

Types of Merging in Git

In Git, merging is the process of integrating changes from different branches. There are several merge strategies that Git employs to manage how these changes are combined. Here's an overview of the various merge strategies available in Git:

1. Fast-Forward Merge(It is also Called 2-way merge)

- **Description:** This strategy occurs when the branch being merged has no new commits since the branching point. In this case, Git simply moves the branch pointer forward to the latest commit.
- **Use Case:** Useful when you want a linear history and the feature branch is directly ahead of the base branch.
- **Example:** When merging a feature branch back into the main branch after it has not been updated.

2. No Fast-Forward Merge (FF)(also called as 3 way merge)

- **Description:** With this strategy, even if a fast-forward merge is possible, Git creates a new merge commit to preserve the history of the branch.
- **Use Case:** Useful for maintaining the context of when features were developed and merged, keeping a clear project history.
- **Command:** Use `git merge --no-ff <branch>` to explicitly perform this merge.

More illustrations among 2way and 3 way merge

The essential logic of a three-way merge tool is simple:


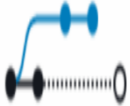
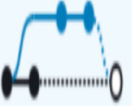
- Compare base, source, and target files
- Identify the "chunks" in the source and target files file:
 - Chunks that don't match the base
 - Chunks that do match the base
- Then, put together a merged result consisting of:
 - The chunks that match one another in all 3 files
 - The chunks that don't match the base in either the source or in the target but not in both
 - The chunks that don't match the base but that do match each other (i.e., they've been changed the same way in both the source and the target)
 - Placeholders for the chunks that conflict, to be resolved by the user.

Note that the "chunks" in this illustration are purely symbolic. Each could represent lines in a file, or nodes in a hierarchy, or even files in a directory. It all depends on what a particular merge tool is capable of.

In day-to-day work, we would do or rather let git do a 2-way merge whenever possible. The question would be why is a 3-way merge needed, instead of the default 2-way merge.

The default merge strategy is a fast-forward merge(see left picture below), but if git can't do a fast-forward merge because the development history has diverged, example extra-commits has been committed to the source branch(but not by you), git has to do some work and it uses a three-way merge (right picture below). Of course, with all things git, you can also decide to do a 3-way merge by using the option `--no-ff`, even when it is not required.

Merge strategy [Info](#)
Determines the way in which the current pull request will be merged into the destination branch

<p><input type="radio"/> Fast forward merge <code>git merge --ff-only</code></p> <p>Merges the branches and moves the destination branch pointer to the tip of the source branch. This is the default merge strategy in Git.</p> 	<p><input type="radio"/> Squash and merge <code>git merge --squash</code></p> <p>Combines all commits from the source branch into a single merge commit in the destination branch.</p> 	<p><input checked="" type="radio"/> 3-way merge <code>git merge --no-ff</code></p> <p>Creates a merge commit and adds individual source commits to the destination branch.</p> 
---	---	---

3.Octopus Merge

- **Description:** This strategy is used when merging more than two branches simultaneously. It creates a merge commit that combines all specified branches.
- **Use Case:** Useful for scenarios where multiple feature branches are merged at once into a single branch.
- **Limitations:** It may not handle conflicts as well as the recursive strategy and is generally not used for complex merges.

4. Rebase Merge

- **Description:** While not a traditional merge strategy, rebasing is a way to integrate changes by moving the entire branch to a new base, applying each commit on top of the target branch.
- **Use Case:** Useful for maintaining a clean and linear commit history, especially in collaborative environments.
- **Command:** Use `git rebase <branch>` to apply this strategy.

5. Squash Merge

- **Description:** This strategy combines all changes from the feature branch into a single commit on the target branch.
- **Use Case:** Useful for condensing a series of small commits into a single coherent commit to keep the history clean.
- **Command:** Use `git merge --squash <branch>`.

6. Ours Merge

- **Description:** This strategy keeps the changes from the current branch while ignoring changes from the branch being merged. The result will reflect the current branch state.
- **Use Case:** Useful in scenarios where you want to record that a merge occurred but don't want to include any of the changes from the merged branch.
- **Command:** Use `git merge -s ours <branch>`.

Conclusion

Each merge strategy in Git serves different purposes and is suited to various workflows. Choosing the right strategy can help maintain a clean project history, handle complex merges, and accommodate team collaboration. Understanding the implications of each strategy is essential for effective source control management.

what is ort strategy in merging in git

The **ORT strategy** (which stands for "Ours, Recursive, and Three-way") is a relatively new merge strategy introduced in Git version 2.34. It is designed to improve performance and conflict resolution during merges, particularly in complex scenarios. Here's a detailed overview of the ORT strategy:

Key Features of the ORT Strategy

1. **Three-Way Merge:**
 - Like the recursive strategy, ORT performs a three-way merge, which means it considers the two branches being merged and their common ancestor to determine how to combine changes.
2. **Improved Performance:**
 - The ORT strategy is optimized for speed and efficiency. It handles merges faster than the previous recursive strategy, particularly in repositories with large histories and many files.
3. **Conflict Resolution:**
 - ORT employs advanced algorithms for conflict resolution, aiming to reduce the number of conflicts that need manual resolution. It analyzes changes more intelligently to make better decisions during the merge process.
4. **Directory Renames:**
 - The ORT strategy is more adept at handling directory renames, ensuring that such changes are recognized and merged correctly.
5. **Better Handling of Large Merges:**
 - ORT is designed to work well with large repositories and extensive changes, making it suitable for projects with significant development activity.

Usage of ORT Strategy

The ORT strategy is the default merge strategy in Git versions 2.34 and later. When you perform a merge without specifying a strategy, Git will automatically use ORT unless another strategy is explicitly requested.

Example of Using ORT Strategy

To perform a merge using the ORT strategy, you can simply run the following command:

```
bash
Copy code
git merge <branch-name>
```

If you want to explicitly specify the ORT strategy (although it's the default), you can do so like this:

```
bash
Copy code
git merge -s ort <branch-name>
```

Conclusion

The ORT strategy represents a significant improvement in Git's merging capabilities, especially for larger projects. Its performance enhancements and better conflict resolution make it a valuable tool for developers working in collaborative environments. If you're using Git version 2.34 or newer, you can benefit from the ORT strategy automatically during merges.