Understanding Git: Rebasing

Introduction 1: Repositories 2: Branching 3: Merging 4: Collaborating 5: Rebasing

In This Section

Rebasing

Git offers a unique feature called `rebasing` as an alternative to merging. Its syntax is:

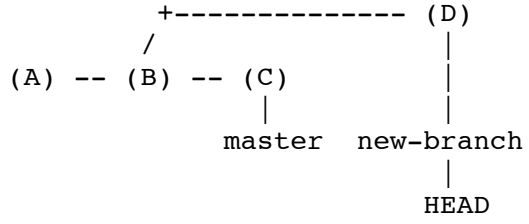
```
git rebase [new-commit]
```

When run, Git performs the following steps:

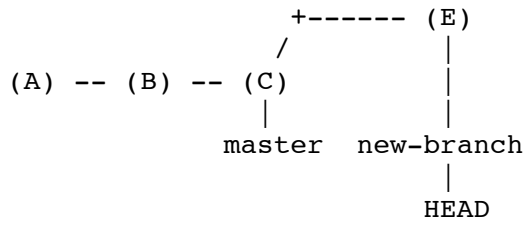
- Identifies each commit that is an ancestor of the current commit but not of [new-commit]. This can be thought of as a two-step process: first, find the common ancestor of [new-commit] and the current commit; call this the *ancestor commit*. Second, collect all the commits between the *ancestor commit* and the current commit.
- Determines what changed for each of those commits, and puts those changes aside.
- Sets the current head to point to [new-commit].
- For each of the changes set aside, replays that change onto the current head and creates a new commit.

The result is that HEAD, the current commit, is a descendant of [new-commit], but it contains all the changes as if it had been merged with [new-commit].

For example, imagine that a repository looks like:



Performing `git rebase master` would produce the following:



where (E) is a new commit that incorporates the changes between (B) and (D), but reorganized to place those changes on (C).

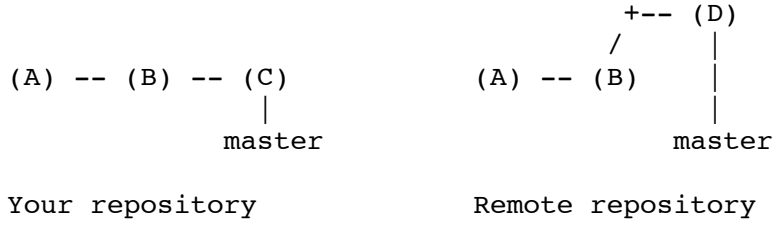
Rebase has the advantage that there is no merge commit created. However, because HEAD is not a descendant of the pre-rebase HEAD commit, rebasing can be problematic. For one thing, it means that a rebased head cannot be pushed to a remote server, because it does not result in a fast forward merge. Moreover, it results in a loss of information. It is no longer known that (D) was once on the *new-branch* head. This results in a "changing of history" that could confuse someone who already knows about commit (D).

Common Rebasing Use Practices

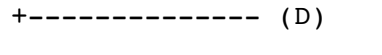
Because of this danger of rebasing, it is best reserved for two situations.

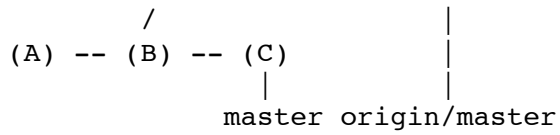
First, if you are developing a branch on your own and not sharing it with anyone, you could rebase it to keep the branch up to date with respect to the main branch. Then, when you finally merge your developed branch into the main branch, it will be free of merge commits, because it will appear that your development branch was a descendant of the main head. Moreover, the main branch can move forward with a fast forward merge rather than a regular merge commit.

Second, if you commit to a branch but that branch changes at the same time on a remote machine, you can use rebase to shift your own commits, allowing you to push your commits to the remote repository. From the example above:



If you perform a `fetch`, your repository will look like this:

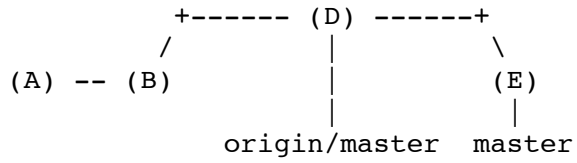




Now, assuming *master* is your current head, you run:

```
git rebase origin/master
```

And your repository will look like:



Commit (E) contains the changes you made between commits (B) and (C), but now commit (E) is a descendant of (D). This allows you to push your head *master*, as a fast forward merge, to update *origin/master*.

Moving On

There are many more things possible with Git. The manual pages generally document, in fairly good detail, the possible operations.

If you understand how the Git repository is a tree of commits and see how operations like branching, merging, pushing, and pulling manipulate that tree, then understanding other Git commands should not be difficult. You should be able to visualize how any command will search or modify the tree and specify the right commits on the tree for Git to operate on.

So, go forth and code!
