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

```
1  git checkout <commit-hash>  # point to this commit
2  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.

```
1  ... some code above
2  <<<<< (C)    # hunk 1
3  ========
4  >>>>> (C')   # hunk 2
5  ... 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

```
<<<<< (C4)    # hunk 1
- x = 4
========
- x = 4
>>>>> (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.

Definition 2.8 (Amending a Commit)

If you have some staged edits and you decide that these edits should go into some previous commit rather than a new one, you can **amend** the old commits. In lazygit, you can stage which edits you want to amend with, then go to the commit in your working branch and press `<shift-a>` to amend it.

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 2.9 (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 (regular rebasing just picks all commits) but must often be done **interactively**,

---

which allows for more operations listed below. 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.


```
Jan 25 pick  Muchang Bahng     (origin/master, origin/HEAD, master) maybe will expand on CS tools
Jan 25 pick  Muchang Bahng     updated textit in algebra notes to textbf
Jan 25 pick  Muchang Bahng     finished notes on cmake
Jan 25 pick  Muchang Bahng     added stuff about makefiles and CMake
Dec 24 pick  Muchang Bahng     added cookbook
Dec 24 pick  Muchang Bahng     removed blogs for 171
Dec 24 pick  Muchang Bahng     added more stuff on classes and reorganized types and typecasting
Dec 24 pick  Muchang Bahng     final updates for CS 316
Dec 24 pick  Muchang Bahng     updated C++ notes
Dec 24       Muchang Bahng     o <-- YOU ARE HERE --- (HEAD) updates for final exam
Dec 24       Muchang Bahng     o updated CPP notes
```

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

---

[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.

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.

Definition 2.10 (Patching)

An easier way to modify your edits in old commits is through **patching**.[a] Within a commit, a **patch** is simply a diff file that you can add and remove to. It's like having a mini-staging area in a commit. When you have selected the different files/lines you have added to your patch, you can either choose to:
1. Remove them from the commit.
2. Add them to another commit.
3. Move them from the original commit to a new commit.

Theorem 2.1 (Splitting Commits Into Two Different Commits)

If you want to split commits,
1. create a dummy commit
2. go to the commit you want to split, get its patches, and add them to the dummy commit.
3. Do an interactive rebase to swap the dummy commits down until you have it at the desired location.

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 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 2.11 (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 2.12 (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.

---

[a]Though patching really just does an interactive rebase in the backend.

```
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 2.13 (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.

Sometimes, you may want to just copy a commit from one branch to another. You can do this using an interactive rebase, but this may be overkill since it is mainly used to work with a sequence of commits.

> **Definition 2.14 (Cherry-Picking and Pasting)**
>
> You can do copy a commit by **cherry picking** it and **pasting** it somewhere else.

2.4.2 Integrating Branches

The reason you want to have different branches is so that you have independent workflows that may hopefully be integrated into the master branch. So how does one actually perform this integration? There are two general ways to do this: a 3-way merge or a rebase. For both methods, we will use this example.

```
1   main      : A1 --- A2 --- A3
2                               \
3   feature1 :                   B1 --- B2 --- B3
4                                       \
5   feature2 :                   C1 --- C2 --- C3
```

> **Definition 2.15 (Fast-Forward Merge)**
>
> If you want to merge `main` and `feature1`, notice that `feature1` is really just ahead of `main` by some number of commits. The easiest way to merge is to add the additional commits in `feature1` to `main`. This is called a **fast-forward merge**, which we can call using
>
> ```
> 1   git checkout main
> 2   git merge --ff-only feature1
> ```
>
> Doing so will result in
>
> ```
> 1   main      : A1 --- A2 --- A3 --- B1 --- B2 --- B3
> 2                               \
> 3   feature1 :                   --- B1 --- B2 --- B3
> ```

```
4                                       \
5   feature2 :                   --- C1 --- C2 --- C3
```

and we can delete `feature1` since it's not needed.

In fact, if a fast-forward merge is possible, then calling `git merge feature1` will automatically do a fast-forward. We can explicitly set it to only attempt or never attempt fast-forward by adding the `-ff-only` or `-no-ff` flags.

Definition 2.16 (3-Way Merge)

If you want to merge two divergent branches, e.g. `feature1` and `feature2`, then a fast-forward is not possible. Rather, you want to choose to merge `feature2` *into* `feature1`. Git will rather do a **three-way merge** between the divergent node `A3` and the heads of the respective branches `B3` and `C3`. After you resolve conflicts, the tree should look something like

```
1   main     : A1 --- A2 --- A3
2                               \
3   feature1 :                   --- B1 --- B2 --- B3 --- M1
4                               \                        /
5   feature2 :                   --- C1 --- C2 --- C3 ---
```

Note that we could choose to merge `feature1` into `main` subsequently, resulting in both feature branches merged.

```
1   main     : A1 --- A2 --- A3 ------------------------------- M2
2                               \                              /
3   feature1 :                   --- B1 --- B2 --- B3 --- M1 ---
4                               \                        /
5   feature2 :                   --- C1 --- C2 --- C3 ---
```

Note that a three-way merge may result in a pretty ugly tree, especially if we are working with dozens of branches. What we would like to do is a three-way merge in a fashion that *looks like* a fast-forward merge. That is, we want the main branch to have a linear structure rather than a series of diverging and converging nodes. In fact, we already have the tools to do this. Let's revisit the interactive rebase again. We have seen that we can do an interactive rebase from a start commit by doing

```
1   git rebase -i <start-commit-hash>
```

What we would like to do is to rebase from a commit in a different branch.

Definition 2.17 (Rebase)

If we want to linearly merge `feature2` into `feature1`, this is called "**rebasing `feature2` *onto* `feature1`.**" We run

```
1   git checkout feature2
2   git rebase feature1
```

which means "take my current branch's unique commits and replay them on top of whatever branch I am rebasing on (in here, `feature1`)." This will result in

```
1   main     : A1 --- A2 --- A3
```

```
2                                    \
3   feature1 :                    B1 --- B2 --- B3
4                                                 \
5   feature2 :                                   C1' --- C2' --- C3'
```

where the `C'` are the same commits but with different hashes since they start from a different parent.

Example 2.2 (Updating Feature Branch with Changes from Main)

A common workflow you would do in a large project with multiple developers is as follows. Consider that you are working on `feature1` and another developer is working on `feature2`.

```
1   main     : A1 --- A2 --- A3
2                              \
3   feature1 :                  --- B1 --- B2 --- B3
4                              \
5   feature2 :                  --- C1 --- C2
```

Your friend pushes their changes to `main`, which leads to this structure.

```
1   main     : A1 --- A2 --- A3 --- C1 --- C2
2                              \
3   feature1 :                  --- B1 --- B2 --- B3
4                              \
5   feature2 :                  --- C1 --- C2
```

Your branch has diverged from main, so you will need to rebase your own branch onto main. You checkout to `feature1` and run `git rebase main`. After settling conflicts, your branch will look like the following, updated with the most recent commits from `main`.

```
1   main     : A1 --- A2 --- A3 --- C1 --- C2
2                                            \
3   feature1 :                                --- B1' --- B2' --- B3'
4                              \
5   feature2 :                  --- C1 --- C2
```

Example 2.3 (Converting a Merge Into Rebase)

Say that you already merged `feature1` and `main`.

```
1   main     : A1 --- A2 --- A3 ----------------------- M1
2                              \                        /
3   feature1 :                  --- B1 --- B2 --- B3 ---
```

You realized that you actually wanted to do a fast-forward so that it looks linear! How do you do this?

1. You first undo the merge.

```
1   git checkout main
2   git reset --hard A3
```

2. Then do the rebase of `feature1` onto `main`.

```
1   git checkout feature1
```

```
2    git rebase main
```

3. Then fast-forward to main to include `feature1`'s commits.

```
1    git checkout main
2    git merge --ff-only feature1
```

We will have this in the end.

```
1    main     : A1 --- A2 --- A3 --- B1' --- B2' --- B3'
2    feature1 : A1 --- A2 --- A3 --- B1' --- B2' --- B3'
```

2.5   Remote Trees

So far, we've talked about how you can use git to keep track of your edit history locally, but another benefit is to store these changes in the cloud. This is done through a third-party provider, and they are completely separate entities from git. The three most dominant ones are

1. *Github.* Owned by Microsoft and is the default for most open-source projects, with 100 million users.

2. *Gitlab.* Owned by Gitlab and is slowly gaining popularity due to better control of repositories and after Microsoft acquired github.

3. *Bitbucket.* Owned by Atlassian and used for private repositories in enterprise settings.

Again, all three platforms still use *git*, but the cloud storage is managed separately. All of these platforms provide a remote server that stores all of these git histories of millions of repositories around the world. The motivation behind the need of a remote workspace is that it is a common ground in which many developers can communicate and track the progress of their entire repository.

Definition 2.18 (Remote Repository)

The first step to setting up a cloud-based git tree is to place it on some server (IP address) in some directory with the proper permissions. This remote location containing the git tree and the corresponding code is called the **remote repository**, and it is encoded in either a URL or a SSH host link of the form

```
1    https://github.com/user/repo.git
```

For now, we will consider a local repository having at most 1 remote repository that it can communicate to.[a] Conventionally, the primary[b] remote repository goes under the alias **origin**.

None of the commands that we have introduced so far does anything to the remote repository. The first command we should know is how to set up one from scratch.

Definition 2.19 (Add Remote)

Given that git is initialized, we can initialize a corresponding remote repository by running

```
1    git remote add <remote-name> <remote-url>
```

Again, conventionally the `<remote-name>` is put as `origin`.

---

[a]When we get to forking, we will talk about multiple remote repositories.
[b]Again, for now it's really the only remote.

Great, we have set a remote repository up, but there is an extra step to do. Most synchronizations happens at a branch level rather than the whole tree itself, so what we would need to do for each local branch is create a corresponding remote branch and then *connect* those two so that git knows which branches to sync together. The local branch is called the **downstream** branch and its corresponding remote branch is called **upstream**.

In order to understand how the process of setting upstream branches is done, we introduce another variable in our local repository. In our local git tree, we have stated that for each branch B, there is a HEAD pointer living at the most recent commit, denoted as B/HEAD or just B.

Definition 2.20 (Remote References)

There is actually a second pointer called the **remote reference (ref)** in the local branch called origin/B (more generally <remote>/<branch>), which is a symbolic link that points to the head of the remote branch. They are git's way of keeping track of the state of branches in your remote repositories.

For some local branch, the existence of a remote ref tells you where the corresponding upstream branch is located (i.e. at <remote>/<branch>). If you do not see the remote ref, this means that you have not yet connected your local branch to its remote upstream, which can happen if the remote counterpart doesn't exist (i.e. you created a completely new branch) or you have never connected it. You can find all local branches and the remote references by calling

```
1   git branch -vv
2   # Shows all branches with their tracking info
3   # Example output:
4   # * main    abc123 [origin/main] Latest commit message
5   #   feature def456 [origin/feature: ahead 2] Some work
```

Definition 2.21 (Push with Set Upstream)

We consider two cases, where we do not see the remote refs at all.
1. Say that you have a local branch main with some committed changes, and the remote branch does not exist at all. Your tree will look something like this

```
1   Remote:
2   Local : A <- B <- C <- (main) D
```

2. Say that you have a local branch main with some committed changes, and the remote branch does exist but the upstream is not set.

```
1   Remote: A <- (main) B
2   Local : A <- B <- C <- (main) D
```

We can synchronize our local commits with the remote repo by **pushing** them. As we push, we also set the upstream with the -u or -set-upstream flag.

```
1   git push -u <remote> <branch>
2
3   # for example, if we want to push our current checked out branch to origin/main
4   git push -u origin main
```

This will set your remote refs, and in both cases we will have

```
1   Remote: A <- B <- C <- (main) D
```

```
2   Local : A <- B <- C <- (main, origin/main) D
```

Definition 2.22 (Push)

If your remote ref is already set and you have some further commits,

```
1   Remote: A <- B <- C <- (main) D
2   Local : A <- B <- C <- (origin/main) D <- E <- (main) F
```

then you can just push your changes with `git push`, which will push the commits up to HEAD and update the remote ref to HEAD.

```
1   Remote: A <- B <- C <- D <- E <- (main) F
2   Local : A <- B <- C <- D <- E <- (main, origin/main) F
```

In all of these scenarios, we have only seen cases when `origin/main` and `main` point to the same commit. However, remember that the remote ref is a *local* pointer to the remote repo's head, and so it is only updated every time your local repo interacts with the remote repo. Therefore, they can be different.

Example 2.4 (Remote Ref is Different From True Remote Head)

Say that you are `Local1` and a friend is `Local2`. We first start off with this.

```
1   True Remote: A <- B <- C <- (main) D
2   Remote1    : A <- B <- C <- (main) D
3   Local1     : A <- B <- C <- (main, origin/main) D
4   Remote2    : A <- B <- C <- (main) D
5   Local2     : A <- B <- C <- (main, origin/main) D
```

Your friend makes commits `E` and `F` and pushes them while you are working on commit `G`.

```
1   True Remote: A <- B <- C <- D <- E <- (main) F
2   Remote1    : A <- B <- C <- (main) D
3   Local1     : A <- B <- C <- (origin/main) D <- (main) G
4   Remote2    : A <- B <- C <- D <- E <- (main, origin/main) F
5   Local2     : A <- B <- C <- D <- E <- (main, origin/main) F
```

In this case, your friend's push will update the head of the remote main branch to `F` and update their remote ref to `F` as well. But you have worked only locally and have no interacted with the remote repo for a while, so your remote ref is still `D`. When you therefore try to push your changes, git will complain to you that your future ref and the remote head does not match. Since this creates divergent histories which splits from `D`, it will not let you push.

The scenario above is clearly a problem, and it stems from the fact that we have a remote ref. Why need remote refs at all if they seem redundant and at the same time they might introduce this problem? The answer is convenience. Imagine that there were no remote refs. This means that every time someone pushed to the remote main I would need to be notified that there were changes to main. However, I may not have access to the remote repo while I am working on my code, so I may still have the same problem when I am able to connect. The extra remote ref pointer allows for quick checking to see if my local branch matches its upstream. It turns out that if the upstream changed, git does indeed warn you about it as soon as possible.

Definition 2.23 (Fetch)

Say that for a branch, it looks like this, and your local remote is not updated with the true remote's latest commits.

```
1  True Remote: A <- B <- C <- D <- E <- (main) F
2  Remote1    : A <- B <- C <- (main) D
3  Local1     : A <- B <- C <- (origin/main, main) D
```

To update your remote ref to match the current upstream head, we can **fetch** it to get the following.

```
1  True Remote: A <- B <- C <- D <- E <- (main) F
2  Remote1    : A <- B <- C <- D <- E <- (main) F
3  Local1     : A <- B <- C <- (main) D <- E <- (origin/main) F
```

It comes in three commands.

```
1  git fetch <remote-url> <branch> # fetches from specific remote branch
2  git fetch <remote-url>          # fetches from all branches in a remote repo
3  git fetch                       # fetches from all branches of all remote repos
```

Definition 2.24 (Fast-Forward)

In the case above, note that even though the commits are synchronized to your local remote, the changes aren't actually reflected in your current code since HEAD has not moved. If you want head to also move to the most recent remote ref, then you can do a **fast-forward merge**.

```
1  True Remote: A <- B <- C <- D <- E <- (main) F
2  Remote1    : A <- B <- C <- D <- E <- (main) F
3  Local1     : A <- B <- C <- D <- E <- (origin/main, main) F
```

You can do this in git by checking out to the local branch you are on. After fetching, you should have the remote ref that you want to update your HEAD to, which you put in as an argument.

```
1  git merge --ff-only <remote-url/branch>
2  # e.g.
3  git merge -ff-only origin/main
```

Definition 2.25 (Pull)

The combination of doing a fetch to get the latest commits and update the remote ref, and then doing a (fast-forward) merge to update your HEAD pointer to the remote ref, is so common that we refer to doing these two things sequentially as a **pull**. Therefore, the two commands are the same.

```
1  git pull <remote-url> <branch>
2
3  git fetch <remote-url> <branch>
4  git merge <remote-url/branch>
```

```
1  git pull origin main
2
3  git fetch origin main
4  git merge origin/main
```

> **Definition 2.26 (Clone)**
>
> Finally, you can take an existing remote repository and **clone** it, which creates a complete copy of a
> remote repository. It does the following.
>   1. Does a `git init` to set up git.
>   2. Does `git remote add origin <remote-url>` to connect to the remote repo and set the `origin`
>   alias to it.
>   3. Does `git fetch origin` to fetch all branches of the remote repo, setting up the local remote
>   branches and their corresponding remote refs.
>   4. Does `git checkout -b main origin/main` which creates the local branch **main**, checks out
>   to it, and sets `origin/main` as its upstream.

2.6 Reflog

> **Definition 2.27 (Reflog)**
>
> Git's **reflog** is a super-history that records every change to your branch tips. It's essentially an undo
> tree for all git actions (sort of like a meta undo tree). Therefore, if you ever screwed something up,
> git allows you to undo your actions by undoing the entries in the reflog. It contains the following
> types of actions.
>   1. Branch Operations
>   - checkout: switching branches
>   - branch: creating/deleting branches
>   - merge: merging branches
>   - rebase: rebasing branches
>   - pull: pulling from remote
>   2. Commit Modifications
>   - commit: new commits
>   - reset: moving HEAD
>   - revert: reverting commits
>   - amend: amending commits
>   - cherry-pick: copying commits
>   3. Stash Operations
>   - stash: stashing changes
>   - stash apply: applying stash
>   - stash pop: popping stash
>   4. History Rewrites
>   - rebase -i: interactive rebase
>   - filter-branch: rewriting history
>   5. Reference Updates
>   - HEAD@n: moving HEAD pointer
>   - refs/heads/*: branch pointer updates
>   - refs/remotes/*: remote branch updates
>   - refs/stash: stash reference updates
>   6. Remote Operations
>   - clone: cloning repository
>   - fetch: fetching from remote
>   - push: pushing to remote
>   - remote add: adding remote
>
> You can access it using the following command.
>
> ```
> $ git reflog
> ab12345 HEAD@{0}: reset: moving to HEAD~1    # Most recent action
> ```

```
3  bc23456 HEAD@{1}: commit: Add feature X
4  cd34567 HEAD@{2}: rebase: onto main          # Rebase happened here
5  ef45678 HEAD@{3}: commit: Initial commit
```

2.7   Pull Requests and Forking

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