When does a merge conflict happen?

What is merge conflict

Merge conflict arises when you have two people changing the same line in a file OR if one person deleted the file but the other modified (effectively keeping that file), **then** you tried to integrate the changes together via either `git merge` or `git rebase`. Git cannot determine which one of the changes is correct.

The merge process will come to a halt and need the developer's manually intervention in order to proceed with the merge.

Creating merge conflict with `git merge`

To create a merge conflict with `git merge` it is relatively simple, say you are on the `main` branch, spawn off a branch called `feature-a`. As of now, `main` and `feature-a` are in sync because no commits have been made yet.

Now say in `main` you create a file named "lmao", and appended "this is cool" to the file. Stage it and commit it.

Now checkout into `feature-a` and create the same file named "lmao", but this time append "this is not cool" to the file.

You see that we created the same file for both branches, but the content they have are different and they are both at line number 1.

Now if you attempt to say merge `feature-a` into `main` now by doing `git merge` at `main`, you will get a merge conflict, because git sees that both of the branches are changing the same file at the same line, and it doesn't know which one of changes do you want.

Creating merge conflict with `git rebase`

Basing off from the same scenario as above, if you are currently at `feature-a` branch and say you want to integrate the changes that your co-worker have been making in `main`. If you run `git rebase main` you will also run into merge conflict.

But wait, isn't `git rebase` "replaying" the commits that you have made after it moves the `feature-a` branch to sync with `main`? Well yes, but you will have to define what "replaying" is actually doing.

Under the hood, `git rebase` uses `git cherry-pick`, which will let you add arbitrary commits to the current checked out branch. How does `git cherry-pick` work underneath? It internally finds the diffs and then apply the patches which is the same exact way how `git merge` merges. So `git rebase` is actually using `git merge` in a way under the hood, which as you know will have merge conflicts.

If you want to read more about why `git rebase` results in merge conflicts this is a good stackoverflow question: Link

How does git merge algorithm work?

The exact algorithm that is used is something called 3-way merge algorithm and here is the basics of how it works:

1. Given two version of your code base X and Y. First find a suitable merge base B, this merge base is the common ancestor of both of the version X and Y, this can be done easily by walking up the the commits
2. Perform diff of X with B and Y with B. You will get two diff patches telling you how to get from B to both X and Y.
3. Walk through each of the change blocks in the diff patches, if one of the patch touches a block and the other leaves it, keep the version that had the change. If both diff patches have the same change, keep either one. Now if both patches have changes in the same spot and they don't match, mark it as a conflict to be resolved manually.

Another scenario that I encounter when you would have bunch of merge conflict is that say you're in the main branch. In the main branch you made a couple of commits that touched files A, B, C. Then you forked off from the main branch and spawned a feature branch called feature-a.

Now you go back to main branch and changed A and B and made those commits. Now let's go back to feature-a, and pretend you did some kind of upgrade which removed files A and B, and created a new file D. Now if you try to merge, feature-b into main, you will get merge conflicts because from the base commits where all three files still exists, to main it kept A and B and made some changes. But to feature-b branch, it deleted A and B and created D, Git sees that A and B are both changed but doesn't know which know of the changes you actually want, thus merge conflict.

Now let's go back to the previous previous paragraph's scenario when you touched A, B, C and forked off from main branch. Let's say from main you made a change to C and committed the change. Then switched to feature-b and did an upgrade which removed A and B. Now when you merge feature-b into main, you will NOT get a merge conflict, because from the base commit, main didn't touch A and B, only feature-b touched A and B which removed them from the version control, and Git knows that A and B should be removed and thus no merge conflict will occur.

Revision #3
Created 2023-04-07 13:14:40 UTC by Tamarine
Updated 2023-04-14 23:33:00 UTC by Tamarine