Git, part 2

UNC COMP 523

Mon Sep 28, 2020

Prof. Jeff Terrell

Announcements

- music: "I Remember" by deadmau5

Assignment checkin

- Due this week:
  - A6: Application Architecture ☕☕☕☕
  - A7: Architecture Diagram ☕
- Due in two weeks: A8: Walking Skeleton ☕☕☕☕☕
  - disastrous naive approach: wait until the last week (or day) to integrate everything
  - walking skeleton requires you to *integrate first*
  - skeleton: find the simplest set of features that could show the pieces working together, and ignore the rest
  - walking: it should actually work and be deployed
  - remember that the App Lab is ready to help
  - don't wait until week after next to start!

A motivating note

Why do we use version control systems at all? Why not just use a shared folder?

- Seeing past versions of the code and what exactly changed is very useful.
- You can rollback to an old commit.
- Being able to annotate a commit can convey useful info to collaborators.
- How can you be confident you're not clobbering your teammate's changes?
  - By agreements about who's editing a given file at the moment?
  - By comparing last-changed timestamps of files?
  - (These are tedious and error-prone approaches.)

Conceptual review

- Recall concepts like commit, ref/branch, remote, and fetching/pushing/pulling
- "I was confused about what it means to have a branch checked out."
  - this just means that the `HEAD` meta-ref currently points to that branch
- "I don't understand a good flow for creating different branches."
  - we'll talk about that this time
- "What's the difference between a ref and a commit?"
  - the commit is a snapshot (with metadata), and the ref is a pointer to a commit
- "In what situation would you want to fetch and not pull?"
  - if you want more control over the automatic merge, e.g. if you want to rebase instead of merge
- "What is the difference between downloading and cloning?"
  - cloning is the initial download; fetching and pulling are downloads that happen after the initial clone

Outline

- ~~review from last time~~
- merge strategies, continued
- advanced collaborative recipe
- other tips

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
```

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?

Problem: intervening teammate push

- Try `git l @ origin/master`; you'll see something like the below
- The histories have *diverged* and have different opinions about the commit after `f6ad223`
- Goal of a merge strategy: incorporate `971f7a7`'s changes into local history
- Both a `merge` and a `rebase` get us to the exact same commit snapshot

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

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. identify the `merge base`, i.e. the nearest common ancestor, `f6ad223` here
2. for each commit from `HEAD` to the merge base, push the changes of the commit on a stack (1x 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 parallel commits

Alternate rebase example

- See first section of Pro Git (book), ch. 3.6

Merging vs. rebasing

`git merge origin/master`:

```
* 76f2fff (HEAD -> master) <merge>
|\
| * 971f7a7 (origin/master) <other>
* | 2162deb <mine>
|/
* f6ad223 <common parent commit>
* d41d8cd <grandparent commit>
...
```

`git rebase origin/master`:

```
* 4e286c6 (HEAD -> master) <mine>
* 971f7a7 (origin/master) <other>
* f6ad223 <common parent commit>
* d41d8cd <grandparent commit>
...
```

- 5 commits vs. 4
- non-linear history vs. linear

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

Caution

Do not rebase commits that exist outside your repository and that people may have based work on.

Outline

- ~~review from last time~~
- ~~merge strategies, continued~~
- advanced collaborative recipe
- other tips

Advanced collaborative recipe

My recommendation: use **trunk-based development** (TBD), with a single long-lived branch, the trunk, typically `master`.

1. when starting a new feature, branch from (latest) master
2. add commits to your feature branch
3. to merge, rebase atop latest master and fast-forward master (*could* do a merge instead)
4. push master to `origin`
5. deploy latest `master` from `origin` (if continuous deployment not enabled)

(This is almost identical to the "GitHub Flow" model, and very different than the "git flow" model.)

Freedom to rewrite history

> Do not rebase commits that exist outside your repository and that people may have based work on.

- Among team, I recommend treating a feature branch as owned by the developer working on that branch by default.
- Dev should push work-in-progress commits to `origin/feature-XYZ` branch to avoid losing them in case of computer failure.
- Dev should also feel free to rebase atop latest master (see CI below) and otherwise rewrite history *on that branch*.
- However, no dev should rewrite history for `master` or another dev's branch except by mutual consent.

Code reviews

- Code reviews are not an opportunity for shaming teammates
- Rather, can help share knowledge among team (about programming in general and this project/codebase in particular)
- Remember that Heroku review apps let you see the proposed code in action
- Recommendation: rebase before opening the pull request
- Pro tip: use fine-grained, single-purpose commits *before* review, then squash them together *after* review
  - reviewing is easier with fine-grained commits
  - squashing means to collapse multiple commits into 1
  - squashing can be friendlier for forensic work from the trunk (more on this later)
  - squashing is possible with an "interactive rebase", i.e. `rebase -i`

Merging considerations

- test your rebased branch before pushing (esp. if there was a conflict while rebasing)
- if squashing, to rebase pull request and get it to be properly marked as "merged" instead of "closed":
  - rebase atop latest master locally
  - push feature branch
  - squash locally
  - push feature branch
  - merge feature into master locally
  - push master branch
  - delete feature branch, locally and remotely

Outline

- ~~review from last time~~
- ~~merge strategies, continued~~
- ~~advanced collaborative recipe~~
- other tips

Continuous integration

- Note: the longer the time gap, the more painful the reconciliation (whether merge or rebase)
- Tip: merge/rebase often, not just as a final step.
- This is called *continuous integration* (CI), i.e. keeping your feature branch *integrated* with the trunk
- Consider rebasing every day before starting development
- If there's a conflict, the info needed for resolution is near top of mind for you and your teammate
- Helpful to have a "CI server" that integrates with `origin` and runs tests for you on new commits
- (Plenty of free CI services out there for public repos.)
- CI is a great pairing with (and arguably a prerequisite for) CD

Stashing

- Say you want to integrate but your tree is dirty
- Useful tool: `git stash`
- Stows your working tree as a stash, which you can restore later
- So stash, integrate/rebase/whatever, then "pop" your stash

Leveraging history

- use `git log FILE` to see which commits affected `FILE`
- `git log` has LOTS of options; you can (for example):
  - only print commits from certain authors
  - only print commits whose messages match a given string
  - only print commits whose diffs match a certain string
- use `git blame FILE` to see the most recent commit to affect each line of `FILE`
- use `git show COMMIT:FILE` to see the version of FILE at the given COMMIT (works with refs too)
- ...then diff it with the current file: `git show COMMIT:FILE | diff - FILE`
- to refer to 3rd parent commit, you can use `HEAD^^^` or `HEAD~3` or `@^^^` or `@~3`; see `git rev-parse` for details

Commit messages

- History is only as useful as you make it
- The interpretive layer can be *very* valuable (i.e. commit messages)
- I recommend thinking of commit messages as email messages to your teammates or to a future developer of this codebase
- First line is the subject line: imperative tense, ideally <= 50 characters
- Then blank line, then email body: paragraphs, lists, whatever
- Especially valuable: what alternatives did you consider? Why did you go with that approach?
- *This can be much more valuable than comments because*
  - *they're attached to a version and*
  - *they don't get in the way of editing code, but are there on demand if anybody needs them.*

Merge conflicts

- If git can't automatically merge two histories, it relies on you to do it.
- This is true whether you use merge or rebase as your merge strategy.
- Useful reading to understand merge conflicts, from the Pro Git book:
  - Basic Merge Conflicts
  - Advanced Merging
- Best tip I can think to give you: think carefully about what a merge conflict is, in order to understand the output you see and choices you can make

Bisecting

- If you encounter a bug that you don't think you caused, first stash your dirty changes and test your belief
- If you have several commits unique to your feature branch, you can also test your local `master` branch
- If you still see the error, check out something older, maybe `@~20` or something
- Once you identify a working commit, you can `bisect` to identify when the bug was introduced
- This does a *binary search* on the commits, so O(lg(N)) steps, where N = # of commits
- At each step, bisect checks out a commit and asks you if it's bad or good
- Can also be scripted; e.g. run tests and determine bad/good by exit status of tests
- This is a good reason to have a linear history!

Public key authentication

- Generate a keypair
- Give GitHub/GitLab your public key
- Ensure your private key successfully authenticates you to github.com/gitlab.com
- This enables "passwordless" (but still secure) authentication
- For best security, use a passphrase-protected private key but also use a key agent to cache passwords

The reflog

- This is often considered deep magic for wizards only
- But it's not that hard
- git maintains a log of which commit a ref points to
- You can see the log with `git reflog <branch>` to fix some big mistakes
- Go ahead, be the hero

Other tips

- If the options to `git reset` are confusing, read Reset Demystified in the Pro Git book.
- Try Magit, a git porcelain that is so good, it might compel you to switch to an ancient text editor :-)
- With Magit or another "porcelain" (i.e. git frontend), try your hand at selective commits and interactive rebases
- If you are experimenting with your local git repo, try creating a branch with your pre-experimentation state to avoid losing it

Quiz 3