

A short introduction to git

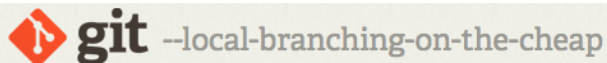
John Ladan

University of Waterloo
john@ladan.ca

January 19, 2015

Overview

- 1 What is git?
- 2 More detail
- 3 Cloning a Repo
- 4 Making Changes
- 5 Committing Changes
- 6 Sharing with others
- 7 Rolling Your Own
- 8 Version Control
- 9 In Practice



Git is a **free and open source** distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is **easy to learn** and has a **tiny footprint with lightning fast performance**. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like **cheap local branching**, convenient **staging areas**, and **multiple workflows**.

Git (/t/) is a distributed revision control system with an emphasis on speed, data integrity, and support for distributed, non-linear workflows. Git was initially designed and developed by Linus Torvalds for Linux kernel development in 2005, and has since become the most widely adopted version control system for software development.

As with most other distributed revision control systems, and unlike most clientserver systems, every Git working directory is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server. Like the Linux kernel, Git is free software distributed under the terms of the GNU General Public License version 2.

Features

- Fast
- Nonlinear development (via rapid branching, and DAG based structure)
- Distributed – each copy of the repo is complete by itself
 - each copy of the repo is itself a repo
 - changes can be pushed/pulled to any other instance
 - do not need network access
- Ubiquitous
- Good *free* repository hosting (github)

Structure of a repository

- Everything is stored in files, with all the metadata (esp. history) in `.git`.
- The history is structured as a Directed Acyclic Graph (DAG). It's basically just a tree where branches can join back together.
- The history is (for the most part) stored as changes (diffs) between the nodes (commits).

Git Terminology

A few terms to help you out.

- Commit: *n.* A particular snapshot of the project, labelled with a unique hash.
- remote: *n.* A remote repository that should have the same commit history (to a point)
- branch: *n.* A pointer to a specific commit. When the branch is checked out, committing changes will advance that pointer.
- master: the main branch of the repository
- merge: *v.* The process of combining two branches
- rebase: *v.* cutting of a branch, and grafting it on somewhere else.
- HEAD: *n.* The currently checked out commit.
- tag: *n.* A fancy branch label that stays put, and can be signed by the author

Basic workflow

- clone/create a repository
- branch
- make changes
- commit
- merge changes into main branch
- push changes upstream

Git Clone

Command

```
git clone <repository-location> [<local-path>]
```

It's a lot like downloading a tarball, except

- only the necessary files are included (no build files)
- lots of option, like pulling a specific commit, or not downloading full history

Submodules

Word of warning: some packages use *submodules*. One way to handle it:

Command

```
git clone --recursive <repo>
```

Branching

You should always branch first:

Branching

You should always branch first:

- it's a good habit to be in;
- the last good state is left labeled;
- easy to checkout other branches, even when you're in the middle of something;
- pulling origin/master is easier; and
- branches are cheap!

Git Branch

Command

```
git branch <branch-name>  
git checkout <branch-name>
```

Alternatively,

Command

```
git checkout -b <branch-name>
```

When editing, you can use whatever tools you want, with one (or two) exceptions:

Editing

When editing, you can use whatever tools you want, with one (or two) exceptions: moving and removing files.

Command

```
git mv <file> <new-filename>
```

Command

```
git rm <file>
```

This way, git understands that it is the same file (or that it is intentionally gone).

Seeing your changes

Before committing, it's a good idea to see what changes you made

Command

```
git status
```


Seeing your changes

Before committing, it's a good idea to see what changes you made

Command

```
git status
```

And if you want the details,

Command

```
git diff <file>
```

Staging files

Next, we need to let git know what files have changed, and what new files it should track

Command

```
git add <file(s)>
```

Staging files

Next, we need to let git know what files have changed, and what new files it should track

Command

```
git add <file(s)>
```

Often, it's good to check the *status* again before the next step

Commit!

Finally, we commit our changes to the tree.

Command

```
git commit [-m "Message text"]
```

Commit!

Finally, we commit our changes to the tree.

Command

```
git commit [-m "Message text"]
```

(I recommend not using the `-m` flag)

Practice safe committing

Here are a few best practices:

- Commits should be small and non-breaking
- The message should complete the sentence “This commit will...”
- Keep the commit to one feature/bug-fix
- Don't use the `-m` flag

Speeding up commits

There's a faster way...

Command

```
git commit -a
```

The `-a` flag stages *all* modified files, but not any untracked files.

Merging

After some commits, you'll probably want to add those changes to the main branch.

Command

```
git checkout master  
git merge <other-branch>
```

If there are no conflicts (i.e. the commit is a direct descendent), the current branch will be fast-forwarded. This just means the pointer gets moved ahead.

If there are conflict, we have to fix them manually.

Pushing

What about sending the changes upstream?

Command

```
git push [<remote-name>]
```

Pushing

What about sending the changes upstream?

Command

```
git push [<remote-name>]
```

If the remote branch has changed you'll have to pull first

Command

```
git pull [<remote-name>]
```

and then push.

Pushing

What about sending the changes upstream?

Command

```
git push [<remote-name>]
```

If the remote branch has changed you'll have to pull first

Command

```
git pull [<remote-name>]
```

and then push. Because of this, it's probably better to pull before merging your working branch. That should keep the tree of commits cleaner.

Rolling your own

- What normally happens.
- What ideally happens.

Typical Case

This is what often happens...

- start coding, and realize you want this in a git repo
- create a new repo

Command

```
git init ; git add... ; git commit
```

- continue coding, then realize you want a remote backup
- create a repo on github, then...

Command

```
git remote add origin <repo-url>; git push -u master origin
```

Problems

Pros:

- you can always retroactively add version control to a project

Cons:

- All the past revision history is lost.
- These commands are more obscure and harder to remember.

Ideal Case

- Have the foresight to create a repo (e.g. on github)
- Clone that repo onto your workstation
- Start coding and committing as usual

One more thing

If you don't want to share your project publicly, and have access to a server,

Command

```
git init --bare
```

This creates a "headless" repo, storing all the data, but without the working tree. You can then push/pull/clone from this remote repo as usual.

One more thing

If you don't want to share your project publicly, and have access to a server,

Command

```
git init --bare
```

This creates a "headless" repo, storing all the data, but without the working tree. You can then push/pull/clone from this remote repo as usual.

E.g. I put all of my repos in a directory on CSC servers.

```
jladan@taurine.uwaterloo.ca:/users/jladan/git-repos/
```

Version Control

Now for the reasons *why* we actually go through all this effort.

Checking out

Any point in the DAG can be accessed by checking it out

Command

```
git checkout <branch-name/commit-hash>
```

This changes the working tree to be in the state of that commit. However, no untracked files are changed.

Undo! Undo!

The easiest time to undo your changes is before committing them.

Command

```
git checkout -- <filename>
```

Undo! Undo!

But what if you didn't mean to commit that change?

Command

```
git reset [--hard] HEAD~1
```

Undo! Undo!

But what if you didn't mean to commit that change?

Command

```
git reset [--hard] HEAD~1
```

HEAD~n is a synonym for 'n commits before head'

Undo! Undo!

But what if you didn't mean to commit that change?

Command

```
git reset [--hard] HEAD~1
```

HEAD~n is a synonym for 'n commits before head'

Do not delete commits that are already upstream

Git Reset

The reset command actually performs a lot of different functions. It's like the 'undo' of git.

It's probably a good idea to read the man-page for it

Command

```
man git-reset
```

There are a lot of useful recipes at the end of that document.

I made a mistake in that last commit!

If you made a mistake in what changes were staged or the commit message has a typo,

Command

```
git commit --amend
```

I made a mistake in that last commit!

If you made a mistake in what changes were staged or the commit message has a typo,

Command

```
git commit --amend
```

This creates a new commit. Each commit is labeled by its hash, so amending a commit actually replaces it with a new one. Thus you cannot amend commits that have already been pushed.

Tagging and Versions

Saving the location in the tree of a specific version is easy

Command

```
git tag <tag-name>
```

A tag is essentially just a branch name that is fixed in place. It can also be signed by the developer, so that no one tampers with it later.

What you'll use a lot

- `git status`: tells you where you are, and what commands you might want to use
- `git log`: tells you where everything is (reminder: `git glog`)
- `git commit`

Getting Help


There are 3 main sources of help for git:

- git status
- man pages: `man git-<command>`
- google

Github is also good at telling you the commands to do various things

github example

Quick setup — if you've done this kind of thing before

 Set up In Desktop or ☐ HTTPS ☒ SSH `git@github.com:jladan/git-presentation.git`

We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
touch README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:jladan/git-presentation.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin git@github.com:jladan/git-presentation.git
git push -u origin master
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

[Import code](#)

Other functions

Stashing can be quite useful when you haven't branched

Command

```
git stash  
git checkout -b <new-branch>  
git stash pop
```

Other functions

Stashing can be quite useful when you haven't branched

Command

```
git stash  
git checkout -b <new-branch>  
git stash pop
```

Rebasing can help keep the graph clean

Command

```
git rebase <new-root>
```

You can combine and reorder commits in this process too. However, some people frown on this practice, because it doesn't accurately track the history.

A few other things

- You'll have to delete the old branch names from time to time
- To contribute to other projects, you'll likely need to do a 'pull request'
- You should always have a license and readme file
- Keep a clean setup with `.gitignore`