Development Tools

Muchang Bahng

Fall 2024

Contents

1   Neovim

The first thing you do when coding is typing something, and this requires a text editor.

2   Git

Git is a pretty complex version control tool. It allows you to perform different actions. We'll go over them, starting with the most basic to the most complex. In order to learn this, we should know the structure of the git history.

2.1   Local Git Repository

When you do `git init` in a repository, you are essentially saying that you want to keep track of the history of this repository. This can obviously be done with an undo tree, which comes out-of-box in almost all text editors, but it is much more powerful.

Definition 2.1 (Local Git Tree)

The history of our repository is essentially a tree, with each node representing some edits composed of
1. adding a new file
2. modifying a file
3. deleting a file

Each node is represented by a hash generated from its previous node and the corresponding edits. You can see your history using

```
1   git log
```

`HEAD` is a pointer to the node that reflects the state of your current repository (minus your uncommitted edits), which is usually the most recent node.

Unlike most undo trees, these nodes are not added automatically. You must add them manually through a 2-step process.

Definition 2.2 (Stage)

You want to take a set of edits and **stage** them. This essentially tells git that these staged files/lines are going to be a part of the next node.

Definition 2.3 (Commit)

Then you commit your changes, which does the following.
1. This takes all of your staged changes and packages them in a node A.
2. It looks at `HEAD`, uses `HEAD`'s hash to generate the hash of A, and appends A to `HEAD` by having A point to `HEAD`.[a]
3. It moves `HEAD` to A.

Therefore, when you make your first commit, you are creating a genesis node from which every other edit will be based off of. Your `HEAD` then points to this commit. This is great start, and let's add more functionality.

---

[a]So nodes actually point to *previous nodes.*

> **Definition 2.4 (Checkout a Commit)**
>
> You can move `HEAD` to point to a specific commit by using
>
> ```
> git checkout <commit-hash>  # point to this commit
> git checkout HEAD~N         # point to the commit $N$ nodes before HEAD
> ```
>
> This leaves you in a `detached head state`, which means that your head is not pointing to the end node. This is useful if you want to
> 1. *explore the codebase at a commit's snapshot in time.*

Note that so far, we have described git as a linked list plus some extra head pointer. Adding to this linked list is easy since we are simply adding new edits, but deleting can be very tricky. We will first introduce how to delete the most recent K commits, which is the easiest way to delete.

> **Definition 2.5 (Reset)**
>
> Say that your history is
> $$(A) \leftarrow (B) \leftarrow (C) \leftarrow (H \mapsto D) \tag{1}$$
> If we want to throw away commits C and D, we can **reset** to B, which deletes C, D and has H point to B, giving us
> $$(A) \leftarrow (H \mapsto B) \tag{2}$$
> 1. A **soft reset** means that the edits introduced in C and D will still be kept as unstaged changes, and so you may use them as a starting point to make your next commit.
> 2. A **hard reset** means that the edits are also completely deleted.

Most beginners in git really know these commands when working with their history, but this is really just a glorified stack. The additional operations can be daunting because they have the risk of introducing *conflicts*.

2.2 Conflicts

> **Definition 2.6 (Conflicts)**
>
> A **conflict** arises when two commits contain edits that change some location independently at the same time. They occur most frequently when working with multiple branches, but they can happen even when working on a single branch. Git will tell you when there is conflict between commits C and C' at a certain location. At this point, you will have to manually go to that location and compare the changes introduced in C and C', called **hunks**. The conflict looks generally like this.
>
> ```
> ... some code above
> <<<<< (C)    # hunk 1
> ========
> >>>>> (C')  # hunk 2
> ... some code below
> ```
>
> In order to fix this conflict, you can
> 1. select hunk 1 (and ignore hunk 2)
> 2. select hunk 2
> 3. select both hunks (i.e. incorporate both edits)
> 4. manually delete the ≫, ===, ≪ and directly edit the file to make a custom change that overrides both hunks.

Choosing the option to fix a conflict may sometimes be complicated, since you may not always want to select the hunk reflected in your most recent changes, because doing that might introduce another conflict in a

later commit that actually modified the old code into the new code.

> **Definition 2.7 (Revert Commit)**
>
> Say that you have history
> $$(C_1) \leftarrow (C_2) \leftarrow (C_3) \leftarrow (H \mapsto C_4) \tag{3}$$
>
> You can choose to **revert** and of the 4 commits above. Given any commit C, reverting a commit means that you simply add a new commit C' with the changes that are the exact opposite of C. If we want to revert commit C_2, our history looks like
> $$(C_1) \leftarrow (C_2) \leftarrow (C_3) \leftarrow (C_4) \leftarrow (C_2') \tag{4}$$
>
> So really, we are "deleting" our history by adding.

> **Example 2.1 (Conflicts in Reverting)**
>
> Say that you have history
> $$(C_1) \leftarrow (C_2) \leftarrow (C_3) \leftarrow (H \mapsto C_4) \tag{5}$$
>
> If you try to revert H, this is fine and will never have conflicts. Say that you made an edit in (C_3) where you added $x = 4$ to some python script, and then you removed this line in (C_4). Then if you add (C_3') to undo it, it tries to delete a line that isn't even there! Therefore you will get a conflict that looks something like
>
> ```
> 1  <<<<< (C4)    # hunk 1
> 2  - x = 4
> 3  ========
> 4  - x = 4
> 5  >>>>> (C3')  # hunk 2
> ```
>
> Obviously you can just select either one of the hunks to get what you want.

Conflicts are unavoidable and you will have to get comfortable with them.

2.3   Interactive Rebasing

Even though we can revert commits, we haven't actually found out how to truly *delete* a commit from your history which modifies
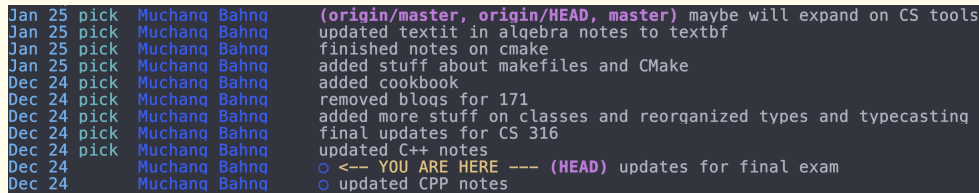$$(A) \leftarrow (B) \leftarrow (C) \leftarrow (D) \tag{6}$$

to something like
$$(A) \leftarrow (B) \leftarrow (D) \tag{7}$$

> **Definition 2.8 (Rebasing)**
>
> Essentially, we want to *directly* (unlike a revert) modify our history that goes *beyond* (unlike a reset) the last K commits. Any actions that modifies the history is known as **rebasing**, which can be done automatically by git but must often be done **interactively**. When you want to start an interactive rebase, you want to tell git from which commit C_s you want to start the interactive rebase on.
>
> ```
> 1  git rebase -i <start commit hash>
> ```
>
> You are saying that from commit C_s and beyond until the end C_n, I may arbitrarily modify them, but commits previous to C_s will be untouched. When you do this, all commits C_i where $i \geq s$ will be shown as below.

Figure 1: Interactive rebase shown in LazyGit.

There are a fixed set of supported operations allows in an interactive rebase.[a]
1. **Pick**. This just means that you are leaving the commit alone, i.e. picking it to be in the rebase.
2. **Reword**. Just edits the commit message.
3. **Squash**. Given commit $C_i \leftarrow C_{i+1}$, you can label C_{i+1} with `squash` to merge it into C_i, turning 2 nodes into one. This almost never causes conflicts. The new commit message is just those of C_i, C_{i+1} concatenated.
4. **Fixup**. Like squash, but discard the commit's message.
5. **Drop**. This deletes a commit and removes it entirely.
6. **Break**. Stop at this commit to edit it. I think you can change which edits you have committed, choose which edits to keep, and choose which edits to remove (back into your unstaged changes).
7. **Edit**. Stop at this commit to amend it.
8. You can also swap commits by editing the text file so that the commits are in a different order.

```
1  # Original order in rebase editor:
2  pick abc123 First commit
3  pick def456 Second commit
4
5  # After swapping lines in editor:
6  pick def456 Second commit
7  pick abc123 First commit
```

After you edit the rebase text file and continue the rebase, git will do the following sequentially:
1. `HEAD`, which pointed at C_n, will point towards C_s.
2. While `HEAD` is pointing at $C_i \neq C_n$ (i.e. not at the end), we do the following.
   (a) It will attempt to perform all the operations you have specified for the next commit C_{i+1}.
   (b) If the operations are finished, we increment `HEAD` to point to C_{i+1} and continue.
   (c) If there is a conflict, it will pause, state that there are conflicts between `HEAD` $= C_i$ and C_{i+1}, and ask you to resolve them. Once resolved it will continue.
3. Then we are done with the rebase since we have went through all commits, modified them, and resolved all conflicts.

Interactive rebasing is an extremely powerful way to modify your commit history, and it's probably the operation where you'll spend the most time on git.

Again, note that if you have changed anything in commit C_i, then the hash of C_i every C_j after will get changed. This causes git to interpret these changed commits as completely new ones, even if we only picked a given commit without any modifications. For single-branch rebases, this is fine, but this causes some nasty problems when rebasing over multiple branches, as we will talk about later.

2.4   Branches

Okay, so we now have much better control over our git history, but we've only been treating our history as a linked list. In order to introduce the tree structure, we need to introduce the *branch*. This is especially

---

[a]Note that pick and reword will never cause conflicts. Squash and fixup will most likely not cause conflicts. Drop, break, edit, and swapping may cause conflicts.

important if we have a particular previous commit $C_{k<n}$ where we would like to make some different changes to, giving us **diverging histories** with next nodes $C_k \leftarrow C_{k+1}$ and $C_k \leftarrow C_{k'+1}$.

Definition 2.9 (Branch)

A **branch** is a path from the root commit to any leaf node. It represents a unique history from genesis to `HEAD`. To list all branches, use

```
1  >> git branch
2     feature/threading
3  * main
4     test/tensor
```

The asterisk represents which branch you are currently on. The first branch you start off with is a special branch called **main**, or **master** branch.

Therefore, really our linked-list history is a git tree with a single branch.

Definition 2.10 (Creating/Switching Branches)

From any (main or non-main) branch you can create new branches by choosing the commit to split from.
1. Create a new branch from HEAD of current branch.

```
1  git branch <new-branch-name>
```

2. Create a new branch from certain commit of current branch.

```
1  git branch <new-branch-name> <commit-hash>
```

3. To switch to another branch

```
1  git checkout <branch>
```

2.4.1 Working Between Branches

If you are simultaneously working on multiple branches, you may have to checkout/switch between branches frequently. Often, you may have uncommitted changes before you checkout, and git does not allow you to do this. Therefore, we can *stash* them.

Definition 2.11 (Stash)

Stashing changes mean that you can take uncommitted changes and store them in a temporary node but not have it point to any existing commit in a branch. This allows you to save your changes without having to commit incomplete work to a branch, and you can pop them back whenever you need.

2.4.2 Merge

Definition 2.12 (Fast-Forward Merge)

2.4.3   Rebase

> **Definition 2.13 (Rebase)**

> **Definition 2.14 (Rebasing Diamond Merges)**

2.5   Remote Trees

This allows you to keep track of your change history, but another benefit is to store these changes in the cloud. In order to do this, you must have access to a remote server.

> **Definition 2.15 (Remote References)**
>
> In your local branch, it

> **Definition 2.16 (Remote Git Tree)**

> **Definition 2.17 (Fetch)**

> **Definition 2.18 (Fast-Forward)**

> **Definition 2.19 (Push)**

3   Continuous Integration (CI)

Continuous integration (CI), or **continuous development (CD)**, refers to any automated process that runs whenever you perform some action on a repository. These can include:

1. Compiling your package upon pushing to a git branch. This saves the time of manually compiling it yourself.

2. Compiling and/or running unit tests on your package, over possibly different compiler/interpreter versions on different operating systems and different architectures, whenever someone opens a pull request. This is usually done by automatically creating docker images and running a script that sets up the environment for your system.

3. Automatically publishing a new package version to PyPI upon a push to the master branch of a repository.

Github actions provide **workflow scripts** that you can include in your repository's `github/workflows/` directory that automates this. They are essentially yaml files that activate upon some command, whether that'd be a push to a branch, a pull request, or even the completion/failure of another workflow. This gives great convenience in deploying code.

4