Version Control Systems (VCSs)

Version control systems (VCSs) are tools designed to track changes in source code and other collections of files and directories. As their name suggests, these tools maintain a history of changes and facilitate collaboration. VCSs take snapshots of a folder and its contents, capturing the entire state of the files and directories at a given time. They also keep metadata, such as who created each snapshot and associated messages.

The Benefits of Version Control

Even for individual developers, version control provides the ability to review past project snapshots, maintain a log of changes made, work on different development branches, and much more. When collaborating with others, VCSs are essential for understanding modifications made by teammates and resolving conflicts during simultaneous development.

Modern VCSs can help easily answer questions such as:

- Who authored this module?
- When was a specific line in a file edited? By whom? What was the reason for the change?
- Over the last 1000 revisions, when did a particular unit test start failing and why?

While there are various VCS options, Git has become the standard. An XKCD comic humorously illustrates Git’s reputation.

Git's interface can be confusing due to its complexity, leading some to memorize commands without understanding their purpose, akin to using magic spells when issues arise.

Despite its challenging interface, Git’s underlying design is elegant. A clear understanding of its data model helps make sense of its commands. Thus, we’ll begin with an explanation of Git’s data model before diving into its command-line interface.

Git’s Data Model

Git's design supports essential version control features like history tracking, branch support, and collaboration.

Snapshots

Git represents the history of files and directories as a series of snapshots. In Git terminology, files are called “blobs,” which are simply collections of bytes. Directories are referred to as “trees,” which map names to blobs or trees, allowing for nested directories. The top-level snapshot is called a tree. For example, a tree structure might look like this:

```
<root> (tree)

|

+- foo (tree)

| |

| + bar.txt (blob, contents = "hello world")

|

+- baz.txt (blob, contents = "git is wonderful")
```

The top-level tree contains two elements: a tree called “foo” (which includes a blob named “bar.txt”) and a blob named “baz.txt.”

Modeling History: Relationships Between Snapshots

How does a VCS relate these snapshots? A simple approach would be to have a linear history of snapshots. However, Git uses a more complex structure.

In Git, the history is represented as a directed acyclic graph (DAG) of snapshots. Each snapshot refers to a set of “parents,” which are the preceding snapshots. This allows for multiple parent snapshots, such as when two branches are merged.

These snapshots are called “commits.” The visualization of a commit history might look like this:

```
o <-- o <-- o <-- o

      ^
      \
       --- o <-- o
```

In this diagram, the “o” represents individual commits. The arrows indicate parent relationships. After the third commit, the history branches into two paths, reflecting the independent development of two features. Eventually, these branches may be merged, resulting in a new commit that incorporates both features:

```
o <-- o <-- o <-- o <---- o
      ^      /
      \     v
       --- o <-- o
```

Commits in Git are immutable. This means that instead of modifying existing commits, new commits are created to reflect changes, and references are updated to point to the new commits.

Data Model in Pseudocode

Here’s a simplified representation of Git’s data model:

```
// A file is a collection of bytes
type blob = array<byte>


// A directory consists of named files and directories
type tree = map<string, tree | blob>


// A commit includes parents, metadata, and the top-level tree
type commit = struct {
    parents: array<commit>
    author: string
    message: string
    snapshot: tree
}
```

This model provides a straightforward representation of version history.

Objects and Content-Addressing

In Git, an “object” can be a blob, tree, or commit:

```
type object = blob | tree | commit
```

All objects are stored in a content-addressed manner using their SHA-1 hash.

```
objects = map<string, object>

def store(object):
    id = sha1(object)
    objects[id] = object

def load(id):
    return objects[id]
```

Blobs, trees, and commits are all treated as objects. When they reference each other, they do not contain the actual data in their on-disk format; rather, they reference the data by its hash.

For instance, the tree structure mentioned earlier could be represented in Git like this:

```
100644 blob 4448adbf7ecd394f42ae135bbeed9676e894af85    baz.txt

040000 tree c68d233a33c5c06e0340e4c224f0afca87c8ce87    foo
```

This tree points to its contents, “baz.txt” (a blob) and “foo” (a tree). Viewing the content for “baz.txt” via its hash reveals:

```
git is wonderful
```

References

All snapshots can be identified by their SHA-1 hashes, which is not very user-friendly. Git solves this by using human-readable names, called “references,” that point to commits. Unlike objects, which are immutable, references can be updated:

```
references = map<string, string>


def update_reference(name, id):

   references[name] = id


def read_reference(name):

   return references[name]
```

```
def load_reference(name_or_id):
  if name_or_id in references:
    return load(references[name_or_id])
  else:
    return load(name_or_id)
```

This allows users to refer to commits using recognizable names, like “master,” instead of cumbersome hexadecimal strings.

Git also maintains a special reference called “HEAD” to denote the current position in the commit history, which is essential for creating new snapshots.

Repositories

A Git repository consists of the data objects and references. Essentially, all Git commands manipulate the commit DAG by adding objects and updating references.

Whenever executing commands, consider how they alter the underlying data structure. If you need to perform a specific action, such as discarding uncommitted changes, Git likely has a corresponding command.

Staging Area

The staging area is an important aspect of Git that enables selective snapshot creation. Unlike some VCSs that automatically create snapshots of the current state, Git allows you to specify which modifications to include in the next snapshot. This is particularly useful in scenarios where you want to create distinct commits for different features or changes.

Git Command-Line Interface

For detailed command explanations, refer to the recommended resource "Pro Git" or related lecture videos.

Basic Commands

- `git help <command>`: Get help for a specific Git command.

- `git init`: Create a new Git repository, stored in the `.git` directory.

- `git status`: Check the current status of your repository.

- `git add <filename>`: Add files to the staging area.

- `git commit`: Create a new commit.

- `git log`: View a flattened history of commits.

- `git log --all --graph --decorate`: Visualize history as a DAG.

- `git diff <filename>`: Show changes relative to the staging area.

- `git checkout <revision>`: Update the current branch to a specific revision.

Branching and Merging

- `git branch`: List branches.

- `git branch <name>`: Create a new branch.

- `git checkout -b <name>`: Create and switch to a new branch.

- `git merge <revision>`: Merge changes into the current branch.

- `git mergetool`: Use a tool to resolve merge conflicts.

Working with Remotes

- `git remote`: List remote repositories.

- `git remote add <name> <url>`: Add a remote repository.

- `git push <remote> <local branch>:<remote branch>`: Send changes to a remote repository.

- `git fetch`: Retrieve updates from a remote repository.

- `git clone`: Download a repository from a remote location.

Undoing Changes

- `git commit --amend`: Edit the last commit's contents or message.

- `git reset HEAD <file>`: Unstage a file.

- `git checkout -- <file>`: Discard changes to a file.

Advanced Git

- `git config`: Configure Git settings.

- `git clone --depth=1`: Perform a shallow clone without the entire history.

- `git add -p`: Interactive staging.

- `git rebase -i`: Interactive rebasing.

- `git blame`: Show who last edited each line.

- `git stash`: Temporarily save modifications.

- `git bisect`: Perform a binary search through history for regressions.

- `.gitignore`: Specify files to ignore in version control.