pro-git

Table of Contents

1 DONE Getting Started

1.1 Version Constrol

A system that records changes of a file or set of files.

1.2 Evolution

1. Local Version Control Systems
2. Centralized Version Control Systems
3. Distributed Version Control Systems

1.3 Git Basics

1.3.1 Snapshots, Not Differences

Git thinks of its data more like a set of snapshots of a miniature filesystem. Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.
To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored.

1.3.2 Nearly Every Operation Is Local

1.3.3 Git Has Integrity

Everything in Git is check-sumed before it is stored and is then referred to by that checksum. The mechanism that Git uses for this checksumming is called a SHA-1 hash. In fact, Git stores everything in its database not by file name but by the hash value of it contents.

1.3.4 Git Generally Only Adds Data

1.3.5 The Three States

Git has three main states that your files can reside in:

Name	Meaning
Committed	means that the data is safely stored in your local database.

Name	Meaning
Modified	means that you have changed the file but have not committed it to you database yet.
Staged	means that you have marked a modified file in its current version to go into your next commit snapshot.

This leads us to the three main sections of a Git project:
the Git directory, working directory, and the staging area.
The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.
The working directory is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit.

1.4 First-Time Git Setup

Git comes with a tool called "git config" that lets you get and set configuration variables that control all aspects of how Git looks and operates.
These variables can be stored in three different places:

1. /etc/gitconfig file: Contains values for every user on the system and all their repositories. If you pass the option –system to git config, it reads and writes from this file specifically.
2. ~/.gitconfig or ~/.config/git/config file: Specific to your user. You can make git read and write to this file specifically by passing the –global option.
3. config file in the Git directory (that is, .git/config) of whatever repository you're currently using: Specific to that single repository.

Each level overrides values in the previous level.

1.4.1 Your Identity

git config –global user.name "Michael Chyson"
git config –global user.email chyson@aliyun.com

1.4.2 Your Editor

git config –global core.editor emacs

1.4.3 Checking Your Settings

git config –list
git config user.name

1.5 Getting Help

```
man git
git <cammand> -h  # simple information help
git <command> --help  # verbose information help
```

2 DONE Git Basics

2.1 Getting a Git Repository

2.1.1 Initializing a Repository in an Existing Directory

git init

```
git init -h
```

This creates a new subdirectory named .git that contains all of your necessary repository files – a Git repository skeleton.

2.1.2 Cloning an Existing Repository

git clone https://github.com/libgit2/libgit2

That creates a directory named "libgit2", initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version.

git clone https://github.com/libgit2/libgit2 mylibgit

The target directory is called mylibgit.

2.2 Recording Changes to the Repository

Each file in your directory can be in one of two states: tracked or untracked.
Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged.
Untracked files are everyhting else – any files in your directory that were not in your last snapshot and are not in your staging area.

2.2.1 Checking the Status of Your Files

git status

```
git status -h
```

2.2.2 Tracking New Files

git add <file>
git add *

```
git add -h
```

The git add command takes a path name for either a file or a directory; if it's a directory, the command adds all the files in that directory recursively.

2.2.3 Staging Modified Files

git add <file>

2.2.4 Short Status

git status -s
git status –short

new files that aren't tracked have a ?? next to them;
new files that have been added to the staging area have an A;
modified files have an M;

2.2.5 Ignoring Files

emacs .gitignore (in the project directory)
(or .git/info/exclude)

2.2.6 Viewing Your Staged and Unstaged Changes

git diff

This command compares what is in your working directory with what is in your staging area.

to see what you've staged that will go into your next commit, you can use:
git diff –staged

This command compares your staged changes to your last commit.

2.2.7 Committing Your Changes

```
git commit
git commit -h
```

For an even more explicit reminder of what you've modified, you can pass the -v option to git commit.
Doing so also puts the diff of you change in the editor so you can see exactly what changes you're committing:

```
git commit -v
```

you can type your commit message inline:

```
git commit -m "Story 182: Fix benchmarks for speed"
```

2.2.8 Skipping the Staging Area

If you want to skip the staging area, adding the -a option to the git commit command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the git add part:

```
git commit -a
```

2.2.9 Removing Files

```
git rm <file>
git rm -h
```

remove a file from your tracked files and removes the file from your working directory.

If you modified the file and added it to the index already, you must force the removal with the -f option:
git rm -f <file>

to keep the file in your working tree but remove it from your staging area:
git rm –cached <file>

2.2.10 Moving Files

Git doesn't explicitly track file movement.
If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file.
However, Git is pretty smart about figuring that out after the fact.

If you want to rename a file in Git:
git mv <file_from> <file_to>

is same as:
mv <file_from> <file_to>
git rm <file_from>
git add <file_to>

2.3 Viewing the Commit History

The most basic and powerful tool to see what has happened:

```
git log
git log -h
git log --help
```

By default, with no argument, git log lists the commit made in that repository in reverse chronological order.

2.3.1 commain options

Options	Meaning
-p	shows the difference introduced in each commit (patch)
-2	limits the output to only the last two entries
–stat	to see some abbreviated stats for each commit
–pretty	change the log output to formats other than the default
–graph	adds a nice little ASCII graph showing your branch and merge history

```
git log --pretty=oneline
```

The most interesting option is format, which allows you to specify your own log output format. This is especially useful when you're generating output for machine parsing.

```
git log --pretty=format:"%h - %an, %ar : %s"
```

2.3.2 userfull option for git log –pretty=format

(lowercase: abbreviated)

%H	commit hash
%h	abbreviated commit hash
%T	tree hash
%t	abbreviated tree hash
%P	parent hashes
%p	abbreviated parent hashes
%an	author name
%ae	author email
%ad	author date
%ar	author date, relative
%cn	commit name
%ce	commite email
%cd	commit date
%cr	commit date, relative
%s	subject

The author is the person who originally wrote the work, whereas the commiter is the person who last applied the work.

2.3.3 options to limit the output of git log

-(n)	shwo only the last n commits
--since, --after	limite the commits to those made after the specified date
--until, --before	limit the commits to those made before the specified date
--author	only show commits in which the author entry matches the specified string
--commiter	only show commits in which the committer entry matches the specified string
--grep	only show commits with a commit message containing the string
-S	only show commits adding or removing code matching the string

```
git log --sine=2.weeks
```

note that: if you want to specify both author and grep options, you have to add –all-match or the command will match commits with either.

```
git log -Sfunction_name
```

If you specify a directory or file name, you can limit the log output to commits that introduced a change to those files.
This is always the last option and is generally preceded by double dashes(–) to seperate the paths from the options.

to see which commits modifying test files in the Git source code hisotry are merged and were committed by Junio Hamano in the month of October 2008:

```
git log --pretty="%h - %s" --author="Junio Hamano" --since="2008-10-01" --before="2008-10-01" --no-merges -- test
```

2.4 Undoing Things

One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to try that commit again, you can run commit with the –amend option:

```
git commit --amend
```

you end up with a single commit – the second commit replaces the result of the first.

2.4.1 Unstaging a Staged File

to unstage:

```
git reset HEAD <file>
git reset -h
git reset --help  # verbose help information
```

2.4.2 Unmodifying a Modified File

to discard changes in working directory:

```
git checkout -- <file>
git checkout -h
```

It's important to understand that git checkout – <file> is a dangerous command. Any changes you made to that file are gone.

2.5 Working with Remotes

2.5.1 Showing Your Remotes

```
git remote
git remote -v
git remote -h
```

2.5.2 Fetching and Pulling from Your Remotes

```
git fetch [remote-name]
git fetch -h
```

If you clone a repository, the command automatically adds that remote repository under the name "origin".

The git fetch command only downloads the data to your local repository - it doesn't automatically merge it with any of your work or modify what you're currently working on.

```
git pull [remote-name]
git pull -h
```

If your current branch is set up to track a remote branch, you can use the git pull command to automatically fetch and then merge that remote branch into your current branch.

2.5.3 Pushing to Your Remotes

```
git push [remote-name] [branch name]
git push -h
```

If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to fetch their work first and incorporate it into yours before you'll be allowed to push.

2.5.4 Inspecting a Rmote

```
git remote show [remote-name]
```

2.5.5 Removing and Renaming Remotes

```
git remote rename [remote_name_from] [remote_name_to]
git remote rm [remote_name]
```

2.6 Tagging

Git has the ability to tag specific points in history as being important.

2.6.1 Listing Your Tags

```
git tag
git tag -l <pattern>
git tag --list <pattern>

git tag -l "v1.8.5*"

git tag -h
```

2.6.2 Creating Tags

Git uses two main types of tags: lightweight and annotated.
A lightweight tag is very much like a branch that doesn't change - it's just a pointer to a specific commit.
Annotated tags are stored as full objects in the Git database. They are checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).

1. Annatated Tags

   ```
   git tag -a v1.4 -m "my version 1.4"
   # a: annotated
   # m: message
   # s: sign

   git show v1.0
   ```

2. Lightweight Tags

   to create a lightweight tag, don't supply tha -a, -s, or -m option:

   ```
   git tag v1.0-lw
   ```

2.6.3 Tagging Later

to see the checksums:

```
git log --pretty=oneline
```

to tag that commit, you specify the commit checksum (or part of it):

```
git tag -a v1.1 <checksum>
```

2.6.4 Sharing Tags

By default, the git push command doesn't transfer tags to remote servers.

```
git push origin [tagname]
git push origin --tags  # transfer all of your tags to the remote server that are not already there.
```

2.7 Git Aliases

Git doesn't automatically infer your command if you type it in partially. If you don't want to type the entire text of the Git commands, you can easily set up an alias for each command using git config.

```
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.st status
git config --global alias.last 'log -1 HEAD'
```

This technique can also be very useful in creating commands that you think should exist.

```
git config --global alias.unstage 'reset HEAD --'
```

This makes the following two commands equivalent:

```
git unstage fileA
git reset HEAD -- fileA
```

3 Git Branching

Branching means you diverge from the main line of development and continue to do work without messing with that main line.

3.1 Branching in a Nutshell

A branch in Git is simply a lightweight movable pointer to one of the commits.

The "master" branch in Git is not special branch. It is exactly like any other branch. The only reason nearly every repository has one is that the git init command creates it by dafault and most people don't bother to change it.

3.1.1 Creating a New Branch

git branch <branch_name>

What happens if you create a new branch?
Doing so creates a new pointer for you to move around.

How dose Git know what branch you're currently on?
It keeps a special pointer called HEAD.

The git branch command only created a new branch - it didn't switch to that branch.

to show where the branching pointers are pointing:
git log –oneline –decorate

3.1.2 Switching Branches

git checkout <branch_name>

If your working directory or staging area has uncommitted changes that conflict with the branch you're checking out, Git won't let you switch branches.

git log –oneline –decorate –graph –all

3.2 Basic Branching and Merging

3.2.1 Basic Branching

to create a branch and switch to it at the same time:
git checkout -b <branch_name>

to merge hotfix into master:
git checkout master
git merge hotfix

to delete a branch:
git branch -d <branch_name>

master hotfix
—

—
C2 <–— — — C4

Because the commit C4 pointed to by the branch hotfix you merged in was directly ahead of the commit C2 you're on, Git simply moves the pointer forward.
To phrase that another way, when you try merge one commit with a commit that can be reached by following the first commit's history, Git simplifies things by moving the pointer forward because there is no divergent work together – this is called a "fast-forward".

3.2.2 Basic Merging

git checkout master
git merge iss53

common master
ancestor master(result)
snapshot to merge into
C2 <-— — C4 — — — — — — — — — — C6
\ /
\ /
\ /
\ /
\ /
\ /
\ /
C3 –— — — C5
snapshot to merge into

iss53

Because the commit on the branch you're on isn't a direct ancestor of the branch you're mergingin, Git has to do some work.
In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.
Instead of moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it.

3.2.3 Basic Merge Conflicts

If you changed the same part of the same file differently in the two branches you are merging, Git won't be able to merge them cleanly.

to see the merge conflict:
git status

edit the unmerged files with emacs or vi and git add them.

to use a praphical tool to resolve these issues:
git mergetool

git config –global merge.tool emerge

C-]	emerge-abort
.	emerge-find-difference
<	emerge-scroll-left
>	emerge-scroll-right
^	emerge-scroll-down
a	emerge-select-A
b	emerge-select-B
e	emerge-edit-mode
f	emerge-fast-mode
j	emerge-jump-to-difference
l	emerge-recenter
m	emerge-mark-difference
n	emerge-next-difference
p	emerge-previous-difference
q	emerge-quit
v	emerge-scroll-up
|	emerge-scroll-reset
c a	emerge-copy-as-kill-A
c b	emerge-copy-as-kill-B
d a	emerge-default-A
d b	emerge-default-B
i a	emerge-insert-A
i b	emerge-insert-B
s a	emerge-auto-advance
s s	emerge-skip-prefers
x 1	emerge-one-line-window
x C	emerge-combine-versions-register
x c	emerge-combine-versions
x f	emerge-file-names
x j	emerge-join-differences
x l	emerge-line-numbers
x m	emerge-set-merge-mode
x s	emerge-split-difference
x t	emerge-trim-difference
x x	emerge-set-combine-versions-template

3.3 Branch Management

The git branch command does more than just create and delete branches.

to get a simple listing of your current branches:
git branch

to see the last commit on each branch:
git branch -v

The usefull –merged and –no-merged can filter this lists to branches that you have or have not yet merged into the branch you're currently on:
git branch –merged

Branches on the list without the * in front of them are generally fine to delete.

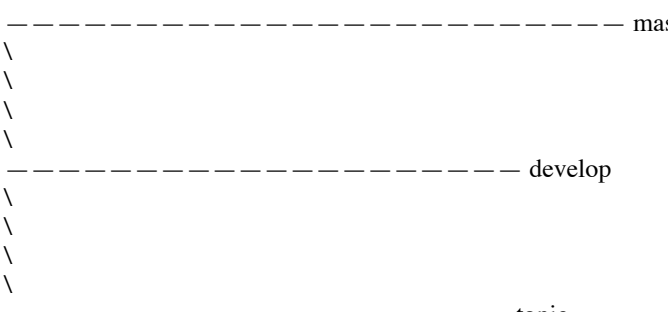
git branch –unmerged (deprecated)
git branch –no-merged (new)

3.4 Branching Workflows

3.4.1 Long-Running Branches

Many Git developers have a workflow that having only code that is entirely stable in their master branch. They have another parallel branch named develop that they work from or use to test stability.

It's generally easier to think about them as work silos, where sets of commits graduate to a more stable silo when they're fully tested.



The idea is that your branches are at various levels of stability; when they reach a more stable level, they're merged into the branch above them.

3.4.2 Topic Branches

A topic branch is a short-lived branch that you create and use for a single particular feature or related work.

It's important to remember when you're doing all this that these branches are completely local.

3.5 Remote Branches

Remote references are references (pointers) in your remote repositories, including branches, tags, and so on.

To get a full list of remote references:
git ls-remote [remote]
git remote show [remote]

a more common way is to take advantage of remote-tracking branches:
Remote-tracking branches are references to the state of remote branches.
They are local references that you can not move; they are moved automatically for you whenever you do any network communication.
Remote-tracking branches act as bookmark to remind you where the branches in your remote repositories were the last time you connected to them.

form:
(remote)/(branch)

note:
Just like the branch name "master" does not have any special meaning in Git, neither does "origin".
"origin" is the default name for a remote when you run git clone.

3.5.1 Pushing

git push <remote> <remote_branch>
git push <remote> <local_branch>:<remote_branch>

git push origin serverfix
This is a shortcut. Git automatically expands the serverfix branchname out to
serverfix:serverfix

git push origin serverfix:serverfix
Take my serverfix and make it the remote's serverfix

When you do a fetch that brings down new remote-tracking branches, you don't automatically have local, editable copies of them.

git fetch origin (new branch serverfix is pulled suppose)
To merge this work into your current working branch, you can run

git merge origin/serverfix

If you want your own serverfix branch that you can work on, you can base it off your remote-tracking branch:
git checkout -b serverfix origin/serverfix
b: branch

3.5.2 Tracking Branches

Checking out a local branch from a remote-tracking branch automatically creates what is called a "tracking branch" (and the branch it tracks is called an "upstream branch"). Tracking branches are local branches tha have a direct relationship to a remote branch.

If you are on a tracking branch and type git pull, Git automatically knows which server to fetch from and branch to merge into.

You can set up other tracking branches if you wish:
git checkout -b [branch] [remotename]/[remote_branch]
shorthand:
git checkout –track origin/serverfix

To have different name with origin:
git checkout -b <different_branchname> <origin>/<branch>

If you already have a local branch and want to set it to a remote branch:
git branch -u <remote>/<branch>
or
git branch –set-upstream-to <remote>/<branch>

To see what tracking branches you have set up:
git branch -vv
It is important to note that these numbers are only since the last time you fetch from each server. This command does not reach out to the server.

3.5.3 Pulling

git pull

This will look up what server and branch your current is tracking, fetching from that server and then try to merge in that remote branch.

3.5.4 Deleting Remote Branches

git push <remote> –delete <remote_branch>

Basically all this does is remove the pointer from the server. The Git server will generally keep the data there for a while until a garbage collection runs.

3.6 Rebasing

In Git, there are two main ways to integrate chages from one branch into another:
the merge and the rebase

3.6.1 The Basic Rebase

C4
/
/ experiment
/
/
/
/
/
/
/
C1 <— — — — C2 <- — — — — —C3

master

you can take the patch of the change that was introduced in C4 and reapply it on top of C3. In Git this is called rebasing.
With the rebase command, you can take all the changes that were committed on one branch and replay them on another one.

git checkout experiment
git rebase master

It works by going to the common ancestor of the two branches, getting the diff introduced by each commit of the branch you're on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

C1 <-— — — — C2 <-— — — — — — C3 <-— — — — — — C4'
master experiment

At this point, you can go back to the master branch and do a fast-forward merge
git checkout master
git merge experiment

There is no difference in the end product of the integration (comparing to merge), but rebasing makes for a cleaner history.

Rebasing replays changes from one line of work onto another inthe order they are introduced, whereas merging takes the endpoints and merges them together.

3.6.2 More Interesting Rebases

You can have your rebase replay on something other than the rebase target branch.

master
C1 <-— — C2 <-— — C5 <— — — C6
\
\
\
\
\
\
\ server
C3 <— — — C4 <-— — C10
\
\
\
\
\
\
\
\
\ C8 <-— — — C9
client

You can take the changes on client that aren't on server (C8 and C9) and replay them on your master branch by:
git rebase –onto master server client.

This basically says, "checkout out the client branch, figure out the patches from the common ancestor of the client and server branches, and then replay them onto master.

master client
C1 <— — C2 <— — C5 <— — C6 <-— C8' <-— C9'
\
\
\
\
\
\ C3 <-— — C4 <— — C10
server

The Perils of Rebasing
Do not rebase commits that exist outside your repository.

3.6.3 Rebase vs. Merge

One point of view is that your repository's commit history is a record of what actually happened.
The opposing point of view is that the commit history is the story of how your project was made.

It's up to you to decide which one is best for your particular situation.

In general the way to get the best of both world is to rebase local changes you've made but haven't shared yet before you push them in order to clean up your story, but never rebase anything you've pushed somewhere.

4 Git on the Server

Running a Git server is straightforward.
First, you choose which protocols you want your server to communicate with.
Second, setups using those protocols and get your server running with them.

A remote repository is generally a bare repository - a Git repository that has no working directory.
In the simplest terms, a bare repository is the contents of your project's .git directory and nothing else.

4.1 The Protocols

4.1.1 Local Protocol

the most basic is the Local Protocol, in which the remote repository is in another directory on disk.

1. When to Use

   If you have a shared mounted filesystem, then you can clone, push to, and pull from a local file-based repository.

2. form

   git clone /opt/git/project.git
   or
   git clone file:///opt/git/project.git

   If you just specify the path, Git tries to use hardlinks or directly copy the files it needs.
   If youu specify file://, Git fires up the process that it normally uses to transfer data over a network which is generally a lot less efficient method of transferring the data. The main reason that specify the file:// prefix is if you want a clean copy of the repsitory with extraneous references or objects left out.

3. The Pros

   simple and use existing file permissions and network access.

4. The Cons

   shared access is generally more difficult to set up and reach from multiple locations than basic network access.
   this protocol does not protect the repository against accidental damage.

4.1.2 The Http Protocols

1. smart http

   The "smart" http protocol opertates very similarly to the ssh or git protocols but runs over standard http/s prots and can use various http authentication mechanisms.
   read and write

2. dumb http

   readonly

3. The Pros

   The simplicity of having a single URL for all types of access and having the server prompt only when authentication is needed makes things very easy for the end user.

   a very fast and efficient protocol similar to the ssh one.

4. The Cons

Git over HTTP/S can a little more tricky to set up compared to SSH on some servers.

4.1.3 The SSH Protocol

A common transport protocol for Git when self-hosting is over SSH.

1. form

   git clone ssh://user@server/project.git
   or
   git clone user@server:project.git

2. The Pros

   First, relatively easy to set up.
   Second, access over SSH is secure.
   Third, like the HTTP/S, Git and Local protocols, SSH is efficient, making the data as compact as possible before transferring it.

3. The Cons

   you can't server anonymous access of your repository over it.

4.1.4 The Git Protocol

This is a special daemon that comes packaged with Git; it listens on a dedicated port(9418).
In order for a repository to be served over the Git protocol, you must create the git-daemon-export-ok file - the daemon won't serve a repository without that file in it - but other than that there is no security

1. The Pros

   fatest network transfer protocol available.

2. The cons

   the lack of authentication.

4.2 Getting Git on a Server

4.2.1 First, get a bare repository

In order to initially set up any Git server, you have to export an existing repository into a new bare repository - a repository that doesn't contain a working directory.
git clone –bare my_project my_project.git
or
git init –bare

4.2.2 Second, copy it to a server

put the bare repository on a server and set up your protocols.

suppose, the server called git.example.com, store all your Git repository under the /srv/git directory:
scp -r my_project.git user@git.example.com:/srv/git

At this point, other users who have ssh access to the same server which has read-access to the /srv/git directory can clone your repository:
git clone user@git.example.com:/srv/git/my_project.git

If a user SSHs into a server and has write access to the /srv/git/my_project.git directory, they will alse automactically have push access.

Git will automatically add group write permissions to a repository properly:
cd /srv/git/my_project.git
git init –bare –shared

4.2.3 note

to collaborate with a couple of people on a private project, all you need it a SSH sever and a bare repository.

4.2.4 Small Setups

One of the most complicated aspects of setting up a Git server is user management.

There are a few ways you can give access to everyone on your team:
First, set up accounts for everybody.
Second, create a single git user on the machine, ask every user who is to have write access to send you an SSH public key, and add that key to the ~/.ssh/authorized_keys file of your new git user.
Third, have your SSH server authenticate from an LDAP server or some other centralized authentication source.

As long as each user can get shell access on the machine, any SSH authentication mechanism you can think of should work.

4.3 Setting Up the Server

sudo adduser git
su git
cd
mkdir .ssh && chmod 700 .ssh
touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
cat *tmp/id_rsa.michael.pub* >> ~.ssh/authorized_keys
cd /opt/git
mkdir project.git
cd project.git
git init –bare

You can easily restrict the git user to only doing Git activities with a limited shell tool called git-shell that comes with Git.

cat /etc/shell # see if 'git-shell' is already in there
which git-shell # make sure git-shell is installed on your system
sudo vim /etc/shells # add the path to git-shell

sudo chsh git # enter the path ot git-shell

4.4 Git Daemon

yum install git-daemon

This is common choice for fast, unauthenticated access to your Git data.

git daemon –reuseaddr –base-path=/opt/git/ *opt/git*

–reuseaddr allows the server to restart without waiting for old connections to timeout, –base-path option allows people to clone projects without specifying the entire path, and the path at the end tells the Git daemon where to look for repositories to export.

If you are running a firewall, you will also need to punch a hole in it at port 9418.

cd /path/to/project.git
touch git-daemon-export-ok

4.5 Smart HTTP (not done)

4.5.1 Ubuntu

sudo apt-get install apache2 apache2-utils (on Ubuntu)

4.5.2 CentOS

yum install httpd
yum install gitweb

chgrp -R apache /home/git/project.git

4.6 GitLab

GitLab is a database-backed web application.

4.6.1 Install

1. Install and configure the necessary dependencies

   yum install -y curl policycoreutils-python openssh-server
   systemctl enable sshd
   systemctl start sshd
   firewall-cmd –permanent –add-service=http
   systemctl reload firewalld

   Next, install Postfix to send notification emails.
   yum install postfix
   systemctl enable postfix
   systemctl start postfix

2. Add the GitLab package repository and intall the package

   curl https://packages.gitlab.com/install/repositories/gitlab/gitlab-ee/script.rpm.sh | sudo bash

   Next, install the GitLab package.

   sudo EXTERNAL_URL="http://chyson.net" yum install -y gitlab-ee

3. Browse to the hostname and login

   On your first visit, you'll be redirected to a password reset screen. Provide the password for the initial administrator account and you will be redirected back to the login screen. Use the default account's username root to login.

Author: Mingming Li
Created: 2019-10-15 Tue 13:56
Emacs 25.2.2 (Org mode 8.2.10)
Validate