https://longair.net/blog/2009/04/16/git-fetch-and-merge/

# GIT: FETCH AND MERGE, DON'T PULL

2009-04-16 MARK 127 COMMENTS

*This is too long and rambling, but to steal a joke from ~~Mark Twain~~ Blaise Pascal I haven't had time to make it shorter yet. There is some discussion of this post on the git mailing list, but much of it is tangential to the points I'm trying to make here.*

One of the git tips that I find myself frequently passing on to people is:

> Don't use git pull, use git fetch and then git merge.

The problem with *git pull* is that it has all kinds of helpful magic that means you don't really have to learn about the different types of branch in git. Mostly things Just Work, but when they don't it's often difficult to work out why. What seem like obvious bits of syntax for *git pull* may have rather surprising results, as even a cursory look through the manual page should convince you.

The other problem is that by both fetching and merging in one command, your working directory is updated without giving you a chance to examine the changes you've just brought into your repository. Of course, unless you turn off all the safety checks, the effects of a *git pull* on your working directory are never going to be catastrophic, but you might prefer to do things more slowly so you don't have to backtrack.

## Branches

Before I explain the advice about *git pull* any further it's worth clarifying what a branch is. Branches are often described as being a "line of development", but I think that's an unfortunate expression since:

- If anything, a branch is a "directed acyclic graph of development" rather than a line.

- It suggests that branches are quite heavyweight objects.

I would suggest that you think of branches in terms of what defines them: they're a name for a particular commit and all the commits that are ancestors of it, so each branch is completely defined by the SHA1sum of the commit at the tip. This means that manipulating them is a very lightweight operation – you just change that value.

This definition has some perhaps unexpected implications. For example, suppose you have two branches, "stable" and "new-idea", whose tips are at revisions E and F:

```
A-----C----E ("stable")
 \
  B-----D-----F ("new-idea")
```

So the commits A, C and E are on "stable" and A, B, D and F are on "new-idea". If you then merge "new-idea" into "stable" with the following commands:

```
git checkout stable   # Change to work on the branch "stable"
git merge new-idea    # Merge in "new-idea"
```

... then you have the following:

```
A-----C----E----G ("stable")
 \          /
  B-----D-----F ("new-idea")
```

If you carry on committing on "new idea" and on "stable", you get:

```
A-----C----E----G---H ("stable")
 \          /
  B-----D-----F----I ("new-idea")
```

So now A, B, C, D, E, F, G and H are on "stable", while A, B, D, F and I are on "new-idea".

Branches do have some special properties, of course – the most important of these is that if you're working on a branch and create a new commit, the branch tip will be advanced to that new commit. Hopefully this is what you'd expect. When merging with *git merge*, you only specify the branch

you want to merge into the current one, and only your current branch advances.

Another common situation where this view of branches helps a lot is the following: suppose you’ re working on the main branch of a project (called “master” , say) and realise later that what you’ ve been doing might have been a bad idea, and you would rather it were on a topic branch. If the commit graph looks like this:

```
 last version from another repository
   |
   v
M---N-----O----P---Q ("master")
```

Then you separate out your work with the following set of commands (where the diagrams show how the state has changed after them):

```
git branch dubious-experiment

M---N-----O----P---Q ("master" and "dubious-experiment")

git checkout master

# Be careful with this next command: make sure "git status" is
# clean, you're definitely on "master" and the
# "dubious-experiment" branch has the commits you were working
# on first...

git reset --hard <SHA1sum of commit N>

    ("master")
M---N-------------O----P---Q ("dubious-experiment")

git pull # Or something that updates "master" from
       # somewhere else...

M--N----R---S ("master")
   \
    O---P---Q ("dubious-experiment")
```

This is something I seem to end up doing a lot... :)

# Types of Branches

The terminology for branches gets pretty confusing, unfortunately, since it has changed over the course of git's development. I'm going to try to convince you that there are really only two types of branches. These are:
(a) "Local branches": what you see when you type *git branch*, e.g. to use an abbreviated example I have here:

```
$ git branch
  debian
  server
* master
```

(b) "Remote-tracking branches": what you see when you type *git branch -r*, e.g.:

```
$ git branch -r
cognac/master
fruitfly/server
origin/albert
origin/ant
origin/contrib
origin/cross-compile
```

The names of tracking branches are made up of the name of a "remote" (e.g. origin, cognac, fruitfly) followed by "/" and then the name of a branch in that remote respository. ("remotes" are just nicknames for other repositories, synonymous with a URL or the path of a local directory – you can set up extra remotes yourself with "git remote", but "git clone" by default sets up "origin" for you.)
If you're interested in how these branches are stored locally, look at the files in:

- `.git/refs/heads/` [for local branches]
- `.git/refs/remotes/` [for tracking branches]

Both types of branches are very similar in some respects – they're all just stored **locally** as single SHA1 sums representing a commit. (I emphasize "locally" since some people see "origin/master" and assume that in

some sense this branch is incomplete without access to the remote server – that isn't the case.)

Despite this similarity there is one particularly important difference:

- The safe ways to change remote-tracking branches are with *git fetch* or as a side-effect of *git-push*; you can't work on remote-tracking branches directly. In contrast, you can always switch to local branches and create new commits to move the tip of the branch forward.

So what you mostly do with remote-tracking branches is one of the following:

- Update them with *git fetch*
- Merge from them into your current branch
- Create new local branches based on them

## Creating local branches based on remote-tracking branches

If you want to create a local branch based on a remote-tracking branch (i.e. in order to actually work on it) you can do that with *git branch –track* or *git checkout –track -b*, which is similar but it also switches your working tree to the newly created local branch. For example, if you see in *git branch -r* that there's a remote-tracking branch called origin/refactored that you want, you would use the command:

```
git checkout --track -b refactored origin/refactored
```

In this example "refactored" is the name of the new branch and "origin/refactored" is the name of existing remote-tracking branch to base it on. (In recent versions of git the "–track" option is actually unnecessary since it's implied when the final parameter is a remote-tracking branch, as in this example.)

The "–track" option sets up some configuration variables that associate the local branch with the remote-tracking branch. These are useful chiefly

for two things:

- They allow *git pull* to know what to merge after fetching new remote-tracking branches.
- If you do *git checkout* to a local branch which has been set up in this way, it will give you a helpful message such as:

> Your branch and the tracked remote branch 'origin/master'
> have diverged, and respectively have 3 and 384 different
> commit(s) each.

... or:

> Your branch is behind the tracked remote branch
> 'origin/master' by 3 commits, and can be fast-forwarded.

The configuration variables that allow this are called "branch.<local-branch-name>.merge" and "branch.<local-branch-name>.remote", but you probably don't need to worry about them.

You have probably noticed that after cloning from an established remote repository *git branch -r* lists many remote-tracking branches, but you only have one local branch. In that case, a variation of the command above is what you need to set up local branches that track those remote-tracking branches.

> You might care to note some confusing terminology here: the word "track" in "–track" means tracking of a remote-tracking branch by a local branch, whereas in "remote-tracking branch" it means the tracking of a branch in a remote repository by the remote-tracking branch. Somewhat confusing...

Now, let's look at an example of how to update from a remote repository, and then how to push changes to a new repository.

## Updating from a Remote Repository

So, if I want get changes from the remote repository called "origin" into my local repository I'll type *git fetch origin* and you might see some output like this:

```
remote: Counting objects: 382, done.
remote: Compressing objects: 100% (203/203), done.
remote: Total 278 (delta 177), reused 103 (delta 59)
Receiving objects: 100% (278/278), 4.89 MiB | 539 KiB/s, done.
Resolving deltas: 100% (177/177), completed with 40 local objects.
From ssh://longair@pacific.mpi-cbg.de/srv/git/fiji
   3036acc..9eb5e40  debian-release-20081030 -> origin/debian-release-20081030
 * [new branch]      debian-release-20081112 -> origin/debian-release-20081112
 * [new branch]      debian-release-20081112.1 -> origin/debian-release-20081112.1
   3d619e7..6260626  master     -> origin/master
```

The most important bits here are the lines like these:

```
   3036acc..9eb5e40  debian-release-20081030 -> origin/debian-release-20081030
 * [new branch]      debian-release-20081112 -> origin/debian-release-20081112
```

The first line of these two shows that your remote-tracking branch origin/debian-release-20081030 has been advanced from the commit 3036acc to 9eb5e40. The bit before the arrow is the name of the branch in the remote repository. The second line similarly show that since we last did this, a new remote tracking branch has been created. (*git fetch* may also fetch new tags if they have appeared in the remote repository.)

The lines before those are *git fetch* working out exactly which objects it will need to download to our local repository's pool of objects, so that they will be available locally for anything we want to do with these updated branches and tags.

*git fetch* doesn't touch your working tree at all, so gives you a little breathing space to decide what you want to do next. To actually bring the changes from the remote branch into your working tree, you have to do a *git merge*. So, for instance, if I'm working on "master" (after a *git checkout master*) then I can merge in the changes that we've just got from origin with:

```
git merge origin/master
```

(This might be a fast-forward, if you haven't created any new commits that aren't on master in the remote repository, or it might be a more complicated merge.)

If instead you just wanted to see what the differences are between your branch and the remote one, you could do that with:

```
git diff master origin/master
```

This is the nice point about fetching and merging separately: it gives you the chance to examine what you've fetched before deciding what to do next. Also, by doing this separately the distinction between when you should use a local branch name and a remote-tracking branch name becomes clear very quickly.

## Pushing your changes to a remote repository

How about the other way round? Suppose you've made some changes to the branch "experimental" and want to push that to a remote repository called "origin". This should be as simple as:

```
git push origin experimental
```

You might get an error saying that the remote repository can't fast-forward the branch, which probably means that someone else has pushed different changes to that branch. So, that case you'll need to fetch and merge their changes before trying the push again.

> Aside
> If the branch has a different name in the remote repository ("experiment-by-bob", say) you'd do this with:
>
> ```
> git push origin experimental:experiment-by-bob
> ```
>
> On older versions of git, if "experiment-by-bob" doesn't already exist, the syntax needs to be:
>
> ```
> git push origin experimental:refs/heads/experiment-by-bob
> ```
>
> ... to create the remote branch. However that seems to be no longer the case, at least in git version 1.6.1.2 – see Sitaram's comment below.
> If the branch name is the same locally and remotely then it will be created
> automatically without you having to use any special syntax, i.e. you can just do git push origin experimental as normal.

> In practice, however, it's less confusing if you keep the branch names the same. (The <source-name>:<destination-name> syntax there is known as a "refspec", about which we'll say no more here.)

~~An important point here is that this *git push* doesn't involve the remote-tracking branch origin/experimental at all – it will only be updated the next time you do *git fetch*.~~ *Correction: as Deskin Miller points out below, your remote-tracking branches will be updated on pushing to the corresponding branches in one of your remotes.*

## Why not git pull?

Well, *git pull* is fine most of the time, and particularly if you're using git in a CVS-like fashion then it's probably what you want. However, if you want to use git in a more idiomatic way (creating lots of topic branches, rewriting local history whenever you feel like it, and so on) then it helps a lot to get used to doing *git fetch* and *git merge* separately.