GIT & GITHUB
Basics of Distributed Version Control

Charles Liu
Overview

0. What is Git? Installation and setup
1. Introduction to version control; basic workflow in Git
2. Branching, merging, and rebasing
3. Working with remotes and Github
What is Git?

- A *distributed* version control system
- A few use cases:
  - Keep a history of previous versions
  - Develop simultaneously on different branches
    - Easily try out new features, integrate them into production or throw them out
  - Collaborate with other developers
    - “Push” and “pull” code from hosted repositories such as Github
Key improvements

- A *distributed* version control system
  - Everyone can act as the “server”
  - Everyone mirrors the entire repository instead of simply checking out the latest version of the code (unlike svn)

- Many local operations
  - Cheap to create new branches, merge, etc.
  - Speed increases over non-distributed systems like svn
Installation and setup

- **Linux**: `apt-get install git-core`
  - Git bash
- Eclipse extensions such as eGit
First time setup

- `git config --global user.name “Charles Liu”`
- `git config --global user.email “cliu2014@mit.edu”`

- Line breaks (`\r\n` in Windows vs. `\n` in Mac/Linux)
  - Mac/Linux: `git config --global core.autocrlf input`
  - Windows: `git config --global core.autocrlf true`
Use case #1: history of versions

- Basic workflow
- Adding and committing files
- The git log
- The staging area
- Removing files
- Viewing diffs of files
Big ideas

- Snapshots, not deltas
- Everything is confined to the .git directory
- Most operations are safe – they only add data
  - We’ll talk about two commands that are not safe today
- 3 possible states for a file
  - Changed
  - Staged
  - Committed
Basic workflow

- **git init** – create git project in existing directory
  - Make Git start to “watch” for changes in the directory

- The basic workflow:
Basic workflow

- Add files to be committed with `git add <filename>`
  - Puts the file in the “staging area”
- Create a commit (a “snapshot”) of added files with `git commit`, followed by a commit message
- Use `git status` to see the current status of your working tree
The git status output

```
cliu:git charlesliu$ git status
# On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
      new file:   b

Changed but not updated:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:  a

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    c
```
The git log output

```
cliu:git charlesliu$ git log
commit d2fb80a79c3188da6fdflf5b578dde68603feed0
Author: Charles Liu <cliu2014@mit.edu>
Date:   Tue Jan 8 01:37:15 2013 -0500

    Commit #2

    You can enter a longer commit message here. Try to keep the first line short

commit 336993e77c782190506c00c0857e757460f45c76
Author: Charles Liu <cliu2014@mit.edu>
Date:   Tue Jan 8 01:29:14 2013 -0500

    initial commit
```
The staging area

- `git add` takes the snapshot of the file that will be committed → you can change the file after adding it.

```
cliu:git charlesliu$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
###
   new file:   c

Changed but not updated:
   (use "git add <file>..." to update what will be committed)
   (use "git checkout -- <file>..." to discard changes in working directory)
###
   modified:   c
```
The staging area

- To unstage a file, but retain your changes in the working tree:
  - `git reset HEAD <filename>`

- To discard current changes in the working tree, and make it look like the last commit:
  - `git checkout -- <filename>`

- Be careful! You will lose your changes and not get them back!
Removing a file

- To remove a file from the working tree and in the next commit, simply `git rm <filename>`
- To remove it from the next commit, but keep the file in the working tree, do `git rm --cached <filename>`
Viewing diffs of files

```
HEAD

- git diff --cached

Index

- git diff

Working tree

- git diff HEAD
```
Viewing diffs of files

```bash
cliu:git charlesliu$ git diff HEAD
diff --git a/d b/d
index 2d9d46e..8163fc1 100644
--- a/d
+++ b/d
@@ -1,5 +1,5 @@
    blah blah blah this is version 1
-blah blah blah this is line 2 of version 1
+this is line 2 of version 1
   this is line 3
-and 4
  6.470 is awesome
+added new line here
```
Use case #2: branching

- What is a branch?
- Branching commands
- The HEAD pointer
- Basics of merging
- Basics of rebasing
- Aside: the git reset command
What is a branch?

- Visualize a project’s development as a “linked list” of commits.
- When a development track splits, a new branch is created.
- In Git, branches are actually just a pointer to these commits.
Branching commands

- List all branches in the project – `git branch`
- Create a new branch – `git branch <branchname>`
- Switch to a branch – `git checkout <branchname>`
- Create and immediately switch – `git checkout –b <branchname>`
- Delete a branch – `git branch –d <branchname>`
Stashing

- Working tree must be clean when switching branches
- Stash changes that you don’t want to commit at that time – `git stash`
  - Puts a stash onto the stack
- Later, apply the most recent stashed changes and remove that stash – `git stash pop`
- Also, `git stash apply` to apply changes but not remove the stash from the stack
The HEAD pointer

- Recall: all branches simply a pointer to a commit
- HEAD: special pointer to the current branch, moves around as you switch branches
Merging
Merging

1. git checkout master
Merging

1. `git checkout master`
2. `git merge iss53`
Merging

1. `git checkout master`
2. `git merge iss53`

Note: merge creates a **new commit** with 2 parents!
Merge conflicts

- Sometimes, two branches will edit the same piece of code in different ways.
- Must resolve the conflict manually, then add the conflicting files and explicitly commit.
- Demo
Rebasing

- Instead of a merge, which creates a new commit originating from both branches, a rebase takes the contents of one branch after the “split” and moves them to the end of the other branch.

- The command `git rebase <basebranch>` takes your currently checked out branch and replays the diffs on top of basebranch.
Rebasing

1. `git checkout server`
Rebasing

1. `git checkout server`
2. `git rebase master`
1. `git checkout server`
2. `git rebase master`
Why rebase?

- Creates a linear history; often cleaner and easier to read.
- But... **DO. NOT. EVER.** rebase anything that has already been pushed to a repo someone else has access to.
  - Rebasing removes commits and writes new ones; but someone else might have already based their work off your old commits!
An aside...the git reset command

- 3 versions...and often the source of much confusion!
  - `git reset --soft <commit / pointer to commit>`
  - `git reset --mixed <commit / pointer to commit>` (or simply `git reset`)
  - `git reset --hard <commit / pointer to commit>`

- Reset proceeds in 3 steps:
  1. Move the HEAD pointer
  2. Update the index/staging area to the new contents of HEAD
  3. Update the working directory
3 steps to reset

1. Move the HEAD pointer — soft stops here.
2. Update the index/staging area to the new contents of HEAD — mixed stops here.
3. Update the working directory — hard stops here

Note: reset --hard overwrites the working directory. This is another command that can potentially cause loss of data!
Use case #3: collaboration

- Creating a repo on Github
- Remotes
- Remote-tracking branches
- Push, fetch, and pull
- The git clone command
Remotes

- A target computer that has Git repos that you can access
  - Via http(s), ssh, or git protocols
- `git remote add <remotename> <remoteaddress>`
- `git remote -v` (view remotes)
- `git remote rm <remotename>`
- Often, with one remote, we name it “origin”
Pushing and fetching

- **git push <remotename> <branchname>** sends your code in the branch up to the remote
  - Often just git push: depends on settings but often equivalent to git push origin master

- **git fetch <remotename>**
Remote tracking branches

- When you do `git fetch`, you don’t immediately see the changes. Why?
- Changes are fetched to a “remote tracking branch”
  - Branches associated with a remote, but treat them like a local branch
  - Can merge with your current master (`git checkout master; git merge origin/master`)
  - Even better…rebase
Remote tracking branches
Remote tracking branches
Github demo
In summary...

- **Basic workflow in git**
  - Adding, committing, viewing diffs

- **Branches**
  - The HEAD pointer, merging, and rebasing

- **Remotes**
  - Pushing and fetching; quick introduction to Github
Lots of other topics

- Tags and version numbers
- Interactive rebase: squashing and amending commits
- Relative pointers from HEAD (e.g. HEAD^^, HEAD~3)
- Submodules
- Using your own server as a git server (bare repos)
- Git as a filesystem (git grep, git ls-files, etc.)
- GUIs to view trees and graphical merge tools
- …more!
For more information

- The book *Pro Git* (which I based much of this presentation on), available for free!
  - [https://github.s3.amazonaws.com/media/progit.en.pdf](https://github.s3.amazonaws.com/media/progit.en.pdf)
  - Covered Chapters 1-3 in detail, very simple ideas from Chapters 4-6
- Git documentation: do `git help <commandname>`
- Google, StackOverflow, etc.