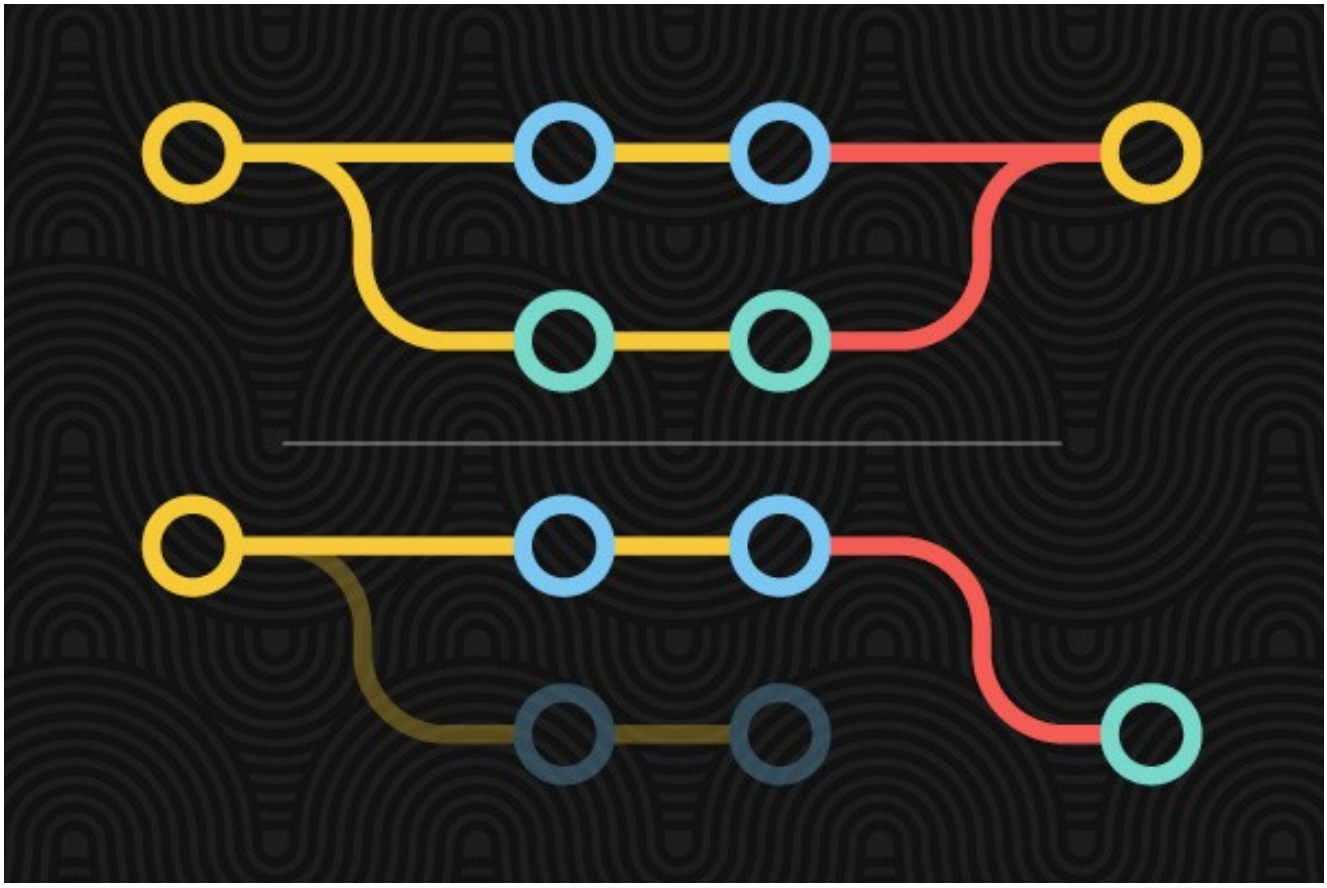
14 NOVEMBER 2018 / #GIT

An introduction to Git merge and rebase: what they are, and how to use them



by Vali Shah

As a Developer, many of us have to choose between Merge and Rebase. With all the references we get from the internet, everyone

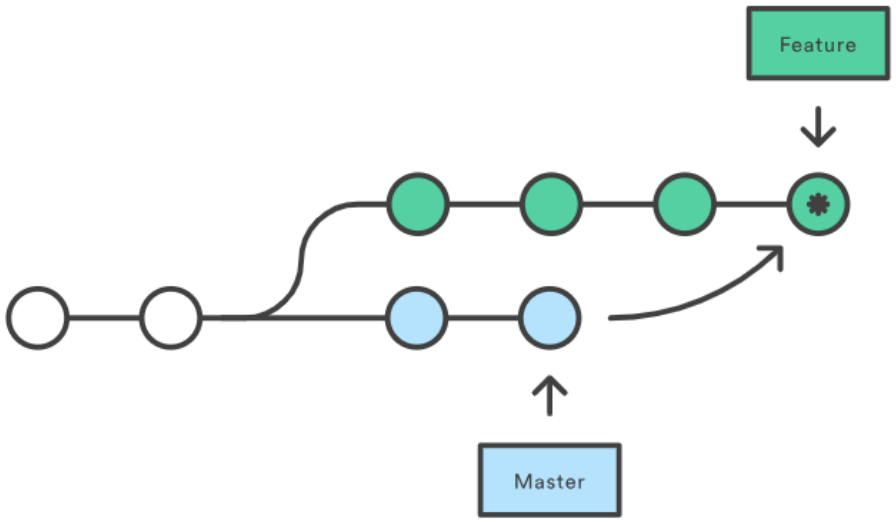
will explain what merge and rebase are, why you should (and shouldn't) use them, and how to do so.

Git Merge and Git Rebase serve the same purpose. They are designed to integrate changes from multiple branches into one. Although the final goal is the same, those two methods achieve it in different ways, and it's helpful to know the difference as you become a better software developer.

This question has split the Git community. Some believe you should always rebase and others that you should always merge. Each side has some convincing benefits.

Git Merge

Merging is a common practice for developers using version control systems. Whether branches are created for testing, bug fixes, or other reasons, merging commits changes to another location. To be more specific, merging takes the contents of a source branch and integrates them with a target branch. In this process, only the target branch is changed. The source branch history remains the same.



Pros

- Simple and familiar
- Preserves complete history and chronological order
- Maintains the context of the branch

Cons

- Commit history can become polluted by lots of merge commits
- Debugging using `git bisect` can become harder

How to do it

Merge the master branch into the feature branch using the `checkout` and `merge` commands.

```
$ git checkout feature
$ git merge master

(or)

$ git merge master feature
```

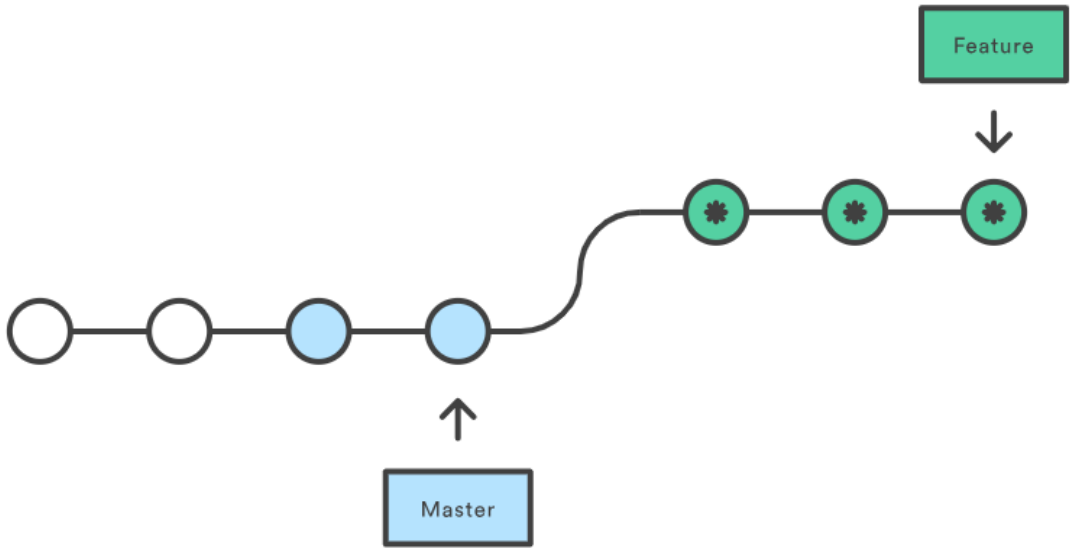
This will create a new "**Merge commit**" in the feature branch that holds the history of both branches.

Git Rebase

Rebase is another way to integrate changes from one branch to another. Rebase compresses all the changes into a single "patch."

Unlike merging, rebasing flattens the history because it transfers the completed work from one branch to another. In the process, unwanted history is eliminated.

> *Rebases are how changes should pass from the top of the hierarchy downwards, and merges are how they flow back upwards*



Rebase feature branch into master

Pros

- Streamlines a potentially complex history
- Manipulating a single commit is easy (e.g. reverting them)
- Avoids merge commit “noise” in busy repos with busy branches
- Cleans intermediate commits by making them a single commit, which can be helpful for DevOps teams

- Squashing the feature down to a handful of commits can hide the context
- Rebasing public repositories can be dangerous when working as a team
- It's more work: Using rebase to keep your feature branch updated always
- Rebasing with remote branches requires you to *force push*. The biggest problem people face is they force push but haven't set git push default. This results in updates to all branches having the same name, both locally and remotely, and that is **dreadful** to deal with.

> *If you rebase incorrectly and unintentionally rewrite the history, it can lead to serious issues, so make sure you know what you are doing!*

How to do it

Rebase the feature branch onto the master branch using the following commands.

```
$ git checkout feature
$ git rebase master
```

This moves the entire feature branch on top of the master branch. It does this by re-writing the project history by creating brand new commits for each commit in the original (feature) branch.

Interactive Rebasing

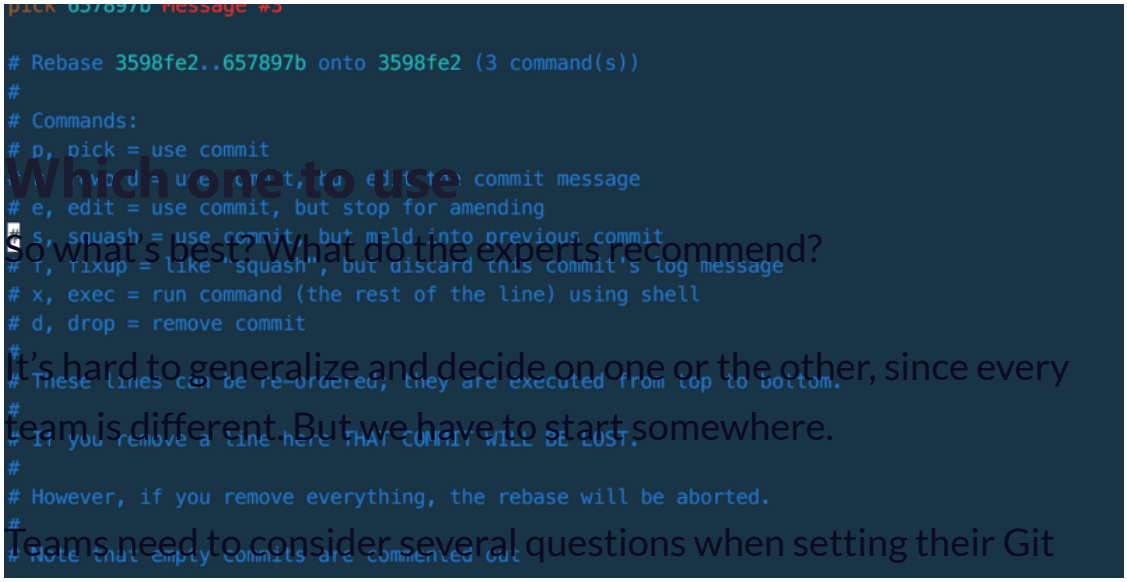
branch. This is more powerful than automated rebase, as it offers complete control over the branch's commit history. Typically this is used to clean up a messy history before merging a feature branch into master.

```
$ git checkout feature
$ git rebase -i master
```

This will open the editor by listing all the commits that are about to be moved.

```
pick 22d6d7c Commit message#1
pick 44e8a9b Commit message#2
pick 79f1d2h Commit message#3
```

This defines exactly what the branch will look like after the rebase is performed. By re-ordering the entities, you can make the history look like whatever you want. For example, you can use commands like `fixup`, `squash`, `edit` etc, in place of `pick`.



rebase vs. merge policies. Because as it turns out, one workflow strategy is not better than the other. It is dependent on your team.

Consider the level of rebasing and Git competence across your organization. Determine the degree to which you value the simplicity of rebasing as compared to the traceability and history of merging.

Finally, decisions on merging and rebasing should be considered in the context of a clear branching strategy (**Refer this article** to understand more about branching strategy). A successful branching strategy is designed around the organization of your teams.

What do I recommend?

As the team grows, it will become hard to manage or trace development changes with an **always merge policy.** To have a clean and understandable commit history, using **Rebase** is reasonable and effective.

By considering the following circumstances and guidelines, you can get best out of **Rebase:**

- **You're developing locally:** If you have not shared your work

over merging to keep your history tidy. If you've got your personal fork of the repository and that is not shared with other developers, you're safe to rebase even after you've pushed to your branch.

- **Your code is ready for review:** You created a pull request. Others are reviewing your work and are potentially fetching it into their fork for local review. At this point, you should not rebase your work. You should create 'rework' commits and update your feature branch. This helps with traceability in the pull request and prevents the accidental history breakage.
- **The review is done and ready to be integrated into the target branch.** Congratulations! You're about to delete your feature branch. Given that other developers won't be fetch-merging in these changes from this point on, this is your chance to sanitize your history. At this point, you can rewrite history and fold the original commits and those pesky 'pr rework' and 'merge' commits into a small set of focused commits. Creating an explicit merge for these commits is optional, but has value. It records when the feature graduated to master.

Conclusion

I hope this explanation has given some insights on **Git merge** and **Git rebase.** Merge vs rebase strategy is always debatable. But perhaps this article will help dispel your doubts and allow you to adopt an approach that works for your team.

I'm looking forward to writing on **Git workflows** and concepts of **Git**. Do comment on the topics that you want me to write about next. Cheers!

Donate

About

Alumni Network

Open Source

Shop

Support

Sponsors

Academic Honesty

Code of Conduct

Privacy Policy

Terms of Service

Copyright Policy

2019 Web Developer Roadmap

Python Tutorial

CSS Flexbox Guide

JavaScript Tutorial

Python Example

HTML Tutorial

Linux Command Line Guide

JavaScript Example

Git Tutorial

React Tutorial

Java Tutorial

Linux Tutorial

CSS Tutorial

jQuery Example

SQL Tutorial

CSS Example

React Example

Angular Tutorial

Bootstrap Example

How to Set Up SSH Keys

WordPress Tutorial

PHP Example