Tutorial on Git

Distributed Version Control and Development Workflow

Jed Brown jedbrown@mcs.anl.gov
Thanks to Doug Jacobsen, Jeff Johnson, James Foucar, Susannah Burrows, Rob Jacob, Andy Salinger

ACME All-Hands Meeting, 2015-05-06

This talk: http://59A2.org/files/20150506-GitTutorial.pdf
Distributed Version Control

- Directed Acyclic Graph (DAG) history
  - Every commit has one or more ancestors
  - 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 (if any) 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.

Output from `gitk`
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 <em>name</em></td>
<td>move HEAD to specified branch and update working tree to match</td>
</tr>
<tr>
<td>branch <em>name</em></td>
<td>create new branch label</td>
</tr>
<tr>
<td>tag <em>name</em></td>
<td>create (stationary) tag on commit indicated by HEAD</td>
</tr>
<tr>
<td>merge <em>commitish</em></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 -- <em>path</em></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.name@lab.gov`
- `git config --global push.default nothing`
- `git config --global color.ui auto`


Optional: [https://raw.githubusercontent.com/git/git/master/contrib/completion/git-completion.bash](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 git@github.com:ACME-Climate/git-tutorial`
or `https://YourName@github.com/ACME-Climate/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..`

```
11:30 jedbatura5 integration= ~/acme/git-tutorial$ git log --graph --oneline --decorate
* 143f6ab (HEAD, origin/integration, integration) Merge branch 'b/dev' into integration
  |
  | * 2fafd4f (origin/b/dev, b/dev) Merge branch 'c/bug' into b/dev
  |
  | * 68c03ad (origin/c/bug, c/bug) B: make bug
  | * d84a4ab B: ready to test
  | * e0f0d86 B: continue work on my own
  |
  * 9d94b78 B: new table
  * 4fa51c1 Merge branch 'a/dev' into integration
  |
  | * 18dac63 (origin/a/dev, a/dev) A: finish A
  | * db0cf91 A: incremental work
  | * 5afed62 A: start working
  |
* c887c4e Initial project
```
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)
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`
Hands-on: make a commit to show you were here

- `git checkout attendees`
- `git checkout -b jed/attendee`
- `echo 'Argonne SEG' > Jed_Brown`
- `git add Jed_Brown`
- `git commit -m "I'm at the Git tutorial"
- Submit changes
  - `git push -u acme-ssh jed/attendee`
- Turn to your neighbor and rock-paper-scissors to elect an integrator.
  - Integrator: review your neighbor’s branch
  - `git checkout neighbor/attendee && git log -p`
  - If it looks good: `git checkout me/attendee && git merge neighbor/attendee`
  - Make a pull request to 'master' at https://github.com/ACME-Climate/git-tutorial
## 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 path</td>
<td>Set staging area to match path in HEAD</td>
</tr>
<tr>
<td>rebase commit</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 commit</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>
Git Workflow Objectives

- ‘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 (7)

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

- feature release
- latest feature
- branch
- "graduation"
- "maintenance"
- "master"
- "maint"
- "next"

- merge history (not first-parent)
- first-parent history of branch

- 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)

- testing & users
- bug fixes tested like features
- reviewed, thought to be complete
- fix issue found by external client

- testing and “eager” users, bugs here only affect integration, not development
- after each release, the old ‘next’ is discarded and recreated
- upcoming feature release will be tagged on ‘master’

- ‘master’ is a stable base for new features, always ready to release
- ‘maint’ contains ‘master’
- ‘master’ contains ‘maint’
- ‘next’ contains ‘master’

- risky feature
- bug fix for release

- latest feature release
- ‘maint’ contains latest feature release

- review pull req
- typical feature branch

- time

- review, thought complete
- test periods overlap
- merges to be discarded when ‘next’ is rewound at next release

- v3.0
- upcoming feature release
- will be tagged on ‘master’
- master
- maint
- next

- latest feature release
- feature did not graduate for v2.0

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

- time

- first-parent history of branch
- merge history (not first-parent)
ACME Best Practices

- Every branch has a purpose
- Distinguish integration branches from topic branches
- Do all development in topic branches
  - `git checkout -b my/component/short-feature-description master`
- Namespace your branches
- 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

Integrators

- Merge integration branches “forward”
  - `maint → master → next`
  - `git checkout -b my/bugfix-branch maint`
- Always merge topics 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`
- 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
- Commit often, then organize with `git rebase -i`
  - See also `rebase.autosquash` and `git commit --fixup`
  - Do not rebase commits that have been published
- You can clean up from almost anything, `reflog` can help
- Learn to summarize and search history
- Check out merge strategies `git merge --help`
- Git can remember conflict resolutions `rerere.enabled=true`
- [https://acme-climate.atlassian.net/wiki/display/D)ocs/Repository+and+Development](https://acme-climate.atlassian.net/wiki/display/D)ocs/Repository+and+Development