**集中式(svn)**

svn因为每次存的都是差异 需要的硬盘空间会相对的小一点 可是回滚的速度会很慢  
优点:   
  代码存放在单一的服务器上 便于项目的管理  
缺点:   
  服务器宕机: 员工写的代码得不到保障  
  服务器炸了: 整个项目的历史记录都会丢失

**分布式(git)**

git每次存的都是项目的完整快照 需要的硬盘空间会相对大一点  
  (Git团队对代码做了极致的压缩 最终需要的实际空间比svn多不了太多 可是Git的回滚速度极快)  
优点:  
  完全的分布式  
缺点:      
  学习起来比SVN陡峭

### 底层命令

git对象  
  git hash-object -w fileUrl : 生成一个key(hash值):val(压缩后的文件内容)键值对存到.git/objects  
tree对象  
  git update-index --add --cacheinfo 100644 hash test.txt : 往暂存区添加一条记录(让git对象 对应 上文件名)存到.git/index  
  git write-tree : 生成树对象存到.git/objects  
commit对象  
  echo 'first commit' | git commit-tree treehash : 生成一个提交对象存到.git/objects  
对以上对象的查询  
  git cat-file -p hash       : 拿对应对象的内容  
  git cat-file -t hash       : 拿对应对象的类型

### 查看暂存区

git ls-files -s

### 安装

git --version

### 初始化配置

git config --global user.name "damu"  
git config --global user.email damu@example.com      
git config --list

### 初始化仓库

git init

### C(新增)

在工作目录中新增文件  
git status  
git add ./  
git commit -m "msg"

### U(修改)

在工作目录中修改文件  
git status  
git add ./  
git commit -m "msg"

### D(删除 & 重命名)

git rm 要删除的文件 git mv 老文件 新文件 git status git status git commit -m "msg" git commit -m "msg"

### R(查询)

git status : 查看工作目录中文件的状态(已跟踪(已提交 已暂存 已修改) 未跟踪) git diff : 查看未暂存的修改 git diff --cache : 查看未提交的暂存 git log --oneline : 查看提交记录

### 分支

分支的本质其实就是一个提交对象!!!  
HEAD:   
  是一个指针 它默认指向master分支 切换分支时其实就是让HEAD指向不同的分支  
  每次有新的提交时 HEAD都会带着当前指向的分支 一起往前移动  
git log --oneline --decorate --graph --all : 查看整个项目的分支图    
git branch : 查看分支列表  
git branch -v: 查看分支指向的最新的提交  
git branch name : 在当前提交对象上创建新的分支  
git branch name commithash: 在指定的提交对象上创建新的分支  
git checkout name :     切换分支  
git branch -d name : 删除空的分支 删除已经被合并的分支  
git branch -D name : 强制删除分支

### git分支本质

分支本质是一个提交对象,所有的分支都会有机会被HEAD所引用(HEAD一个时刻只会指向一个分支)  
当我们有新的提交的时候 HEAD会携带当前持有的分支往前移动

### git分支命令

创建分支           : git branch branchname  
切换分支           : git checkout branchname  
创建&切换分支     : git checkout -b branchname  
版本穿梭(时光机) : git branch branchname commitHash    
普通删除分支     : git branch -d branchname  
强制删除分支     : git branch -D branchname  
合并分支         : git merge branchname  
  快进合并 --> 不会产生冲突  
  典型合并 --> 有机会产生冲突  
  解决冲突 --> 打开冲突的文件 进行修改 add commit

查看分支列表 : git branch  
查看合并到当前分支的分支列表: git branch --merged  
  一旦出现在这个列表中 就应该删除  
查看没有合并到当前分支的分支列表: git branch --no-merged  
  一旦出现在这个列表中 就应该观察一下是否需要合并

### git分支的注意点

在切换的时候 一定要保证当前分支是干净的!!!  
  允许切换分支:   
      分支上所有的内容处于 已提交状态      
      (避免)分支上的内容是初始化创建 处于未跟踪状态  
      (避免)分支上的内容是初始化创建 第一次处于已暂存状态  
  不允许切分支:  
        分支上所有的内容处于 已修改状态 或 第二次以后的已暂存状态    
          
在分支上的工作做到一半时 如果有切换分支的需求, 我们应该将现有的工作存储起来  
  git stash : 会将当前分支上的工作推到一个栈中  
  分支切换 进行其他工作 完成其他工作后 切回原分支  
  git stash apply : 将栈顶的工作内容还原 但不让任何内容出栈   
  git stash drop : 取出栈顶的工作内容后 就应该将其删除(出栈)  
  git stash pop   :     git stash apply + git stash drop   
  git stash list : 查看存储

### 后悔药

撤销工作目录的修改   : git checkout -- filename  
撤销暂存区的修改     : git reset HEAD filename  
撤销提交             : git commit --amend

### reset三部曲

git reset --soft commithash   ---> 用commithash的内容重置HEAD内容  
git reset [--mixed] commithash ---> 用commithash的内容重置HEAD内容 重置暂存区  
git reset --hard commithash   ---> 用commithash的内容重置HEAD内容 重置暂存区 重置工作目录

### 路径reset

所有的路径reset都要省略第一步!!!  
  第一步是重置HEAD内容 我们知道HEAD本质指向一个分支 分支的本质是一个提交对象   
  提交对象 指向一个树对象 树对象又很有可能指向多个git对象 一个git对象代表一个文件!!!  
  HEAD可以代表一系列文件的状态!!!!  
git reset [--mixed] commithash filename    
    用commithash中filename的内容重置暂存区

### checkout深入理解

git   checkout brancname 跟   git reset --hard commithash特别像  
  共同点  
      都需要重置 HEAD   暂存区   工作目录  
  区别  
        checkout对工作目录是安全的   reset --hard是强制覆盖  
        checkout动HEAD时不会带着分支走而是切换分支  
        reset --hard时是带着分支走  
          
checkout + 路径  
    git checkout commithash filename     
          重置暂存区  
          重置工作目录  
    git checkout -- filename    
        重置工作目录

### eslint

js代码的检查工具  
下载: npm i eslint -D  
使用:  
  生成配置文件 npx eslint --init  
  检查js文件   npx eslint 目录名  
  命中的规则:  
      字符串必须使用单引号  
      语句结尾不能有分号  
      文件的最后必须要有换行

### eslint结合git

husky: 哈士奇, 为Git仓库设置钩子程序  
使用  
  在仓库初始化完毕之后 再去安装哈士奇  
  在package.json文件写配置  
      "husky": {  
          "hooks": {  
            "pre-commit": "npm run lint"     
            //在git commit之前一定要通过npm run lint的检查  
            // 只有npm run lint不报错时 commit才能真正的运行  
          }  
        }

### 三个必须懂得概念

本地分支  
远程跟踪分支(remote/分支名)  
远程分支

### 远程协作的基本流程

第一步: 项目经理创建一个空的远程仓库  
第二步: 项目经理创建一个待推送的本地仓库  
第三步: 为远程仓库配别名 配完用户名 邮箱  
第四步: 在本地仓库中初始化代码 提交代码  
第五步: 推送  
第六步: 邀请成员  
第七步: 成员克隆远程仓库  
第八步: 成员做出修改  
第九步: 成员推送自己的修改  
第十步: 项目经理拉取成员的修改

### 做跟踪

克隆才仓库时 会自动为master做跟踪  
本地没有分支  
  git checkout --track 远程跟踪分支(remote/分支名)  
本地已经创建了分支  
  git branch -u 远程跟踪分支(remote/分支名)

### 推送

git push

### 拉取

git pull

### pull request

让第三方人员参与到项目中 fork

### 使用频率最高的五个命令

git status  
git add  
git commit  
git push  
git pull

.git/objects/06/e21bb0105e2de6c846725a9a7172f57dd1af96 workspae项目的第一个版本(树对象) .git/objects/56/0a3d89bf36ea10794402f6664740c284d4ae3b test.txt文件的第一个版本(git对象)

.git/objects/9d/74ec4055e0f1edc1921d749c250380ca7b5ebd workspae项目的第二个版本(树对象) .git/objects/c3/1fb1e89d8b6b3ef34cdb5a2f999d6e29b822ba test.txt文件的第二个版本(git对象) .git/objects/ea/e614245cc5faa121ed130b4eba7f9afbcc7cd9 new.txt文件的第一个版本(git对象)

git操作最基本的流程 创建工作目录 对工作目录进行修改 git add ./ git hash-object -w 文件名(修改了多少个工作目录中的文件 此命令就要被执行多少次) git update-index ... git commit -m "注释内容" git write-tree git commit-tree git高层命令(CRUD) git init 初始化仓库 git status 查看文件的状态 git diff 查看哪些修改还没有暂存 git diff --staged 查看哪些修改以及被暂存了 还没提交 git log --oneline 查看提交的历史记录 git add ./ 将修改添加到暂存区 git rm 文件名 删除工作目录中对应的文件 再将修改添加到暂存区 git mv 原文件名 新文件名 将工作目录中的文件进行重命名 再将修改添加到暂存区 git commit git commit -a git commit -a -m 注释 将暂存区提交到版本库

git高层命令(分支) git branch 显示分支列表 git branch 分支名 创建分支 git checkout 分支名 切换分支 git branch -D 分支名 强制删除分支

### 切换分支

最佳实践: 每次切换分支前 当前分支一定得是干净的(已提交状态)  
坑:   
  在切换分支时 如果当前分支上有未暂存的修改(第一次) 或者 有未提交的暂存(第一次)  
      分支可以切换成功 但是这种操作可能会污染其他分支  
动三个地方  
  HEAD  
  暂存区  
  工作目录

### 后悔药

工作区  
  如何撤回自己在工作目录中的修改 : git checkout --filename  
暂存区  
  如何何撤回自己的暂存 : git reset HEAD filename  
版本库                
  如何撤回自己的提交   : git commit --amend  
      1.注释写错了,重新给用户一次机会改注释

### reset

git log   :    
git reflog : 主要是HEAD有变化 那么git reflog机会记录下来  
三部曲  
  第一部： git rest --soft HEAD~ (--amend)    
      只动HEAD (带着分支一起移动)        
  第二部: git reset [--mixed]   HEAD~   
      动HEAD   (带着分支一起移动)    
      动了暂存区  
  第三部: git reset --hard HEAD~     
        动HEAD   (带着分支一起移动)    
        动了暂存区  
        动了工作目录

### checkout

git checkout commithash   &   git reset --hard commithash           
  1. checkout只动HEAD   --hard动HEAD而且带着分支一起走  
  2. checkout对工作目录是安全的   --hard是强制覆盖工作目录  
​  
git checkout commithash  
git checkout --filename    
    相比于git reset --hard commitHash --filename    
    第一 第二步都没做  
    只会动了工作目录  
git checkout commithash <file>      
    将会跳过第 1 步   
    更新暂存区   
    更新工作目录

### 路径reset

git reset HEAD filename     (reset 将会跳过第 1 步)      
  动了暂存区

### 远程分支

### 远程跟踪分支

### 本地分支

正常的数据推送 和 拉取步骤  
  1. 确保本地分支已经跟踪了远程跟踪分支  
  2. 拉取数据 : git pull  
  3. 上传数据: git push  
     
一个本地分支怎么去跟踪一个远程跟踪分支  
  1. 当克隆的时候 会自动生成一个master本地分支(已经跟踪了对应的远程跟踪分支)  
  2. 在新建其他分支时 可以指定想要跟踪的远程跟踪分支  
          git checkout -b 本地分支名 远程跟踪分支名  
          git checkout --track 远程跟踪分支名   
  3. 将一个已经存在的本地分支 改成 一个跟踪分支     
          git branch -u 远程跟踪分支名

### 团队协作

1. 项目经理初始化远程仓库  
  一定要初始化一个空的仓库; 在github上操作  
2. 项目经理创建本地仓库  
  git remote 别名 仓库地址(https)  
  git init ; 将源码复制进来  
  修改用户名 修改邮箱  
  git add  
  git commit   
3. 项目经理推送本地仓库到远程仓库  
  清理windows凭据  
  git push 别名 分支 (输入用户名 密码;推完之后会附带生成远程跟踪分支)  
4. 项目邀请成员 & 成员接受邀请  
    在github上操作    
​  
5. 成员克隆远程仓库  
  git clone 仓库地址 (在本地生成.git文件 默认为远程仓库配了别名 orgin)  
              只有在克隆的时候 本地分支master 和 远程跟踪分支别名/master 是有同步关系的  
6. 成员做出贡献  
  修改源码文件  
  git add   
  git commit   
  git push 别名 分支 (输入用户名 密码;推完之后会附带生成远程跟踪分支)   
     
7. 项目经理更新修改  
  git fetch 别名 (将修改同步到远程跟踪分支上)  
  git merge 远程跟踪分支

### 冲突

git本地操作会不会有冲突?  
  典型合并的时候  
git远程协作的时候 会不会有冲突?  
  push  
  pull