

Merge or Rebase?

By [Steve](#) on August 21, 2012

As you're no doubt aware, Git and Mercurial are great at re-integrating divergent lines of development through merging. They have to be, since their design strongly encourages developers to commit changes in parallel in their own distributed environments. Eventually some or all of these commits have to be brought together into a shared graph, and merging and rebasing are two primary ways that let us do that. So which one do you use?

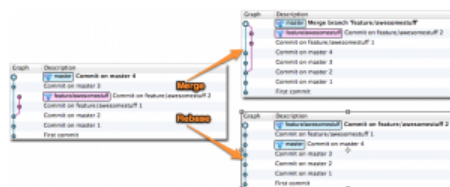
What does Merge or Rebase mean?

Let's start by defining what merging and rebasing are.

Merging brings two lines of development together while preserving the ancestry of each commit history.

In contrast, **rebasing** unifies the lines of development by re-writing changes from the source branch so that they appear as children of the destination branch – effectively pretending that those commits were written on top of the destination branch all along.

Here's a visual comparison between merging and rebasing a branch 'feature/awesomestuff' back to the master branch (click for full size):



So merging keeps the separate lines of development explicitly, while rebasing always ends up with a single linear path of development for both branches. But this rebase requires the commits on the source branch to be *re-written*, which changes their content and their SHAs. This has important ramifications which we'll talk about below.

[An aside: merging in Git can sometimes result in a special case: the 'fast forward merge'. This only applies if there are no commits in the destination branch which aren't already in the source branch. Fast-forward merges create no merge commit and the result looks like a rebase, because the commits just move over to the destination branch – except no history re-writing is needed (we'll talk about this re-writing in a second). You can turn fast-forward merges off in SourceTree so that a merge commit is always created if you want – check the 'Create a commit' option in the Merge dialog or set it globally in Preferences > Git.]

So, what are the pros and cons of merging and rebasing?

Pros and Cons

Merging Pros

- Simple to use and understand.
- Maintains the original context of the source branch.
- The commits on the source branch remain separate from other branch commits, provided you don't perform a fast-forward merge. This separation can be useful in the case of feature branches, where you might want to take a feature and merge it into another branch later.
- Existing commits on the source branch are unchanged and remain valid; it doesn't matter if they've been shared with others.

Merging Cons

- If the need to merge arises simply because multiple people are working on the same branch in



About SourceTree

SourceTree is a free Mac client for Git and Mercurial version control systems.

[Learn More »](#)

Download Free

Microsoft Windows 7+

Download Free

Mac OS X 10.6+



Follow @sourcetree



Subscribe to the Blog

Looking for Git hosting?

Meet [Bitbucket](#) – our free Git and Mercurial code hosting site with unlimited public and private repositories.

Recent Posts

[We're just getting started with SourceTree](#)

[Atlassian update for Git and Mercurial vulnerability](#)

[SourceTree for Mac 2.0 Released!](#)

[SourceTree for Windows 1.6 – Now Available!](#)

[Help us translate SourceTree for Mac!](#)

[View Blog Archives »](#)

parallel, the merges don't serve any useful historic purpose and create clutter.

Rebase Pros

- Simplifies your history.
- Is the most intuitive and clutter-free way to combine commits from multiple developers in a shared branch

Rebase Cons

- Slightly more complex, especially under conflict conditions. Each commit is rebased in order, and a conflict will interrupt the process of rebasing multiple commits. With a conflict, you have to resolve the conflict in order to continue the rebase. SourceTree guides you through this process, but it can still become a bit more complicated.
- Rewriting of history has ramifications if you've previously pushed those commits elsewhere. In Mercurial, you simply cannot push commits that you later intend to rebase, because anyone pulling from the remote will get them. In Git, you may push commits you may want to rebase later (as a backup) but *only* if it's to a remote branch that *only you use*. If anyone else checks out that branch and you later rebase it, it's going to get very confusing.

Practical tips

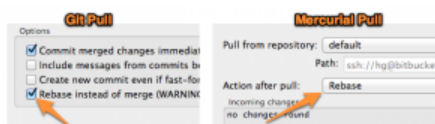
Looking at the above pros/cons, it's clear that it's generally not a case of choosing between one or the other, but more a case of using each at the appropriate times.

To explore this further, let's say you work in a development team with many committers, and that your team uses both shared branches as well as personal feature branches.

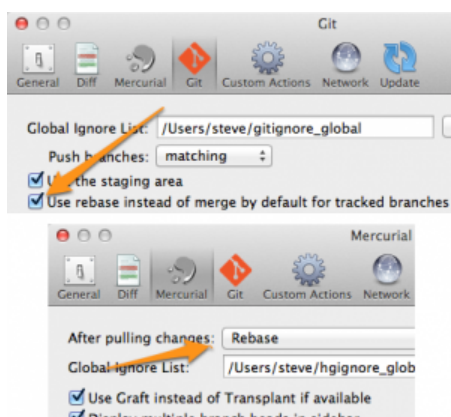
Shared branches

With shared branches, several people commit to the same branch, and they find that when pushing, they get an error indicating that someone else has pushed first. In this case, I would always recommend the 'Pull with rebase' approach. In other words, you'll be pulling down other people's changes and immediately rebasing your commits on top of these latest changes, allowing you to push the combined result back as a linear history. It's important to note that your commits must not have been shared with others yet.

In SourceTree, you can do this in the Pull dialog:



You can also set this as the default behavior in your Preferences if you like:



By taking this approach, when developing in parallel on a shared branch, you and your colleagues can still create a linear history, which is much simpler to read than if each member merges whenever some commits are built in parallel.

The only time you shouldn't rebase and should merge instead is if you've shared your outstanding commits already with someone else via another mechanism, e.g. you've pushed them to another public repository, or submitted a patch or pull request somewhere.

Feature branches

Now let's take the case where you deliberately create a separate branch for a feature you're developing, and for the sake of this example, you are the only person working on that feature

branch. This approach is common with [git-flow](#) and [hg-flow](#) for example. This feature branch may take a while to complete, and you'll only want to re-integrate it into other lines of development once you're done. So how do you manage that? There are actually two separate issues here that we must address.

The final merge: When building a feature on a separate branch, you're usually going to want to keep these commits together in order to illustrate that they are part of a cohesive line of development. Retaining this context allows you to identify the feature development easily, and potentially use it as a unit later, such as merging it again into a different branch, submitting it as a pull request to a different repository, and so on. Therefore, you're going to want to merge rather than rebase when you complete your final re-integration, since merging gives you a single defined integration point for that feature branch and allows easy identification of the commits that it comprised.

Keeping the feature branch up to date: While you're developing your feature branch, you may want to periodically keep it in sync with the branch which it will eventually be merged back into. For example, you may want to test that your new feature remains compatible with the evolving codebase well before you perform that final merge. There are two ways you can bring your feature branch up to date:

1. Periodically merge from the (future) destination branch into your feature branch. This approach used to cause headaches in old systems like Subversion, but actually works fine in Git and Mercurial.
2. Periodically rebase your feature branch onto the current state of the destination branch

The pros and cons of each are generally similar to those for merging and rebasing. Rebasing keeps things tidier, making your feature branch appear quite compact. If you use the merge approach instead, it means that your feature branch will always branch off from its original base commit, which might have happened quite a long time ago. If your entire team did this and there's a lot of activity, your commit history would contain a lot of parallel feature branches over a long period of time. Rebasing continually compacts each feature branch into a smaller space by moving its base commit to more recent history, cleaning up your commit graph.

The downside of rebasing your feature branches in order to keep them up to date is that this approach rewrites history. If you never push these branches outside your development machine, this is no problem. But assuming that you do want to push them somewhere, say for backup or just visibility, then rebasing can cause issues. On Mercurial, it's always a bad idea – you should never push branches you intend to rebase later. With git, you have a bit more flexibility. For example, you could push your feature branches to a different remote to keep them separate from the rest, or you could push your feature branches to your usual remote, as your development team is aware that these feature branches will likely be rewritten so they should not check them out from this remote.

Conclusion

The consensus that I come across most frequently is that both merge and rebase are worth using. The time to use either is entirely dependent on the situation, the experience of your team, and the specific DVCS you're using.

1. When multiple developers work on a shared branch, pull & rebase your outgoing commits to keep history cleaner (Git and Mercurial)
2. To re-integrate a completed feature branch, use merge (and opt-out of fast-forward commits in Git)
3. To bring a feature branch up to date with its base branch:
 1. Prefer rebasing your feature branch onto the latest base branch if:
 2. You haven't pushed this branch anywhere yet, or
3. You're using Git, and you know for sure that other people will not have checked out your feature branch
4. Otherwise, merge the latest base changes into your feature branch

I hope this helps! Please let me know in the comments if you have any questions or suggestions.

33 Comments

SourceTree Blog

 huyilong ▾

 Recommend 15

 Share

Sort by Best ▾



Join the discussion...

**Stephen Ball** · 2 years ago

Rather than decide between merge bubble and straight line of history, you can also use rebase to allow for clean, non-fast forward feature branch merges.

1. rebase the feature branch against the destination
2. use ``merge --no-ff feature_branch`` to pull it in to the destination branch

This allows for a clear merge commit of a feature (which means it's easily revertable as well as exceedingly easy to see in a graph) with no back and forth merge bubbling.

11 ^ | v · Reply · Share

**Marnen Laibow-Koser** → Stephen Ball · a year ago

What's wrong with merge bubbling? Rebasing means your history lies, since it claims you branched off master where you didn't. See <http://paul.stadig.name/2010/1...> for why this is usually a bad idea. (I do occasionally rebase the feature branch, but I more often merge it. And I *always* merge it if I've pushed the feature branch to a remote—which I usually have—because force-pushing to a remote is *far* worse than merge bubbling.)

In general, I think rebasing more than once in a blue moon is the sign of a broken Git workflow, because it rewrites history too much.

I notice that as a Git novice, I used to rebase a lot and hardly ever merge. As an expert Git user today, I merge a lot and hardly ever rebase. I think that's significant.

1 ^ | v · Reply · Share

**Stephen Ball** → Marnen Laibow-Koser · a year ago

Nothing is inherently "wrong" with merge bubbling: it all depends on your team and how you're using your Git history.

For us we value seeing a linear feature branch history over preserving what work was being done in parallel. With 8 devs even trivially parallel work quickly makes the log graph unusable and reduces the effectiveness of git bisect for getting context around a commit.

We use rebase in two ways:

1. present our development as a cohesive story (i.e. refactor 20, 30 commits down to a few that explain the actual feature development instead of the false paths).
2. Align feature branches with the current master/HEAD so that the merge comes in as a linear history.

Basically, how Linus describes in the classic "how to rebase" post: <http://www.mail-archive.com/dr...>

I'm curious: what issues did you have rewriting history "too much"? I agree that rebase (or squashing) can taken to an unpleasant extreme such as when a large feature is compressed down into a single poorly written commit. But rebase only changes HEAD and then plays back the commits: nothing is lost until you tell it to get lost.

1 ^ | v · Reply · Share

**Marnen Laibow-Koser** → Stephen Ball · a year ago

> For us we value seeing a linear feature branch history over preserving what work was being done in parallel.

Ah, I see. In my experience, a desire to see a linear branch history is generally a sign of not using Git *as Git*. Of course many of us (myself included) grew up on tools like Subversion that encourage a linear history.

But Git really is different—it encourages a non-linear history, and IMHO it's at its best when that is respected, because it's easier for Git's amazing history analysis tools to work if the history is not prematurely linearized.

> With 8 devs even trivially parallel work quickly makes the log graph unusable

How are you trying to use it? I've worked on larger teams than that without rebase and had no problem at all.

see more

2 ^ | v · Reply · Share ›



Stephen Ball → Marnen Laibow-Koser · a year ago

Interesting. I'm 80% sure you're trolling, shame on me. It's like you didn't even read my post or Linus's post. I'll hit on your points and share some more links though. :-)

You take a very strange view to only use a subset of the full Git. Git is the only system I've used into that can keep a linear history in such an elegant manner. SVN? Please.

Ah, when I say "git log" I mean the DAS version of git log --graph. The pretty git log gets exceedingly noisy with a parallel history the vertical bars pushing out all the other data.

For Git bisect I mean that using it with a linear/feature history you can easily see the feature context surrounding the commit.

All respect to Paul Stadig, but that's a very shortsighted view of Git's potential. Rewriting history is nothing to be scared of. Commits like "Fixed typo" are literally noise in your signal and

see more

1 ^ | v · Reply · Share ›



Marnen Laibow-Koser → Stephen Ball · a year ago

> Interesting. I'm 80% sure you're trolling, shame on me.

Nope. Sorry if I gave that impression. I don't pull my punches when discussing things like this, but I'm trying to approach this as a real discussion.

> It's like you didn't even read my post or Linus's post. I'll hit on your points and share some more links though. :-)

I had not read Linus's post, but I have now; thanks for the reference. I do use rebase occasionally on my private branches as he suggests, but that's about it.

And of course I read your comment before replying. If there's something you think I'm not getting, I'd love to know what it is.

> You take a very strange view to only use a subset of the full Git.

see more

^ | v · Reply · Share ›



Oliver Zhou · 3 years ago

Rebase is now much safer than before thanks to hg phases support

4 ^ | v · Reply · Share ›



stevestreeting → Oliver Zhou · 3 years ago

Indeed - Mercurial will actually stop you rewriting changes (including via Rebase) that have already been pushed if you use 2 2 or above

rebase, that have already been pushed if you use 2.2 or above.

^ | v · Reply · Share ›



Design by Adrian · 2 years ago

I find the 'Rebase current changes onto [otherbranch]' text to be scary. I don't want to affect 'otherbranch', only my current branch. Is it possible to change the message to something in the lines with 'Rebase and use [otherbranch] as source', or something...

2 ^ | v · Reply · Share ›



stevestreting → Design by Adrian · 2 years ago

Sorry, I think the current text is clearer. With rebase it's very important to understand which commits will be changing, and the text makes clear that it's your current commits that will move & be placed on top of the other branch you selected - ie the other branch isn't modified, it's just what the commits move on to.

1 ^ | v · Reply · Share ›



MikeSchinkel → stevestreting · 2 years ago

I agree with Adrian. That wording does not give me enough information to ever be willing to risk using it. I'll sadly do from the command line instead.

Maybe what would help is if there were some way to preview the Git command that will be run for all menu options?

^ | v · Reply · Share ›



stevestreting → MikeSchinkel · 2 years ago

Yes, this is the conclusion we reached:

<https://jira.atlassian.com/bro...>

^ | v · Reply · Share ›



Jerry → stevestreting · 2 years ago

Totally agree. When I right-click my feature branch (which is checked out), the menu gives me the choice to "Rebase current changes onto <my feature="" branch="" name="">". So that doesn't tell me what's going to be changed. You're assuming people already know what rebasing does, and that this terminology makes it clear in which direction it's happening. I originally branched off of develop, not master. When they say "current changes", do they mean in develop, or master? Does the system automatically know which branch to pull from to do the rebase? Vague, non-descript menus like these can cause people to mess up entire repos, and I know, because I've seen it happen.

^ | v · Reply · Share ›



stevestreting → Jerry · 2 years ago

I think a preview of the graph would be the best way to address this (a bit like we do with git-flow) for people who are not familiar with the rebasing action.

<https://jira.atlassian.com/bro...>

^ | v · Reply · Share ›



robrecord → stevestreting · 2 years ago

that isn't very clear to be honest. I am with Adrian.

^ | v · Reply · Share ›



Dominick · 3 years ago

very clear and informative post; I feel like I understand these two approaches much better now. thanks Steve!

2 ^ | v · Reply · Share ›



Abbie Kressner · 3 years ago

Great blog post!

2 ^ | v · Reply · Share ›

**Eugene Dubinin** · 2 years ago

Cool! Thank you very much. I was in doubt before reading this.

1 ^ | v · Reply · Share ›

**Phill Sparks** · 3 years ago

I've recently started using git smart-pull. It's a ruby script that detects the best way to update the local branch (rebase or merge). It also stashes local changes. Maybe it's something you can consider for SourceTree?

<http://github-displayer.herokuapp.com/>

1 ^ | v · Reply · Share ›

**Marnen Laibow-Koser** → **Phill Sparks** · a year ago

That would just make it easier to rebase, which is usually a bad thing. Merge is to be preferred in nearly all cases, since it doesn't rewrite history.

^ | v · Reply · Share ›

**ybart** · 3 years ago

Would be create to have the option to show the original commit date instead of the rebased commit date.

1 ^ | v · Reply · Share ›

**stevestreeting** → **ybart** · 3 years ago

It's already there in 1.5.3 - in Preferences > Git, check 'Display author date instead of commit date in log'. Both dates are also displayed in the commit details when you select the line.

^ | v · Reply · Share ›

**ybart** → **stevestreeting** · 3 years ago

Didn't saw that ! Thanks for the tip. I noticed the information was available in commit details panel, but it's definitely simpler to have the info directly in the list.

^ | v · Reply · Share ›

**Ritesh Kale** · 11 days ago

I am facing an issue with SourceTree (on Windows) merge. In the Log/History tab, I am able to see the two parents that were used in the merge. But the Log/History tab shows only the files that were affected by one parent. How can I see the files that were changed by the second parent in the merge? Am I missing something here?

^ | v · Reply · Share ›

**stevestreeting** → **Ritesh Kale** · 6 days ago

On merge commits there's a couple of extra options at the bottom of the 'gear' menu at the top right of the diff; at the bottom you'll see 'Diff vs parent' and 'Dif vs merged'.

^ | v · Reply · Share ›

**katopz** · 2 months ago

What to do when I finished feature(merged to develop) but need to revisit that feature and/or hotfix?

- A. create new feature with same name
- B. create new feature with same name + "_revisit"
- C. create new feature with new name
- D. create new hotfix for that feature
- E. checkout old feature branch and continue from that point

^ | v · Reply · Share ›

**stevestreeting** → **katopz** · 2 months ago

My personal view on this is to create a new feature branch. For people who name feature branches after tracker issues this comes more naturally, but if you don't do this creating a new feature branch with a 'v2' suffix or something can work. I wouldn't use '_revisit' in case you have to come back to it again ;)

The trouble with trying to re-use the old feature branch is that you then have

to merge in all the work that has been done on the develop branch back into it, and it's just less clear on the merge graph afterwards what's happening. If it's very soon after merging the first version it's not so bad, so you could go with that if you wanted, but on a general case where this might be some time later, a new feature branch is cleaner.

^ | v · Reply · Share ›



bpatters7 · a year ago

Excellent article. One of the best posts I've seen on the 'always more to learn/consider' topic of merge versus rebase.

^ | v · Reply · Share ›



robrecord · 2 years ago

Where can I read about how to deal with rebase conflicts? I have no idea what SourceTree is showing me when this happens, which file is from where, or what stage I am at. It's extremely confusing.

^ | v · Reply · Share ›



stevestreeting → robrecord · 2 years ago

Rebase conflicts are essentially the same as merge conflicts, except that your changes are 'replayed' on top of the target branch, one at a time, meaning the rebase has to stop as soon as there's a conflict. You have to resolve the conflicts the same way as you do with a merge (e.g. launching external merge tools) then continue the rebase process, since there may be more commits to 'replay' after this one. Rebase is a multi-stage process rather than a single stage like merge because each commit has to be modified as it is rebased. SourceTree automatically prompts you to continue the rebase process if you click Commit or other commit functions while you have a rebase in progress so you don't have to remember to use the 'Continue Rebase' feature explicitly once you've resolved the conflicts.

^ | v · Reply · Share ›



Magbic → stevestreeting · a year ago

I can't seem to get this working on SourceTree 1.9.0, I created a merge conflict scenario for testing Pull with Rebase instead of Merge from Develop to my Feature Branch.

SourceTree creates "(no branch / rebasing ...)", okay so when I resolve my conflict, I commit it, then "HEAD" Appears with my resolved file, but if I switch to my Feature Branch it doesn't appear to be rebased, when I get prompted to Continue, this seems to do thing and I find myself back to square 1.

What am I not doing right?

*** FIGURED IT OUT ***

When you're done with your conflict(s), click on the Action menu item, in your SourceTree toolbar at the OS bar at top (Mac user).

Then click Rebase Continue, and you should be Rebased.

^ | v · Reply · Share ›



Erik van der Neut · 2 years ago

Thanks Steve! This is a wonderfully clear explanation that gives me extra confidence in how to use GIT/SourceTree.

^ | v · Reply · Share ›



saurav · 2 years ago

Thanks , it cleared my doubts.


Cheers

<http://lotusmediacentre.com/di...>

^ | v · Reply · Share ›


Introducing SourceTree for Windows
a free desktop client for Git

58 comments • 2 years ago

 **Chris Post** — Are those tabs for multiple repositories in one window that I see there?! When will all of us ...


SourceTree for Mac 1.6.0b2 Now Available

2 comments • 2 years ago

 **stevestreeting** — Excellent point, edited - thanks!


SourceTree for Mac 1.6.0b2

11 comments • 2 years ago

 **wpostma** — Can I just say WOW! Thank you guys. This product is amazing. I use version 1.5.8 every day and it is ...

New installer / updater for SourceTree for Windows

28 comments • 2 years ago

 **Craig Fox** — I try to install SourceTreeSetup_1.5.2.exe on windows 7 and it keeps hanging ...

 [Subscribe](#)

 [Add Disqus to your site](#)

 [Privacy](#)

DISQUS

« [Smart branching with SourceTree and Git-flow](#)

[In Dublin this weekend? Come to our drinkup!](#) »

[Download](#) · [Blog](#) · [Support](#) · [Terms of Service](#) · [Privacy Policy](#)

