Difference Between git merge and rebase

Last updated: February 6, 2024

Written by: baeldung

Git
Git Basics

1. Overview

While working with git as our Version Control System (VCS), we may follow any of the branching strategies, but eventually, we may need to integrate changes from one of the feature branches to the main or main branch.

In this tutorial, we'll look at the two different ways in which we can integrate changes from one branch to another.

2. Git Rebase

To put it simply, git rebase **takes your entire feature branch and moves it to the tip of the main branch**. It creates brand new commits for each commit in the original feature branch.

Let's create a new repository and a feature branch in the repository to understand how to rebase works:

```
git clone <your_repository_here>
git branch testBranch1
git branch testBranch2
```

Let's create a new file in the *testBranch1* feature branch and commit the changes:

```
git add .
git commit -m "<Commit_Message_Here>"
git push --set-upstream origin testBranch1
git log
```

Executing these commands will give us the output below:

```
$ git log
commit 2b3a2e47de35d5ba98694585b0cb53412ff06fbc (HEAD -> testBranch1, origin/testBranch1)
Author:
Date:   Wed May 18 10:55:30 2022 +0530

    Rebase:New file feature branch 1

commit db52dd6691b5696c86e9b668aa346f5304bcc404 (origin/main, origin/HEAD, testBranch2, main)
Author:
Date:   Wed May 18 09:22:52 2022 +0530

    Initial commit
```

Now, let's try to rebase this branch on the *main* branch:

```
git rebase main
```

This will result in the following message:

```
$ git rebase main
Current branch testBranch1 is up to date.
```

Since there are no commits in the *main* branch, we shouldn't expect any changes, as evident above.

Now, let's merge the feature branch onto the main branch:

```
git checkout main
git merge testBranch1
git push
git log
```

These commands will output the following:

```
$ git log
commit 2b3a2e47de35d5ba98694585b0cb53412ff06fbc (HEAD -> main, origin/testBranch1, origin/main, origin/HEAD, testBranch1)
Author:
Date:   Wed May 18 10:55:30 2022 +0530

    Rebase:New file feature branch 1

commit db52dd6691b5696c86e9b668aa346f5304bcc404 (testBranch2)
Author:
Date:   Wed May 18 09:22:52 2022 +0530

    Initial commit
```

There is no change to the commit ids from the feature branch while merging to the main branch. This is similar to what happens with fast-forward merge.

Since we have already merged *testBranch1* to the *main* branch, *testBranch2* is missing the commits from where it was cut.

Let's take a look at how *testBranch2* is rebased and merged.

Let's create a new file in the *testBranch2* feature branch and commit the changes:

```
git checkout testBranch2
git add .
git commit -m "<Commit_Message_Here>"
git push --set-upstream origin testBranch2
git log
```

And after these command's completion, we'll see:

```
$ git log
commit 13b72f23989aed4396f8e1e55109b2148682e2a6 (HEAD -> testBranch2, origin/testBranch2)
Author:
Date:   Tue May 17 11:09:38 2022 +0530

    New File Feature Branc2

commit 29a72c86a0701dca032e07abe4d3487f58db4bae
Author:
Date:   Sun May 15 11:04:43 2022 +0530

    Initial commit
```

Now let's try to rebase this branch on the *main* branch:

```
git rebase main
```

And this should give us a different message from the previous case:

```
$ git rebase main
First, rewinding head to replay your work on top of it...
Applying: Rebase:New file feature branch 2
```

Since there are some commits on the *main* branch, the feature branch was rebased on it. Now let's merge the featureBranch2 on the main branch. We should **expect the commit ids to be different for *featureBranch2* before and after rebase:**

```
git checkout main
git merge testBranch2
git push
git log
```

These commands will output the following:

```
$ git log
commit 6dd751e75a720ac6f65b21ef73b413ba1fb62798 (HEAD -> main, origin/main, origin/HEAD, testBranch2)
Author:
Date:   Wed May 18 11:48:10 2022 +0530

    Rebase:New file feature branch 2

commit 2b3a2e47de35d5ba98694585b0cb53412ff06fbc (origin/testBranch1, testBranch1)
Author:
Date:   Wed May 18 10:55:30 2022 +0530

    Rebase:New file feature branch 1

commit db52dd6691b5696c86e9b668aa346f5304bcc404
Author:
Date:   Wed May 18 09:22:52 2022 +0530

    Initial commit
```

The commit ids are different as expected, and if we take a look at the git log graph, we will see that the repo has a linear history:

```
git log --graph --oneline
```

The above command shows a graph structure displaying commit info in a single line:

```
$ git log --graph --oneline
* 6dd751e (HEAD -> main, origin/main, origin/HEAD, testBranch2) Rebase:New file feature branch 2
* 2b3a2e4 (origin/testBranch1, testBranch1) Rebase:New file feature branch 1
* db52dd6 Initial commit
```

3. Git Merge

Git merge will take the two branches we are merging, **find the common base commit and then play the commit sequence from the two branches on the base commit to merge the branches**.

Let's create a new repository and a couple of feature branches to understand how merge works:

Clone the repository in your local machine and create a new feature branch:

```
git clone <your_repository_here>
git branch testBranch1
git branch testBranch2
```

Let's create a new file in the *testBranch1* feature branch and commit the changes:

```
git add .
git commit -m "<Commit_Message_Here>"
git push --set-upstream origin testBranch1
git log
```

Executing these commands will give us the output below:

```
$ git log
commit 183f2e5089c265878d3d8288b9923a5791cf3d11 (HEAD -> testBranch1, origin/testBranch1)
Author:
Date:   Tue May 17 10:58:42 2022 +0530

    New File Feature Branch1

commit 29a72c86a0701dca032e07abe4d3487f58db4bae (origin/main, origin/HEAD, testBranch2, main)
Author:
Date:   Sun May 15 11:04:43 2022 +0530

    Initial commit
```

Now let's merge this feature branch onto the *main* branch using the merge command:

```
git checkout main
git merge testBranch1
git push
git log
```

These commands will output the following:

```
$ git log
commit 183f2e5089c265878d3d8288b9923a5791cf3d11 (HEAD -> main, origin/testBranch1, origin/main, origin/HEAD, testBranch1)
Author:
Date:   Tue May 17 10:58:42 2022 +0530

    New File Feature Branch1

commit 29a72c86a0701dca032e07abe4d3487f58db4bae (testBranch2)
Author:
Date:   Sun May 15 11:04:43 2022 +0530

    Initial commit
```

We can notice that the latest commit ids are the same as the previous image, but the HEAD pointer is pointing to the *main* branch.

The above was a simple merge wherein there were no changes in the *main* branch while we were working on our *feature* branch.

Let's look at another scenario where there are changes in both main and feature branches and how git handles them.

Let's create a new file in the *testBranch2* feature branch and commit the changes:

```
git checkout testBranch2
git add .
git commit -m "<Commit_Message_Here>"
git push --set-upstream origin testBranch2
git log
```

And after these command's completion, we will get the following:

```
$ git log
commit 13b72f23989aed4396f8e1e55109b2148682e2a6 (HEAD -> testBranch2, origin/testBranch2)
Author:
Date:   Tue May 17 11:09:38 2022 +0530

    New File Feature Branc2

commit 29a72c86a0701dca032e07abe4d3487f58db4bae
Author:
Date:   Sun May 15 11:04:43 2022 +0530

    Initial commit
```

Now let's merge this feature branch onto the *main* branch using the merge command:

```
git checkout main
git merge testBranch2
git log
```

We can then see it in the terminal:

```
$ git log
commit 22ecb75ae8371bd7dbd6bf178795750851398 89d (HEAD -> main, origin/main, origin/HEAD)
Merge: 183f2e5 13b72f2
Author:
Date:   Tue May 17 11:11:52 2022 +0530

    Merge branch 'testBranch2' into main

commit 13b72f23989aed4396f8e1e55109b2148682e2a6 (origin/testBranch2, testBranch2)
Author:
Date:   Tue May 17 11:09:38 2022 +0530

    New File Feature Branc2

commit 183f2e5089c265878d3d8288b9923a5791cf3d11 (origin/testBranch1, testBranch1)
Author:
Date:   Tue May 17 10:58:42 2022 +0530

    New File Feature Branch1

commit 29a72c86a0701dca032e07abe4d3487f58db4bae
Author:
Date:   Sun May 15 11:04:43 2022 +0530

    Initial commit
```

There is a separate merge commit on which the HEAD is pointing now while the original commits are present for both the feature branches. The topmost commit also has an additional information key, "Merge", which has the commit ids for both the branches.

We can also check the branch graph and verify the history of the repository:

```
git log --graph --oneline
```

The above command shows a graph structure displaying commit info in a single line:

```
$ git log --graph --oneline
*   22ecb75 (HEAD -> main, origin/main, origin/HEAD) Merge branch 'testBranch2' into main
|\
| * 13b72f2 (origin/testBranch2, testBranch2) New File Feature Branc2
* | 183f2e5 (origin/testBranch1, testBranch1) New File Feature Branch1
|/
* 29a72c8 Initial commit
```

4. Use Cases

Whenever we require our repository history to be linear, we should go for rebasing. But we should be careful about using rebase instead of merging on commits outside our repositories as other collaborators may have their own work based on the existing commits.

Rebasing already pushed commits on a public repo will result in different commit ids, which might make git think that the other developers' main branch and your rebased main branch have diverged. This could create a potentially difficult situation for merging/syncing if there are multiple collaborators.

5. Conclusion

In this article, we covered the basic difference between git merge and git rebase which every developer should know while working with git VCS.