Git

Piotr Karamon

Contents

1 The perfect commit

- add the **right** changes
- compose a **good** commit message
- changes should focus on **only one** topic/point of interest. A commit should not be a random collection of changes to seemingly unrelated files.
- commits ought to be small, don't worry if their number is huge because you can always squash them

- try to minimize the amount of changed files

1.1 git add -p file.txt

The `git add -p` brings you down to a patch level. Meaning you can select only parts of changes that happened to a file. This is useful because you can separate your changes into multiple single-purpose and concise commits.

1.2 The perfect commit message

- subject - concise summary of what happened. If you are unable to write a concise summary that indicates that a commit contains too many changes. Try separating the commit into smaller fine-grained pieces.
- body - more detailed explanation
  - what is different than before?
  - what's the reason for the change?
  - is there anything to watch out for/anything really remarkable?

```
summary

body
```

Always include an empty line when your commit message has a body

2 Branching strategies

- git allows you to work with branches but it does not dictate how you should use them
- you need a written best practice of how work is ideally structured in your team - to avoid mistakes and collisions.
- branching strategy highly depends on:
  - your team
  - team size
  - project
  - handling of releases

2.1 Integrating changes & structuring releases

2.1.1 Mainline development ("Always integrate")

- few branches
- relatively small commits
- high-quality testing & QA standards

2.1.2 State, Release, and Feature Branches

- different types of branches
- branches fulfill different types of jobs:
  - feature branches
  - experiment branches
  - develop branches

2.2 Types of branches

2.2.1 Long-Running

- exist through the complete lifetime of the project
- the mirror "stages" in dev life cycle
  - development branch (new changes, not fully tested)
  - main branch (fully tested, reliable production code)
- common convention: "no direct commits!" meaning you first add your change to the development branch from which then it will merged with main branch

2.2.2 Short-Lived

- they are created to carry out a single purpose:
  - adding new features
  - refactoring
  - bug fixes
  - experiments

- will be deleted after integration with (dev/main) branches
- they are based on long-running branches

2.3 GitHub Flow

Very simple, very lean: only one long-running branch "main" + feature branches

2.4 Git flow

- more structure, more rules
- long-running: main, develop
- short-lived: features, releases, hot-fixes

3 Pull Request

They are not a core git features. They are provided by gitlab/github/bitbucket etc. Pull requests are about communicating about and reviewing code. Without a pull request once you've finished a feature you would simply merge it into main. A pull request invites reviewers to provide feedback before merging. Pull requests allow you to contribute to projects to which you do not have a direct access. Let's say Bob has hosted an open source library on github. You cannot directly alter that code. Instead you fork his repository, make your changes and create a pull request. Then Bob and other contributors can discuss, comment on your change and if they like it they can approve it.

4 Fork

A fork is your own personal copy of a git repository.

5 Merge conflicts

They happen when integrating commits from different sources.

- `git merge`
- `git rebase`

- `git pull`
- `git cherry-pick`
- `git stash apply`

Most of the time git will be able to figure out how to merge files on its own. However there are still a lot of situations in which git can't do that. For example if one commit changed a certain line to one string, and another commit changed that same line to yet again a different string. In this case git cannot know which one is correct, it needs you do make that decision. In case of a conflict you can undo a merge or a rebase. Simply by doing

```
git merge/rebase --abort
```

5.1 How conflicts actually look like?

The problem are is specially surrounded

```
<<<<<<< HEAD
...problematic area...
=======
>>>>>>> develop
```

Solving the conflict means deleting those files and make changes that you actually want. You can solve those conflicts using a simple text editor. However, some tools or sophisticated text editors/ides have special tools which make your job a lot easier.

6 Merge vs Rebase

Merge and rebase are ways of integrating branches.

6.1 Merge

In order to perform a merge you need to have two branches. Typically they are:

- feature branch
- main/develop branch

Then git looks for three commits:

- common ancestor (split point, when typically feature branch was created)
- last commit on branch-A
- last commit on branch-B

6.1.1 Fast forward merge

If the last commit on one branch is the same as common ancestor git can perform a special kind of merge called fast-forward merge. It will just take commits from one branch and apply them to the other. It is a seamless, effortless merge, however they are not that common. They can happen fairly often if the only developer on a project is you.

6.1.2 More typical kind of merge

Usually in order to perform a merge we need something called a merge commit. It's different from a lot of commits because it is created automatically, but we can still alter its message and so. The merge commit purpose is to connect two branches.

6.2 Rebase

Rebase is a different(not better or worse) way of integrating two branches. Some people want their project history to look like a straight line, without any signs it was split, and then combined. Rebase will:

1. Remove temporarily commits from branch-a.
2. Copy commits from branch-b to branch-a.
3. Reapply those removed commits back.

```
c1 <- c3 (branch-a)
 |<- c2 <- c4 (branch-b)


c1 <- c2 <- c4 (branch-b, branch-a)

c1 <- c2 <- c4 <- c3* (branch-a)
        (branch-b)
```

This preserves the original commit structure. Rebase rewrites the commit history. `c3*` is not the same as `c3`. The contents of those commits is identical,

but they are different. `c3*` has a different parent. `c3*` and `c3` obviously have different hashes. A good rule is:

> DO NOT use rebase on commits that you've already pushed/shared on a remote repository
>
> Instead, use it for cleaning up your local commit history before merging it into a shared team branch

7 Interactive rebase

It is a tool for optimizing and cleaning up your commit history. You can

- change a commit message
- delete commits
- reorder commits
- combine multiple commits into one
- edit / split an existing commit into multiple ones

Interactive rebase rewrites your commit history. The same rule as with normal rebase still applies.

> DO NOT use interactive rebase on commits that you've already pushed/shared on a remote repository

That means you should you interactive rebase for cleaning up / fixing your feature branch's commit history before merging it into a shared team branch.

7.1 Step by step

1. How far back you want to go? What should be the "base" commit?
2. `git rebase -i HEAD~3` The number following the ~ indicates how far back you want to go?
3. In the editor, only determine which actions you want to perform. Don't change commit data in this step, yet!

7.2 What can you do with `rebase -i`?

It is important to know that the order of commits in `rebase -i` is different from that used by `git log`. In `rebase -i` the oldest commits are at the top. Lines inside of the rebase window have the following format

```
command <commit> <commit-message>
```

- `pick` does nothing
- `reword` lets you change the commit message
- `drop` lets you delete a commit
- `squash` combines the current line with the line above. If you put `squash` on line 3, it will squash it with line 2.

8 Cherry picking

Cherry picking is used for integrating single, specific commits. It can be useful when you for example, created a commit on a wrong branch.

```
git checkout feature-branch
git cherry-pick 26b1fb48
git checkout master
git reset --hard HEAD~1
```

This snippet of code picks the specified commit and pushes it to the feature-branch. Then we delete that commit from the master branch.

9 Searching for certain commits

- `--before` commits before a certain date
- `--grep` commits with messages matching the passed regex
- `--author` commits written by specified author
- `-- <filename>` commits that include changes to specified file you can pass multiple filenames
- `--after` commits after a certain date
- `<branch-name>` commits on a certain branch

The `git commit` command accepts arguments that you can use to search only for certain commits. You can combine those options.

10 Submodules

Say you need some third-party code. One option is to just copy and paste their library or parts of it into your project. However that means you have to mix someone else code with yours. Also updating the external code is a **manual process**.

A submodule is a git repository inside of another git repository.

To add a submodule you do:

```
git submodule add <repo-url>
```

When you clone a repository which contains submodules after a clone you have to run

```
git submodule update --init --recursive
```

You can however do the same thing using just one command.

```
git clone --recursive-submodules <repo-url>
```

11 Undoing mistakes in git

11.1 Discarding all local changes in a file

```
git restore <filename>
```

The `filename` will discard any changes that you've made to that file.

> You cannot undo this operation. Your changes will be gone forever.

11.2 Restoring a deleted file

You may accidentally delete a file. In this case you can very easily bring it back.

```
git restore <filename>
```

11.3 Discarding chunks/lines in a file

Sometimes you want to keep only certain changes that you've made to a file.

```
git restore -p <filename>
```

`-p` flags brings you down to a patch level, meaning you can interactively select which changes ought to be kept.

11.4 Discarding all local changes

```
git restore .
```

> You cannot undo this operation. Your changes will be gone forever.

11.5 Fixing the last commit

Fixing the **last** commit in git is very easy.

```
git commit --ammend -m "<new-commit-message>"
```

This will take your staged changes and apply them to the last commit. Also it will change the commit message to the one you've passed.

> `--amend` rewrites history! **Never** change history for commits that have already been **pushed to a remote repository.**

11.6 Reverting a commit in the middle

`git revert` creates a new commit that reverts the effects of a specified commit. It will apply reversed changes to the ones in the specified commit. It works in a very non destructive way. All you need to revert a certain commit is to run `git revert <commit-hash>`. You then will be prompted for a commit message, which will be filled with a default.

11.7 Resetting to an old revision

`git reset` sets your `HEAD` pointer to an older revision. It has the effect of deleting a certain number of commits starting from the head.

```
git reset --hard <commit-hash>
```

This command "takes you back in time" when the passed commit was the `HEAD`.

- `--hard` means no local changes should survive
- `--mixed` this keeps the changes we are going to undo as local changes

11.8 Resetting a file to an old revision

```
git restore --source <old-revision-commit-hash> <filename>
```

11.9 Reflog

It is a journal. You can for example recover deleted commits with it.

```
git reflog
```

shows you the journal, each line starts with a hash. You can just do to for example recover deleted commits. Often you want to do those recoveries inside of another branch to do that:

```
git branch <name-of-new-branch> <hash-from-reflog>
```

Reflog can also be used to recover deleted branches. Just do

```
git branch <name-of-new-branch> <hash-from-reflog>
```

12 Stash

Stashing is handy when you've been working on a change but for some reason you need to change context. This for example happens when you need to quickly fix a bug. `git stash` allows you to save your changes and reapply them latter to do a full commit.

12.1 `git stash`

This command saves your changes(both staged an unstaged) and reverts you to the last commit on the current branch. The `-p` options brings you down to the patch level. That means you can interactively select which changes you wish to stash. There are lots of options to choose `(y,n,q,a,d,e,?)`. To view information about every one of those options type `?`. **By default git will not stash changes made to untracked or ignored files**

12.2 Stashing untracked and ignored files

In order to stash untracked files you need to pass the `-u` or `--include-untracked` option. If you also want to stash changes made to ignored files you need to pass the `-a` or `--all` flag.

12.3 `git stash save`

This command allows you to stash your changed and provide a message describing the stash. For example `git stash save "add footer"`

12.4 `git stash list`

This command allows you to list all your stashes.

12.5 `git stash drop`

You can a stash name to this command and it deletes it.

12.6 `git stash clear`

This command deletes all of your stashes.

12.7 `git stash pop`

This command allows you to reapply the changes you've stashed. However, all the changes introduced by reapplying the stash will be unstaged, even if you've stashed some staged changes. By default this command will reapply changes in the most recently created stash. To pop a different stash use `git stash pop stash@{<index>}`.

12.8 `git stash apply`

Does the same thing as `git stash pop` but keeps your stash, so you can for example apply it on many different branches.

12.9 `git stash show`

This command shows you a description of changes in the stash. To view it in more detail pass `-p` option.

12.10 `git stash branch <branch-name> <stash>`

This checks out a new branch based on the commit that you created your stash from, and then pops your stashed changes onto it.