Learn Git the hard way, notes

Quoc-Quan Truong

November 2023

Contents

Chapter 1

GIT BASIC

Within the .git folder lay many files, of which the HEAD and the config.

The HEAD file is key: it points to the current branch or commit ID you are currently on within your Git repository.

The config file stores information about your repository's local configuration. For example, the branches and remote repositories your repository is aware of

If you want to look at the history of this repository, then run the git log command

The git status command tells you where the HEAD is pointing at and that if there exists something to commit.

Untracked files are files that Git acknowledges their existances, but are unaware by the repo. To make Git aware of them, the files needs to be added to the repository.

The git add Command The add command tells Git to start tracking the files in the local index

You have added a file to the index. It is ready to be committed to the repository.

Remember the four stages you looked at before.

You created your file ((1) local changes). You added/staged it to the index (2). Still, you have no history! Git has simply been made aware of the file, and you must make a commit to initiate Git's history.

The git commit command The git commit command tells Git to take a snapshot of all added content at this point.

The git diff command To see the changes that has happened to the files, use the git diff command.

You can commit changes to files and add at the same time by doing git commit -a

Chapter 2

CLONE A REPOSITORY

The git clone command The git clone command helps you create copies of Git repositories to work on.

The git reset command The git reset command helps you return to a previous or known state.

```
1    git log
```

This shows you a default history of the repository. Page through it a few times by hitting space or down. You will see how far it goes back. Hit q to stop viewing it, and return to the command line.

```
1    git log --oneline
```

Another way to view the log is one line per commit, which is much more concise and useful for many purposes. Obviously, some information is lost here.

```
1    git log --oneline --graph
```

You've added the –graph flag, and now you get a visual representation of the history. Parsing this graph can be tricky; don't worry about understanding it exactly. But keep it in mind. It is helpful if you ever have to figure out what went on in a repository's past.

Chapter 3

GIT BRANCHING

```
1    git branch
2    git checkout
```

This introduction shall discuss in length how branches are created in Git To create a new branch, type in:

```
1    git branch <branch_name>
```

After typing this, a new branch with name ¡branch_name¿ will be created. However, the HEAD will still be pointed to the initial branch. You can check it by using git status. To see all branches, execute git branch. Another flag of git log is –decorate. This gives you useful information about the reference names (branches and tags) at various commit points.

To go on to a branch, type in:

```
1    git checkout <branch_name>
```

Another flag of git log is added, the –all flag. –all shows all branches, not just the branch you happen to be on.

3.0.1 Detached HEAD

Detaching

The HEAD pointer can be moved to an arbitrary point. In fact, git checkout does exactly this. You can specify a reference (like master or newfeature) or a specific commit ID.

For example:

```
1    git checkout e36355ed00ac3af009d7113a9dd281c269a79afd
```

The detached HEAD state:

- Detached HEAD means that the HEAD pointer is not pointing towards a branch, but to a commit ID.
- You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

Chapter 4

TAGS

Tags are the same as branches, but they do not have history. Tags point to commits, but they do not change. Syntax:

```
    git checkout <branch_name>
    git tag <name_of_tag> # tag where you are.
```

Or you can tag wherever a branch pointer is pointed at in your repository, even if you are not on it.

```
    git checkout e36355ed00ac3af009d7113a9dd281c269a79afd
    git branch -f newfeature
```

The -f flag means –force. If a branch of newfeature already exists, then Git will not allow you to override it unless you use the -f flag.

```
    git checkout master git tag remember_to_tell_bob_to_rewrite_this newfeature
```

Verify the tags with git tag

Chapter 5

Conclusion: Git Branching

A branch is just a pointer

Remember these points: A branch is a pointer to the end of a line of changes. A tag is a pointer to a single change. HEAD is where your Git repository is right now. Detached HEAD means you are at a commit that has no reference (branch or tag) associated with it.

Chapter 6

MERGING

6.1 Introduction

```
1    git merge
```

In this chapter, you will learn about:

- Merge conflicts and how to resolve them
- Merges and log histories

6.1.1 Remembering branches

You've already covered the basics of branching in the previous chapter. As you will recall, branching gives you the ability to work on parallel streams of development in the same codebase.

6.1.2 Merging vs. branching

In a sense, merging is the opposite of branching. When you merge, you take two separate points in your development tree and fuse them together.

It's important to understand merging, as it's a routine job of a repository maintainer to merge branches together.

It's also really important to take this chapter slowly and make sure you understand every step. It's quite painful to wrap your head around merging properly. But once you do, your Git skills will improve.

6.2 Pre - merge

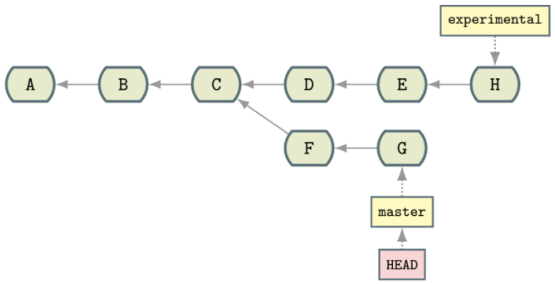

Figure 6.1: Pre-merge diagram

In this particular diagram, the repo is currently positioned at the tip of master **G**, indicated by the HEAD, which is pointing at **G**.

6.3 Post - merge

If the ‘experimental‘ branch is merged into master with a command:

```
1    git merge experimental
```

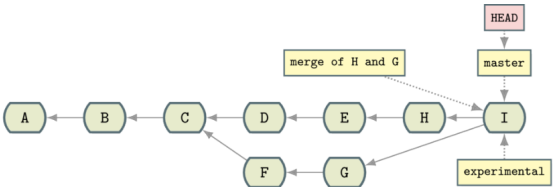

Figure 6.2: Post-merge diagram

A new change has been made (**I**). This change merges together the changes made on experimental with the changes made on master.

6.4 Merging, step by step

6.4.1 Step 1: Creating a repository, adding some files, and committing them

Code:

```
1    mkdir -p lgthw_merging
2    cd lgthw_merging
3    git init
4    echo A > file1
5    git add file1
6    git commit -am 'A'
7    echo B >> file1
```

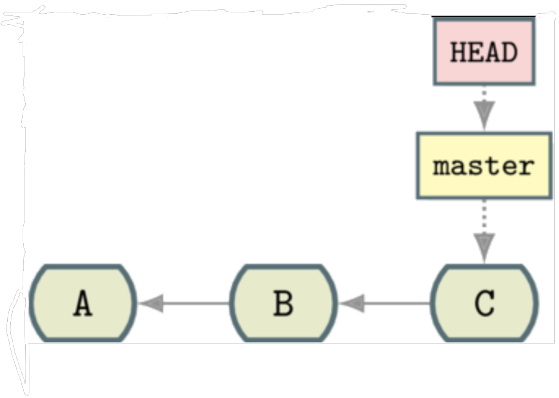

Figure 6.3: Repository after step 1

```
8     git commit -am 'B'
9     echo C >> file1
10    git commit -am 'C'
```

After this, the repository will look like this:

6.4.2 Step 2: Creating a branch and checking it out

Code:

```
1     git branch experimental
2     git checkout experimental
3     git branch
4     echo E >> file1
5     git commit -am 'E'
6     echo H >> file1
7     git commit -am 'H'
```

After this, the repository will look like this:

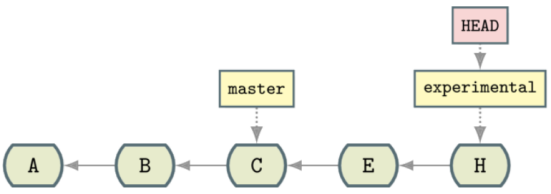

Figure 6.4: Repository after step 2

6.4.3 Step 3: Checking out master, and making more changes

Return to master, and make changes D, F and G:
Code:

```
1     git checkout master
2     echo D >> file1
3     git commit -am 'D'
4     echo F >> file1
```

```
5     git commit -am 'F'
6     echo G >> file1
7     git commit -am 'G'
```

After this, the repository will look like this:

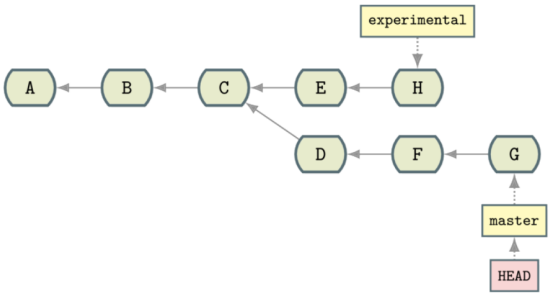

Figure 6.5: Repository after step 3

6.4.4 Step 4: Attempting to merge the master and experiment branch

Code:

```
1     git merge experimental
```

The result of this command is a merge conflict. Git will not allow you to merge the two branches together, because there are conflicting changes on the same line of the same file.

6.4.5 Merge conflicts

When you run a merge, Git looks at the branch you are on (here it is master) and the branch you are merging in (experimental) and works out the first common ancestor. In this case, it's point C, as that's where you branched experimental.

It then takes the changes on the branch that you are merging in from that first common ancestor and applies them to the branch you are on in one go. These changes create a new commit, and the git log graph shows the branches joined back up.

Sometimes the changes made on the branches conflict with one another. That means the changes altered the same lines. In this case, the D, F, and G of the master changed the same lines as the E and H of experimental.

Chapter 7

Git Stash

7.1 Introduction

Often when you are working, you want to return to a pristine state but you don't want to lose the work you have done so far. Traditionally, with other source control tools, you've copied files that have changed locally aside, then updated your repository, and then diffed and re-applied the changed files.

Stash away your changes

Git has a concept of the "stash" to store all local changes ready to re-apply at will. You can get very sophisticated with the stash.

```
[do some work]
[get interrupted]
git stash
[deal with interruption]
git stash pop
```

7.2 A hypothetical scenario

What if I want to work on a new feature, but there's an important update to the code that I need to pull from BitBucket? I don't want to commit my changes, because they're not ready yet. I don't want to lose them either. I want to be able to pull from upstream, and then come back to my current work later. Git stash is the answer.

First, we need to see the what changes we have made locally with *git diff*:

```
git diff
```

After that, we run *git stash* to stash away our changes:

```
git stash
```

A quick *git status* will show that we have no changes:

```
git status
```

What happened to the changes? They are stashed away. You can see them with *git stash list*:

```
git stash list
```

You can also see the graph with *git log*:

```
git log --graph --oneline --decorate --all
```

The resulting return shall give you an overview of what happened:

```
root@educative:/lgthw_git_stash_1# git log --oneline --graph --decorate --all
*   24f8db4 (refs/stash) WIP on master: 56a08f1 initial
|\
| * accb0c2 index on master: 56a08f1 initial
|/
* 56a08f1 (HEAD -> master) initial
```

Git stash has committed the state of the index and then committed the local change to the refs/stash branch and merged them as a child of the HEAD on a new refs/stash branch.

Don't worry too much about the details yet; it has basically stored all the changes you've made (but not committed), ready to be re-applied.

The stash branch is a special one that is kept local to the current repository. The "commit" message WIP on master and index on master is added automatically.

The master branch is still where it was before you stashed, and the HEAD pointer is pointed at the master branch.

Other work can now be done (in this case, pulling the latest changes from a remote. Remember, we will cover remotes in a later section) without concern for whether it conflicts with those changes.

7.3 Retrieving the stash list

To retrieve the stash list, you can use the *git stash list* command:

```
git stash list
```

7.4 Popping stashed work

Once ready, you can re-apply those changes on the same codebase by running git stash pop:

```
git stash pop
```

It "pops" the zero-numbered change off the stash stack and restores the changes that was stashed, applied to wherever we've ended up.

7.4.1 What's a Stack?

A stack is a computer science concept of a series of items stored in such a way that you can "push" an item to the top of it for any number of times and then "pop" an item off the top until it's empty. When you pop, you only get the most recent item that was pushed on. This means that when you pop your Git stash, you get the most recent one you pushed (which is likely to be what you wanted).

Chapter 8

Git add interactive

This chapter will cover the git add -i command and the following topics:

- Interactively adding files in Git
- Splitting hunks
- The difference between staging and committing

8.1 Recap

Previously, you've learnt about the four stages of working in Git:

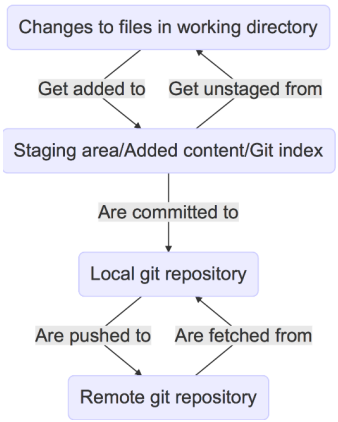

Figure 8.1: The four stages of working in Git

So far you have been shown the difference between adding (staging) and committing, but this still leaves a lot of room for confusion. What's the point of the seperation between the two? Why not just commit everything?

8.2 The Add Interactive command

The git add -i command is a powerful tool that allows you to interactively add files to the index. It allows you to add files in hunks, which means that you can add parts of a file, rather than the whole file.

8.3 The git add -i command

Let us demonstrate how you might want to use this with a simple example.

```
mkdir lgthw_add_i
cd lgthw_add_i
git init
echo 'This is file1' > file1
echo 'This is file2' > file2
git add file1 file2
git commit -am 'files added'
cat > file1 << END
> Good change
> This is file1
> Experimental change
> END
cat > file2 << END
> All good
> This is file2
> END
```

Now run the following, and input the characters (or just "enter"/"return") at the appropriate points:

```
git add -i
What now> p
Patch update>> 1
```

Note that in the next line, you only need to input the "enter"/"return" key. Just hit "return".

```
Patch update>>
Stage this hunk [y,n,q,a,d,/,s,e,?]? s
Stage this hunk [y,n,q,a,d,/,s,e,?]? y
Stage this hunk [y,n,q,a,d,/,s,e,?]? n
What now> q
git status
git diff
```

Now you have staged the good change, but have not lost the other changes you made. This gives you more granular control over the changes committed.

A lot went on in that last block of output so it's worth reading over it carefully.

First, you were presented with a set of choices:

```
1: status   2: update   3: revert   4: add untracked
5: patch    6: diff     7: quit     8: help
```

8.4 Patch

We're only going to focus on "patch" (number 5) here. There's no point exhaustively listing all the choices and their meanings, as most of them are self-explanatory, and I did not feel the need to use the rest of them.

8.5 Hunk

As you have probably figured out already, a "hunk" is a contiguous (or nearly contiguous) section of a diff.

You're presented with a set of numbered changes. There were files that had changes to them presented to you.

```
            staged     unstaged path
  1:   unchanged        +2/-0 file1
  2:   unchanged        +1/-0 file2
```

8.6 Staged

The first "staged" column tells you what has been staged so far.

8.7 Unstaged

The second "unstaged" column tells you how many lines have been added/removed. In the above example, two lines have been added and none removed.

8.8 Path

The third is the "path" of the file.

An asterisk (*) indicates that the option is the chosen one. So by hitting that number, followed by the enter/return key, you only need to further hit the enter/return key once to choose the first hunk. Then you are presented with the hunk itself and a bewildering series of options:

```
  diff --git a/file1 b/file1
  index 6a00e12..014f6e4 100644
  --- a/file1
  +++ b/file1
  @@ -1 +1,3 @@
  +Good change
  This is file 1
  +Experimental change
  Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

8.9 Splitting

At this point, you chose s, which stands for "split" the hunk.

If you're not sure what's going on, you can choose the ? option here, which explains what the various options mean. If I'm stuck, I can never remember what they all do, so I depend heavily on ?. Read through the options now, and make sure you understand them all.

Once split, you can "stage" the hunks one at a time by choosing y for "yes" when prompted.

If you are happy with the changes, you can commit all the changes you have made.

8.10 Why split hunks?

Why is this splitting useful? It's most commonly used to avoid committing lines you might want to keep for local development but not persist in the repository history.

A typical example of this is printed debug lines that you don't want to get to production but want to keep for local development. Another example might be chunks of notes that only make sense to you while you are developing.

8.11 Why stage at all

Committing will commit all the changes you have staged. What is the point of staging then? It is to confirm that you want to commit some changes made locally, but not others. These changes are added to the index (as opposed to the repository).

Remember: index == staging == adding

By committing, it commits your staged changes to the local repository (as opposed to adding). After which, these commits can be pushed to remote repositories.

If (like me) you run git commit -am "your commit message" frequently, then you skip over these steps, which can result in commits with stray lines that you would not want to be part of the history.

The -a flag stands for "automatically add". Confusingly, the -a flag is aliased to –all, even though not all files are necessarily added; only the already-added ones are.

Chapter 9

git reflog

References to your history in a sequential format.

¡¡After updating a few commits to the git project¿¿

You accidentally:

```
    git checkout HEAD^
    git branch -f master
    git checkout master
    git log
```

And you find out that your latest commit has *disappeared*.

You slayed. But badly.

You have fully reverted the master branch to where it was before. Even git log –all does not show it because it's not on a branch.

Retrieve commit This is where Git's reflog can help.

Git reflog records all movements of branches in the repository. As with git stashes, it is local to your repository.

The reflog is called that because it's a "REFerence LOG".

Git's reflog is a history of the changes made to the HEAD (remember the HEAD is a pointer to the current location of the repository).

```
    git reflog
```

To restore state, use git reset.

If you run git reset:

```
root@educative:/lgthw_reflog# git reflog
86f146a HEAD@{0}: checkout: moving from 86f146a8e8eb8f730ee7b090e760d116f2a8f018
    to master
86f146a HEAD@{1}: checkout: moving from master to HEAD^
c11d00d HEAD@{2}: commit: second commit message for file1.1
86f146a HEAD@{3}: commit (initial): first commit
```

this should be the output.

You are returned to where you were.

The –hard flag updates both the index (staging/added) and the working tree that you saw previously.

The relfog contains references to the state if the repository at various points, even if those points are no longer apparently reachable within the repository.