How to Use Git and GitHub

Instructor Notes for Lesson 2

Contents

1 Morsel 3: Initializing a Repository
1.1 Git repositories and directories
2 Morsel 6: Staging Area
3 Morsel 9: Committing Changes
3.1 Git cheat sheet
4 Morsel 10: git diff Revisited
5 Morsel 11: Commit the Bug Fix
6 Morsel 17: Branches for Collaboration
6.1 Checking out the coins branch
6.2 Viewing the commit history
7 Morsel 23: Merging Coins into Master
8 Morsel 24: Merging on the Command Line
8.1 Checking out the coins branch
8.2 A note about `git merge`
8.3 Merge conflict
8.4 Merge conflict (Newline characters between Windows and Unix systems)
8.5 Comparing a commit to its parent
9 Morsel 28: Update Easy Mode
9.1 Diagramming Tools
10 Morsel 30: Committing the Conflict Resolution

Morsel 3: Initializing a Repository

Git repositories and directories

Each Git repository is tied to a specific directory - the directory where you ran `git init`. Only files from that directory (and subdirectories inside that directory) will be contained in that repository, and you can have different repositories in different directories.

Note: it's often the case that a Git repository in some directory will only contain, or track, **some** of the files in that directory, rather than **all** of them. You'll see how this works later this lesson.

Morsel 6: Staging Area

If you accidentally add a file to the staging area, you can remove it using `git reset`. For example, if you accidentally add lesson_2_reflections.txt, but don't want it to be committed yet, run `git reset lesson_2_reflections.txt` and the file will be removed from the staging area, but it will still be in your working directory.

Morsel 9: Committing Changes

Git cheat sheet

You've just learned several Git commands pretty quickly. If you'd like some help keeping track of them, you might be interested in this Git cheat sheet. By now, we've covered the commands in the sections "Configure Tooling", "Create Repositories", and "Make Changes". You'll be learning many of the remaining commands on this sheet throughout the rest of this course.

Morsel 10: git diff Revisited

If you are following along, you should run `git checkout master` before you commit. This is because your HEAD is still 'detached' from Lesson 1 when you checked out an old commit, and running this command will fix that. You'll learn more about what this command does later this lesson.

Morsel 11: Commit the Bug Fix

You may notice that our commit id is different from yours, even though we made the same change, while the commit ids up to this point have all been the same. That's because if there is any difference between two commits, including the author or the time it was created, the commits will have different ids. Check out this page if you're interested in learning more about Git internals and how commit ids are generated.

Morsel 17: Branches for Collaboration

Checking out the coins branch

Note that you'll need to check out the coins branch using the command `git checkout coins` before you can view the coins branch using `git log`.

Viewing the commit history

The full command Caroline types to see the visual representation of the commit history is `git log --graph --oneline master coins`.

Morsel 23: Merging Coins into Master

If a branch is deleted and leaves some commits unreachable from existing branches, those commits will continue to be accessible by commit id, until Git's garbage collection runs. This will happen automatically from time to time, unless you actively turn it off. You can also run this process manually with `git gc`.

Morsel 24: Merging on the Command Line

Checking out the coins branch

If you haven't already checked out the coins branch, you'll need to do so now with the command `git checkout coins` before you'll be able to refer to it. Once you've done that, decide whether you should keep it checked out or check out a different branch before completing the merge.

A note about `git merge`

`git merge` will also include the currently checked-out branch in the merged version. So if you have branch1 checked out, and you run `git merge branch2 branch3`, the merged version will combine branch1 as well as branch2 and branch3. That's because the branch1 label will update after you make the merge commit, so it's unlikely that you didn't want the changes from branch1 included in the merge. For this reason, you should always checkout one of the two branches you're planning on merging before doing the merge. Which one you should check out depends on which branch label you want to point to the new commit.

Since the checked-out branch is always included in the merge, you may have guessed that when you are merging two branches, you don't need to specify both of them as arguments to `git merge` on the command line. If you want to merge branch2 into branch1, you can simply `git checkout branch1` and then type `git merge branch2`. The only reason to type `git merge branch1 branch2` is if it helps you keep better mental track of which branches you are merging.

Also, since the two branches are merged, the order in which they are typed into the command line does not matter. The key is to remember that `git merge` always merges all the specified branches into the currently checked out branch, creating a new commit for that branch.

Merge conflict

If you get a message like this

```
Auto-merging game.js
CONFLICT (content): Merge conflict in game.js
Automatic merge failed; fix conflicts and then commit the result.
```

then your files were not in the same state as Caroline's when you started the merge. To fix this, complete the following steps:

1. Restore your files to their state before you started the merge by running `git merge --abort`
2. Double check the state of your files. If you run `git log` while the master branch is checked out, you should see Caroline's "Add color" commit as the second-most-recent, and the most recent should be your commit fixing the bullet bug. If you use `git diff` to compare your commit to Caroline's, your commit should introduce the line `this.delayBeforeBullet = 10;` on line 424. The line should be indented to the same level as the line below it using only spaces (no tabs), and the line should have no spaces after it.
3. Once your file is in the correct state, create a new commit with your changes.
4. Try the merge again.

Merge conflict (Newline characters between Windows and Unix systems)

Context: Whenever we hit the "Enter" key on the keyboard, we are actually telling the computer to insert an invisible character into our text file to indicate to the computer that there should be a new line. Unix systems adds one character called the "line feed" character or LF or \n while Windows systems adds two characters, "carriage return" and "line feed" or CRLF or \r\n.

Caroline's files have LF because her files were edited on Mac OSX, which uses LF. If a Windows user were to edit Caroline's files, the Windows text editor might convert all LF to CRLF to make editing files possible. When the Windows user merges her file with Caroline's files, a merge conflict will result due to the different LF and CRLF characters.

To fix this, Windows users should set the global autocrlf attribute to true: `git config --global core.autocrlf true`. More information can be found here: https://help.github.com/articles/dealing-with-line-endings/#platform-all

Comparing a commit to its parent

The command Caroline mentions to compare a commit to its parent is `git show commit_id`

Morsel 28: Update Easy Mode

Diagramming Tools

- gliffy
- yUML

If you have a favorite that we don't have listed here, mention it in a forum post so that others can find it!

Morsel 30: Committing the Conflict Resolution

Near the beginning of the video, Caroline accidentally says "I still need to let Git know that the conflict is reserved." She meant "resolved", not "reserved".

This page was last edited on 2015/03/12 21:06:28.