**Github使用总结**

**一、Github推送提交流程及常用命令**

**1）、推送流程：**

git –version #首先检查笔记本有没有安装git,mac上默认有安装，git没有安装按此网页（https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/00137396287703354d8c6c01c904c7d9ff056ae23da865a000）

mkdir 文件名 #创建git目录文件

git init #在创建的Git目录文件下开通Git仓库

cp 文件路径 git路径 # 拷贝要上传的文件

git add README.md # 把上传文件添加到git仓库

git commit -m "first commit" # 把文件提交到git仓库 -m 后为有意义说明

git remote add origin <https://gh.51dojo.com/xxxxx/项目名.git> # 第一次需要设置远程库，以后此步骤可省略

git push -u origin master # 推送至GitHub，第一次加 -u ，以后去掉 -u

**2）、常用命令：**

git status #查看状态，查看修改信息有没有被提交

git log #查看所有提交记录的详细信息

git log -N (N是正整数) #只显示最近N次提交的详细信息

git log --before=日期(日期格式：yyyy-MM-dd) #只显示某个日期当天及以前的提交记录

git log --after=日期(日期格式：yyyy-MM-dd) #只显示某个日期当天及以后的提交记录

git log –reverse #**倒序查看提交记录**

git checkout -b dev #创建dev分支并切换到dev分支

git checkout dev #切换到dev分支

git checkout master #切换回master分支

git branch #查看当前分支

git branch –r #查看远程的所有分支

git branch –a #查看本地以及远程的所有分支

git config --global user.name "Your Name" #github账号

git config --global user.email “[email@example.com](mailto:email@example.com)” #github Email地址

git config –list #当前账号信息的查询

git diff filename #对比本地的某文件与远程仓库中的该文件的差异

git add filename1, filename2, filename3, ... #将多个文件纳入到git版本控制范围内, 此时还没提交到本地git仓库中

git add . #将当前新修改的或新增的所有文件纳入到git版本控制范围内, 此时还没提交到本地git仓库中

git rm filename #从版本库中删除该文件

git rm --cached filename #如果只希望删除 git 版本控制中的该文件/文件夹, 而不删除本地硬盘上的该文件/文件夹, 可以使用 –cached 选项

git checkout – filename #用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”

git reset commitId #版本回退到某一次提交后的情况, 当前本地新修改的代码在本地不会删除, 只会从版本控制中检出

git reset --hard commitId #版本回退到某一次提交后的情况, 当前本地新修改的代码会被删除

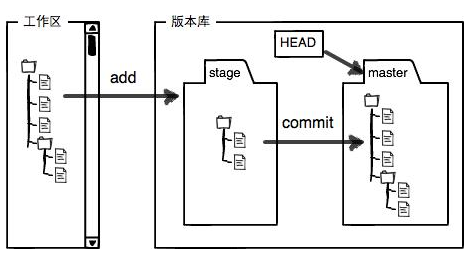
git reset [–hard] HEAD^ #回退到最近一次提交的上一次提交. 最近一次提交是用 HEAD 表示

**二、Github推送提交流程及常用命令**

Git是目前世界上最先进的分布式版本控制系统,版本控制是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统

创建一个版本库并进行初始化，Git自动为我们创建了唯一一个master分支  
提交更改，实际上就是把暂存区的**所有内容提交到当前分支**。

工作区与版本库

**工作区**  
就是你在电脑里能看到的目录  
**版本库**  
工作区有一个隐藏目录.git,就是Git的版本库  


创建版本库

**切换目录位置** cd F:/task  
**显示当前目录位置** pwd  
**创建repository** mkdir <file>(在工作区生成文件夹)  
**初始化repository** git init//把这个目录变成Git**可以管理**的仓库  
(初始化后,在当前目录下会出现一个名为.git的目录,所有git需要的数据和资源都存放在这个目录中，.git目录，这个目录默认是隐藏的，用ls -ah命令就可以看见。

工作区文件操作

**cat** <**file**> //查看当前工作区文件具体内容

git checkout -- <**file**>//其实是用版本库里的最新版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

**rm** <**file**> //从工作区删除文件

git版本库暂存区操作

git **add** <file> //添加文件到暂存区

git **add** -u//只会处理已修改或者已删除的文件，但是不会处理新建的文件

git reset HEAD file //将文件的版本库的最新版本重新放回工作区(已**add**)

git版本库操作

**保存快照到分支:**git commit -m '说明' //提交更改，实际上就是把暂存区的所有内容提交到**当前分支。**  
**查看快照记录**：git log

git log --pretty=oneline //简化信息，方便查看

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

git reflog //查看每一次**commit**以及**reset**，以便查看**commit**-**id**

**版本回退**:git reset --hard HEAD^ //回退到当前工作区版本的上一个版本

HEAD^:上一个版本

HEAD^^:上上一个版本

HEAD~100:上100个版本

**commit**-**id**(对应版本**id**)

**从版本库删除文件:**git rm <file>   
**从远程仓库删除文件:**先在工作区删掉,然后git commit -a -m "A file was deleted"，最后git push

文件状态跟踪记录

**状态查看**:git status  
***三种状态***  
1.untraked file(工作区的文件未被跟踪的文件)  
2.changes to be commited(暂存区的文件,将要被提交的修改)  
3.changes not staged for commit(没有准备提交的修改或文件之前提交过,然后进行了修改但还未add)

文件修改查看

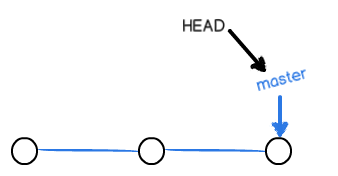
git diff <file>//(add提交前)//查看具体修改内容  
git diff HEAD -- readme.txt//查看版本库最新版本与工作区最新版本的区别

Git分支

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，所以分支就是一条commi时间线。

一个项目可以对应多条分支，也可以对应一条分支

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。**



**创建分支并切换**  
一步完成:git checkout -b <name>(-b参数表示创建并切换，)

两步完成:

git branch <**name**>//创建分支

git checkout <**name**>//切换分支到当前分支

**查看分支**:git branch  
**合并分支**:

git **merge** <**name**>//合并指定分支到当前分支,**fast** forward模式

git **merge** --no-ff <name>//可以看出曾经合并过

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。--no-ff参数，表示禁用Fast forward  
**删除分支**:

git branch -d <**name**>

git branch -D <**name**>//强行删除

**查看分支合并图**

git log --graph

git log --graph --pretty=oneline --abbrev-commit//简化信息

**本地分支与远程分支同步**

git checkout -b branch-**name** origin/branch-**name**//在本地创建和远程分支对应的分支

git branch --**set**-upstream branch-**name** origin/branch-**name**//建立本地分支和远程分支的关联

标签管理

一个版本库保存一个分支master，各自的版本库保存着各自的分支，各自的主分支名称都为master。  
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。

标签是指向某个commit的指针  
**打标签**:git tag <name> //***默认***标签是打在最新提交的commit上

git **tag** **<name**> commit-id(可选)

git **tag** **-a** <name> -m '说明' commit-id(可选)

**查看标签**：git tag  
**查看标签信息**：git show <tagname>  
**删除标签**：git tag -d <tagname>//(本地)

**推送标签到远程**:

git **push** origin <tagname>//推送单个

git **push** origin --tags//推送多个

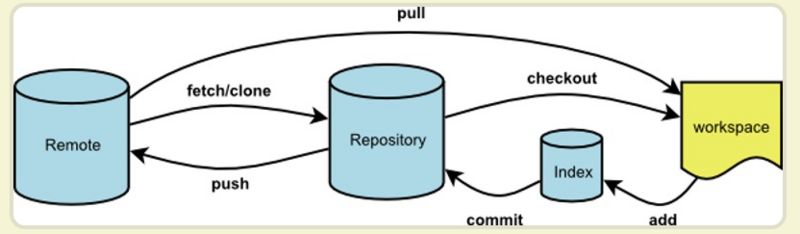
远程仓库

远程仓库是指托管在网络上的项目仓库，可能会有好多个，其中有些你只能读，另外有些可以写  
**查看当前远程库列表**：  
git remote//它会列出每个远程库的简短名字  
git remote -v //-v 选项，显示对应的克隆地址  
**查看远程仓库信息**  
git remote show [remote-name]//查看某个远程仓库的详细信息  
**添加远程仓库**  
git remote add [shortname] [url]  
**删除远程仓库**  
git remote rm [remote-name]  
**远程仓库重命名**  
git remote rename [old] [new]

远程操作

远程引用是对远程仓库的引用（指针），包括分支、标签等等。  
**远程分支**

git **ls**-remote//显示获得远程引用的完整列表



**本地仓库推送至远程库(推送分支)**   
git push [remote-name] [branch-name]  
推送分支，就是把该分支上的**所有本地提交**推送到远程库。推送时，要***指定本地分支***，这样，Git就会把**该分支**推送到远程库***对应***的***远程分支***上

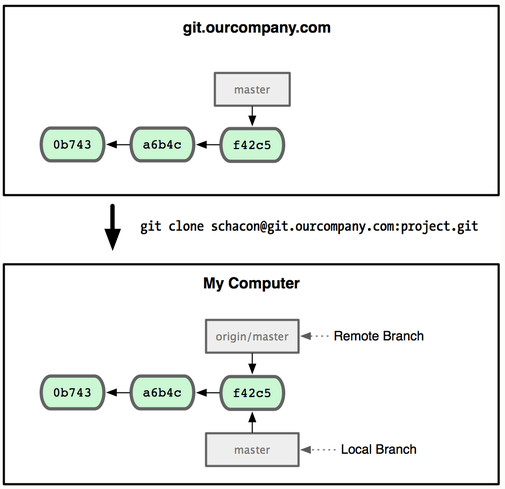
1.git remote add origin git@github.com:murphyuwu/task0001.git//***添加远程仓库origin***

git push origin master//实际上是把当前分支master推送到远程库origin。

2.git push -u origin master//推送当前分支master  
(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)

**从远程库克隆仓库到本地**  
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，如图

git clone git@github.com:murphyuwu/task0001.git



分支的创建与合并

分支创建发生的几件事

1.创建新分支时，新分支会指向现有分支最新的commit对象，如下图。  
2.现有分支在工作区以及暂存区的状态，新分支也拥有同样的状态。  
3.切换到新分支工作时，会改变之前的状态，并与新分支同步。  
正因为切换到新分支工作时，现有分支的状态会与新分支的状态保持同步，因此为了避免这一状况，可以保住现有工作区的状态，再切换到新分支工作，此时新分支的状态就是干净的

git stash //保存当前工作现场

git stash drop //删除保存

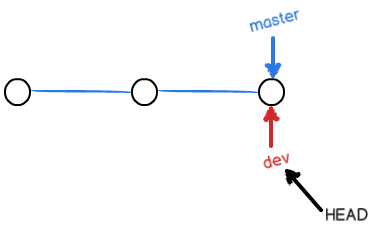
git stash apply //恢复现场，但并不删除保存

git stash pop //恢复现场，并删除保存

git stash list //查看保存工作情况

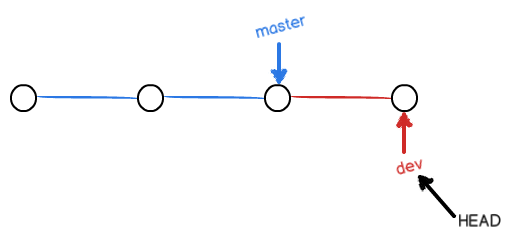
**Fast Forward**

$ git checkout -**b** dev//创建并切换分支到dev分支



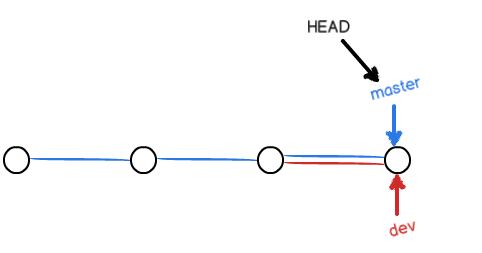
$ git **add** readme.txt

$ git commit -m "branch test"//dev分支向前移一步

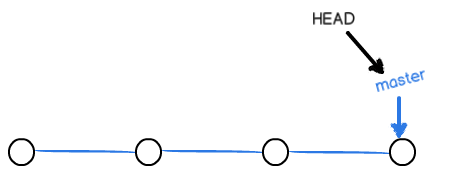


$ git checkout maste//切换到master分支

$ git **merge** dev//master向前一步与dev分支合并



$ git branch -d dev//删除dev分支

  
**冲突修改合并**

git checkout -b feature1

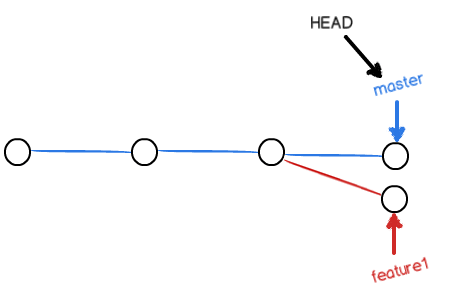
$ git **add** readme.txt

$ git commit -m "AND simple"

$ git checkout master

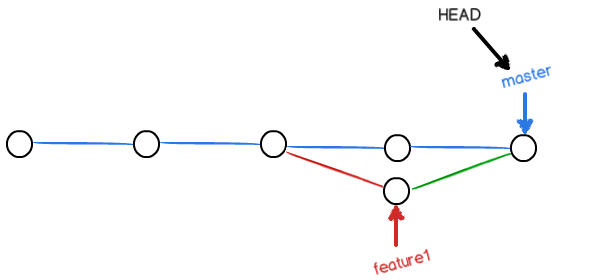
$ git **add** readme.txt

$ git commit -m "& simple"

  
这种情况下，Git无法执行“快速合并”，**只能试图把各自的修改合并起来**，但这种合并就可能会有冲突，收到修改冲突后提交

$ git **add** readme.txt

$ git commit -m "conflict fixed"



**三、Github常见问题及解决办法**

1、pull合并两个不同的项目，报错如下：

fatal: refusing to merge unrelated histories

解决办法：git pull origin master ----allow-unrelated-histories #告诉系统允许合并两个不相关的项目内容

2、待发现补充