$ git init  // 初始化一个Git仓库  
$ git status   // 查看仓库的状态  
$ git add .   // 将所有修改添加到暂存区  
$ git add \*  // Ant风格添加修改  
$ git add \*Controller   // 将以Controller结尾的文件的所有修改添加到暂存区  
$ git add Hello\*   // 将所有以Hello开头的文件的修改添加到暂存区 例如:HelloWorld.txt,Hello.java,HelloGit.txt ...  
$ git add Hello?   // 将以Hello开头后面只有一位的文件的修改提交到暂存区 例如:Hello1.txt,HelloA.java 如果是HelloGit.txt或者Hello.java是不会被添加的  
可以多次添加然后在提交  
$ git commit -m "comment"  // 将暂存区的修改提交到仓库 后面添加上有意义的注视信息  
$ git diff  file  // 在file被修改了还未提交的时候查看修改的部分(和版本库中最新版本的不同 diff == difference 不同)  
$ git log  // 查看git的commit信息,每次提交的信息包括注视在内,从最新提交到最久提交  
$ git log --pretty=oneline   // 将commit 信息简化成一行显示  
注意Git的版本号(commit id)是SHA1值ea34578d5496d7dd233c827ed32a8cd576c5ee85  
$ git reset --hard HEAD^   // 退回到相对于当前版本的上一个版本 HEAD 表示当前版本  
$ git reset --hard HEAD^^  // 退回到相对于当前版本的上上一个版本  HEAD 代表当前版本  
$ git reset --hard HEAD~100 //退回到相对于当前版本的上 100 个版本去  HEAD 表示当前版本  
$ git reset --hard 3628164  // 退回到指定的版本   这里不需要全部写commit id  Git 回去自动适配  
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append GPL  
$ git reflog   // 在退回到旧版本之后可以查看旧版本之前的提交日志  
当我们想从一个旧版本退回到新版本但是我们关闭了shell窗口,不能查看之前的commit id了,就可以通过  
$ git reflog 查看到之前的版本的commit id  
$ git reset --hard 3628164  
  
  
工作区和暂存区  
工作区: 就是我们通$ git init 创建的代码库的所有文件但是不包括 .git文件(版本库)  
暂存区: 我们通过$ git add ./\*/\*Xxx/Xxxx\* 添加的修改,都是进入到暂存区了,肉眼不可见 通过 $ git status  可以看到修改的状态  
什么是修改？  
  比如你新增了一行，这就是一个修改，  
  删除了一行，也是一个修改，  
  更改了某些字符，也是一个修改，  
  删了一些又加了一些，也是一个修改，  
  甚至创建一个新文件，也算一个修改。  
  
  
修改只能在被add 到暂存区以后才能被提交  
在file 已经修改还未add的时候  
$ git checkout --file  // 表示丢弃工作区的修改退回原始状态(不包括以及添加到暂存区的修改)  
file已经修改和添加到了暂存区,还未commit  
$ git reset HEAD file   // 丢弃file已经添加到暂存区的修改  HEAD 表示最新版本  
  
  
如果file修改已经提交到本地仓库  
$ git reset --hard HEAD^   //退回到上一个版本  
  
  
$ rm file  // 从文件系统中删除文件(Git 中还是有记录),$ git status 的时候Git 会告诉你有一个file 被删除了  
$ git rm file  // 从Git版本库中删除文件(同时从文件系统中删除文件) $ git status  正常  
  
  
$ git rm file  // 从git版本库中删除文件  
$ git commit -m "delete file" // 提交删除  
$ git checkout file  // 想要从git恢复 会出现:error: pathspec 'README.txt' did not match any file(s) known to git.  
$ git reset --hard HEAD^  // 恢复到上一个版本 可以恢复文件  
  
  
$ rm file   //在文件系统中删除文件  
$ git checkout file  // 由于是通过文件系统删除的所以可以通过checkout恢复  
  
  
$ ssh-keygen -t rsa -C "youremail@example.com"  // 创建SSH Key  
由于这个Key也不是用于军事目的，所以也无需设置密码,所以一路回车就好,  
如果顺利会在user下UserName目录中生成一个.ssh目录里面有id\_rsa和id\_rsa.pub两个文件  
id\_rsa是私钥，不能泄露出去，id\_rsa.pub是公钥,可以告诉他人还有在github上添加的也是这个公钥  
登陆GitHub，打开“Account settings”，“SSH Keys”页面,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  
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  
  
  
添加远程库  
做全局的配置  
$ git config --global user.name "zhangsan"  
$ git config --global user.email "zhangsan@139.com"  
$ git remote add origin git@github.com:zhangsan/ylez.git // 添加一个远程仓库  
  // add origin 就是添加一个远程仓库  
  // git@github.com:zhangsan/ylez.git  是远程仓库的地址  
  // git@github.com 主机的地址  我们可以通过GitLab大家自己的git服务器  
  // zhangsan 是你的用户名  
  // /ylez.git 是你的仓库名  
$ git push -u origin master  // 将本地的master分支推送到远程的master分支中  
$ git push -u origin dev    // 本地切换到dev分支然后将本地的dev分支推送到远程  
  
  
克隆远程仓库到本地  
Git支持多种协议，包括https，但通过ssh支持的原生git协议速度最快。  
$ git clone git@192.168.0.8:zhangsan/test.git  // 讲一个远程的仓库克隆到本地  
  
  
  
  
每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支  
在Git里，默认有一个主分支，即master分支  
HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支  
master分支是一条线，Git用master指向最新的提交，再用HEAD指向master  
每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长  
Git新建了一个指针叫dev，指向master相同的提交，再把HEAD指向dev，就表示当前分支在dev上  
  
  
$ git branch   // 查看git的所有分支  
$ git branch dev  // 创建dev分支  
$ git checkout dev  // 切换到一个已经存在的分支  
$ git checkout -b dev  // 创建dev分支,并切换到dev分支  
$ git branch -d dev  // 删除dev分支  
  
  
$ git checkout master  // 切换到主分支  
$ git merge dev   // 将dev分支合并到master分支  
Updating d17efd8..fec145a  
Fast-forward  
 readme.txt |    1 +  
 1 file changed, 1 insertion(+)  
Fast-forward信息，Git告诉我们，这次合并是“快进模式”,就是直接把master指向dev的当前提交，所以合并速度非常快  
$ git branch -d dev  // 删除dev分支  
  
  
现在只是剩下master分支,感觉什么事情都没又发生一样  
  
  
处理冲突  
$ git checkout -b feature1  
Switched to a new branch 'feature1'  
  
  
修改readme.txt最后一行，改为：  
Creating a new branch is quick AND simple.  
  
  
$ git add readme.txt  
$ git commit -m "AND simple"  
[feature1 75a857c] AND simple  
 1 file changed, 1 insertion(+), 1 deletion(-)  
  
  
$ git checkout master  
Switched to branch 'master'  
Your branch is ahead of 'origin/master' by 1 commit.  
Git还会自动提示我们当前master分支比远程的master分支要超前1个提交。  
  
  
在master分支上把readme.txt文件的最后一行改为：  
Creating a new branch is quick & simple.  
  
  
$ git add readme.txt  
$ git commit -m "& simple"  
[master 400b400] & simple  
1 file changed, 1 insertion(+), 1 deletion(-)  
  
  
$ git merge feature1  
Auto-merging readme.txt  
CONFLICT (content): Merge conflict in readme.txt  
Automatic merge failed; fix conflicts and then commit the result.  
  
  
$ git status  
# On branch master  
# Your branch is ahead of 'origin/master' by 2 commits.  
#  
# Unmerged paths:  
#   (use "git add/rm <file>..." as appropriate to mark resolution)  
#  
#       both modified:      readme.txt  // 冲突了  
#  
no changes added to commit (use "git add" and/or "git commit -a")  
  
  
Git is a distributed version control system.  
Git is free software distributed under the GPL.  
Git has a mutable index called stage.  
Git tracks changes of files.  
<<<<<<< HEAD  
Creating a new branch is quick & simple.  
=======  
Creating a new branch is quick AND simple.  
>>>>>>> feature1  
  
  
Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，我们修改如下后保存：  
Creating a new branch is quick and simple.  
  
  
再提交：  
$ git add readme.txt  
$ git commit -m "conflict fixed"  
[master 59bc1cb] conflict fixed  
  
  
用带参数的git log也可以看到分支的合并情况:  
$ git log --graph --pretty=oneline --abbrev-commit  
$ git log --graph --pretty=oneline --abbrev-commit  
\*   59bc1cb conflict fixed  
|\  
| \* 75a857c AND simple  
\* | 400b400 & simple  
|/  
\* fec145a branch test  
...  
  
  
  
  
删除feature1分支：  
$ git branch -d feature1  
Deleted branch feature1 (was 75a857c).  
  
  
通常，合并分支时，如果可能，Git会用Fast forward模式，但这种模式下，删除分支后，会丢掉分支信息。  
如果要强制禁用Fast forward模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息  
$ git merge --no-ff -m "comment" dev  // 准备合并dev分支，请注意--no-ff参数，表示禁用Fast forward  
  
  
$ git log --graph --pretty=oneline --abbrev-commit  //图形化展示分支情况  
\*   7825a50 merge with no-ff  
|\  
| \* 6224937 add merge  
|/  
\*   59bc1cb conflict fixed  
...  
  
  
修复一个代号101的bug的任务时，很自然地，你想创建一个分支issue-101来修复它  
在修复bug时候,手里还有工作没有完成但是又不能提交可以将进度冻结  
在你当前的工作区(分支)  
$ git stash  // 冻结当前的分支修改  
  
  
查看所有的stash  
$ git stash list  // 列出所有的工作现场存储  
$ git stash list  
stash@{0}: WIP on dev: 6224937 add merge  
$ git stash apply (stash@{n}) // 恢复工作现场(只有一个的时候可以省略后面的,如果有多个可以恢复指定的某一个stash)但是不删除存储中的stash  
$ git stash drop (stash@{n})  // 删除存储的工作现场(只有一个的时候可以省略后面的,如果又多个就指定删除某一个)  
$ git stash pop (stash@{n})  // 恢复的同时把stash内容也删了(只有一个的时候可以省略后面的,如果又多个就指定删除某一个)  
  
  
$ git branch -d branchName  // 删除某一个分支,前提是在该分支和产生该分支的主分支已经合并了(merge)  
$ git branch -D branchName  // 删除某一个分支,无论是否合并都强制删除  
  
  
  
  
当你从远程仓库克隆时，实际上Git自动把本地的master分支和远程的master分支对应起来了，并且，远程仓库的默认名称是origin。  
  
  
要查看远程库的信息，用git remote：  
$ git remote  
origin  
  
  
或者，用git remote -v显示更详细的信息：  
$ git remote -v  
origin  git@github.com:michaelliao/learngit.git (fetch)  
origin  git@github.com:michaelliao/learngit.git (push)  
  
  
并不是一定要把本地分支往远程推送，那么，哪些分支需要推送，哪些不需要呢？  
  master分支是主分支，因此要时刻与远程同步；  
  dev分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；  
  bug分支只用于在本地修复bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个bug；  
  feature分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。  
  
  
推送的时候如果有人在你之前已经推送了,推送失败  
$ git push origin dev  
  To git@github.com:michaelliao/learngit.git  
   ! [rejected]        dev -> dev (non-fast-forward)  
  error: failed to push some refs to 'git@github.com:michaelliao/learngit.git'  
  hint: Updates were rejected because the tip of your current branch is behind  
  hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')  
  hint: before pushing again.  
  hint: See the 'Note about fast-forwards' in 'git push --help' for details.  
因为你的小伙伴的最新提交和你试图推送的提交有冲突，解决办法也很简单，  
Git已经提示我们，先用git pull把最新的提交从origin/dev抓下来，然后，在本地合并，解决冲突，再推送  
$ git pull  //抓去远程的分支的提交到本地  
  
  
没有指定本地dev分支与远程origin/dev分支的链接,推送也会失败,所以首先我们需要建立本地分支和远程分支的联系  
$ git branch --set-upstream dev origin/dev  // 建立本地dev 分支和远程dev分支的联系  
Branch dev set up to track remote branch dev from origin.  
  
  
总结:  
查看远程库信息，使用git remote -v；  
本地新建的分支如果不推送到远程，对其他人就是不可见的；  
从本地推送分支，使用git push origin branch-name，如果推送失败，先用git pull抓取远程的新提交；  
在本地创建和远程分支对应的分支，使用git checkout -b branch-name origin/branch-name，本地和远程分支的名称最好一致；  
建立本地分支和远程分支的关联，使用git branch --set-upstream branch-name origin/branch-name；  
从远程抓取分支，使用git pull，如果有冲突，要先处理冲突。  
  
  
发布一个版本时，我们通常先在版本库中打一个标签，这样，就唯一确定了打标签时刻的版本。  
将来无论什么时候，取某个标签的版本，就是把那个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。  
Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像对不对？但是分支可以移动，标签不能移动），所以，创建和删除标签都是瞬间完成的。  
  
  
标签一般打在master分支上  
$ git tag v1.0  // 在当前版本HEAD上打一个名称为v1.0 的标签  
$ git tag   // 查看所有标签,会列表出所有的标签名  
  
  
为历史commit打tag(有时候忘记打标签了本来该星期一打结果星期五才想起来)  
$ git log --pretty=oneline --abbrev-commit  
6a5819e merged bug fix 101  
cc17032 fix bug 101  
7825a50 merge with no-ff  
6224937 add merge  
59bc1cb conflict fixed  
400b400 & simple  
75a857c AND simple  
fec145a branch test  
d17efd8 remove test.txt  
  
  
$ git tag v0.8 59bc1cb // 为commit id 为59bc1cb...的commit打上v0.8 标签  
标签不是按时间顺序列出，而是按字母排序的。可以用git show <tagname>查看标签信息  
  
  
还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字：  
$ git tag -a v0.1 -m "version 0.1 released" 3628164  // 为commit id为3628164...的commit打上v0.1 的标签注视内容是version 0.1 released  
  
  
$ git tag v0.1 //可以查看标签的信息包括文字说明  
还可以通过-s用私钥签名一个标签：  
$ git tag -s v0.2 -m "signed version 0.2 released" fec145a  
签名采用PGP签名，因此，必须首先安装gpg（GnuPG），如果没有找到gpg，或者没有gpg密钥对，就会报错：  
gpg: signing failed: secret key not available  
error: gpg failed to sign the data  
error: unable to sign the tag  
如果报错，请参考GnuPG帮助文档配置Key。  
  
  
用命令git show <tagname>可以看到PGP签名信息：  
$ git show v0.2  
  
  
$ git tag -d v0.1   // 删除标签 v0.1  
$ git push -u origin v1.0  // 将标签v1.0 推送到远程仓库  
$ git push -u origin --tags   // 一次性推送全部标签  
  
  
如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：  
$ git tag -d v0.9  
Deleted tag 'v0.9' (was 6224937)  
  
  
然后，从远程删除。删除命令也是push，但是格式如下：  
$ git push origin :refs/tags/v0.9  
To git@github.com:michaelliao/learngit.git  
 - [deleted]         v0.9  
  
  
在Git工作区的根目录下创建一个特殊的.gitignore文件，然后把要忽略的文件名填进去，Git就会自动忽略这些文件  
忽略文件的原则是：  
  
  
忽略操作系统自动生成的文件，比如缩略图等；  
忽略编译生成的中间文件、可执行文件等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如Java编译产生的.class文件；  
忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。  
一个完整的.gitignore文件  
# Windows:  
Thumbs.db  
ehthumbs.db  
Desktop.ini  
  
  
#project  
\*.class  
\*.war  
classes  
target  
  
  
首先从git上checkout最新的代码默认是checkout主分支master分支的代码,所以我们需要再次checkout开发分支的代码  
master分支是最新稳定版,而dev分支是最新的开发版  
一定要记得从服务器上checout而不是在本地创建新分支,如果做新的功能是在前面2步后,在本地的dev分支上checkout新的子分支

然后开发,提交合并到dev,push到代码服务器中.