

Intermediate Git

Making Better Use of Git

October 1, 2018

Peter J. Jones

✉ pjones@devalot.com

🐦 @devalot

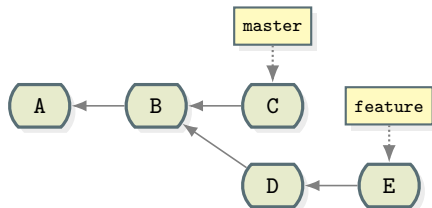
<http://devalot.com>



What's In Store

Before Lunch	After Lunch
TBD	TBD

Separate Lines of Development

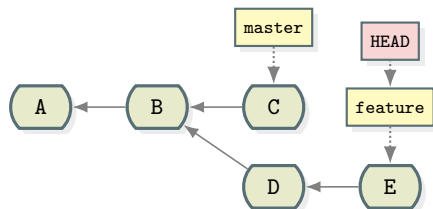


- Branches allow you to work independently of others
- The default branch in Git is called *master*
- Branches can be joined back together via *merging*

Quiz: Current Branch

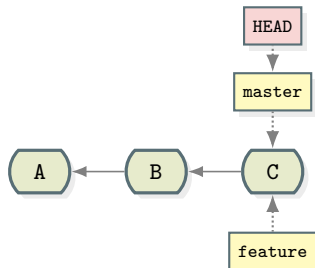
How does Git know what branch you are currently using?

What Branch Are You Using?



- Git uses a pointer called HEAD to track the current branch
- The `.git/HEAD` file references a branch's head
- A branch's head is its latest commit ID

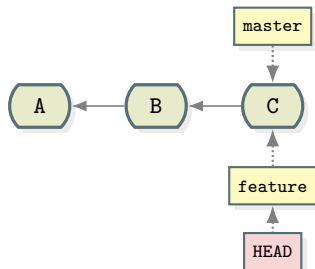
How Do You Create a Branch?



```
$ git branch feature
```

- Creates a branch called feature
- HEAD still points at master

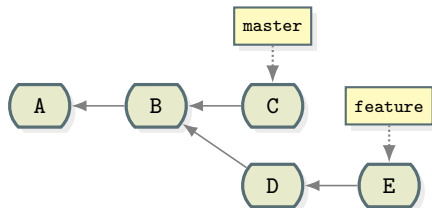
How Do You Switch Branches?



```
$ git checkout feature
```

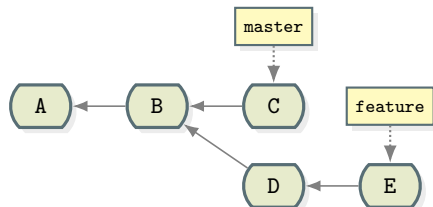
- Changes where HEAD points
- (HEAD now points to the feature branch)

Merging Branches



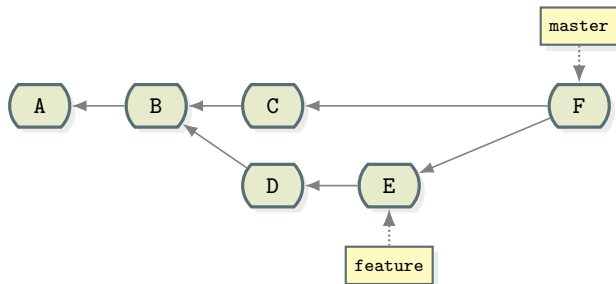
- Work on the feature branch is complete
- We would like commits D and E to move to the `master` branch
- We will do this through merging

Revisiting the Merge Base



- The feature branch and master branch share a common ancestor
- That ancestor is commit B
- Git will perform a 3-way merge with B, C, and E
- The result is a commit with two parents

Merges Cause Multiple Parents



- A new commit is created: F
- This is called a *merge commit* and has two parents

Performing a Merge

1. Switch to the destination branch:

```
$ git checkout master
```

2. Merge in the other branch:

```
$ git merge feature
```

3. Write a commit message in your text editor

Exercise: Merging a Branch

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create and switch to a new branch that starts at v1.0:
`$ git checkout -b NAME v1.0`
3. Merge the `origin/feature` branch into your new branch

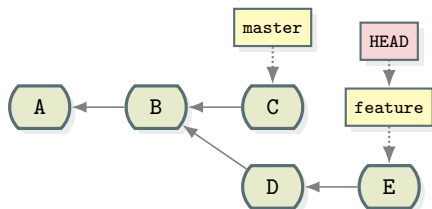
What is Rebasing?

When rebasing a topic branch (i.e. `feature`) onto an ancestor branch (i.e. `master`):

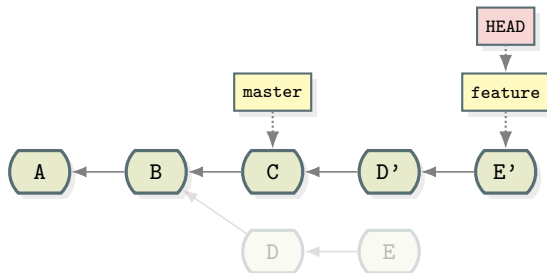
- A new commit history is created for `feature` so that it comes *after* all work on `master`
- New commits are created, old commits are abandoned
- Commits are created by *replaying* diffs on top of `master`

Visualizing a Rebase

Before the rebase:



After the rebase:



A Word of Caution

While rebasing is completely safe, it can make things difficult if you are not careful.

- Rebasing published branches can lead duplicated commits for other team members
- Best practice: Rebase before pushing commits to a remote branch
- Each team member should work on their own branches
- We'll talk about ways to fix things if you break these rules

How to Rebase a Branch

1. Use `git checkout` to move HEAD to the branch you want to rebase. Typically this will be a topic branch.
2. The index and working directory should be clean. Commit or stash changes as necessary.
3. Run `git rebase` listing the ancestor branch you want to rebase onto.
4. If there are merge conflicts, resolve them and continue with `git rebase --continue`.

Example Rebase

Rebasing feature onto master:

```
$ git checkout feature
```

```
Switched to branch 'feature'
```

```
$ git rebase master
```

```
First, rewinding head to replay your work on top of it...
```

```
Applying: ...
```

```
...
```

Exercise: Rebasing a Branch

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME merge-start`
3. Use `git log --oneline` to review the commit history
4. Rebase onto of the `master` branch
5. Review the commit history again and identify the rewritten commits and the merged commits from `master`

Exercise: Rebasing with Conflicts

1. Change to the directory holding a clone of the following repository:
`repos/conflicts.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME conflict-start`
3. Use `git log --oneline` to review the commit history
4. Rebase onto of the `master` branch
5. Fix the conflict and use `git rebase --continue` to resume
6. Review the commit history again and identify the rewritten commits and the merged commits from `master`

Pulling and Rebasing

When rebasing onto a remote branch you have two options:

1. Use `git fetch` to fetch the remote branch, followed by `git rebase` to start the rebase process:

```
$ git fetch origin master  
$ git rebase origin/master
```

2. Use `git pull --rebase`

```
$ git pull --rebase origin master
```

When `git pull` is given the `--rebase` flag it will do a rebase after the fetch instead of the usual merge.

Using the Rebase Command Interactively

When you want to have full control of the rebase process you can perform an interactive rebase instead. An interactive rebase allows you to:

- Select a range of commits to rewrite (start with the *parent* of the commit you want to start with)
- Optionally stop at each commit so you can change the commit message or add more files
- Squash multiple commits into a single commit (we'll do this later)
- Run a command after stopping on specified commits

Starting an Interactive Rebase

After selecting a range of commits, start the rebase with either the `--interactive` flag or the corresponding short version: `-i`. Here's an example:

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

The rebase command will start your text editor with a rebase script where you can specify what you want to do with each commit in the given range.

(Note: Remember, you are rewriting the commit history.)

Rebase Order and Commands

The rebase script allows you to perform several actions:

- pick: Keep the commit as is
- edit: Stop and amend the commit
- squash: join commit to preceding commit
- Reorder script lines: reorder commits
- Delete script lines: delete commits

Overall Flow of an Interactive Rebase

1. Pick a range of commits you want to rewrite
2. Edit the rebase script and exit the text editor
3. Do necessary work on each commit, then continue the rebase

Exercise: Interactive Rebasing

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
3. Interactively edit the commit which has the message “Add a version number”
4. Amend the commit when the rebase stops on it:

```
$ git commit --amend
```
5. Resume the rebase:

```
$ git rebase --continue
```

What Does It Mean to Squash Commits?

Have you ever made several small commits while debugging something?
Ever wanted to join all those commits into a final, good commit?
That's what squashing is.

Squashing Commits with a Rebase

To squash commits:

1. Start an interactive rebase for all commits to squash
2. Leave the first listed commit as “pick”
3. Change all remaining commits to “squash”
4. Exit your text editor

(Note: If you want to rebase all commits you will need to give the `--root` flag to the rebase command.)

Exercise: Squashing Commits

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME merge-start`
3. Take note of the commit history
4. Start an interactive rebase with the first commit in the history
5. Squash all commits into the first listed commit
6. Review the commit history

Rebasing Published Branches

First, avoid rebasing commits that have been published.

However, if you must, then keep these items in mind:

- Rebasing creates *new* commits and *abandons* existing commits
- Your teammates will see duplicates of all rebased commits
- Anyone who has based work on rebased commits will be confused and will probably end up having a very complicated commit history
- Those people should run: `git pull --rebase`

Everyone Must Pull

When you rebase published commits *everyone else* should immediately perform a `git pull --rebase` to update their branches.

This type of pull is smart enough to detect rebased commits and fix any local commits that relied on them.

You may even want to make this the default kind of pull:

```
$ git config --global pull.rebase true
```

What a Reset Can Affect

Before figuring out what a `git reset` is, let's first talk about what it can change:

- The working directory: the project's files
- The index: what changes will be in your next commit
- HEAD: the tip of the current branch

What is a Reset?

A `git reset` changes the state of one or more trees. It's meant as a permanent change unlike `git checkout`.

There are three types of a reset:

- Soft (only change the tip of the current branch)
- Mixed (soft + change the index)
- Hard (mixed + update the working directory)

(Note: Like with rebase, a reset will change your Git history. You should only do this for unpublished commits.)

Performing a Soft Reset

A *soft* reset will:

- Move HEAD back to the specified commit
- Keep the index at the current commit
- Leave the working directory alone

(This is very similar to `git commit --amend.`)

Exercise: Soft Reset

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME merge-start`
3. Use `git log --oneline` to review the commit history
4. Use `git reset --soft` to back up 2 commits:
`$ git reset --soft HEAD~2`
5. Use `git log --oneline` to see how the history changed
6. Use `git status` to review the index and working directory
7. Use `git commit` to create a new commit

Performing a Mixed Reset

A *mixed* reset will:

- Move HEAD back to the specified commit
- Move the index back to the specified commit
- Leave the working directory alone

(This is the default type of reset if you don't use any flags.)

Exercise: Mixed Reset

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME merge-start`
3. Use `git log --oneline` to review the commit history
4. Use `git reset --mixed` to back up 2 commits:
`$ git reset --mixed HEAD~2`
5. Use `git log --oneline` to see how the history changed
6. Use `git status` to review the index and working directory
7. Use `git commit` to create a new commit

Performing a Hard Reset

A *hard* reset will:

- Move HEAD back to the specified commit
- Move the index back to the specified commit
- Update the working directory to the specified commit

(Warning: A hard reset **can delete files** from your working directory. It's a good idea to commit or stash changes before doing a reset.)

Exercise: Hard Rest

1. Change to the directory holding a clone of the following repository:
`repos/basic.git`
2. Create a branch that starts at the commit named `merge-start`
`$ git checkout -b NAME merge-start`
3. Use `git log --oneline` to review the commit history
4. Use `git reset --hard` to back up 2 commits:
`$ git reset --hard HEAD~2`
5. Use `git log --oneline` to see how the history changed
6. Use `git status` to review the index and working directory
7. Notice that the working directory was also changed

Unstaging Changes

A reset is most often used to *unstage* a file. Since the default type of reset is a mixed reset, and the default commit is HEAD, you can unstage a file using:

```
$ git reset -- file.c
```

Which is the same as

```
$ git reset --mixed HEAD file.c
```

What is Interactive Staging?

When you give the `--interactive` option to the `git add` command you enter an interactive shell where you can:

- See the differences between the index and the working directory
- Select files to unstage
- Stage individual patch hunks

Starting an Interactive Staging Session

Anytime you would normally use `git add` or `git rm` to update the index you can use:

```
$ git add --interactive
```

or:

```
$ git add -i
```

instead.

Staging Individual Patch Hunks

There are two ways to stage individual patch hunks:

1. Enter the interactive staging tool:

```
$ git add -i
```

2. Select the patch tool:

```
What now> patch
```

Alternatively, you can jump right into the patch tool with:

```
$ git add --patch main.c
```

Unstaging Individual Patch Hunks

If you have staged more changes than you want to commit you can unstage individual patch hunks:

```
$ git reset --patch main.c
```

This will drop you into the interactive patch tool for unstaging.

Exercise: Staging Patch Hunks

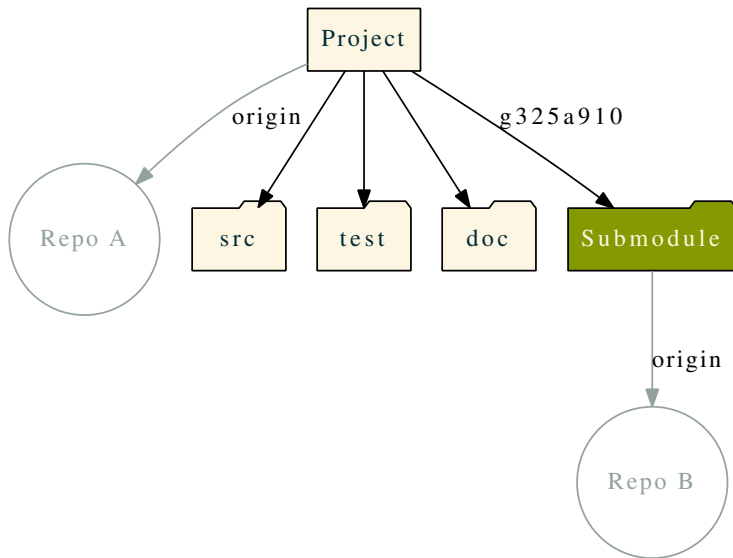
1. Change to the directory holding a clone of the following repository:
`repos/conflicts.git`
2. Create a branch that starts off the `origin/feature` branch
3. Edit `main.c`; add comments to the top and bottom of the file
4. Review the difference between the working directory and the index:

```
$ git diff
```
5. Interactively stage one of the patch hunks by splitting the first hunk:

```
$ git add -p main.c
...
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? s
```
6. Review the updated index:

```
$ git status
$ git diff --cached
```

What are Submodules?



Submodules allow you to embed one repository into another.

Adding an External Repository

Use the `git submodule add` command to import another repository.

1. You need the URL for the other repository, we'll use:

```
https://github.com/pjones/emacsrmc.git
```

2. You need to decide what directory to place the repository into, we'll use:

```
emacsrmc
```

3. Add the external repository:

```
$ git submodule add \  
    https://github.com/pjones/emacsrmc.git emacsrmc
```

Cloning Projects with Submodules

When you clone a repository with submodules Git *will not* fetch the submodules by default. You have two choices:

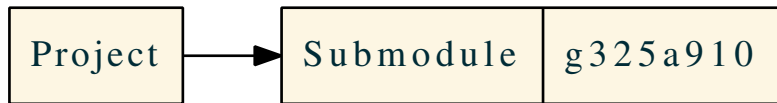
1. Clone then initialize submodules:

```
$ git clone <url>
$ cd <dir>
$ git submodule update --init
```

2. Use a recursive clone:

```
$ git clone --recurse-submodules <url>
```

Understanding Submodules



Project The main repository that is using submodules

Locked Commit Each submodule in a project is locked to a specific commit

Upstream The original repository the submodule was cloned from

Updating a Submodule

After doing a `git pull` the *locked commit* for a submodule may have changed. There are two ways to update the submodule:

1. Update all submodules:

```
$ git pull  
$ git submodule update
```

2. When doing the pull, fetch submodules too:

```
$ git pull --recurse-submodules
```

Updating a Submodule from Its Upstream Source

From time to time you may want to bring in upstream changes to your submodules. You can do this with `git submodule update`:

```
$ git submodule update --remote --checkout
```

When you do this:

- The update will pull from the upstream's master branch by default (you can change this with `git config`)
- `git status` will report whether or not the submodule changed
- `git diff` will show changes to the locked commit
- `git diff --submodule` will show a nicer view of this

Making Changes to a Submodule

Git tracks submodules by keeping them on a *locked commit* and **not a branch**. Git reports this as a **detached HEAD**.

Before you work on a submodule you should put it on a branch. You have several options:

- Bring the submodule up-to-date with its upstream branch:

```
$ git submodule update --remote --checkout  
$ cd <dir>  
$ git checkout master
```

- Manually move it to a branch and pull:

```
$ cd <dir>  
$ git checkout master  
$ git pull
```

Publishing Changes to Submodules

When you are done working on a submodule you will likely want to publish those changes so other team members can see them.

If you push to your project's upstream before pushing submodule changes to their upstream you will **break** the repository for others.

To push the submodule changes you can either:

1. Push each submodule individually:

```
$ git submodule foreach 'git push'
```

2. Push the project and submodules at the same time:

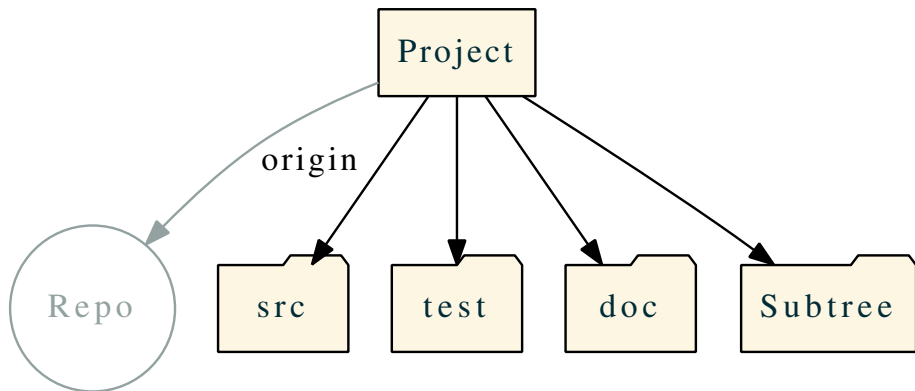
```
$ git push --recurse-submodules=on-demand
```

Exercise: Experimenting with Submodules

For each of the following steps you should create a unique branch off of master so you can easily jump back to the previous state.

1. Change to the directory holding a clone of the following repository:
`repos/submodules.git`
2. Go into the other submodule and see what branch it is on
3. Update the submodule to its latest upstream commit
4. Commit and push the project
5. Make a change to the submodule and commit it
6. Commit to the project and push everything
7. Add another submodule

What are Subtrees?



Subtrees include the *contents* of one repository into another.

Adding an External Repository

Add a subtree bringing in all of its history:

```
$ git subtree add -P <dir> <url> <branch>
```

Add a subtree but squash its history:

```
$ git subtree add --squash -P <dir> <url> <branch>
```

(Note: As far as Git is concerned, you just added files to the local repository.)

Making Changes to a Subtree

- Nothing special, just edit the files in the subtree like any other directory in your project.
- Git sees the changes like all other local changes

Pushing Commits Upstream

When you want to share your changes with upstream:

```
$ git subtree push -P <dir> <url> <branch>
```

You may need to pull changes from upstream first:

```
$ git subtree pull -P <dir> <url> <branch>
```

If there are merge conflicts you may have to resort subtree splitting and merging. See the `git-subtree` manual for more details.

Exercise: Experimenting with Subtrees

1. Change to the directory holding a clone of the following repository:
`repos/subtrees.git`
2. Alter the `other/main.c` file and commit the change
3. Notice that all files are local to the repository
4. Read the `notes.txt` file and then push the subtree

Submodules vs. Subtrees

Submodules:

- Easier to work with
- Pushing/pulling works as expected
- Need to watch out for a detached HEAD
- Need to remember to update after a pull

Subtrees:

- Best when you don't plan on editing files in the subtree
- A push can be rejected for non-obvious reasons
- Never need to worry about detached HEAD or updating

What is the Blame Command?

TBD

Reviewing the Output of `git blame`

TBD

Exercise: Using the Blame Command

(Instructions forthcoming.)

What is the Bisect Command?

TBD

The Bisect Workflow

TBD

Exercise: Bisecting a Project

(Instructions forthcoming.)

Automating a Binary Search

TBD

Introduction to Branch Management

If you don't have a plan it's easy to create a mess of confusing branches in your repository. Branch management is a way for all team members to agree on a plan for:

- Keeping the master branch in a releasable state
- Creating branches for new features and bug fixes
- Where to merge these feature branches
- How to make and record software releases

Mainline Branches

It's generally advisable to have two main branches in your repository:

- `master` represents an always releasable stable branch
- `next` or `develop` is where work for the next release goes

Typically you never directly commit to these branches.

Topic Branches

When you need to create a new feature or fix a bug you:

1. Create a new *topic* branch off of the `next` branch
2. Work on your branch until it is ready for release
3. Merge your topic branch back into `next`

(Note: In a lot of cases your topic branch should be pushed back to a private clone and only show up in the main repository after it is merged.)

Release Branches

When the time comes to create a release you:

1. Create a *release* branch from the `next` branch (at this point `next` can resume being used for a future release)
2. Finalize the release on the release branch (i.e. bump the version number)
3. Merge the release branch into `master` and tag the release
4. Merge the release branch back into `next`

Tools to Automate Branch Management

If you are interested in using this model of branch management you may be interested in a way to automate it.

That's precisely what gitflow does.