**分布式版本控制系统Git**

参考：

<http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000>

<http://www.jianshu.com/u/c47a51d2278e>

**集中式vs分布式**

集中式版本控制系统：**版本库（各个版本的代码）**是集中存放在中央服务器。

分布式版本控制系统：每台机器上都存在一个完整的**版本库**，且没有中央服务器。

**使用**：

使用集中式版本控制系统工作时首先每台机器需要从中央服务器获取最新版本，然后干活，干完后，再将自己的修改推送到中央服务器，必须得联网。

使用分布式版本控制系统并不需要联网，因为每台机器上都有一个完整的版本库，当多人对项目有修改时只要将修改文件推送给对方就可以了。在实际使用分布式版本控制系统时很少人是会互相推送修改的，一般都是建立一个远程中央仓库，当多人都对项目修改后，将自己的修改推送到这个远程仓库交换修改。

**安全性**：分布式比集中式安全，当集中式的服务器数据丢失，版本库是找不回来的，而分布式的中央仓库丢失的话，可以任意推送一份本地的版本库上去就可以了。

**Git基础**

1、直接记录快照，而非差异性比较

Git只关心文件数据的整体是否发生变化，而不是比较文件内容的具体差异。Git每次commit时都会**完整的存储当前版本所有修改的文件（**即暂存区的文件**，**不只是存储diff即具体修改内容**），**并保存一个这个修改的索引。

2、近乎所有的操作都是本地执行

Git的绝大多数操作都只需要访问本地就行，因为本地有完整的版本库，远程仓库只是交换修改的地方。例如想要看历史版本的更新内容，你不用去远程服务器上去获取历史版本，Git会直接从本地取出历史版本展示给你看。

1. 本地仓库结构

本地仓库说白了就是你计算机上的一个目录，当你安装好Git环境后，在这个目录下（文件夹下）执行git init命令，这个目录就变成了你的一个本地的Git仓库。本地仓库主要分为两部分：**工作区（Working Directory）、版本库（Repository）。**

**工作区（Working Directory）**：本地仓库下（目录下）就是我们的工作区，也就是我们直接修改文件的地方。

**版本库（Repository）：**本地仓库下（目录下）有一个隐藏目录 .git，这个隐藏目录就是版本库，这个目录下存储了我们修改过的全部版本。

版本库里也包含了很多的东西，其中比较重要的就是暂存区（stage），还有创建仓库时Git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD。

本地仓库主要结构如下:



1. 本地仓库中的文件状态（不是重点，因为有好多版本的叫法）

文件状态是随着对文件的操作而产生的。仓库中的文件从它产生的那一刻开始就有了状态：

Untracked(未跟踪)：指工作区新增的文件。

Tracked(已跟踪)：只要工作区的新增文件添加到暂存区，那么这个文件以后就都是已跟踪状态（同时也是已暂存状态）。

Staged(已暂存)：只有暂存区的文件才是已暂存状态。

Committed（已提交状态）：暂存区中的文件commit后就变成了已提交状态，此时也被添加到了master分支。

Modified（已修改状态）：指当commit后的文件被修改了后的状态。注意当文件在暂存区被修改，那么它应该也是已修改状态，个人理解只要工作区的文件被修改了那么他就是已修改状态。

5、版本管理原理

刚刚创建好仓库后，仓库是空的什么也没有，我们需要在工作区里面创建一个项目，这时候工作区就有了内容，接下来我们需要把项目文件添加到暂存区（git add），然后从暂存区提交（git commit）到master分支，提交成功后暂存区内容被清空，这样版本库就有了第一个版本。当我们修改了工作区的文件内容时，我们依然按照上面的步骤先把修改的文件添加到暂存区，再由暂存区提交到master分支，这样版本库就有了第二个版本。

* **安装&创建仓库**

**安装**：

Windows下要使用很多Linux/Unix的工具时，需要Cygwin这样的模拟环境，Git也一样。Cygwin的安装和配置都比较复杂，但是不用太折腾，直接使用已经把模拟环境和Git都打包好的程序，名叫msysgit，只需要下载一个单独的exe安装程序，其他什么也不用装。

msysgit是Windows版的Git，从[https://git-for-windows.github.io](https://git-for-windows.github.io/" \t "http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)下载，然后一路按默认选项安装即可。安装完成后，在开始菜单里找到“Git”->“Git Bash”，蹦出一个类似命令行窗口的东西，就说明Git安装成功！

**配置**：

安装完成后，还需要最后一步进行git环境设置，在刚才弹出的命令窗口输入：

$ git config --global user.name "Your Name"

$ git config --global user.email ["email@example.com"](mailto:\"email@example.com\")

因为Git是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和Email地址，名字任意取，邮箱选择一个任意属于你自己的邮箱。

注意：git config命令的--global参数，用了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以去掉这个参数对你机器上其它某个仓库指定不同的用户名和Email地址。

**创建仓库：**

我们可以随便找一个磁盘目录右击在列表中选择Git Base Here，Git命令窗口会自动的定位到这个目录下，然后执行命令：git init这样这个目录就变成了一个本地仓库，仓库里面包含一个隐藏的目录.git这个目录下就是我们的版本库，千万不要手动改这个目录下的文件。

注意：为了避免遇到各种莫名其妙的问题，请确保目录名（包括父目录）不包含中文。

* **远程仓库**

Git是分布式版本控制系统，同一个Git仓库可以分布到不同的机器上，只要别的机器从这个机器上克隆这个原始版本，这样就完成了分布式。每台机器上的版本库其实是一样的没有主次之分。**多人协作开发实际情况往往是这样的我们需要一个远程服务器来管理一个仓库，其它的机器每次修改之后都来服务器仓库交换修改就可以了，其实就是把各自的修改push到服务器仓库，然后pull别人的修改**。我们这里使用GitHub来充当远程服务器的的角色，这个网站就是提供Git仓库托管服务的。

由于你的本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，支持SSH协议。我们需要本地生成一下SSH Key，因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。当然，GitHub允许你添加多个Key。假定你有若干电脑，你一会儿在公司提交，一会儿在家里提交，只要把每台电脑的Key都添加到GitHub，就可以在每台电脑上往GitHub推送了

**生成和配置SSH Key：**

1. 创建SSH Key。在用户主目录C:\Users\你的电脑用户名，看看有没有.ssh目录，如果有，再看看这个目录下有没有id\_rsa和id\_rsa.pub这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：

$ ssh-keygen -t rsa -C ["youremail@example.com"](mailto:\"youremail@example.com\")

这个邮件地址就是你配置git环境的邮箱，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有id\_rsa和id\_rsa.pub两个文件，这两个就是SSH Key的秘钥对，id\_rsa是私钥，不能泄露出去，id\_rsa.pub是公钥，可以放心地告诉任何人。

登陆GitHub，打开“Account settings”-->“SSH Keys”-->点“Add SSH Key”，填上任意Title，在Key文本框里粘贴id\_rsa.pub文件的内容。

**关联远程库：**

关联远程仓库实际是先将本地仓库与远程库关联，然后将本地仓库内容推送到远程库。我们首先在GitHub上创建一个仓库作为我们的远程仓库。然后执行以下命令将本地仓库与远程仓库关联起来：

$ git remote add origin [git@github.com:gitbuhName/gitbuhRepository.git](mailto:git@github.com:michaelliao/learngit.git)

其中origin是本地仓库给远程仓库起的名字，可以随意叫，githubName改成你的账户名，gitHubRepository是你在gitHub网站上创建的仓库名字。

下一步，就可以把本地仓库的内容推送到远程仓库了：

git pull --rebase origin master //一个好的习惯，先将远程仓库与本地仓库同步下。

git push -u origin master//注意,推送前确认下本地仓库工作区的内容是否已经commit。

由于远程库是空的，我们第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令，将-u去掉就行了。

注意：

当你第一次使用Git的clone或者push命令连接GitHub时，会得到一个警告：

The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.

RSA key fingerprint is xx.xx.xx.xx.xx.

Are you sure you want to continue connecting (yes/no)?

这是因为Git使用SSH连接，而SSH连接在第一次验证GitHub服务器的Key时，需要你确认GitHub

的Key的指纹信息是否真的来自GitHub的服务器，输入yes回车即可。

Git会输出一个警告，告诉你已经把GitHub的Key添加到本机的一个信任列表里了：

Warning: Permanently added 'github.com' (RSA) to the list of known hosts.

**从远程克隆：**

从远程仓库克隆很简单直，在你本地找一个目录不必是已经建好的仓库目录下，接执行以下命令：

$ git clone [git@github.com:gitbuhName/gitbuhRepository.git](mailto:git@github.com:michaelliao/learngit.git)

或

$ git clone <https://github.com/gitbuhName/gitbuhRepository.git>

Git支持多种协议，默认的git://使用ssh必须设置SSH Key。SSH，Secure shell(安全外壳协议)，专为远程登陆会话与其他网络服务提供安全性的协议， 而SSH传输的数据是可以经过压缩的，可以加快传输的速度，出于安全性与速度，我们优先考虑使用SSH协议。使用SSH url克隆的话，你必须是项目的拥有者或管理员，而且还要添加SSH Key，否则会无法克隆。

也可以使用https协议，https每次push都必须输入口令，但是不需要设置SSH Key，使用https可以任意克隆Github上的项目。

**push到别人远程库方式：**

1、登录你自己的github网站，在自己仓库页面下Settings-->Collaborators，下填写其他的github的账户名或邮箱，即可添加一个作者。

2、在别人仓库下点击Fork按钮，把这个项目fork到自己的账号下，然后Clone到本地，然后做你想做的修改，commit提交，然后push到自己账号里的仓库，在自己仓库里点击Pull requests，然后新建一个pull request，按步骤填写，最后提交，这个时候开源项目的作者就会收到一个pullrequest的请求。

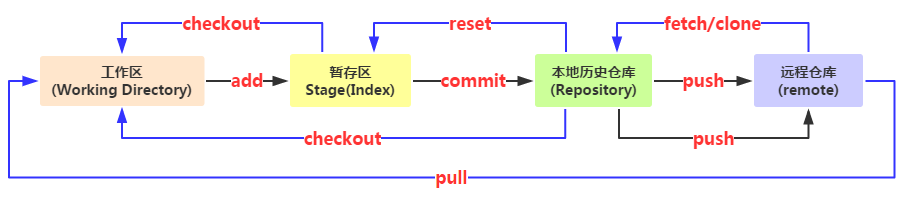
* **分支**

我们每次commit代码时，在版本库就会产生一个版本，而这些版本是在不同的时间产生的，随着时间的累积版本库中的版本也在不断地产生，这样**在这条时间线上的版本**就是一个分支。

一开始版本库默认只有一条时间线就是默认的master分支，当然我们还可以在主分支的**内容基础上**创建其它的时间线，即创建其他分支。每个分支上做的修改对其他分支没有任何影响，除非进行合并。

此外我们还需要知道版本库里有一个HEAD指针，这个指针指向我们本地仓库当前的分支。

* **仓库操作**



1. **添加到暂存区操作**

注意：无论将命令窗口定位在仓库根目录或者仓库下更深的目录执行以下命令，那么命令将作用于整个工作区。假如将命令定位在仓库更深的目录，例如仓库中app项目根目录下，执行以下命令时在命令末尾添加上 **.**则表示命令作用于app项目，对其他目录的文件无效，例如git add -u .或 git add -A .或git add .等。

git add a.txt :将新建的文件添加到暂存区。

git add a.txt b.txt c.txt :将多个新建的文件添加到暂存区。

git add -u :将工作区内tracked(已跟踪)文件中被修改或者被删除的文件添加到暂存区，不 会添加untracked(未跟踪文件)文件。

git add -A :将工作区内untracked(未跟踪文件)文件和tracked(已跟踪)文件中被修改或者被删 除的文件添加到暂存区。

git add . :将**当前目录**所有文件都添加到暂存区，注意不包括删除。

1. **分支操作**

1）查看分支以及历史版本

git branch :查看本地当前分支列表。

git branch -r :查看远程仓库分支列表。

git branch -a :显示本地分支列表和远程仓库分支列表。

git log -g <yourbranch> ：查看分支历史版本。

注意：想要查看分支详细信息：

1. 打开：.git/logs/refs/heads/<yourbranch>目录
2. 执行命令 git log -g <yourbranch>

2）添加分支

git branch <branchname> ：在当前分支内容基础上创建一个新的分支。

git checkout <branchname> :切换到相应分支，此时HEAD指针指向当前切换出来的分支。

git checkout -b <branchname> :创建并切换到这个分支。相当于执行上面的两个命令。

3）删除分支

git branch -D <branchname> :删除分支。

4）合并分支

git merge <branchname> :将某分支内容合并到当前分支。快速合并模式。

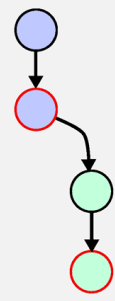
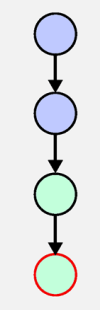
git merge --no-ff -m "xxx" <branchname> :将某分支内容合并到当前分支。**推荐使用。**

上面两种合并差别很大，举个例子，将dev分支的修改，合并到master分支上：

**git merge dev**

快速合并模式合并后，就是把dev分支的commit历史版本信息直接添加到了master分支时间线上，在master分支时间线上看不到dev分支。

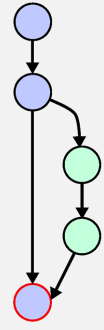
合并前后：

**git merge --no-ff -m "合并dev分支" dev**

首先解释下 --no-ff 表示禁用快速合并模式。-m 表示本次合并其实是一个commit,可以理解为把dev分支上修改的所有东西全部commit到master分支上，所以需要写一个提交备注。合并后master分支时间线上能看到dev分支，而且还可以看到dev分支上的commit历史版本信息。

合并前后：



以上两种合并删除dev分支后master分支线上仍然可以看到dev分支commit历史版本信息。

5）存储分支

git stash :切换分支时，将当前未commit的工作存储起来。

git stash list :查看当前分支存贮列表。

git stash apply :恢复存储列表中最后一次存储,恢复后stash内容并不会被删除。假如多次stash，必须在commit当前已经恢复的内容之后，才能继续恢复stash list当中的stash。假如使用这个命令在恢复之后，继续git stash，那么stash list中又会增加一条一模一样的stash内容。

git stash pop :恢复存储列表中最后一次存储,并删除stash内容。

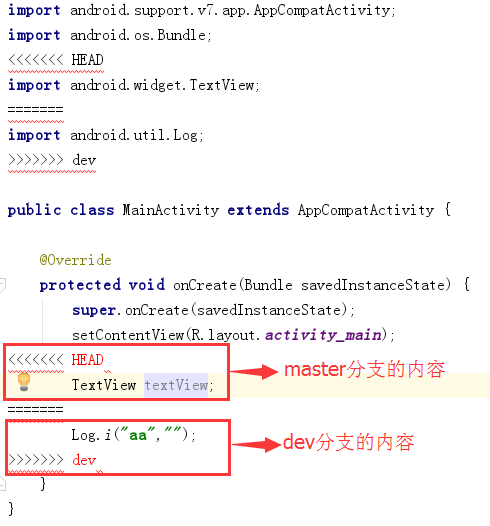
git stash drop :不管stash list中的存储有没有被恢复，从最后一次stash开始依次删 除stash list中的存储。

git stash apply stash@{1} :效果同git stash apply，恢复指定的stash内容。

git stash drop stash@{1} :效果同git stash drop ，删除指定的stash内容。

6）分支冲突

举个例子dev分支上在MainActivity中做修改，然后add并commit。之后切换到master分支上在同一文件也做修改，然后add并commit。接下来我们进行合并git merge dev，由于两个分支对同一文件做了修改，合并的时候一定会起冲突:



Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，并且在这些符号后面写清了属于哪个分支，比如第一个红框内<<<<<<<<< HEAD表示的是当前分支master，因为HEAD是代表版本库中的指针，这个指针都是指向当前分支的，而当前分支是master，所以第一个框里的内容是master分支提交的。第二个框里更明显了使用>>>>>>> dev说明是dev分支提交的内容，两个分支提交的内容使用=====分割。

我们需要修改这些冲突，修改完后在当前分支add然后commit（不要要再次merge）,这样分支的内容就合并成功了。

1. 撤销修改

git checkout -- <filename> :撤销工作区中文件的修改，被add到暂存区的修改不会被撤回，例如第一次修改add到了暂存区，第二次修改后没有add到暂存区，使用这个命令后只能撤回工作区第二次修改的内容，然后回到第一次修改的状态。

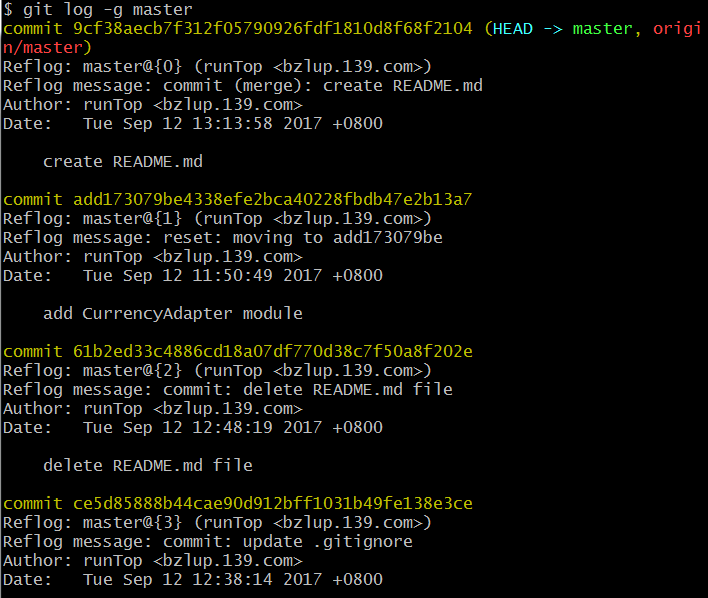
git reset HEAD <filename> :撤销add到暂存区的文件，但是工作区的修改不会被撤销。

1. 版本回退

当你本地仓库已经都commit了之后，想回退到以前的某个版本，用HEAD表示当前版本，也就是你最新提交的版本，上一个版本就是HEAD^,上上个版本就是HEAD^^，使用^不好表示的话可以写成HEAD~3：

git reset -hard HEAD^ :回退上上一个版本。

另外一种方法，需要知道提交的版本名称，知道这个名称之后，可以任意的跳到任何一个版本，不管是当前版本之前的还是当前版本之后的版本，例如：



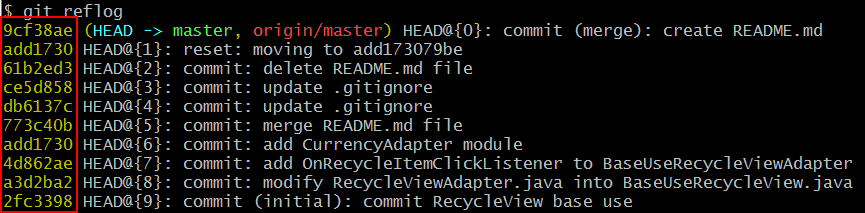
上图中是获取的当前master分支上的commit版本日志，黄色commit后面的字符串就是提交的版本号，通过：

**git reset -hard <版本号>**：回退到特定的版本。

例如使用git reset -hard 61b2ed33 就可回退到delete README.md file这个版本，其中版本号没有必要写全，主要包含足够使用的前几位就好。当然我们回退到61b2ed33之后，我们还可以回退到上图中的第一个版本9cf38ae这个版本，前提是我们必须知道9cf38ae这个版本号。Git的版本回退速度非常快，因为Git在内部有个指向当前版本的HEAD指针，当你回退版本的时候，Git仅仅是把HEAD从指向那个版本，然后顺便把工作区的文件更新了。所以你让HEAD指向哪个版本号，你就把当前版本定位在哪。

如上图，如果我们会退了到了61b2ed33 这个版本后关闭了命令框后悔了，不想回退了，怎么办？办法还是有的：

**git reflog**：打印你的每一次执行commit命令，其中包括版本号以及提交注释。



通过git reflog命令，可以找到你commit的所有的版本号（上图红框内），在通过git reset命令就又可以跳到任意的版本了。

* **实际操作遇到的问题**

1. [更新.gitignore 配置](http://blog.csdn.net/u010940300/article/details/49204597)文件

[git](http://lib.csdn.net/base/git" \o "Git知识库" \t "http://blog.csdn.net/u010940300/article/details/_blank)有一条规则，“**如果一个文件一开始没有添加到.gitignore里面并且被跟踪，中途添加进去。git 将不会忽略这个文件**。”在这种情况下，必须使用 git rm --cached < file name > 这个命令来移除对这个文件的跟踪，然后将这个不想被跟踪的文件添加到 .gitignore文件里面就可以了。

1. 本地仓库和远程仓库同时修改后出现的问题。

两个仓库同时修改会不限版本不匹配问题，当你本地pull的时候，会报错，这时候你需要将本地仓库修改的东西都commit之后，将远程的仓库内容合并（merge）到本地：**git merge --no-ff -m "注释" <远程仓库名>，**然后在提交。