Advanced Git Techniques

Rebasing, Cherry Picking, and More

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

Overview of Advanced Features in Git

Git, as a distributed version control system, offers a plethora of features beyond the basic commands used in day-to-day version control operations. Learning these advanced features can significantly enhance your productivity and ability to manage complex projects.

- **Rebasing:** This allows you to move or combine a series of commits to a new base commit. It's useful for cleaning up commit history and integrating changes from different branches seamlessly.
- **Cherry Picking:** This command allows you to apply changes from specific commits on one branch into another branch without merging the entire branch. It's handy for applying hotfixes.
- **Advanced Merging Techniques:** Beyond the basic merge, Git provides more sophisticated merging strategies like recursive merges and octopus merges, which handle complex branching scenarios.
- **Bisecting:** This powerful tool helps you find the specific commit that introduced a bug by using a binary search algorithm. It's extremely useful for debugging.
- **Hooks:** Git hooks are scripts that run automatically in response to specific events in the Git lifecycle. They can enforce policies, run tests, and automate tasks, improving workflow efficiency.

When to Use Advanced Techniques

While basic Git commands are sufficient for most version control needs, there are scenarios where advanced techniques become necessary:

- When dealing with long-running projects with extensive commit histories, using techniques like rebasing and interactive rebase can help keep the history clean and comprehensible.
- In environments where multiple developers are working on the same project, advanced merging techniques and branching strategies become critical to avoid conflicts and ensure smooth integration of changes.
- Bisecting is particularly useful when you need to identify the exact commit that introduced a bug, especially in large codebases where manual search would be time-consuming.
- Hooks can automate checks and tests before changes are committed or pushed, ensuring that code quality and project standards are maintained consistently.
- Advanced configuration allows teams to customize Git to better suit their specific workflows and project requirements, increasing efficiency and reducing friction.

Importance of Mastering Advanced Git Commands

Understanding advanced Git commands offers several significant benefits:

- Advanced Git commands streamline workflows, reduce time spent on resolving conflicts, and automate repetitive tasks, allowing developers to focus more on coding and less on version control.
- Mastery of advanced features facilitates better collaboration among team members, especially in large teams or open-source projects, by reducing integration issues and maintaining a clean project history.
- Advanced techniques enable more effective project management by providing tools to maintain a clear, organized, and navigable commit history, making it easier to track progress, revert changes, and understand the evolution of the codebase.
- Automated hooks and rigorous use of advanced commands can enforce coding standards and quality checks, ensuring that only well-reviewed and tested code is integrated into the main project.
- Advanced features provide greater control over the version control process, allowing for more flexible and tailored workflows that can adapt to the unique needs of any project or team.

Rebasing

Rebasing is a powerful feature in Git that allows you to move or combine a sequence of commits to a new base commit. Unlike merging, which creates a new commit to combine the histories of two branches, rebasing rewrites the commit history by creating new commits for each commit in the original branch.

Definition and Concept

Rebasing fundamentally changes the base of your branch from one commit to another. It involves the following steps:

- Check out the branch to be rebased: Switch to the branch that you want to rebase onto another branch.
- Reapply commits: Git takes the commits from your current branch and re-applies them one by one onto the new base commit.

For example, if you have a branch `feature` that branches off from `master`, and you want to rebase `feature` onto the latest commit in `master`, you would use the following command:

```
git checkout feature
git rebase master
```

This operation will replay the commits from the `feature` branch onto the tip of the `master` branch, creating a linear history.

Benefits and Risks of Rebasing

Rebasing has its advantages and disadvantages, which are important to understand to use this feature effectively.

Pros: Cleaner Project History

- Rebasing helps maintain a linear project history. This is particularly useful in projects with multiple contributors, as it avoids the clutter created by numerous merge commits.
- A rebased history is generally easier to follow and understand, as it presents a straightforward timeline of changes without the branching and merging paths that can complicate the history.
- Conflicts are resolved during the rebase process, which means they are handled once and don't appear as merge commits later in the history.

Cons: Potential for Conflicts

- Rebasing can introduce conflicts, especially if there have been significant changes in the base branch. These conflicts need to be resolved manually, which can be time-consuming and complex.
- When you rebase, you are rewriting the commit history. This can be problematic if you have already shared your commits with others, as it changes the commit hashes. This can lead to confusion and coordination issues among team members.
- If not done correctly, rebasing can potentially lead to data loss. For example, if you accidentally drop commits or fail to resolve conflicts properly, you might lose important changes.

Example: Basic Rebase

Imagine you have the following commit history:

```
A---B---C feature
 \
 D---E master
```

To rebase the `feature` branch onto `master`, you would use:

```
git checkout feature
git rebase master
```

After the rebase, the commit history would look like this:

```
A---B---C
 \
D---E---B'---C' feature
```

Interactive Rebase

Interactive rebase is a feature in Git that allows you to modify commit history in a flexible and controlled manner. Using `git rebase -i`, you can reorder, squash, and edit commits, providing a powerful way to clean up your commit history before sharing your work with others.

How to Perform an Interactive Rebase

Determine the range of commits you want to rebase. For example, to rebase the last three commits, use the following command:

```
git rebase -i HEAD~3
```

Edit the Rebase Todo List

Git will open the default text editor with a list of commits to be rebased. Each commit is listed with the default action `pick`:

```
pick f7f3f6d Message for commit 1
pick 310154e Message for commit 2
pick a5f4a0d Message for commit 3
```

You can change the action for each commit by replacing `pick` with other commands like `edit`, `squash`, or `reword`.

Reordering Commits

To reorder commits, simply change the order of the lines in the todo list. For example, to swap the order of the first and second commits, edit the list as follows:

```
pick 310154e Message for commit 2
pick f7f3f6d Message for commit 1
pick a5f4a0d Message for commit 3
```

Save and close the editor to proceed with the rebase. Git will apply the commits in the new order.

Squashing Commits

Squashing commits combines multiple commits into a single commit. To squash a commit, replace `pick` with `squash` (or the shorthand `s`) on the commits you want to squash into the previous commit. For example:

```
pick f7f3f6d Message for commit 1
squash 310154e Message for commit 2
pick a5f4a0d Message for commit 3
```

When you save and close the editor, Git will open another editor window to combine the commit messages. You can edit the combined message to reflect the squashed commits.

Editing Commits

To edit a commit, replace `pick` with `edit` (or the shorthand `e`). For example, to edit the second commit:

```
pick f7f3f6d Message for commit 1
edit 310154e Message for commit 2
pick a5f4a0d Message for commit 3
```

After saving and closing the editor, Git will pause at the specified commit, allowing you to make changes. You can then amend the commit:

```
git commit --amend
```

After editing, continue the rebase process with:

```
git rebase --continue
```

Example 1: Reordering Commits

Suppose you have the following commit history:

```
A---B---C---D
```

You want to change the order of commits C and D. Start an interactive rebase for the last two commits:

```
git rebase -i HEAD~2
```

In the editor, change the order of the commits:

```
pick c1c2c3c Commit message for D
pick b2b2b2b Commit message for C
```

Save and close the editor. Git will replay the commits in the new order.

Example 2: Squashing Commits

If your history looks like this:

```
A---B---C---D
```

And you want to squash commits C and D into a single commit, start an interactive rebase:

```
git rebase -i HEAD~2
```

In the editor, change `pick` to `squash` for commit D:

```
pick b2b2b2b Commit message for C
squash c1c2c3c Commit message for D
```

Save and close the editor. Git will combine the commits and prompt you to edit the commit message.

Example 3: Editing Commits

To edit commit C:

```
git rebase -i HEAD~2
```

In the editor, change `pick` to `edit` for commit C:

```
pick b2b2b2b Commit message for C
edit c1c2c3c Commit message for D
```

Save and close the editor. Git will stop at commit C, allowing you to make changes:

```
# Make your changes
git commit --amend
git rebase --continue
```

Cherry Picking

Cherry picking in Git is a powerful feature that allows you to apply specific commits from one branch to another without merging the entire branch. This can be particularly useful when you need to selectively integrate individual changes, such as applying bug fixes or small feature enhancements, from one branch to another.

What is Cherry Picking?

Cherry picking enables you to pick a specific commit from one branch and apply it to another branch. This operation is akin to choosing specific cherries from a tree, hence the name "cherry picking." This process helps in maintaining a clean project history by allowing selective integration of changes rather than merging entire branches, which can be cluttered with irrelevant commits.

Definition and Concept

Cherry picking is the process of selecting a commit from one branch and applying it to another branch. This operation creates a new commit on the target branch that contains the changes from the selected commit.

When you cherry-pick a commit, Git creates a new commit on the target branch that replicates the changes from the original commit. This is different from merging or rebasing, which integrate changes from one branch to another in bulk. Cherry picking is useful when you only want specific changes without bringing in all the commits from the source branch.

Using Cherry Pick

Identify the Commit(s) to Cherry Pick

Use `git log` or a similar command to find the commit hash of the changes you want to cherry-pick. The commit hash is a unique identifier for each commit.

```
git log
```

This will display the commit history, allowing you to locate the commit hash you need.

Check Out the Target Branch

Switch to the branch where you want to apply the changes.

```
git checkout master
```

Execute the Cherry Pick Command

Use the `git cherry-pick` command followed by the commit hash to apply the changes.

```
git cherry-pick <commit-hash>
```

For example, to cherry-pick a commit with the hash `abcd123`:

```
git cherry-pick abcd123
```

Cherry Picking Multiple Commits

You can cherry-pick multiple commits by specifying multiple hashes or a range of commits.

```
git cherry-pick <commit-hash-1> <commit-hash-2>
```

Or using a range:

```
git cherry-pick <commit-hash-start>^..<commit-hash-end
```

Practical Tips and Common Pitfalls

Practical Tips

- Conflicts can occur during cherry-picking if the changes in the commit conflict with the code in the target branch. Git will pause the cherry-pick process and allow you to resolve the conflicts manually.

  ```
  git status
  ```

  This command will show you the files with conflicts. Edit the files to resolve the conflicts, then continue the cherry-pick process:

  ```
  git add <resolved-files>
  git cherry-pick --continue
  ```

- If needed, you can amend the commit message while cherry-picking by using the `-e` option, which opens the commit message in an editor for you to modify.

```
git cherry-pick -e <commit-hash>
```

- Cherry-picking commits that introduce large or complex changes can be risky, as they are more likely to cause conflicts or issues. It's often better to cherry-pick smaller, self-contained commits.
- Keep a record of which commits have been cherry-picked and why, especially in collaborative environments. This can help avoid confusion and duplicate efforts.

Common Pitfalls

- Conflicts are the most common issue when cherry-picking, especially if the codebase has diverged significantly. Carefully resolving conflicts is crucial to maintaining code integrity.
- Cherry-picking rewrites history, which can cause problems if the commits have already been pushed to a shared repository. Ensure that the team is aware of cherry-pick operations to avoid confusion.
- Cherry-picking can create duplicate commits if the same changes are merged or cherry-picked multiple times. This can clutter the commit history and make it harder to track changes.
- Cherry-picking individual commits can sometimes lose the context provided by surrounding commits. Ensure that the cherry-picked commit still makes sense in the new branch.

Use Cases for Cherry Picking

Common Scenarios

- When a critical bug is fixed on a development branch but needs to be applied to the main branch without merging all the changes from the development branch.
- When a specific feature or enhancement developed on a feature branch needs to be integrated into the main branch independently of other changes.
- When an urgent fix is applied to the production branch and needs to be propagated to other branches without waiting for a full merge.
- When certain commits from a long-running feature branch are ready to be integrated into the main branch, but the branch as a whole is not yet complete or stable.

Example 1: Cherry Picking a Single Commit for a Bug Fix

Scenario

You have identified a bug in the feature branch and fixed it in commit `fix123`. You need to apply this fix to the `master` branch.

Identify the Commit

Use `git log` on the feature branch to find the commit hash of the bug fix.

```
git log
```

Locate the hash for the bug fix commit (e.g., `fix123`).

Check Out the Target Branch

Switch to the `master` branch where you want to apply the bug fix.

```
git checkout master
```

Cherry Pick the Commit

Use the `git cherry-pick` command followed by the commit hash.

```
git cherry-pick fix123
```

Resolve Any Conflicts

If there are conflicts, Git will pause and prompt you to resolve them. Edit the conflicting files, add them to the staging area, and continue the cherry-pick process.

```
git status
```

Resolve conflicts, then:

```
git add <resolved-files>
git cherry-pick --continue
```

Verify the Changes

After cherry-picking, verify that the bug fix has been applied correctly by reviewing the commit history and testing the fix.

```
git log
```

Example 2: Cherry Picking Multiple Commits

Scenario

You have multiple commits on the feature branch that you want to apply to the `master` branch.

Identify the Commits

Use `git log` on the feature branch to find the commit hashes you want to cherry-pick.

```
git log
```

Suppose the commits are `commit1` and `commit2`.

Check Out the Target Branch

Switch to the `master` branch.

```
git checkout master
```

Cherry Pick the Commits

Use the `git cherry-pick` command with the commit hashes.

```
git cherry-pick commit1 commit2
```

Resolve Any Conflicts

If conflicts arise, resolve them manually and continue the cherry-pick process.

```
git status
```

Resolve conflicts, then:

```
git add <resolved-files>
git cherry-pick --continue
```

Verify the Changes

Check that both commits have been applied correctly.

```
git log
```

Example 3: Cherry Picking a Range of Commits

Scenario

You want to cherry pick a range of commits from the feature branch to the `master` branch.

Identify the Range of Commits

Use `git log` on the feature branch to identify the start and end of the commit range.

```
git log
```

Suppose the range is from `commitA` to `commitC`.

Check Out the Target Branch

Switch to the `master` branch.

```
git checkout master
```

Cherry Pick the Range of Commits

Use the `git cherry-pick` command with the range of commits.

```
git cherry-pick commitA^..commitC
```

This command cherry picks all commits from `commitA` (excluding `commitA`) to `commitC` (including `commitC`).

Resolve Any Conflicts

If conflicts occur, resolve them and continue the cherry-pick process.

```
git status
```

Resolve conflicts, then:

```
git add <resolved-files>
git cherry-pick --continue
```

Verify the Changes

Ensure all commits in the range have been applied correctly.

```
git log
```

Advanced Merging Techniques

Three-way Merge

A three-way merge is one of the most common methods used by Git to merge changes from two branches. Understanding this technique is helpful for managing complex merge scenarios and resolving conflicts effectively.

Concept and Process

A three-way merge involves three points of reference: the common ancestor (base commit) of the two branches being merged, and the latest commits on each branch. The common ancestor is the last commit that both branches share, which serves as a reference point for identifying changes in each branch.

Process

1. Git automatically determines the common ancestor of the two branches.
2. Git compares the changes made in the first branch (usually the current branch) and the second branch (the branch being merged).
3. Git attempts to merge the changes from both branches. If there are changes in different parts of the files, Git combines them automatically. If there are conflicting changes in the same part of the files, Git flags these conflicts for manual resolution.

Visualization

The three-way merge process can be visualized as follows:

```
A---B---C (master)
 \
  D---E (feature)
```

Here, `A` is the common ancestor, `C` is the latest commit on `master`, and `E` is the latest commit on `feature`.

Performing a Three-way Merge

To perform a three-way merge:

```
git checkout master
git merge feature
```

Git will compare `A` with `C` and `E`, merging the changes from `E` into `C`.

When to Use a Three-way Merge

Three-way merges are useful in various scenarios, particularly in collaborative environments and complex projects:

- When multiple developers work on different branches, three-way merges help integrate changes from multiple branches into a single branch, ensuring all contributions are combined.
- When a feature branch is ready to be integrated into the main branch, a three-way merge incorporates the changes while preserving the history of both branches.
- In release management, three-way merges are used to integrate changes from development branches into release branches, ensuring that all updates and bug fixes are included in the final release.

Practical Example: Three-way Merge

Scenario

You have a `master` branch and a `feature` branch. The `feature` branch introduces new functionality that you want to merge into the `master` branch.

Identify the Branches to Merge

Ensure you know the latest commits on both branches. Use `git log` to inspect the commit history if needed.

```
git log --oneline
```

Check Out the Target Branch

Switch to the branch you want to merge changes into (e.g., `master`).

```
git checkout master
```

Merge the Feature Branch

Use the `git merge` command to merge the changes from the `feature` branch into `master`.

```
git merge feature
```

Resolve Any Conflicts

If there are conflicts, Git will notify you and pause the merge process. Use `git status` to see which files have conflicts.

```
git status
```

Resolve the conflicts by editing the affected files. Conflicted areas will be marked with conflict markers:

```
<<<<<<< HEAD
Content from master
=======
Content from feature
>>>>>>> feature
```

Edit the file to resolve the conflicts, then add the resolved files to the staging area:

```
git add <resolved-file>
```

Continue the merge process:

```
git commit
```

Verify the Merge

Check the commit history to ensure that the merge was successful and all changes have been integrated.

```
git log
```

Recursive Merge

Recursive merge is a more advanced merging strategy used by Git when dealing with complex branch histories. It extends the basic three-way merge process to handle more intricate scenarios, such as multiple merge bases and complex branching structures.

Concept and Process

Recursive merge is a strategy used by Git to handle merges when there are multiple common ancestors (merge bases) between the branches being merged. It recursively merges the common ancestors to create a virtual merge base and then performs a standard three-way merge using this virtual base.

Process

1. Identify Multiple Merge Bases: Git detects all common ancestors between the two branches.
2. Recursive Merging: Git recursively merges these common ancestors to create a single, virtual merge base. This virtual merge base is a synthetic commit that represents the combined changes from all the common ancestors.
3. Three-way Merge: Git then uses this virtual merge base to perform a standard three-way merge with the tips of the two branches.

The recursive merge process is particularly useful in situations where the branches have a complex history of merges and diverges. By recursively combining common ancestors, Git can handle complex branching structures more effectively than with a simple three-way merge.

Use Cases

Recursive merges are particularly useful in the following scenarios:

- When branches have a long history with multiple merges and diverges, recursive merging ensures that all changes are appropriately combined. This is common in large projects with many contributors and feature branches.
- In situations where there are multiple common ancestors between branches, recursive merging can handle these scenarios more gracefully than a basic three-way merge. This often occurs in projects with cyclical merges and remerges.
- Recursive merges help maintain the integrity of the code by ensuring that all relevant changes are included in the merge, even in complex histories. This reduces the risk of missing important changes that might be overlooked in simpler merge strategies.

Practical Example: Recursive Merge

Scenario You have a `master` branch and a `feature` branch with a complex history, including multiple merge bases.

Identify the Branches to Merge Use `git log --graph` to visualize the commit history and identify the complexity of the branches.

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

Check Out the Target Branch Switch to the branch you want to merge changes into (e.g., `master`).

```
git checkout master
```

Merge the Feature Branch Use the `git merge` command. Git will automatically use the recursive merge strategy for complex histories.

```
git merge feature
```

Resolve Any Conflicts If conflicts occur, Git will notify you and pause the merge process. Use `git status` to see which files have conflicts.

```
git status
```

Resolve conflicts by editing the affected files, then add the resolved files to the staging area:

```
git add <resolved-file>
```

Continue the merge process:

```
git commit
```

Verify the Merge Check the commit history to ensure the merge was successful and all changes have been integrated.

```
git log
```

Octopus Merge

An octopus merge is a specialized merging strategy in Git designed to handle the merging of more than two branches simultaneously. This can simplify the process of integrating multiple feature branches into a single branch, but it comes with its own set of considerations.

Definition and Scenarios

An octopus merge is a merge strategy that allows merging multiple branches into a single branch in one go. Unlike a typical three-way merge, which involves only two branches and their common ancestor, an octopus merge can involve three or more branches at once. This method is particularly useful when you need to consolidate several branches into a main branch or another feature branch.

Scenarios

- When you have multiple feature branches that have been developed in parallel and need to be integrated into the main branch at the same time.
- When preparing for a release, you might want to merge several stable branches into the release branch.
- In projects with complex development workflows where multiple branches need to be combined regularly, an octopus merge can streamline the process.

However, it's important to note that octopus merges are best suited for scenarios where the branches being merged do not have conflicts. If conflicts are present, Git does not handle them well in an octopus merge, and it is usually better to merge the branches in pairs.

Handling Multiple Branches Simultaneously

To perform an octopus merge, follow these steps:

Identify the Branches to Merge

Determine the branches you need to merge. For example, you have branches `feature1`, `feature2`, and `feature3` that you want to merge into `master`.

Check Out the Target Branch

Switch to the branch where you want to merge the other branches.

```
git checkout master
```

Perform the Octopus Merge

Use the `git merge` command with all the branches you want to merge.

```
git merge feature1 feature2 feature3
```

This command attempts to merge all specified branches into `master` simultaneously.

Resolve Any Conflicts (if applicable)

Ideally, there should be no conflicts in an octopus merge. However, if conflicts arise, Git will not handle them well, and it might be necessary to abort the merge and handle the branches in pairs.

```
git merge --abort
```

If conflicts are expected, consider merging the branches one by one or in pairs to resolve conflicts more easily.

Practical Example: Octopus Merge

Scenario You have three feature branches (`feature1`, `feature2`, `feature3`) that have been developed independently and are now ready to be integrated into the `master` branch.

Identify the Branches to Merge Ensure you know the latest commits on each branch. Use `git log` or `git branch` to inspect the branches.

```
git branch
```

Check Out the Target Branch Switch to the `master` branch where you want to perform the merge.

```
git checkout master
```

Perform the Octopus Merge Execute the merge command with all the branches you want to merge.

```
git merge feature1 feature2 feature3
```

Verify the Merge Check the commit history to ensure all branches have been successfully merged.

```
git log
```