A few git tips you didn't know about

By Mislav on **23 Jul 2010** in git

Notice: some of these commands or flags require git version ***1.7.2****.*
On OS X, upgrade easily with Homebrew: `brew install git`

Show branches, tags in git log

```
 $ git log --oneline --decorate

7466000 (HEAD, mislav/master, mislav) fix test that fails if current dir is not "hub"
494a414 fix cherry-pick of a commit URL
4277848 (origin/master, origin/HEAD, master) whoops
d270fae bugfix: git init -g
9307af3 test deps
8ccc17e http://github.com/defunkt/hub/contributors
64bb19c bugfix: variable name
546726a dont need you
3a8d7af (tag: v1.3.1) v1.3.1
197f429 (tag: v1.3.0) v1.3.0
a1e1a50 not important
3c6af16 magic `cherry-pick` supports GitHub commit URLs and "user@sha" notation
```

Diff by highlighting inline word changes instead of whole lines

```
 $ git diff --word-diff

# Returns a Boolean.
def command?(name)
  `type -t [-#{command}`-]{+#{name}`+}
  $?.success?
end
```

This flag works with other git commands that take diff flags such as `git log -p` and `git show`.

Short status output

```
 $ git status -sb

 ## thibaudgg...thibaudgg/master [ahead 1, behind 2]
  M ext/fsevent/fsevent_watch.c
 ?? Makefile
 ?? SCEvents/
 ?? bin/fsevent_watch
```

The default, verbose `status` output is fine for beginners, but once you get proficient with git there is no need for it. Since I check the status often, I want it to be as concise as possible.

Push a branch and automatically set tracking

```
$ git push -u origin master

# pushes the "master" branch to "origin" remote and sets up tracking
```

"Tracking" is essentially a link between a local and remote branch. When working on a local branch that tracks some other branch, you can `git pull` and `git push` without any extra arguments and git will know what to do.

However, `git push` will by default push all branches that have the same name on the remote. To limit this behavior to just the current branch, set this configuration option:

```
$ git config --global push.default tracking
```

This is to prevent accidental pushes to branches which you're not ready to push yet.

Easily track a remote branch from someone else

```
$ git checkout -t origin/feature

# creates and checks out "feature" branch that tracks "origin/feature"
```

Once your teammate has shared a branch he or she was working on, you need to create a local branch for yourself if you intend to make changes to it. This does that and sets up tracking so that you can just `git push` after making changes.

Checkout a branch, rebase and merge to master

```
# on branch "master":
$ git checkout feature && git rebase @{-1} && git checkout @{-2} && git merge @{-1}

# rebases "feature" to "master" and merges it in to master
```

The special "@{-n}" syntax means "n-th branch checked out before current one". When we checkout "feature", "@{-1}" is a reference to "master". After rebasing, we need to use "@{-2}" to checkout master because "@{-1}" is a reference to the same branch ("feature") due to how rebasing works internally.

Update: Björn Steinbrink points out that this can be done in just 2 commands:

```
$ git rebase HEAD feature && git rebase HEAD @{-2}
```

Pull with rebase instead of merge

```
$ git pull --rebase

# e.g. if on branch "master": performs a `git fetch origin`,
# then `git rebase origin/master`
```

Because branch merges in git are recorded with a merge commit, they are supposed to be meaningful—for example, to indicate when a feature has been merged to a release branch. However, during a regular daily workflow where several team members sync a single branch often, the timeline gets polluted with unnecessary micro-merges on regular `git pull`. Rebasing ensures that the commits are always re-applied so that the history stays linear.

You can configure certain branches to always do this without the `--rebase` flag:

```
# make `git pull` on master always use rebase
$ git config branch.master.rebase true
```

You can also set up a global option to set the last property for every new tracked branch:

```
# setup rebase for every tracking branch
$ git config --global branch.autosetuprebase always
```

Find out if a change is part of a release

```
$ git name-rev --name-only 50f3754

"tags/v2.3.8~6"
```

It's not rare that you know a SHA-1 of a commit but aren't sure where is it located in project's history. If you're like me, you probably want to know was that change a part of some release or not. You can use `git show` to see the commit message, date and the full diff, but this doesn't help us much—especially since comparing commit dates in a project's history doesn't necessarily correspond to the order in which they were applied.

The `name-rev` command can tell us the position of a commit relative to tags in the project. The example above is from the Ruby on Rails project. This tells us that this commit is located 6 commits before "v2.3.8" was tagged—we can be certain that this change is now part of Rails 2.3.8, then.

The command goes even further in its usefulness. Suppose you follow a discussion in which someone mentions a few commits:

> *This bug was introduced in e6cadd422b72ba9818cc2f3b22243a6aa754c9f8 but fixed in 50f3754525c61e3ea84a407eb571617f2f39d6fe, if I recall correctly.*

You can copy that to clipboard and pipe the comment to `git name-rev`, which will recognize commit SHAs and append tag information to each:

```
$ pbpaste | git name-rev --stdin

"This bug was introduced in e6cadd422b72ba9818cc2f3b22243a6aa754c9f8 (tags/
v2.3.6~215)
but fixed in 50f3754525c61e3ea84a407eb571617f2f39d6fe (tags/v2.3.8~6), if I recall
correctly."
```

See also: `git help describe`

Find out which branch contains a change

```
$ git branch --contains 50f3754
```

This filters the lists of branches to only those which have the given commit among their ancestors. To also include remote tracking branches in the list, include the "-a" flag.

See which changes from a branch are already present upstream

```
# while on "feature" branch:
$ git cherry -v master

+ 497034f2 Listener.new now accepts a hash of options
- 2d0333ff cache the absolute images path for growl messages
+ e4406858 rename Listener#run to #start
```

The `cherry` command is useful to see which commits have been cherry-picked from a development branch to the stable branch, for instance. This command compares changes on the current ("feature") branch to upstream ("master") and indicates which are present on both with the "–" sign. Changes still missing from upstream are marked with "+".

Show the last commit which message matches a regex

```
$ git show :/fix
# shows the last commit which has the word "fix" in its message
```

```
$ git show :/^Merge
# shows the last merge commit
```

Fetch a group of remotes

```
$ git config remotes.default 'origin mislav staging'
$ git remote update

# fetches remotes "origin", "mislav", and "staging"
```

You can define a default list of remotes to be fetched by the `remote update` command. These can be remotes from your teammates, trusted community members of an opensource project, or similar. You can also define a named group like so:

```
$ git config remotes.mygroup 'remote1 remote2 ...'
$ git fetch mygroup
```

Write commit notes

```
$ git notes add
# opens the editor to add a note to the last commit
```

Git notes are annotations for existing commits. They don't change the history, so you are free to add notes to any existing commits. Your notes are stored only in your repo, but it's possible to share notes. There are interesting ideas for possible use-cases for notes, too.

Install "hub"

Hub teaches git about GitHub. If you're using repos from GitHub on a regular basis, you definitely want to install hub and save a lot of keystrokes—*especially* if you're involved in opensource.

Also on this site

09 Aug 2011 ▸ will_paginate v3.0: Rails 3, Sinatra and more in ruby

11 Jul 2011 ▸ Faraday: advanced HTTP requests made easy in ruby

22 Jun 2011 ▸ Ruby verbose mode and how it's broken in ruby

Got your attention? Find out about my projects, subscribe to the blog or follow me on Twitter.

← ALL POSTS