目录

[安装 3](#_Toc526857862)

[配置 3](#_Toc526857863)

[个人信息 3](#_Toc526857864)

[查看配置信息 3](#_Toc526857865)

[使用 4](#_Toc526857866)

[创建版本库init 4](#_Toc526857867)

[克隆clone 4](#_Toc526857868)

[添加到暂存区add 5](#_Toc526857869)

[查看状态status 5](#_Toc526857870)

[提交commit 5](#_Toc526857871)

[注意 5](#_Toc526857872)

[从暂存区回退reset 5](#_Toc526857873)

[移除rm 6](#_Toc526857874)

[mv 6](#_Toc526857875)

[撤销修改checkout -- 6](#_Toc526857876)

[三个会退情景 7](#_Toc526857877)

[日志log 7](#_Toc526857878)

[pretty 7](#_Toc526857879)

[命令记录reflog 8](#_Toc526857880)

[标签tag 8](#_Toc526857881)

[rebase 9](#_Toc526857882)

[流程 9](#_Toc526857883)

[合并多次commit 9](#_Toc526857884)

[cherry-pick 9](#_Toc526857885)

[stash 10](#_Toc526857886)

[git stash 10](#_Toc526857887)

[git stash list 10](#_Toc526857888)

[git stash pop [--index] [<stash>] 10](#_Toc526857889)

[git stash branch <branchname> <stash> 10](#_Toc526857890)

[其他指令 10](#_Toc526857891)

[分支管理 11](#_Toc526857892)

[创建branch 11](#_Toc526857893)

[创建空分支 11](#_Toc526857894)

[创建子分支 11](#_Toc526857895)

[切换checkout 11](#_Toc526857896)

[注意 11](#_Toc526857897)

[列出branch 11](#_Toc526857898)

[删除分支 –d 12](#_Toc526857899)

[重命名分支 -m 12](#_Toc526857900)

[清理分支 12](#_Toc526857901)

[分支合并merge 12](#_Toc526857902)

[查看代码 12](#_Toc526857903)

[当前代码改动 12](#_Toc526857904)

[某次提交的改动 12](#_Toc526857905)

[查看某行代码的改动 13](#_Toc526857906)

[github 13](#_Toc526857907)

[规范 13](#_Toc526857908)

[ssh 13](#_Toc526857909)

[配置ssh位置 13](#_Toc526857910)

[生成公钥 13](#_Toc526857911)

[添加git remote add [shortname] [url] 14](#_Toc526857912)

[绑定 14](#_Toc526857913)

[测试 14](#_Toc526857914)

[克隆 14](#_Toc526857915)

[查看当前的远程库 14](#_Toc526857916)

[删除 14](#_Toc526857917)

[提取远程仓库 14](#_Toc526857918)

[下载分支fetch 14](#_Toc526857919)

[下载并合并pull 15](#_Toc526857920)

[推送到远程仓库 15](#_Toc526857921)

[流程 15](#_Toc526857922)

[规范 15](#_Toc526857923)

[type 16](#_Toc526857924)

[scope 16](#_Toc526857925)

[subject 16](#_Toc526857926)

[body 16](#_Toc526857927)

[footer 17](#_Toc526857928)

[子模块 17](#_Toc526857929)

[更新 17](#_Toc526857930)

[子模块冲突 18](#_Toc526857931)

[子模块分支问题 18](#_Toc526857932)

[.gitmodules 18](#_Toc526857933)

[常见错误 18](#_Toc526857934)

[failed to push some refs to git 18](#_Toc526857935)

[src refspec xxx does not match any 19](#_Toc526857936)

[仓库的文件修改过 19](#_Toc526857937)

[丢失文件 19](#_Toc526857938)

常用分支

Dev：开发

Bug：调试

删除分支

-D强制

-d非强制

免费的git网站：Git#osc

mac中参数不能在最后面，紧跟着命令。

# 安装

centos自带Git，7.x版本自带git 1.8.3.1（应该是，也可能不是），  
安装新版本之前需要使用yun remove git卸载（安装后卸载也可以）。

# 配置

配置在.gitconfig 文件夹里面

## 个人信息

Git 为你的每一个提交都记录你的名字与电子邮箱地址，所以第一步需要配置用户名和邮箱地址。

配置个人的用户名称和电子邮件地址：

$ git config --global user.name "a755956503"

$ git config --global user.email 755956503@qq.com

如果用了 **--global** 选项，那么更改的配置文件就是位于你用户主目录下的那个，以后你所有的项目都会默认使用这里配置的用户信息。

如果要在某个特定的项目中使用其他名字或者电邮，只要**去掉 --global 选项重新配置即可**，新的设定保存在当前项目的 .git/config 文件里。

## 查看配置信息

要检查已有的配置信息，可以使用 git config --list 命令：

$ **git config --list**

http.postbuffer=2M

user.name=runoob

user.email=test@runoob.com

有时候会看到重复的变量名，那就说明它们来自不同的配置文件（比如 /etc/gitconfig 和 ~/.gitconfig），不过最终 Git 实际采用的是最后一个。

这些配置我们也可以在 **~/.gitconfig** 或 **/etc/gitconfig** 看到，如下所示：

**vim ~/.gitconfig**

显示内容如下所示：

[http]

postBuffer = 2M

[user]

name = runoob

email = test@runoob.com

也可以直接查阅某个环境变量的设定，只要把特定的名字跟在后面即可，像这样：

$ **git config user.name**

runoob

# 使用

## 创建版本库init

$ mkdir learngit

$ cd learngit

$ git init

## 克隆clone

git clone <repo> <directory>

**directory:**本地目录。

$ git clone git://github.com/schacon/grit.git mygrit

Git 会按照你提供的 URL 所指示的项目的名称创建你的本地项目目录。 通常就是该 URL 最后一个 / 之后的项目名称。如果你想要一个不一样的名字， 你可以在该命令后加上你想要的名称。

## 添加到暂存区add

$ git add readme.txt

## 查看状态status

$ git status -s

AM README

A hello.php

**-s** 参数，以获得简短的结果输出。如果没加该参数会详细输出内容

"**AM**" 状态的意思是，这个文件在我们将它添加到缓存之后又有改动。改动后我们在执行 git add 命令将其添加到缓存中：

?? 表示还没有加入暂存区。

**status不显示已经提交后没有再修改的文件。**

## 提交commit

$ git add file1.txt

$ git add file2.txt file3.txt

$ git commit -m "add 3 files."

-m后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样你就能从历史记录里方便地找到改动记录。

commit可以一次提交很多文件，所以你可以多次add不同的文件

如果你觉得 git add 提交缓存的流程太过繁琐，Git 也允许你用 -a 选项跳过这一步。命令格式如下：

git commit -a

git commit -am '修改 hello.php 文件'

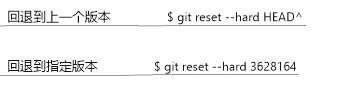
### 注意

都改好了再commit，因为在push时，所有的commit都会push上去。

## 从暂存区回退reset

**从仓库回退到暂存区，不是工作空间**

"取消缓存"的意思就是将缓存区恢复为我们做出修改之前的样子。



用**HEAD**表示当前版本，也就是最新的提交3628164...882e1e0（注意我的提交ID和你的肯定不一样），上一个版本就是**HEAD^**，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

版本号没必要写全，前几位就可以了，Git会自动去找。

指定版本可以是会退之后的版本，所以有后悔药可以吃。

//仅仅只是撤销已提交的版本库，不会修改暂存区和工作区

git reset --soft 版本库ID

//仅仅只是撤销已提交的版本库和暂存区，不会修改工作区

git reset --mixed 版本库ID

//彻底将工作区、暂存区和版本库记录恢复到指定的版本库

git reset --hard 版本库ID

## 移除rm

将文件从缓存区和你的硬盘中（工作目录）删除。

如果你要在工作目录中留着该文件，可以使用 **git rm --cached**：

文件夹用 git rm -r /a/b

**如果要删除仓库里的文件，需要用reset**

## mv

git mv 命令做得所有事情就是 **git rm --cached** 命令的操作， 重命名磁盘上的文件，然后再执行 git add 把新文件添加到缓存区。

git mv README README.md

## 撤销修改checkout --

git checkout -- readme.txt

没有--，就变成了“切换到另一个分支”的命令

一种是readme.txt自修改后还**没有被放到暂存区**，现在，撤销修改就回到和版本库一模一样的状态；

一种是readme.txt已经**添加到暂存区后**，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

## 三个会退情景

场景1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令git checkout -- file。

场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令git reset HEAD file，就回到了场景1，第二步按场景1操作。

场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考[版本回退](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000)一节，不过前提是没有推送到远程库。

## 日志log

查看某个分支的记录。

### pretty

如果嫌输出信息太多，看得眼花缭乱的，可以试试加上--pretty=oneline参数：

oneline 将 每个提交 放在一行显示，这在提交数很大时非常有用。另外还有short，full 和fuller 可以用

最有意思的是 format，可以定制要显示的记录格式，这样的输出便于后期编程提取分析，像这样git log --pretty=format:"%h - %an, %ar : %s"

选项 说明

%H 提交对象（commit）的完整哈希字串

**%h 提交对象的简短哈希字串**

%T 树对象（tree）的完整哈希字串

%t 树对象的简短哈希字串

%P 父对象（parent）的完整哈希字串

%p 父对象的简短哈希字串

**%an 作者（author）的名字**

%ae 作者的电子邮件地址

%ad 作者修订日期（可以用 -date= 选项定制格式）

%ar 作者修订日期，按多久以前的方式显示

%cn 提交者(committer)的名字

%ce 提交者的电子邮件地址

%cd 提交日期

%cr 提交日期，按多久以前的方式显示

**%s 提交说明**

用 --graph 选项，查看历史中什么时候出现了分支、合并。

用 '--reverse'参数来逆向显示所有日志。

如果只想查找指定用户的提交日志可以使用命令：git log --author ,

-6 最近6次

如果你要指定日期，可以执行几个选项：--since 和 --before，但是你也可以用 --until 和 --after。

git log --oneline --before={3.weeks.ago} --after={2010-04-18} --no-merges

当我们执行 git log --decorate 时，我们可以看到我们的标签了

**-p 选项展开显示每次提交的内容差异**

--stat，仅显示简要的增改行数统计

## 命令记录reflog

reflog查看项目所有分支所有记录，包括merge,切换分支这些。

$ git reflog

ea34578 HEAD@{0}: reset: moving to HEAD^

3628164 HEAD@{1}: commit: append GPL

ea34578 HEAD@{2}: commit: add distributed

cb926e7 HEAD@{3}: commit (initial): wrote a readme file

## 标签tag

git tag -a v1.0

a 选项意为"创建一个带注解的标签"。 不用 -a 选项也可以执行的，但它不会记录这标签是啥时候打的，谁打的，也不会让你添加个标签的注解。 我推荐一直创建带注解的标签。

当我们执行 git log --decorate 时，我们可以看到我们的标签了：

git tag -a v0.9 85fc7e7

显示所有标签

git tag

指定标签信息命令：

git tag -a <tagname> -m "w3cschool.cc标签"

PGP签名标签命令：

git tag -s <tagname> -m "w3cschool.cc标签"

## rebase

### 流程

git rebase

while(存在冲突) {

git status

找到当前冲突文件，编辑解决冲突

git add -u

git rebase --continue

if( git rebase --abort )

break;

}

-i: 进入交互模式，控制合并过程。

git rebase --skip

作用：用来替代 merge，用来修改 commit 历史(比如合并多个commit)

### 合并多次commit

git rebase -i HEAD~4 将几次commit合并

git rebase -i master 基于master分支合并commit

## cherry-pick

摘的只是当时提交修改的代码，不是对两个分支完全比对。

git cherry-pick 38361a68

用于把另一个本地分支的commit修改应用到当前分支

-x 保留原提交者信息

 git cherry-pick <start-commit-id>..<end-commit-id>

cherry-pick一个区间的commit(左开右闭，)

git cherry-pick <start-commit-id>^..<end-commit-id>

闭区间

## 比较代码

git diff [branchA] [branchB] > xx.diff

也可以用绝对路径。

## HEAD

HEAD 就是你当前的工作目录所处的位置，可以用 checkout 命令改变 HEAD 指向的位置。注意 HEAD 不一定指向一个分支，也可以指向一个 commit

detached HEAD（checkout到某次提交以后，就会出现）。

Git 1.8.5版本之后，HEAD有一个缩写形式@，确实可以省略几个字符。

# stash

保存和恢复只是针对修改的代码，不是整个仓库一起。所以可以在一个分支保存，在另一个分支恢复，转移代码

## git stash

git stash save -u "xxx"

**对于在项目里加入了代码新文件的开发来说，-a选项才会将新加入的代码文件同时放入暂存区。**

**-u: -a会将忽略的文件，比如Nodemodules里的库文件也一起保存，如果不想保存，需要加上-u,** **表示untracked files**

对当前的暂存区和工作区状态进行保存，使两者回到HEAD状态。

git stash [save message] [-k|--no-keep-index] [--patch]

比如：*git stash save "这是保存的进度"*

-k和--no-keep-index指定保存进度后，是否重置暂存区

--patch 会显示工作区和HEAD的差异,通过编辑差异文件，排除不需要保存的内容。和git add -p命令类似

## git stash list

查看保存列表

## git stash pop [--index] [<stash>]

恢复指定状态

--index 参数：不仅恢复工作区，还恢复暂存区

<stash> 指定恢复某一个具体进度。如果没有这个参数，默认恢复最新进度

## git stash branch <branchname> <stash>

基于进度创建分支

## 其他指令

git stash apply [--index] [<stash>] 不删除已恢复的进度，其他同git stash pop   
git stash drop [<stash>] 删除某一个进度，默认删除最新进度   
git stash clear 删除所有进度

# 分支管理

切换分支的时候，Git 会用该分支的最后提交的快照替换你的工作目录的内容， 所以多个分支不需要多个目录。

当你执行 git init 的时候，缺省情况下 Git 就会为你创建"master"分支。

## 创建branch

git branch (branchname)

### 创建空分支

创建：

git checkout --orphan doc

--orphan 不会有其他分支的提交记录。

删除内容：

git rm -rf .

## 创建子分支

git checkout -b 新分支名 老分支名（这种写法的意思为基于老分支创建新的分支）

## 切换checkout

git checkout (branchname)

git checkout -b dev

-b参数表示创建并切换

### 注意

修改东西后不commit在切换分支时会自动带过去。

所以切换分支时都要commit或stash当前的工作区

## 列出branch

列出所有分支，当前分支前面会标一个\*号。

git branch 默认只显示本地分支。

-a 所有

-r 只看远程

### 删除分支 –d

git branch -d testing

-D：强制删除

### 重命名分支 -m

git branch -m devel develop

### 清理分支

git remote prune origin

清理远程不存在的分支。

### 分支合并merge

合并某分支到当前分支：git merge <name>

# 查看代码

## 当前代码改动

* 尚未缓存的改动：**git diff**
* 查看已缓存的改动： **git diff --cached**
* 查看已缓存的与未缓存的所有改动：**git diff HEAD**
* 显示摘要而非整个 diff：**git diff --stat**

## 某次提交的改动

只是commit的改动。。。

git show <commit-hashId> 显示某次提交的修改内容

git show <commit-hashId> filename 显示某次提交的某个内容的修改信息。

## 查看某行代码的改动

vscode有一个gitlens插件。

git blame -L 160,+10 sha1\_file.c

ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700 160)}

ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700 161)

0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700 162)/\*

git blame -L

"-L"参数在命令(blame)中指定开始和结束行

# github

如果不使用公钥，则每次push都要使用密码。

有两种方式

已经在本地写了项目：remote

本地还没有建项目 clone

## 规范

把远程主仓库取名为main

把自己fork的远程仓库取名为origin

## ssh

### 配置ssh位置

换电脑以后用

ssh不匹配往往是这个原因。

ssh-add ~/.ssh/youraccount\_rsa

ssh-add ~/.ssh/241337

### 生成公钥

$ ssh-keygen -t rsa -C "youremail@example.com"

### 添加git remote add [shortname] [url]

git remote add menzhenbao https://github.com/a755956503/menzhenbao

### 绑定

ssh -T [git@github.com](mailto:git@github.com)

### 测试

ssh -T git@github.com

### 其他操作

修改ssh密码

ssh-keygen -f id\_dsa（私钥文件名） -p

## 克隆

git clone [git@github.com:a755956503/menzhenbao.git](mailto:git@github.com:a755956503/menzhenbao.git)

## 查看当前的远程库

git remote -v

-v 参数，你还可以看到每个别名的实际链接地址。

## 删除

git remote remove <name>

## 提取远程仓库

### 下载分支fetch

git fetch

git fetch origin

该命令执行完后需要执行git merge 远程分支到你所在的分支。

git merge origin/master

### 下载并合并pull

git pull <远程主机名> <远程分支名>:<本地分支名>

git pull

git pull --rebase origin master

该命令就是在执行 git fetch 之后紧接着执行 git merge 远程分支到你所在的任意分支。

使用git pull时默认是merge， **加 --rebase参数使其使用rebase方式**

**(rebase是在后面增加commit的方式，尽量使用这种方式，因为这样可以避免本地代码污染远程代码，因为merge是合并)**

假设你配置好了一个远程仓库，并且你想要提取更新的数据，你可以首先执行 **git fetch [alias]** 告诉 Git 去获取它有你没有的数据，然后你可以执行 **git merge [alias]/[branch]** 以将服务器上的任何更新（假设有人这时候推送到服务器了）合并到你的当前分支。

## push

git push <remote> <local branch name>:<remote branch to push into>

git push origin HEAD:master

推送本地最新的位置

git push origin :master

只有冒号，会删除远程对应的分支！！！

## 流程

先本地add,commit

pull最新代码，对代码进行合并。

push上去，这时候如果没有冲突就不用add,commit了，因为是分支合并

## 规范

<type>(<scope>) : <subject>

<空行>

<body>

<空行>

<footer>

### type

feat :新功能

fix :修复bug

doc : 文档改变

style : 代码格式改变

refactor :某个已有功能重构

perf :性能优化

test :增加测试

build :改变了build工具 如 grunt换成了 npm

revert: 撤销上一次的 commit

### scope

用来说明此次修改的影响范围 可以随便填写任何东西，commitizen也给出了几个 如：location 、browser、compile，不过我推荐使用

all ：表示影响面大 ，如修改了网络框架  会对真个程序产生影响

loation： 表示影响小，某个小小的功能

module：表示会影响某个模块 如登录模块、首页模块 、用户管理模块等等

### subject

用来简要描述本次改动，概述就好了

### body

具体的修改信息 应该尽量详细

### footer

放置写备注啥的，如果是 bug ，可以把bug id放入

最后生成的 commit 信息大概是这样的  简洁又明了



# 子模块

增加子模块以后会生成一个.gitmodules文件理子模块。

git submodule add https://github.com/chaconinc/DbConnector

git submodule add ~/submd/repos/lib1.git libs/lib1

git diff 默认不跟踪子模块，要查看的话传递 --submodule 选项。如果你不想每次运行 git diff 时都输入 --submodle，那么可以将 diff.submodule 设置为 “log” 来将其作为默认行为。git config --global diff.submodule log

父模块判断子模块是否有变化是根据提交记录，和分支无关，所以合并父模块时，子模块只要都是最新的提交，就不会有冲突

## 更新

git submodule init 用来初始化本地配置文件，

git submodule update 则从该项目中抓取所有数据并检出父项目中列出的合适的提交。

如果给 git clone 命令传递 --recursive 选项，它就会自动初始化并更新仓库中的每一个子模块。

git submodule update --remote（默认是master）

相当于进入子模块执行pull操作。

推送父模块时不会推送子模块。也就是子模块修改的代码不会更新到线上的父模块对应目录。所以要先提交子模块再提交父模块

对父模块来说，子模块只是一条引用记录，所以需要先提交子模块，父模块对应的引用才有变化。

### 子模块冲突

Submodule src/bsl e72ee487b..8a8da5b81:

可以进入子模块，reset到当前分支匹配的版本，上面两个sha-A表示两个冲突的版本。

Submodule src/bsl contains modified content

### 子模块分支问题

git submodule update --remote时， Git 将会获得这些改动并更新子目录中的文件，但是会将子仓库留在一个称作 “游离的 HEAD” 的状态。 这意味着没有本地工作分支（例如 “master”）跟踪改动。 所以你做的任何改动都不会被跟踪。 解决方法：git submodule update --remote --merge ，合并到子模块当前分支（和父模块是什么分支没关系）（前提是子模块定指定了分支，）

## .gitmodules

git config -f .gitmodules submodule.DbConnector.branch stable

[submodule "DbConnector"]

path = DbConnector

url = https://github.com/chaconinc/DbConnector

+ branch = stable

# 常见错误

## failed to push some refs to git

在使用git 对源代码进行push到gitHub时可能会出错，信息如下

　　此时很多人会尝试下面的命令把当前分支代码上传到master分支上。

　　$ git push -u origin master

　　但依然没能解决问题

　　出现错误的主要原因是github中的README.md文件不在本地代码目录中

　　可以通过如下命令进行代码合并【注：pull=fetch+merge]

　　git pull --rebase origin master

执行上面代码后可以看到本地代码库中多了README.md文件

## src refspec xxx does not match any

可能是没找到本地对应的分支，所以手动指定。

git push origin HEAD:branch

## non-fast-forward

## 仓库的文件修改过

error: failed to push some refs to 'git@gitlab.intra.knownsec.com:tanl/weekly.git'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

先pull代码，merge后再push。

git push -f origin master.

强制push

## 丢失文件

使用git fsck --lost-found这个命令，通过.git/lost-found/other这个路径，你可以恢复任何你git add过的文件！再通过find .git/objects -type f | xargs ls -lt | sed 60q这个命令，你就可以找到最近被你add到本地仓库的60个文件，综上所述，希望对你有所帮助！（或者直接到.git/lost-found/other中寻找）

当然咯，也就是but的but，不幸中的不幸，如果你没有git add过的文件呢，被git reset --hard这个命令整过之后呢，就如你自己执行delete命令一样，再也尸骨难寻啦（也就是真的毛都不剩了！！默哀三秒）！！！！