A Git Tutorial: More Topics

Patrick Sanan

https://bitbucket.org/psanan/gittutorial

Who is this for?

More Helpful Tools

Helper Scripts

Dangerous Git
    Rewriting Branch Histories
    Deleting Things

Section 1

Who is this for?

Assumed Audience

Assumptions about the audience:

- You have seen and understood the first part of this tutorial
- You are interested in some additional information about git

Section 2

More Helpful Tools

Helpful git commands and options

- `git status`
  - `git status --short`
  - `git status --untracked=no`
- `git log`
  - `git log -10`
  - `git log --graph`
  - `git log --pretty=oneline`
  - `git log --pretty=format"%h - %an, %ar : %s"`
  - For more, see "Pretty Formats" in `git help log`
- `git diff`
  - `git diff file.ext`
  - `git diff a7869c2`
  - `git diff a7869c2 file.ext`
  - `git diff --staged`
  - `git difftool` [one of the few times that a GUI is perhaps worth it]

.gitignore

- You often want to mark certain files as ineligible for commits
- Temporary files (`*.DS_Store`,`.*.swp`)
- Objects and executables
- Configuration files
- Large files
- Git will ignore any files listed in `.gitignore`
- Subdirectories can have their own `.gitignore`
- It has a sophisticated set of syntax options

```
git help gitignore
```

Git Bisect

- Having a (quickly verified) “good” state and a (quickly verified) “bad” state, with respect to a given problem you would like to repair, is a position of enormous strength.
- You can find out which is the first “bad” commit by running at most $\lceil \log_2 N \rceil + 1$ tests, where N is the number of commit between a known “good” commit and a known “bad” commit.
- `git bisect` allows specifiying commits as good or bad, until the first bad commit is identified

```
git bisect start
git bisect bad
git bisect good 28490a89
...
git bisect good
...
git bisect bad
...
eb694f0286c957aa9e303b194cad8b622547e825 is the first bad commit
...
git bisect reset
```

Git Bisect, continued

- Also try `git bisect log`
- You can move `HEAD` around during a bisection, say if one commit represents a non-working state.

```
git checkout HEAD~2
```

- `git bisect skip` attempts to do this automatically (choosing a nearby commit)
- You can limit to commits which affect certain paths

```
git bisect start -- some/path some/other/path
```

- You can even automate the tests

```
git bisect run test_exec args
```

Git Blame

- Sometimes you want to know who introduced a change.
- Git blame tells you who last touched each line of a file

```
git blame Lecture11.tex
a3d0cd08 (Patrick Sanan 2015-11-11 14:18:15 +0100 1) \input{"../templates/Lecture.tex.
     inc"}
a3d0cd08 (Patrick Sanan 2015-11-11 14:18:15 +0100 2)
35459acc (Patrick Sanan 2015-11-11 16:40:18 +0100 3) \title[CEFCS Week 11]{Software
     Engineering for Computational Science: Week 11}
...
```

- Combined with regression tests that give line numbers with errors, this is powerful.

Section 3

Helper Scripts

Helpful Scripts to use with Git

The git repository for git has extras:
`https://github.com/git/git/tree/master/contrib/`

- It is very helpful to add git information to your git prompt
  - `completion/git-prompt.sh`
  - `source` this in your login scripts.

```
source $PDSRC_ROOT/git-prompt.sh
GIT_PS1_SHOWUPSTREAM=true
GIT_PS1_SHOWDIRTYSTATE=true
GIT_PS1_SHOWSTASHSTATE=true
PS1='\[\e[1;31m\][\[\e[0;33m\]\H: \[\e[1;31m\]\W\[\e
    [0;36m\]$(__git_ps1 " (%s)")\[\e[1;31m\]]\$\[\e[0m
    \] '
```

  - Don't forget to include `\[ \]` around non-printing characters in your `PS1` string.
- Another time-saver is git autocomplete
  - `completion/git-completion.bash`

Section 4

Dangerous Git

Dangerous Git

What makes some operations dangerous?

- Might not be undoable at all (deleting untracked files)
- Might not be undoable trivially (complicated rebasing)
- Might actually destroy information (prune, gc)
- Might rewrite history, hence require a forced push to a remote
- Might require you to remember something (git stash)

However, you will need these dangerous operations, so it is a great idea to learn about them now, when you are calm and rested. Know what they do to the data git stores, and you won't ruin anything by blindly entering commands.

- As a rule of thumb, be careful with operations which require `--hard` or `-f` flags, and check the documentation for `-n` or `--dry-run` flags which can let you see what damage you might do.

Dangerous Git Commands: Overview

- Fixing your last commit:
  - git commit –amend
  - git reset
- Cleaning up
  - git clean
  - git branch -D
- Moving commits around
  - git rebase
  - git cherry-pick
- Rewriting history
  - git rebase -i
- ...

- Taking a detour
    - git stash
- Overwriting things on a remote
    - git push -f
- Reversing a commit
    - git revert
- Data management
    - git filter-branch
    - git prune
    - git gc

Subsection 1

Rewriting Branch Histories

Amending

- `git commit --amend` allows you to change the last commit. This changes its name (since it now describes a different state), so avoid doing this if you have pushed to a remote.
- Useful is `git commit --amend --no-edit` if you forgot to add a file, made a typo, or otherwise don't want to edit the commit message.

Resetting

- Recall from earlier that git stores many different versions of your project
  - Many snapshots described by commits. The latest of these is pointed to (via a branch) by `HEAD`
  - A state on your working directory
  - A state in the staging area (the *index*)
- `git checkout` takes the state defined by a commit (or a branch or other ref pointing there), makes `point` there, and copies it to the working directory, clearing the staging area (making the index be identical to the state in `HEAD`)

Resetting, Cont.

- `git reset` gives you some more options, by making 1-3 changes.
- `git reset --soft` Simply move the branch pointed to by `HEAD`. The index (staging area) and the working directory are left the same.
- Unless you specify `--soft`, the index is also updated to match what `HEAD` now points to. The information about what you have staged is **lost**.
- `git reset --hard` additionally updates the working directory to the same stage as pointed to by `HEAD`. This **destroys information**.

Reset with path

- You can also `reset` individual paths
- This always leaves the branch pointed to by `HEAD` alone, but otherwise behaves the same as `git reset`
- Stage changes to revert a file by five commits:

```
git reset HEAD~5 file1
```

- Note that if you reset a file to the state in `HEAD` (the default), `git reset` is the opposite of `git add`, unstaging any changes.

```
git reset file1   # unstage any changes to file1
git reset HEAD file1 # the same
```

Interactive Rebasing

- A way to make changes further back in history
- Replays commits one at a time, allowing you to modify them:
  - Reorder
  - Edit commit message
  - Amend (aka edit)
  - Squash
  - Fixup (like squash but throw away commit message)
  - Skip
  - Run an arbitrary command
- Done with a simple text interface, explained to you by git.
- A common workflow is you hack away on your local repo, and when ready to share (or make a pull request) you back up your branch (`git branch myhandle/my-backup-branch`) and then perform an interactive rebase onto `master` (say), with `git rebase -i master`. Now you can combine any small fixes and improve your commit messages. Make sure you still pass your tests!

Splitting a commit

- How do I make one commit into two?
- `edit` that commit
- `reset HEAD^`
- Now make two separate commits (don't forgot to re-add new files!)
- Proceed with `git rebase --continue`
- Don't forget `git add -i` and `git gui` if you need to add only some of the changes to a given file.

General Rebasing

- We have seen rebasing as a way to replay commits and edit them
- Commits can also be replayed onto another point in the tree
- This creates new commits, and may require manual resolution of conflicts
    - These are resolved just like the merge conflicts we saw before.

The Typical rebase

- You are working on a local feature branch from `master`

```
git checkout master
git checkout -b myhandle/my-feature
...work...
git add file1 file2
git commit
```

- You would like to get any latest changes from master. One way is a merge

```
git fetch
git merge master
```

This works fine but introduces a merge commit, which some workflows discourage

The Typical Rebase, Cont.

- An alternative, **if** no one else depends on your feature branch, is to simply replay your commits on top of the new `master`

```
git checkout master
git pull
git checkout myhandle/my-feature
git rebase master [myhandle/my-feature]
```

- This is in some ways cleaner, but it is a rewrite of history, so should only be used before your branch is used by other people.
- How did this work? Git found the common ancestor commit of `myhandle/my-feature` and `master`, and played all commits that were on `myhandle/my-feature` onto `master`.

More complex rebases

- You are not restricted to replay commits from the point of common ancestry
- Suppose, as is common, you are working on `myhandle/my-feature` and you make another branch

```
git checkout -b myhandle/my-feature-sub-feature
```

- Later, I decide that I want the changes from `myhandle/my-feature-sub-feature` only to be rebased onto `master`
- git allows this with the `--onto` flag with `git rebase`

```
git rebase --onto master myhandle/my-feature myhandle/my-
    feature-sub-feature
```

rerere

- Problem: one often needs to resolve the same merge conflicts many times
  - Merging `master` into topic branches
- A solution : remember information on past merges
- **Re**use **Re**corded **Re**solution
- `git rerere` is usually not used directly by the user, but it's important to understand that it's there when doing complicated merges.
- Records conflicted merge files (with <<<<<<< etc. in them) and a set of resolved files.
- See `git help rerere` if you ever need to tell git to forget some of this information.

Subsection 2

Deleting Things

`git clean`

- Sometimes you want to remove untracked files - `git clean` does this
- Some useful flags
  - `-x` : delete things ignored by `.gitignore`
  - `-X` : delete only things ignored by `.gitignore`
  - `-d`: also delete untracked directories
  - `-f` : required by default, for safety (unless you do an interactive clean or dry run)
- Best practices: run `git clean -n` to do a dry run. This allows you to make sure you are going to delete exactly what you want to
- `git stash` is related, but allows you to reapply what you have removed.

Deleting branches

- Branches can be deleted with `git branch -d`
- If deleting the branch would orphan commits (the branch hasn't been merged into another branch) then you must use `git branch -D` (and remember `git reflog` if you decide this was an accident)
- To delete a remote branch, push it with the `--delete` option, or with `:`

```
git push --delete origin/psanan/mybranch
git push :origin/psanan/mybranch
```

Overwriting branches

- To force a remote branch to match your local one, use `git push -f`
- To force your local branch to match a remote one, use

```
git checkout mybranch
git fetch myremote
git reset --hard myremote/mybranch
```

Destroying information

- Sometimes you specifically want to destroy information in your git repo
- Some common reasons:
  - You commited a huge file accidentally
  - You commited a file with sensitive data (passwords, keys, personal information, etc.)
- Obviously, as the point is precisely destroy information, it is irreversible, hence dangerous.
- Git provides `git filter-branch` to do this (and more)
- Example
  - Say we have a file, `PURGETHISILE`, with a password in it.
  - To purge this file everywhere, `git filter-branch --tree-filter 'rm -f PURGETHISFILE' HEAD`
- Use `-all` to run the filter on all branches
- You can make a backup of your branch before doing this, as with a rebase, to test.

Git revert

- If you want to undo a commit that you have pushed to a remote repository, how can you avoid rewriting history?
- One obvious option is to make changes manually and push a new commit.
- `git revert` allows you to apply the “inverse” of a given commit
- Be cautious about doing this with complicated workflows. For more, see `git-scm.com/2010/03/02/undoing-merges.html`.

Cleaning up the git repository

- Usually, the fact that git rarely deletes anything is an advantage.
- We have already seen how to use `git filter-branch` to purge specific files.
- However, if you work with a large repository and rebase often, you might end up with a lot of unreachable commits
- The tool `git gc` (garbage collect) automatically does things like organize pack files and remove old, unreachable commits. Usually this is all you should need to run.
- By default, doesn't touch recent entries in the reflog (newer than 90 days, or 30 days for unreachable entries).
- You can prune unreachable objects with a different cuttoff with the `--prune=<date>` option.
- There are more tools, some of which are called by `git gc`
  - `git prune`, `git fsck`, `git prune-packed`, `git repack`, `git pack-objects`

Cherry-picking

- A quick and easy way to apply the changes introduced by a commit as a new commit to the current branch

```
git cherry-pick 132a53
```

- Not dangerous in the ways we have been discussing
- However, this is a duplication of information
- Routine use suggests a confused workflow!

The End

For More Help:

- This presentation: https://bitbucket.org/psanan/gittutorial
- The Git Book: git-scm.com/book
- Tutorials on GitHub and Bitbucket
- Beware StackOverflow: do not panic and start pasting mysterious commands, and note that a lot of the advice is specific to old versions of git.

"... git actually has a simple design, with stable and reasonably well-documented data structures. In fact, I'm a huge proponent of designing your code around the data, rather than the other way around, and I think it's one of the reasons git has been fairly successful ... I will, in fact, claim that the difference between a bad programmer and a good one is whether [he or she] considers [his or her] code or [his or her] data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

–Linus Torvalds, 2006