't a blob but a pointer to another sub-tree:

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

Examining each output, we see 4 columns:

```
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README.md
(mode)(type) (SHA-1 checksum hash)                        (name)
```

- `mode` - based on the octal digits, the last three of which represent the Linux permissions of the file.
  - (`644` means `rw-r--r--`)
  - (`000` specifies a symbolic link)
- `type`
- `SHA-1 checksum hash` of the *referred* object. After all, a tree just represents a pointer to a bunch of other objects.
- `name` - filename or dir_name

3. `Commit` Object

From above, you now have two trees that represent the different snapshots of your project that you want to track, but the earlier problem remains: you must remember all two 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.

Every commit object points to two things. First is the tree the commit represents. The second is the parent commit object(s).

```
$ git cat-file -p master
tree aa3ca55b785ab21cfbbca0a89843151d389dcac8
parent 65422241d84b3087142adcf0906b5f86e69230e3
```

```
author Vincent Lin <vinlin24@outlook.com> 1668666720 -0800
committer Vincent Lin <vinlin24@outlook.com> 1668666720 -0800

Add 11/16 lecture notes
```

We see that every commit object contains information about its parent, author and committer.

You can think of `git log` (a *porcelain* command) as pretty-printing what `cat-file -p REF` (a *plumbing* command) tells us.

Three levels going on with every commit object:

```
(commit object)...
       |
       |                 +---------> (other objects)
       v                 |
(commit object) --> (tree object) --> (other objects)
       |                 |
       |                 +---------> (other objects)
       v
(commit object)...
```

Every commit points to a different tree, but the tree can share the objects they reference. When you make a commit for small changes, you don't have to rebuild a bunch of objects. Unchanged objects are *reused*.

However, because changing a file would update the tree containing it, changing a deeply nested file would require new tree objects all the way up to the project root for the new commit.

If a file is extremely large, Git can store a *diff* instead of a full copy and just remember how to restore the full content when the blob is needed.

Form for Blobs

Suppose a blob represents some byte stream `B`, say 47 bytes long. The form it is stored in has a header that tells the type of the object, some control information like how large it is, and then the actual content:

```
+-+-+-+-+-+-+-+-+--+--------------------------------+
|b|l|o|b| |4|7|\0|<--------------B--------------->|
+-+-+-+-+-+-+-+-+--+--------------------------------+
```

This string is then compressed using zlib, and that's the form it is stored in on the file system.

This string is also what is fed to the SHA-1 checksum algorithm.

> **ASIDE:** Mathematicians in the past decade have been working to try and crack SHA-1 and have succeeded for the most part. SHA-1 is no longer "reliable", but the chance of a *random* collision (not one with file contents engineered to hash a collision) is still astronomically small.

The `.git/objects` Directory

`objects` is the most important folder. It contains the commit objects, commit history, etc.

Objects are stored two levels deep. The object is named with its 40 hex-digit SHA-1 hash value. The first two digits are the name of the subdirectory it resides in, and the remaining 38 digits are the file name. For example, an object with hash value `24b390b0e3489b71977f5c7242a4679287349242` would be stored at `.git/objects/24/b390b0e3489b71977f5c7242a4679287349242`.

Most Git objects are actually zlib-compressed, so when you directly open a Git object file (such as a commit, blob, or tree object) with a text editor like Vim, you're likely to see a mix of readable text and gibberish:

```
$ vim .git/objects/24/b390b0e3489b71977f5c7242a4679287349242
```

Inside the Vim Editor, a sequence of gibberish shows out.

```
x^AKÊÉOR02`p,ÊMT(Ë,*Í-,MUHNÌË×ã^B^@<85>!        :
```

This happens because Git objects stored in the `.git/objects` directory are compressed using **zlib**, a data compression library. The content of these files is not plain text but rather binary data that has been compressed to save space.

You can read the contents of Git objects in Python by decompressing (decoding) it to the format of `utf-8` with the `zlib` standard library module:

`decompress.py`

```
import zlib

object_path = '.git/objects/24/b390b0e3489b71977f5c7242a4679287349242'
with open(object_path, "rb") as f:
    content = zlib.decompress(f.read()).decode('utf-8')

print(content)
```

When you run your `decompress.py` script in the shell, the following content will show:

```
$ python3 decompress.py
blob 20Arma virumque cano.
parent 1a410efbd13591db07496601ebc7a059dd55cfe9
author John Doe <john@example.com> 1590771487 -0400
committer John Doe <john@example.com> 1590771487 -0400

Initial commit
```

where

- `blob [length][content]` shows length of blob content following with its content
- `parent Hash_ID` points to the commit's immediate ancestor

Except decompressing with `zlib` library module in Python, git provides us with a more efficient and straightforward way to saftly view the content of a Git object via the plumbing command:

```
$ git cat-file -p 24b390b0e3489b71977f5c7242a4679287349242
Arma virumque cano.
```

Topological Sorting

We can use **Kahn's algorithm** to implement **topological sorting**. No nodes are listed first in this ordering. More specifically, for every child-parent relationship, the parent appears before the child in the sequence. The pseudocode from Wikipedia:

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error   (graph has at least one cycle)
else
    return L   (a topologically sorted order)
```

Now, let's implement this in Python:

```
from collections import deque

def kahn_topological_sort(graph):
    # Count incoming edges for each node
    in_degree = {u: 0 for u in graph}
    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1

    # Collect nodes with no incoming edges
    queue = deque()
```

```
    for u in in_degree:
        if in_degree[u] == 0:
            queue.appendleft(u)

    # Perform the Kahn's algorithm
    l = []  # list that will contain the sorted elements
    while queue:
        u = queue.pop()  # choose the node with zero in-degree
        l.append(u)  # add it to the topological order
        for v in graph[u]:
            in_degree[v] -= 1  # decrement in-degree of connected nodes
            if in_degree[v] == 0:
                queue.appendleft(v)  # add new zero in-degree nodes to
queue

    # Check if there was a cycle
    if len(l) == len(graph):
        return l
    else:
        # If there are still edges in the graph, there was a cycle
        raise ValueError("The graph has at least one cycle and cannot be
topologically sorted.")
```

Example usage:

```
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['D', 'F'],
    'D': ['E'],
    'E': ['F'],
    'F': []
}

sorted_nodes = kahn_topological_sort(graph)
print(sorted_nodes)
```

where the graph is represented as an adjacency list where a key is a node, and the value is a list of all nodes directed from the key node.

The `.git/refs` Directory

If you were interested in seeing the history of your repository reachable from commit, say, `1a410e`, you could run something like `git log 1a410e` to display that history, but you would still have to remember that `1a410e` 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
$ find .git/refs -type f
```

To create a new reference that will help you remember where your latest commit is via:

```
$ echo 1a410efbd13591db07496601ebc7a059dd55cfe9 > .git/refs/heads/master
```

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

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d 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
1a410efbd13591db07496601ebc7a059dd55cfe9
```

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 cac0ca
```

Then, your `test` branch will contain only work from that commit down:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Tag Object

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.

A **branch** is a lightweight movable name for a commit.

A **tag** is a lightweight constant name. It's like a label that references a commit, and unlike a branch, it does not move when new commits are added. It contains:

- Name
- Commit ID
- Other metadata

As discussed in Git Basics, there are two types of tags: annotated and lightweight. You can make a lightweight tag by running something like this:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

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 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'Test tag'
```

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

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Listing your tags:

```
git tag
```

Tags tend to be used for releases. For ordinary (unsigned) tags, the following tags `HEAD` with the ref `v38`.

```
git tag v38
```

Publishing Tags

Created tags are *local* to your repository. To *publish* this tag to the outside world...

`git push` won't work because what `push` actually does is look at current `HEAD` and the branches associated with that head and pushes those changes upstream. In other words, it pushes the *current branch* upstream. Tags are just names for commits, so they're not part of any branches.

Developers may also be in the habit of using many tags, and with many such developers at the same time, automatically pushing tags with every `$ git push` would very quickly pollute the upstream repository. Thus, not pushing tags by default is *by design*.

You must manually push tags:

```
git push --tags
```

Signed Tags

In a large codebase, chances are there will be many commits and tags that are in a bad state. Signing is like cryptographically "approving" a commit. Tag signing is used for security-sensitive software.

Creating a signed tag:

```
git tag -s v37 -m "Good version 37"
```

The `-s` flag signs the ref named `v37` with a message `Good version 37`. The signing creates a **GPG (GNU Privacy Guard) key**.

Other people can then check the tag to verify that it was signed:

```
git tag -v v37
```

We can take a Git tag, say `v37`, which references some commit with ID `09cf...`. What signing a tag does it taking the tag and signing it with a private key `K` and then publishing the result, which looks like a random bit string. Any other user can then use the signer's public key `U` to decrypt the tag.

A weakness is that if the private key `K` is leaked out, attackers can put out a bad version of the software that looks like it was signed by authority.

ASIDE: Cryptography

Private key system

The simplest system, where the sender and recipient share some key `K`.

You take your message `M` encrypted with a key `K`, and that's what's published. Attackers can't directly see the contents of the data. The recipient then uses their same `K` to decrypt `M`:

```
{M}K ===> [{M}K]K = M
```

Public key system

You have a pair of keys `(U, K)`, a **public key** and **private key** respectively.

The sender keeps its private key `K` secret and doesn't reveal it even to the recipient. The recipient publishes its public key `U`.

The sender encrypts the message with the recipient's public key `M` and publishes it. The recipient then uses their private key `K` to decrypt the message. Because of some math magic, `U` and `K` can undo one another despite being distinct:

```
{M}U ===> [{M}U]K = M
```

This also works vice versa. The sender can encrypt the message with their private key while the recipient uses the sender's public key to decrypt it:

```
{M}K ===> [{M}K]U = M
```

This system is what the GPG *keys* in Git use.

HW6: The .git/refs Directory

Branch names often assume the pattern of file paths because that's exactly how they're stored under `.git/refs` subdirectories.

Local branches are under `heads/`:

- `main` -> `.git/refs/heads/main`
- `fix/issue2` -> `.git/refs/heads/fix/issue2`

Remote branches are under `remotes/`:

- `origin/main` -> `.git/refs/remotes/origin/main`
- `origin/fix/issue2` -> `.git/refs/remotes/origin/fix/issue2`

Tags are under `tags/`:

- `v37` -> `.git/refs/tags/v37`
- `fixed/issue4` -> `.git/refs/tags/fixed/issue4`

Notice that ref names with slashes in them are stored as files nested within subdirectories accordingly. The *leaves* (last part of the path) are the actual files containing information about the branch the path corresponds to. So the files with content are the ones returned by:

```
$ find .git/refs -type f
$ find .git/refs -type f
```

```
.git/refs/heads/main
.git/refs/remotes/origin/main
.git/refs/tags/test/tag
```

In the above example, the file named `tag` at relative path `.git/refs/tags/test/tag` encodes the branch named with the subpath `test/tag`.

Branch File Anatomy

The files representing the branches have a singular line of uncompressed data in it:

```
$ cat .git/refs/heads/main
608a9573dc8fc1f9fdec898ae7a81d47ec91923a
```

It's simply the 40 hex-digit SHA value of the commit at their branch tip.

Reflog

```
git reflog
```

This inspects the `reflog` file. It is not maintained as an object in the object database. It is a *separate* log file that logs the *changes* that you made to the object database (most recent first).

`git reflog` lists the changes you make to a repository. It keeps track of things like moving `HEAD` as a result of committing, checking out, etc.

You can think of it as a "second order derivative" of your repository with respect to time, with the first order being the commit history, and the original function being your working files. Reflog is like changes *about* changes.

Object Naming Algebra

If you read the `man` pages for any Git commands, chances are you will encounter some recurring vocabulary and syntax patterns.

Firstly, names fall into two major categories:

pathspec

This refers to working file names, stuff like `src/main.c`.

Example of using a pathspec in a command:

```
git diff foo.c
```

commit names

This refers to commits and their aliases, stuff like:

- The `HEAD` pointer.
- Commits relative to another ref, like `HEAD^^2~`.
- Branch names like `main`, which really refer to the commit at their branch tip.
- Tag names like `v37`.

Example of using a commit name in a command:

```
git diff main
```

These two sets have different naming conventions. Notice that for the `git diff` examples, what if there's an ambiguity, like a file named `main`? Git commands support a special argument `--` that separates the part of a Git command that talks about *commit names* and parts that talk about *pathspecs*.. For example:

```
git diff main -- foo.c
```

This says, "show the difference between `foo.c` in the index and the one that's in the commit referenced by `main`."

If the `--` delimiter is missing, Git will try to guess. It general, you should try to avoid ambiguity by explicitly supplying the `--` token:

The equivalent commands from the prior examples:

```
git diff -- foo.c
git diff main --
```

Commit Name Notation

The `^` suffix on a commit name denotes a *parent* of that commit. You can think of the `^` as an operator. `^` means "get the parent", and it can take an optional integer "argument" to specify *which* parent in the case of a merge commit. So `HEAD^` is short for `HEAD^1`.

```
N^    # parent of N, equivalent to N^1
N^2   # second parent of N
N^^   # first parent of the first parent of N
N^2^  # first parent of the second parent of N
```

NOTE:

`N^` or `N^1` is the first parent of N.

`N^2` refers to the **second parent** of commit N. This only applies to merge commits that have more than one parent. It specifies the commit from the branch that was merged into the branch where N was.

`N^^` is equivalent to N^1^1, and it indicates the **first parent of the first parent** of commit N. It's a way to go two commits back in the history on the same branch.

Let's illustrate a commit history where N is a merge commit:

```
  A---B---C feature
 /         \
D---E---F---N master
```

Here's a visual representation of these references:

```
          N^2 (second parent of N, commit C)
          |
  A---B---C (feature)
 /         \
D---E---F---N (master)
    |   |
    |   └─ N^ (first parent of N, commit F)
    |
    └─ N^^ (first parent of the first parent of N, commit E) (Grandparent)
```

You can also use `~` to automatically take the first option when the ancestors of commits happen to branch into different parents. In other words, `N~k` is shorthand for `N` followed by `k` carets.

```
N~2  # N's grandparent, equivalent to N^^
```

You can combine the notations (evaluating left-to-right). This is kind of like *chaining* the operations, applying one after another. From above, we know that something like `HEAD^^` is actually shorthand for `HEAD^1^1`, which in math notation you can think of as something like `(HEAD^1)^1`: you first take the first parent of `HEAD`, then take **its** first parent. An example of a mixture:

```
N^^~2  # The grandparent of the grandparent of N
```

Logical NOT

You can also put the `^` in front, which is can be thought of as negation:

```
^N    # "not N"
```

This is a name for all commits that are *unreachable* fromm N.

Logical AND

You can combine such conditions:

```
^M N   # reachable from N but not from M
```

This pattern is so common that it has a shorthand:

```
M..N
```

We've seen this before with a linear history. In such cases, it's like the notion of a "range" of commits. This can be generalized to non-linear histories, where they technically mean "reachable from N but not from M".

Logical XOR

```
M...N
```

This means reachable from M or from N, but not both. This is useful in things like `git log`:

```
git log main...maint
```

This tells us the **"symmetric difference"** between the two branches: commits that are in `main` but not `maint`, commits that are in `maint` but not `main`, but not commits common to both.

Submodules

SCENARIOS:

- Your project needs some other project's source code. For example, perhaps you want your system to be buildable on platforms even if another system is absent.
- There are some software packages that are intended be used only in source code form.

CASE STUDY: The grep Source Code

In `grep`, there's a submodule called `gnulib`. This is intended to be a **portability library** (source code only). It helps "emulate" Linux atop other operating systems like MacOS and Windows by implementing certain features that are absent on those systems.

Usage

You can create a submodule from some source:

```
git submodule add https://github.com/a/b/c
```

This is similar to cloning, but instead of creating a new repository complete with its own `.git`, it:

- Creates an empty subdirectory.
- Create a `.gitmodules` file in the main project that establishes the relationship between the main project and the subdirectory.

Then, you run this, which initializes the submodule and then gets the source code from the provided link.

```
git submodule init
git submodule update
```

Rationale

The other project(s) is evolving. This sort of setup gives you the ability to update to the latest version of the submodule at your convenience. You want a **controlled update**.

You can provide the commit ID to `git submodule update` to update to a specific version of the submodule based on the remote repository it's tracking.

Furthermore, updating a submodule is very simple. Instead of making edits to possibly to many files if you kept the dependency as part of your repository, the changes you make to the main project are simply which commit ID to use for the submodule. This keeps both your code and cognitive load modularized.

Also, you don't want to change the subproject directly. That's the responsibility of their maintainers. You as the consumer simply use what they have released to their repository, and if something is wrong with it, submit a bug report/pull request/etc.