Duane’s Incredible Brief Introduction to Git

a working document for working students

Duane A. Bailey

(http://www.cs.williams.edu/˜bailey/git.pdf)

[THIS IS A DRAFT DOCUMENT]

CONTENTS

* Meet the local git
* Creating a shared repository
* Cloning a shared repository
* Git for singles
* Populating the project
* Day-to-day workflow
* You need to git help

Git ("the stupid content tracker") is a popular distributed version control system that is suitable for keeping track of versions of a project. Git can be used to support many different types of workflows, but we’ll think of it as a means for (say) students to share code as they develop a common project.

As with other version control systems, the life of a project involves the following repository operations:

1. The repository is created and shared with all participants.
2. Each participant clones the repository.
3. Concurrently, the participants
   a. Pull changes from the global repository.
   a. Make changes to their local copies of the code, commiting the changes.
   b. Pushing commits into the global repository.

This document suggests only one approach for setting up and maintaining a Git repository. It is suitable for keeping track of a small project, possibly between a couple of programmers. For more advanced approaches, you should refer to the Pro Git reference manual at

    https://git-scm.com/documentation

The book is also available in pdf format from:

    https://git-scm.com/book

N.B. Using Git can simplify the process of working on a shared project, but it cannot resolve difficulties that step from communication problems among team members. The Mantra: Communicate.

MEET THE LOCAL GIT

Git has a global configuration file (˜/.gitconfig) that keeps track of your identity as a contributor to your various projects. Before you use Git for the first time it is helpful if you provide reasonable values for some of these parameters. For example, Rita Decoder would type (don’t type the %):

```
    % git config --global user.name "Joseph Cool"
    % git config --global user.email "jcool@cs.williams.edu"
    % git config --global push.default simple
    % git config --global core.editor emacs
```

The first two lines describe you. The ’push.default’ line describes how your changes get pushed up to the master repository (the simplest technique possible). The last line sets your preferred editor (Git will occasionally ask you to edit log messages or merge file versions). These configuration parameters should be specified whenever you’re first using a new machine (you’ve started using a lab, you’ve bought a new laptop, etc.).

CREATING A SHARED REPOSITORY

In a close-knit environment where all partners in a project have access to the same private machines, it is useful (and often preferred) to create a project database, somewhere, that can be used as a clearing house for changes submitted by project members. The method I suggest is for one member to be the ad hoc *project leader*; this programmer maintains the master repository in their own space, but gives all members of a unix group shared access. Let’s assume that’s Rita Decoder, whose unix id is "decoder".

First, of course, you need to make sure that all members of the project are in the same unix group. Ask your system administrator (Mary) to create a new unix group with all the team members. She’ll give you an identifier; let’s assume this is ’ourgrp’. Rita can see the groups she’s a member of with

```
    % groups
    decoder ourgrp
```

The first group is Rita’s default group and is typically the same as her username. When you do this, if you don’t see the group name your administator gave you, logout and re-login.

The project leader (let’s assume it’s decoder) types the following commands to establish the project database. If this is Rita’s first Git project, she would set up a master directory that will hold all her projects:

```
    % mkdir ~/git
    % chgrp ourgrp ~/git
```

She’ll store our Git project, ’prj’, as a subdirectory of this master directory:

```
    % cd ~/git
    % mkdir prj.git
    % chgrp ourgrp prj.git
```

The suffix, .git, though unnecessary, helps identify that this is a repository that is maintained by Git. The chgrp command changes the owning group to ’ourgrp’ (it was probably ’decoder’ beforehand).

We now set up the empty database:

```
    % git init --bare --shared=group
```

This tells Git that this directory will hold the Git database. It will be ’bare’---this is not a directory that we will work in, it just contains the project database. All members of prj.git’s group (ourgrp) will have shared access to this database.

CLONING A SHARED REPOSITORY

Once a repository has been created, anyone in the group can make a full copy---they can ’clone’---the repository. The repository is cloned using a Uniform Resource Locator (URL) that (ideally) will be stable during the life of the project. The URL can use any number of different protocols; in secure environments I suggest using secure shell, ’ssh’. Because contributing machines---especially laptops---can move between domains, it is best to specify a machine that is valid from any location. At Williams, we suggest using rath.cs.williams.edu.

Every contributor clones Rita’s empty prj.git project:

```
    % git clone http://rath.cs.williams.edu/~decoder/git/prj.git ~/myprj
    Cloning...
    jcool@rath.cs.williams.edu’s password:
    warning: You appear to have cloned an empty repository.
```

This accesses the machine rath.cs.williams.edu using Joe Cool’s password. The repository is ’prj.git’, hosted by ’decoder’. The files stored within this repository will be brought into a local copy of the entire repository in the directory ’myprj’. The first time you clone the repository, it may give you a warning that the repository is empty. Well, yeah.

GIT FOR SINGLES

If you’re working on your own, you can pretend to be working in a group with a single contributor. You can either follow the above instructions (you needn’t have a special group), or you can simply create a Git repository in your project’s working directory. It’s as easy as

```
    % cd ˜/myprj
    % git init
    Initialized empty Git repository in /home/jcool/myprj.
```

Git provides you all the same security of being able to track changes, to back out of faulty implementations, etc. In this local mode, of course, you needn’t worry about synchronizing work with others.

POPULATING THE PROJECT

As a contributor, you do your work in your own account in (say) ˜/myprj:

```
    % cd ˜/myprj
```

Here, you create new files or copy files from elsewhere. For example, maybe your first file is README:

```
    % ls
    README
```

You can ask Git to give you the status of an individual file, or if none is specified, the current directory (--short generates a more compact report):

```
    % git status
    On branch master
    Initial commit

    Untracked files:
      (use "git add <file>..." to include in what will be committed)
      README
    nothing added to commit but untracked files present
    (use "git add" to track)
    % git status --short
    ?? README
```

Git is telling us that the README file is not currently part of the set of files that are included in the project. To add files to a project, you need tell Git they should be *staged*, or added to the set of files next committed:

```
    % git add README
```

Git is silent. If you’re concerned, you can get the status again and it will report (the -s switch is just a shorthand for --short):

```
    % git status
    ...
    Changes to be committed:
      (use "git rm --cached <file>..." to unstage)
      new file:   README
    % git status -s
    M  README
```

After you’ve staged other files, you can commit to a new revision of the project:

```
    % git commit -m ’Added README.  The project begins!’
    [master (root-commit) 74d039f] Added README.  The project begins!
    1 file changed, 0 insertions(+), 0 deletions(-)
    create mode 100644 README
```

The -m switch specifies a message that describes what changes you’ve made. The message should be put into single quotes to keep the shell from interpreting any of the special characters. The result is a new revision, numbered 74d039f. This identifier is a unique prefix for a much longer checksum derived from the files in the project.

As the project progresses, you change files, or expand your project to include new files. Just before you commit to a new version of the project, you stage files whose revisions are important to that version. There may be other files whose modifications are not important---don't add them. For example perhaps you've created a Makefile and you're in the process of modifying the README:

```
    % git status -s
     M README
    ?? Makefile
```

If a collaborator is interested in the Makefile, you add it to the staging area and commit it:

```
    % git add Makefile
    % git status -s
     M README
    A  Makefile
    % git commit -m 'Started Makefile.'
    [master 1ac0118] Started Makefile
     1 file changed, 0 insertions(+), 0 deletions(-)
     create mode 100644 Makefile
    % git status -s
     M README
```

This commits this snapshot of your project to your local copy of the repository. Next, you need to upload those to Rita Decoder's master repository where the changes can be seen by others. This synchronization process is called "pushing":

```
    % git push
    jcool@rath.cs.williams.edu's password:
    Counting objects: 5, done.
    Delta compression using up to 4 threads.
    Compressing objects: 100% (3/3), done.
    Writing objects: 100% (5/5), 459 bytes | 0 bytes/s, done.
    Total 5 (delta 0), reused 0 (delta 0)
    To ssh://rath.cs.williams.edu/~decoder/git/prj.git
     * [new branch]      master -> master
```

At this point your local repository's changes can be found in the project leader's database. If others want to see what you've done, they "pull" the changes to their local repositories. Here's what Rita Decoder sees:

```
    % git pull
    decoder@rath.cs.williams.edu's password:
    remote: Counting objects: 5, done.
    remote: Compressing objects: 100% (2/2), done.
    remote: Total 3 (delta 0), reused 0 (delta 0)
    Unpacking objects: 100% (3/3), done.
    From ssh://rath.cs.williams.edu/~decoder/git/prj.git
       074a133..cbc9c7f  master     -> origin/master
    Updating 074a133..cbc9c7f
    Fast-forward
     Makefile | 1 +
     1 file changed, 1 insertion(+)
```

As projects progress, many files are created as the side effect of the development process (object files, editor backups, etc.). You can tell Git to explicitly ignore these files with the '.gitignore' file. We typically create this file in the top directory of the project. Here's what one typically looks like:

```
    % cat .gitignore
    *~
    *.o
```

This file, like other files, should be staged and committed to the project
so that others can take advantage of it.

DAY-TO-DAY WORKFLOW

As you get more accomplished using Git, you will settle into a workflow that
allows contributors to work effectively together. Working with others
still requires a little focussed effort. Here are some hints on how to
get along.

* Work together as people. Git will not replace personal
  communication. Before you dive into the next stage of a project, discuss
  who will do what. Your number one goal is to avoid editing the same file
  at the same time. If you find yourself needing to make an
  "insignificant" change to a file, clear it with your colleages, first.
* When you begin work, start with
  ```
  % git pull
  ```
  this will bring your repository up-to-date.
* As you work, commit frequently. As you make progress, use git commits to
  celebrate your small victories. You can ’add’ and ’commit’ all files
  that are currently being tracked with the command:
  ```
  % git commit -a -m ’Finally passed the robotic intake unit test!’
  ```
* When you reach point where you think your code is useful to others,
  commit, pull down their changes, and push yours:
  ```
  % git commit -a -m ’Potato warming interface fleshed out.  Whew.’
  % git pull
  % git push
  ```

* When you quit work, synchornize your changes (no matter their state) and
  log out of the machine:
  ```
  % git commit -a -m ’Ethanol blender unit partially built; headed home.’
  % git pull
  % git push
  % logout
  ```

YOU NEED TO GIT HELP

Git provides many, many other features. You can read about these on-line,
of course, but Git also can provide help locally:

* Git has man pages. Begin with
  ```
  % man git
  ```
* Git carries its own help. You can ask Git to give you general help:
  ```
  % git help
  ```
  or Git can give help on a specific command:
  ```
  % git help pull
  ```
  or
  ```
  % git help help
  ```
  or Git can help you with a concept:
  ```
  % git help everyday
  ```
* The ’Pro Git, 2nd Ed’ book is freely downloadable from the web in PDF
  format at https://git-scm.com/book