# 全局配置git信息

1. git config --global user.name test # 全局配置信息
2. git config --global user.email test@qq.com # 全局配置信息
3. git config --global color.ui true # 全局配置信息
4. git config --list # 查看配置信息
5. git config --unset --global user.name # 删除全局配置的信息
6. git remote show origin # 查看远程仓库基本信息

# 移动仓库

# 1.先克隆老项目的镜像

git clone --mirror old.git # (old.git 为老项目的git地址)

# 2.进入老项目的目录

cd old.git

# 3.移除老项目的地址替换成新项目

git remote set-url --push origin new.git # (new.git 为新项目的git地址)

# 4.将镜像推到远程

git push --mirror # 这一步需要输入新的git的账号和密码

# 移除仓库

1. rm -rf .git
2. git init

# 初始化

1. git init

# 克隆代码

1. git clone http://test.com # 克隆代码
2. git clone http://test.com newDev # 克隆代码到 newDev 文件夹下
3. git clone -b old http://test.com # 克隆分支

# 代码克隆后配置项目级个人信息（提交是默认带上）

1. git config user.name "Mazer"
2. git config user.email ["2920140779@qq.com"](mailto:\"2920140779@qq.com\")

# 获取最新远程文件

* git pull
* git pull --rebase origin test # 获取其他分支并到本地
* git pull origin dev # pull 对应分支

1. git fetch # 提取远程 origin/master 分支的更新
2. git status # 查看是否有更新
3. git merge origin/master # 合并更新到本地分支

# 拉取远程分支

1. git fetch origin dev # 将远程 dev 分支更新到本地
2. git checkout -b dev origin/dev # 将远程分支拉取到本地新建的分支

pull的问题：

git pull 的问题是它把过程的细节都隐藏了起来，以至于你不用去了解 git 中各种类型分支的区别和使用方法

pull = fetch + merge，而将下载（fetch）和合并（merge）放到一个命令里的另外一个弊端是，你的本地工作目录在未经确认的情况下就会被远程分支更新

在实际使用中，git fetch 更安全一些，因为在 merge 前，我们可以查看更新情况，然后再决定是否合并

# 上传

1. git status # 查看当前工作目录状态
2. git add . # 将所有文件都添加到暂存区
3. git add 文件夹/ # 添加某个文件夹及内容
4. git add \*.文件类型 # 添加目录中所有此文件类型的文件
5. git commit -m 'test' # 添加描述信息
6. git commit -am "test" # 简写 add . 和 添加描述

# 首次上传

1. git commit -am "test" # 提交并添加描述信息
2. git remote add origin https://test.com # 添加远程地址
3. git remote -v # 查看远程地址
4. git push -u origin master # 跟踪上传 master 分支到远程版本库
5. git push -u origin test:master # 本地库（test）和远程库（master）名称不一致的上传
6. git push origin dev # 不跟踪上传 dev 分支到远程版本库
7. git push origin test:dev # 本地库（test）和远程库（dev）名称不一致的上传

# 以后上传

1. git commit -am "test"
2. git push
3. git push origin dev # 提交某分支

# 部分提交

1. git add .
2. git commit src/path -m "test"
3. git push

# 新建-查看-切换-删除分支

1. git branch dev # 新建 dev 分支
2. git push --set-upstream origin dev # 新建远程分支 dev（先 checkout 到需要上传的分支，之后再新建）
3. git branch # 查看所有分支
4. git branch -r # 查看远程分支
5. git checkout old # 切换分支
6. git checkout -b dev # 切换（如果不存在并新建）分支
7. git checkout other # 先切换到别的分支
8. git branch -d dev # 删除本地分支
9. git branch -D dev # 强制删除本地分支
10. git push origin --delete dev # 删除远程分支（本地分支不会删除）
11. git branch -m dev newDev # 重命名分支为 newDev

# 远程分支重命名

1. # 重命名远程分支对应的本地分支

git branch -m old-local-branch-name new-local-branch-name

1. # 删除远程分支

git push origin :old-local-branch-name

1. # 上传新命名的本地分支

git push origin new-local-branch-name

# 合并分支

合并分支全部内容：

1. git checkout master # 先切换到要合并的分支
2. git merge dev # 将 dev 合并到当前 master 分支（注意冲突问题）

合并分支部分内容：

# path 可以是多个，中间空格隔开

1. git checkout source\_branch <path>...

# 本地仓库

1. git stash save "save message" # 执行存储时，添加备注，方便查找，只有 git stash 也要可以的，但查找时不方便识别。
2. git stash list # 查看stash了哪些存储
3. git stash show # 显示做了哪些改动，默认show第一个存储,如果要显示其他存贮，后面加stash@{$num}，比如第二个 git stash show stash@{1}
4. git stash show -p # 显示第一个存储的改动，如果想显示其他存存储，命令：git stash show stash@{$num} -p ，比如第二个：git stash show stash@{1} -p
5. git stash apply # 应用某个存储,但不会把存储从存储列表中删除，默认 使用第一个存储,即stash@{0}，如果要使用其他个，git stash apply stash@{$num} ， 比如第二个：git stash apply stash@{1}
6. git stash pop # 命令恢复之前缓存的工作目录，将缓存堆栈中的对应stash删除，并将对应修改应用到当前的工作目录下,默认为第一个stash,即stash@{0}，如果要应用并删除其他stash，命令：git stash pop stash@{$num} ，比如应用并删除第二个：git stash pop stash@{1}
7. git stash drop stash@{$num} # 丢弃stash@{$num}存储，从列表中删除这个存储
8. git stash clear # 删除所有缓存的stash

# 查看提交日志

1. git log # 查看完整日志（q 键退出）
2. git log --oneline 查看简单日志
3. git log --oneline --author="xiao" # 查看肖泽提交的所有日志
4. git log --oneline --grep="index.html" # 查看仅包含 index.html 的修改
5. git log --oneline --before="2018-09-12" # 查看某天（也可以是 `3 days` 之类的）的提交日志
6. git log --oneline --decorate # 查看并美化显示
7. git log --oneline --decorate --all # 查看所有的
8. git log --oneline --decorate --all -10 -graph # 查看所有的，显示十条，并带 点图形效果

# 恢复提交（revert）重置提交（reset）

1. git log --oneline # 查看并获取提交 id
2. git revert ae77607 # revert 前面的提交 id，恢复提交

# 填写 :wq 退出信息添加

# reset 后面跟对应提交的 id

1. git reset --soft # 不会影响工作区和暂存区的东西（重置到 commit 前）
2. git reset --hard # 会将工作区和暂存区充值到指定的状态（重置到文件修改前）
3. git reset --mixed # 默认选项，将暂存区重置到指定地方（重置到 add 前）

# 强行 pull 远程文件并覆盖到本地

1. git reset --hard origin/master
2. git pull

# 代码恢复（恢复reset、stash等）

# 查找历史 ID

1.git reflog

2.git fsck --lost-found

# 对比修改

3.git show ID

# 恢复

4.git reset ID

# 打tag（可以固定版本号）

# 列出已有 tag

git tag

# 新建 tag 默认为当前分支最后的一个 commit 创建的 tag

git tag <tagName>

# 以某一个特定的提交为 tag

git tag -a <tagName> <commitId>

# 加上 -a 参数来创建一个带备注的 tag

git tag -a <tagName> -m "my tag"

# 查看 tag 详细信息

git show <tagName>

# 将 tag 推送到远程

git push origin <tagName> # 推送某个

git push origin --tags # 全推

# 切换到某个 tag，注意，这种切换之后修改提交，将不属于任何分支，并且将无法访问，如果需要修改，推荐用下面的方式切换

git checkout <tagName>

# 检出标签，如果修改，虽然分支中的代码改变了，但是 tag 标记的 commit 还是同一个，标记的代码是不会变的

git checkout -b <branchName> <tagName>

# 删除某个 tag

git tag -d <tagName>

# 远程删除

git push origin :refs/tags/<tagName>