Merging Branches in Git

Contents

The `git merge` Command

Once you have finished up some work on a branch and you are ready to integrate it to your main project (or any other branch), you can merge the branch that you worked on into the main branch or any other target branch of your interest. You can also use merging to combine work that other people have done with your own and vice versa.

To merge a branch, branch_A, into another branch, branch_B, switch to branch_A via:

```
git checkout branch_A
```

Merge it into branch_B by:

```
git merge branch_B
```

Merging will not be possible if there are changes in either your working directory or staging area that could be written over by the files that you are merging in. If this happens, there are no merge conflicts in individual files. You need to commit or stash the files it lists and then try again. The error messages are as follows:

```
error: Entry 'your_file_name' not update. Cannot merge. (Changes in working
```

or

```
error: Entry 'your_file_name' would be overwritten by merge. Cannot merge. (
```

Good practice

First and foremost, your **main branch should always be stable**. Only merge work that is finished and tested (for example, on a different branch). If your project is collaborative, then it is a good idea to merge changes that others make into your own work frequently or share your changes with your collaborators. If you do not do it often, it is very easy for merge conflicts to arise (next section).

Merge Conflicts

When changes are made to the same file on different branches, sometimes those changes may be incompatible. This most commonly occurs in collaborative projects, but it happens in solo projects too. Say there is a project that contains a file with this line of code:

```
print('hello world')
```

Suppose one person, on their branch, decides to “pep it up” a bit and changes the line to:

```
print('hello world!!!')
```

while someone else, on another branch, decides to change it to:

```
print('Hello World')
```

They continue doing work on their respective branches and eventually decide to merge. Their version control software then goes through and combines their changes into a single version of the file; *but*, when it gets to the `hello world` statement, it does not know which version to use. This is a merge conflict: incompatible changes have been made to the same file.

When a merge conflict arises, it will be flagged during the merge process. Within the files with conflicts, the incompatible changes will be marked so you can fix them:

```
<<<<<<< HEAD
print('hello world!!!')
=======
print('Hello World')
>>>>>>> main
```

`<<<<<<<`: Indicates the start of the lines that had a merge conflict. The first set of lines are the lines from the file that you were trying to merge the changes into.

`=======`: Indicates the breakpoint used for comparison. It separates the changes the user has committed (above), from the changes coming from the merge (below), for visual comparison.

`>>>>>>>`: Indicates the end of the lines that had a merge conflict.

You resolve a conflict by editing the file to manually merge the parts of the file that Git had trouble merging. This may mean discarding either your changes or someone else's or doing a mix of the two. You will also need to delete the `<<<<<<<`, `=======`, and `>>>>>>>` in the file. In this project, the users may decide in favour of one `hello world` over another, or they may decide to replace the conflict with:

```
print('Hello World!!!')
```

Once you have fixed the conflicts, commit the new version. You have now resolved the conflict. If during the process, you need a reminder of which files the conflicts are in, you can use `git status` to find out.

If you find there are particularly nasty conflicts, and you want to abort the merge you can use:

```
git merge --abort
```

Good practice

Before you start trying to resolve conflicts, make sure you fully understand the changes and how they are incompatible to avoid the risk of making things more tangled. Merge conflicts can be intimidating to resolve, especially if you are merging branches that diverged many commits ago and now have numerous incompatibilities. However, it is worth remembering that your previous versions are safe and that you can go about fixing this issue without affecting the past versions. This is why it is good practice to **merge other's changes into your work frequently**.

There are tools available to assist in resolving merge conflicts, some are free; some are not. Find and familiarise yourself with one that works for you. Commonly used merge tools include KDiff3, Beyond Compare, Meld, and P4Merge. To set a tool as your default do:

```
git config --global merge.tool name_of_the_tool
```

and launch it with:

```
git mergetool
```

Fundamentally, the best way to deal with merge conflicts is, as far as it is possible, to try to avoid them in the first place. You can improve your odds on this by keeping branches clean and focused on a single issue and involving as few files as possible. Before merging, make sure you know what is in both branches. If you are not the only one that has worked on the branches, then keep the lines of communication open, so you are all aware of what the others are doing.