# Git使用说明文档

## 简介

Git 是一个开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。

Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。

Git 与常用的版本控制工具 CVS, Subversion 等不同，它采用了分布式版本库的方式，不必服务器端软件支持。

**仓库介绍**

Git是分布式版本控制系统（Distributed Version Control System，简称 DVCS），分为两种类型的仓库： 本地仓库和远程仓库

本地仓库：是在开发人员自己电脑上的Git仓库

远程仓库：是在远程服务器上的Git仓库

Clone：克隆，就是将远程仓库复制到本地

Push：推送，就是将本地仓库代码上传到远程仓库

Pull：拉取，就是将远程仓库代码下载到本地仓库

**工作区、暂存区以及版本库的概念**

工作区：包含.git 隐藏文件夹的目录，就是工作目录，主要是用于存放开发的代码

暂存区：.git 目录里的index文件就是暂存区，临时保存修改文件的地方

版本库：.git 隐藏文件夹就是版本库，版本库中存储了很多配置信息，日志信息和文件版本信息等

注：git工作区下的文件存在两种状态：

untracked 未跟踪（未被纳入版本控制）

tracked 已跟踪（被纳入版本控制）

Unmodified 未修改状态

Modified 已修改状态

Staged 已暂存状态

## 安装git

Linux平台安装

Debian/Ubuntu ： # apt-get install git

Win平台安装

[git win版本下载安装链接](https://gitforwindows.org/)

下载exe安装文件并运行，完成后会有两个工具：Git Bash (命令行工具) \ Git GUI (图形化界面)

在开始菜单里找到"Git"->"Git Bash"，会弹出 Git 命令窗口，你可以在该窗口进行 Git 操作。

## 常用命令

### 3.1 Git配置

Git 提供了一个叫做 git config 的工具，专门用来配置或读取相应的工作环境变量。

可以使用git config 命令配置具体信息，配置信息保存~/.gitconfig文件中

设置用户信息(每次git提交都会使用该用户信息)

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

$ git config --global user.email test@runoob.com

查看配置信息

$ git config --list

user.name=runoob

user.email=test@runoob.com

查看指定的环境变量设置

$ git config user.name

runoob

### 3.2 创建远程仓库

远程仓库：可以借助互联网上提供的一些代码 托管服务来实现，其中比较常用的有GitHub、码云、GitLab等

gitHub（ 地址：<https://github.com/> ）是一个面向开源及私有软件项目 的托管平台，因为只支持Git 作为唯一的版本库格式进行托管，故名 gitHub

码云（地址： <https://gitee.com/>）是国内的一个代码托管平台，由于服务器在国内，所以相比于GitHub，码云速度会更快

GitLab （地址：<https://gitlab.com/gitlab-com>）是一个用于仓库管理系统 的开源项目，使用Git作为代码管理工具，并在此基础上搭建起来的web服务

### 3.3 创建本地仓库

本地初始化创建Git仓库

1创建一个空目录作为本地的git仓库

2 进入目录

3 执行 git init

4 查看当前目录是否有.git 隐藏文件夹，如果有则说明创建本地仓库操作成功

或者克隆远程仓库到本地

# git clone <远程git仓库地址>

# git clone <https://github.com/AleoHQ/snarkVM.git>

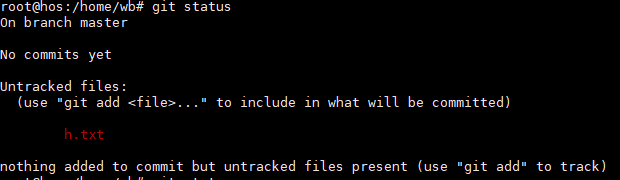
# git clone -b <branchname> <远程git仓库地址> 克隆指定分支的代码

### 3.4 操作

#### 本地仓库操作

查看仓库当前的状态，显示有变更的文件

# git status 查看文件状态



# git status -s 以简洁的信息输出文件状态

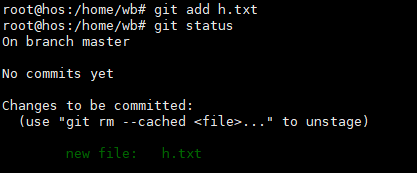


添加文件到缓存区

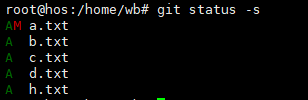
# git add <filename1> <filename2>… 将一个或多个未追踪的文件加入缓存区

# git add . 将当前目录下所有文件添加到缓存区

# git add <dirname> 添加指定目录到暂存区，包括子目录 (不能提交空目录)



如果此时修改已经添加到缓存区的文件的内容，使用git status -s 查看会发现改动文件的标识为 AM，此时需要再次执行git add <filename>将其添加到缓存区中，保持缓存区中文件的最新版本，从而保存历史版本



回退版本,可以指定退回某一次提交的版本

# git reset [--soft | --mixed | --hard] [HEAD]

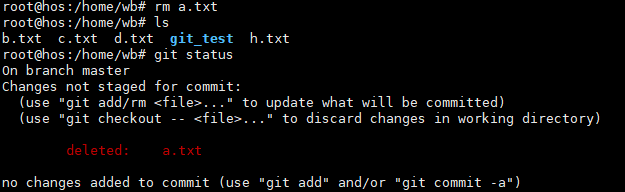
将缓存区的文件修改提交到本地仓库

# git commit <filename1> <filenam2>… -m <message> 将缓存区内容添加到本地仓库中

# git commit -am <message> 节省git add 这一步，但只对修改和删除文件有效，新文件还是执行git add ，否则就是untracked 状态

删除工作区文件

简单地从工作目录中手工删除文件，运行 git status 时就会在 Changes not staged for commit 的提示



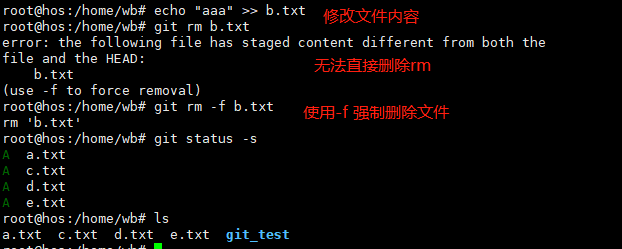
可以使用git rm 命令删除

# git rm <filename> 从缓存区和工作区中删除文件

# git rm --cached <filename> 只是将文件从缓存区移除，工作区仍保留

# git rm -f <filename> 删除修改过并且已经放到缓存区的文件，则必须使用强制删除标识-f

如果需要更新远程仓库的文件状态也需要commit\push以更新



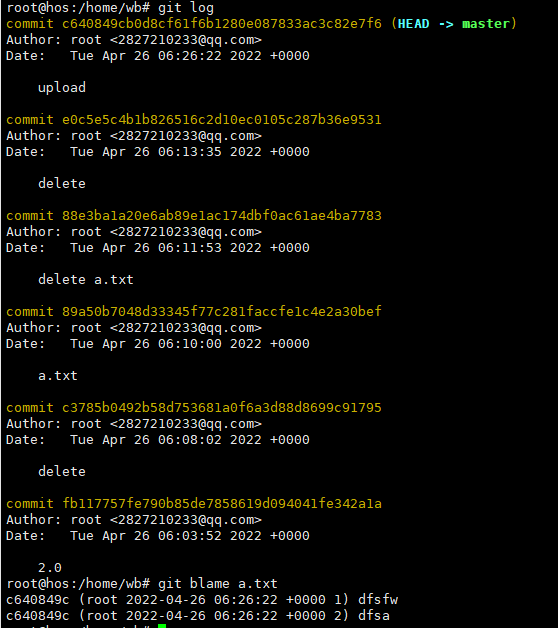
查看历史提交记录

# git log

# git log –oneline 查看历史记录的简洁版本

以列表形式查看指定文件的历史修改记录

# git blame <filename> 查看指定文件的修改记录



比较文件之间的不同

# git diff：显示工作目录(working tree)与索引区(即暂存区快照，index，就是git add过的)之间的文件变更，即显示未被add的文件变更。

# git diff --cached 或 git diff --staged：显示索引区和最后一次commit(HEAD)之间的文件更改，即显示已add但还未commit的文件变更。也即用"git commit"（不带-a）将被提交的文件变更。

# git diff HEAD：显示工作目录与最后一次commit之间的文件变更，即显示所有未commit（包括未add和add两类）的文件变更。也即用"git commit -a”将被提交的文件变更。

# git diff <分支名1> <分支名2> ：比较两个分支上最后 commit 的内容的差别。比如，先git fetch下，然后比较远程的master与当前分支已经commit的内容差别：git diff origin/master feature/20170831\_1214776\_xiufu\_1。此时是以origin/master分支为基础进行比较的，若把xiufu分支写在前面，则是以xiufu分支为基础进行比较的。 也可写作：git diff <分支名1>..<分支名2>

# git diff arsv1\_testnet3\_4267ff1c master --stat 可以通过--stat参数查看哪些文件发生了变化

将文件添加至忽略列表

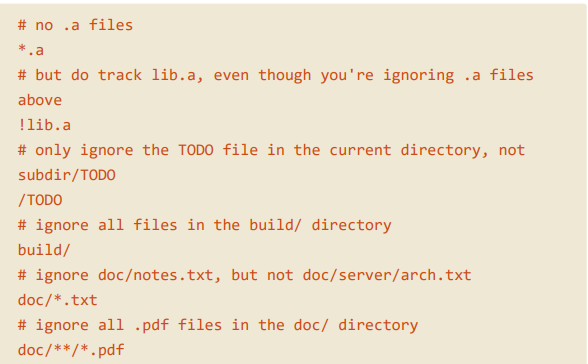
一般我们总会有些文件无需纳入Git 的管理，也不希望它们总出现在未跟踪

文件列表。 通常都是些自动生成的文件，比如日志文件，或者编译过程中

创建的临时文件等。 在这种情况下，我们可以在工作目录中创建一个名为

.gitignore 的文件（文件名称固定），列出要忽略的文件模式。下面是一

个示例：



注 .gitignore文件的基本规则

1) 任何以哈希（#）开头的行都是注释。

2) \ 字符可以转义特殊字符。

3) / 字符表示该规则只适用于位于同一文件夹中的文件和文件夹。

4) 星号（\*）表示任意数量的字符（零个或更多）。

5) 两个星号（\*\*）表示任意数量的子目录。

6) 一个问号（?）代替零个或一个字符。

7) 一个感叹号（!）会反转特定的规则（即包括了任何被前一个模式排除的文件）。

8) 空行会被忽略，所以你可以用它们来增加空间，使你的文件更容易阅读。

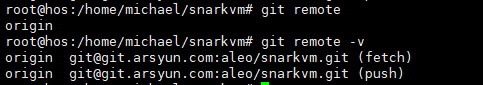
9) 在末尾添加 / 会忽略整个目录路径。

#### 远程仓库操作

查看远程仓库

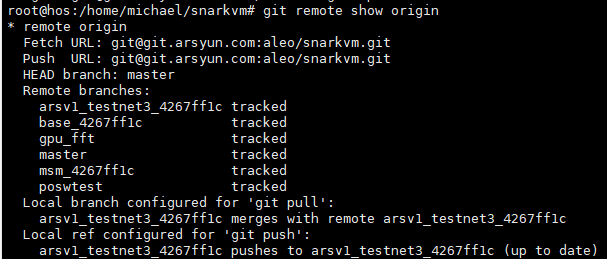
# git remote

# git remote -v



注：以上origin是远程地址的别名

# git remote show < alias> 查看指定远程仓库的具体信息



添加远程仓库

# git remote add <alias> <url>

# git remote add origin git@github.com:tianqixin/runoob-git-test.git

注：添加一个新的远程git仓库，同时指定一个可以引用的简写，本例中即是origin

删除远程仓库

# git remote rm <alias> 只是从本地移除远程仓库的记录，并不会真正影响到远程仓库

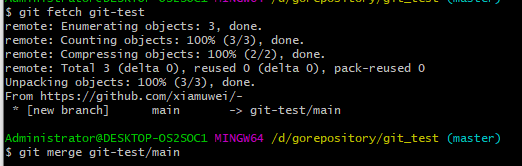
修改仓库名

# git remote rename old\_name new\_name

从远程仓库抓取或者获取最新版本到本地仓库

# git fetch <alias> 从远程仓库获取最新版本到本地仓库，不会自动merge。该命令执行完后需要执行 git merge

# git fetch <alias>/<branch> 将服务器上的任何更新合并到当前分支



# git pull <alias> <本地分支名>:<本地分支名> 是从远程仓库获取最新版本并merge到本地仓库

推送本地代码到远程仓库

# git push <alias> <本地分支名> 将本地分支推送到远程服务器

patch\am相关命令

1. 创建patch

.patch文件带有记录文件改变的内容，也带有commit记录信息,每个commit对应一个patch文件

# git format-patch <commit sha1 id> 从哪个版本开始，到当前最新版本的增量更新文件

# git format-patch 2a2fb4539925bfa4a141fe492d9828d030f7c8a8

# git format-patch <commit sha1 id> -n 某次提交（含）之前的n次提交

# git format-patch 2a2fb4539925bfa4a141fe492d9828d030f7c8a8 -2

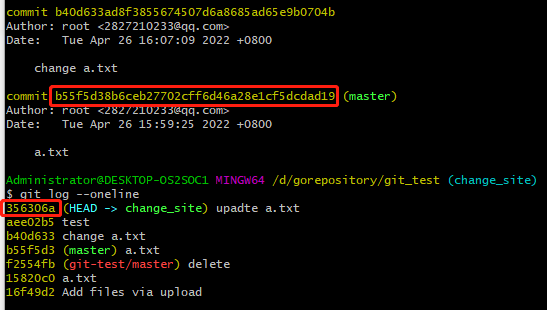
# git format-patch <commit sha1 id> -1 某次提交的patch

# git format-patch 2a2fb4539925bfa4a141fe492d9828d030f7c8a8 -1

# git format-patch <commit sha1 id>..<commit sha1 id> 某两次提交之间的所有patch

# git format-patch 2a2fb4539925bfa4a141fe492d9828d030f7c8a8..89aebfcc73bdac8054be1a242598610d8ed5f3c8

commit sha1 id可以通过git log 或者git log –oneline查看



1. 应用patch

# git apply --stat <path/to/xxx.patch> 检查patch文件

# git apply --check <path/to/xxx.patch> 检查patch是否能正常打入

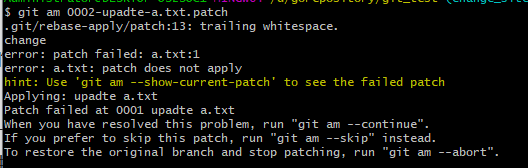
# git am --abort 为了保证执行一次全新干净的am操作，最好先执行该条命令，放弃之前残留的一些am信息

# git apply <path/to/xxx.patch> 打入patch

# git am <path/to/xxx.patch> 或者使用该条命令打入patch

3. 冲突解决

打补丁过程中有时候会出现冲突的情况，然后打补丁失败



解决：

* 找出冲突部分

**#** git apply --reject xxxx.patch 自动合入 patch 中不冲突的代码改动，同时保留冲突的部分为.rej文件 (例如发生冲突的文件是a.txt，那么运行完这个命令后，发生conflict的部分会保存为a.txt.rej)

* 根据.rej文件，通过编辑该patch文件的方式解决冲突。并执行git add.添加改动到暂存区.
* 接着执行git am --resolved或者git am --continue

## Git分支

使用分支意味着你可以把你的工作从开发主线上分离开来，以免影响开发主线。Git 的**master分支并不是一**个特殊分支。它跟其它分支没有区别。之所以几乎每一个仓库都有 master 分支，是因为git init 命令默认创建它

### 4.1 查看分支

查看所有本地分支

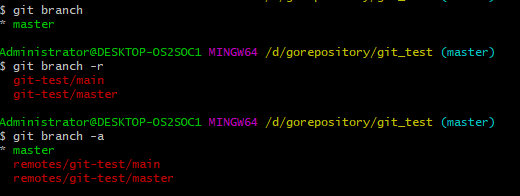
# git branch

查看所有远程分支

# git branch -r

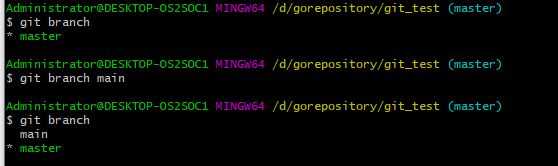
查看本地和远程的所有分支

# git branch -a



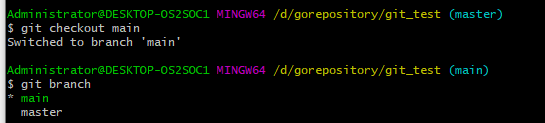
### 4.2 创建分支

# git branch <branchname>



### 4.3 切换分支

# git checkout <branchname>



以上的两部操作可以使用以下一条命令达到

# git checkout -b <branchname> 创建新分支并立即切换到该分支下，从而在该分支中操作

### 4.4 删除分支

# git branch -d <branchname>

### 合并分支

* Merge

非线性合并，有分叉。一旦某分支有了独立内容，你终究会希望将它合并回到你的主分支。 你可以使用以下命令将任何分支合并到当前分支中去：

# git merge <branchname> 将任何分支合并到当前所在的分支下

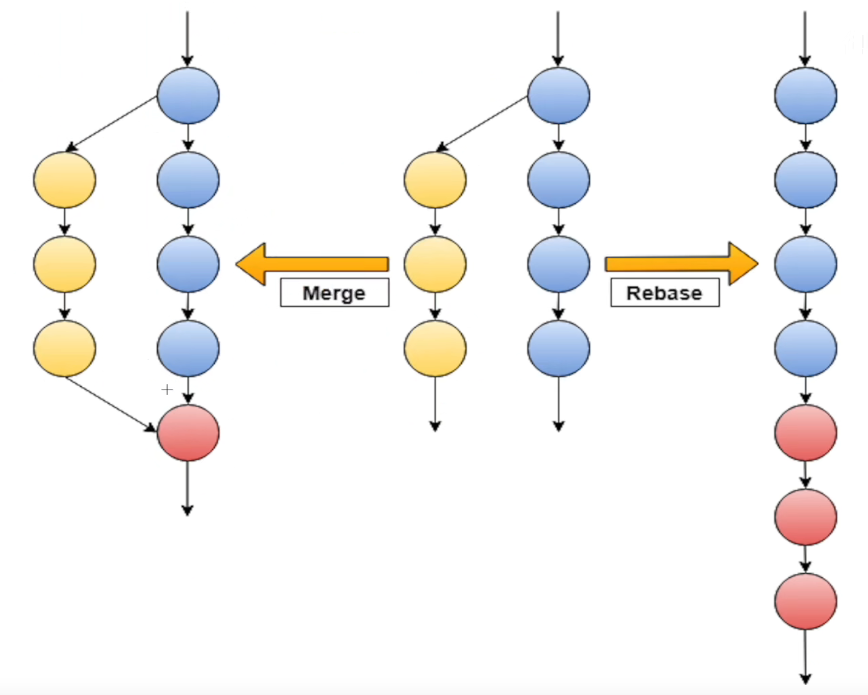
* Rebase

线性合并，无分叉。git rebase: 变基/衍合，意即改变分支的根基。rebase的功能类似于merge，从某种程度上来说，rebase与merge可以完成类似的工作，不过二者的工作方式有着显著的差异

rebase合并往往又被称为变基，这里的基就是一个基点、起点的意思。变基就是改变当前分支的起点。注意是当前所在分支的起点。Release命令后面紧接着的就是基分支。就是将当前基分支与当前分支的差异提交获取到，然后在「基分支」最新提交点后面将差异提交逐个再次提交，最后将当前分支的 HEAD 指针指向最新的提交点

# git rebase <branchname>

二者区别图示：



### 合并冲突

使用git merge\ git rebase 命令合并分支时，如果没有冲入的话，merge完成。

如果有冲突的话,git 会提示那个文件有冲突,并给出类似如下信息

<<<<<<< HEAD:test.c

printf (“test1″);

=======

printf (“test2″);

>>>>>>> issueFix:test.c

======= 隔开的上半部分，是 HEAD（即 master 分支，在运行 merge\rebase 命令时检出的分支）中的内容，下半部分是在 issueFix 分支中的内容。解决冲突的办法无非是二者选其一或者亲自整合到一起。在解决了所有文件里的所有冲突后，运行 git add 将把它们标记为已解决（resolved）。因为一旦暂存，就表示冲突已经解决

命令行查看提交图

# git log --graph --pretty=oneline --abbrev-commit



## vscode 插件

[git graph](https://blog.csdn.net/weixin_55783373/article/details/119571301)

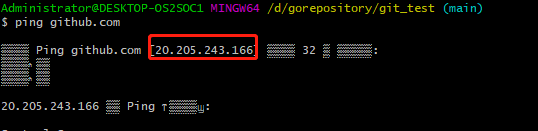
## 问题

6.1 使用git过程中出现：Git fatal: unable to access '': Failed to connect to github.com port 443: Operation timed out？



解决方式：

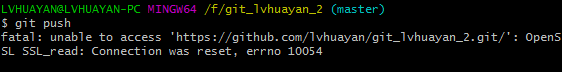
ping github，找到github的ip



更改known\_hosts文件中对应的github.com的ip

保存后再继续执行操作

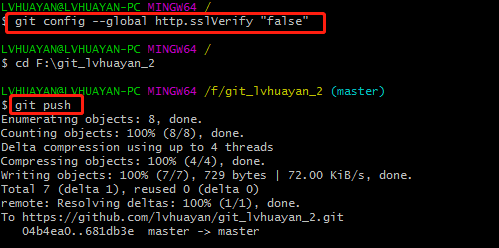
6.2 报错fatal: unable to access ‘https://github.com/.......‘: OpenSSL SSL\_read: Connection was reset？



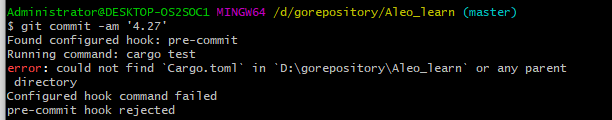
产生原因：一般是这是因为服务器的SSL证书没有经过第三方机构的签署，所以才报错

参考网上解决办法：解除ssl验证后，再次git即可

# git config --global http.sslVerify "false"



6.3 git提交出现 running pre-commit hook错误？



原因：提交之前进行了eslint的检查。发现错误过多，导致提交失败。

解决：在项目的.git\hooks文件夹中，找到pre-commit文件，然后删除。重新提交即可。