new blog post: git rebase: what can go wrong?

1 message

Julia Evans <julia@wizardzines.com> Fri, Nov 10, 2023 at 10:57 PM
To: mperdikeas@gmail.com

one blog post this week, about git again:

git rebase: what can go wrong?

Hello! While talking with folks about Git, I've been seeing a comment over and over to the effect of "I hate rebase". People seemed to feel pretty strongly about this, and I was really surprised because I don't run into a lot of problems with rebase and I use it all the time.

I've found that if many people have a very strong opinion that's different from mine, usually it's because they have different experiences around that thing from me.

So I asked on Mastodon:

> today I'm thinking about the tradeoffs of using `git rebase` a bit. I think the goal of rebase is to have a nice linear commit history, which is something I like.
> but what are the *costs* of using rebase? what problems has it caused for you in practice? I'm really only interested in specific bad experiences you've had here – not opinions or general statements like "rewriting history is bad"

I got a huge number of incredible answers to this, and I'm going to do my best to summarize them here. I'll also mention solutions or workarounds to those problems in cases where I know of a solution. Here's the list:

- fixing the same conflict repeatedly is annoying
- rebasing a lot of commits is hard
- undoing a rebase is hard
- force pushing to shared branches can cause lost work
- force pushing makes code reviews harder
- losing commit metadata
- more difficult reverts
- rebasing can break intermediate commits
- accidentally run git commit –amend instead of git rebase –continue
- splitting commits in an interactive rebase is hard
- complex rebases are hard
- rebasing long lived branches can be annoying
- rebase and commit discipline
- a "squash and merge" workflow
- miscellaneous problems

My goal with this isn't to convince anyone that rebase is bad and you shouldn't use it (I'm certainly going to keep using rebase!). But seeing all these problems made me want to be more cautious about recommending rebase to newcomers without explaining how to use it safely. It also makes me wonder if there's an easier workflow for cleaning up your commit history that's harder to accidentally mess up.

my git workflow assumptions

First, I know that people use a lot of different Git workflows. I'm going to be talking about the workflow I'm used to when working on a team, which is:

- the team uses a central Github/Gitlab repo to coordinate
- there's one central `main` branch. It's protected from force pushes.
- people write code in feature branches and make pull requests to `main`
- The web service is deployed from `main` every time a pull request is merged.
- the only way to make a change to `main` is by making a pull request on Github/Gitlab and merging it

This is not the only “correct” git workflow (it’s a very “we run a web service” workflow and open source project or desktop software with releases generally use a slightly different workflow). But it’s what I know so that’s what I’ll talk about.

two kinds of rebase

Also before we start: one big thing I noticed is that there were 2 different kinds of rebase that kept coming up, and only one of them requires you to deal with merge conflicts.

1. **rebasing on an ancestor**, like `git rebase -i HEAD^^^^^^^` to squash many small commits into one. As long as you’re just squashing commits, you’ll never have to resolve a merge conflict while doing this.
2. **rebasing onto a branch that has diverged**, like `git rebase main`. This can cause merge conflicts.

I think it’s useful to make this distinction because sometimes I’m thinking about rebase type 1 (which is a lot less likely to cause problems), but people who are struggling with it are thinking about rebase type 2.

Now let’s move on to all the problems!

fixing the same conflict repeatedly is annoying

If you make many tiny commits, sometimes you end up in a hellish loop where you have to fix the same merge conflict 10 times. You can also end up fixing merge conflicts totally unnecessarily (like dealing with a merge conflict in code that a future commit deletes).

There are a few ways to make this better:

- first do a `git rebase -i HEAD^^^^^^^^^^` to squash all of the tiny commits into 1 big commit and then a `git rebase main` to rebase onto a different branch. Then you only have to fix the conflicts once.
- use `git rerere` to automate repeatedly resolving the same merge conflicts (“rerere” stands for “reuse recorded resolution”, it’ll record your previous merge conflict resolutions and replay them). I’ve never tried this but I think you need to set `git config rerere.enabled true` and then it’ll automatically help you.

Also if I find myself resolving merge conflicts more than once in a rebase, I’ll usually run `git rebase --abort` to stop it and then squash my commits into one and try again.

rebasing a lot of commits is hard

Generally when I’m doing a rebase onto a different branch, I’m rebasing 1-2 commits. Maybe sometimes 5! Usually there are no conflicts and it works fine.

Some people described rebasing hundreds of commits by many different people onto a different branch. That sounds really difficult and I don’t envy that task.

undoing a rebase is hard

I heard from several people that when they were new to rebase, they messed up a rebase and permanently lost a week of work that they then had to redo.

The problem here is that undoing a rebase that went wrong is **much** more complicated than undoing a merge that went wrong (you can undo a bad merge with something like `git reset --hard HEAD^`). Many newcomers to rebase don’t even realize that undoing a rebase is even possible, and I think it’s pretty easy to understand why.

That said, it is possible to undo a rebase that went wrong. Here’s an example of how to undo a rebase using `git reflog`.

step 1: Do a bad rebase (for example run `git rebase -I HEAD^^^^^` and just delete 3 commits)

step 2: Run `git reflog`. You should see something like this:

```
<code>ee244c4 (HEAD -> main) HEAD@{0}: rebase (finish): returning to
refs/heads/main
ee244c4 (HEAD -> main) HEAD@{1}: rebase (pick): test
fdb8d73 HEAD@{2}: rebase (start): checkout HEAD^^^^^^^
ca7fe25 HEAD@{3}: commit: 16 bits by default
073bc72 HEAD@{4}: commit: only show tooltips on desktop
</code>
```

step 3: Find the entry immediately before `rebase (start)`. In my case that's `ca7fe25`

step 4: Run `git reset --hard ca7fe25`

A couple of other ways to undo a rebase:

- Apparently `@` always refers to your current branch in git, so you can run `git reset --hard @{1}` to reset your branch to its previous location.
- Another solution folks mentioned that avoids having to use the reflog is to make a "backup branch" with `git switch -c backup` before rebasing, so you can easily get back to the old commit.

force pushing to shared branches can cause lost work

A few people mentioned the following situation:

1. You're collaborating on a branch with someone
2. You push some changes
3. They rebase the branch and run `git push --force` (maybe by accident)
4. Now when you run `git pull`, it's a mess – you get the a `fatal: Need to specify how to reconcile divergent branches` error
5. While trying to deal with the fallout you might lose some commits, especially if some of the people are involved aren't very comfortable with git

This is an even worse situation than the "undoing a rebase is hard" situation because the missing commits might be split across many different people's and the only worse thing than having to hunt through the reflog is multiple different people having to hunt through the reflog.

This has never happened to me because the only branch I've ever collaborated on is `main`, and `main` has always been protected from force pushing (in my experience the only way you can get something into `main` is through a pull request). So I've never even really been in a situation where this *could* happen. But I can definitely see how this would cause problems.

The main tools I know to avoid this are:

- don't rebase on shared branches
- use `--force-with-lease` when force pushing, to make sure that nobody else has pushed to the branch since your last fetch

Apparently the "since your last **fetch**" is important here – if you run `git fetch` immediately before running `git push --force-with-lease`, the `--force-with-lease` won't protect you at all.

I was curious about why people would run `git push --force` on a shared branch. Some reasons people gave were:

- they're working on a collaborative feature branch, and the feature branch needs to be rebased onto `main`. The idea here is that you're just really careful about coordinating the rebase so nothing gets lost.
- as an open source maintainer, sometimes they need to rebase a contributor's branch to fix a merge conflict
- they're new to git, read some instructions online that suggested `git rebase` and `git push --force` as a solution, and followed them without understanding the consequences
- they're used to doing `git push --force` on a personal branch and ran it on a shared branch by accident

force pushing makes code reviews harder

The situation here is:

- You make a pull request on GitHub
- People leave some comments
- You update the code to address the comments, rebase to clean up your commits, and force push
- Now when the reviewer comes back, it's hard for them to tell what you changed since the last time you saw it – all the commits show up as "new".

One way to avoid this is to push new commits addressing the review comments, and then after the PR is approved do a rebase to reorganize everything.

I think some reviewers are more annoyed by this problem than others, it's kind of a personal preference. Also this might be a Github-specific issue, other code review tools might have better tools for managing this.

losing commit metadata

If you're rebasing to squash commits, you can lose important commit metadata like `Co-Authored-By`. Also if you GPG sign your commits, rebase loses the signatures.

There's probably other commit metadata that you can lose that I'm not thinking of.

I haven't run into this one so I'm not sure how to avoid it. I think GPG signing commits isn't as popular as it used to be.

more difficult reverts

Someone mentioned that it's important for them to be able to easily revert merging any branch (in case the branch broke something), and if the branch contains multiple commits and was merged with rebase, then you need to do multiple reverts to undo the commits.

In a merge workflow, I think you can revert merging any branch just by reverting the merge commit.

rebasing can break intermediate commits

If you're trying to have a very clean commit history where the tests pass on every commit (very admirable!), rebasing can result in some intermediate commits that are broken and don't pass the tests, even if the final commit passes the tests.

Apparently you can avoid this by using `git rebase -x` to run the test suite at every step of the rebase and make sure that the tests are still passing. I've never done that though.

accidentally run `git commit --amend` instead of `git rebase --continue`

A couple of people mentioned issues with running `git commit --amend` instead of `git rebase --continue` when resolving a merge conflict.

The reason this is confusing is that there are two reasons when you might want to edit files during a rebase:

1. editing a commit (by using `edit` in `git rebase -i`), where you need to write `git commit --amend` when you're done
2. a merge conflict, where you need to run `git rebase --continue` when you're done

It's very easy to get these two cases mixed up because they feel very similar. I think what goes wrong here is that you:

- Start a rebase
- Run into a merge conflict
- Resolve the merge conflict, and run `git add file.txt`
- Run `git commit` because that's what you're used to doing after you run `git add`
- But you were supposed to run `git rebase --continue`! Now you have a weird extra commit, and maybe it has the wrong commit message and/or author

splitting commits in an interactive rebase is hard

The whole point of rebase is to clean up your commit history, and **combining** commits with rebase is pretty easy. But what if you want to split up a commit into 2 smaller commits? It's not as easy, especially if the commit you want to split is a few commits back! I actually don't really know how to do it even though I feel very comfortable with rebase. I'd probably just do `git reset HEAD^^^` or something and use `git add -p` to redo all my commits from scratch.

One person shared their workflow for splitting commits with rebase.

complex rebases are hard

If you try to do too many things in a single `git rebase -i` (reorder commits AND combine commits AND modify a commit), it can get really confusing.

To avoid this, I personally prefer to only do 1 thing per rebase, and if I want to do 2 different things I'll do 2 rebases.

rebasing long lived branches can be annoying

If your branch is long-lived (like for 1 month), having to rebase repeatedly gets painful. It might be easier to just do 1 merge at the end and only resolve the conflicts once.

The dream is to avoid this problem by not having long-lived branches but it doesn't always work out that way in practice.

miscellaneous problems

A few more issues that I think are not that common:

- **Stopping a rebase wrong**: If you try to abort a rebase that's going badly with `git reset --hard` instead of `git rebase --abort`, things will behave weirdly until you stop it properly
- **Weird interactions with merge commits**: A couple of quotes about this: "If you rebase your working copy to keep a clean history for a branch, but the underlying project uses merges, the result can be ugly. If you do rebase -i HEAD~4 and the fourth commit back is a merge, you can see dozens of commits in the interactive editor.", "I've learned the hard way to *never* rebase if I've merged anything from another branch"

rebase and commit discipline

I've seen a lot of people arguing about rebase. I've been thinking about why this is and I've noticed that people work at a few different levels of "commit discipline":

1. Literally anything goes, "wip", "fix", "idk", "add thing"
2. When you make a pull request (on github/gitlab), squash all of your crappy commits into a single commit with a reasonable message (usually the PR title)
3. Atomic Beautiful Commits – every change is split into the appropriate number of commits, where each one has a nice commit message and where they all tell a story around the change you're making

Often I think different people inside the same company have different levels of commit discipline, and I've seen people argue about this a lot. Personally I'm mostly a Level 2 person. I think Level 3 might be what people mean when they say "clean commit history".

I think Level 1 and Level 2 are pretty easy to achieve without rebase – for level 1, you don't have to do anything, and for level 2, you can either press "squash and merge" in github or run `git switch main; git merge --squash mybranch` on the command line.

But for Level 3, you either need rebase or some other tool (like GitUp) to help you organize your commits to tell a nice story.

I've been wondering if when people argue about whether people "should" use rebase or not, they're really arguing about which minimum level of commit discipline should be required.

I think how this plays out also depends on how big the changes folks are making – if folks are usually making pretty small pull requests anyway, squashing them into 1 commit isn't a big deal, but if you're making a 6000-line change you probably want to split it up into multiple commits.

a "squash and merge" workflow

A couple of people mentioned using this workflow that doesn't use rebase:

- make commits
- Run `git merge main` to merge main into the branch periodically (and fix conflicts if necessary)
- When you're done, use GitHub's "squash and merge" feature (which is the equivalent of running `git checkout main; git merge --squash mybranch`) to squash all of the changes into 1 commit. This gets rid of all the "ugly" merge commits.

I originally thought this would make the log of commits on my branch too ugly, but apparently `git log main..mybranch` will just show you the changes on your branch, like this:

```
<code>$ git log main..mybranch
756d4af (HEAD -> mybranch) Merge branch 'main' into mybranch
20106fd Merge branch 'main' into mybranch
d7da423 some commit on my branch
85a5d7d some other commit on my branch
</code>
```

Of course, the goal here isn't to **force** people who have made beautiful atomic commits to squash their commits – it's just to provide an easy option for folks to clean up a messy commit history ("add new feature; wip; wip; fix; fix; fix; fix; fix;") without having to use rebase.

I'd be curious to hear about other people who use a workflow like this and if it works well.

there are more problems than I expected

I went into this really feeling like "rebase is fine, what could go wrong?" But many of these problems actually have happened to me in the past, it's just that over the years I've learned how to avoid or fix all of them.

And I've never really seen anyone share best practices for rebase, other than "never force push to a shared branch". All of these honestly make me a lot more reluctant to recommend using rebase.

To recap, I think these are my personal rebase rules I follow:

- stop a rebase if it's going badly instead of letting it finish (with `git rebase --abort`)
- know how to use `git reflog` to undo a bad rebase
- don't rebase a million tiny commits (instead do it in 2 steps: `git rebase -i HEAD^^^^` and then `git rebase main`)
- don't do more than one thing in a `git rebase -i`. Keep it simple.
- never force push to a shared branch
- never rebase commits that have already been pushed to `main`

Thanks to Marco Rogers for encouraging me to think about the problems people have with rebase, and to everyone on Mastodon who helped with this.