# Git: Your Favorite Tool

Linus Arver

July 2, 2015

## Outline

## Who am I?

- Coder @ Twin Prime
- Listener @ linus@twinprime.com
- Blogger @ funloop.org

## Git is awesome

- Extremely robust
    - Bit-perfection guaranteed with SHA-1 everywhere
- Extremely powerful
    - Tons of commands

# No one uses all of git all the time

- There are over 100 commands
- Linus Torvalds himself uses a handful of commands (Don Marti interview, 2009)
- Ultimately, it all comes down to discipline
- **`git-flow`** can help, but understand **`git`** first!

# Main Git components

- `.git` folder
- Working tree
- Index (aka Staging Area)
- Commits and Branches

## `.git` folder

- Single monolithic place that has everything git needs
- Low-level (not that interesting)
- But **`.git/config`** (per-project home of **`git-config(1)`**) is high-level and human-friendly!

## Working tree

- Everything *tracked* in your repo except **`.git`** folder
  - "tracked" means tied to the current commit
- Home of what Git calls "local changes"
- Local changes are fragile!
- Requires housekeeping (**`.gitignore`**)

## Index (aka Staging Area)

- Holding place for next commit
- Gives you fine-grained (intra-file) control
- Less fragile than working tree
- Makes Git awesome, but is an alien concept to other source control systems

"*Git is the only DistributedSCM that exposes the concept of index or staging area.*" — *Mercurial*

## Commit

- Git really only cares about commits
- Every commit has a parent (except the first)
- Commits are automatically "insured" with `git reflog`
- The older the commit, the more painful it is to kill

## Branch

- Default is `master`, but *no branch is special*
- Pointer to a commit
- By default, you're always on a branch

## Recap

- Commit - working tree = local changes (stageable things)
- Branches are just convenience pointers to commits

# Practical git usage (basic)

The bare minimum you need to get real work done

## Work work work

- Make changes (aka "local changes") and save files
- Until you commit (or at least stage into index), your changes are fragile!

## `git diff`

- Diffs your work tree against your latest commit
- Visually easier than `git status`
    - Less reading involved — it's either something or nothing

## `git add -p`

- Interactively marks content as "to be committed"
  - *Content*, not necessarily *files*
- For surgical precision, press **e** to add specific lines/characters

## `git diff --cached`

- Shows you what's in the index (aka staging area)
- Always run this before you do **`git commit`**

## `git status`

- Shows you both **`git diff`** and **`git diff --cached`** (abbreviated)
- Especially useful when **`git diff [--cached]`** is too long (e.g., directory renames)
- Also shows you your *untracked* files
  - Use this for updating **`.gitignore`**

## `git reset`

- Clears your index (clears out **`git diff --cached`**)
  - Basically undos all your **`git add`** stuff
- Working tree is not touched (unless **`--hard`** flag)

# `git commit`

- Converts index (`git diff --cached` not `git diff`) into a commit
- Write short and sweet commit messages
- The point is to make them grep-able
- Use `-m` flag for 1-liner commit messages
- Use `--verbose` flag as a reminder of `git diff --cached`

## `git log -p`

- Show commits with diffs b/n them
- `git log FILE`
  - Show commits that touched `FILE`

## **`git push`** (pre vs post v2.0 era)

- Use **`git config --global push.default simple`**
- Use **`matching`** with caution — actually, just don't use it
- Only push if your work is final
  - Exceptions: you have your own branch, or your own (non-github) repo for backups
- Use **`--force`** with caution

## `git pull`

- Get (newest) upstream commits
- Same as `git fetch` then `git merge`
- If your branch is ahead of the remote, nothing happens
  - "Already up-to-date." is a bit misleading

## Practical git usage (the fun part)

Some more tools for your daily routine

## `git branch`

- Lists local branches, including the one you’re on
- **`-d`** deletes local branches
- **`-D`** forces deletion (careful)
- **`-r`** lists remote branches
- **`-a`** lists both remote and local branches

## `git checkout -b NEW_BRANCH_NAME`

- Use current commit as a base for a new branch

# `git merge OTHER_BRANCH`

- Merge `OTHER_BRANCH` into current one
- Conflict?
    - Fix conflicted files
        - Make conflicted areas look like the way you want them to be
    - `git add` those files
    - `git commit` to resolve the conflict
- Merge responsibly, not randomly

## `git reset --hard`

- Clears your index and your working tree (**`git diff`** and **`git diff --cached`** are empty now!)
- Be *very* careful — anything uncommitted will be lost!
- Thankfully, any *untracked* file in your working tree is left alone

## `git reset --hard HEAD~5`

- Like `git reset --hard`, but also moves your branch 5 commits back
- Basically chops off the 5 latest commits from your branch

# `git checkout FILE`

- Undos working tree changes (local changes) you made to **`FILE`**
  - IOW, clears **`git diff`** for **`FILE`**
- Careful — only way to undo this is from your text editor's memory!

## `git checkout DIR`

- Undos working tree changes (local changes) you made to all files in **`DIR`**
- Be careful!

## `git checkout COMMIT_HASH`

- Time travel to an older commit
- Good for examining an old commit’s entire working tree

## `git show COMMIT_HASH`

- Like `git log`, but for a single commit
- Shows all info (diff, author, etc.)

## `git rebase -i HEAD~N`

- Time travel N commits back, and (potentially) amend (or even delete!) commits as needed
- My secret weapon
- Can also use `git rebase -i COMMIT_HASH`

## Edge cases

Uncommon, but still useful, commands

## `vim PROJECT/.gitignore`

- Necessary housekeeping
- Defines the line between *tracked* vs. *untracked*

## `git blame FILE`

- Who touched **`FILE`** last?
- Poor man's documentation
- Logical continuation: **`git show COMMIT`** or **`git log -p FILE`**

## `git mv`, `git rm`

- Like UNIX `mv` and `rm`, but Git-aware
- Automatically performs a `git add` on the paths

## `git reflog`

- Tracks all operations involving a commit hash
- Can undo `git reset --hard`
- The best place to look if you messed up big time

## `git stash` (poor man's commit)

- Save local changes away (so `git diff` shows nothing), but outside the realm of commits
- Show what's stashed with `git stash list`
- `git stash pop`
    - Apply saved changes to working tree
- Only use for temporary one-off things to save time

# git cherry-pick

- Instead of merging an entire branch, merge in only parts of it
- Sounds nice, but only really useful for large projects

## `git grep`

- Self-explanatory
- But, I’m lazy and abuse **`git log -p`** with **`ag`**

## `git tag -a`

- Only for maintainers who cut releases
- Traditional workflow:
  - Change a "VERSION" line in some file
  - `git add -p`
  - `git commit -m 'project_name 1.6'`
  - `git tag -a 'v1.6' -m 'project_name 1.6'`

## Common problems

"Oops" moments

## "I did `git commit` but I'm not ready yet"

- Vim: `:cq`
- My preferred "dumb" way: delete entire commit message `ggVGd:x`

## "I forgot to add something to my commit"

- **`git add -p`**
- **`git commit --amend`** (reuses your existing commit as a template)

## "My new commit is too big!"

- **`git reset HEAD~1`**
- **`git add -p`**
- **`git commit -c ORIG_HEAD`** (uses big commit's original commit message as a template)
- Now do **`git add -p`** and **`git commit`** as many times as necessary

## "I have a typo in my commit message"

- `git commit --amend`

## "I want to combine some of my older commits into one"

- **`git rebase -i HEAD~N`** where **`N`** is how far back you want to change things
- Use **`s`** to squash (combine) a commit into its parent
    - **`f`** is like squash, but discards its commit message

## "I want to undo a commit"

- Local, unpushed commit? **`git reset HEAD~1`**
  - Index will now have the undone commit’s changes
  - Throw away index? **`git reset --hard`**
  - Shortcut: **`git reset --hard HEAD~1`**
- Old, already-pushed commit? **`git revert COMMIT_HASH`**
  - Creates a new commit that undoes the old one
  - Good for history

## git pull

- "I have unfinished (but commited) work, but I still want upstream commits too. But, I don’t want to merge yet (again, I have unfinished work)."
  - `git pull --rebase`

# `git merge`

- Undo
  - Bail out of pending merge? **`git reset --hard`**
  - Undo a merge commit? **`git reset --hard HEAD~1`**
- Avoid creating automated merge commit when updating? **`git pull --rebase`**
  - But if you have a long-running branch, merge responsibly
    - **`git rerere`** may help

# Tips, tools, and more

How to make Git less stupid/painful

# Heavily clean up your commits before pushing

- Use `git add -p`, `git commit --amend` and `git rebase -i` aggressively
- The smaller your commits, the easier it becomes to debug later!
- At the end of the day, Git is for looking at history
- Rebase to re-order commits
- Use commits as backup

## Track things sensibly

- Only do **`git add`** against human-written code
  - **`.gitignore`** the rest
- Git does not really care about file permissions/ownership
  - Git does track execute bits though

## Use aliases

- Git offers aliases, but I prefer shorter shell aliases

```
alias g='git'
alias gdf='git diff'
alias gdfc='git diff --cached'
alias gcm='git commit --verbose'
alias gst='git status'
etc...
```

# **`tig`** (ncurses GUI)

- Basically **`git log -p`** and **`git show`** on steroids
- Recent versions also do **`git status`** by default
- **`brew install tig`**
- In your bashrc/zshrc: `alias tig='tig -n1000'`

## `tig` (continued)

- **`t`** for tree view (working tree view)
- **`b`** for blaming
- Makes **`git blame`**, **`git show`**, **`git log`** much faster

# Docs

- Good high-level tips: `http://gitready.com`
- Official docs: `http://git-scm.com/doc`
- CVS/SVN users, please google "torvalds git tech talk"
- To look up a git command, do **`man git-COMMAND`** (note the dash)

## Thank you!

Have a nice day!