🌳🚀 CS Visualized: Useful Git Commands

Lydia Hallie · Apr 1 · 9 min read

#git #computerscience #tutorial

Although Git is a very powerful tool, I think most people would agree when I say it can also be... a total nightmare 😐 I've always found it very useful to visualize in my head what's happening when working with Git: how are the branches interacting when I perform a certain command, and how will it affect the history? Why did my coworker cry when I did a hard reset on `master`, `force push`ed to origin and `rimraf`'d the `.git` folder?

I thought it would be the perfect use case to create some visualized examples of the most common and useful commands! 🥳 Many of the commands I'm covering have optional arguments that you can use in order to change their behavior. In my examples, I'll cover the default behavior of the commands without adding (too many) config options! 😄

Merging

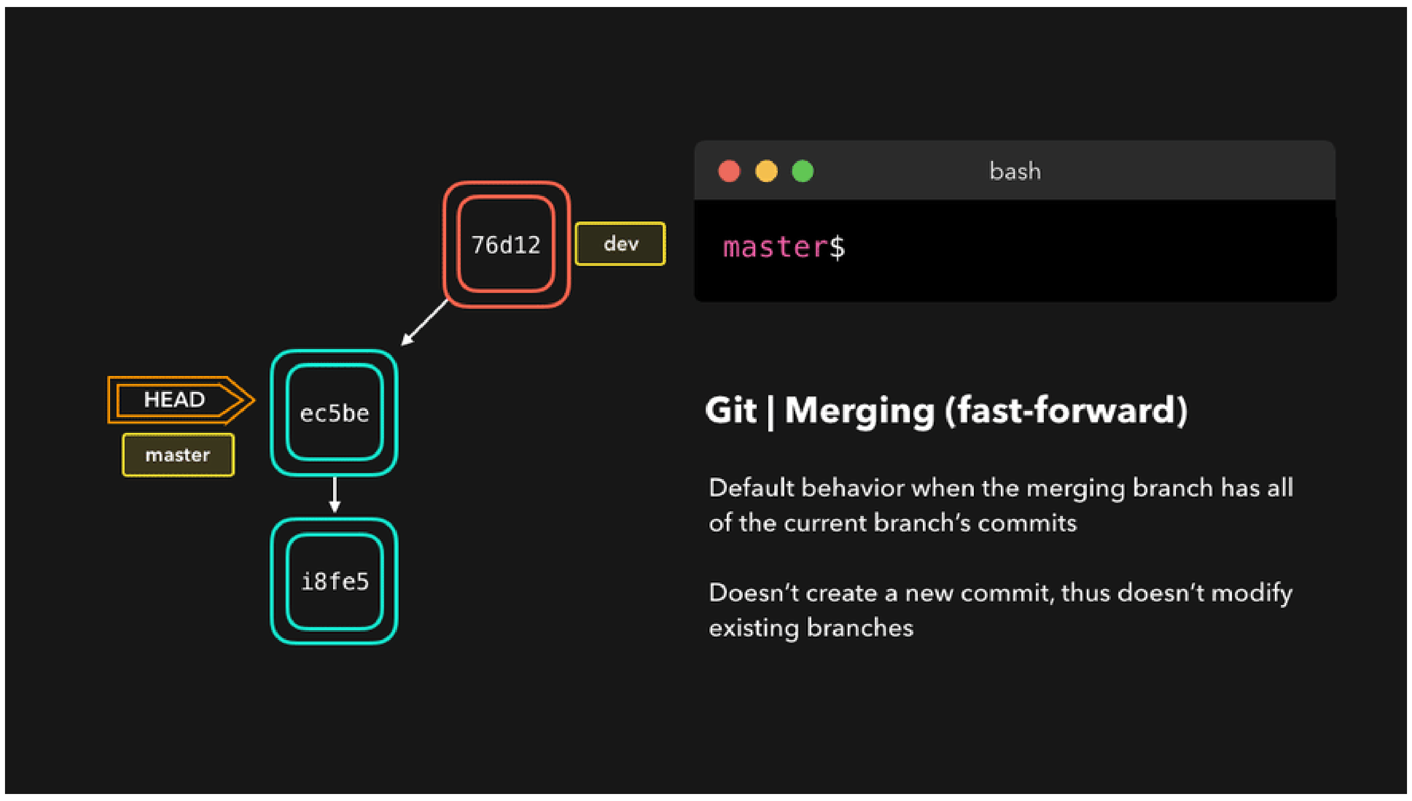
Having multiple branches is extremely convenient to keep new changes separated from each other, and to make sure you don't accidentally push unapproved or broken changes to production. Once the changes have been approved, we want to get these changes in our production branch!

One way to get the changes from one branch to another is by performing a `git merge`! There are two types of merges Git can perform: a **fast-forward**, or a **no-fast-forward** 🐢

This may not make a lot of sense right now, so let's look at the differences!

Fast-forward (`--ff`)

A **fast-forward merge** can happen when the current branch has no extra commits compared to the branch we're merging. Git is... *lazy* and will first try to perform the easiest option: the fast-forward! This type of merge doesn't create a new commit, but rather merges the commit(s) on the branch we're merging right in the current branch 🥳

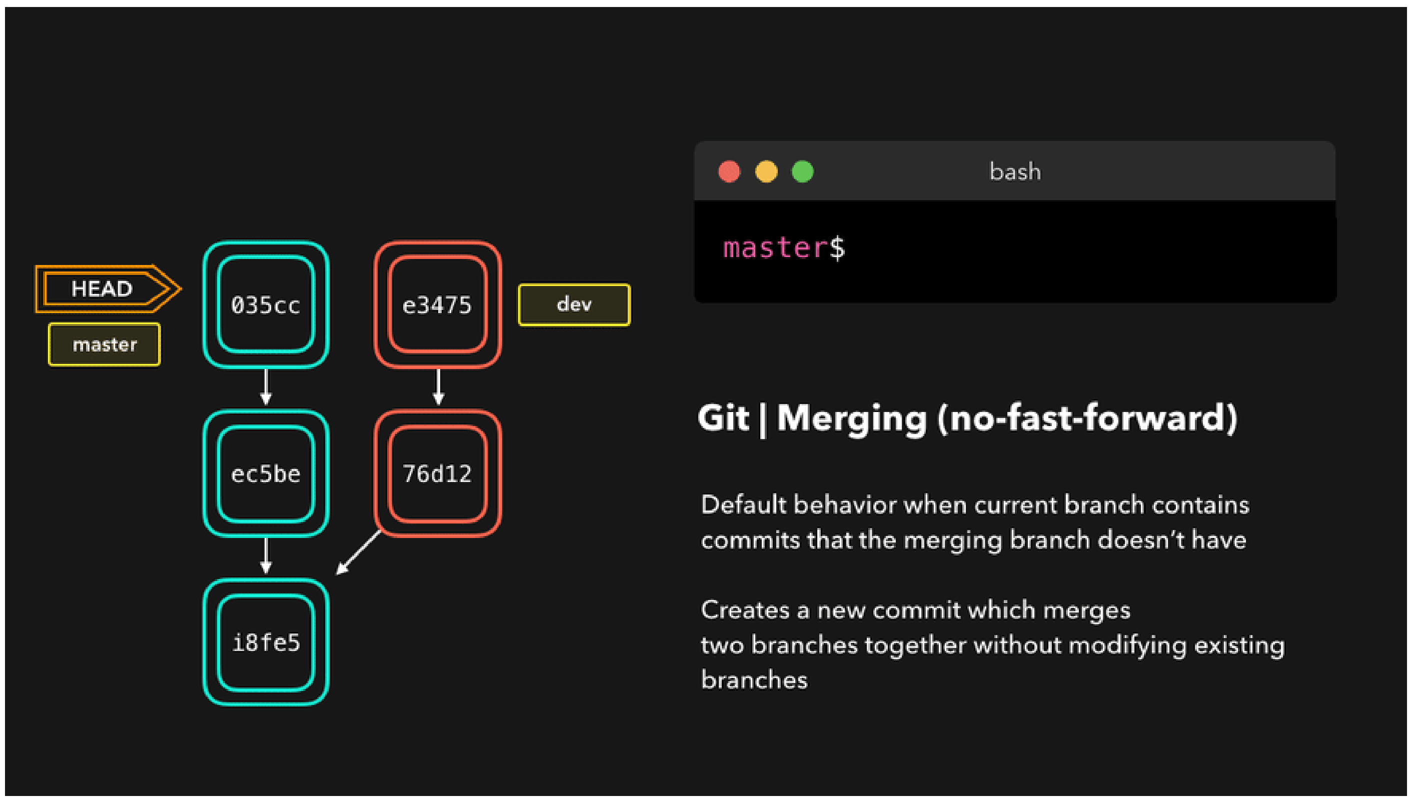

76d12
dev
HEAD
master
ec5be
i8fe5
bash
master$
Git | Merging (fast-forward)
Default behavior when the merging branch has all of the current branch's commits
Doesn't create a new commit, thus doesn't modify existing branches

Perfect! We now have all the changes that were made on the `dev` branch available on the `master` branch. So, what's the **no-fast-forward** all about?

No-fast-foward (`--no-ff`)

It's great if your current branch doesn't have any extra commits compared to the branch that you want to merge, but unfortunately that's rarely the case! If we committed changes on the current branch that the branch we want to merge doesn't have, git will perform a *no-fast-forward* merge.

With a no-fast-forward merge, Git creates a new *merging commit* on the active branch. The commit's parent commits point to both the active branch and the branch that we want to merge!



```
master$
```

Git | Merging (no-fast-forward)

Default behavior when current branch contains commits that the merging branch doesn't have

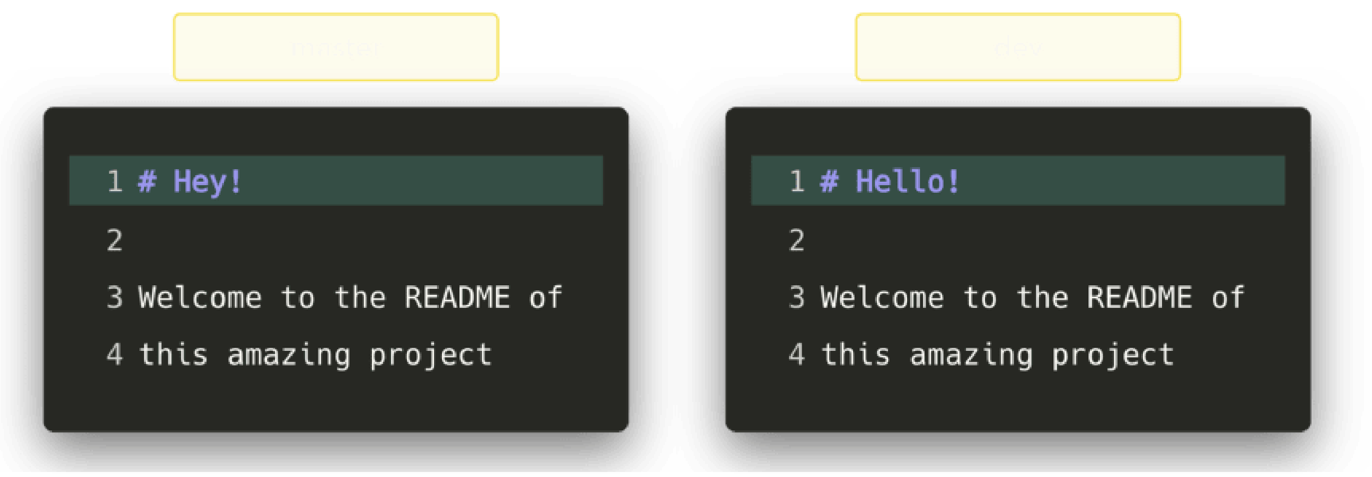
Creates a new commit which merges two branches together without modifying existing branches

No big deal, a perfect merge! 🎉 The `master` branch now contains all the changes that we've made on the `dev` branch.

Merge Conflicts

Although Git is good at deciding how to merge branches and add changes to files, it cannot always make this decision all by itself 🙂 This can happen when the two branches we're trying to merge have changes on the same line in the same file, or if one branch deleted a file that another branch modified, and so on.

In that case, Git will ask you to help decide which of the two options we want to keep! Let's say that on both branches, we edited the first line in the `README.md`.

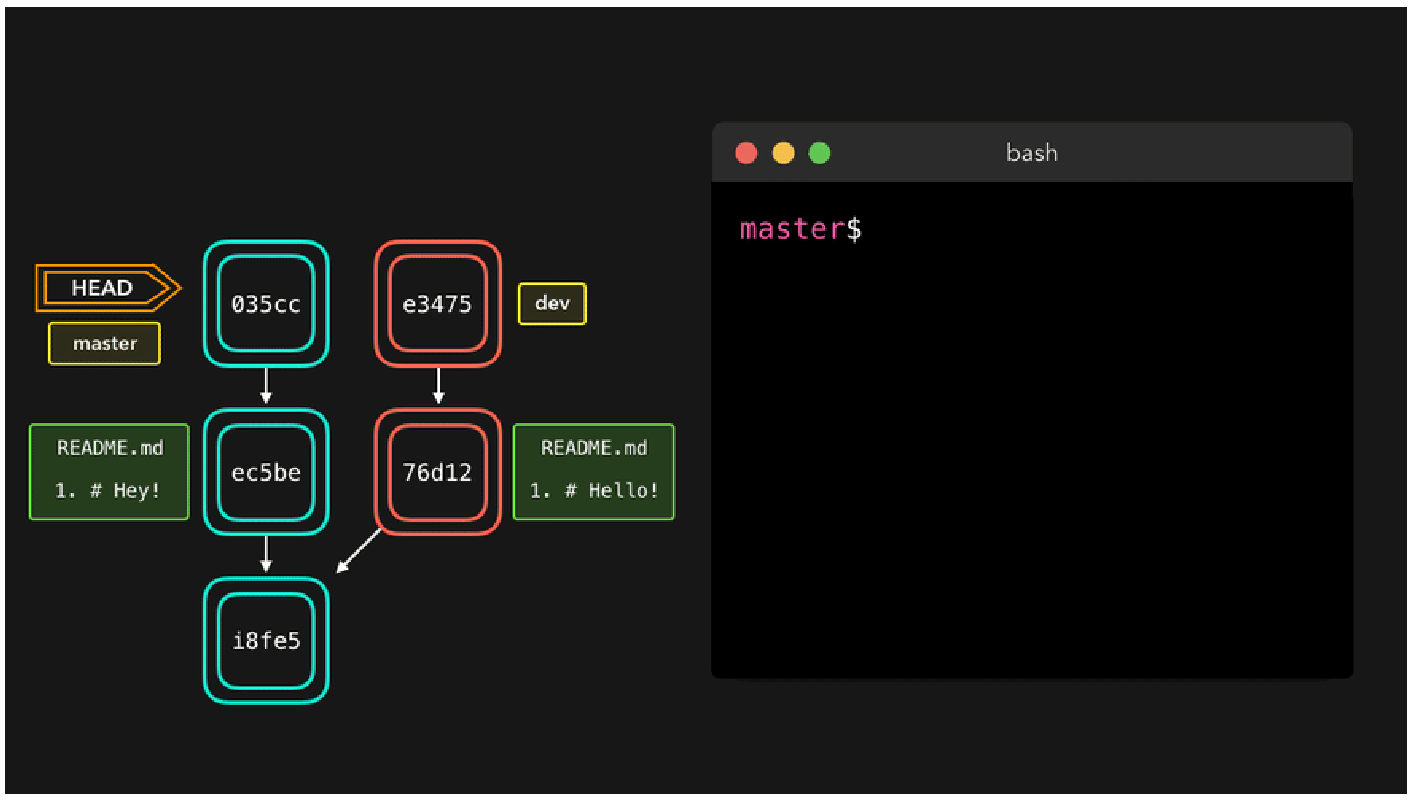


```
# Hey!

Welcome to the README of
this amazing project
```

```
# Hello!

Welcome to the README of
this amazing project
```

If we want to merge `dev` into `master`, this will end up in a merge conflict: would you like the title to be `Hello!` or `Hey!`?

When trying to merge the branches, Git will show you where the conflict happens. We can manually remove the changes we don't want to keep, save the changes, add the changed file again, and commit the changes

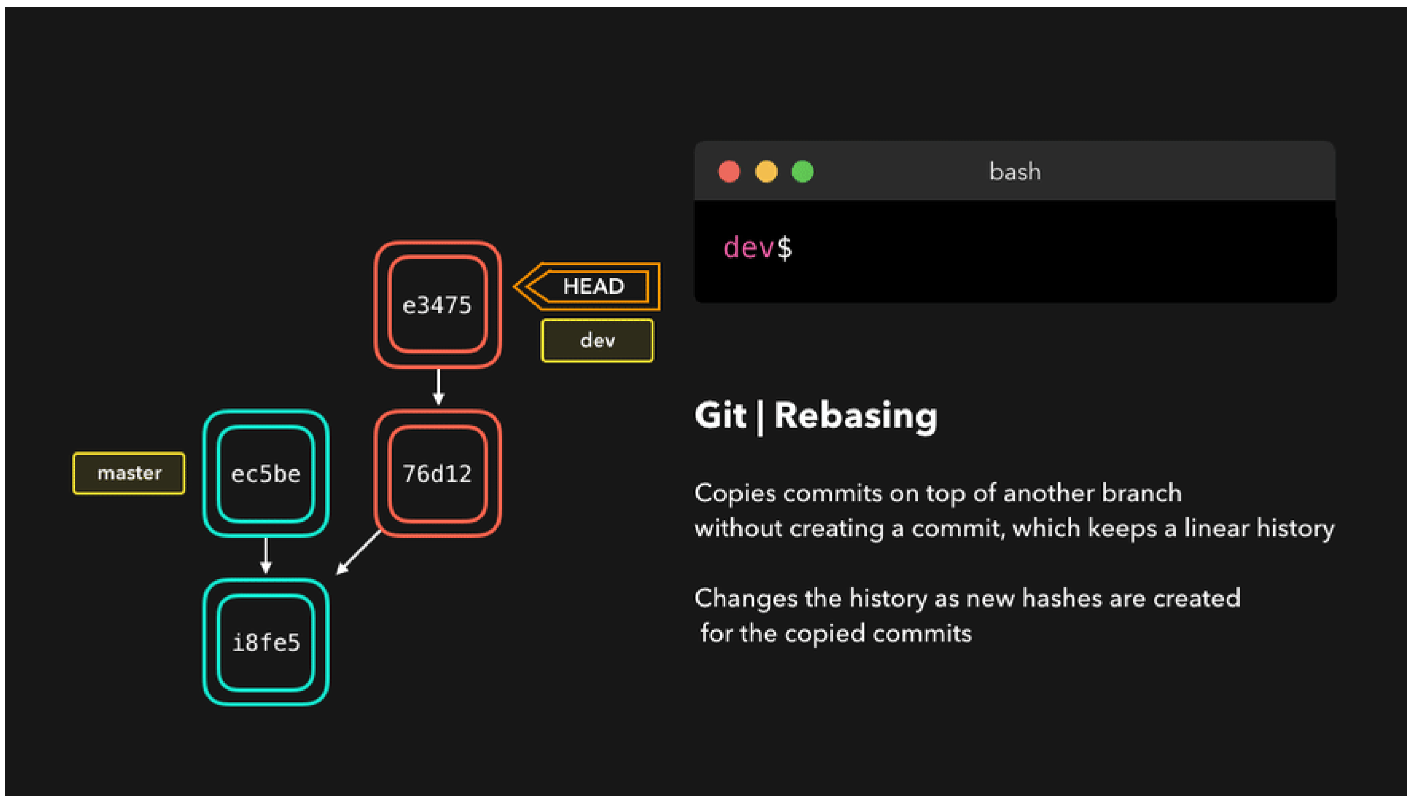


Yay! Although merge conflicts are often quite annoying, it makes total sense: Git shouldn't just *assume* which change we want to keep.

Rebasing

We just saw how we could apply changes from one branch to another by performing a `git merge` . Another way of adding changes from one branch to another is by performing a `git rebase` .

A `git rebase` *copies* the commits from the current branch, and puts these copied commits on top of the specified branch.



Git | Rebasing

Copies commits on top of another branch without creating a commit, which keeps a linear history

Changes the history as new hashes are created for the copied commits

Perfect, we now have all the changes that were made on the `master` branch available on the `dev` branch!

A big difference compared to merging, is that Git won't try to find out which files to keep and not keep. The branch that we're rebasing always has the latest changes that we want to keep! You won't run into any merging conflicts this way, and keeps a nice linear Git history.

This example shows rebasing on the `master` branch. In bigger projects, however, you usually don't want to do that. A `git rebase` **changes the history of the project** as new hashes are created for the copied commits!

Rebasing is great whenever you're working on a feature branch, and the master branch has been updated. You can get all the updates on your branch, which would prevent future merging conflicts! 😄

Interactive Rebase

Before rebasing the commits, we can modify them! 😃 We can do so with an *interactive rebase*. An interactive rebase can also be useful on the branch you're currently working on, and want to modify some commits.

There are 6 actions we can perform on the commits we're rebasing:

- `reword` : Change the commit message
- `edit` : Amend this commit
- `squash` : Meld commit into the previous commit
- `fixup` : Meld commit into the previous commit, without keeping the commit's log message
- `exec` : Run a command on each commit we want to rebase
- `drop` : Remove the commit

Awesome! This way, we can have full control over our commits. If we want to remove a commit, we can just `drop` it.

Or if we want to squash multiple commits together to get a cleaner history, no problem!

Interactive rebasing gives you a lot of control over the commits you're trying to rebase, even on the current active branch!

Resetting

It can happen that we committed changes that we didn't want later on. Maybe it's a `WIP` commit, or maybe a commit that introduced bugs! 🐛 In that case, we can perform a `git reset`.

A `git reset` gets rid of all the current staged files and gives us control over where `HEAD` should point to.

Soft reset

A *soft reset* moves `HEAD` to the specified commit (or the index of the commit compared to `HEAD`), without getting rid of the changes that were introduced on the commits afterward!

Let's say that we don't want to keep the commit `9e78i` which added a `style.css` file, and we also don't want to keep the commit `035cc` which added an `index.js` file. However, we do want to keep the newly added `style.css` and `index.js` file! A perfect use case for a soft reset.

When typing `git status`, you'll see that we still have access to all the changes that were made on the previous commits. This is great, as this means that we can fix the contents of these files and commit them again later on!

Hard reset

Sometimes, we don't want to keep the changes that were introduced by certain commits. Unlike a soft reset, we shouldn't need to have access to them any more. Git should simply reset its state back to where it was on the specified commit: this even includes the changes in your working directory and staged files! 💣

Git has discarded the changes that were introduced on `9e78i` and `035cc`, and reset its state to where it was on commit `ec5be`.

Reverting

Another way of undoing changes is by performing a `git revert`. By reverting a certain commit, we create a *new commit* that contains the reverted changes!

Let's say that `ec5be` added an `index.js` file. Later on, we actually realize we didn't want this change introduced by this commit anymore! Let's revert the `ec5be` commit.

Perfect! Commit `9e78i` reverted the changes that were introduced by the `ec5be` commit. Performing a `git revert` is very useful in order to undo a certain commit, without modifying the history of the branch.

Cherry-picking

When a certain branch contains a commit that introduced changes we need on our active branch, we can `cherry-pick` that command! By `cherry-pick`ing a commit, we create a new commit on our active branch that contains the changes that were introduced by the `cherry-pick`ed commit.

Say that commit `76d12` on the `dev` branch added a change to the `index.js` file that we want in our `master` branch. We don't want the *entire* we just care about this one single commit!

Cool, the master branch now contains the changes that `76d12` introduced!

Fetching

If we have a remote Git branch, for example a branch on Github, it can happen that the remote branch has commits that the current branch doesn't have! Maybe another branch got merged, your colleague pushed a quick fix, and so on.

We can get these changes locally, by performing a `git fetch` on the remote branch! It doesn't affect your local branch in any way: a `fetch` simply downloads new data.

We can now see all the changes that have been made since we last pushed! We can decide what we want to do with the new data now that we have it locally.

Pulling

Although a `git fetch` is very useful in order to get the remote information of a branch, we can also perform a `git pull`. A `git pull` is actually two commands in one: a `git fetch`, and a `git merge`. When we're pulling changes from the origin, we're first fetching all the data like we did with a `git fetch`, after which the latest changes are automatically merged into the local branch.

Awesome, we're now perfectly in sync with the remote branch and have all the latest changes! 🤩

Reflog

Everyone makes mistakes, and that's totally okay! Sometimes it may feel like you've screwed up your git repo so badly that you just want to delete it entirely.

`git reflog` is a very useful command in order to show a log of all the actions that have been taken! This includes merges, resets, reverts: basically any alteration to your branch.

)

If you made a mistake, you can easily redo this by resetting `HEAD` based on the information that `reflog` gives us!

Say that we actually didn't want to merge the origin branch. When we execute the `git reflog` command, we see that the state of the repo before the merge is at `HEAD@{1}`. Let's perform a `git reset` to point HEAD back to where it was on `HEAD@{1}`!

We can see that the latest action has been pushed to the `reflog`!

Git has so many useful porcelain and plumbing commands, I wish I could cover them all! 😄 I know there are many other commands or alterations that I didn't have time for to cover right now - let me know what your favorite/most useful commands are, and I may cover them in another post!

And as always, feel free to reach out to me! 😊

Twitter	Instagram	GitHub	LinkedIn	YouTube	Email

Lydia Hallie +FOLLOW
@theavocoder | i like da codez
@lydiahallie lydiahallie lydiahallie www.lydiahallie.dev

Add to the discussion

PREVIEW SUBMIT

Cody Pearce Apr 1

Awesome visualizations as usual!

It's interesting there's a few different syntaxes for selecting a previous commit:

```
HEAD~2          // previous two commits fro head
HEAD~~         // previous two commits from head
HEAD@{2}     // reflog order
18fe5              // previous commit hash
```

8 REPLY

Lydia Hallie Apr 1

Good idea to add that! I'm thinking of generally creating a "cheatsheet" format that also covers all that stuff :) Will do in the next one or when I update the format 😄

12 REPLY

toby Apr 15

Came here to comment specifically along these lines - I can never ever remember what the difference between `HEAD~2` and `HEAD^2` is !

Had completely forgotten about `HEAD@{n}` syntax :D

2 REPLY

usertest619 Apr 16

test

♡ 1 REPLY

▼

Uday Vunnam Apr 2

Nice visualizations! What do you use to create them?

♡ 7 REPLY

▼

Patrik Apr 2

I'd love to know the answer to that too! :)

♡ 3 REPLY

▼

D. Guhl Apr 3

Lydia answered in another post of hers, which I recommend myself, that she used Keynote (the presentation software by Apple) to make the animations and then screen-recorded the slides.

The Post is "JavaScript Visualized: the JavaScript Engine"

♡ 4 THREAD

ThiagoBL Apr 8

Lol! Her profile description has the asnwer. :P

♡ 2 REPLY

▼

Mansoor Apr 6

Awesome animations. It is great to see a diagrammatic representation of these git commands.

For "git pull", is the animation correct? I would have expected it to fetch the commits and then do a ff merge.

For clarity, it would also help to distinguish between the remote repository and the local remote branches in the animations.

♡ 4 REPLY

▼

Fabian Apr 15

I also expected the ff merge. But now she already has an animation for `git pull --no-ff` if ever she needs one. 😉

♡ 3 REPLY

▼

Dani Amsalem Apr 11

Amazing article Lydia!

I prefer to use Git in terminal as opposed to a GUI like the others on my team so I can face my Git fears. However, most of the documentation I read online is very complicated. Yours is the first long-form article I got to the bottom of and didn't have 2X the confusions as when I started!

Please write more Git visualized articles. I'll devour them, I swear.

3 REPLY

Vaibhav Khulbe Apr 1

Wow! Never knew about something like Reflog! Thanks for your efforts 🤗

5 REPLY

Borek Bernard Apr 9

Beautifully done! Seeing `rebase --onto` visualized would be great too 😄.

3 REPLY

Varun Apr 2

Amazing work Lydia....Extremely helpful for all dev peeps.

4 REPLY

Sajal Apr 2

TheAvocoder back with another masterpiece. 👊

4 REPLY

Lydia Hallie Apr 2

thank uu 🐊😎

4 REPLY

Gino Mempin Apr 9

Great visualizations :)

It might be worth noting that `git reset` does not delete untracked files, even with `--hard`. It only affects those that are already tracked.

So, for example, some recent commits causes your app to generate some files, then you later `reset` those commits, it will leave those files as untracked.

For that, you need `git clean`.

1 REPLY

Chirag Shah Apr 7

I have one question.
For example, I am working on the dev branch. Meanwhile, my colleague has pushed 2 more commits.

What should I do? Should I pull first then commit or should I commit first and then pull?
I am always confused over here.

1 REPLY

Thamaraiselvam Apr 15

You cannot pull before commit because git does not know what do with unchanged things in local.

This is what we do.

- commit local changes
- git pull --rebase (This will copy commits to top. without rebase commits be will merged)
- git push

if you dont want to commit ur changes and still you want to pull data you do stash

stash will push changes to stack and you can get it from it later or you can auto stash

git pull --rebase --auto-stash

2 REPLY

Chirag Shah Apr 16

Thank you so much for the answer. Very Helpful!

2 REPLY

Peter Uithoven Apr 15

Very clarifying, the visualizations I wish where there when I first learned Git.
2 tiny text issues:

1. "we can cherry-pick that command!" should probably be "we can cherry-pick that commit!".
2. "We don't want the entire we just" should probably be "We don't want the entire branch we just"

Keep up the good work!

1 REPLY

aRtoo Apr 3

You really are the lost supergirl. Thank you super champ. :(

3 REPLY

Bedrich Horak Apr 2

Really awesome!! Thank you.
Do you think you could make an animation of `git pull --rebase` 🙏? I would like to show it in comparison to `git pull`.

2 REPLY

Győri Sándor Apr 2

pull is fetch + merge
pull --rebase is fetch + rebase

3 REPLY

▼

Thien Nguyen Apr 2

Very useful!

3 REPLY

▼

Marco Pestrin Apr 2

wooow! amazing :) very good job

3 REPLY

▼

Nico Braun Apr 1

Amazing post, like every inch of it.

3 REPLY

▼

MrCricket Apr 16

..reset moves HEAD to the specified commit..

I think it should be ***reset moves the branch that HEAD is pointing to***

2 REPLY

▼

Chidiebere Ogujeiofor Apr 2

This is very cool. Really love the visualisation.

What tool did you use to make those visualisation?

Great job!

2 REPLY

▼

Yang Yang Apr 14

you are superstar!

2 REPLY

▼

Rosered Apr 3

I like it.

2 REPLY

▼

Akash Preet

Apr 1

This is interesting, Thanks for this post :-)
Added in my reading list

2

REPLY

kapeel kokane

Apr 6

In the animation for **Pulling**, shouldn't the 2 commits (7e456 and efi81) get copied over to the left side?

2

REPLY

Mohammed El-Afifi

Apr 7

Nice read. One note though about rebasing is that we can still run into conflicts just as we do with merges.

2

REPLY

petr7555

Apr 8

I can read about these basic commands many times a year and each time learn something new and refresh what I have forgotten. Thanks for the great article!

2

REPLY

Vivek Ojha

Apr 6

Awesome. I wish we could see this visualization on our actual git repository.
Is there any way ?

2

REPLY

Rodrigo Andrés Contreras Vilina

Apr 8

This is a really good guide, awesome work, thanks!!

2

REPLY

Utkarsh Talwar

Apr 7

Quite the informative piece, Lydia! I shared it in our Telegram newsletter/channel for devs. 👉 Link

1

REPLY

John

Apr 7

Great post

1

REPLY

Matthieu Cneude

Apr 7

Really, really nice work. It's one of the best git tutorial/cheatsheet I've seen!

1

REPLY

Andrey Ginger

Apr 7

I wonder what changes in the tasks of the project corresponds to all this hell?

1

REPLY

VIEW FULL DISCUSSION (64 COMMENTS)

code of conduct - report abuse

Classic DEV Post from Jun 21 '19

Is Ubuntu Or Fedora A Better Distro For Programmers?

Prahlad Yeri

102 97

From one of our Community Sponsors

Feature Flags and GitOps. 5 Use Cases to Help You 'Git'r Done.

Kristin Baskett

Even if adopting GitOps is still aspirational for your team, you can use CloudBees Rollout to manage your feature flags.

25 2

Another Post You Might Like

Notes on algorithms

Emilie Gervais

notes on algorithms after CS50 week 3

730 13

Another Post You Might Like

BaseCS Season 2 Video Series Is Coming Your Way

Vaidehi Joshi

Computer science concepts explained for everyone

140 11

Yggdrasil as Distributed PubSub

Alex de Sousa - Apr 16

Git Commands I Always Forget

David Dal Busco - Apr 16

GitHub Pages: Static web hosting made simple

Roelof Jan Elsinga - Apr 16

Modern React Redux Tutorials with Redux toolkit - 2020

GaneshMani - Apr 16

Home About Privacy Policy Terms of Use Contact Code of Conduct