Git, part 1

UNC COMP 523

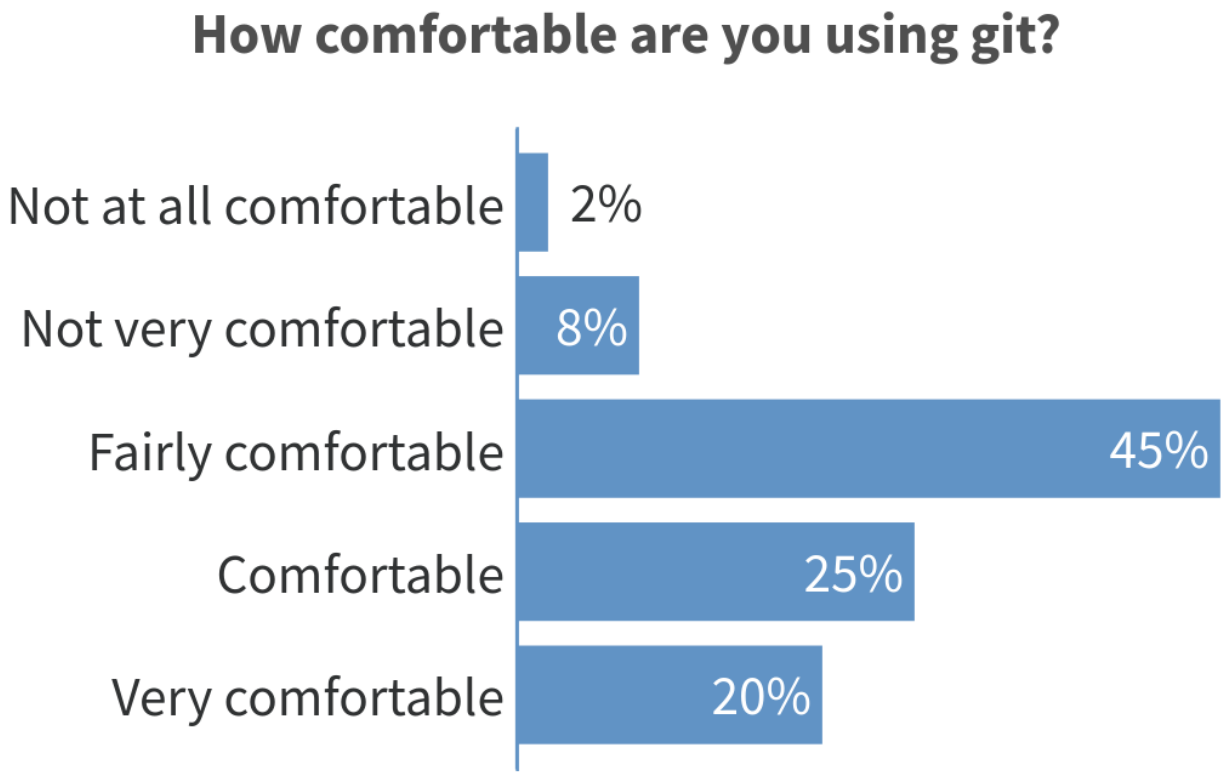
Wed Sep 23, 2020

Prof. Jeff Terrell

Announcements

- music: *Polynomial C* by Aphex Twin
- you shouldn't have to pay for infrastructure! (most) clients have agreed to pay
- grades up to date as of 11am this morning
- pivoting a bit from my previous git plans

Prior git knowledge



Plan for today

- general goal: equip you to use git well *together*
- briskly cover the foundational concepts
- cover my recommendations
- emphasis on concepts and principles, to avoid ignorant following of recipes
- but also include recipes and demos

Outline

- intro
- foundational concepts
- basic collaborative recipe
- merge strategies
- advanced collaborative recipe
- other topics (continuing into next lecture)

Outline

- **intro: what is git? why study it?**
- foundational concepts
- basic collaborative recipe
- merge strategies
- advanced collaborative recipe
- other topics (continuing into next lecture)

What is git?

git is **a tool for storing and sharing snapshots of the contents of a directory**

- git != GitHub, which is a git hosting service
- git is a command-line tool
- many "porcelains" (other UIs) for git exist
- we'll focus on using CLI git as the common denominator, and because it is closer to the raw concepts

Why study git?

- it's incredibly dominant in industry
- it's not going anywhere
- it's actually really neat!
- knowing how to use it can save you lots of headache
- win the respect of your peers :-)

Outline

- ~~intro~~
- **foundational concepts: repos, commits, branches, and remotes**
- basic collaborative recipe
- merge strategies
- advanced collaborative recipe
- other topics (continuing into next lecture)

Concept 1: repository

- a repository, or "repo", is a *codebase*, i.e. a code database
- each repo contains code (or at least data, e.g. the markup for the class website)
- each repo has its own history
- when you go to the code for your project on GitHub/GitLab, that's showing you info from the repo
- when you clone something, you're cloning a repo
- repos are stored locally in a directory on your hard disk
- (remember that git is a tool for storing and sharing snapshots *of the contents of a directory*)
- there's a hidden `.git` subdirectory with the codebase data

Concept 2: commit

- remember that git is a tool for storing and sharing *snapshots* of the contents of a directory
- in git, we call these snapshots *commits*
- commits contain metadata: author, timestamp, message, and parent commits
- (technically, a tree is a snapshot, and the commit is a snapshot + metadata)
- commits are addressed by their contents, so are effectively immutable
- when you say `git commit`, you create a commit with your info, the current timestamp, and the message you provide, linked to the previous commit
- ever notice how GitHub includes a random-seeming hex ID with the commit, e.g. `9a1d67c7`? that's the address.

Concept 3: the working tree

- the "working tree" is what's actually on your hard disk
- if you have a "dirty tree", you've changed a tracked file since the last commit
- `git status` shows you what you've changed

Concept 4: the index

- when you say `git commit`, you're *not* committing the working tree, but rather the *index*
- the index is a "staging area" to prepare a commit
- fine-grained control over what to include in a commit, without losing your changes
- great innovation over last-generation tools like Subversion
- (example of the concept of indirection that you might remember from COMP 301/401)
- to adjust the index, see commands given by `git status` output, e.g. `git add`

Concept 5: branch/ref

- branches, also called *refs*, simply refer to a commit
- if you know C/C++, this is the same idea as a pointer
- branches are named, e.g. `master` or `feature-xyz`
- note: branches don't contain any data (apart from the name)! they merely refer to it.
- this has deep implications and is a key to understanding git
- `git branch` lists the branches/refs in your repo

Concept 6: HEAD

- the `HEAD` ref is somewhat meta: can refer to a commit (like a ref) or another ref
- indicates *what is currently checked out*, i.e. the "current" branch
- `git branch` indicates which branch `HEAD` is pointing to, if any
- `git rev-parse HEAD` will resolve HEAD to a commit address
- `git checkout` changes `HEAD`
- if `HEAD` points to a ref, `git commit` updates that ref
- the output of `git status` depends on the working tree and `HEAD`
- so same working tree can produce different `git status` output if `HEAD` is different

Concept 7: remote

- you and your teammates (and GitHub/GitLab) each have an *instance* of the same repo
- a remote is a URL pointing to another instance
- one special remote, named `origin`, is added automatically when you first clone a repo
- key idea: *collaboration works by sharing* ***commits*** *with* ***remotes***
- git is *decentralized*: its design does not require an authoritative "source of truth"
- that is, you can collaborate directly with a teammate, bypassing GitHub/GitLab
- (however, most people use GitHub/GitLab as the centralized source of truth)
- (Linux kernel development is fascinating in this regard; it's much more distributed)

Concept 8: fetching, pushing, and pulling

- `git fetch`: download data from a remote, e.g. `git fetch origin`
- `git pull`: fetch, then merge new commits from remote branch into current branch
- example: `git pull origin master` will fetch from `origin` remote then merge commits from origin's `master` branch into current branch
- `git push`: send commits to a remote and update remote refs
- example: `git push origin master` will share your current branch's commits with the origin remote and update origin's `master` branch to point to your current `HEAD` commit
- push is rejected when the pushed branch does not include the remote branch's `HEAD` commit (a "non-fast-forward merge")
- tip: set an "upstream tracking branch" for your local branches with `git push -u`, to enable `git pull` and `git push` commands with implicit remote branch targets

Outline

- ~~intro~~
- ~~foundational concepts~~
- **basic collaborative recipe**
- merge strategies
- advanced collaborative recipe
- other topics (continuing into next lecture)

Basic recipe for collaboration

```
git pull origin  master # fetch the latest code and update working tree
# <add a feature, fix a bug, or somehow update the working tree>
git status              # see a summary of what's changed
git diff [FILE]         # see details of what's changed [for a file]
git add FILE            # add a file to the index; repeat as necessary
git status              # confirm what is going to be committed
git commit              # create a new commit from the index
git push origin master  # update GitHub's master ref to include new commit
```

Outline

- ~~intro~~
- ~~foundational concepts~~
- ~~basic collaborative recipe~~
- **merge strategies**
- advanced collaborative recipe
- other topics (continuing into next lecture)

Problem with the basic recipe

What if somebody pushed a commit in between your pull and your push?

Then your push will fail.

What to do?

Aside: one-line logging

- `git log` shows the recorded history from your current `HEAD`
- (technically, displays commit metadata and follows the commit chain given by commit parent info)
- seeing the chain's structure more clearly can be useful, esp. to compare refs
- open up your `~/.gitconfig` file (i.e. the `.gitconfig` file in your home directory) in an editor
- add an `[aliases]` section if necessary
- under the `[aliases]` section, add the following line:
- `l = log --graph --decorate --oneline`
- now `git l` in your repo directory shows chain structure more clearly
- `git l <ref1> <ref2> ...` will show histories of multiple refs together

Problem: intervening teammate push

- First, say `git fetch origin` to get the latest commits
- Note that, locally, `origin/master` is the name of the `master` ref at the `origin` remote (at the time of the most recent fetch)
- Note also: `@` is a synonym for `HEAD` (which we assume points to `master`)
- Try `git l @ origin/master`; you'll see something like this (modulo commit details)

```
* 971f7a7 (origin/master) <teammate's commit>
| * 2162deb (HEAD -> master) <my commit>
|/
* f6ad223 <common parent commit>
* d41d8cd <grandparent commit>
...
```

Understanding the problem

What does this mean?

```
* 971f7a7 (origin/master) <teammate's commit>
| * 2162deb (HEAD -> master) <my commit>
|/
* f6ad223 <common parent commit>
* d41d8cd <grandparent commit>
...
```

- we have *two competing theories on what the commit after* `f6ad223` *is*
- in a word, the histories have *diverged*
- typically, we consider `origin` as the source of truth, so `971f7a7` "wins" (it was the first commit to hit `origin/master`)
- so our goal is change our local repo to reconcile the divergence

Reconciling histories 1: merging

- `git merge` incorporates changes from another branch into the current one
- so `git merge origin/master` would meet our goal
- how does this work?
- if the other commit contains our current commit in its history, `git merge` will merely update our ref to point to that commit
- if our `HEAD` commit contains the other one, no-op
- otherwise, create a new commit with **two** parents
- (remember, a commit is just a snapshot)
- in many cases, git can automatically reconcile the histories
- otherwise, you'll need to manually intervene (more on "merge conflicts" later)

Merging results

Result of `git merge origin/master` then `git l @ origin/master`:

```
*   76f2fff (HEAD -> master) Merge remote-tracking branch 'origin/master' in
|\
| * 971f7a7 (origin/master) <teammate's commit>
* | 2162deb <my commit>
|/
* f6ad223 <common parent commit>
* d41d8cd <grandparent commit>
...
```

- note the ascii-art graph showing parallel commits followed by a merge
- this is a *non-linear history*, which we'll discuss later
- now the merge commit includes `origin/master`'s commit in its history, so a push will succeed

Reconciling histories 2: rebase

- a second strategy is to "rebase" your commit(s) "atop" or "onto" `origin/master`

```
* 971f7a7 (origin/master) <teammate's commit>
| * 2162deb (HEAD -> master) <my commit>
|/
* f6ad223 <common parent commit>
* d41d8cd <grandparent commit>
...
```

1. push the changes represented by f6ad223 -> 2162deb on a stack
2. repeat until "merge base" found (no repeats in this case)
3. advance the local ref until we reach `origin/master` (1 advance in this case)
4. replay the changes from the stack to create new commits

Rebasing results

Result of `git rebase origin/master` then `git l @ origin/master`:

```
* 4e286c6 (HEAD -> master) <my replayed commit>
* 971f7a7 (origin/master) <teammate's commit>
* f6ad223 <common parent commit>
* d41d8cd <grandparent commit>
...
```

- the changes in `4e286c6` are the same as in my first commit (modulo merge conflicts)
- my rebased commit includes `origin/master`'s commit in its history, so a push will succeed

Merge vs. rebase

key collaborative decision: whether to merge or to rebase

- consider discussing this with your team and adding decision to your team rules
- rebasing is a little harder to understand than merging, but produces linear histories
- rebasing involves rewriting history
- philosophical divide: git as *record of what happened* or git as *history (an ordered telling thereof)*
- we (should) take time to revise code for legibility
- this takes time, but is worth it, because code tends to be read (by humans) much more than it's written (by humans)
- my opinion: we should take the same care to make our histories readable (i.e. rebase is preferred)
- also, linear histories have other benefits

Tip: the longer the time gap, the more painful the reconciliation (whether merge or rebase), so merge/rebase often, not just as a final step.

Outline

- ~~intro~~
- ~~foundational concepts~~
- ~~basic collaborative recipe~~
- ~~merge strategies~~
- **advanced collaborative recipe**
- other topics (continuing into next lecture)