git使用

常用的命令和概念

> 环境配置
>
> 获取Git仓库
> 工作目录、暂存区以及版本库概念
> Git工作目录下文件的两种状态（未跟踪/已跟踪）
> 本地仓库操作
> 远程仓库的使用
> 分支
> 标签

环境配置

当安装Git后首先要做的事情是设置用户名称和email地址。因为每次Git提交都会使用该用户信息

设置用户信息

config:配置

global：全局

```
git config --global user.name "用户名，一般是指定上传者的名字"
git config --globaluser.email "登录时的邮箱'
```

查看已配置的用户信息

```
gitconfig --list
git config user.name
```

git config #查看本机是否配置了个人信息

以上信息，皆保存在:~/.gitconfig文件中（windows下：c:\user\用户名）

获取仓库

要使用Git对我们的代码进行版本控制，首先需要获得Git仓库

获取Git仓库通常有两种方式

1. 在本地初始化一个Git仓库

```
git init        #初始化仓库(在所在文件夹创建一个隐藏的.git目录)
Initialized empty Git repository in D:/git/test_heima01/.git/
(显示以上信息，表示初始化完成)
```

2. 从远程仓库克隆

```
git clone https://gitee.com/f-sir/test_heima_clone.git
（克隆和远程仓库的所有文件）
```

工作目录、暂存区以及版本库概念

版本库:前面看到的.qit隐藏文件夹就是版本库，版本库中存储了很多配置信息、日志信息和文件版本信息等

工作目录(工作区) : 包含.git文件夹的目录就是工作目录，主要用于存放开发的代码

暂存区: .git文件夹中有很多文件，其中有一个**index文件**就是暂存区，也可以叫做stage。暂存区是一个临时保存

Git工作目录下文件的两种状态

这些文件的状态会随着我们执行Git的命令发生变化

- untracked未跟踪(未被纳入版本控制)（新建的文件，是不被跟踪的）
- tracked已跟踪(被纳入版本控制)（一旦创建，包括克隆就已被跟踪）
    - Unmodified未修改状态（已跟踪，未修改时，在命令中文件不可见）
    - Modified已修改状态（修改创建/克隆后的文件）
    - Staged已暂存状态（使用git add命令后，就已暂存）

---

git常用命令

本地仓库的操作

> git status查看文件状态————>也可以使用git status -S使输出信息更加简洁
>
> git add 文件名.后缀：将未跟踪的文件加入暂存区（添加成功后，为绿色提示）
>
> git reset 文件名.后缀：将暂存区的文件取消暂存
>
> git commit -m "" ：将暂存区的文件修改提交到本地仓库（不加-m，会进入linux下的一种编辑器，输入i，可编辑，编辑完成按esc；再输入:wq；即可退出）
>
> git commit -m "": （m：message的简写）提交时的日志信息，字符串形式
>
> git log：查看日志记录（包含作者name和邮箱，及时间，日志信息）

fatal: not a git repository (or any of the parent directories): .git（未进入仓库，没有.git隐藏文件）

> **git status查看文件状态————>也可以使用git status -S使输出信息更加简洁**

最后第二句表示：分支机构是最新的

最后一句：没有添加要提交的更改（使用“git add”和/或“git commit-a”）

> **git add 文件名.后缀：将未跟踪的文件加入暂存区**
>
> git reset 文件名.后缀：将暂存区的文件取消暂存

A：暂存区状态

> **git commit ：将暂存区的文件修改提交到本地仓库（不加-m，会进入linux下的一种编辑器，输入i，可编辑，编辑完成按esc；再输入:wq；即可退出）**
>
> git commit -m "": （m：message的简写）提交时的日志信息，字符串形式
>
> git diff ： 查看文件修改内容

已跟踪，未修改，是不可见的

> **git rm 文件名称.后缀： 删除文件；执行完成后，文件就被删除了，但是将工作区的文件删除，因为没有提交；所以还是存在的**

提交后，也表示删除成功了（删除时，会把删除的文件，加到缓存区）

直接在windows文件夹内删除，和使用命令删除的区别：

windows内删除（删除时，不会把删除的文件加到缓存区）

仍可手动提交，但比较麻烦

将文件添加至忽略列表

> 一般我们总会有些文件无需纳入Git 的管理，也不希望它们总出现在未跟踪文件列表。通常都是些自动生成的文件，比如日志文件或者编译过程中创建的临时文件等。
>
> 在这种情况下， 我们可以在工作目录中创建一个名为 .gitignore的文件(文件名称固定，但是在windows低版本下，不能创建此文件；可以通过命令创建)，列出要忽略的文件模式。
>
> #：代表注释；
>
> *.a：代表所有以.a结尾的文件忽略；
>
> !lib.a：代表lib.a被管理
>
> /TODO：代表TODO文件需要忽略
>
> build/：代表build目录下的文件被忽略
>
> doc/*.txt：代表doc目录下，以txt结尾的文件忽略
>
> ```
> doc/**/*.pdf：代表doc目录及子目录下，以pdf结尾的文件忽略
> ```

所以以class结尾的文件，是读取时被忽略的

> **git log：查看日志记录（包含作者name和邮箱，及时间，日志信息）**
>
> 可以敲回车查看后面的日志，以（end）结尾
>
> 输入q退出界面
>
> git log --pretty=oneline　　//单行显示
>
> git reset --hard HEAD^　　//回退到上一个版本，其中（HEAD^^(上上版本),HEAD~100(往上100个版本))
>
> git reflog　　//查看历史命令

远程仓库的操作

> 查看远程仓库：git remote/git remote -v（更详细的信息）
>
> 添加远程仓库：git remote add
>
> 从远程仓库克隆：git clone
>
> 移除无效的远程仓库：git remote rm 远程仓库名称
>
> 从远程仓库中抓取与拉取：git fetch(不自动合并)/git pull(自动合并)
>
> 推送到远程仓库：git push [remote-name] [branch-name]

> **查看远程仓库**
>
> 如果想查看已经配置的远程仓库服务器，可以运行git remote命令。它会列出指定的每一 个远程服务器的简写。
>
> 如果已经克隆了远程仓库,那么至少应该能看到origin,这是Git克隆的仓库服务器的默认名字
>
> git remote -v：可查看信息
>
> 如果两个地址是一样的，那么说明两个仓库是建立了关系的；可以从远程仓库抓取，也可以从本地推送到远程仓库

git remote show origin：查看指定窗口，更详细的信息

手动创建的本地仓库，是和远程的仓库没关系的(没推送之前)；输入以上信息是没有任何提示的

> **添加远程仓库**
>
> **远程仓库和本次仓库的的名称，可以不一致；建议一样**
>
> 运行git remote add 添加一个新的远程Git仓库，同时指定一个可以引 用的简写
>
> 添加时仓库服务器的名称可以更改，也可以默认origin
>
> 同时本地仓库也可以添加多个远程仓库，并且在远程仓库服务器名称不一致的情况，同一个地址的远程仓库也可以添加多个

> **从远程仓库克隆**
>
> 如果你想获得一份已经存在了的Git仓库的拷贝，这时就要用到git clone命令。Git 克隆的是该Git仓库服务器上的几乎所有数据(包括日志信息、历史记录等)，而不仅仅是复制工作所需要的文件。当你执行git clone命令的时候，默认配置下远程Git仓库中的每一一个文件的每- 一个版本都将被拉取下来。
>
> 克隆仓库的命令格式是：git clone [url]

> **移除无效的远程仓库**
>
> 如果因为一些原因想要**移除一个远程仓库** ，可以使用**git remote rm 远程仓库名称**
>
> 注意:此命令只是从本地移除远程仓库的记录，并不会真正影响到远程仓库

> **从远程仓库中抓取与拉取**

> git fetch：是从远程仓库狱取最新放本到本地仓库，不会目动merge(合并)
>
> git pull：是从远程仓库获取最新版本并merge到本地仓库
>
> 注意:如果当前本地仓库不是从远程仓库克隆，而是本地创建的仓库,并且仓库中存在文件,此时再从远程仓库拉取文件的时候会报错(fatal: refusing to merge unrelated histories)，解决此问题可以在git pull命令后加入参数-- allow-unrelated-histories

> 当手动将本次仓库的所有文件删除后，那么就和对应的远程仓库没有了关系；
>
> 那么就需要使用：git remote add 添加对应的远程仓库
>
> 使用：git remote -v 或者git remote show 查看远程仓库是否建立关系

重新初始化后，和对应的远程仓库建立关系，且查看

在远程仓库中获取到了仓库，这儿本地仓库却不显示(在.git/object目录下)，因为git fetch (origin master因为是默认，所以可选择性写)，并不会merge(合并)，可再通过git merge（合并）仓库服务器名称/分支名称；此处是因为我的远程仓库没有文件，空仓库

使用git pull (origin master因为是默认，所以可选择性写)；和git fetch使用一致

如果初始化本地仓库后，仍然添加了一个文件，那么使用git pull (origin master因为是默认，所以可选择性写)拉取时会报异常，那么就需要以下操作

> 因为此时的远程仓库没有任何文件，所以进行不了操作

1. 重新删除本地仓库所有文件
2. 查看其状态
3. 和远程仓库建立关系
4. 强行拉去远程仓库的文件：git pull origin master --allow-unrelated-histories
5. 如果出现了linux中的编辑器，那么输入：(:wq)退出即可

> ## 推送到远程仓库
>
> 当你想分享你的代码时，可以将其推送到远程仓库。 命令形式: git push [remote-name] [branch- name]

如果修改了远程仓库上，拉取的项目，并且也进行了修改，那么可以执行以下操作

第一种方式：git add 文件名.后缀；　　添加到暂存区后，再使用commit一并提交，两步操作

第二种方式：git commit -a -m "" ；　　先把已修改的添加到暂存区(-a)，然后再提交；一步操作完成

最后使用git push [remote-name] [branch- name]；提交到远程仓库

---

git分支

几乎所有的版本控制系统都以某种形式支持分支。使用分支意味着你可以把你的工作从开发主线上分离开来,以免影响开发主线。Git 的master分支并不是一个特殊分支。 它跟其它分支没有区别。之所以几乎每一个仓库都有master分支,是因为git init命令默认创建它，并且大数人都懒得去改动它。

常用的分支命令:

> 查看分支
>
> 创建分支
>
> 切换分支
>
> 推送至远程仓库分支
>
> 合并分支
>
> 删除分支

查看分支:

branch：分支

```
#列出所有本地分文

$ git branch

#列出所有远程分支（-r：remote简写，远程的）

$ git branch -r

#列出所有本地分支和远程分支

$ git branch -a
```

*：代表在哪个分支下

创建分支：

> 创建本地仓库的分支：git branch 分支名称

本地分支重命名(未推送)：

> git branch -m oldName newName

远程分支重命名 (已经推送远程-假设本地分支和远程对应分支名称相同)

> git branch -m oldName newName：重命名远程分支对应的本地分支

删除远程分支

> git push --delete origin oldName

上传新命名的本地分支

> git push origin newName

把修改后的本地分支与远程分支关联

> git branch --set-upstream-to origin/newName

切换分支：

> git checkout 分支名称

本地分支推送到远程仓库分支：

> git push 远程仓库的服务名称(默认origin) 需要推送的分支名称(默认master)
>
> 在某一个分支下，创建的其他分支，那么其他的分支就和父分支的内容一样

合并分支：

> 在需要的分支下操作：git merge 指定的另一个分支名称
>
> 合并时，如果出现linux编辑器，那么直接:wq完成即可；如果需要编辑，那么**按下i即可编辑**；完成输入:wq即可

有时候合并操作不会如此顺利。如果你在两个不同的分支中，对同一个文件的同一个部分进行了不同的修改, Git 就没办法合并它们，同时会提示文件冲突。此时需要我们打开冲突的文件并修复冲突内容,最后执行git add命令来标识冲突已解决

CONFLICT：冲突

fix：解决

可以看到，以下发生了(合并)冲突

在User.java文件下发生了合并冲突

自动合并失败：解决冲突后，提交结果

打开冲突文件，手动删除，没有的信息；保存

把新添加的文件，推送到远程仓库：

> git push [remote-name] [branch- name] （远程仓库名称 远程仓库分支名称（本地的分支推送到远程的分支，都有追踪关系））
>
> 要死需要添加到多个分支，那么推送语句就需要多写几条

> done：成功
>
> remote：远程
>
> Powered：由.....提供

删除分支：

> git branch -d 分支名称
>
> 删除分支，指的是删除本地分支

不推荐：如果要删除的分支中进行了-些开发动作,此时执行上面的删除命令并不会删除分支（远程的分支和本地的分支，发生变化）,如果坚持要删除此分支，可以将命令中的-d参数改为-D（在当前分支，删除分支是不可行的）

fully：完全，全部

sure ： 确定

删除远程仓库的分支：

> git push [remote-name]（远程仓库的名称） -d [branch- name] （远程仓库的分支名称）

综合练习

需求场景

```
开发某个网站。
为实现某个新的需求，创建一个分支(dev)。
在这个分支上开展工作。
正在此时，你突然接到一个电话说有个很严重的问题需要紧急修补。你将按照如下方式来处理:
切换到你的线上分支(master)。
为这个紧急任务新建一个分支(fix)，并在其中修复它。
在测试通过之后，切换回线上分支(master),然后合并这个修补分支(fix),最后将改动推送到线上分支(master)。
切换回你最初工作的分支上(dev),继续工作。
```

fix：解决；修理

1、先创建指定分支的名称————>git branch dev

2、切换到dev分支————————>git checkout dev

3、在dev分支下，创建一个用于测试的java文件————>在文件夹或编辑器内实现

4、添加到暂存区，且提交到本地仓库———>git add UserDao.java——>git commit UserDao.java

5、再次切换到master分支（此时在dev上创建的文件，在master分支是没有的）——>git checkout master

6、创建修复bug的fix分支————>git branch fix

7、切换到fix分支————————>git checkout fix

8、修复好指定文件的bug后，添加到暂存区；提交到本地仓库

git add Test.java————>git commit Test.java

9、切换到master分支，由于是在fix分支下，修复的代码；修改部分是不可见的—>git checkout master

10、把fix分支，合并到master分支————————> git merge fix

11、把修改后的项目，推送到远程仓库————>git push origin master

12、切换回dev分支；继续工作————————>git checkout dev

git标签

像其他版本控制系统(VCS) 一样, Git 可以给历史中的某一个提交打 上标签,以示重要。比较有代表性的是人们会使用这个功能来标记发布结点(v1.0、 v1.2等) 。标签指的是某个分支某个特定时间点的状态。通过标签,可以很方便的切换到标记时的状态。

> 列出已有的标签
>
> 创建新标签
>
> 将标签推送至远程仓库
>
> 检出标签
>
> 删除标签.

列出已有的标签：

tag：标签

> #列出所有tag：$ git tag
>
> #查看tag信息
>
> $ git show [tag]

创建新标签：

> 创建一个tag：git tag [tagName]

重命名标签：

> git tag new_tag old_tag：创建新标签（旧标签依旧在）
>
> git tag -d old_tag：删除旧标签

将标签推送至远程仓库：

> git push origin --tags：推送本地所有标签
>
> git push [remote] [tag]：作为一个版本，状态则是推送时，项目中现有的所有的文件

检出标签：

> git checkout -b [branch] [tag]：新建一个分支，指向某个tag

切换到某个tag, 从tag切换回当前分支签：

> git checkout tag_name
>
> 但是，这时候 git 可能会提示你当前处于一个“detached HEAD" 状态。(detached 分离的)
>
> 因为 tag 相当于是一个快照，是不能更改它的代码的。
>
> 直接使用命令git checkout test 分支，出现以下错误
>
> error: pathspec 'origin/XXX' did not match any file(s) known to git.
> 项目上有一个分支test，使用git branch -a看不到该远程分支，直接使用命令git checkout test报错如下：
>
> 解决方法是：
>
> 1、执行命令git fetch取回所有分支的更新
>
> 2、执行git branch -a可以看到test分支（已经更新分支信息）
>
> 3、切换分支git checkout test

删除标签.：

> #删除本地tag： git tag -d [tag]
>
> #删除远程tag： git push origin :refs/tags/[tag]

使用ssh协议传输数据

> 1、使用命令ssh-keygen-trsa生成公钥和私钥， 执行完成后在window本地用户.ssh目录C:\Users\用户名.ssh下面生成如下名称的公钥和私钥
>
> 2、复制公钥文件内容至码云服务器

----------git常用操作----------

说明，以下整理来自廖雪峰大神的《git教程》。

各位童鞋要下载git但是网速不给力的，可以从这里下载：https://pan.baidu.com/s/1qYdgtJY

1、安装git

git config --global user.name 'XXX'

git config --global user.email 'XXX'

2、创建本地库

mkidir learngit //自定义文件夹
cd learngit

touch test.md //创建test.md文件
pwd //显示当前目录

3、常用CRT

git init //初始化代码仓库
git add learngit.txt //把所有要提交的文件修改放到暂存区
git commit -m 'add a file' //把暂存区的所有内容提交到当前分支
git status //查看工作区状态
git diff //查看文件修改内容
git log //查看提交历史
git log --pretty=oneline //单行显示

git reset --hard HEAD^ //回退到上一个版本，其中（HEAD^^(上上版本),HEAD~100(往上100个版本))

commit id //(版本号) 可回到指定版本
git reflog //查看历史命令

其中说明【
工作区（Working Directory)
版本库（Repository） #.git
stage(index) 暂存区
master Git自动创建的分支
HEAD 指针

】

git diff HEAD -- //查看工作区和版本库里最新版本的区别
git checkout -- //用版本库的版本替换工作区的版本，无论是工作区的修改还是删除，都可以'一键还原'
git reset HEAD //把暂存区的修改撤销掉，重新放回工作区。
git rm //删除文件，若文件已提交到版本库，不用担心误删，但是只能恢复文件到最新版本

4、创建SSH Key，建立本地Git仓库和GitHub仓库之间的传输的秘钥

ssh-keygen -t rsa -C 'your email' //创建SSH Key
git remote add origin git@github.com:username/repostery.git //关联本地仓库，远程库的名字为origin
//第一次把当前分支master推送到远程，-u参数不但推送，而且将本地的分支和远程的分支关联起来
git push -u origin master
git push origin master //把当前分支master推送到远程
git clone git@github.com:username/repostery.git //从远程库克隆一个到本地库

5、分支
git checkout -b dev //创建并切换分支
#相当于git branch dev 和git checkout dev
git branch //查看当前分支，当前分支前有个*号
git branch //创建分支
git checkout //切换分支
git merge //合并某个分支到当前分支
git branch -d //删除分支
git log --graph //查看分支合并图
git merge --no-ff -m 'message' dev //禁用Fast forward合并dev分支

git stash //隐藏当前工作现场，等恢复后继续工作
git stash list //查看stash记录
git stash apply //仅恢复现场，不删除stash内容
git stash drop //删除stash内容
git stash pop //恢复现场的同时删除stash内容
git branch -D //强行删除某个未合并的分支

//开发新feature最好新建一个分支
git remote //查看远程仓库
git remote -v //查看远程库详细信息

git pull //抓取远程提交
git checkout -b branch-name origin/branch-name //在本地创建和远程分支对应的分支
git branch --set-upstream branch-name origin/branch-name //建立本地分支和远程分支的关联

6、其他---标签

```
git tag v1.0                                  //给当前分支最新的commit打标签
git tag -a v0.1 -m 'version 0.1 released' 3628164        //-a指定标签名，-m指定说明文字
git tag -s  -m 'blabla'                    //可以用PGP签名标签
git tag                                  //查看所有标签
git show v1.0                               //查看标签信息
git tag -d v0.1                             //删除标签
git push origin                          //推送某个标签到远程
git push origin --tags                          //推送所有尚未推送的本地标签
```