Lecutre 06: Version Control (Git)

Git's data model

Snapshots

Git models the history of a collection of files and folders within some top-level directory as a series of **snapshots**.

- A file is called a "**blob**", and it's just a bunch of bytes.
- A directory is called a "**tree**", and it maps names to blobs or trees
- A snapshot is the top-level tree that is being tracked

```
<root> (tree)
|
+- foo (tree)
|  |
|  + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is wonderful")
```

Modeling history: relating snapshots

In Git, a history is a **directed acyclic graph** (DAG) of snapshots. Git calls these snapshots "**commit**"s. Visualizing a commit history might look something like this:

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

In the future, these branches may be merged to create a new snapshot that incorporates both of the features, producing a new history that looks like this, with the newly created merge commit shown in bold:

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

Data model, as pseudocode

Write down GIt's data model in pseudocode:

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

// a directory contains named files and directories
type tree = map<string, tree | file>

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

Objects and content-addressing

An “object” is a blob, tree, or commit:

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

In Git data store, all objects are content-addressed by their SHA-1 hash.

```
objects = map<string, object>

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

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

References

Git’s solution to this problem is human-readable names for SHA-1 hashes, called “**references**”. References are pointers to commits. Unlike objects, which are immutable, references are mutable (can be updated to point to a new commit). For example, the `master` reference usually points to the latest commit in the main branch of development.

```
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)
```

In Git, that "where we currently are" is a special reference called "`HEAD`".

Repositories

We can define what (roughly) is a Git **repository**: it is the data objects and references.

Conversely, if you're trying to make a particular kind of change to the commit DAG, e.g. "discard uncommitted changes and make the 'master' ref point to commit `5d83f9e`", there's probably a command to do it (e.g. in this case, `git checkout master; git reset --hard 5d83f9e`).

Staging area

Git accommodates such scenarios by allowing you to specify which *modifications* should be included in the next snapshot through a mechanism called the "**staging area**".

Git command-line interface

For Git commands, see the highly recommended Pro Git for more information, or watch the lecture video.

Basics

- `git help <command>`: get help for a git command
- `git init`: creates a new git repo, with data stored in the `.git` directory
- `git status`: tells you what's going on
- `git add <filename>`: adds files to staging area
- `git commit`: creates a new commit
    - Write good commit messages!

    - Even more reasons to write good commit messages!
- `git log`: shows a flattened log of history
- `git log --all --graph --decorate`: visualizes history as a DAG
- `git diff <filename>`: show differences since the last commit
- `git diff <revision> <filename>`: shows differences in a file between snapshots
- `git checkout <revision>`: updates HEAD and current branch

Branching and merging

- `git branch`: shows branches
- `git branch <name>`: creates a branch
- `git checkout -b <name>`: creates a branch and switches to it
    - same as `git branch <name>; git checkout <name>`
- `git merge <revision>`: merges into current branch
- `git mergetool`: use a fancy tool to help resolve merge conflicts
- `git rebase`: rebase set of patches onto a new base

Remotes

- `git remote`: list remotes
- `git remote add <name> <url>`: add a remote
- `git push <remote> <local branch>:<remote branch>`: send objects to remote, and update remote referencerever
- `git branch --set-upstream-to=<remote>/<remote branch>`: set up correspondence between local and remote branch
- `git fetch`: retrieve objects/references from a remote
- `git pull`: same as `git fetch; git merge`
- `git clone`: download repository from remote

Undo

- `git commit --amend`: edit a commit's contents/message

- `git reset HEAD <file>`: unstage a file
- `git checkout -- <file>`: discard changes

Advanced Git

- `git config`: Git is highly customizable
- `git clone --depth=1`: shallow clone, without entire version history
- `git add -p`: interactive staging
- `git rebase -i`: interactive rebasing
- `git blame`: show who last edited which line
- `git stash`: temporarily remove modifications to working directory
- `git bisect`: binary search history (e.g. for regressions)
- `.gitignore`: specify intentionally untracked files to ignore

Miscellaneous

- **GUIs**: there are many GUI clients out there for Git. We personally don't use them and use the command-line interface instead.
- **Shell integration**: it's super handy to have a Git status as part of your shell prompt (zsh, bash). Often included in frameworks like Oh My Zsh.
- **Editor integration**: similarly to the above, handy integrations with many features. fugitive.vim is the standard one for Vim.
- **Workflows**: we taught you the data model, plus some basic commands; we didn't tell you what practices to follow when working on big projects (and there are many different approaches).
- **GitHub**: Git is not GitHub. GitHub has a specific way of contributing code to other projects, called pull requests.
- **Other Git providers**: GitHub is not special: there are many Git repository hosts, like GitLab and BitBucket.

Resources

- Pro Git is **highly recommended reading**. Going through Chapters 1–5 should teach you most of what you need to use Git proficiently, now that

you understand the data model. The later chapters have some interesting, advanced material.

- Oh Shit, Git!?! is a short guide on how to recover from some common Git mistakes.
- Git for Computer Scientists is a short explanation of Git's data model, with less pseudocode and more fancy diagrams than these lecture notes.
- Git from the Bottom Up is a detailed explanation of Git's implementation details beyond just the data model, for the curious.
- How to explain git in simple words
- Learn Git Branching is a browser-based game that teaches you Git.

Others