Beginning Git Version Control

Essential tips & tricks you need to get Git up and running.

By Makzan.

Beginning Git Version Control

Essential tips & tricks you need to get Git up and running.

By Makzan.

This book is for sale at http://leanpub.com/beginning-git-version-control

This version was published on 2021-04-01.

Table of Contents

Introduction

Hi there, this is Thomas Mak, a.k.a. Makzan.

In this book, we will go through the essential concept of Git version control.

Why this git book?

Git is getting more and more powerful. It means more and more commands for us to control every little detail of code changes. That's good. But as a new learner, I trim the content to provide only the essential concepts and the best practices.

For example, I recommend git fetch + git merge always instead of git pull. You'll learn all these commands elsewhere, but in this book, I tell the why and their pros and cons. (In chapter 5)

About the Author

I am Thomas Seng Hin Mak, but you may find me on the Internet as "Makzan".

Thomas has written books since 2010. His books include Flash Multiplayer Virtual World, HTML5 Games Development, and Mobile First Web Design.

Thomas has been using git version control since 2009. Before moving to git, I used SVN in a game development team. I changed to use git personally and then encouraged the whole team to move to git. Afterward, I tried different git version control flow in different teams, including HTML5 games development, book writing, iOS app development, and web projects. I also used different remote repository approaches, such as GitHub, Bitbucket, and self-hosted git.

Chapter 1: Basic Command

In this chapter, we will explore the fundamental things to get started using git version control.

We will learn to:

- Install and init a git repository.
- Commit the changes as snapshots into the repository.
- Write helpful commit messages.
- View the status of the current changes.
- View the history log of the commit snapshots.

Install and Config

This section covers the git program installation and configuration.

Installation

Linux and Mac already come pre-installed git clients. Usually, the built-in version is new enough and we can directly use it.

For Windows, we will need to install a git client.

There are different Git clients in the market. Some have a graphical user interface (GUI). Some are command-line interfaces (CLI).

In my daily workflow, I prefer to use a command-line interface. It allows me to operate faster without leaving the keyboard. For those common commands, we can create aliases to type fewer keystrokes.

Occasionally, we will have GUI Git client installed. For example, the Visual Studio Code editor comes with Git support built-in. For those GUI, the application concepts are the same, usually, just the commands become a button in the interface.

What we learn in this book applies to both the command-line interface and graphical user interface.

Configuration

After installing the git program, we need to config it. At least, we need to set the username and email address so that we can know who is making changes to the code.

In the command prompt, run the following command.

NOTE: This is my information, you may want to set yours.

Set up a username and email:

```
$ git config --global user.name "Thomas Seng Hin Mak"
$ git config --global user.email "mak@makzan.net"
```

This is the information that we can trace who has been working on what later when we need to collaborate the code with other members of the team.

NOTE: You can set project-specific name and email by running the git config command without the `--global` option.

Init and First Commit

In order to get familiar with git. We start by creating a new web project to create some code inside. We will use some HTML, CSS, and JS code for demonstration purposes. But Git works for any plain text format indeed.

In the following steps, we will learn the very basic commands to control a project folder with git. Then we will make changes to the project files and save the snapshots into the git repository.

Time for Action—Project initialization

1. Create a folder for our project. Let's name it "Sample Project".
2. Create some files inside the project folder. Optionally, put some dummy content in the files. For example, a `README.txt` with a title and current date is enough for our practice.
3. Run the git init command.

```
$ git init
```

The git init initials the current folder as a git-controlled repository.

4. Run the `git add .` command to mark the files to be version controlled.

```
$ git add .
```

`git add` requires the input of files to be added. The `.` means adding all files in the current directory.

5. Run the `git status` to preview what we are going to commit.

```
$ git status
```

6. Run the following `git commit` command to commit the marked files into the repository. This is the step that really saves the changes as a snapshot.

```
$ git commit -m "First commit."
```

NOTE: As a beginner, it's easy to forget the commit message. `git commit` requires the message to exist, but sometimes beginners try to avoid it by using `-m ""`. This will lead to rejection from Git with the message *"Aborting commit due to empty commit message."*

Further steps

We have learned the `add` and `commit` commands. By using these commands, now you can make changes to your files/source code. When you feel that you need a snapshot, you can git add the changed files, and `git commit` them into the version-controlled repository.

Selectively git add files

In the project initialization steps we just executed, we used `git add .`. We did that to include all files in the project folder to the first git commit.

But most beginners only use `git add .` for the rest of their life. This command commits all changes and without reviewing them. Please use it carefully.

Having too much unrelated code committed to the same commit will make our future harder. We will not be able to review our history easily and move back to a certain history point if our commit history is messy.

Sometimes, we need to add only certain changes instead of all. Most likely this happens when we have worked on two features at the same time. It may not be intentional, but we may on one hand implement the current feature, on the other hand, fixing a typo or a minor CSS style on the fly. This results in having changes that belong to two different purposes: the planned feature implementation, and minor changes not related to the feature.

We can selectively add only certain files by using `git add <filename>` or `git add <folder path>`.

Then we `git commit` with those added files only, and repeat to commit the other files.
Occasionally, we will have two feature changes at the same file, in that case, we will need to add only part of the code within a file. To do so, we will need `git add -i`, which stands for git add interactively. We will have an example of `git add -i` in chapter 3, essential techniques.

Note: The `git add .` means adding all the changes in the current directory. If you are sure that you need all changes into the next commit, you can also combine the add and commit into one command: `git commit -am "<MESSAGE>"`.

Ignoring files with .gitignore

We do not want to add everything into version control. Some files are never tracked. Usually we would like to ignore the build and output folders. We also want to ignore the private key, passwords for databases, or secret hashes for applications.

We can create an `.gitignore` file at the base of the project folder. The `.gitignore` file allows us to list files that we don't want to track at all. These files won't appear as "Untracked files" when viewing the `git status` result.

You can ignore an entire folder by specifying the folder name. Or you can ignore certain types of files with the wildcard character. For example: `build/*.exe`.

NOTE: You can still force adding files that are ignored by using the `git add -f` option.

Writing git commit message

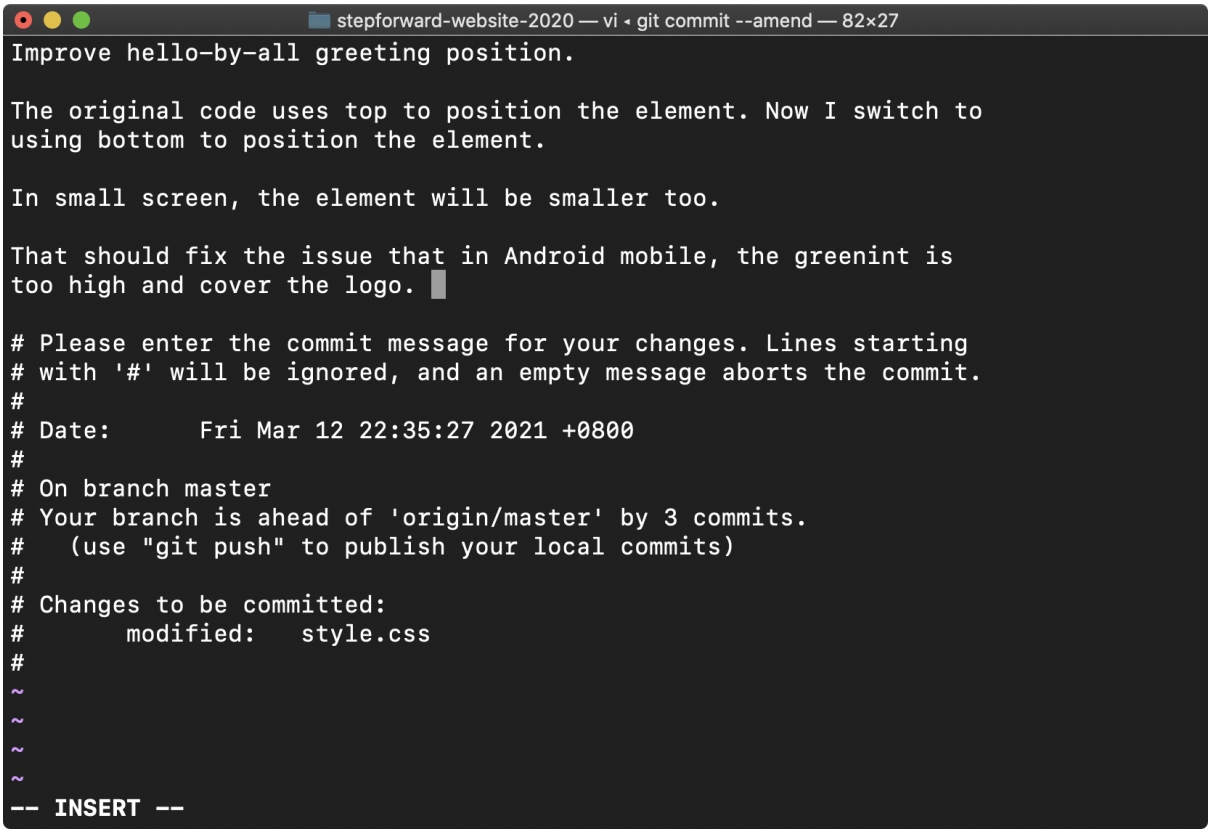
We need a message for every commit. When we commit with `git commit -m "One line message here"`, we create the commit with a one line remark. When we commit with just `git commit`, an editor is opened and we can write long commit message there.

In the future, we will need to refer to the history of the code and have a summary on what each commit does. Here are some tips on writing the commit messages.

1. Use the present tense.
2. Begins with a verb. Just omit the subject because it is always “I”, the developer.
3. Use the first line as a summary.
4. For a long message, use the text editor instead of the one-line option `-m`. For a long message, add two line breaks and write the details after the first summary line.

It’s not required to mention file names in messages. The file changes can be viewed in the log. No need to mention the file unless you need to refer to the file in your message.

Here is an example of a long message.



Current Changes Status

We can observe the current status by using the `git status` command.

For example, when we run the `git status` command in a directory with changes, we will see the following result.

```
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

      modified:   app/assets/javascripts/application.js
      modified:   app/views/documents/_form.html.erb
      modified:   app/views/documents/show.html.erb

Untracked files:
  (use "git add <file>..." to include in what will be committed)

      app/assets/javascripts/components/last-save.js.jsx
      vendor/assets/javascripts/moment.js
```

For the status, you may observe that there are two stages of files: (Actually there are three, we will go through it later)

1. Un-staged file changes, in red color.
2. Staged file changes, in green color.

The third stage is committed to file changes, which are stored as a snapshot in the git repository and hidden in the `git status` command.

Help message

In the git status, there are helping messages that guide you to the next actions you can perform on those changes.

For example, `git status` tells you that you can use add to stage the changes. And you can use checkout to un-stage the changes.

Short version

There is a short version of the status. If you are really familiar with the status output and what you can do with those changes, you can turn off all the unnecessary output by using the option `--short`, or `-s`.

Commit log

The log shows the history of the commit snapshots.

There are several log options. The default one isn't actually that useful.

I usually view my log with the following options.

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

Explaining the option

So what do those options mean?

The `--oneline` shows the simplified version.

The `--graph` shows the commit relationship path on the left. So that the sequence of log becomes a committed path.

The `--all` shows all branches, including the remote branch. Otherwise, the git log only shows the current branch.

The `--decorate` shows the branch pointers to the commits. In the latest Git version, the `decorate` option is on by default.

The `--color`, as the name suggests, makes the log output colorful and easier to read. In the latest Git version, the `color` option is on by default in some systems.

Git alias configuration

It's a very long command, right? So I usually set it as an alias command. This can be done by setting the alias in the .gitconfig file.

We can set aliases in the `.gitconfig` file. There are several places where we can put the config file. Most likely, you will want to create this file in your home directory. That applies to all your terminals and projects under your user account.

For your reference, here is my alias to show all branches with one-line log commits and graph view.

```
[alias]
    st = status
    ll = log --all --oneline --graph --decorate --color
```

Chapter 2: Behind the scene

In this chapter, we learn how git works behind the scene. Specifically, we will learn:

- How git manages snapshots.
- How git manages branch pointers.
- The 3 stages of tracking changes.

The .git folder

After git init the current folder, git creates a `.git` folder in the project directory.

The `.git` folder contains everything git needs to maintain the repository.

Note: please do not change any files in this folder when you browse it. It is automatically managed by the Git software.

Removing the entire Git version history in a project

If somehow you want to remove the entire Git version history for a project, you can remove the entire `.git` folder.

If this project folder is the only copy that contains the `.git` folder, your version history will be forever lost. But if you have backups, or remote repositories for the project, you are safe to restore the history if it is an accidental operation.

Copying the project folder with version history

On the other hand, if we need to clone the project folder and preserve the version history. We can just copy the project folder, which already includes the `.git` folder.

But if you are not copying the outer folder. Instead if you are just picking the files and folders inside the project folder to copy. Please make sure you also copy the hidden `.git` folder.

Snapshots links

When we commit changes into a snapshot, git saves the diff. Each snapshot points to its parent. And there are branch pointers pointing to the last commits for each link list..

So given any snapshot, Git can trace the changes back to their root, first, snapshot. Git can construct the files by applying the diff patches from its root snapshot.

You can foresee from their mechanism that this design approach is good for source code and plain text files. It uses text patches to maintain the version changes.

But if we do version control with binary files, such as `.psd` format, it needs to store the whole file for each commit.

Commit hash

For each commit, Git generates a SHA1 hash as the commit identifier. The hash is based on the current time, the author, the parent commit ID, and some other information to generate the SHA1 hash.

The commit hash is generated by Git automatically. We get a new hash each time. For example, if we created a commit and then delete it and commit again, we will get a different commit hash.

When we refer to a specific commit, we can use the beginning several characters to refer to it. For example, given the following commit:

```
commit b0d3c42d003152d1195a299ee5a11bb7b82c07d4 (HEAD -> development)
Author: Thomas Seng Hin Mak <mak@makzan.net>
Date:   Wed Mar 10 20:55:54 2021 +0800

    Example commit.
```

We can use `b0d3c4` to refer to this commit. The shortest length is 4 characters. Show we may be able to use `b0d3` to refer to this commit, if there is only one commit that begins with these 4 characters. Otherwise, we may need to specify more characters to refer to a commit.

By the way, we can use `development` to refer to this commit, because the branch is pointing to this commit currently. The branch may move. So `development` will refer to whichever commit it is pointing to now.

Branches references

Git manages branches by using reference pointing to the commits. This mechanism allows instant branch creation without cloning the whole project folder.

If we take a look at the `HEAD` file, it stores the current commit hash.

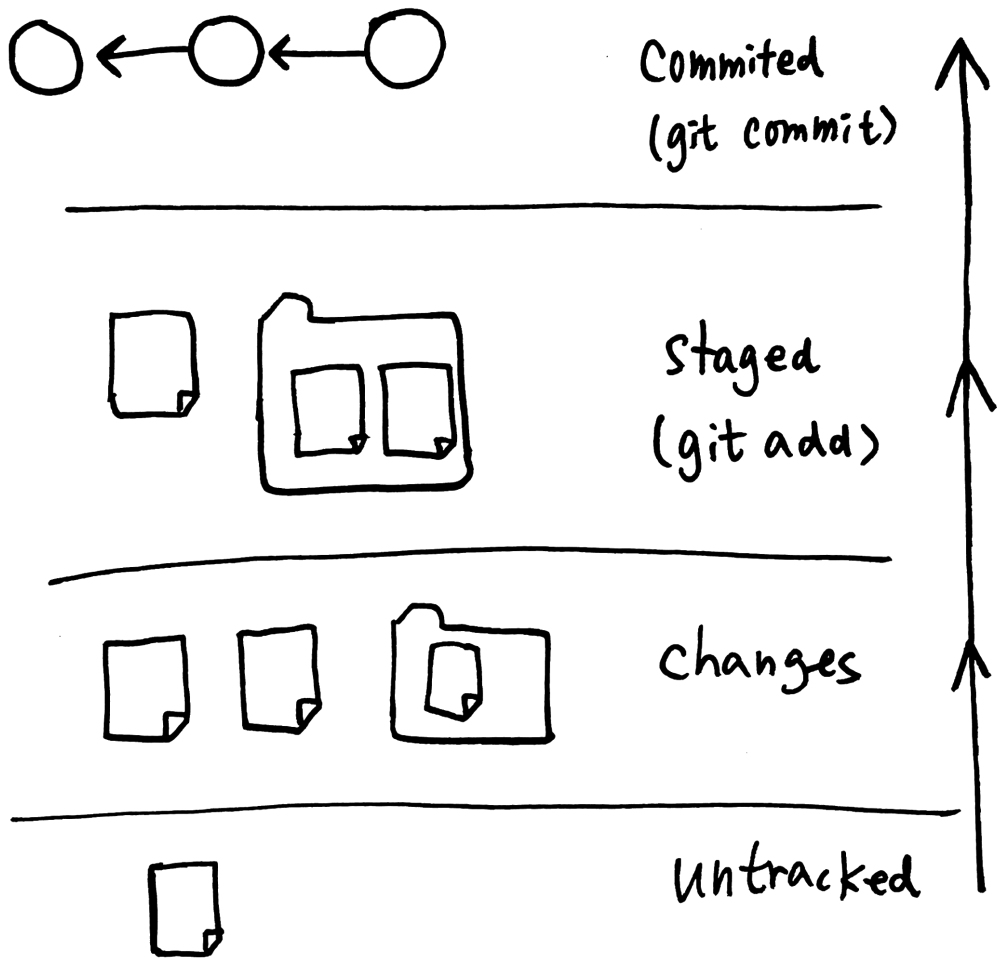
If we go into the `refs` folder, we see a file named `master`, which is our only branch now. When we create new branches later, those branches will have references stored in this folder for Git software to manage.

That is why creating a branch in Git is instant fast. It is just a cloning of a plain text file that stores the same commit hash.

The different stages of changes tracking

There are several stages for a file. They are:

1. untracked files
2. tracked files with non-staged changes
3. tracked files with staged changes
4. committed changes



You can see the stages of the changes when you `git status` your project folder.

```
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
```

```
directory)

        modified:   app/controllers/folders_controller.rb
        modified:   app/views/folders/show.html.erb
        modified:   config/routes.rb

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        app/views/folders/_public_list_document.html.erb
        app/views/folders/public_folder.html.erb

no changes added to commit (use "git add" and/or "git commit -a")
```

Untracked files

The file is usually new and hasn't been tracked yet.

NOTE: If you don't want a specific file to be tracked. You should add it to `.gitignore` file. Don't just leave that file there showing up every time when you view the `git status`. It's because we want to make sure we can have a clean state of working directory.

Here is how a clean git status looks like:

```
On branch master

nothing to commit, working directory clean
```

Changed but unstaged

The file is tracked and there are changes since the last commit.

Staged

The changes are marked and will be added to the repository in the next commit.

Committed

The changes are stored in the git repository. You can't see it in `git status`. But you can view them by using `git log`.

When we learn how to undo changes by using `git reset`, we will need the concept of these stages, but in reverse order.

Git tracks files, not folder

Git doesn't track folders. This may bother you when you create empty folders as placeholders in a project. For example, an empty images folder in the beginning of a web project.

If we need to track folders, we usually add a hidden file there. Communities often use an empty `.gitkeep` file for this purpose. But it's not a standard. You may pick any file name to commit the file and track the folder path.

For example, if we create a folder named `images` and we call `git status`, we won't see the newly created folder. But if we put a file there, we will see it.

```
$ mkdir images
$ touch images/.gitkeep
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        images/
```

Chapter 3: Essential Techniques

We have learned the basic commands to keep adding changes as snapshots. It's time to learn more techniques.

Essentially, we will learn to:

- Create branches
- Checkout a specific commit
- Merge branches
- Check the diff between commits
- Keeping a clean commit history
- Git add interactively to select only the changes we want to commit

Creating branch

We can create a branch by using `git switch -c`.

For example, the following command creates a new feature branch.

```
git switch -c new_feature
```

This creates a new branch. But actually, nothing really changes here until we create new commits. When you git log the commits, you still see the same commits graph. The only difference is now there is a new_feature branch name that appears where the HEAD at.

We will see how the branch works when we make changes and commit it. Afterward, the git log shows a commit-graph that contains the new_feature and the master, which points to two different commits. This is because we created a branch from where the master pointed at and then moved forward with new commits.

To make things more interesting, now we go back to where `master` points at. To go back, we `git switch master`.

Now we make some other changes on master and commit it. When we git log to see the commit-graph, we see two commits are diverse into two directions, both from the same parent commit but now one is master and the other is `new_feature`. The HEAD points to the current commit. When we `git switch new_feature` again, we can see the HEAD now points to `new_feature`.

NOTE: If you are using an older version of Git client, you may not have the `git switch` command. You can use `git checkout branch_name` to switch between branches and `git checkout -b new_branch_name` to create a new branch.

Check out a specific commit

We can check out specific commits by using the following command.

```
git checkout <commit hash>
```

The commit hash is a 40 characters length of alphanumeric. We don't necessarily need to write all 40 characters. Just include the first few characters to allow git to identify that commit. For example: `git checkout a1b2c3` would be enough most of the time to identify a unique commit.

When you check out a certain commit. You'll notice two things:

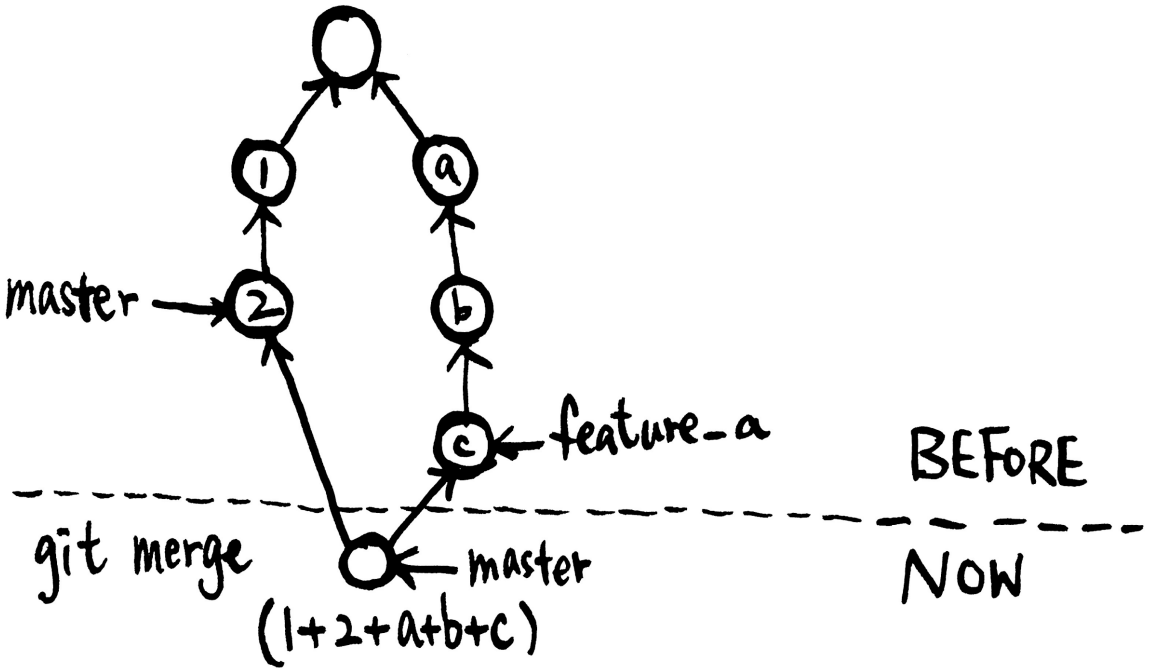
1. The git checkout command prints a large paragraph of text. It is to warn you that you have checked out a branch without any pointers. That means that is a detached branch without any branch name. We usually check out and work on branches instead of directly referring to commit hash.
2. The files in the working directory change to the state of that commit. Don't worry, our commits are safe and we can check out our latest code by doing `git checkout master`.

Merging branches

After we created different branches, we need `git merge` to merge the current branch into the other branch.

```
(master branch)$ git merge feature_a
```

Its result is a recursive merge. This happens when we merge 2 branches that have a different commit history.



Afterward, we may check out the `feature_a` branch and merge it to `master`.

```
$ git checkout feature_a
$ git merge master
```

Its result is a fast-forward merge. This happens when we merge branches that are in the same timeline.

Git diff

Diff between unstaged changes and staged / committed changes:

```
git diff
```

This compares the current non-added changes with the last commit in the current branch.

```
git diff --cached
```

`--cached` compares the staged changes (added but not committed) with the last commit in the current branch.

```
git diff branch_a branch_b
```

Why need to diff before committing?

We want to check what's the changes between the latest code and the last committed code.

Assume the following `git status` result. We see that there are two file changes. One file is `last-save.js.jsx` and the other one is `app.css.scss`.

An example of git status with two changes files:

```
$ git status
On branch master

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

        modified:   app/assets/javascripts/components/last-save.js.jsx
        modified:   app/assets/stylesheets/app.css.scss

no changes added to commit (use "git add" and/or "git commit -a")
```

Then, we use the `git diff` to check the code changes before committing them into the repository.

In the diff, we find that the 2 files are actually 2 different things. One file is a React JS view component and the other is a CSS styling improvement. We didn't commit before making changes to another feature.

We add 1 file and commit it, then we commit another file. And result in two different commits for two different purposes.

Maintaining a clean commit history: git add with status check

Before we commit, we may want to clean the commit to make it contain only what's relevant. So each commit should make changes for only one purpose.

Here is a typical `git add` workflow that contains a reveal and double check process to ensure that only the changes we want to commit are added.

1. git status to see what has changed since the last commit. Reveal the list to check if all changes are for the same purpose.
2. `git add .` if all the changes are for one purpose. or `git add <folder/file>` to select the changes to separate different purposes into different commits.
3. `git status` to reveal the added (staged) changes. Those are the changes that will be committed.
4. `git commit -m` to commit the change with one-line comment. Or `git commit` and write a long message if it is an important commit.
5. Repeat step 1 to 4 until all changes are committed.

Further double check strategy

It is also a good habit to check the changes difference with `git diff`.

If the changes involve many files and different parts of the same files that belong to several purposes, I suggest to also use `git diff` and `git diff --cached` to ensure the added changes are what we want to commit.

Git add interactively

`git add` comes with an interactive mode. We can micro-manage which part of which files to be staged in this mode, down to line-by-line level.

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +2/-2 index.html

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

What now> 2
           staged     unstaged path
  1:    unchanged        +2/-2 index.html
Update>> 1
           staged     unstaged path
* 1:    unchanged        +2/-2 index.html
Update>>
updated one path

What now> 1
           staged     unstaged path
  1:        +2/-2      nothing index.html

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

You can use the command to add files by choosing their index number.

You can also use `patch` to add line by line inside a file. In this mode, we will be asked what to do for each hunk. A hunk is several lines of code in the file. We can decide to stage that hunk or not. We can also split the hunk into smaller hunks. We need to split when Git selects too many lines of code for us and the hunk contains more than one purpose.

```
Stage this hunk [y,n,q,a,d,/,s,e,?]? ?
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk nor any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk nor any of the later hunks in the file
```

```
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

Here is a full example that we add parts of the `index.html` file by using `git add -i`.

```
What now> s
           staged     unstaged path
  1:    unchanged        +2/-2 index.html

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now> p
           staged     unstaged path
  1:    unchanged        +2/-2 index.html
Patch update>> 1
           staged     unstaged path
* 1:    unchanged        +2/-2 index.html
Patch update>>
diff --git a/index.html b/index.html
index 5ba3e85..df49622 100644
--- a/index.html
+++ b/index.html
@@ -1,5 +1,5 @@
-This is HTML Sample.
+This is HTML Sample for my Git course.

 Title: This is a sample.

-Footer: Copyright.
\ No newline at end of file
+Footer: Copyright 2016.
\ No newline at end of file
Stage this hunk [y,n,q,a,d,/,s,e,?]? ?
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk nor any of the remaining ones
```

```
a - stage this hunk and all later hunks in the file
d - do not stage this hunk nor any of the later hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
@@ -1,5 +1,5 @@
-This is HTML Sample.
+This is HTML Sample for my Git course.

 Title: This is a sample.

-Footer: Copyright.
\ No newline at end of file
+Footer: Copyright 2016.
\ No newline at end of file
Stage this hunk [y,n,q,a,d,/,s,e,?]? s
Split into 2 hunks.
@@ -1,4 +1,4 @@
-This is HTML Sample.
+This is HTML Sample for my Git course.

 Title: This is a sample.

Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? y
@@ -2,4 +2,6 @@

 Title: This is a sample.

-Footer: Copyright.
\ No newline at end of file
+Footer: Copyright 2016.
\ No newline at end of file
Stage this hunk [y,n,q,a,d,/,K,g,e,?]? d

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now> s
           staged     unstaged path
  1:        +1/-1        +1/-1 index.html
```

```
*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now> q
Bye.
$ git status
On branch index_b
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

        modified:   index.html

$ git diff --cached
diff --git a/index.html b/index.html
index 5ba3e85..68b050a 100644
--- a/index.html
+++ b/index.html
@@ -1,4 +1,4 @@
-This is HTML Sample.
+This is HTML Sample for my Git course.

 Title: This is a sample.

$ git diff
diff --git a/index.html b/index.html
index 68b050a..df49622 100644
--- a/index.html
+++ b/index.html
@@ -2,4 +2,4 @@ This is HTML Sample for my Git course.

 Title: This is a sample.

-Footer: Copyright 2019.
+Footer: Copyright 2021.
```

Chapter 4: Flow Control

There is a flow control approach that we can follow to represent the stable state of commits. Usually the stable branch moves slowly forward. A compilable working branch is based on the stable branch and moves forward faster. We may call it nightly build. Based on the working implementation, there are feature branches that are being implemented and often not working.

Flow control is the control of different levels of stability by using different branches convention to represent them. And the ability to switch between them in different scenarios.

The flow of branches, from stable to work-in-progress

Not available in preview.

Stable production branch

Not available in preview.

Hotfix branch

Not available in preview.

Working Development branch

Not available in preview.

Feature development branch

Not available in preview.

Chapter 5: Remote and Collaboration

The beauty of git version control is that it enables flexible collaboration between team members or even external developers.

Different roles in Git team

Not available in preview.

Team Hierarchy

Not available in preview.

Creating a remote repository

Not available in preview.

Adding remote branch

Not available in preview.

Cloning a remote branch into a new local folder

Not available in preview.

Git push

Not available in preview.

Deleting a remote branch

Not available in preview.

Git pull and git fetch

Not available in preview.

Git and GitHub

Not available in preview.

Chapter 6: Handling Conflicts

Not available in preview.

How does conflict exist

Not available in preview.

Resolve Conflicts

Not available in preview.

Chapter 7: Undo Changes

In git, we have different undo approaches for different scenarios.

It depends on what we need to do.

For example, if we want to:

- Amend the last commit
- Reset several commits
- Revert changes after pushing to a remote branch
- Rebase commits into another commit.

Stash working directory

Not available in preview.

Git reset

Not available in preview.

Undo a hard reset with reflog

Not available in preview.

Amend last commit message

Not available in preview.

Revert changes from remote

Not available in preview.

Grouping multiple commits into one commit with git reset

Not available in preview.

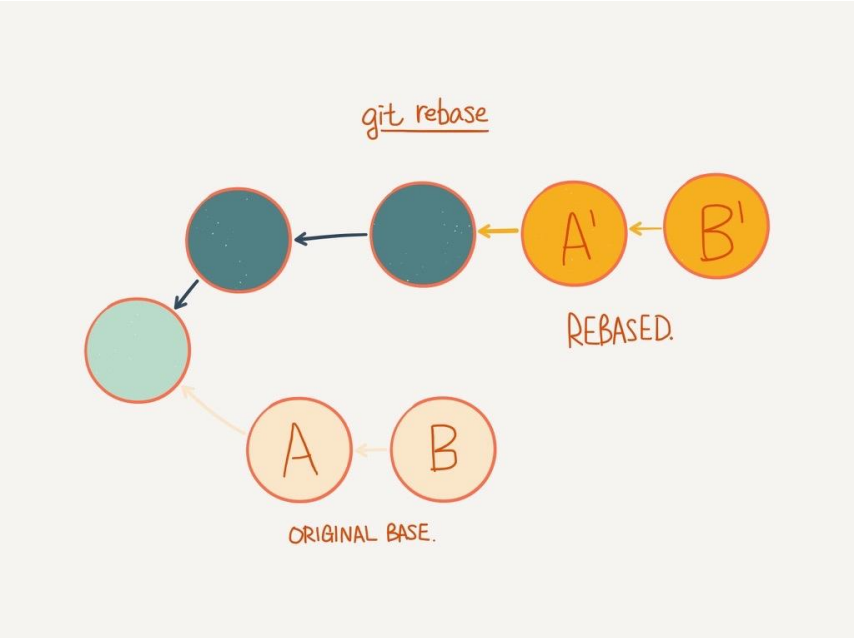
Git Revert

Not available in preview.

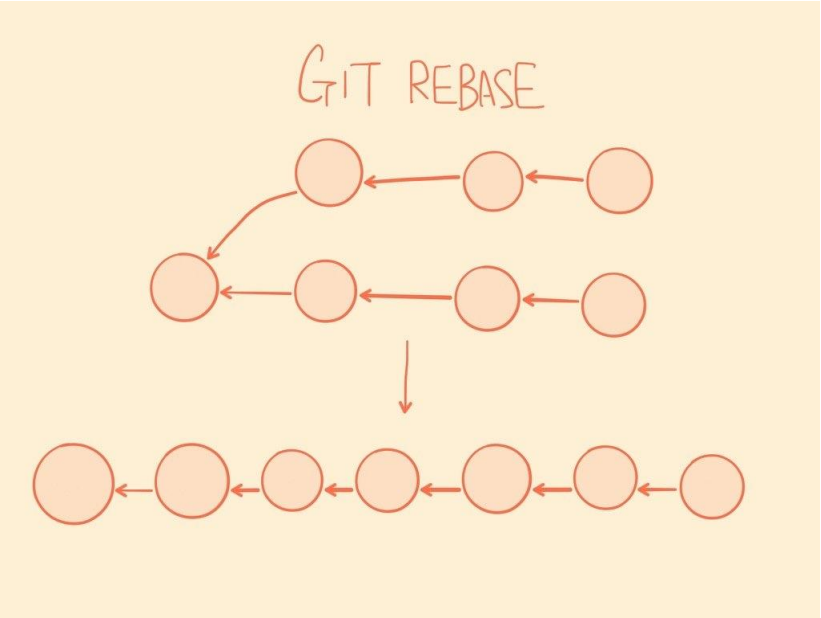
Chapter 8: Rebase and Cherry Pick

Git rebase is an advanced and powerful technique. It is like merge, but it doesn't create new commits from 2 branches. It grabs the diffs from a branch and re-apply those diffs to the other branch.

Rebase allows us to move a tree of snapshots into a new parent snapshot.



After the rebase, the log history looks like a single timeline.



Rebase Example

Not available in preview.

Git pull with rebase

Not available in preview.

Fast-forward with Rebase

Not available in preview.

Cherry Pick

Not available in preview.

Summary

In this book, we have discussed the essential Git commands to get started version controlling your source code, or any plain text. We also learned how the Git works and have a glimpse at the `.git` folder. We discussed the branch creation strategy to make better use of version control when exploring new features, or fixing existing bugs. We discussed how we can use Git to collaborate with teammates and handle merge conflicts when both modified the same file. At last, we learned how we can undo changes and rescue our commits if we unintentionally reset the history.

Version History

- 2021-04-01: Completed the book, version 1.
- 2021-03-21: Finished the missing part.
- 2021-03-09: Updated all code blocks style.
- 2021-01-08: Migrated the content into Google Docs.
- 2016-11-21: Add config of quote path.
- 2016-11-20: Minor fixes and PDF output.
- 2016-11-07: Organize content into HTML format.
- 2015-12-28: Write more about merge, rebase, cherry-pick and submodule.
- 2015-12-07: Working on the remote content.
- 2015-11-21: First Draft.