Git Internals: Plumbing Commands

Plumbing vs. Porcelain

The **plumbing** part of a program is the low-level part, where the actual work is done. It is not easy to use and not meant to be easy.

The **porcelain** is the high-level commands for the end users.

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.

Hard Links

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.

We see that a Git clone is created quickly partly because it did not have to make copies of the objects, and rather it used **hard links**.

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).

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` - obsolescent.
- `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

Creating Them

Generate a checksum from string content:

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

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

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

```
$ echo 'Arma virumque cano.' | git hash-object --stdin -w
24b390b0e3489b71977f5c7242a4679287349242
$ # This object now exists in the file system
$ 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 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 your with your consequences I guess.

Examining Objects

Decoding the content 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
```

Object Types

- **blob**: represents any bytes sequence, like regular files in a file system
- **tree**: represents a node in a tree of objects; maps names to SHA-1 checksums of blobs or other tree

You can see the organization in action with something like:

```
$ git cat-file -p 'main^{tree}'
100644 blob fa86c55e687fef76cb5801776756a6cb204cd2f9  .gitignore
100644 blob 630e72d7faef82103be5f27139030b67ef942ca0  README.md
100644 blob 1c6781665caca7fa3350bdfcb4a820fbec2dab05  week1.md
100644 blob f1f8f3890378332cffb6ddfb3dbc84c20e3a9470  week2.md
100644 blob d94ba050bf8045d989617e2fdf75206b2d7d97a4  week3.md
100644 blob 1131c977a2f8db9031e081a16c028bf493dfd02a  week4.md
100644 blob 4923796df144352f4784e7a90cf58a98e64d7e04  week5.md
```

```
100644 blob 8017ea5aaff41016d6813e329f75a008d2b8cd0a  week6.md
100644 blob e34d49dc0e8754b9de4e1a46449541c15e31af86  week7.md
100644 blob 8110b211ea75e284e8ad5a55b8e97c9927a0dc8a  week8.md
```

Each commit points to a tree. The tree is like a directory in a file system. Above, `main^{tree}` refers to the tree object referenced by the commit object referenced by the branch named `main`.

Examining the output, we see 4 columns:

```
100644 blob fa86c55e687fef76cb5801776756a6cb204cd2f9  .gitignore
(mode)(type) (SHA-1 checksum)                          (name)
```

- **1st column:** octal digits, the last three of which represent the Linux permissions of the file in question (so `644` means `rw-r--r--`).
- **2nd column:** the type, **blob**, another **tree**, etc.
- **3rd column:** the SHA-1 checksum of the *referred* object. After all, a tree just represents a pointer to a bunch of other objects.
- **4th column:** the name of the file.

Object Anatomy

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 main
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 attempting to view the contents directly with `cat` or in an editor would give you gibberish. You can safely see the context of a Git object with the plumbing command:

```
git cat-file -p 24b390b0e3489b71977f5c7242a4679287349242
```

You can read the contents of Git objects in Python by decompressing it with the `zlib` standard library module:

```
import zlib
object_path = ".git/objects/24/b390b0e3489b71977f5c7242a4679287349242"
with open(object_path, "rb") as fp:
    compressed = fp.read()
content = zlib.decompress(compressed).decode("utf-8")
print(content)
```

HW6: 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)
```

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.

Tagging

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

A **tag** is a heavier-weight 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

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.

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
```

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 second parent 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.