

# Git教程

## 一.git和github/gitee/gitlab的区别

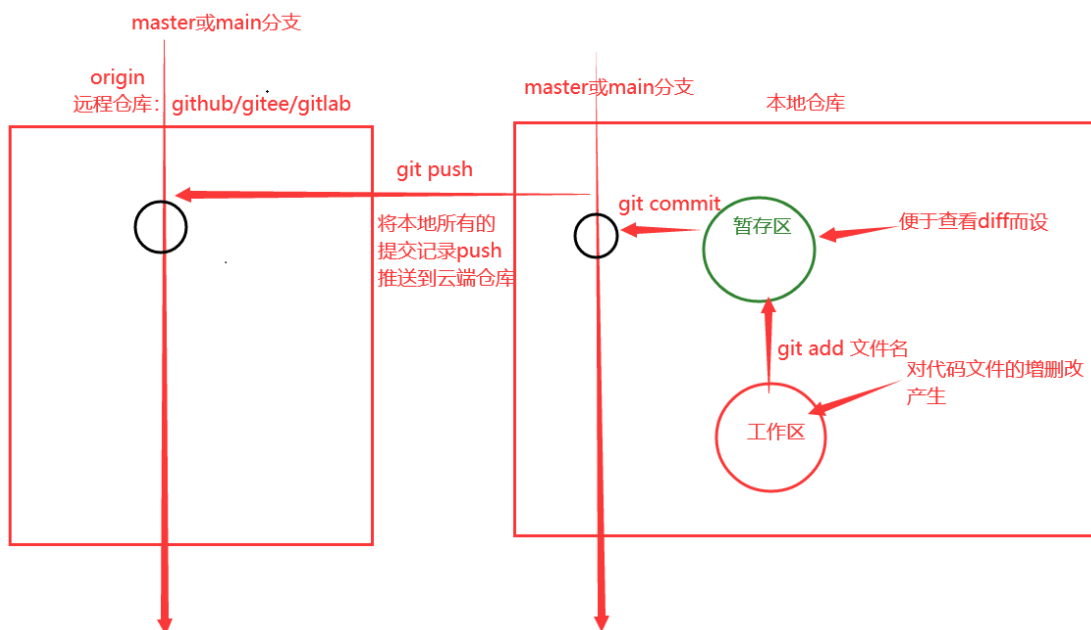
- git是一个工具，用来管理代码文件，以及开发
- github/gitee/gitlab 远程的代码托管仓库

## 二.创建并初始化本地仓库

1. 初始化一个git托管的本地仓库: `git init`
2. 将仓库(目录)中的所有文件添加进git管理: `git add .`
3. 提交结点: `git commit -m "提交信息"`

## 三.工作区，暂存区，commit结点

1. **工作区**顾名思义就是我们日常开发中对文件操作(增删改)后产生的区域，这些修改是游离于分支外的，git能够分辨出哪些文件被我们修改以及我们新增或者删除了哪些文件
2. **暂存区**可以说是一个缓存区，这个和内存映射相关，设计者之所以增加这个区域是为了提高查看修改前后diff的速度
3. **commit结点**就是最终的状态了，其实按照正常的思路来说，有工作区，和提交结点就够了，为了操作效率才增加了要给暂存区



## 四.分支，本地仓库，远程仓库

1. **分支**: 分支可以理解为一个单位，我们所有的操作都是**针对当前所在分支**，无论是本地还是远端仓库都可以有很多条分支，分支由一个一个的提交结点组成，有点类似于一个时间轴
2. **本地仓库**: 本地仓库，简单粗暴的理解它就是一个目录，或者从开发的角度来说是一个项目，存储所有的代码以及配置文件，在上面已经展示了如何创建并初始化一个本地git仓库
3. **远程仓库**: 和本地仓库一一对应，如果想要我们的本地仓库被其他人看见以及让他们来操作和你协作开发，就必须将我们本地的仓库push推送到云端

git仓库是一个形式化的概念：可以认为 **目录 + git = git仓库**，它有备忘录的功能，可以帮我们找回误删的文件

## 五.配置ssh秘钥

上面提到本地仓库和远程仓库，如何去让两者产生关联，也就是——的对应关系，就必须得用到ssh密钥，只有配置了ssh密钥，远端仓库才能识别到我们本地得推送

### 1. 配置author信息，也就是我们开发者的个人信息

```
git config --global user.name "你的名字"
git config --global user.email "你的邮箱"
```

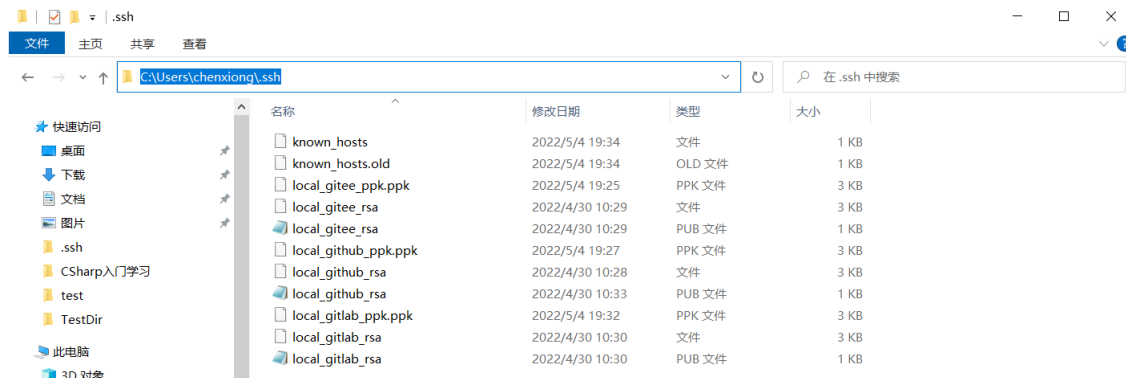
`git config --global -l` 可以查看你配置的信息

### 2. 生成ssh密钥

```
ssh-keygen -t rsa -c "你的邮箱"
```

生成过程中一路回车，不要输入密钥

生成好了后，就可以到如下图所示的地方看见你配置的密钥



C:\Users\chenxiong\.ssh

### 3. 本地的密钥有了，得在远程仓库中注册你的密钥

- 拷贝pub后缀文件中得内容



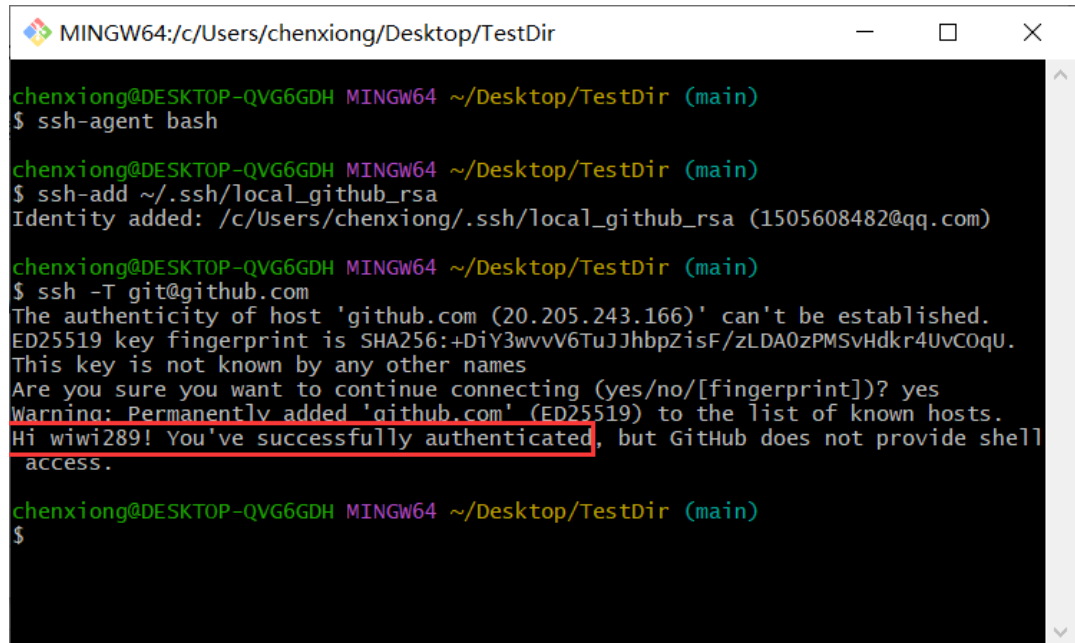
- 本地其实有一个密钥系统，需要添加我们得密钥进去，才能进行本地的推送认证

随便进入要给git仓库，鼠标右键，点击**Git Bash Here**，打开Git操作窗口，这个窗口其实和在CMD终端操作是一样的，不过专用的Git操作窗口功能更强大

按序输入如下命令

```
ssh-agent bash
ssh-add ~/.ssh/你的密钥名称
ssh -T git@github.com
```

出现如下字样，就说明我们已经和远程仓库建立好了联系，后续就可以进行本地仓库的推送以及拉取远程仓库的一系列操作



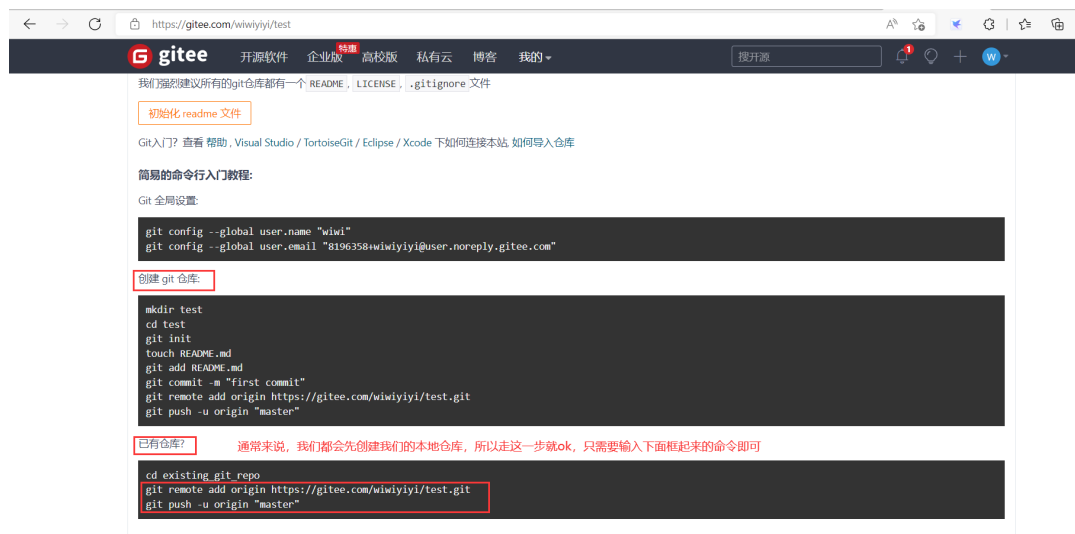
```
MINGW64:/c/Users/chenxiong/Desktop/TestDir
chenxiong@DESKTOP-QVG6GDH MINGW64 ~/Desktop/TestDir (main)
$ ssh-agent bash

chenxiong@DESKTOP-QVG6GDH MINGW64 ~/Desktop/TestDir (main)
$ ssh-add ~/.ssh/local_github_rsa
Identity added: /c/Users/chenxiong/.ssh/local_github_rsa (1505608482@qq.com)

chenxiong@DESKTOP-QVG6GDH MINGW64 ~/Desktop/TestDir (main)
$ ssh -T git@github.com
The authenticity of host 'github.com (20.205.243.166)' can't be established.
ED25519 key fingerprint is SHA256:+DiY3wvV6TuJJhbpZisF/zLDA0zPMSvHdkr4UvCOqU.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'github.com' (ED25519) to the list of known hosts.
Hi wiwi289! You've successfully authenticated, but GitHub does not provide shell
access.

chenxiong@DESKTOP-QVG6GDH MINGW64 ~/Desktop/TestDir (main)
$
```

4. 配置好ssh后，我们就可以按照github/gitee/gitlab给我们的提示进行本地仓库和远程仓库的关联操作



## 六.开发流程，work flow

走完如上流程，我们就可以进行写作开发了，这里需要明确如何进行写作开发，创建分支的规范是什么，以及怎么去看你与小伙伴的工作流

### 1. 分支规范

- 每个仓库都有一个默认分支，要么叫master(gitee)，要么叫main(github/gitlab)，这个主干分支是统领全局，管理整个项目的，我们不能直接在这个分支上进行需求开发，否则会污染分支，甚至造成混乱
- 所以我们每个开发人员都需要在自己的独立分支上进行需求开发

`git checkout -b "你的分支名"`，你的分支名，按照公司的要求就是feature\_需求名称\_你的名称全拼，例如**feature\_add\_a\_pack\_of\_cards\_chenxiong**，我们平时开发可以按照这样的规范，但是必要性不是很大，就简单**dev\_需求名称\_chenxiong**即可，或者需求名称都不要

- 此后我们就在自己的分支开发需求即可，开发完了后，肯定需要将我们的修改(Commit节点)弄到主干分支上去，这个过程叫**合回**，或者Merge，常用的方式就是Merge，不要rebase

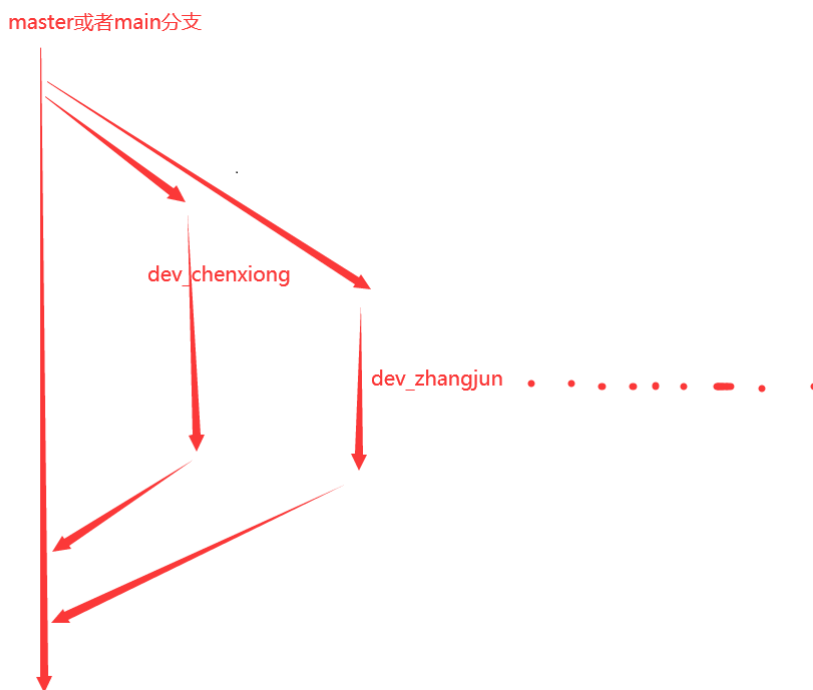
```
// 首先切换回主干分支
git checkout main
// 在主干分支合并的分支
git merge dev_chenxiong
```

如此一来，你的所有提交结点就合并到主干分支上了，这里再简单提一嘴，合并分支有两种方式：

1. Merge
2. Rebase

区别就是，Merge更能体现出工作流，也就是你期间做的所有提交，而Rebase比较特别，我基本上不用，官方说法就是可以**压缩结点**，让整体开发呈**直线型**，这显然不是我们想要的

2. 说完每个程序员的开发规范后，还得整体把握开发规模，也叫Work Flow：工作流，下面通过一张图来展示常见的工作流



这样的图比较清晰明了，但是在Git可视化工具，比如Source Tree中看到的工作流不会这种切换过程，上述就是简单的工作流展示，平时开发就这样

## 七.怎么解冲突

冲突在日常开发中太常见了，并且比较难的就是解冲突，并且要解得优雅，不能出现任何失误，特别是大公司的项目，必须慎之又慎

### 1. 什么是Git冲突

简言之就是你做的修改和其他人做的修改冲突了，比如你们修改了同一行代码，这个时候，Merge是失败的，你得先解决你和他的冲突才能合并成功

### 2. 为什么会产生Git冲突

Git虽然强大，但依然是一个电脑程序，基本原理就是根据**行号**来识别你的增删改操作，并不能很智能地识别你和其他人地实际修改内容，所以Git没有办法帮你继续合并下去，就类似于抛异常一样，提示你，让你手动来决定这行代码该怎么写，删除或是保留

### 3. 如何解冲突

现在很多主流地IDE都集成了Git可视化解冲突界面，非常方便，动动小手指就能解决，比如Android Studio，那么如果你用到IDE没有提供这种功能，比较推荐地可视化工具是p4Merge，当你合并分支出现冲突时，输入如下命令就能打开可视化操作界面(当面展示怎么用)

```
git mergetool
```

随后就在弹出来地可视化窗口进行解冲突即可，很nice

## 八.技巧总结

其实上述目前来说是纯命令地方式来操作Git，时间久了就觉得很鸡肋，并且命令容易敲错，太多了，也记不住，下面推荐使用Source Tree，完成日常的开发不成问题

下面还想说一下，我常用的一些工作技巧

1. 场景一：如果你当前已经做了一个需求的一半，但是还没做完，你的Mentor让你优先做另一个需求，这个时候，你得切新的分支，那之前做的内容怎么保存

解决方案：常人都会将之前做的东西作为一个Commit结点，提交到分支上去，虽然可以解决问题，但是不好，因为分支中的每一个结点都代表一个任务，它们是一个个独立的单元，一个未完成的任务又怎么能成为一个独立的结点，所以我选择**暂存**

```
// 首先将你的所有修改加到暂存区
// 然后输入以下命令，将任务打包成一个特殊的stash节点，悬浮在开发分支上
git stash
```

stash是一个栈结构，如果你有很多个这样未完成的任务，多次stash，依次压入栈，后进先出，下面就是与之搭配的常用的操作

```
git stash pop: 弹出最新的一个保存，并删除暂存区中的记录

git stash apply: 恢复最新的一个保存，但并不删除暂存区中的记录

git stash drop stash{i}: 删除暂存区中的一个记录

git stash apply stash{i}: 恢复指定的保存记录
```

这个命令还有其他好处，比如你当前做的所有修改，你发现没有任何意义，直接把它压入栈，然后drop扔掉即可，一键删除，非常棒，就不用通过 `git restore 文件名` 的方式去丢弃你做的修改，太麻烦了，也很鸡肋

2. 场景二：提交结点的回滚，如果你发现当前任务已经commit了，但后续发现还遗留了一点操作没弄，你又不希望遗留的那点东西成为一个新的结点，它本就属于上一个结点，所以这时候就可以**软回滚**

```
git reset --soft HEAD^
```

这个操作就是废弃你上一次的提交结点，重新来

还有一个是**硬回滚**，这个也很有用，它可以在同一条分支中的结点间自由切换，这里还得介绍一下 commit 结点哈希值，每一个提交结点都会有一个唯一的哈希值与之对应，那么我们就可以通过操作哈希值来切换结点

```
git reset --hard 结点哈希值
```

此后你就定位到了目的哈希值的提交结点，这里有一个小问题，硬回滚后 `git log` 看不到这个结点后面的提交记录，不过还有一个命令可以帮助我们找回

```
git reflog
```

它会罗列出我们所有的操作结点，包括提交结点

3. 场景三：分支推送，这一块也很重要，大公司都是有自己的MR开发平台，MR指Merge Request，你得推送你的分支到云端，提MR让你得Mentor Review代码后才能合入，你自己是合不进去得
- 上面在梳理建立本地仓库和远程仓库的关联时，只提到了如何推送到主干分支，下面依次展示如何推送你的分支到远端

```
// 单纯推送你的分支到云端
git push origin 你的本地分支名:你的本地分支名
// 将你的本地分支推送合入到云端仓库的一个开发分支
git push origin 你的本地分支名:云端开发分支名
// 如果你的本地分支有了新的修改，比如你回滚了，那这个时候你再推是不成功的
git push origin 你的本地分支名:你的本地分支名 --force // 使用强制推送
// 基本格式
git push origin srcBranch:destBranch
```

那么拉取云端分支到本地也是同理

```
git pull origin 远端分支名:本地分支名 // 其实这里远端分支名和拉取后的本地分支名相同即可
// 基本格式
git pull origin srcBranch:destBranch
```

基本格式都很类似，明确好本地分支名和云端分支名即可

4. 场景四：建立本地分支和云端分支的关联，有的时候你会不清楚自己的分支是否落后云端的开发分支，那么建立一个关联就是最简单的方式

```
git push --set-upstream origin 分支名，将本地分支推送到远端仓库并建立映射关系
git branch --unset-upstream //解除关联
```

5. 场景五：基于远端分支创建一个与之对应的本地分支，这个也比较常见，有的时候你可能因为电脑问题，先将本地分支推送到云端，然后换一台电脑还想继续对之前推送到云端的分支进行操作

```
git checkout -b 本地分支名 origin/云端分支名，基于远程分支建立本地分支
```

这个也是协作开发常用的技巧，比如你的队友发生了他无法解决的bug，需要你帮忙看看，你就可以以这种方式把他的分支拉下来，解决问题

6. 场景六：删除分支，删除分支分为删除本地分支和云端分支

```
// 删除本地分支  
git branch -D 分支名  
// 删除云端分支  
git push origin --delete 云端分支名
```