Diving Deep Into Git Internals

Abhradeep Chakraborty

github.com/Abhra303

Overview

- VCS and Git
- Terms you need to know
- What is packing?
- Counting Objects
- Reaching Bitmaps
- References

VCS and Git

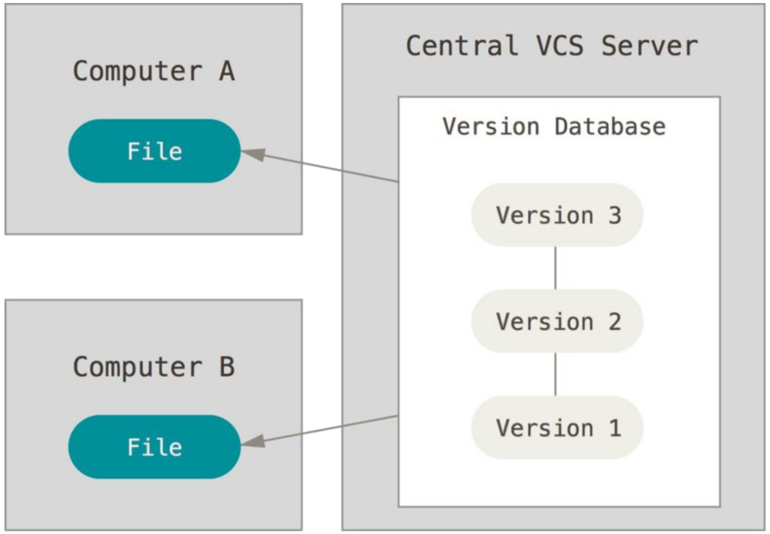
Version Control System is a system that records changes to a file or set of files over time so that you can recall specific versions later.

There are two types of version control system depending on the architecture -

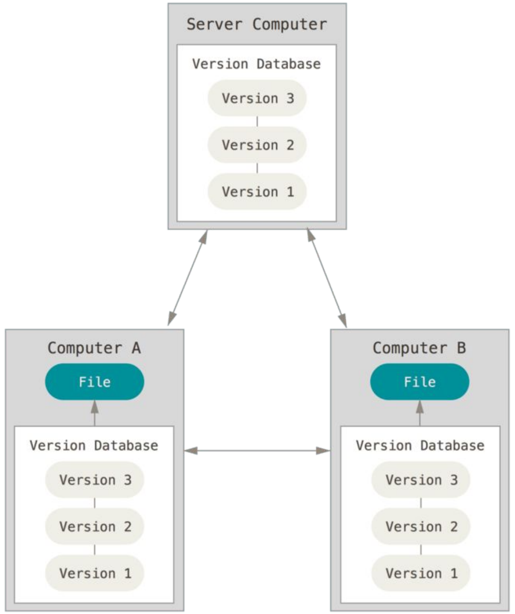
- **Centralized VCS -** A central server is out there. Clients check out servers from that server.
- **Distributed VCS -** Every client has the whole repository in the machine with each and every version of the repository files.

Because of the reliability and distributed control, Distributed VCSs are more popular than the CVS.

VCS and Git



Centralised VCS



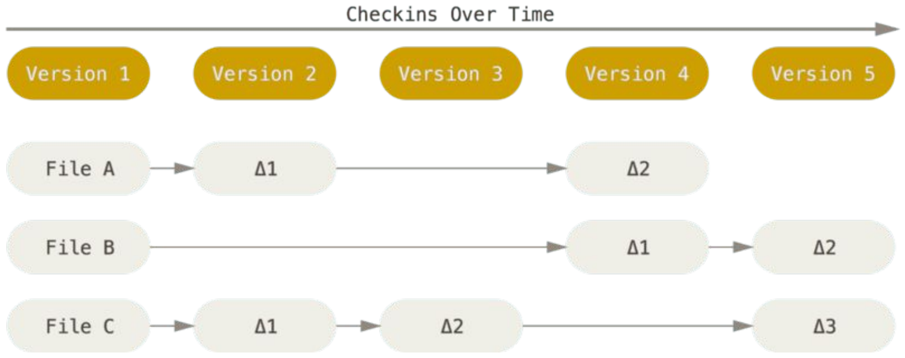
Distributed VCS

VCS and Git

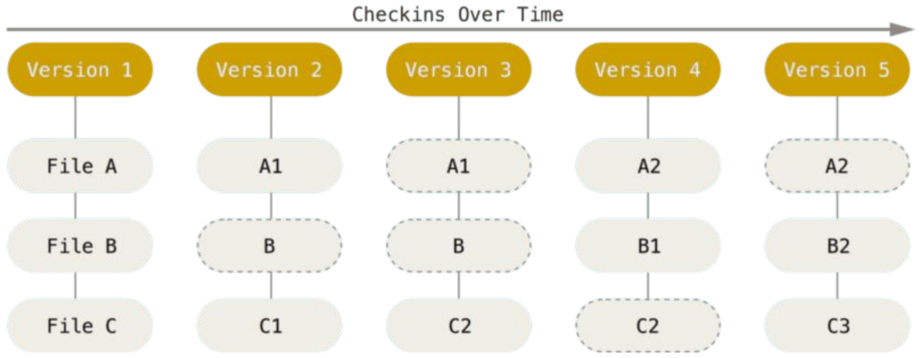
Git is a fully Distributed Open Source Version Control System designed to efficiently handle small to large repositories. Some points you may want to know -

- It was initially developed by Linus Torvalds to maintain the Free Linux Software codebase.
- Git is a part of Free Software Foundation and hence it uses GPL license.
- Due to its unique architecture, it is relatively faster than most of its competitors.
- Git checksums everything before storing. You can also reference the exact version using the checksum making it non-modifiable.
- Unlike other traditional VCS where file changes are stored as delta, Git uses snapshots to save the state of the codebase. This make Git extremely powerful.

VCS and Git



Delta-based VCS



Git Snapshots

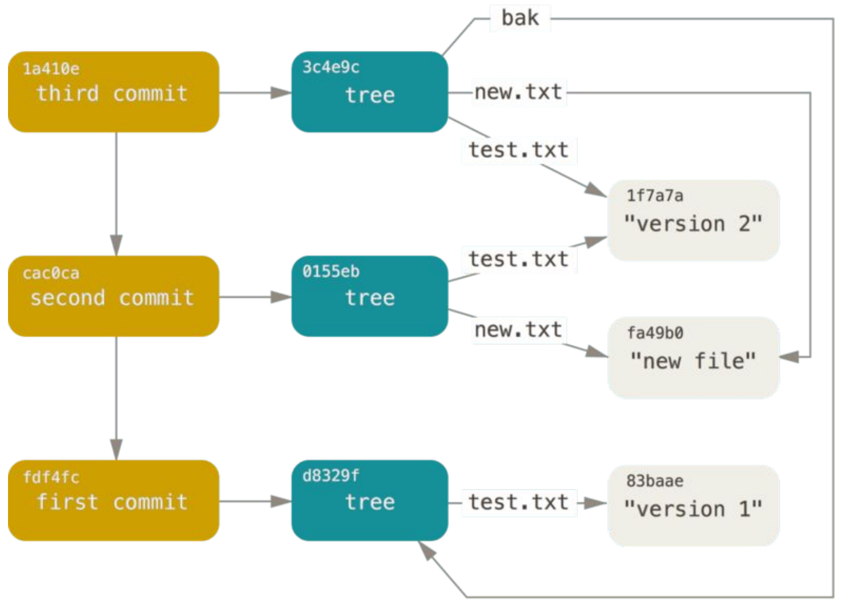
Terms you need to know

Plumbing and Porcelain Commands - In its initial time, Git was a toolkit for a version control system rather than user friendly VCS. Later it focused on user experience. As a result, Git has two types of commands - Plumbing commands and Procelain Commands. Commands that are intended to used by users are called Porcelain commands and commands used for low-level operations are called Plumbing commands.

Git Objects - Git stores its data into a **Directed Acyclic Graph** like data-structure. There are mainly four types of data that are stored in the graph - commit, tree, blob and tag. These are called **objects**. Related objects are linked with each other.

Blob stores the raw file content, tree represents directories that stores list of blob references.

Terms you need to know



Terms you need to know

Every Git objects has their own checksum and they are stored in the `.git/objects/` directory.

For example, if a Git object has checksum `d670460b4b4aece5915caf5c68d12f560a9fe3e4`, you can inspect the content of the object by going to the `.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4` file.

Use `cat-file` command to see the raw content of an object -

```
$ git cat-file -p .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4      # blob
Test content
```

Other plumbing commands to manually create objects - `hash-objects`, `write-tree`, `update-index` and `commit-tree`

Terms you need to know

```
$ git cat-file -p "master^{tree}"    # tree; points to the tree of last commit in the master branch
100644 blob a906cb2a4a904a152e80877d4088654daad0c859     README
100644 blob 8f94139338f9404f26296befa88755fc2598c289     Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0     lib
```

Assuming that the master HEAD points to a root directory having README, Rakefile and lib directory in it. The first column specifies the mode e.g. 100644 to denote normal file. Next column denotes the type of object. Here you can see there are two files in the root directory and a lib sub-directory. Note that, the last row is a pointer to another tree.

Packfiles

Imagine you have a large file in your repository (e.g. 10k bytes). You may need to modify some content in the file.

Each time you change the file, it takes snapshots of it and stores it in a new blob, each having a huge size. It would unnecessarily increase the size of the repository. Can we do any kind of optimization here?

Here comes Packfiles into picture. For a large repository, Git creates a packfile in the `.git/objects/pack` directory where it stores all the related git objects in a deltified form. Deltas are nothing but the changes between a file and its another version. So each time you modify the large file, it will create delta of the previous changes. Note that, the most recent version has the full file content.

Packfiles

Git always looks for optimization internally. That’s why you don’t need to create packfiles manually. Git has a beautiful command `gc` which Git uses to check if further optimization is needed.

Every packfile has its own .idx file (i.e. index file). The index file is like an index of a book. It stores the index of every objects in the packfile along with other information (i.e. file size, object type etc.)

Use `git verify-pack -v` to see the content of the .idx file.

Packfiles

```
$ git verify-pack -v .git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
fe879577cb8cffcdf25441725141e310dd7d239b tree   136 136 996
.
.
.
```

Counting objects

When you clone a repo or fetch something from a remote server, you often notice the `counting objects:` step in your terminal. So what is this `Counting objects`?

```
$ git clone https://github.com/torvalds/linux
Cloning into 'linux'...
remote: Counting objects: 4350078, done.                                   # this one
remote: Compressing objects: 100% (4677/4677), done.
Receiving objects:   4% (191786/4350078), 78.19 MiB | 8.70 MiB/s
```

Counting objects

When you fetch or clone something from a remote server, the remote server (can be Github, Gitlab etc.) needs to find out which objects are already present in the local machine.

In the case of cloning, the local machine doesn’t have any objects in it. So the remote server has to send all the related git objects to the local machine. The `counting objects` step computes the number of objects it needs to send to the local repo.

In the case of fetching, there is a need of conversation between the local git client and the remote server to decide which objects are already fetched and which objects are not.

Reachability bitmaps

So how does Git count objects? After all, Git knows nothing more than the branch heads (branch heads are the latest commit for each branch) and it can’t send all the objects in the remote because some objects may be not reachable at all.

The answer is simple. Git traverses each and every commit reachable from the heads, and from each commit, its trees, sub-trees and blobs are also traversed just to decide which objects to send.

The bigger the size of the repo will be, the bigger the graph would be in size and longer time will be taken to compute reachability.

Reachability bitmaps

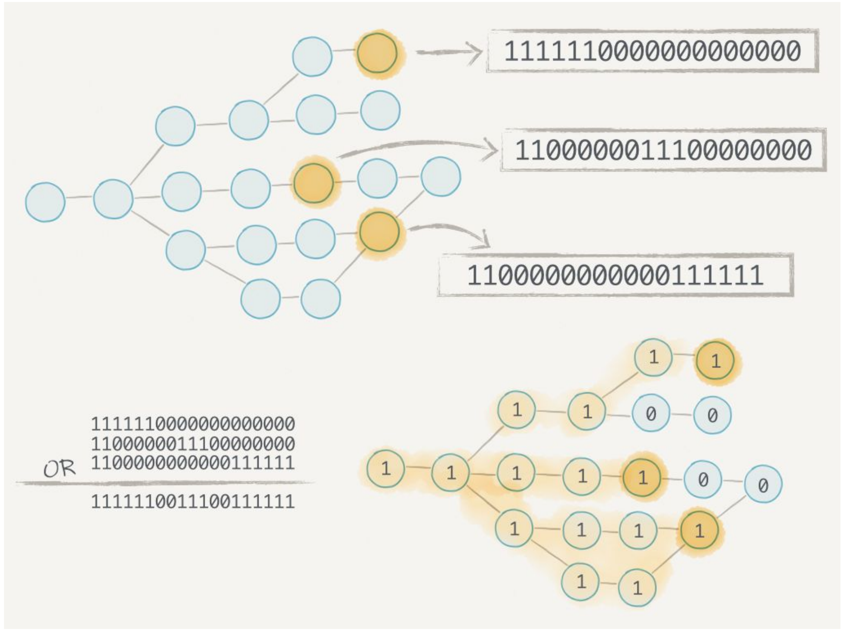
Can we do any kind of optimization here? We can’t load the whole graph of the linux repository to count objects.

Here comes Reachability bitmaps. Reachability bitmaps are bitmaps that can quickly say if a certain object is reachable from another object.

The idea is that commits will have their own bitmaps. Bitmaps will be sorted in the pack order i.e. i’thindex of the bitmap will denote the i’th object in the pack file.

If i’th object is reachable from the owner commit, the i’th position of the bitmap is set to 1 and 0 otherwise. To clone, the remote server loads bitmaps of every commits and then OR it to get the reachability bitmaps.

Reachability bitmaps



Reachability bitmaps

Commits are still needed to be traversed. But we don’t have to traverse trees, sub-trees and blobs anymore.

The latest 100 commits are selected for bitmaps and after that a heuristic approach is taken to decide which commit should have a bitmap.

For the fetching, the local Git computes the reachability bitmap of its objects and the remote server computes its own reachability bitmap. Then a basic ANDNOT operation will give back a bitmap containing the needed objects.

References

The images and references are taken from two sources -

- Pro Git book - https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control
- Counting objects | The Github Blog - https://github.blog/2015-09-22-counting-objects/

Thank You!