Git 操作说明

——Bash命令语句

# 安装Git (Windows)

在Windows上使用Git，可以从Git官网 (<https://git-scm.com/downloads>) 直接下载安装程序，然后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！

安装完成后，还需要最后一步设置，在命令行输入：

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

$ *git config --global user.email "email@example.com"*

设置自己的名称和Email地址。

※--global参数表示本机所有Git仓库都会使用该配置。

# 创建版本库

**创建版本库**

首先，选择一个合适的地方，创建一个空目录：

$ *mkdir learngit*

$ *cd learngit*

$ *pwd*  用于显示当前目录（Windows系统，为了避免遇到各种莫名其妙的问题，必须确保目录名（包括父目录）不包含中文）。

然后，通过git init命令把这个目录变成Git可以管理的仓库。

$ *git init*  初始化Git仓库。

**把文件添加到版本库**

第一步，用命令git add告诉Git，把文件添加到仓库。

$ *git add <file>*  可反复多次使用，添加多个文件。

第二步，用命令git commit告诉Git，把文件提交到仓库。

$ *git commit -m "message"*  可以一次提交多个文件，message用于自己记录本次提交的用意。

# 时光穿梭机

git status命令查看仓库当前的状态。

$ *git status*

git diff命令查看某个文件的修改内容。

$ *git diff <file>*

git log命令查看历史记录，结果显示最近到最远的提交日志。

$ *git log*  用于查看提交历史，以便确定要回退到哪个版本

$ *git log --pretty=oneline*  单行显示各个版本的提交id，即版本号。

git reset命令指定回到某个版本。

$ *git reset --hard HEAD^*  回退到上一版本。

$ *git reset --hard <commit id>*  指定回到某个版本，id输入前几位即可。

git reflog命令用于记录每一次命令。

$ *git reflog*  结果显示每一次提交命令记录，可以在关机后查找版本号。用于查看命令历史，以便确定要回到未来的哪个版本

cat命令查看txt文件内容。

$ *cat <file>*  结果显示该文件内容。

git diff HEAD命令查看工作区和版本库里最新版本的区别。

$ *git diff HEAD -- <file>*

git checkout 命令丢弃工作区的修改。

$ *git checkout -- <file>*  撤销该文件在工作区的修改，让该文件回到最近一次git commit或git add时的状态。用于纠正还未git add的文件内容修改。

※git checkout -- <file>命令中的“--”很重要，没有“--”，就变成“切换到另一个分支”的命令。

git reset HEAD命令撤销暂存区的修改，重新放回工作区。

$ *git reset HEAD <file>*  把暂存区的修改回退到工作区，然后再用git checkout命令丢弃工作区修改。用于纠正已经git add的文件内容修改。

rm命令删除工作区文件。

$ *rm <file>*  删除后，工作区和版本库就不一致。

如果确实需要删除，就用git rm命令删掉，并且git commit。

$ *git rm <file>*

$ *git commit -m "remove <file>"*

如果是误删，就用git checkout命令把误删的文件恢复到最新版本。

$ *git checkout -- <file>*

# 远程仓库

**获得Git远程仓库**

需要注册一个GitHub账号，就可以免费获得Git远程仓库。注册网址为<https://github.com/>。由于本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以，需要进行设置。

第1步：创建SSH Key。在用户主目录下，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”*

然后一路回车，使用默认值即可，由于这个Key不是用于军事目的，所以无需设置密码。

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。

第2步：登陆GitHub，打开“Account settings”，“SSH Keys”页面：

然后，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，点击“Add Key”，就能够看到已经添加的Key。

**添加远程库**

首先，登陆GitHub，然后，在右上角找到“Create a new repo”按钮，创建一个新的仓库：

在Repository name填入learngit，其他保持默认设置，点击“Create repository”按钮，就成功地创建了一个新的Git仓库：

目前，在GitHub上的这个learngit仓库还是空的，GitHub告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。

然后，根据GitHub的提示，在本地的learngit仓库下运行命令：

$ *git remote add origin git@github.com:****Your GitHub Name****/learngit.git*

下一步，推送本地库的所有内容至远程库。

$ *git push -u origin master*  实际上是把当前分支master推送到远程。

随后，只要本地库中有更新，就可以通过git push命令把本地master分支的最新修改推送至GitHub。

$ *git push origin master*

**SSH警告**

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

*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.*

这个警告只会出现一次，后面的操作就不会有任何警告了。

**从远程库克隆**

首先，登陆GitHub，创建一个新的仓库，名字叫gitskills。勾选Initialize this repository with a README，这样GitHub会自动创建一个README.md文件。创建完毕后，可以看到该文件。

远程库准备完成后，用git clone命令克隆一个本地库。

$ *git clone git@github.com:****Your GitHub Name****/gitskills.git*

*Cloning into 'gitskills'...*

*remote: Counting objects: 3, done.*

*remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 3*

*Receiving objects: 100% (3/3), done.*

然后进入gitskills目录查看，已经存在README.md文件。

$ *cd gitskills*

$ *ls*

*README.md*

# 分支管理

创建dev分支，然后切换到dev分支。

$ *git checkout -b dev*

git checkout命令加上-b参数表示创建并切换，相当于以下两条命令：

$ *git branch dev*

$ *git checkout dev*

git branch命令查看当前分支。

$ *git branch*

*\* dev*

*master*

git branch命令会列出所有分支，当前分支前面会标一个\*号。

然后，可在dev分支上正常提交。dev分支工作完成后，可以切换回master分支。

$ *git checkout master*

切换回master分支后，由于上次修改内容在dev分支上提交，而master分支此刻的提交点并没有变。所以，需要把dev分支的工作成果合并到master分支上。

git merge命令用于合并指定分支到当前分支。

$ *git merge dev*  合并后，当前分支与被合并分支的最新提交完全一样。

*Updating d46f35e..b17d20e*

*Fast-forward*

*readme.txt | 1 +*

*1 file changed, 1 insertion(+)*

注意到上面的Fast-forward信息，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。

当合并命令中添加--no-ff参数时，表示禁用Fast forward。

$ *git merge --no-ff -m "merge with no-ff" dev*

*Merge made by the 'recursive' strategy.*

*readme.txt | 1 +*

*1 file changed, 1 insertion(+)*

因为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去。

合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而fast forward合并就看不出来曾经做过合并。

合并完成后，可以删除dev分支。

$ *git branch -d dev*

因为创建、合并和删除分支非常快，所以Git鼓励使用分支完成某个任务，合并后再删掉分支，这和直接在master分支上工作效果是一样的，但过程更安全。

当使用git merge命令无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。解决冲突就是把Git合并失败的文件手动编辑为我们希望的内容，再提交。

git log --graph命令可以看到分支合并图。该命令带参数时可以查看具体合并情况。

$ *git log --graph --pretty=oneline --abbrev-commit*

eg *\* cf810e4 (HEAD -> master) conflict fixed*

*|\*

*| \* 14096d0 (feature1) AND simple*

*\* | 5dc6824 & simple*

*|/*

*\* b17d20e branch test*

*\* d46f35e (origin/master) remove test.txt*

*\* b84166e add test.txt*

*\* 519219b git tracks changes*

*\* e43a48b understand how stage works*

*\* 1094adb append GPL*

*\* e475afc add distributed*

*\* eaadf4e wrote a readme file*

git stash命令可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作。

$ *git stash*

git stash list命令查看工作现场。

$ *git stash list*

如果工作现场的修改内容已经提交，工作现场还在，但Git把stash内容存在某个地方，需要恢复。

一可以用git stash apply恢复，但是恢复后，stash内容并不删除，需要用git stash drop来删除；二可以用git stash pop，恢复的同时删除stash内容。

$ *git stash pop*

可以多次stash，恢复的时候，先用git stash list查看，然后恢复指定的stash。

$ *git stash apply <stash name>*

git branch –D命令强行删除某个分支。

$ *git branch –D <branch name>*

git remote命令查看远程库的信息，或git remote –v命令显示更详细的信息。

$ *git remote –v*

git checkout –b命令创建远程分支到本地。

$ *git checkout -b <branch name> origin/<branch name>*  本地和远程分支的名称最好一致

git push命令推送某分支到远程库对应的远程分支上。

$ *git push origin <branch name>*

git pull命令抓取远程分支上最新的提交。

$ *git pull*

如果推送失败，提示no tracking information，则说明本地分支和远程分支的链接关系没有创建，需要设置链接。

$ *git branch --set-upstream-to <branch name> origin/<branch name>* 一般需要关联的本地分支与远程分支名称相同。

# 标签管理

由于版本库中指向某个commit的指针号码较乱，不易记忆管理，所以需要对commit id打上标签，与某个commit绑定，方便查找。

git tag命令可以打一个新标签。

$ *git tag <name>*  默认标签打在最新提交的commit上。

$ *git tag <name> <commit id>*  对指定的提交版本打上标签。

命令git tag查看所有标签。

$ *git tag*  所有标签按字母顺序排列。

git show命令查看某个标签信息。

$ *git show <tagname>*

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 <tagname> -m "message" <commit id>*

如果标签打错，git tag -d命令可以删除。

$ *git tag -d <tagname>*

git push命令推送某个标签至远程。

$ *git push origin <tagname>*

$ *git push origin –tags*  一次性推送所有尚未推送至远程的本地标签。

如果标签已经推送到远程，要删除远程标签需要先从本地删除，然后从远程库中删除。

$ *git tag -d <tagname>*

$ *git push origin :refs/tags/<tagname>*

# 自定义Git——配置别名

由于部分命令中单词过长，容易出错，所以最好可以针对某些较长命令存在简写，方便操作，提高效率。

$ *git config --global alias.<abbreviation> <original word>*

eg $ *git config --global alias.st status*

$ *git config --global alias.co checkout*

$ *git config --global alias.ci commit*

$ *git config --global alias.br branch*

以上四行命令表示用st就表示status，co表示checkout，ci表示commit，br表示branch。