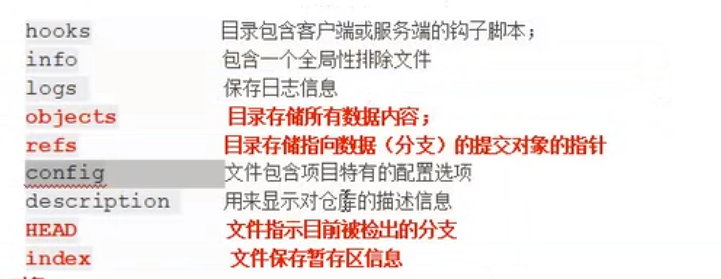
# 分布式的版本控制系统

1. 每个客户端本地都储存了项目的所有历史版本（git人员对其进行极致的压缩）
2. Git中有区域和对象的概念
   1. 区域：
      1. 工作区
      2. 暂存区
      3. 版本库
   2. 对象
      1. Git对象：所有的文件的每个版本的内容都是以哈希值名的压缩文件保存在.git文件夹下的objects文件夹里。Git对象代表文件的一次次版本
         1. Git hash-obejct -w fileUrl ：生成一个键值对存到 .git/objects
      2. 树对象：代表项目的一次次版本
         1. Git update-index - -add - -cacheinfo 100644 hash text.txt ：往暂存区添加一条记录，（让git对象对应文件名）
      3. 提交对象： 我们需要知道的对象，能看到有谁提交了版本，每一个代表项目的一次版本
         1. Echo “first commit” | git commit-tree treehash :生成一个提交对象存到.git/objects
      4. 对以上对象的查询
         1. Git cat-file -p hash : 拿到对应对象的内容
         2. Git cat-file -t hash ：拿到对应对象的类型

# Git初始化配置

1. 初始化配置git，用来说明你是谁。
   1. Git config --global user.name “xxxx”
   2. Git config --global user.email xxxxx
2. 除--global 还--system（对该电脑所有用户都适用），还有啥都不写（只对当前项目有用）
3. Git config –list 查看当前的配置
4. 删除初始化的全局配置
   1. Git config - -global - -unset user.name
   2. Git config - -global - -unset user.email

# GIT的底层命令

1. Git init 初始化仓库
2. .git隐藏文件夹中的文件夹分别为
   1. 
3. Clear 清屏
4. Echo ‘xxxx’ 往控制台输出信息
   1. Echo ‘xxx’ > test.txt 创建test文件，且输入xxx
5. Ll 遍历当前文件夹
6. Find 目录名 ：打印对应目录下的子孙文件和目录
7. Find 目录名 -type f 打印对应目录下的子孙文件
8. Rm 文件
9. Mv 文件 剪切（重命名）
10. Cat 文件，查看文件的内容
11. Vim 文件
    1. 按i进入编辑模式
    2. 按esc退出i模式后， :wq后保存退出
12. Git cat-file -p hash : 拿到对应对象的内容
13. Git cat-file -t hash ：拿到对应对象的类型
14. Git ls-files -s 查看暂存区的内容

# Git的高层命令

1. Git add ./ 将修改添加到暂存区（其实是提到版本库，再拉回暂存区）
   1. Git hash-object -w 文件名（修改了多少个工作目录中的文件，这个命令就被执行多少次）
   2. Git update-index …
2. Git commit -m “xxxx” 将暂存区提交到版本库
   1. Git write-tree
   2. Git commit-tree
3. Git init 初始化新仓库
4. Git status 检查当前文件的状态
   1. Git中的文件有两种状态
      1. 已跟踪 （被git add过的文件）
         1. 已修改 modified
         2. 已暂存 staged
         3. 已提交 commited
      2. 未跟踪 （未被git add过的文件）
5. Git diff 查看哪些还没有暂存
6. Git diff –staged 查看哪些修改已经被暂存了，还没提交的
7. Git commit -m 提交+注释
8. Git commit -a 跳过暂存区直接提交
9. Git commit -a -m
10. Git mv xxxx xxxx ：此步骤相当于下面三步
    1. Mv
    2. Rm
    3. Git add
11. Git rm xxx ： 此步骤相当于下面两步
    1. Rm
    2. Git add
12. Git log 查看提交记录
    1. Git log - - oneline 可以简略显示。显示的hash值是提交对象
    2. Git log -–oneline – - decorate –- graph – -all 查看所有的分支的历史记录
    3. 太长的命令可以用配别名
       1. Git config - -global alias.xxx “log –oneline - -decorate - -。。。。。”
13. Git reflog 查看完整操作记录

# Git分支操作

1. 分支的本质就是一个提交对象
2. HEAD : 是一个指针，它默认指向master分支，切换分支时就是让HEAD指向不同的分支，每次有新的提交时，HEAD就会带着当前的分支往前移动（HEAD移动，分支不会移动）。
3. ,git/refs目录中保存了所有的分支，及其对应的提交对象的hash值（每次提交对象被新提交时，hash值都会改变）
4. 切换分支会改变workspace中的文件。
   1. 每次切换分支前，当前分支一定得是干净的（已提交状态）
   2. 因为在切换分支时，当前分支有未暂存的新创建的modified的文件 或者有未提交的的新创建的staged的文件的时候，会把此文件一次带到切换的分支上，会污染切换的分支
5. 操作命令
   1. Git branch xxx ：在当前提交对象上创建一个新的分支
   2. Git branch xxx Hash值 ：在指定的提交对象上创建新的分支（时光机）
   3. Git branch ：查看分支列表
   4. Git checkout branch 切换分支
      1. Git checkout -b xxx 创建新的分支，并切换至此分支
   5. Git branch -d xxx :删除一个合并了的分支
   6. Git branch -D xxx：强制删除一个分支（不管合并了否）
   7. Git branch -v 可以查看所有分支的最后一次提交

# Git存储

1. 当你大工作做一半，但是有其他小工作要做，又不想提交当前分支的话，可以用存储
2. 常用命令
   1. Git stash 将未完成的modified保存到一个栈上
   2. Git stash list 查看栈上有哪些修改
   3. Git stash apply 将保存到栈上的第一个modified重新取出
      1. Git stash apply stash@{x} 指定取出哪个modified
   4. Git statsh drop 删除删上第一个
   5. Git statsh pop 取出修改，删除栈上的第一个

# Git撤销和重置（主要用下面三个命令）

1. git checkout - - filename : 撤回自己在工作区的修改
2. git reset filename 撤回自己在暂存区的修改
   1. 相当于reset第二步曲的缩写
3. git commit - - amend 再次提交暂存区内已修改的文件(类似撤回自己在版本库的提交，再提交一次)

# reset

1. 三步曲
   1. 第一步：git reset - -soft HEAD~
      1. 一步：
         1. 动HEAD，带着当前branch一起往前一版本后退（撤销）
      2. （类似git commit - -amend，不过- -amend撤回然后提交了）
      3. Git reset - -soft 提交对象的hash值 ：HEAD带着branch去指定的提交对象上
   2. 第二步：git reset [- -mixed] HEAD~
      1. 两步：
         1. 动HEAD ,带着当前branch一起撤退
         2. 动了暂存区。
      2. git reset - -mixed 提交对象的hash值 ：HEAD带着branch去指定的提交对象上
      3. 可缩写成：git reset
   3. 第三步：git rest - -hard HEAD~（reset命令唯一的危险用法）
      1. 三步：
         1. 动HEAD ,带着当前branch一起撤退
         2. 动了暂存区
         3. 动了工作区（会将工作区的内容覆盖）
      2. 一般撤销工作区用：Git checkout - -filename
         1. 跳过第一步，第二步，
         2. 只做第三步：动了工作区

# 标签

1. 标签跟分支很像，只是分支能动，标签不能动
2. 常用命令
   1. Git tag 查看当前标签名
   2. Git tag <标签名> [可选：提交对象的hash值] 新建标签
   3. Git show <标签名> :查看标签
   4. Git tag -d <标签名> ：删除标签
   5. Git checkout <标签名> ：切换到tag
      1. 如果当前标签没有分支时，会产生头部分离。需要在当前标签上创建分支
         1. git checkout -b <分支名>

# 远程仓库

1. git remote add <别名> url ： 让本地仓库跟远程仓库连起来
2. git remote -v ：查看远程仓库
3. git remote show 别名 ：查看远程仓库的更多信息
4. git remote rename 旧别名 新别名 ：重命名远程仓库
5. git remote rm 别名 ：删除远程仓库
6. git push 远程仓库别名 分支名
7. git clone 远程仓url ：将远程仓库拉到本地
   1. git clone下来的远程仓库的别名，自动设置成origin
8. 自己创建的仓库，可以进行push,但是如果是从别人的仓库clone下来的话，如果没有那个人的成员邀请的话，无法进行push

# 团队协作流程

1. 项目经理初始化远程仓库
   1. 在github上初始化一个空的仓库
2. 项目经理创建本地仓库
   1. Git init
   2. 将源码复制进来
   3. 修改用户名，修改邮箱
3. 项目经理将本地仓库跟远程仓库链接起来
   1. Git remote add 别名 url
4. 项目经理将本地仓库推送到远程仓库
   1. 清理windows凭据
   2. Git push 别名 分支 （输入github的用户名，密码，退完后会附带生成远程跟踪分支xxx/xxx）
5. 项目成员在自己本地克隆远程仓库
   1. Git clone 仓库地址
      1. 默认为远程仓库配了别名origin
      2. 附带生成远程跟踪分支xxx/xxx
6. 项目成员做出贡献
   1. Git add
   2. Git commit
   3. Git push 别名 分支（输入github的用户名，密码）
7. 项目经理更新修改
   1. Git fetch 别名（将修改同步到远程跟踪分支上）
   2. Git merge远程跟踪分支（将同步了的远程跟踪分支与本地分支合并）

# 深入理解远程库

远程跟踪分支

1. 远程跟踪分支是远程分支状态的引用，它们是不能自己移动的，在每次网络通信操作时，自动移动
2. 如果本地分支没有跟踪任何远程跟踪分支的话，是无法进行网络通信操作的
3. 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. 在新建其他分支的时候，可以指定想要跟踪的远程跟踪分支
      1. Git checkout -b 本地分支名 远程跟踪分支名
      2. 或者 git checkout - -track 远程跟踪分支名（创建一个跟远程分支同一个名字的本地分支和远程跟踪分支，并让本地分支跟踪远程跟踪分支）
         1. 如果没有对应的远程跟踪分支的话，可以用git fetch去远程仓库拿
   3. 将一个已经存在的本地分支名，跟踪一个远程跟踪分支
      1. Git branch -u 远程跟踪分支名
      2. 没有远程跟踪分支的话，无法进行git push,pull操作
   4. 可以用gir branch -vv 查看当前分支所跟踪的远程跟踪分支

冲突

1. 当git pull的时候，没有将暂存区的内容提交时，会冲突
2. 当git push的时候，远程仓库的内容已经被修改的时候，会冲突

删除远程分支

1. Git push 仓库别名 - -delete 远程分支
2. Git remote prune 仓库别名- -dry-run ：列出已经不存在的运程仓库上，仍在跟踪的无用分支
3. Git remote prune 仓库别名 : 清楚上面列出的无用分支

Pull request

1. 可以通过fork别人的项目，修别人的bug，给别人提request。（此步骤可以用github操作，不需要命令行）
2. 一般只有大神才会用到。。