Tutorial on Git

Distributed Version Control and Development Workflow

Jed Brown jedbrown@mcs.anl.gov

Argonne National Lab, 2014-08-06

This talk: http://59A2.org/files/20140806-GitTutorial.pdf
Distributed Version Control

- Directed Acyclic Graph (DAG) history
  - Labels and namespaces
  - Branch structure to organize workflow
  - Flexible, asynchronous reviewing and quality control
  - Powerful merging
- Work with clones, each is equivalent and fully-functional
  - Social conventions for which is canonical
  - Each has its own branch namespace
- Provenance and auditability via cryptographic hashes
- Operations are local (and fast)
Is linear history good?

- Testing and review? Bugs and fixes are spread out.
- When is a feature complete?

- Merges contain completed features.
- Asynchronous testing and review.
Labeling the DAG

- **HEAD:** cursor naming “current branch” or tag/commit
  - If a branch (usually), committing will advance that branch
  - Implicit reference for many commands (like `git diff`)
- **Branches:** lightweight labels that move with cursor (HEAD) and push/pull
- **Tags:** stationary, can be signed
- **Hashes:** every object is uniquely identifiable by a SHA1 hash

http://eagain.net/articles/
git-for-computer-scientists/
## Basic DAG commands

Git is fundamentally a tool for incrementally updating and analyzing the labeled DAG.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>commit</td>
<td>Create a new node in DAG and advance HEAD</td>
</tr>
<tr>
<td>checkout \textit{name}</td>
<td>Move HEAD to specified branch and update working tree to match</td>
</tr>
<tr>
<td>branch \textit{name}</td>
<td>Create new branch label</td>
</tr>
<tr>
<td>tag \textit{name}</td>
<td>Create (stationary) tag on commit indicated by HEAD</td>
</tr>
<tr>
<td>merge \textit{commitish}</td>
<td>Merge specified branch/tag/commit into current branch, creating new commit and advancing HEAD</td>
</tr>
<tr>
<td>log</td>
<td>Ancestors of HEAD</td>
</tr>
<tr>
<td>log --first-parent</td>
<td>Ancestors of HEAD following only first parent of merges</td>
</tr>
<tr>
<td>log -- \textit{path}</td>
<td>Only those that modify path</td>
</tr>
</tbody>
</table>
Hands-on: configuration

- `git config --global user.name 'Your Name'`
- `git config --global user.email your@email.com`
- `git config --global color.ui auto`
- Optional: `https://raw.githubusercontent.com/git/git/master/contrib/completion/git-prompt.sh`
- Optional: `https://raw.githubusercontent.com/git/git/master/contrib/completion/git-completion.bash`
- `git config --global merge.log true`
Hands-on: clone a repository

- `git clone`  
  https://bitbucket.org/jedbrown/git-tutorial
- `cd git-tutorial`
- **Compare the history**
  - `git log --graph`
  - `git checkout linear && git log --graph`
  - `git checkout integration`
  - `git log origin/a/dev..`
The staging area (or “index”)

- Sometimes we don’t want to commit everything
- It’s nice to incrementally resolve conflicts, then not be shown again
- `git add`, `git rm`, and others need to be logged somehow
- Fast and useful primitive for building tools (in Git and externally)

http://ndpsoftware.com/git-cheatsheet.html
Remotes

- Remotes are named and cached remote repositories
  - more commands can complete locally
- Cache is updated by `git fetch` and similar
- Private namespace for branches (prevents conflicts)
- “origin” is created by default by `git clone`
- `git remote add gh`  
  `git@github.com:jedbrown/git-tutorial`
Hands-on: make a commit to show you were here

- git checkout attendees
- echo MCS > Jed_Brown
- git add Jed_Brown
- git commit -m"I’m at the Git tutorial"
- Submit changes
  - git format-patch origin/attendees
  - and email patch to jed-tutorial@jedbrown.org
  - Fork repository on Bitbucket:
    https://bitbucket.org/jedbrown/git-tutorial
    - git remote add yourname
    - https://yourname@bitbucket.org/yourname/git-tutorial
    - git push yourname attendees
    - Make a pull request to my repository
  - Fork repository on GitHub:
    https://github.com/jedbrown/git-tutorial
**Hands-on: working with branches**

In your browser:
https://pcottle.github.io/learnGitBranching/

- Spend a few minutes with the branching and merging examples
- Advanced commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset <em>path</em></td>
<td>set staging area to match <em>path</em> in HEAD</td>
</tr>
<tr>
<td>rebase <em>commit</em></td>
<td>replay commits in <code>${commit}..</code> on top of <code>${commit}</code>, advancing current branch (old commits will be gc’d if not referenced)</td>
</tr>
<tr>
<td>rebase --abort</td>
<td>go back to state before starting rebase</td>
</tr>
<tr>
<td>rebase -i HEAD~3</td>
<td>interactively amend last three commits</td>
</tr>
<tr>
<td>cherry-pick <em>commit</em></td>
<td>make commit on current branch, effecting the same change as <code>${commit}</code></td>
</tr>
<tr>
<td>reflog</td>
<td>everywhere that HEAD has been in last 90 days (good to recover after a mistake)</td>
</tr>
<tr>
<td>gitk</td>
<td>graphical history visualization</td>
</tr>
<tr>
<td>git citool</td>
<td>graphical incremental commit tool</td>
</tr>
</tbody>
</table>
Workflow ideals

- ‘master’ is always stable and ready to release
- features are complete and tested before appearing in ’master’
- commits are minimal logically coherent, reviewable, and testable units
- related commits go together so as to be reviewable and debuggable by specialist
- new development is not disrupted by others’ features and bugs
- rapid collaboration between developers possible
- `git log --first-parent maint..master` reads like a changelog
- bugs can be fixed once and anyone that needs the fix can obtain it without side-effects
Simplified git workflows

- v1.0
- v2.0
- v2.1
- v3.0

- latest feature release
- merges to be discarded when 'next' is rewound at next release
- reviewed, thought to be complete
- test periods overlap
- 'graduation'
- merged with evidence of stability
- typical feature branch
- merge history (not first-parent)
- merges to be discarded when 'next' is rewound at next release
- merge in first-parent history of 'master' or 'maint' (approximate "changelog")
- merge to branch 'next' (discarded after next major release)
- commit in feature branch (feature branches usually start from 'master')
- commit in bug-fix branch (bug-fix branches usually start from 'maint' or earlier)

- 'master' contains 'maint'
- 'next' contains 'master'
- 'maint' contains latest feature release
- 'master' contains latest feature release
- 'next' contains 'master'
- 'next' contains 'master'
- 'master' is a stable base for new features, always ready to release
- upcoming feature release will be tagged on 'master'
- after each release, the old 'next' is discarded and recreated
- testing and "eager" users, bugs here only affect integration, not development
- feature did not graduate for v2.0
- risk feature
- bug fix for release
- reviewed, thought to be complete
- bug fixes tested like features
- fix issue found by external client
- review pull req
- testing & users
- 'next' contains 'master'
- v1.0
- v2.0
- v2.1
- maint
- master
- next

- time
- first-parent history of branch
Best practices

- Every branch has a purpose
- Distinguish integration branches from topic branches
- Do all development in topic branches
  - `git checkout -b my/feature-branch master`
- Namespace your branches if working on a shared repository
- Merge integration branches “forward”
  - `maint-1 → maint → master → next`
  - `git checkout -b my/bugfix-branch maint-1`
- Write clear commit messages for reviewers and people trying to debug your code
- Avoid excessive merging from upstream
  - Always write a clear commit message explaining what is being merged and why
- Always merge topic branches as non-fast-forward (`merge --no-ff`)
- Gracefully retry if you lose a race to shared integration branch
  - This maximizes utility of `--first-parent` history
Outlook

- `git init` is only 3 more characters than `mkdir`
- Unlimited free private repositories at https://bitbucket.org
- Set up ssh keys so you don't have to type passwords
- Always start work in a new topic branch
  - Easy to checkpoint and context switch away
  - Can rebase or merge to existing branch if it makes sense
- You can clean up from almost anything, `reflog` can help
- Do not rebase commits that have been published
- Commit often, then organize with `git rebase -i`
  - See also `rebase.autosquash` and `git commit -fixup`
- Learn to summarize and search history
- Check out merge strategies `git merge --help`
- Git can remember conflict resolutions `rerere.enabled=true`