

When do you use Git rebase instead of Git merge?

Asked 15 years, 7 months ago Modified 30 days ago

Viewed 1.1m times



When is it recommended to use Git rebase vs. Git merge?

2479

Do I still need to merge after a successful rebase?



git

version-control

git-merge

git-rebase



Share

Improve this question

Follow

edited Dec 9, 2019 at 12:52



Peter Mortensen

31.6k ● 22 ● 109 ● 133

asked Apr 29, 2009 at 20:26



Coocoo4Cocoa

50.7k ● 50 ● 152 ● 176

4 See: stackoverflow.com/questions/457927/... – TStamper
Apr 29, 2009 at 20:27

46 this is good: atlassian.com/git/tutorials/merging-vs-rebasing
– stackexchanger Aug 28, 2015 at 19:47

21 One problem with people who like to use rebase is that it deters them from pushing their code regularly. So wanting

clean history prevents them from sharing their code, which I think is more important. – [static_rtti](#) Oct 6, 2015 at 8:38

21 @static_rtti: That's just not true. You're using a rebase-based flow wrong if it prevents you from pushing your changes regularly. – [juzzlin](#) Jan 9, 2016 at 21:14

12 My heuristic: "Try rebase, if it dissolves into conflict resolution hell give up and merge master into your branch and move on." - more detail in my post timwise.co.uk/2019/10/14/merge-vs-rebase (in the context of commercial development) – [Tim Abell](#) Sep 2, 2020 at 14:55

19 Answers

Sorted by:

Highest score (default)



Short Version

1688



- Merge takes all the changes in one branch and merges them into another branch in one commit.
- Rebase says I want the point at which I branched to move to a new starting point



So when do you use either one?

Merge

- Let's say you have created a branch for the purpose of developing a single feature. When you want to bring those changes back to master, you probably want **merge**.

Rebase

- A second scenario would be if you started doing some development and then another developer made an unrelated change. You probably want to pull and then **rebase** to base your changes from the current version from the repository.

Squashing: All commits are preserved in both cases (for example: "add feature", then "typo", then "oops typo again"...). Commits can be combined into a single commits by squashing. Squashing can be done as part of a merge or rebase operation (--squash flag), in which case it's often called a squash-merge or a squash-rebase.

Pull Requests: Popular git servers (Bitbucket, GitLab, GitHub, etc...) allow to configure how pull requests are merged on a per-repo basis. the UI may show a "Merge" button by convention but the button can do any operations with any flags (keywords: merge, rebase, squash, fast-forward).

Share Improve this answer

Follow

edited Nov 16, 2022 at 10:41



user5994461

6,998 ● 4 ● 42 ● 65

answered Apr 29, 2009 at 20:38



Rob Di Marco

44.9k ● 9 ● 71 ● 57

147 @Rob mentioned maintaining interim commits when merging. I believe by default merging branch B (a feature branch you've been working on) into branch M (the master

branch) will create one commit in M for each commit that was made in B since the two diverged. But if you merge using the `--squash` option, all of the commits made on branch B will be "lumped together" and merged as a single commit on branch M, keeping the log on your master branch nice and clean. Squashing is probably what you want if you have numerous developers working independently and merging back into master. – [spaaarky21](#)
Jan 24, 2013 at 21:46 ✎

38 I believe @spaaarky21's assumption about merging is not correct. If you merge a branch B into the master M, there will be only a single commit on M (even if B has multiple commits), regardless of whether you use a plain or `--squash` merge. What `--squash` will do is eliminate the reference to B as a parent. A good visualization is here: syntevo.com/smartgithg/howtos.html?page=workflows.merge – [jpeskin](#) May 26, 2013 at 15:19 ✎

21 @jpeskin That's not what I'm seeing. I just did a quick test to verify. Create a directory with a text file, `init` a new repo, `add` the file and `commit`. Checkout a new feature branch (`checkout -b feature`.) Change the text file, commit and repeat so that there are two new commits on the feature branch. Then `checkout master` and `merge feature`. In `log`, I see my initial commit on master, followed by the two that were merged from feature. If you `merge --squash feature`, feature is merged into master but not committed, so the only new commit on master will be the one you make yourself. – [spaaarky21](#)
May 27, 2013 at 20:55 ✎

36 @spaaarky21 It looks like we're both half right. When a fast-forward merge is possible (as in your example), git will default to including all commits in the feature branch B (or as you suggest, you can use `--squash` to combine into a single commit). But in the case where there are two divergent branches M and B that you are merging, git will not include all of the individual commits from branch B if

merged into M (whether or not you use --squash). – [jpeskin](#)

May 28, 2013 at 1:36 

- 13 Why is that the "(you don't care about maintaining all of the interim commits)" aside still in this answer? It made no sense in '09 and it makes no sense now. Also, surely you would only want to rebase if another developer made *related changes* that you needed - if they made unrelated changes, your feature branch should merge easily without conflicts anyway, and your history would be maintained.

– [Mark Booth](#) Apr 27, 2016 at 10:59



It's simple. With rebase you say to use another branch as the new **base** for your work.

604



If you have, for example, a branch `master`, you create a branch to implement a new feature, and say you name it `cool-feature`, of course, the master branch is the base for your new feature.



Now, at a certain point, you want to add the new feature you implemented in the `master` branch. You could just switch to `master` and merge the `cool-feature` branch:

```
$ git checkout master
$ git merge cool-feature
```

But this way a new dummy commit is added. If you want to avoid spaghetti-history you can **rebase**:

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

And then merge it in `master` :

```
$ git checkout master  
$ git merge cool-feature
```

This time, since the topic branch has the same commits of master plus the commits with the new feature, the merge will be just a fast-forward.

Share Improve this answer

Follow

edited Jul 19, 2021 at 1:04



[Anton Menshov](#)

2,306 ● 15 ● 36 ● 58

answered Feb 5, 2012 at 6:28



[Aldo 'xoen' Giambelluca](#)

12.4k ● 7 ● 35 ● 39

104 but this way a new dummy commit is added, if you want to avoid spaghetti-history - how is it bad? – [Incerteza](#) May 19, 2014 at 14:07

15 Also, the --no-ff flag of merge is very very useful.
– [Aldo 'xoen' Giambelluca](#) May 22, 2014 at 12:39

18 @アレックス as user [Sean Schofield](#) puts it in a comment: "Rebase is also nice because once u do eventually merge ur stuff back into master (which is trivial as already described) you have it sitting at the "top" of ur commit history. On bigger projects where features may be written but merged several weeks later, you don't want to just merge them into the master because they get "stuffed" into the master way back in the history. Personally I like being able to do git log and see that recent feature right at the "top." Note the commit dates are preserved - rebase

doesn't change that information. " – [Adriano](#) Mar 1, 2016 at 4:37

- 6 I think it bears repeating here -- remember that all these terms (`merge` , `rebase` , `fast-forward` , etc.) are referring to specific manipulations of a directed acyclic graph. They become easier to reason about with that mental model in mind. – [Roy Tinker](#) Jan 27, 2017 at 19:24



- 61 @Aldo There's nothing "clean" or "tidy" about a rebased history. It's generally *filthy* and IMHO awful because you have no idea what really went on. The "cleanest" Git history is the one that actually occurred. :) – [Marnen Laibow-Koser](#) May 14, 2018 at 13:58
-



TL;DR

500

If you have any doubt, use merge.



Short Answer



The only differences between a rebase and a merge are:

- The resulting tree structure of the history (generally only noticeable when looking at a commit graph) is different (one will have branches, the other won't).
- Merge will generally create an extra commit (e.g. node in the tree).
- Merge and rebase will handle conflicts differently. Rebase will present conflicts one commit at a time where merge will present them all at once.

So the short answer is to *pick rebase or merge based on what you want your history to look like*.

Long Answer

There are a few factors you should consider when choosing which operation to use.

Is the branch you are getting changes from shared with other developers outside your team (e.g. open source, public)?

If so, don't rebase. Rebase alters the history of the branch and those developers will have broken/inconsistent repositories unless they use `git pull --rebase`. This is a good way to upset other developers quickly.

How skilled is your development team?

Rebase is a destructive operation. That means, if you do not apply it correctly, **you could lose committed work and/or break the consistency of other developer's repositories.**

I've worked on teams where the developers all came from a time when companies could afford dedicated staff to deal with branching and merging. Those developers don't know much about Git and don't want to know much. In

these teams I wouldn't risk recommending rebasing for any reason.

Does the branch itself represent useful information

Some teams use the branch-per-feature model where each branch represents a feature (or bugfix, or sub-feature, etc.) In this model the branch helps identify sets of related commits. For example, one can quickly revert a feature by reverting the merge of that branch (to be fair, this is a rare operation). Or diff a feature by comparing two branches (more common). Rebase creates a linear history which does not include the branch information. It would not be possible to identify which commits belong to the feature and reverting it would be more difficult.

I've also worked on teams that used the branch-per-developer model (we've all been there). In this case the branch itself doesn't convey any additional information (the commit already has the author). There would be no harm in rebasing.

Might you want to revert the merge for any reason?

Reverting (as in undoing) a rebase is considerably difficult and/or impossible (if the rebase had conflicts) compared to reverting a merge. If you think there is a chance you will want to revert then use merge.

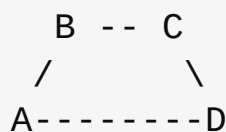
Do you work on a team? If so, are you willing to take an all or nothing approach on this branch?

Rebase operations need to be pulled with a corresponding `git pull --rebase`. If you are working by yourself you may be able to remember which you should use at the appropriate time. If you are working on a team this will be very difficult to coordinate. This is why most rebase workflows recommend using rebase for all merges (and `git pull --rebase` for all pulls).

Common Myths

Merge destroys history (squashes commits)

Assuming you have the following merge:



Some people will state that the merge "destroys" the commit history because if you were to look at the log of only the master branch (A -- D) you would miss the important commit messages contained in B and C.

If this were true we wouldn't have [questions like this](#). Basically, you will see B and C unless you explicitly ask

not to see them (using `--first-parent`). This is very easy to try for yourself.

Rebase allows for safer/simpler merges

The two approaches merge differently, but it is not clear that one is always better than the other and it may depend on the developer workflow. For example, if a developer tends to commit regularly (e.g. maybe they commit twice a day as they transition from work to home) then there could be a lot of commits for a given branch. Many of those commits might not look anything like the final product (I tend to refactor my approach once or twice per feature). If someone else was working on a related area of code and they tried to rebase my changes it could be a fairly tedious operation.

Rebase is cooler / sexier / more professional

If you like to alias `rm` to `rm -rf` to "save time" then maybe rebase is for you.

My Two Cents

I always think that someday I will come across a scenario where Git rebase is the awesome tool that solves the problem. Much like I think I will come across a scenario where Git reflog is an awesome tool that solves my

problem. I have worked with Git for over five years now. It hasn't happened.

Messy histories have never really been a problem for me. I don't ever just read my commit history like an exciting novel. A majority of the time I need a history I am going to use Git blame or Git bisect anyway. In that case, having the merge commit is actually useful to me, because if the merge introduced the issue, that is meaningful information to me.

Update (4/2017)

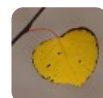
I feel obligated to mention that I have personally softened on using rebase although my general advice still stands. I have recently been interacting a lot with the [Angular 2 Material](#) project. They have used rebase to keep a very clean commit history. This has allowed me to very easily see what commit fixed a given defect and whether or not that commit was included in a release. It serves as a great example of using rebase correctly.

Share Improve this answer

edited Nov 20 at 19:09

Follow

answered Apr 13, 2016 at 2:16



Pace

43.6k ● 14 ● 125 ● 162

42 I mostly love this answer. But: Rebase doesn't make a "clean" history. It makes a more linear history, but that's not the same thing at all, since who knows now much "dirt" each commit is hiding? The cleanest, clearest Git history is the one that keeps branch and commit integrity.

– [Marnen Laibow-Koser](#) May 14, 2018 at 14:01

10 It's worth mentioning that git has recently changed its `git pull` behavior to include the `--rebase` flag by default. That means that doing rebases on branches used by multiple developers is a little less dangerous. A person pulling your changes might be surprised that there are some conflicts to be resolved during such an operation, but there would be no disaster. – [pkubik](#) Jul 31, 2020 at 13:31

4 Another con for rebase, imo it's harder in high-velocity git repos especially where the build time is > average time between merges. If branches are constantly being merged in, you need to keep rebasing until it's your turn which can be tricky if you're also coordinating with a build to pass. e.g. monorepos rebase merges to a shared branch might be difficult – [nijave](#) Aug 11, 2021 at 19:51

4 this is the best answer in this topic – [sekomer](#) Nov 28, 2023 at 18:56

How does rebase destroy the branch? – [user3310334](#) Nov 15 at 17:27



I just created an FAQ for my team in my own words which answers this question. Let me share:

409



What is a `merge`?



A commit, that combines all changes of a different branch into the current.



What is a `rebase`?

Re-committing all commits of the current branch onto a different base commit.

What are the main differences between `merge` and `rebase`?

1. `merge` executes only **one** new commit. `rebase` typically executes **multiple** (number of commits in current branch).
2. `merge` produces a **new** generated commit (the so called merge-commit). `rebase` only moves **existing** commits.

In which situations should we use a `merge`?

Use `merge` whenever you want to add changes of a branched out branch **back** into the base branch.

Typically, you do this by clicking the "Merge" button on Pull/Merge Requests, e.g. on GitHub.

In which situations should we use a `rebase`?

Use `rebase` whenever you want to add **changes of a base branch** back to a branched out branch.

Typically, you do this in `feature` branches whenever there's a change in the `main` branch.

Why not use `merge` to merge changes from the base branch into a feature branch?

1. The git history will include many **unnecessary merge commits**. If multiple merges were needed in a feature branch, then the feature branch might even hold more merge commits than actual commits!
2. This creates a loop which **destroys the mental model that Git was designed by** which causes troubles in any visualization of the Git history.

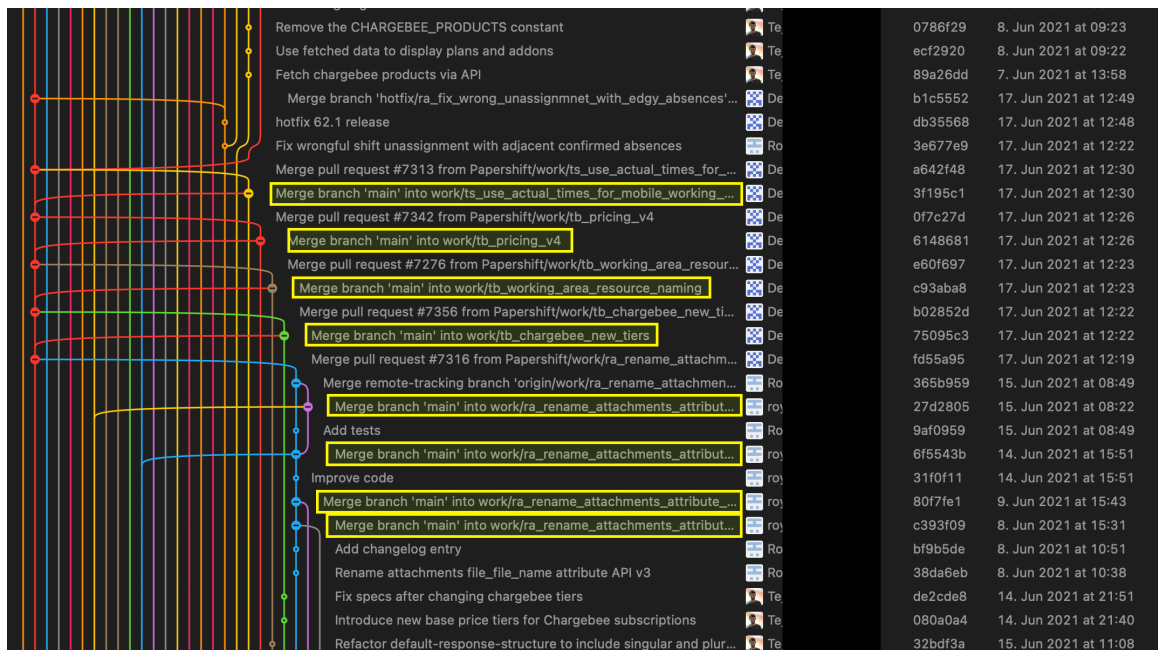
Imagine there's a river (e.g. the "Nile"). Water is flowing in one direction (direction of time in Git history). Now and then, imagine there's a branch to that river and suppose most of those branches merge back into the river. That's what the flow of a river might look like naturally. It makes sense.

But then imagine there's a small branch of that river. Then, for some reason, **the river merges into the branch** and the branch continues from there. The

river has now technically disappeared, it's now in the branch. But then, somehow magically, that branch is merged back into the river. Which river you ask? I don't know. The river should actually be in the branch now, but somehow it still continues to exist and I can merge the branch back into the river. So, the river is in the river. Kind of doesn't make sense.

This is exactly what happens when you `merge` the base branch into a `feature` branch and then when the `feature` branch is done, you merge that back into the base branch again. The mental model is broken. And because of that, you end up with a branch visualization that's not very helpful.

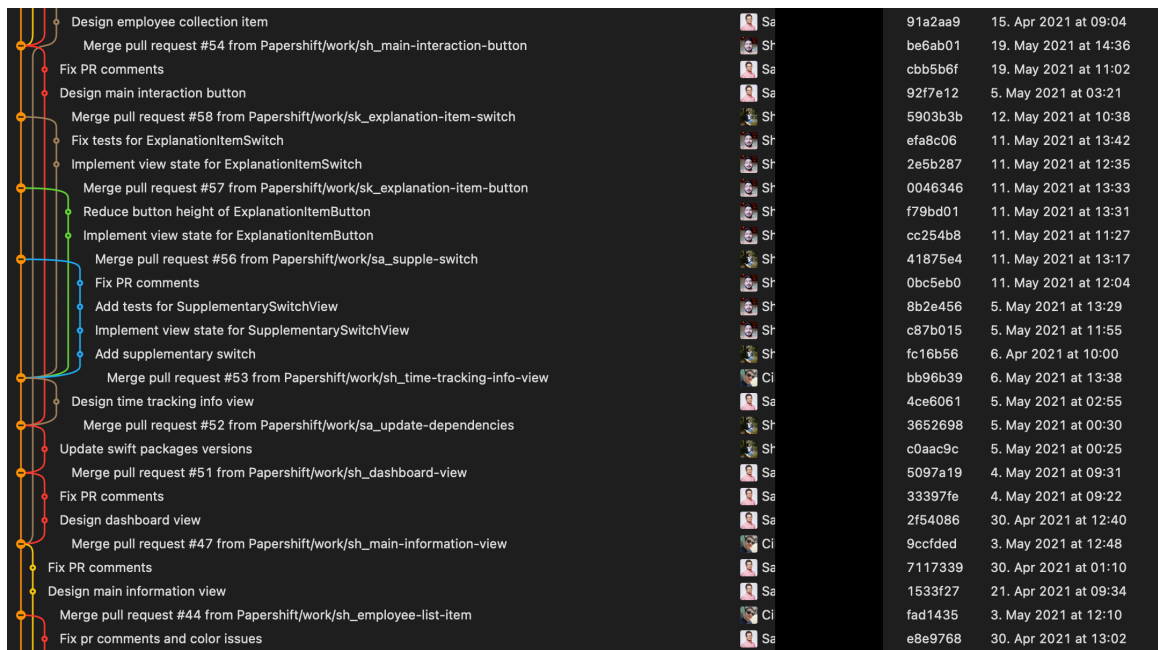
Example Git History when using `merge`:



Note the many commits starting with `Merge branch 'main' into ...`. They don't even exist if you rebase (there, you will only have pull request merge commits).

Also many visual branch merge loops (`main` into `feature` into `main`).

Example Git History when using `rebase` :



Much cleaner Git history with much less merge commits and no cluttered visual branch merge loops whatsoever.

Are there any downsides / pitfalls with `rebase` ?

Yes:

1. Because a `rebase` moves commits (technically re-executes them), the commit date of all moved commits will be the time of the rebase and the **git history might look like it lost the initial commit time**. So, if the exact date of a commit is needed in

all tooling for some reason, then `merge` is the better option. But typically, a clean git history is much more useful than exact commit dates. And the [author-date](#) field will continue to hold the original commit date where needed.

2. If the rebased branch has multiple commits that change the same line and that line was also changed in the base branch, you might need to solve merge conflicts for that same line multiple times, which you never need to do when merging. So, on average, there's more merge conflicts to solve.

Note that there's a common misconception that a `rebase` causes more merge conflicts even if the same line was **not** edited multiple times. That's because multiple commits are executed and so for example 10 conflicts that would have come up in a single `merge` commit are distributed among these multiple commits. So you get the "there are merge conflicts" message of your Git client more often, but the total number of conflicts is still the same (unless you changed the same line multiple times in your branch).

Tips to reduce merge conflicts when using `rebase` :

1. **Rebase often.** I typically recommend doing it at least once a day.
2. Try to **squash changes** on the same line into one commit as much as possible.

answered Oct 12, 2020 at 14:20

**Cihat Gündüz**

21.4k ● 8 ● 64 ● 83

2 I would remove the downside (2) completely from your list because, as you said, squashing is a perfect solution to (2) and it always works – [Nir O.](#) May 18, 2021 at 10:25

1 Thanks, very useful. This highlights an important point of there being multiple merge commits that can be avoided using rebase. – [chetan](#) Nov 8, 2021 at 19:56

1 Rebase has way more issues. Especially when people rebases remote branches (force push (possible overwrite of somebodies work) and git branch reset on pull (you have to check whenever the remote changes were legit and not a malicious actor injected some code into the branch)). It also causes when people rebasing the target branch (continuously you have to update your own branches as it would give conflicts with the target on merge). – [golddragon007](#) Jun 8, 2022 at 13:03

1 @Jeehut There is a need to start a branch from another feature branch, whenever you have a dependency on another ticket's feature and you can't implement it without it. Even worse if you implemented the dependency as you can't review it, but when you have many such tickets that depend on a previous one you can't wait for somebody always to merge the dependency. On top of that rebasing, the remote branch is not a good idea, once published shouldn't be altered. I also saw rebased release branches, in that case, all the merges are gone, but have fun doing a git bisect on it. – [golddragon007](#) Jun 10, 2022 at 15:31

- 3 All the commits look out of order, chronologically and by the possible branch (which is gone). It's all done for "clean history"; I just can't find the "history" part in it.
- [golddragon007](#) Jun 10, 2022 at 15:32 ✎
-



295



To complement [my own answer](#) mentioned [by TSamper](#),

- a rebase is quite often a good idea to do before a merge, because the idea is that you integrate in your branch **Y** the work of the branch **B** upon which you will merge.

But again, before merging, you resolve any conflict in *your* branch (i.e.: "rebase", as in "replay my work in my branch starting from a recent point from the branch **B**).

If done correctly, the subsequent merge from your branch to branch **B** can be fast-forward.

- a merge directly impacts the destination branch **B**, which means the merges better be trivial, otherwise that branch **B** can be long to get back to a stable state (time for you solve all the conflicts)

the point of merging after a rebase?

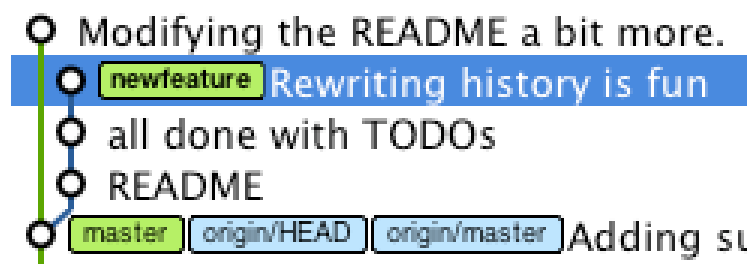
In the case that I describe, I rebase **B** onto my branch, just to have the opportunity to replay my work from a more recent point from **B**, but while staying into my branch.

In this case, a merge is still needed to bring my "replayed" work onto **B**.

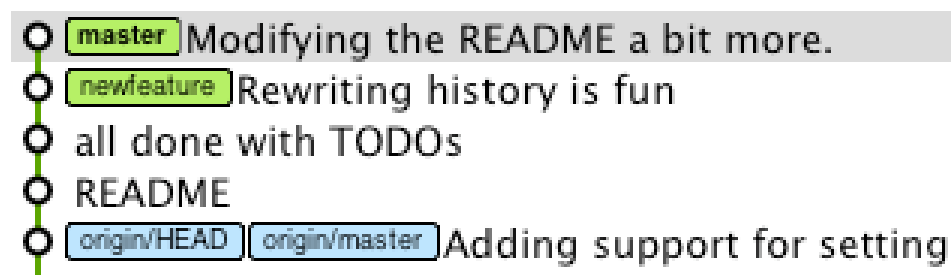
The other scenario ([described in Git Ready](#) for instance), is to bring your work directly in **B** through a rebase (which does conserve all your nice commits, or even give you the opportunity to re-order them through an interactive rebase).

In that case (where you rebase while being in the B branch), you are right: no further merge is needed:

A Git tree at default when we have not merged nor rebased



we get by rebasing:



That second scenario is all about: how do I get new-feature back into master.

My point, by describing the first rebase scenario, is to remind everyone that a rebase can also be used as a preliminary step to that (that being "get new-feature back into master").

You can use rebase to first bring master "in" the new-feature branch: the rebase will replay new-feature commits from the `HEAD master`, but still in the new-feature branch, effectively moving your branch starting point from an old master commit to `HEAD-master`.

That allows you to resolve any conflicts in *your* branch (meaning, in isolation, while allowing master to continue to evolve in parallel if your conflict resolution stage takes too long).

Then you can switch to master and merge `new-feature` (or rebase `new-feature` onto `master` if you want to preserve commits done in your `new-feature` branch).

So:

- "rebase vs. merge" can be viewed as two ways to import a work on, say, `master`.
- But "rebase then merge" can be a valid workflow to first resolve conflict in isolation, then bring back your work.

Share Improve this answer

Follow

edited Dec 9, 2019 at 12:58



Peter Mortensen

31.6k ● 22 ● 109 ● 133



answered Apr 29, 2009 at 20:44



VonC

1.3m ● 558 ● 4.7k ● 5.6k

21 merge after rebase is a trivial fast forward without having to resolve conflicts. – [obecalp](#) Apr 30, 2009 at 19:32

- 5 @obelcap: Indeed, this is kind of the idea: you take all the problem-conflict in *your* environment (rebase master within your new-feature branch), and then co master, merge new-feature: 1 pico-second (fast-forward) if master had no evolutions – VonC Apr 30, 2009 at 21:29
-
- 29 Rebase is also nice because once you do eventually merge your stuff back into master (which is trivial as already described) you have it sitting at the "top" of your commit history. On bigger projects where features may be written but merged several weeks later, you don't want to just merge them into the master because they get "stuffed" into the master way back in the history. Personally I like being able to do git log and see that recent feature right at the "top." Note the commit dates are preserved - rebase doesn't change that information. – Sean Schofield Sep 3, 2009 at 14:11
-
- 3 @Joe: mentally, you are saying "replay any of my changes (done in isolation in my private branch) on top of that other branch, but leave me in my private branch once the rebase is done". That is a good opportunity to clean-up the local history, avoiding "checkpoint commits", broken bisect and incorrect blame results. See "Git workflow": sandofsky.com/blog/git-workflow.html – VonC Aug 15, 2011 at 17:57 
-
- 4 @scoarescoare the key is to see how you local changes are compatible *on top* of the latest upstream branch. If one of your commit introduces a conflict, you will see it right away. A merge introduce only one (merged) commit, which might trigger many conflict without an easy way to see which one, amongst your own local commits, did add said conflict. So in addition to a cleaner history, you get a more precise view of the changes *you* introduce, commit by commit (replayed by the rebase), as opposed to *all* the changes introduced by the upstream branch (dumped into one single merge). – VonC Jan 25, 2013 at 6:17 
-



243



A lot of answers here say that merging turns all your commits into one, and therefore suggest to use rebase to preserve your commits. **This is incorrect. And a bad idea if you have pushed your commits already.**

Merge does *not* obliterate your commits. Merge preserves history! (just look at gitk) Rebase rewrites history, which is a Bad Thing after you've *pushed* it.

Use merge -- not rebase whenever you've already pushed.

[Here is Linus' \(author of Git\) take on it](#) (now hosted on my own blog, as [recovered by the Wayback Machine](#)). It's a really good read.

Or you can read my own version of the same idea below.

Rebasing a branch on master:

- provides an incorrect idea of how commits were created
- pollutes master with a bunch of intermediate commits that may not have been well tested
- could actually introduce build breaks on these intermediate commits because of changes that were made to master between when the original topic branch was created and when it was rebased.
- makes finding good places in master to checkout difficult.

- Causes the timestamps on commits to not align with their chronological order in the tree. So you would see that commit A precedes commit B in master, but commit B was authored first. (What?!)
- Produces more conflicts, because individual commits in the topic branch can each involve merge conflicts which must be individually resolved (further lying in history about what happened in each commit).
- is a rewrite of history. If the branch being rebased has been pushed anywhere (shared with anyone other than yourself) then you've screwed up everyone else who has that branch since you've rewritten history.

In contrast, merging a topic branch into master:

- preserves history of where topic branches were created, including any merges from master to the topic branch to help keep it current. You really get an accurate idea of what code the developer was working with when they were building.
- master is a branch made up mostly of merges, and each of those merge commits are typically 'good points' in history that are safe to check out, because that's where the topic branch was ready to be integrated.
- all the individual commits of the topic branch are preserved, including the fact that they were in a topic branch, so isolating those changes is natural and you can drill in where required.

- merge conflicts only have to be resolved once (at the point of the merge), so intermediate commit changes made in the topic branch don't have to be resolved independently.
- can be done multiple times smoothly. If you integrate your topic branch to master periodically, folks can keep building on the topic branch, and it can keep being merged independently.

Share Improve this answer

edited Jan 23, 2020 at 19:14

Follow

answered Feb 3, 2014 at 22:17



Andrew Arnott

81.7k ● 28 ● 137 ● 180

4 Also, git merge has the "--no-ff" (no fast-forward) option that allows you to revert all the changes introduced by a certain merge really easily. – [Bitcoin Cash - ADA enthusiast](#) Sep 8, 2014 at 22:07 ✎

4 Just to make it more clear: You refer to the situation 'whenever you've already pushed' -- this should be bold. The Link to Linus post is great, btw., clarifies it. – [honzajde](#) Feb 14, 2015 at 14:40

3 but isn't it best practice to "update" from master into your topic branch, before you merge the topic branch into master via PR (to resolve conflicts in your branch, not the master)? We're doing it like that so most topic branches have as a last commit "merge branch master into topic-..." but here this is listed as a "feature" of rebasing and nobody mentions it for merging... ? – [ProblemsOfSumit](#) Jan 7, 2016 at 11:40 ✎

- 4 @AndrewArnott "Most topic branches should be able to merge without conflicts into their target branches" How should that be possible when 20 devs are working on 30 branches? There will be merges while you're working on yours - so of course you have to update you topic branch from target before creating a PR... no? – [ProblemsOfSumit](#) Jan 11, 2016 at 8:21
-
- 4 Not usually, @Sumit. Git can merge either direction just fine even though changes have been made to either or both branches. Only when the same lines of code (or very close) are modified across two branches will you get conflicts. If that happens frequently on any team, the team should rethink how they distribute work since resolving conflicts is a tax and slows them down. – [Andrew Arnott](#) Jan 15, 2016 at 13:29
-



95

TLDR: It depends on what is most important - a tidy history or a true representation of the sequence of development



If a tidy history is the most important, then you would rebase first and then merge your changes, so it is clear exactly what the new code is. **If you have already pushed your branch, don't rebase unless you can deal with the consequences.**



If true representation of sequence is the most important, you would merge without rebasing.

Merge means: Create a single new commit that merges my changes into the destination. **Note:** This new commit will have two parents - the latest commit from your string

of commits and the latest commit of the other branch you're merging.

Rebase means: Create a whole new series of commits, using my current set of commits as hints. In other words, calculate what my changes would have looked like if I had started making them from the point I'm rebasing on to. After the rebase, therefore, you might need to re-test your changes and during the rebase, you would possibly have a few conflicts.

Given this, why would you rebase? Just to keep the development history clear. Let's say you're working on feature X and when you're done, you merge your changes in. The destination will now have a single commit that would say something along the lines of "Added feature X". Now, instead of merging, if you rebased and then merged, the destination development history would contain all the individual commits in a single logical progression. This makes reviewing changes later on much easier. Imagine how hard you'd find it to review the development history if 50 developers were merging various features all the time.

That said, if you have already pushed the branch you're working on upstream, you should not rebase, but merge instead. For branches that have not been pushed upstream, rebase, test and merge.

Another time you might want to rebase is when you want to get rid of commits from your branch before pushing upstream. For example: Commits that introduce some

debugging code early on and other commits further on that clean that code up. The only way to do this is by performing an interactive rebase: `git rebase -i <branch/commit/tag>`

UPDATE: You also want to use rebase when you're using Git to interface to a version control system that doesn't support non-linear history ([Subversion](#) for example). When using the git-svn bridge, it is very important that the changes you merge back into Subversion are a sequential list of changes on top of the most recent changes in trunk. There are only two ways to do that: (1) Manually re-create the changes and (2) Using the rebase command, which is a lot faster.

UPDATE 2: One additional way to think of a rebase is that it enables a sort of mapping from your development style to the style accepted in the repository you're committing to. Let's say you like to commit in small, tiny chunks. You have one commit to fix a typo, one commit to get rid of unused code and so on. By the time you've finished what you need to do, you have a long series of commits. Now let's say the repository you're committing to encourages large commits, so for the work you're doing, one would expect one or maybe two commits. How do you take your string of commits and compress them to what is expected? You would use an interactive rebase and squash your tiny commits into fewer larger chunks. The same is true if the reverse was needed - if your style was a few large commits, but the repository demanded long strings of small commits. You would use a rebase to do

that as well. If you had merged instead, you have now grafted your commit style onto the main repository. If there are a lot of developers, you can imagine how hard it would be to follow a history with several different commit styles after some time.

UPDATE 3: Does one still need to merge after a successful rebase? Yes, you do. The reason is that a rebase essentially involves a "shifting" of commits. As I've said above, these commits are calculated, but if you had 14 commits from the point of branching, then assuming nothing goes wrong with your rebase, you will be 14 commits ahead (of the point you're rebasing onto) after the rebase is done. You had a branch before a rebase. You will have a branch of the same length after. You still need to merge before you publish your changes. In other words, rebase as many times as you want (again, only if you have not pushed your changes upstream). Merge only after you rebase.

Share Improve this answer

edited Dec 11, 2019 at 20:01

Follow


answered Feb 5, 2012 at 6:47



Carl

44.3k ● 10 ● 83 ● 109

-
- 2 A merge with master could result in a fast forward. In a feature branch there may be some commits, which have minor bugs or don't even compile. If you do only unit testing in a feature branch, some errors in integration may slip through. Before merging with master, integration tests are

required and can show some bugs. If these are fixed, the feature could be integrated. As you don't wish to commit buggy code to master, a rebase seems necessary in order to prevent an all-commits-fast-forward. – [mbx](#) Jul 15, 2012 at 18:15 

1 @mbx `git merge` supports the `--no-ff` option which forces it to make a merge commit. – [Gavin S. Yancey](#) Jun 8, 2017 at 22:31

3 It doesn't really "depend" when you can gain the same advantages of a "tidy history" without the massive disadvantages of it by simply using search / filter commands to view the history. Making rebasing practically useless. – [byteWalrus](#) Jan 13, 2022 at 19:10



88

While merging is definitely the easiest and most common way to integrate changes, it's not the only one: **Rebase** is an alternative means of integration.



Understanding Merge a Little Better



When Git performs a merge, it looks for three commits:

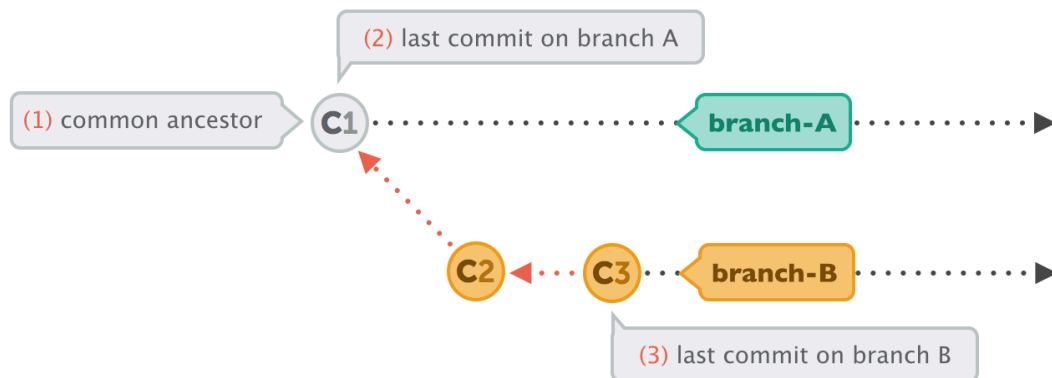


- (1) Common ancestor commit. If you follow the history of two branches in a project, they always have at least one commit in common: at this point in time, both branches had the same content and then evolved differently.
- (2) + (3) Endpoints of each branch. The goal of an integration is to combine the current states of two branches. Therefore, their respective latest revisions

are of special interest. Combining these three commits will result in the integration we're aiming for.

Fast-Forward or Merge Commit

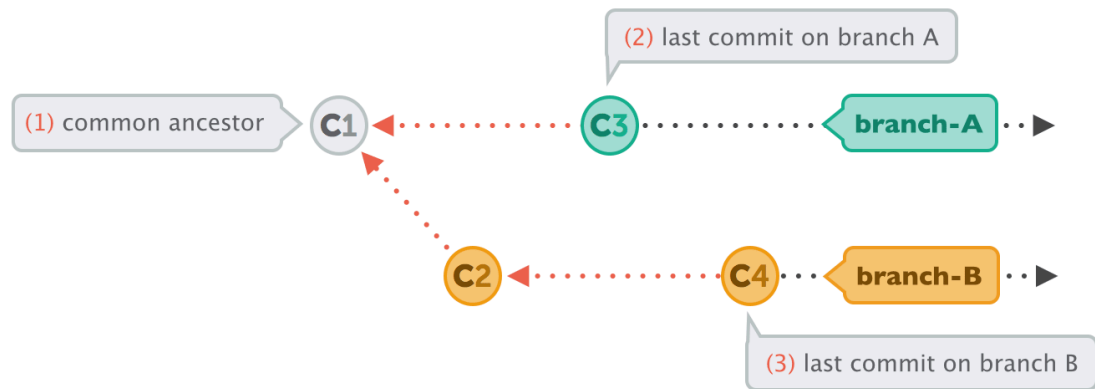
In very simple cases, one of the two branches doesn't have any new commits since the branching happened - its latest commit is still the common ancestor.



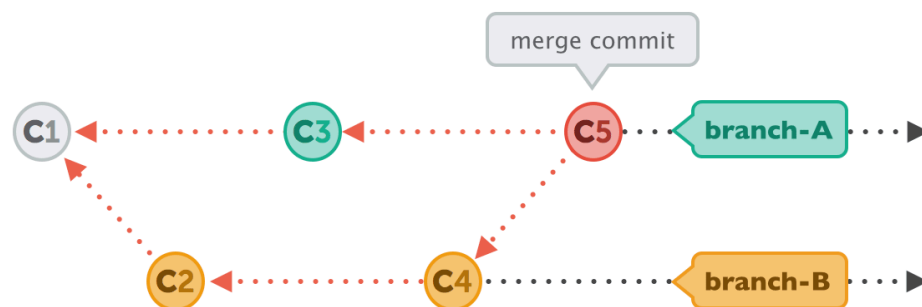
In this case, performing the integration is dead simple: Git can just add all the commits of the other branch on top of the common ancestor commit. In Git, this simplest form of integration is called a "fast-forward" merge. Both branches then share the exact same history.



In a lot of cases, however, both branches moved forward individually.



To make an integration, Git will have to create a new commit that contains the differences between them - the merge commit.



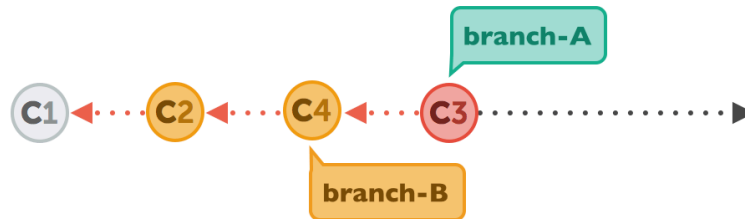
Human Commits & Merge Commits

Normally, a commit is carefully created by a human being. It's a meaningful unit that wraps only related changes and annotates them with a comment.

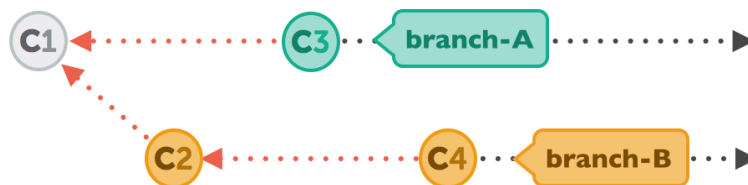
A merge commit is a bit different: instead of being created by a developer, it gets created automatically by Git. And instead of wrapping a set of related changes, its purpose is to connect two branches, just like a knot. If you want to understand a merge operation later, you need to take a look at the history of both branches and the corresponding commit graph.

Integrating with Rebase

Some people prefer to go without such automatic merge commits. Instead, they want the project's history to look as if it had evolved in a single, straight line. No indication remains that it had been split into multiple branches at some point.



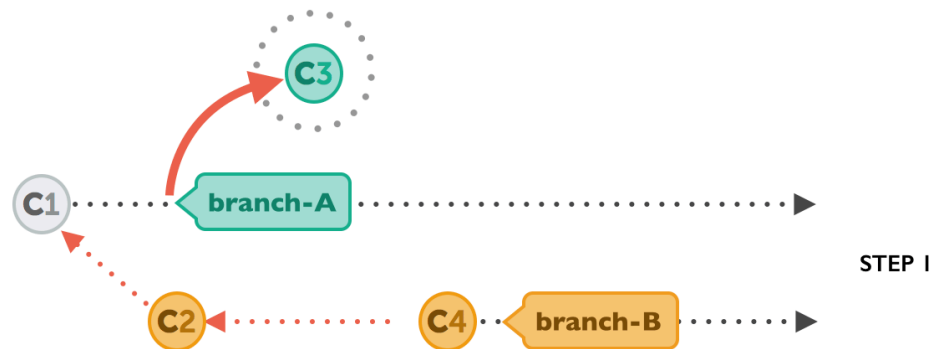
Let's walk through a rebase operation step by step. The scenario is the same as in the previous examples: we want to integrate the changes from branch-B into branch-A, but now by using rebase.



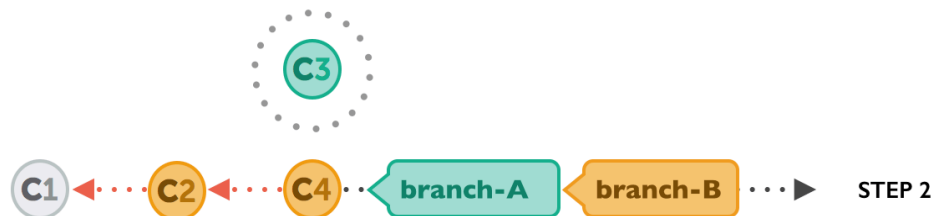
We will do this in three steps

1. `git rebase branch-A` // Synchronises the history with branch-A
2. `git checkout branch-A` // Change the current branch to branch-A
3. `git merge branch-B` // Merge/take the changes from branch-B to branch-A

First, Git will "undo" all commits on branch-A that happened after the lines began to branch out (after the common ancestor commit). However, of course, it won't discard them: instead you can think of those commits as being "saved away temporarily".

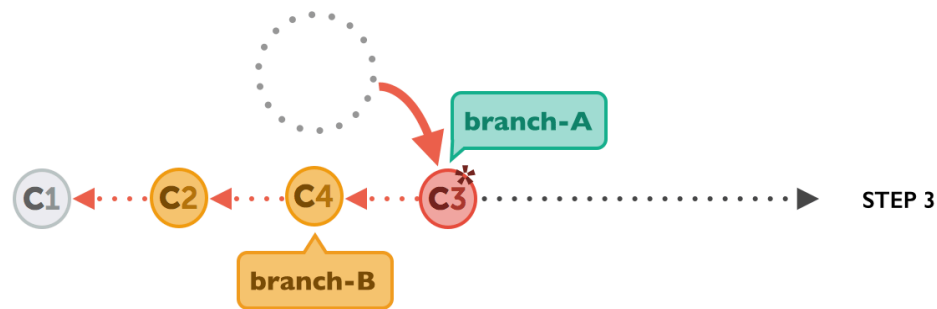


Next, it applies the commits from branch-B that we want to integrate. At this point, both branches look exactly the same.



In the final step, the new commits on branch-A are now reapplied - but on a new position, on top of the integrated commits from branch-B (they are re-based).

The result looks like development had happened in a straight line. Instead of a merge commit that contains all the combined changes, the original commit structure was preserved.



Finally, you get a clean branch **branch-A** with no unwanted and auto generated commits.

Note: Taken from the awesome [post](#) by [git-tower](#). The *disadvantages* of `rebase` is also a good read in the same post.

Share Improve this answer

Follow

edited Dec 9, 2019 at 13:55



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Oct 12, 2017 at 11:52



Abdullah Mohammad
Motiullah

12.5k ● 6 ● 72 ● 82

I think I neither want to need to look into the history of branch-b nor I want to see every commit of branch-b in the history of branch-a. For me branch-b is a messy place for brain-storming and branch-a is the real program. I want only the useful result of the brain-storming in the history of branch-a. I do not think that it is a good idea to copy C2 and C4 into branch-a. I think it is necessary to summarize the results of C2 and C4 when they are merged into branch-a. – [ceving](#)
Jun 21, 2023 at 11:28

Before merge/rebase:



71



```

A <- B <- C    [master]
^
 \
  D <- E        [branch]

```

After `git merge master`:

```

A <- B <- C
^           ^
 \         \
  D <- E <- F

```

After `git rebase master`:

```

A <- B <- C <- D' <- E'

```

(A, B, C, D, E and F are commits)

This example and much more well illustrated information about Git can be found in [Git The Basics Tutorial](#).

Share Improve this answer

Follow

edited Dec 9, 2019 at 13:12



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jun 7, 2012 at 6:19



guybrush

1,609 ● 12 ● 5

(stackoverflow.com/a/804178) Rebase into our own dev branch, then merge into master? "You can use rebase to first bring master "in" the new-feature branch: the rebase will

replay new-feature commits from the HEAD master, but still in the new-feature branch, effectively moving your branch starting point from an old master commit to HEAD-master. That allows you to resolve any conflicts in your branch (meaning, in isolation,). Then you can switch to master and merge new-feature (or rebase new-feature onto master if you want to preserve commits done in your new-feature branch)."

– [crazyTech](#) Aug 20, 2021 at 19:27



40



This answer is widely oriented around [Git Flow](#). The tables have been generated with the nice [ASCII Table Generator](#), and the history trees with this wonderful command ([aliased](#) as `git lg`):

```
git log --graph --abbrev-commit --decorate --date=format=format:'%C(bold blue)%h%C(reset) - %C(bold cyan green)(%ar)%C(reset)%C(bold yellow)%d%C(reset)%n' '%C(white)%s%C(reset) %C(dim white)- %an%C(reset)'
```

Tables are in reverse chronological order to be more consistent with the history trees. See also the difference between `git merge` and `git merge --no-ff` first (you usually want to use `git merge --no-ff` as it makes your history look closer to the reality):

`git merge`

Commands:

Time	Branch "develop"	Branch "fea"
15:04	git merge features/foo	
15:03		git commit -m "

```
15:02                                     git commit -m "  
15:01    git checkout -b features/foo  
15:00    git commit -m "First commit"
```

Result:

```
* 142a74a - YYYY-MM-DD 15:03:00 (XX minutes ago) (HEAD  
features/foo)  
|           Third commit - Christophe  
* 00d848c - YYYY-MM-DD 15:02:00 (XX minutes ago)  
|           Second commit - Christophe  
* 298e9c5 - YYYY-MM-DD 15:00:00 (XX minutes ago)  
           First commit - Christophe
```

git merge --no-ff

Commands:

Time	Branch "develop"	Branch "f
-----	-----	-----
15:04	git merge --no-ff features/foo	
15:03		git commit -m
15:02		git commit -m
15:01	git checkout -b features/foo	
15:00	git commit -m "First commit"	

Result:

```
* 1140d8c - YYYY-MM-DD 15:04:00 (XX minutes ago) (HE  
|\           Merge branch 'features/foo' - Christophe  
| * 69f4a7a - YYYY-MM-DD 15:03:00 (XX minutes ago) (fe  
| |           Third commit - Christophe  
| * 2973183 - YYYY-MM-DD 15:02:00 (XX minutes ago)  
|/           Second commit - Christophe  
* c173472 - YYYY-MM-DD 15:00:00 (XX minutes ago)  
           First commit - Christophe
```

`git merge` VS `git rebase`

First point: **always merge features into develop, never rebase develop from features**. This is a consequence of the [Golden Rule of Rebasing](#):

The golden rule of `git rebase` is to never use it on *public* branches.

[In other words](#):

Never rebase anything you've pushed somewhere.

I would personally add: *unless it's a feature branch AND you and your team are aware of the consequences*.

So the question of `git merge` VS `git rebase` applies almost only to the feature branches (in the following examples, `--no-ff` has always been used when merging). Note that since I'm not sure there's one better solution ([a debate exists](#)), I'll only provide how both commands behave. In my case, I prefer using `git rebase` as it produces a nicer history tree :)

Between feature branches

`git merge`

Commands:

```
Time                Branch "develop"                Branch "f
Branch "features/bar"
-----
-----
15:10    git merge --no-ff features/bar
15:09    git merge --no-ff features/foo
15:08
commit -m "Sixth commit"
15:07
merge --no-ff features/foo
15:06
commit -m "Fifth commit"
15:05
commit -m "Fourth commit"
15:04
git commit -m
15:03
git commit -m
15:02    git checkout -b features/bar
15:01    git checkout -b features/foo
15:00    git commit -m "First commit"
```

Result:

```
*    c0a3b89 - YYYY-MM-DD 15:10:00 (XX minutes ago) (HE
|\
      Merge branch 'features/bar' - Christophe
| * 37e933e - YYYY-MM-DD 15:08:00 (XX minutes ago) (fe
| |
      Sixth commit - Christophe
| *    eb5e657 - YYYY-MM-DD 15:07:00 (XX minutes ago)
| |\
      Merge branch 'features/foo' into featu
| * | 2e4086f - YYYY-MM-DD 15:06:00 (XX minutes ago)
| | |
      Fifth commit - Christophe
| * | 31e3a60 - YYYY-MM-DD 15:05:00 (XX minutes ago)
| | |
      Fourth commit - Christophe
* | |    98b439f - YYYY-MM-DD 15:09:00 (XX minutes ago)
|\ \ \
      Merge branch 'features/foo' - Christ
| | / /
| / | /
| | /
| * 6579c9c - YYYY-MM-DD 15:04:00 (XX minutes ago) (fe
| |
      Third commit - Christophe
```

```
| * 3f41d96 - YYYY-MM-DD 15:03:00 (XX minutes ago)
|/      Second commit - Christophe
* 14edc68 - YYYY-MM-DD 15:00:00 (XX minutes ago)
      First commit - Christophe
```

git rebase

Commands:

Time	Branch "develop"	Branch "f
	Branch "features/bar"	

15:10	git merge --no-ff features/bar	
15:09	git merge --no-ff features/foo	
15:08		
	commit -m "Sixth commit"	
15:07		
	rebase features/foo	
15:06		
	commit -m "Fifth commit"	
15:05		
	commit -m "Fourth commit"	
15:04		git commit -m
15:03		git commit -m
15:02	git checkout -b features/bar	
15:01	git checkout -b features/foo	
15:00	git commit -m "First commit"	

Result:

```
* 7a99663 - YYYY-MM-DD 15:10:00 (XX minutes ago) (HE
|\      Merge branch 'features/bar' - Christophe
| * 708347a - YYYY-MM-DD 15:08:00 (XX minutes ago) (fe
| |      Sixth commit - Christophe
| * 949ae73 - YYYY-MM-DD 15:06:00 (XX minutes ago)
| |      Fifth commit - Christophe
| * 108b4c7 - YYYY-MM-DD 15:05:00 (XX minutes ago)
| |      Fourth commit - Christophe
```

```

* |      189de99 - YYYY-MM-DD 15:09:00 (XX minutes ago)
| \ \      Merge branch 'features/foo' - Christop
| | /
| * 26835a0 - YYYY-MM-DD 15:04:00 (XX minutes ago) (fe
| |      Third commit - Christophe
| * a61dd08 - YYYY-MM-DD 15:03:00 (XX minutes ago)
| /      Second commit - Christophe
* ae6f5fc - YYYY-MM-DD 15:00:00 (XX minutes ago)
      First commit - Christophe

```

From `develop` to a feature branch

`git merge`

Commands:

Time	Branch "develop"	Branch "f
	Branch "features/bar"	

15:10	git merge --no-ff features/bar	
15:09	commit -m "Sixth commit"	
15:08	merge --no-ff develop	
15:07	git merge --no-ff features/foo	
15:06	commit -m "Fifth commit"	
15:05	commit -m "Fourth commit"	
15:04		git commit -m
15:03		git commit -m
15:02	git checkout -b features/bar	
15:01	git checkout -b features/foo	
15:00	git commit -m "First commit"	

Result:

```

*    9e6311a - YYYY-MM-DD 15:10:00 (XX minutes ago) (HE
|\      Merge branch 'features/bar' - Christophe
| * 3ce9128 - YYYY-MM-DD 15:09:00 (XX minutes ago) (fe
| |      Sixth commit - Christophe
| *    d0cd244 - YYYY-MM-DD 15:08:00 (XX minutes ago)
| |\      Merge branch 'develop' into features/b
| | /
| / |
* |    5bd5f70 - YYYY-MM-DD 15:07:00 (XX minutes ago)
|\ \      Merge branch 'features/foo' - Christop
| * | 4ef3853 - YYYY-MM-DD 15:04:00 (XX minutes ago) (
| | |      Third commit - Christophe
| * | 3227253 - YYYY-MM-DD 15:03:00 (XX minutes ago)
| / /      Second commit - Christophe
| * b5543a2 - YYYY-MM-DD 15:06:00 (XX minutes ago)
| |      Fifth commit - Christophe
| * 5e84b79 - YYYY-MM-DD 15:05:00 (XX minutes ago)
| /      Fourth commit - Christophe
* 2da6d8d - YYYY-MM-DD 15:00:00 (XX minutes ago)
      First commit - Christophe

```

git rebase

Commands:

Time	Branch "develop"	Branch "f
	Branch "features/bar"	

15:10	git merge --no-ff features/bar	
15:09		
	commit -m "Sixth commit"	
15:08		
	rebase develop	
15:07	git merge --no-ff features/foo	
15:06		
	commit -m "Fifth commit"	
15:05		
	commit -m "Fourth commit"	
15:04		git commit -m

```
15:03                                     git commit -m
15:02    git checkout -b features/bar
15:01    git checkout -b features/foo
15:00    git commit -m "First commit"
```

Result:

```
*    b0f6752 - YYYY-MM-DD 15:10:00 (XX minutes ago) (HE
|\
| * 621ad5b - YYYY-MM-DD 15:09:00 (XX minutes ago) (fe
| |
| |        Sixth commit - Christophe
| * 9cb1a16 - YYYY-MM-DD 15:06:00 (XX minutes ago)
| |
| |        Fifth commit - Christophe
| * b8ddd19 - YYYY-MM-DD 15:05:00 (XX minutes ago)
|/
|/        Fourth commit - Christophe
*    856433e - YYYY-MM-DD 15:07:00 (XX minutes ago)
|\
|\        Merge branch 'features/foo' - Christophe
| * 694ac81 - YYYY-MM-DD 15:04:00 (XX minutes ago) (fe
| |
| |        Third commit - Christophe
| * 5fd94d3 - YYYY-MM-DD 15:03:00 (XX minutes ago)
|/
|/        Second commit - Christophe
*    d01d589 - YYYY-MM-DD 15:00:00 (XX minutes ago)
        First commit - Christophe
```

Side notes

`git cherry-pick`

When you just need one specific commit, `git cherry-pick` is a nice solution (the `-x` option appends a line that says "*(cherry picked from commit...)*" to the original commit message body, so it's usually a good idea to use it - `git log <commit_sha1>` to see it):

Commands:

Time	Branch "develop"	Branch "f
	Branch "features/bar"	

15:10	git merge --no-ff features/bar	
15:09	git merge --no-ff features/foo	
15:08		
	commit -m "Sixth commit"	
15:07	cherry-pick -x <second_commit_sha1>	
15:06		
	commit -m "Fifth commit"	
15:05		
	commit -m "Fourth commit"	
15:04		git commit -m
15:03		git commit -m
15:02	git checkout -b features/bar	
15:01	git checkout -b features/foo	
15:00	git commit -m "First commit"	

Result:

```

*    50839cd - YYYY-MM-DD 15:10:00 (XX minutes ago) (HE
|\      Merge branch 'features/bar' - Christophe
| * 0cda99f - YYYY-MM-DD 15:08:00 (XX minutes ago) (fe
| |      Sixth commit - Christophe
| * f7d6c47 - YYYY-MM-DD 15:03:00 (XX minutes ago)
| |      Second commit - Christophe
| * dd7d05a - YYYY-MM-DD 15:06:00 (XX minutes ago)
| |      Fifth commit - Christophe
| * d0d759b - YYYY-MM-DD 15:05:00 (XX minutes ago)
| |      Fourth commit - Christophe
* |    1a397c5 - YYYY-MM-DD 15:09:00 (XX minutes ago)
|\ \      Merge branch 'features/foo' - Christop
| | /
| /|
| * 0600a72 - YYYY-MM-DD 15:04:00 (XX minutes ago) (fe
| |      Third commit - Christophe
| * f4c127a - YYYY-MM-DD 15:03:00 (XX minutes ago)
| /      Second commit - Christophe

```

* 0cf894c - YYYY-MM-DD 15:00:00 (XX minutes ago)
First commit - Christophe

`git pull --rebase`

I am not sure I can explain it better than [Derek Gourlay](#)...
Basically, use `git pull --rebase` instead of `git pull` :)
What's missing in the article though, is that [you can enable it by default](#):

```
git config --global pull.rebase true
```

`git rerere`

Again, nicely explained [here](#). But put simply, if you enable it, you won't have to resolve the same conflict multiple times anymore.

Share Improve this answer

edited May 6, 2020 at 10:19

Follow

answered Feb 10, 2017 at 17:26



sp00m

48.7k ● 31 ● 149 ● 258



This sentence gets it:

39

In general, the way to get the best of both worlds is to rebase local changes you've made, but



haven't shared yet, before you push them in order to clean up your story, but never rebase anything you've pushed somewhere.

Source: [3.6 Git Branching - Rebasing, Rebase vs. Merge](#)

Share Improve this answer

Follow

edited Dec 9, 2019 at 13:29



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jan 5, 2015 at 13:50



Joaquin Sargiotto

1,746 ● 13 ● 12



18



Git Merge vs Rebase

Both of them solve the same problem - concatenate branches. Rebase is more advanced technic which solves the main problem of Merge - spaghetti history. Project history becomes cleaner(linear and without extra merge commit) with Rebase. Bigger possibility generates bigger responsibility:

- Rebase Golden Rule - do **not** apply it on public/shared branches because commit's hash is changed(rewrite history). Actually it means that a good practice is work in a local branch(without pushing it to remote)
- Conflicts are solved iteratively

Merge workflow

1. create a new local branch - feature from origin/main
2. development
3. merge origin/main into local feature
4. resolve conflicts
5. push local feature into origin/feature
6. create merge request and merging(Create a merge commit)
7. remove local and remote feature branch

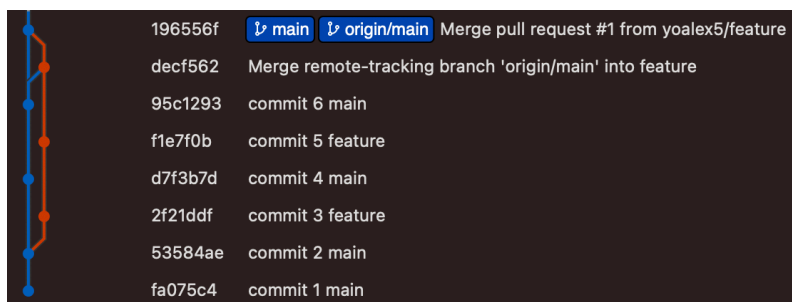
2. local feature branch:



4. resolve conflicts:

Changes from feature	Show Details	Result	Changes from main	Show Details
commit 5 feature	X >> 1	1	commit 6 main	1 << X

7. remove local and remote feature branch:

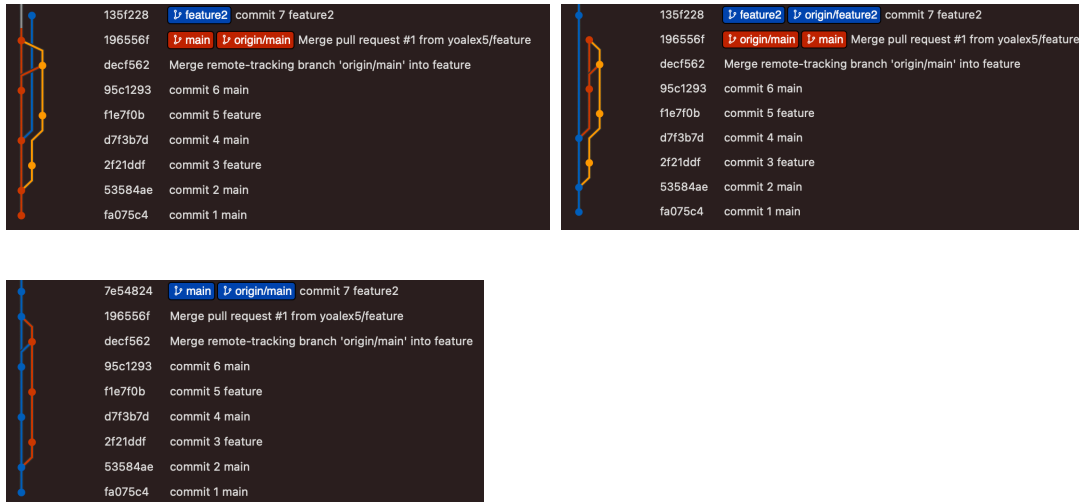


decf562 - 3. merge origin/main into local feature

196556f - 6. create merge request and merging

One more example where merge request creates a merge commit:

- rebase local feature2 branch
- you don't have conflicts. If you have some conflicts you can use merge request(Create a merge commit) instead
- create merge request and rebasing(Rebase and merge)



Rebase workflow

1. create a new local branch - feature from origin/main
2. development
3. rebase feature branch onto(on top of) origin/main branch. It means that you checkout feature and make rebase origin/main
4. resolve conflicts
5. push local feature into origin/feature
6. create merge request and rebasing(Rebase and merge)
7. remove local and remote feature branch

2. local feature branch:

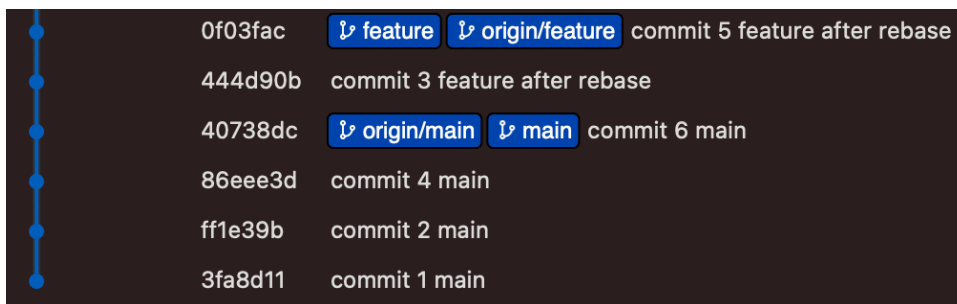


4. resolve conflicts:

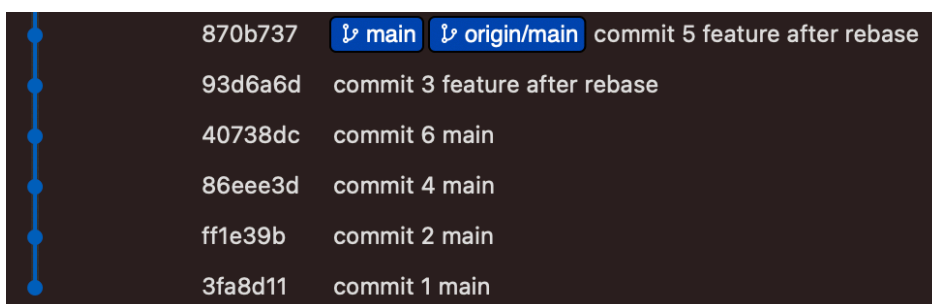
Rebasing d56bef22 from feature		Show Details	Result	Already rebased commits and commits from origin...		Show Details
commit 3 feature	X >> 1	1	commit 2 main	1 << X	commit 6 main	

Rebasing 18a2ac9a from feature		Show Details	Result	Already rebased commits and commits from origin...		Show Details
commit 5 feature	X >> 1	1	commit 3 feature	1 << X	commit 3 feature	
				2	commit 6 main	

5. push local feature into origin/feature

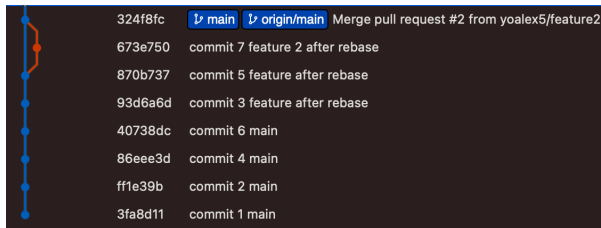
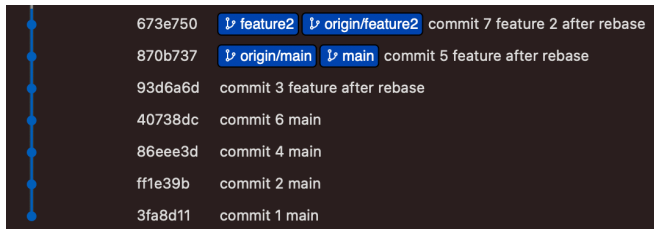
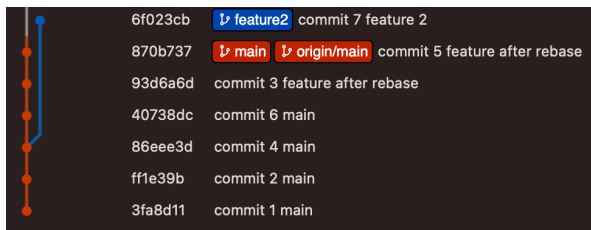


7. remove local and remote feature branch:



One more example where merge request creates a merge commit:

- rebase local feature2 branch
- resolve conflicts
- create merge request and merging(Create a merge commit)



Thanks

Share Improve this answer

edited Apr 25, 2023 at 11:35

Follow

answered Dec 10, 2019 at 12:39



yoAlex5

34k ● 10 ● 223 ● 239

- 1 IMO, this does not answer the question about **when** to use rebase or merge – Nico Haase Feb 4, 2021 at 15:43
- 2 In some cases, you want to rebase your feature branch to get the commits from master. so there is that to consider as well. – BluePie Apr 16, 2021 at 5:11 ✎
- 2 For large projects it is usefull to do rebase to avoid having other peoples commit be marked as yours when you do a Pull Request. – htafoya Jun 28, 2021 at 23:00

This doesn't answer the OP's question: you're just outputting your personal view over not using `rebase` . – [Lucaci Andrei](#)
Sep 6, 2021 at 10:22



Git rebase is used to make the branching paths in history cleaner and repository structure linear.

5



It is also used to keep the branches created by you private, as after rebasing and pushing the changes to the server, if you delete your branch, there will be no evidence of branch you have worked on. So your branch is now your local concern.



After doing rebase we also get rid of an extra commit which we used to see if we do a normal merge.

And yes, one still needs to do merge after a successful rebase as the rebase command just puts your work on top of the branch you mentioned during rebase, say master, and makes the first commit of your branch as a direct descendant of the master branch. This means we can now do a fast forward merge to bring changes from this branch to the master branch.

Share Improve this answer

Follow

edited Dec 9, 2019 at 13:21



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Sep 24, 2013 at 6:11



cvibha

713 ● 5 ● 9



Some practical examples, somewhat connected to large scale development where [Gerrit](#) is used for review and delivery integration:

5



I merge when I uplift my feature branch to a fresh remote master. This gives minimal uplift work and it's easy to follow the history of the feature development in for example [gitk](#).

```
git fetch
git checkout origin/my_feature
git merge origin/master
git commit
git push origin HEAD:refs/for/my_feature
```

I merge when I prepare a delivery commit.

```
git fetch
git checkout origin/master
git merge --squash origin/my_feature
git commit
git push origin HEAD:refs/for/master
```

I rebase when my delivery commit fails integration for whatever reason, and I need to update it towards a fresh remote master.

```
git fetch
git fetch <gerrit link>
git checkout FETCH_HEAD
git rebase origin/master
git push origin HEAD:refs/for/master
```

Share Improve this answer

Follow

edited Dec 9, 2019 at 13:36



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Feb 23, 2016 at 6:13



Martin G

18.1k ● 12 ● 89 ● 102



4



When do I use `git rebase`? Almost never, because it rewrites history. `git merge` is almost always the preferable choice, because it respects what actually happened in your project.

Share Improve this answer

answered Jan 19, 2018 at 21:01

Follow



Marnen Laibow-Koser

6,319 ● 1 ● 30 ● 34

3 @benjaminhull Thanks!—except I hope my answer is fact-based. IMHO opinion has little place in this sort of thing: it's a *fact* that losing your actual history makes life harder later.

– Marnen Laibow-Koser Jun 23, 2019 at 19:08

3 Agree. Merge will never lead to corrupted history etc. (when you rebase your pushed commits) – surfrider Aug 14, 2019 at 11:01

2 I don't know why this answer has lots of downvotes. I had to upvote to minimize the damage. LOL. I partially agree, but I think we can rebase if we're the only one working in the branch to keep everything cleaner. The main problem is if for some unexpected reason others start working on it as well.

– Maf Mar 30, 2021 at 8:31



3

Infographics always help :)



Merge : overlap one branch onto another one

Assume the following history exists and the current branch is "master":

```
      A---B---C topic
      /
D---E---F---G master
```

Then "git merge topic" will replay the changes made on the topic branch since it diverged from master (i.e., E) until its current commit (C) on top of master, and record the result in a new commit along with the names of the two parent commits and a log message from the user describing the changes.

```
      A---B---C topic
      /       \
D---E---F---G---H master
```

Rebase : Moving the change of one branch to the end of another one

Assume the following history exists and the current branch is "topic":

```
      A---B---C topic
      /
D---E---F---G master
```

From this point, the result of either of the following commands:

```
git rebase master
git rebase master topic
```

would be:

```
                A'--B'--C' topic
                /
            D---E---F---G master
```

NOTE: The latter form is just a short-hand of `git checkout topic` followed by `git rebase master`. When `rebase` exits `topic` will remain the checked-out branch.

So we can basically conclude that merging is a safe option that preserves the entire history of your repository, while rebasing creates a linear history by moving your feature branch onto the tip of main.

Credit: help page `git merge --help` and `git rebase --help`

Share Improve this answer

answered Feb 6, 2023 at 11:31

Follow



High Performance
Rangsiman

738 ● 1 ● 9 ● 16



1



Rebase is useful when you are working on a branch and have merged some other work in between - the merges will create changes which will make your diff polluted and thus harder to read.



If you rebase your branch, then your commits will be applied on top of the branch you are rebasing to, which makes it easier to review and the diff output is cleaner.

Share Improve this answer

Follow

answered Sep 21, 2022 at 5:25



[aleksander_si](#)

1,317 ● 1 ● 14 ● 32



1



- If **multiple developers** push to **1 branch** -- then I use `git merge --no-ff` to see after release branch merged, who wrote this line. It allows me to revert the whole feature with 1 commit.

- **Else** I use `git rebase` with squashing to have clear small git commits history. In case of conflicts, after resolving, I use `git push --force-with-lease`

Share Improve this answer

edited Feb 19 at 16:08

Follow

answered May 4, 2023 at 18:44



[Eugene Kaurov](#)

2,941 ● 32 ● 42



0



Let's say you create a new "new_feature" branch. Then, someone else merged their code to the `main` before you. so you want that updated `main` branch in your codebase because you do not want to diverge from the master branch for a super long time without getting those new changes. as you make more work on your branch, someone else or even more than 5 different engineers merged their code. then you need to get those commits onto your "new_feature" branch. and then you merge again. Imagine you had a feature that would take a week

but other engineers had small tasks so they keep merging their code. you might end up having dozens of merge commits that do not add anything to your code. finally, when you finish your task and merged your work onto the "main", you will have a bunch of non-informative merge commits as part of the history. this is what `rebase` solves.

If we rebased instead of merging, we rewrite the history, we are creating new commits based upon the original "new_feature" branch commits. as the name says we are setting up a new base for our "new_feature" branch. you will not have "merge commits" anymore. whatever branch you are working on will contain all of the commits from the master and from your feature. With "rebase" we get a much cleaner project. this makes it easier for someone to review your commits. In open source projects, there are thousands of contributors and maybe millions of work all the time, using rebase will make it easier to read the history of features.

Because `rebase` rewrite the commits, you do not want to rebase commits that other people already have. Imagine you pushed up some branch to Github and your coworkers have that work in their machine, so all the commits, if all of a sudden you rebase those commits, you will end up having some commits that they do not have or they will have some commits that you do not have.

you should rebase the commits that you have on your machine and other people do not have.

Share Improve this answer

answered Mar 5, 2023 at 22:06

Follow



Yilmaz

48.7k ● 18 ● 210 ● 264



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.