

Git学习笔记

课程: [【公开课】Git应用详解【全】风中叶老师](#)

相关命令行命令 (Unix等系统推荐oh my zsh插件)

`ls -al` 列出当前目录下包括隐藏的所有文件

`rm -rf .git` `-rf`表示递归删除 (无需确认), 删除.git文件下所有文件

`cat test.txt` 查看test.txt文件的内容

`pwd` 显示当前目录所

`echo 'welcome' > test.txt` 将文件内容转为welcome

`ctrl A/E` 移动到开头/结尾

`cd -` 返回上一个目录

`cd ~` 返回根目录

`winpty tree.com` 展示当前目录下的文件树, `tree` 命令也可 (自行安装), 但是中文乱码

`:set number` 可以看到行号, `: number` 可以跳到对应行, `dd` 删除对应行

`:start,endd` 删除start行到end行, 如 `2,4d` 删除234行

git 命令

核心命令

`git add .` 提交所有的工作区文件 (建议先完善.gitignore)

`git commit -m message` `message`是对提交文件的注释, 强制要求

`git commit -am message` 直接将工作区的文件添加到暂存区并提交 (只能对版本库中已有文件进行, 新增文件仍需手动add)

`git commit --amend -m message` 修正上一次的提交消息

`git log -n` 显示最近n条提交记录, **倒叙显示**, 包括提交ID (commit ID, 这是一个由SHA1计算的摘要值) 以及每次提交相关的信息, `--pretty=oneline` 极简显示提交信息

`git log --graph` 以图形式显示提交记录

`git log --graph --abbrev-commit --pretty=oneline` `abbrev-commit`表示简写commitID

`git rm --cached test.txt` 将文件从**暂存区**删除

`git reset test.txt` 将之前添加到暂存区的内容移回工作区, 操作的是暂存区

`git reset HEAD test.txt` HEAD可以改为某一版本的commit ID，表示将该版本中的对应文件拉回到暂存区

`git checkout -- test.txt` 丢弃相对于暂存区中最后一个添加的文件内容所做的变更，即操作的是工作区

`git rm test.txt` 主要分为两步，首先删除文件，而后将该修改纳入到暂存区，撤销操作需要先将修改从暂存区回到工作区（reset命令），而后通过checkout撤销删除文件的操作。如果使用 `rm test.txt` 来删除文件，只需直接使用checkout语句即可恢复，因为这种删除操作不经过暂存区，如果要删除版本库中的内容，还需要add并commit

`git status` 显示当前状态（所处分支，文件状态，untracked file表示还在工作区）

`git config --list` 列出config信息

`git mv test.txt test2.txt` 将test文件重命名为test2，实际过程是先删除test文件，新建一个test2文件，其与 `mv` 命令的区别同上，及是否纳入暂存区

git分支操作

`git branch` 显示当前所有分支，当前所在分支前标*

`git branch -av` 显示当前所有分支以，-a表示显示对应的远程分支，-v表示显示最近的提交

`git branch new_branch` 创建new_branch新分支

`git checkout new_branch` 切换到new_branch分支

`git checkout new_branch commit_id` 创建以commit_id版本为基础的新分支

`git checkout -b new_branch` 创建并切换到new_branch分支

`git checkout -` 切换回上一个分支

`git merge new_branch` 将new_branch分支上的修改合并到master主分支

`git branch -d new_branch` 删除new_branch分支，如果在新分支上有操作，需要先合并

`git branch -v` 显示各分支最近一次的提交信息

`git branch -m branch new_name` 重命名分支

`git merge --no-ff dev` 合并时禁用fastforward模式（该模式直接移动对应指针，没有该次合并记录，也就是会丢失原本的分支信息），会增加一份合并的commit记录

版本回退

`git reset --hard HEAD^` 回退到上一个提交，^表示回到上一个，若回退到之前两个，^^即可

`git reset --hard HEAD~n` 回到之前的第n个提交

`git reset --hard commit_id` 回退到commit_id的提交（无需写全，前四个字母左右就行）

`git checkout commit_id` 也可以回退到某一个版本

`git reflog` log记录的时提交日志，reflog记录的是操作日志，防止在回退到之前版本后丢失后来的commitID

stash暂存

`git stash` 保存当前所有工作状态，以应对临时切换分支的需求

`git stash list` 显示所有保存的工作文件

`git stash save message` 更新stash中的保存，message未注释

`git stash pop` 删除stash最新的保存状态，并将其内容恢复到工作区

`git stash apply` 不删除stash最新的保存状态的情况下，将其内容恢复到工作区

`git stash apply stash@{0}` 恢复到stash区中第0个（最新的）状态

`git stash drop stash@{0}` 手动删除stash中第0个（最新的）状态

标签

`git tag v1.0.1` 给最新的commit打上1.0.1的标签

`git tag -a v1.0.1 -m 'release version'` 带有附注的标签

`git tag -d tag_name` 删除标签

`git tag -l 'v*'` 查找v开头的标签

`git show v1.0` 显示对应版本的标签信息以及对应的commit提交信息

标签不依赖于某一特定分支，即所有分支都共享标签

`git push origin v1.0` 需要push特定标签，git push不会默认推送标签

`git push origin --tags` 将本地尚未推送到远程的标签一次性都推送到远程

diff

`git blame` 显示上一次修改的作者信息

`diff -u a b` 比较文件a b，其中+分别代表两个文件，其中@@ -1, 3 +1, 3 @@表示显示第一个文件的1-3行以及第二个文件的1-3行，其中-u默认显示三行对比数据，注意b.txt为目标文件，之后的内容对比，如果前面为空格表示两文件都有该内容，-表示第一个文件减去该内容，+表示第一个文件同时加上该内容后与目标文件相同

```
$ diff -u a.txt b.txt
--- a.txt      2020-04-17 20:14:23.551025000 +0800
+++ b.txt      2020-04-17 20:15:54.460591800 +0800
@@ -1,3 +1,3 @@
 hello world
-hello C
-hello C++
+hello C#
+hello JS
```

`git diff` 比较暂存区与工作区中的文件的差别，其中工作区文件为目标文件

`git diff commit_id` 比较提交与工作区之间的差别，其中工作区文件为目标文件

`git diff HEAD` 比较最新提交与工作区之间的差别，其中工作区文件为目标文件



`git diff --cached` 比较最新提交与暂存区之间的差别，其中暂存区为目标文件

`git diff --cached commit_id` 比较提交与暂存区之间的差别，其中暂存区为目标文件

建立SSH连接的仓库（项目级别）

建立仓库连接时记得选择SSH通道，如果选择https每次提交都需要重新登陆，比较麻烦

Quick setup — if you've done this kind of thing before

 Set up in Desktop or ☐ HTTPS ☒ SSH `git@github.com:ZxyGed/Git_Learning.git` 

生成SSH密钥，首先 `cd ~/.ssh` 进入文件夹，而后输入命令 `ssh-keygen`，全部回车即可，其中id_rsa为私钥，pub文件为公钥

在GitHub对应仓库点击 settings-deploy keys，将pub文件的内容粘贴进去即可，注意勾选ALLOW ADD ACCESS

此时可以使用 `git remote show origin` 查看对应信息

git与远程仓库

`git remote add origin ~/.git` 连接远程仓库，其中origin代表后面~.git的名字

`git push -u origin master` -u表示将本地master与远程origin建立关联，此后便可以直接运行 `git push` 实现本地推送

`git push origin src:dst` git push的完整写法，在没有建立两仓库关联的情况下，可以实现（不同名仓库的内容推送）

`git pull origin src:dst` git pull的完整写法，但是需要注意，push指的是本地：远程，pull指的是远程：本地

`git remote show` 显示与本地仓库关联的所有远程仓库（比如origin）

`git remote show origin` 显示远程仓库origin的详细信息

当push.default设置为matching，push会将本地分支仓库推送到远程对应的分支，当被设置为simple，则会将当前分支内容推送到pull是拉取的分支上面

`git clone ~/.git file_name` 将仓库拷贝到本地对应文件，若该文件夹不存在，则会新建

`git pull` 是 `git fetch` 与 `git merge origin/master` 的结合事实上，本地有远程分支的一个镜像origin，通过fetch使镜像与远程分支同步，通过merge合并镜像与本地分支

git图形化操作

gitk 呈现git提交记录图，同时能显示每一次提交的文件diff信息

`git config --global gui.encoding utf-8` 解决gitk中文乱码的问题

`git gui` 显示GUI操作界面（还是命令行方便，但是GUI好看）

git指令简化

`git config --global alias.br branch` 给branch起别名为br

`git config --global alias.ui '!gitk'` 给gitk起别名为ui，即给别的命令起别名时需要加感叹号

`git config --global --edit` 编辑.gitconfig文件（可直接在里面批量设置别名）

git refspec（分支对应规则）

表示引用规则，即本地分支与远程分支的对应规则

`git push --set-upstream origin develop`（在本地develop分支上），在远程新建一个develop分支，并将两分支相关联

`git checkout -b develop origin/develop` 本地新建一个develop分支并将其与远程develop分支相关联，并切换到develop分支

`git checkout --track origin/test` 含义同上，本地新建一个test分支来追踪远程test分支，并切换到test分支

`git push origin :develop` 删除远程develop分支

`git push origin -delete develop` 含义同上，删除远程develop分支

git gc（垃圾回收）

`git gc` 将所有分支的信息以及标签文件都统一收集到 packed-refs 文件中并对对象进行压缩

git相关知识

对于user.name user.email具有三个方法进行设置

1. /etc/gitconfig（几乎不会使用） `git config --system`
2. ~/.gitconfig（很常用） `git config --global`
3. .git/config 文件中针对特定项目的 `git config --local`

具体增删改查用户名参考命令行提示

.gitignore 用于忽略不需要添加到版本库的文件，.gitignore一般用于忽略项目的配置文件，如库文件等，其本身需要被提交到版本库。其中内容可以使用通配符，如

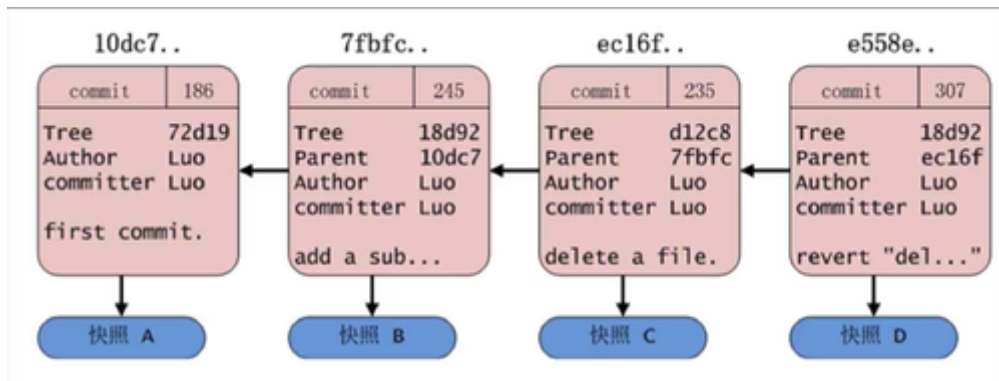
- 忽略所有txt文件可写为 *.txt
- 若要除test.txt外忽略，使用 *.txt !test.txt，！表示除外
- /TODO 表示忽略根目录下的TODO文件
- build/ 表示忽略build/目录下的所有文件
- doc/*.txt 表示忽略doc/目录下所有txt文件
- **/test.txt 表示忽略所有目录下的test文件

基于git分支的开发模型

- develop分支（频繁变化的分支）
- test分支（供测试与产品等人员使用的一个分支，变化不是特别频繁）
- master分支（生产发布分支，变化非常不频繁）
- bugfix(hotfix)（生产系统中出现了紧急bug，用于紧急修复的分支）

对分支的理解

提交的记录通过commit ID形成树结构，注意parent条目的值，即为前一次的ID



HEAD指向当前分支，分支指向最新的一次提交（commitID）

从分支合并到主分支可能会发生合并冲突（主分支合并到从分支不会），如果存在冲突，修改完对应文件后需要重新add、commit并merge