***GIT学习笔记***

# Git简介

Git：分布式的版本控制系统

版本控制系统：集中式版本控制系统（cvs、svn），分布式版本控制系统（GIT）

GIT优势：

本地版本控制、重写提交说明（SVN就不能重写提交说明） 、分支系统、可以后悔

SVN：增量

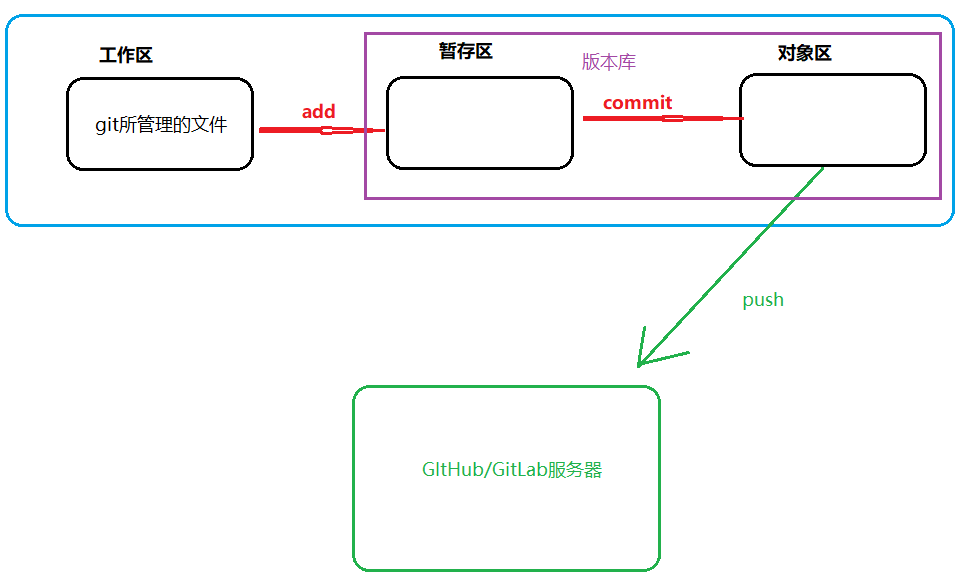
GIT：全量（每一个版本都包含全部文件，时刻保持数据的完成性）

GIT的三种状态(四种)：

已修改（modified）：

已暂存(staged)：

已提交(commited)：



# Git操作命令：

1. Git init:初始化工作区，.git目录，git版本控制目录，默认到master分支
2. 删除命令：rm -rf 目录名/文件名 -- 删除该文件/该目录及其子目录下的所有内容
3. 清屏：crtl + l /clear
4. 暂存区 --> 工作区：git rm –cached 文件名

## 查看日志相关命令：

1. Git log ：查看提交日志
2. 创建文件的命令：touch 文件名
3. Git log -n 查看最近n次的提交日志
4. Git log --pretty=oneline 把每次的提交日志一行展示，只展示id值和提交日志的注释说明
5. git log --pretty=format:"%h - %an ,%ar : %s"
6. git log –-graph
7. git log –-graph –-pretty=oneline –-abbrev-commit
8. 查看远程服务器日志：git log origin/master
   1. Git log remotes/origin/master
   2. Git log refs/remotes/origin/master

## 设置邮箱和用户名

Git config

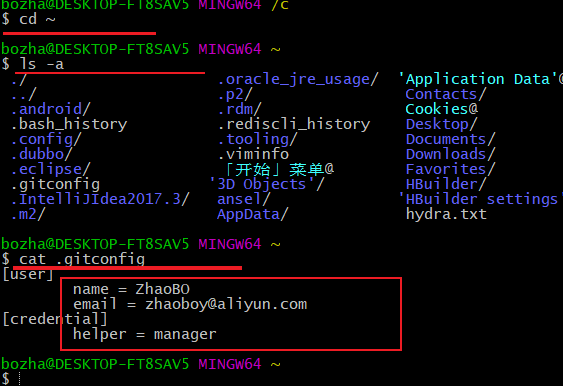
Git config –-global (基本不用，给整个计算机一次性设置)

Git config –-system (给当前用户一次性设置，推荐使用)

git config --system user.name 'zhaobo'

git config --system user.email 'zhaoboy@aliyun.com'

查看用户邮箱：~.gitconfig

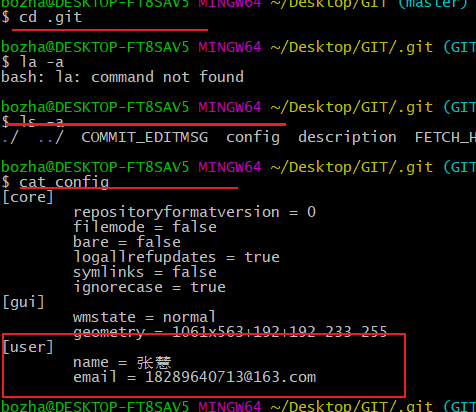


Git config –-local (给当前项目一次性设置)

git config -- local user.name 'zhaobo'

git config -- local user.email 'zhaoboy@aliyun.com'

查看用户邮箱：.git/config



三种设置用户、邮箱方法的优先级：local > system > global

1. 如果某个文件已提交，并且对其进行了修改，可以放弃修改（还原到已提交状态）

Git checkout – 文件名

1. 删除操作：git rm 文件名，执行该命令后，文件被放到了暂存区，

彻底删除：git commit -m “注释说明” 提交删除操作

Git rm 后悔：

1. 恢复到工作区git reset head 文件名
2. Git checkout – 文件名 撤销删除操作
3. 操作系统删除 rm : 1.删除 2.工作区

Git mv 重命名

1. 注释重写，重写提交说明
   1. Git commit –-amend -m “重新修改的注释说明”
2. 忽略文件：.gitignore
   1. Touch .gitignore
   2. Vi 写上要忽略的文件名
   3. 通配符：
      1. \*任意字符：如\*.properties忽略所有properties结尾的文件
      2. ！ 排除
      3. Dir/ 忽略dir目录中的所有文件
      4. Dir/\*.txt
      5. Dir/\*/\*.txt

# 分支：一条commit链，一条工作的记录线

1. 查看分支：git branch
2. 创建分支：git barnch 分支名
3. 切换分支：git checkout 分支名
4. 删除分支：git branch -d 分支名 （不能删除当前分支）
   1. 其它不能删除的情况：（包含未合并的内容）
5. 创建新分支并切换：git checkout -b 分支名
6. 合并分支：git merge 分支名
7. 强行删除分支：包含没有合并的内容也能删除分支：git checkout -D 分支名
8. 细节：
   1. 如果在分支a中进行了写操作，但此操作局限在工作区中进行，那么在master分支能看到改操作，如果分支a中进行了写操作，并且已经进行了提交，文件提交到了对象区，则master分支中无法观察到此操作
   2. 如果在分支a中进行了写操作，但此操作局限在工作区中进行，（没有add 、 commit）,删除分支a是可以成功的
9. Git branch -v:某个分支最近一次提交的sha1值
10. 分支名指向commit，HEAD值指向分支名，
11. 如果有两个分支，一个分支靠前，另一个分支落后，如果不冲突，master可以通过merge直接追赶上dev，称为fast forward
    1. Fast forward:本质是分支指针的移动，跳过的中间commit仍然会保存，merge后两个分支会归于一点
    2. Fast forward：没有分支信息，丢失分支信息
    3. Git在merge时3，默认使用fast forward，也可以禁止：git merge –-no-ff
       1. 如果不使用默认的fast forward：两个分支不归于一点commit（主动发起合并的分支，会前进一步）
       2. 分支信息完整，不丢失分支信息，

## 合并与冲突

1. 合并：如果冲突，需要解决冲突
   1. 解决冲突，修改文件，两个分支中冲突的文件内容
   2. 修改完文件后，git add 文件名；git commit -m “注释信息” {这里的add和平时添加到暂存区写法一样，但是含义不一样，没有涉及到暂存区，对象区，知识告知git冲突解决完了}
      1. 注意：分支在merge时，如果遇到冲突，解决冲突，则解决冲突会进行两次提交，一次是最终提交，一次是将对方被合并的分支的提交信息也拿了过来
      2. 如果一个分支落后，另一个分支靠前，则落后分支可以直接通过merge合并到靠前的分支，

## 版本穿梭

1. 在多个commit之间进行穿梭，回退、前进
   1. 合并add和commit：git commit -am “注释信息”
   2. 修改注释：git commit -amend -m “注释信息”
   3. 回退到上一次commit：git reset –-hard HEAD^
   4. 回退到前n次commit：git reset –-hard HEAD~n
   5. 回退到前任意次commit：git reset –-hard sha1值

## 放弃与游离操作

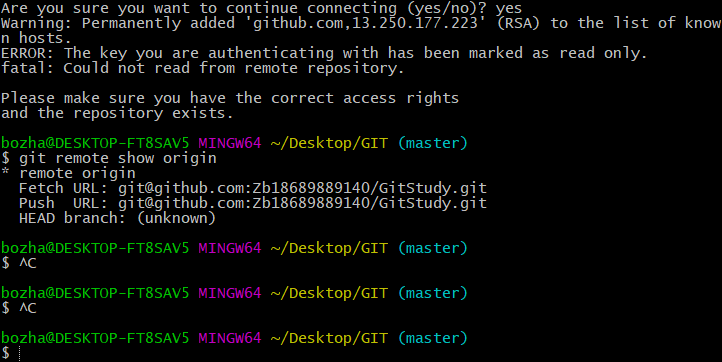
1. Checkout：放弃修改，放弃的是工作区中的修改，相对于暂存区和对象区
   1. Reset：将之前增加到暂存区的内容回退到工作区
   2. Checkout：版本穿梭（游离状态 ）git checkout sha1值
      1. 修改后，必须提交
      2. 创建分支的好时机，
2. 分支重命名：git branch -m 原分支名 新分支名

## 保存现场

1. Stash：保存现场
   1. 建议（规范）：在功能未开发完毕前，不要commit，
   2. 在没有commit之前，不能切换分支
   3. 如果还没有将一个功能开发完毕，就要切换分支：建议1.保存现场（临时保存，stash）命令：git stash
      1. Git stash list :查看有哪些保存的临时场景
      2. 还原现场：git stash pop(将原来保存的删除，用于还原内容)，默认还原最近一次
      3. 还原现场：git stash apply(还原内容，不删除原保存的内容)
      4. 删除保存的现场：git stash drop “临时现场标识”
2. 标签：git tag 版本号 如： git tag v1.0
   1. 查看标签：git tag
   2. Git tag -d 标签名
   3. 模糊查看标签：git tag -l “v\*/标签名中包含的字符”
3. Blame：责任：git blame 文件命，查看文件的所有提交commit sha1值，以及每一行的作者
4. 差异性：diff(属于linux命令)
   1. Diff 文件1 文件2
   2. Diff -u 文件1 文件2
5. Diff:linux中的命令，比较的是文件本身
   1. Git diff :比较的是区中的文件，比较的是暂存区【相当于原文件】和工作区中文件【相当于被比较的文件】的差异

## Github

1. GitHub
   1. Push:本地 -> github
   2. Pull:github -> 本地 Pull = fetch + merge
2. 删除命令
   1. Rm -rf \* :当前目录中的文件、子文件目录全部删除，不会删除隐藏文件，不过回收站
   2. 🡸=====Rm -rf /====🡺:不要执行，删除整个计算机中的全部文件【一定不要执行】
3. 推送：
   1. git remote add origin https://github.com/Zb18689889140/GitStudy.git
      1. git remote add: origin === <https://github.com/Zb18689889140/GitStudy.git> 让git知道以后使用origin来代表后面的远程仓库地址
   2. git push -u origin master :将本地的master分支，推送到远程，将本地和远程关联起来，第一次需要这样写，后续修改推送时，只需要git push就行了，不需要再加-u参数
4. SSH配置：
   1. 本地 私钥，远程 公钥
   2. Ssh-keygen 生成：私钥（本机），公钥（gitHub），生成的密钥在用户目录下
   3. 可以将公钥存放在github中的两个地方：
      1. 项目的setting中，只有当前项目可以和本机免密钥登录
      2. 账号的settings中，账户的所有项目都可以和本机免密钥
   4. git remote add origin [git@github.com:Zb18689889140/GitStudy.git](mailto:git@github.com:Zb18689889140/GitStudy.git)
   5. git remote show origin



查看连接的远程仓库：git remote show origin

Git remote show :查看远程服务器

注意：远程增加ssh的公钥时，1，删除回车符，2，勾选可写权限

1. 各种分支
   1. Dev分支：开发分支，频繁变化
   2. Test分支：基本开发完毕，交给测试实施人员的分支
   3. Master分支：很少变化，生产阶段
   4. Bugfix分支：临时修复bug分支
   5. Dev -> test -> (merge dev) -> master (merge test) ->
2. Git remote show origin:查看远程服务器，
   1. Git remote show origin:查看远程服务器中某一个具体的仓库（这里是origin仓库）
   2. Git 会在本地维护 origin/master分支，通过该分支感知远程github的内容，这个分支可以切换过去，但是切换过去是处在一个游离状态，origin/master 一般建议，不要修改，是一个只读分支

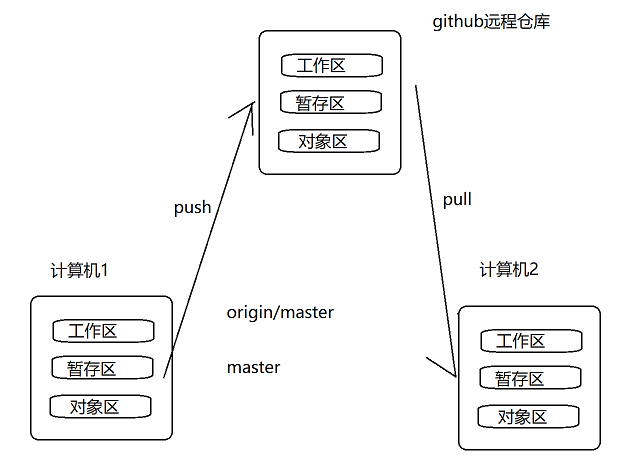
## Git多人远程操作练习

1. 模拟两个项目，项目一：notes、项目二：GitStudy
2. 给两个项目分别设置两个用户使用local级别的用户和邮箱
   1. notes项目：
      1. 用户：git config --local user.name 'zhaobo'
      2. 邮箱：git config --local user.email 'zhaoboy@aliyun.com'
   2. GitStudy项目：
      1. 用户：git config --local user.name 'zhanghui'
      2. 邮箱： git config --local user.email 'zhanghui@aliyun.com'
   3. 从远程仓库github中下载一个项目：
      1. Git clone [git@github.com:Zb18689889140/JavaStudyNote.git](mailto:git@github.com:Zb18689889140/JavaStudyNote.git),这样下载下来的项目，默认以仓库名作为项目名，如果想要指定下载项目的项目名使用如下命令

Git clone [git@github.com:Zb18689889140/JavaStudyNote.git](mailto:git@github.com:Zb18689889140/JavaStudyNote.git) 想要使用的项目名 ，就是在地址后面加上想要的项目名

* + 1. 注意：远程仓库中已经有了版本库，所以我们在下载项目的时候，需要下载到的文件夹目录中不再需要有.git文件夹目录，使用从远程仓库中下载的版本库即可，所以一删除.git目录：rm -rf .git

## Pull和push

1. 作用：push将本地文件推送到远程服务器，pull将远程文件拉去到本地，还能改变指针
2. Fast-forwar：更新，如果发现，更新的内容，比自己先一步（commit的sha1值在自己之前），则会自动合并
3. 如果有冲突：pull = fetch + merge
   1. Merge： vi 解决冲突，（解决冲突，也就是同和你同时修改文件的人协商，修改文件，） -> git add . 🡪 commit
4. 总结：解决冲突的步骤：pull 🡪 add 🡪 commit 🡪 push
5. 

## 分支：就是一个指针，commit的sha1值

查看分支：git branch -av:可以查看本地和远程的所有分支

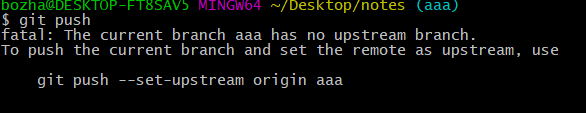
三种分支：远程分支

本地分支

本地的远程分支，（追踪分支），追踪分支是桥梁，是连接本地和远程的桥梁，本地分支通过追踪分支来感知远程分支

本地与远程分支的关联操作：

本地 -> 远程：本地有dev，远程没有dev,在本地dev执行git push,这是本地和远程没关联，直接执行git push是会报错的



解决方法一：

Git push -u origin dev

解决方法二：根据报错提示信息

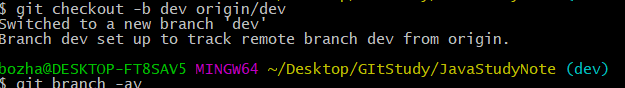
Git push –set-upstream origin 分支名

远程 -> 本地：

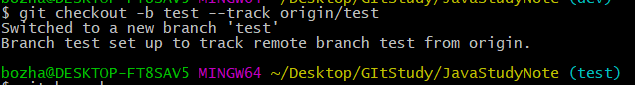
首先git pull 会把远程分支拉取下来，变为追踪分支 远程 -> 追踪

然后再把追踪分支变为本地分支 追踪 -> 本地

方法一：git checkout -b 分支名（比如dev） origin/分支名



方法二：git checkout -b 分支名（比如test） –-track origin/分支名(比如test)



## 操作本地和远程的分支

删除本地分支：git branch -d 分支名

删除远程分支：格式：git push origin src:dest,如删除test分支：git push origin :test(空就代表删除) 方法二：git push origin -–delete 分支名（如dev）

场景：本地没有a分分支，但本地却感知远端a分支

检测：git remote prune origin –-dry-run

清理无效的 追踪分支（本地种感知远程的分支）：git remote prune origin

git push origin src:dest

git push origin dev:dev2

git push origin HEAD:dev2

git pull origin 本地分支名（如dev）:远程分支名,相当于git pull + : git checkout -b dev origin/分支名

将远端分支，拉取到本地某个分支：git pull

## 标签：git tag：查看标签

本地 -> 远程

Git tag v1.0 简单标签，只存储当前的commit的sha1值

Git tag -a v2.0 -m “我的v2.0版本”，（创建一个新对象，会产生一个新的commit/sha1）存储信息，其中包含了当前commit的sha1值

推送标签到远程：git push origin 标签名 . . .(可以写多个标签名，同时推送多个标签)

（推荐使用）：git push origin –-tags : 将本地所有未推送远程的标签推送到远程

推送标签完整的写法：git push origin refs/tags/标签名：refs/tags/标签名

远程 -> 本地

获取远程标签：git pull命令能拉取到远程所有的东西，包括标签

注意：注意git pull命令，如果远端是新增标签，则pull可以将新增标签拉取到本地，但是如果远程是删除标签，则pull无法感知，

这时远程删了，本地还得再手动删：git tag -d 标签名

只从远程拉取某个标签：git fecth origin tag 标签名（如v4.0）

删除远程标签：

Git push origin :标签名（如v4.0）

注意：如果将远程标签删除，其他用户无法直接感知

## Git gc

Git gc就是对分支，分支的提交点进行压缩

Objects、refs中记录了很多commit的sha1值，如果执行git gc.则会将这么多sha1值存放到一个压缩文件中

## Git 裸库/bare

没有工作区的git仓库，叫做裸库，裸库不在本地，在远程，远程是可以没有工作区的，本地就不能够没有工作区，否则将无法进行增删改操作

初始化一个裸库：git init –-bare

裸库可以推到远程，但是无法进行add操作，因为没有工作区

## Submodule:子模块

应用场景：在一个仓库中，引用另一个仓库的代码

A库中有B库，但B库push之后无法直接感知，需要主动操作：更新submodule:pull操作

注意：这里想要pullB库中的内容，不能在A库中pull，要首先进到B库中再pull,然后还要再push一次，在远程的A中才能看到B的内容，推B的时候又要在A里面推B

Pull 的时候进Bpull，推的时候在A推

简化方法：如果一个项目有很多子模块，如果pull的时候都得进到每个子模块中pull就非常的麻烦，这时用下面的命令，不需要进入到子模块中，就能pull每一个子模块

Git submodule foreach git pull

A中有B库，但B push之后 A无法直接感知，需要主动操作：pull

* + - 1. 进入A/B中pull
      2. 直接在A中迭代pull(将A中的所有submodule全部pull):git submodule foreach git pull

B ：修改 -> push -> A (本地 + 远程) 无法感知

本地pull: 本地有，A的远程没：本地add ..commit -> push

如果push B,则B本身能够感知；但是A中的B不能直接感知

如果克隆的项目包含submodule,则克隆空方法：git clone 仓库地址 --recursive

## Git subTree

结论：在做subtree

如果加squash,以后每次都加，如果不加，都不要加

核心流程：子 -> 父工程 有反应

1. 修改子工程push
2. (本地) 将gitHub中的子工程更新到父子模块
3. 将父工程中的子模块更新情况，推送到对应的gitHub上(父 – 子)