**GIT安装配置**

|  |  |  |  |
| --- | --- | --- | --- |
| **文档版本** | | | |
| 版本编号 | 版本日期 | 作者 | 修改描述 |
| 0.1 | 2020-04-20 | 陈迪 | 初稿 |
|  |  |  |  |
|  |  |  |  |

目录

[1. GIT 4](#_Toc9307)

[1.1.1 GIT是什么 4](#_Toc22713)

[1.2 GIT和SVN的区别 4](#_Toc1315)

[1.3 GIT工作流程 5](#_Toc1797)

[1.4 GIT代码托管中心 7](#_Toc236)

[1.4.1 代码托管中心协作方式 7](#_Toc5564)

[1.5 GIT安装 8](#_Toc3119)

[1.6 GIT命令操作 10](#_Toc5343)

[1.6.1 GIT 4大区域命令关系图 10](#_Toc22186)

[1.6.2 Git设置签名 10](#_Toc24041)

[1.6.3 GIT本地仓库命令 11](#_Toc439)

[1.6.4 GIT远程仓库操作命令 24](#_Toc19289)

[1.6.5 Git远程连接异常 29](#_Toc27205)

[1.7 GIT基本原理 31](#_Toc2361)

[1.7.1 GIT保存版本的机制 31](#_Toc11167)

[1.7.2 GIT分支管理机制 33](#_Toc12119)

[1.7.3 Git暂存区的意义 36](#_Toc26982)

[1.8 Git Flow工作流 42](#_Toc6598)

[1.8.1 GitFlow是什么 42](#_Toc8812)

[1.9 Git图形化操作 46](#_Toc23261)

[1.9.1 Eclipse集成Git 46](#_Toc6990)

[1.9.2 Eclipse的Git操作 48](#_Toc14812)

[2 .SourceTree 72](#_Toc9987)

[2.1 SourceTree是什么 72](#_Toc3016)

[2.2 SourceTree基本使用 72](#_Toc1210)

[2.2.1 SourceTree配置忽略文件 78](#_Toc24658)

[2.2.2 SourceTree回滚提交 79](#_Toc1459)

[2.2.3 SourceTree分支管理 85](#_Toc2749)

[2.2.4 SourceTree连接异常 88](#_Toc777)

[2.3 SourceTree GitFlow工作流使用 89](#_Toc3867)

# GIT

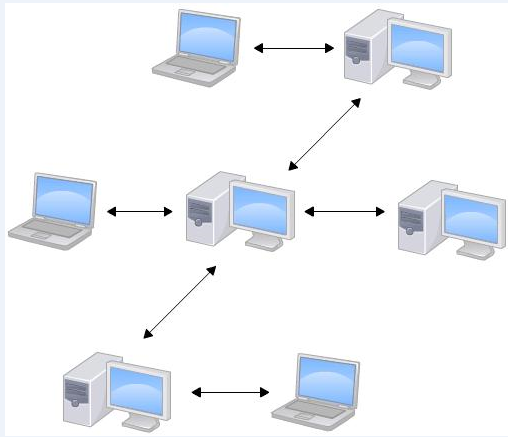
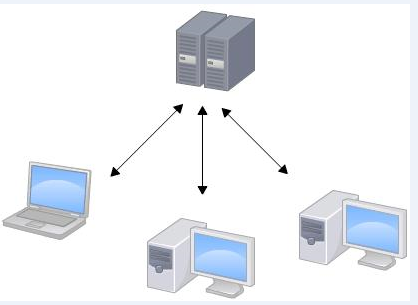
### GIT是什么

* Git 是Linux Torvalds为了帮助管理Linux内核源码而开发的1个开源的分布式版本控制系统;Git可高效地处理任何小型、中型、大型项目;Git占用资源小,性能好;
* Git与常用的版本控制工具SVN,CVS等不同,Git采用分布式概念,不必服务端软件支持;

## GIT和SVN的区别

* Git是分布式的,svn不是,这是最核心区别
* Git把内容按元数据方式存储,而svn是按文件
  + Git元数据都存在.git目录,它是1个克隆版的版本库,拥有中央版本库的所有东西如:标签、分支、版本记录等
  + Svn从中央版本库检出的仅是文件,没有仓库的概念
* Git分支和SVN分支不同,分支管理是Git的另一个核心特点
* Git不依赖网络,本地有完整的版本库,可在本地做历史版本的回滚,而svn依赖网络,本地没有完整版本库,不能本地回滚历史版本
* Git若出现单点故障可从任意1个节点恢复完整历史数据,而svn则不能快速恢复历史数据

Git分布式图 svn集中式图

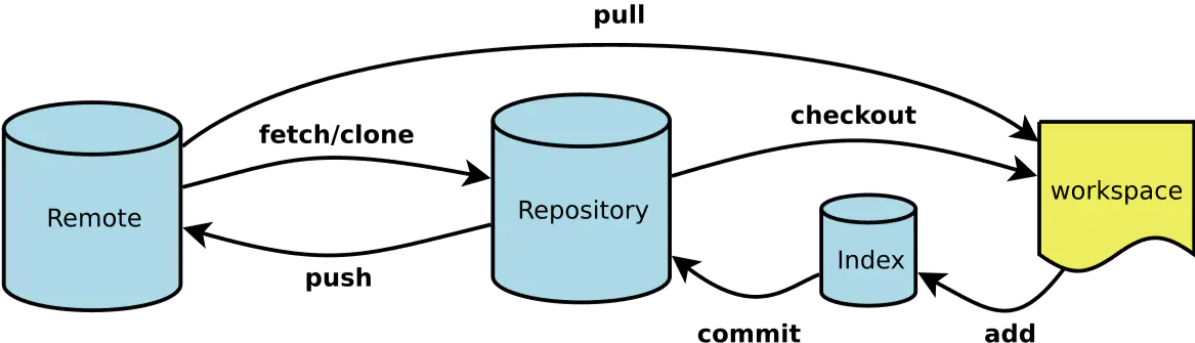
 

## GIT工作流程

* 名词解释
  + Workspace(工作区): 直接编辑文件的地方,平常的开发是拷贝远程库中的1个分支,基于该分支开发,在开发过程中就是对工作区的操作
  + Stage/index(暂存区): .git目录下的index文件,保存待提交的文件列表,标记了工作区中哪些文件是被git管理
  + Repository(仓库区/本地仓库): 存储从暂存区提交过来的文件,之后通过git push同步到远程库
  + Remote Repository(远程仓库): 远程代码共享区域,可能被分布在多个地点的本地仓库所修改,如代码托管平台:Github,GitLab
* Git 4大区域分布图



* Git 4大区域关系图

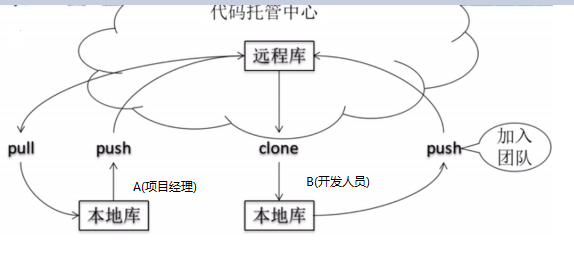


## **GIT代码托管中心**

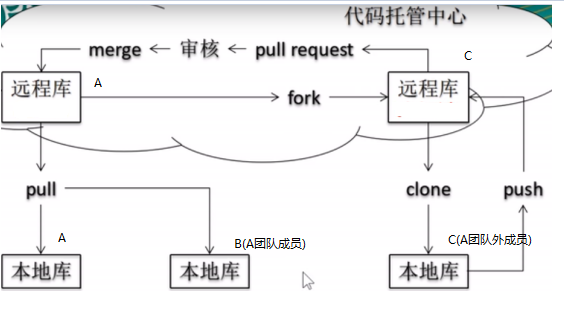
* 代码托管中心任务:维护远程仓库
  + 局域网环境: 自行搭建Gitlab服务器
  + 外网环境: Github+码云

### 代码托管中心协作方式

* 团队内协作： 在A本地有代码先push到远程库,B克隆远程库代码到B本地库,B开发完成后push前要先申请加入到A团队,A将B加入到团队后B才能push到远程,最后A拉取B push上来的代码到A本地

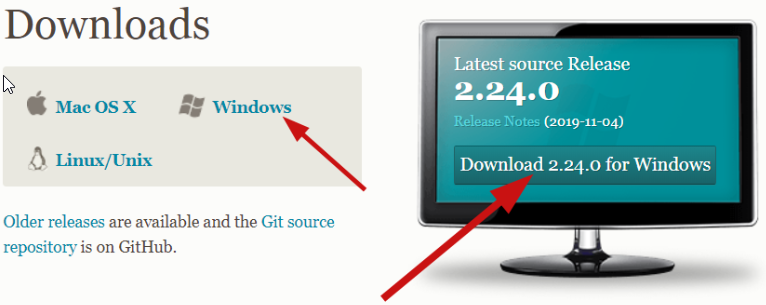


* 跨团队协作: 团队外开发人员B先fork A团队建的远程代码,开发完成后B发起pull request请求,要求远程库拉取B远程代码,经A审核后merge到A的远程库



## GIT安装

* Git安装步骤
  + 官网获取GIT安装程序:[https://git-scm.com/downloads](https://git-scm.com/downloads%EF%BC%8C%E5%A6%82%E4%B8%8B%E5%9B%BE%EF%BC%9A)



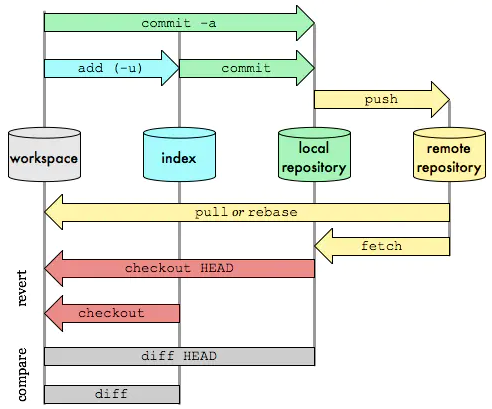


* 运行git安装包,一直下一步(采用默认配置),安装成功:桌面右键会看到有git bash或开始菜单栏搜git有git bash



## GIT命令操作

### GIT 4大区域命令关系图



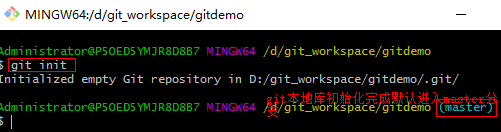
### **Git设置签名**

* 签名作用
  + 区分不同开发人员的身份,明确本次是谁的提交
  + 签名与远程库的登录账号密码无关
* 签名分类
  + 仓库级别: 仅在当前本地库范围内有效(若本地库变更则要重新设置新的签名)
    - git config user.name charles
    - git config user.email charlse@qq.com
    - 签名保存位置: ./.git/config
  + 系统用户级别(推荐): 登录当前操作系统的用户有效
    - git config **--globa**l user.name charles
    - git config **--global** user.email charlse@qq.com (用户名和邮箱均可不存在,每次提交代码签名信息和代码会一起提交到远程)
    - git config --global --list: 查global签名
    - 签名保存位置: ~/.gitconfig

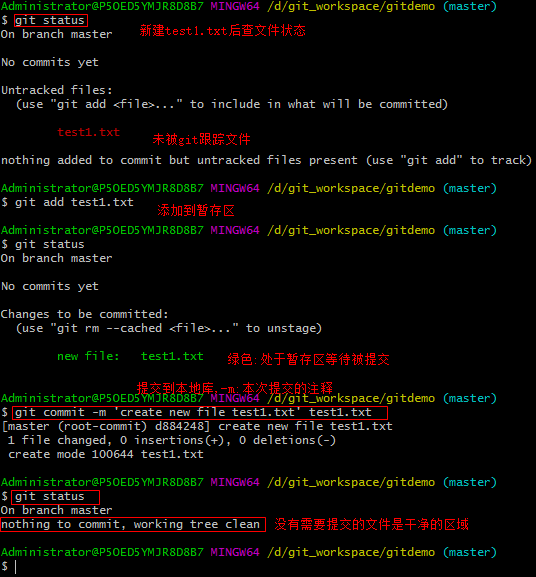
### GIT本地仓库命令

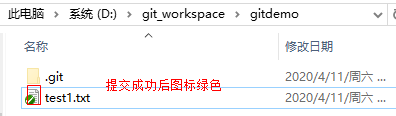
* 初始化本地仓库
* 新建gitdemo目录作为本地仓库D:\git\_workspace\gitdemo
* 进入gitdemo目录右键选git bash,并执行git init,看到.git目录生成则成功



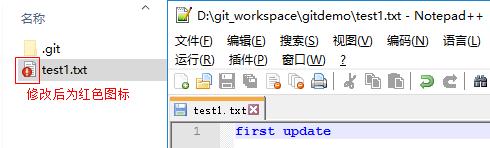


* 新建test1.txt并提交到本地仓库中,由git管理该文件
* git add 文件名: 将test1.txt添加到暂存区
* git commit -m ‘提交注释’: 将test1.txt提交到本地库
* git status: 查看工作区和暂存区的文件状态(是否有未被git跟踪的文件)
* git管理的文件有三种状态:已修改(modified)+已暂存(staged)+已提交(commited)

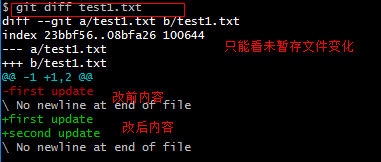




* 修改test1.txt文件内容,添加一行”first update”并提交
* git diff 文件名: 工作区文件修改前后的差异比较
* git diff HEAD^ 文件名: 工作区和版本库某个指定版本进行差异比较

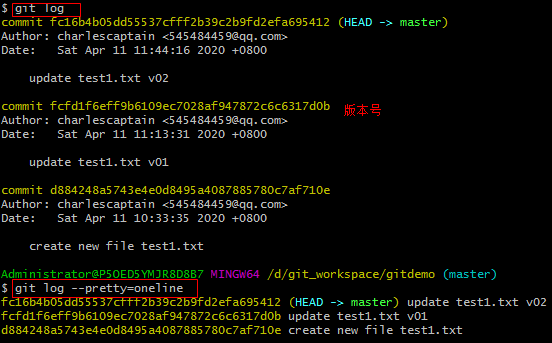




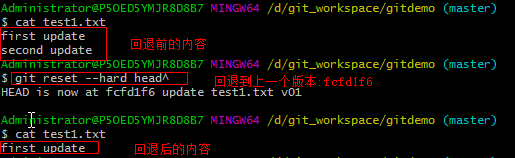


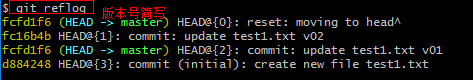
#### Git版本管理

* 查test1.txt文件历史提交记录
* git log: 查指定文件的历史提交记录(只显示基于当前版本的过去的历史记录,不显示未来的记录)
* git log --pretty=oneline: 1行显示历史记录



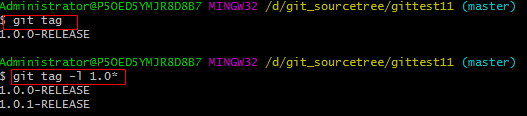
* 版本回退: 将test1.txt回退到上一个提交版本
* git reset --hard head^: 相对于当前版本回退到上一个版本
  + --soft参数(软合并)
    - 在本地库移动HEAD指针(工作区文件内容没变)
  + --mixed参数(混合合并)
    - 在本地库移动HEAD指针
    - 重置暂存区
  + --hard参数(强行合并)(推荐)
    - 在本地库移动HEAD指针
    - 重置暂存区
    - 重置工作区
* git reset --hard head~100: 回退到前第100个版本
* git reset --hard 版本号: 回退到指定版本
* git reflog: 查所有历史提交版本号(会显示回退或前进到指定版本指针需要移动的次数: HEAD@{3},表示需移动3次)



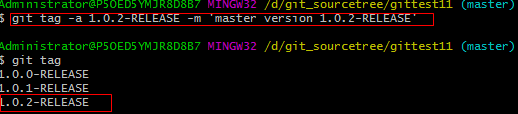


#### **Git标签**

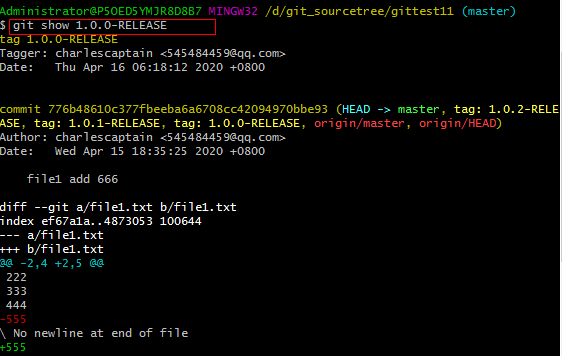
* Git标签的作用: 用标签来记录开发中的关键时刻,如发布版本,重大修改和升级,标签也是版本库的一个快照
* 标签命令使用
  + 列出标签
    - git tag: 列出所有标签
    - git tag -l ‘1.0.\*’: 以特定模式查找标签



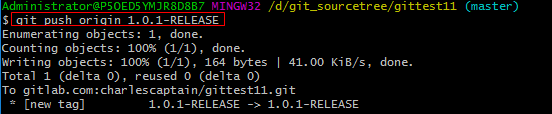
* 创建标签
  + git tag -a 1.0.2-RELEASE -m ‘my version 1.0.2-RELEASE’



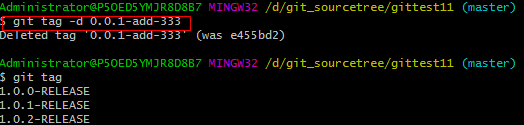
* 查看标签信息
  + git show 1.0.0-RELEASE



* 共享标签
  + git push origin 标签名



* 删除标签
  + 删除本地标签
    - git tag -d 标签名



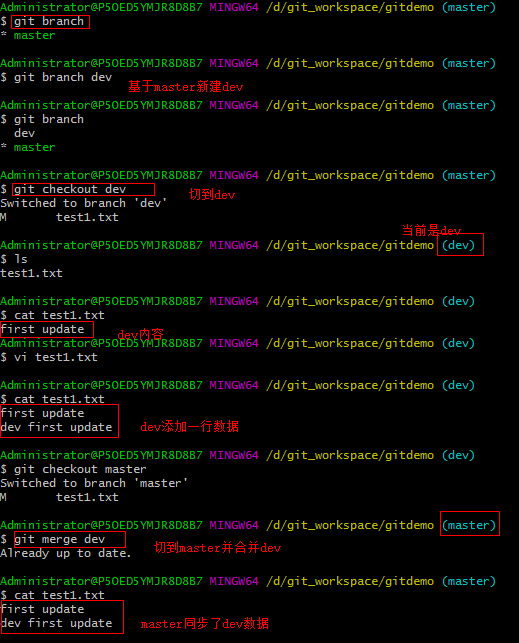
* 删除远程标签
  + git push origin --delete 标签名



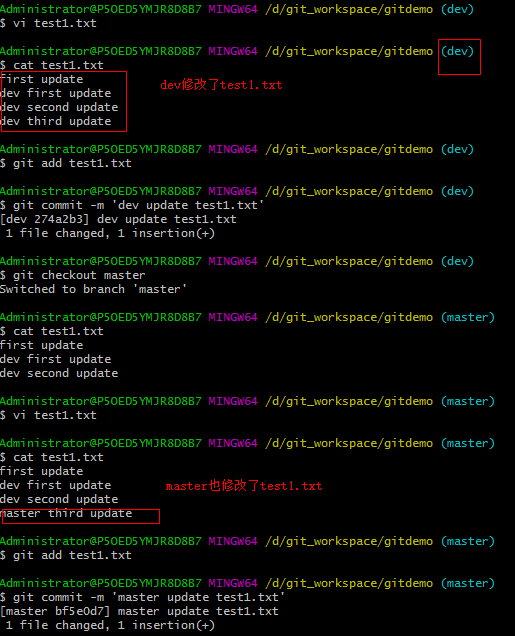
* 基于标签检出新分支,方便获取打标签的版本
  + - git checkout -b 分支名 标签名

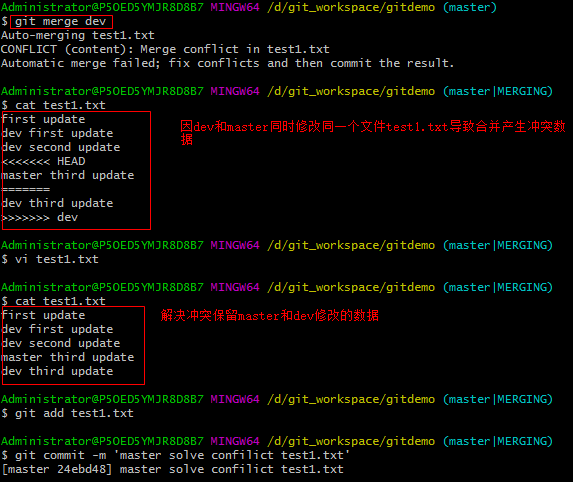
#### Git分支管理

* 分支管理: 允许开发团队在工作过程中使用多条生产线同时推进多个任务(多个功能同时开发),分支间相互隔离,可删除已开发完成的分支
* 分支操作
* git branch -v: 查看当前分支
* git branch 分支名: 创建分支
* git checkout 分支名: 切换分支
* git checkout -b 分支名: 创建并切换到某个分支
* git merge 分支名: 合并某分支到当前分支(注:如master要合并dev,则需先切换到master)
* git branch -d 分支名: 删除分支
* 基于master分支创建dev分支,并将dev分支数据合并到master分支



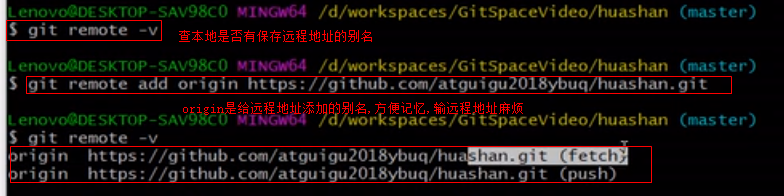
* 解决分支合并冲突: dev中修改test1.txt并提交,切到master修改test1.txt并提交,master合并dev时产生冲突,此时要解决master中test1.txt中的冲突(保留两边的数据),最后提交
* 冲突产生原因:不同分支改了相同的位置,git在合并时不知到底哪个是你想保留的,故提出疑问,让你自己手动选择要保留的内容,从而解决冲突
  + 分支冲突的解决和远程拉取产生冲突是一样的,若本地的修改不是基于远程最新版做的修改则不能推送,必须先拉取远程最新版后解决产生的冲突文件后才能推送
  + 若远程和本地两边版本都变了就要先拉取远程再推送,如此时修改的是同一个文件则拉取会有冲突,若不涉及同一个文件的修改则无冲突





### GIT远程仓库操作命令

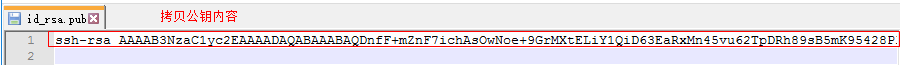
* 本地保存远程库地址
  + git remote -v: 查本地是否有保存远程库的url地址
  + git remote add origin 远程库地址: 将远程库地址保存到本地并起默认别名:origin(在推送前先在本地定义好远程地址,方便推送)

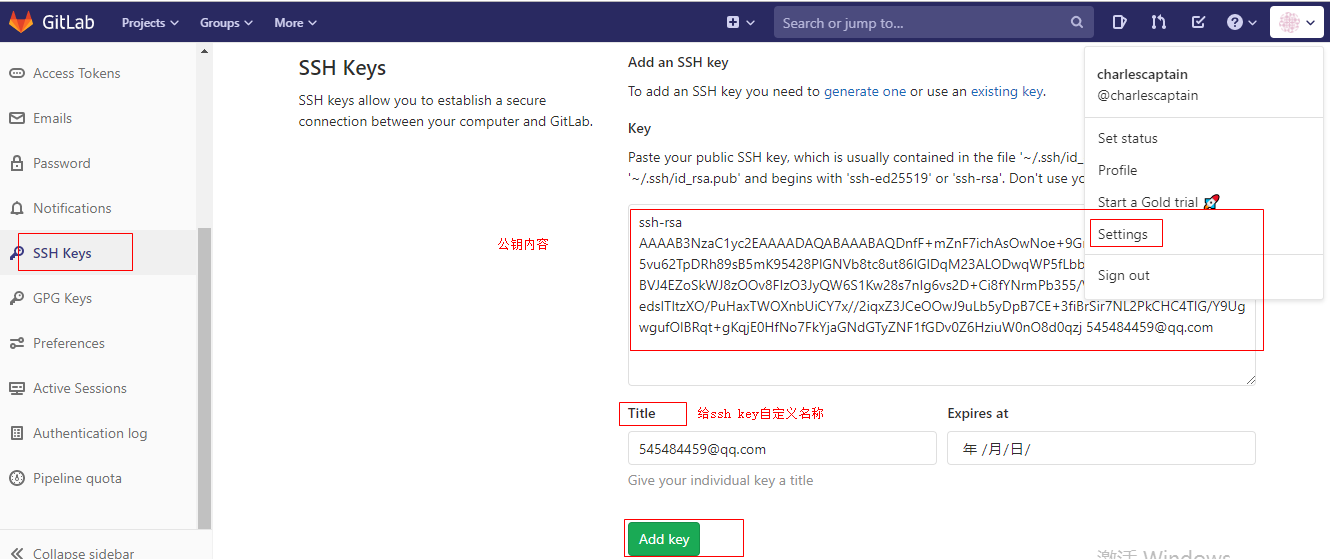


* ssh和http连接方式区别
  + http方式要求创建的是公有项目(public),private和internal不能用http方式连接,而ssh方式在3种模式下均可
  + http每次推送代码都要输密码,而ssh则不用
  + ssh在克隆代码前要先在gitlab中设置好ssh key,且自己必须是该项目的拥有者
* 创建ssh key
* 输入命令ssh-keygen -t rsa -C ‘你的邮箱地址’ ,一直按回车: 会在目录C:\Users\Administrator\.ssh下生成公钥和私钥

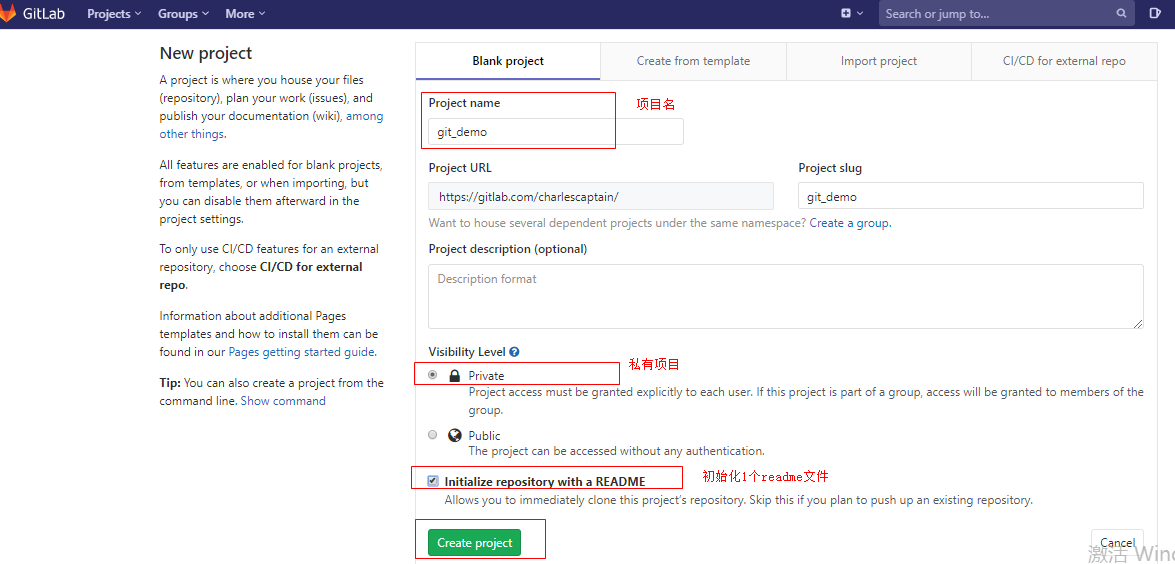


* 为gitlab账号添加ssh key

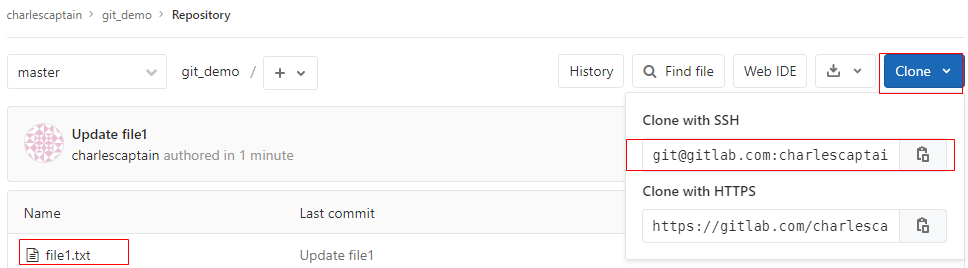




* 在gitlab上新建1个远程仓库并将其克隆到本地仓库
* 新建1个项目git\_demo并添加1个文件:file1.txt



* git clone 远程库地址
  + Clone效果
    - 将git\_demo克隆到本地仓库
    - 自动在本地添加远程库地址别名:origin
    - 自动初始化本地库





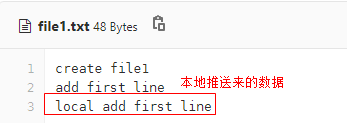
* 拉取远程库的修改
  + git pull origin master: 修改远程仓库git\_demo的file1.txt内容:增加一行“add first line”,并将该文件拉取到本地库





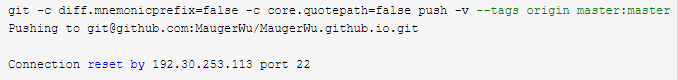
* git fetch和git pull区别
  + git fetch: 获取远程库最新版到本地库的1个临时分支,此时并没合并到本地库(本地库文件还没更新),若想合并再用git merge
  + git pull=git fetch+git merge---获取远程库最新版并合并到本地库
  + 在sourcetree上操作获取后会提示有1个待拉取,需拉取后才能合并到本地库,故直接用拉取更方便
* 推送本地库的修改
  + git push origin master: 在本地库修改file1.txt:添加一行local add first line”,并推送到远程库



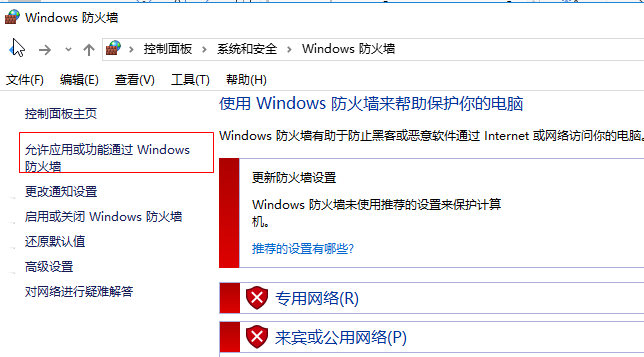


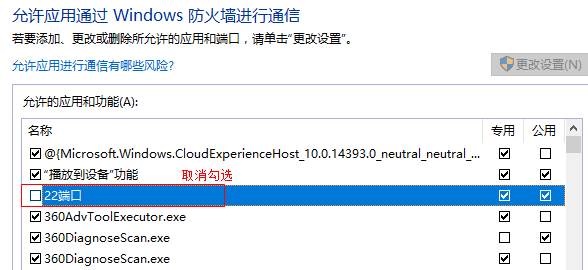
### **Git远程连接异常**

* 若用ssh地址从远程库克隆项目报连接异常如下图



* 两种方式解决:
  + 转用https方式连接,输入github账号密码
    - 取消防火墙阻止端口22上的ssh连接
    - 控制面板--->windows防火墙--->





* 场景: 若一开始是在本地建仓库推送到远程,在给本地库添加远程地址后推送报错如下: 因本地和远程是两个独立的库,无法合并不相关的历史



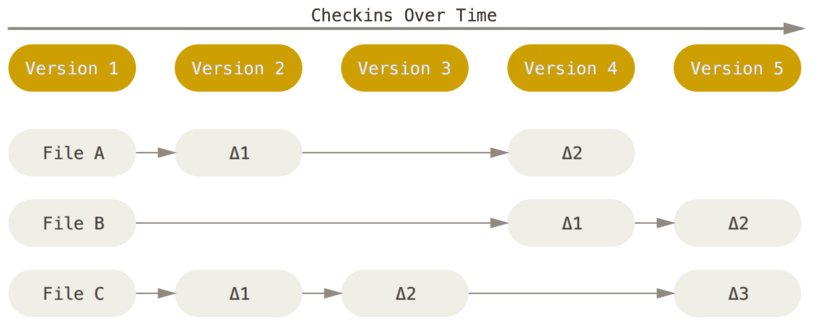
* + 解决: git pull origin master --allow-unrelated-histories
  + 推荐使用git clone从远程拉取工程到本地则不会出现该问题

## GIT基本原理

### **GIT保存版本的机制**

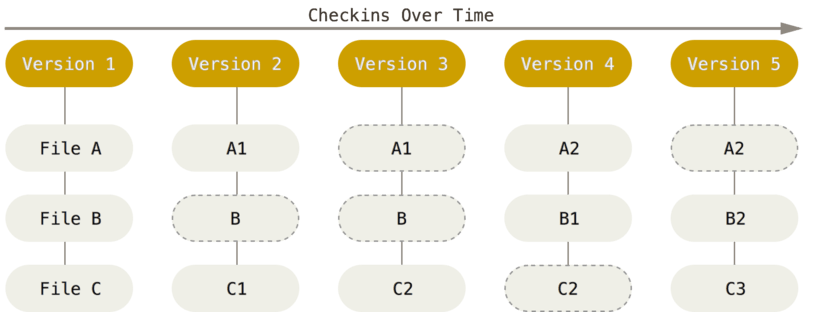
#### Svn文件管理机制

* SVN增量式管理
  + 节约存储空间: 每个版本仅保存增量数据,避免了重复数据的保存,如会把原始文件File C和之后新增的内容A1+A2合并起来作为新版本version3的全量数据

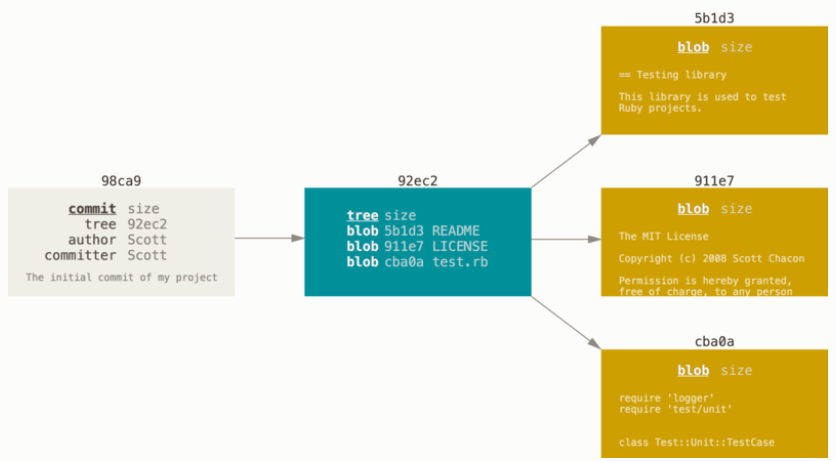


#### **Git文件管理机制**

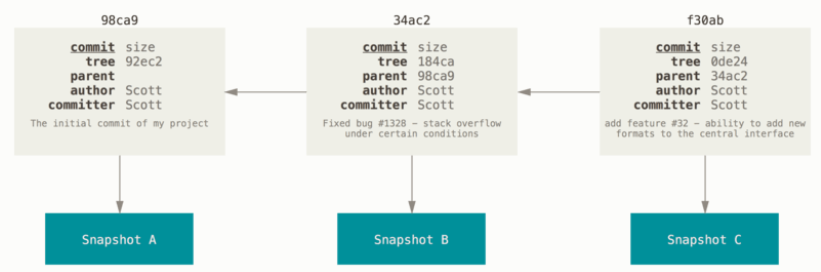
* + Git快照流(全量式管理)
    - 每个版本都形成1个独立的快照,对用户而言每个快照都是一份全量数据,通过文件的快照能恢复到特定时间点的文件状态
    - 没有发生改变的文件只存文件的引用(如version2的B和version3的A1)
    - Svn每个版本只存文件的增量数据,若想拿version3的全量数据需合并version123的所有数据



* Git提交对象
  + 提交时会创建1个提交对象
    - 首次提交产生的提交对象没有指向父对象的指针
    - 普通提交产生的提交对象包含指向父对象的指针
  + 若有3个文件需提交,会先根据每个提交文件内容计算出各自的hash值--->将所有目录下文件的hash值构成1个树对象tree(包含每个具体文件及hash值),树对象自己也有hash值
  + 提交对象有自己的hash值且包含树对象的指针



* 每个版本/快照间是有父子关系的,通过parent建立父子关系形成版本历史记录,一个快照对应一个提交对象

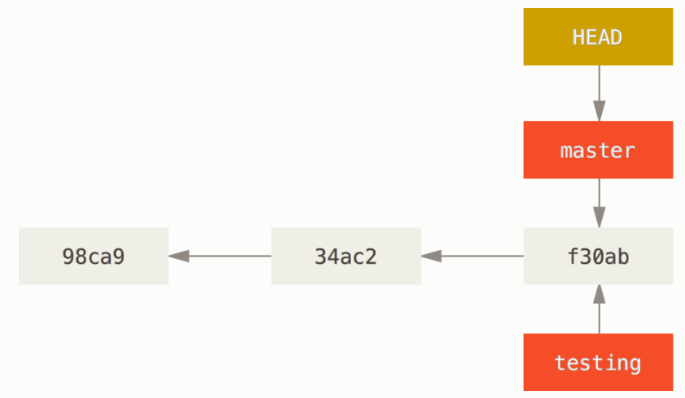


### GIT分支管理机制

* Git分支作用: 方便地进行版本部署和开发

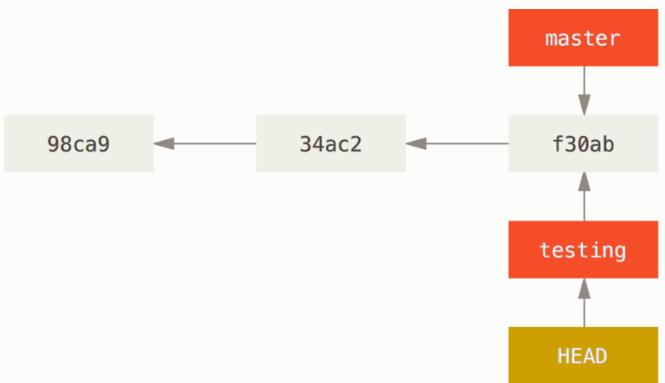
#### 分支的创建

* 分支创建就是新建1个指针testing指向原提交点f30ab(而svn新建分支是做文件的拷贝,耗时)

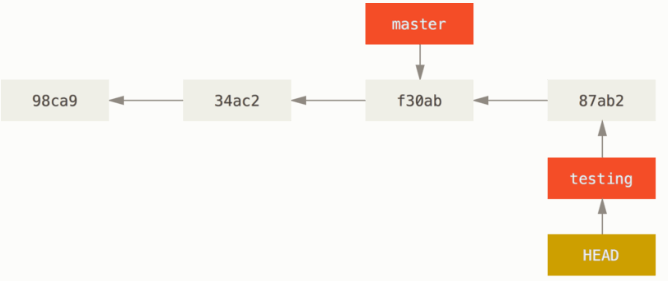


#### **分支的切换**

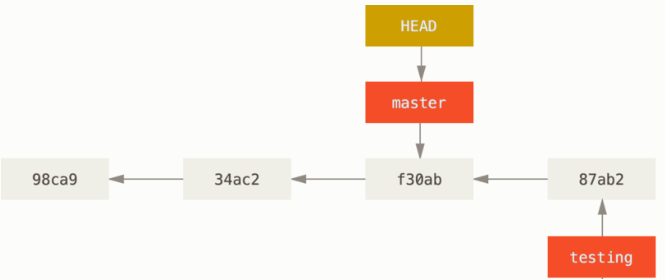
* Head: 特殊指针,指向当前所在的本地分支
  + 下图中HEAD指向testing表示当前本地处于testing分支



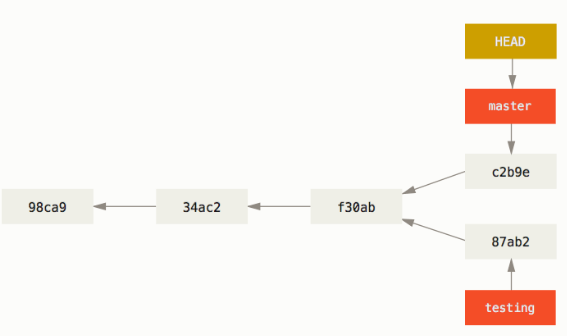
* Testing分支提交后会生成1个新的提交点对象87ab2,而master指针仍留在原提交点f30ab,故分支间互不影响



* + 切换回master只需移动HEAD指针不用做文件的复制操作



* Master提交后才开始有自己独立的分支提交点c2b9e

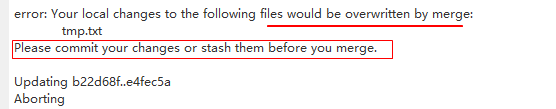


### Git暂存区的意义

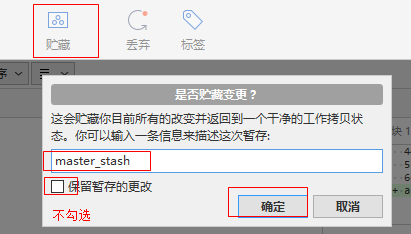
* Git暂存区
  + 暂存区是一个索引文件:.git/index,存的是文件的目录树,记录了文件名、文件的时间戳、文件长度、文件类型以及最重要的SHA-1值，文件的内容并没有存储在其中,文件内容存在.git/objects,每次提交的是暂存区所对应的文件快照(暂存区包含了一次commit的所有文件)
  + 暂存区建立了文件和.git/objects目录下的对象实体间的映射关系
* Git应该如何提交修改
  + 每一个完整的功能应该单独做成一次提交,保证了提交版本历史记录的清晰度,方便回滚,否则当回滚历史时会分不清每个版本包含哪些功能或修复了哪些bug
  + 保证commit的干净和降低commit的粒度
* Git暂存区的作用
  + **保存阶段性的文件提交(功能没写完也可提交)**,而svn需保证提交代码的完整度
  + Stash和stage的区别
    - Stash是一个git命令,将工作区和索引中的内容暂时存到一个堆上,做临时存储
      * 打断当前代码的编写,可切换分支进行其他操作
    - Stage是git中的一块区域,工作区文件提交前要先经过staging area
  + Stash贮藏使用场景
    - 写到一半的代码不适合做一次版本的提交,此时有紧急bug需切换到其他分支操作,切换分支报错,因当前分支有未提交内容不可做分支切换,此时可用git stash先贮藏当前修改内容,贮藏后文件状态是干净的,此时可切换到其他分支修复bug后再切换原分支并取出原贮藏内容继续开发
    - 就可将阶段性的代码先保存到暂存区,等处理完紧急任务后再切回来处理手头的任务,等功能代码写完后再做一次完整的版本提交
  + 还没拉取远程代码就在本地添加了一行”aaa”,此时想起要拉取远程最新代码,而远程也已被修改

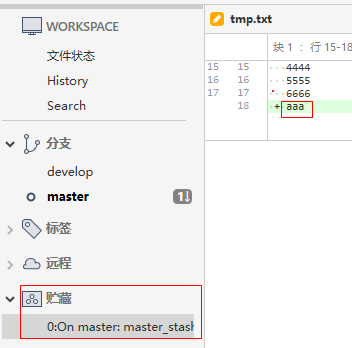


* 拉取merge失败提示本地先commit或stash

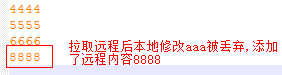


* 点贮藏:贮藏本地的改变，贮藏后本地文件状态是干净的,故可拉取

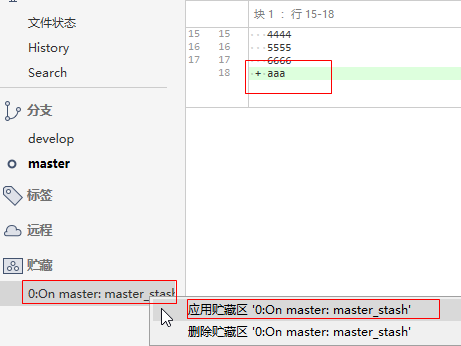


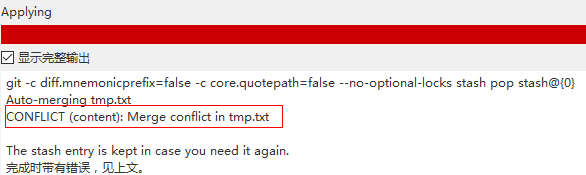


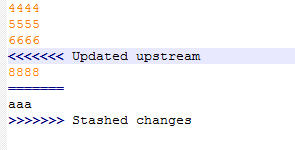
* 贮藏后做拉取,本地修改”aaa”被丢弃,添加了远程内容”8888”



* 恢复贮藏内容,报合并冲突,解决冲突即可







* 解决冲突后,保留了本地修改和远程修改的内容



## **Git Flow工作流**

### **GitFlow是什么**

#### GitFlow定义

* GitFlow是在项目开发中遵循的一种代码管理方案,提供了管理大型项目的框架,给不同分支分配明确的角色并定义分支间如何交互,简化了git操作

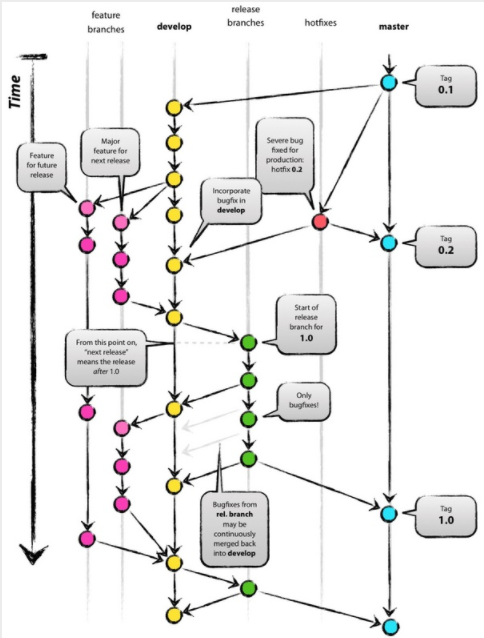
#### **GitFlow分支类型**

* **GitFlow5种分支类型**
  + Master:主分支,存发布正式版本的历史,只要有合并到master分支，就应该打好Tag以方便跟踪
  + Develop: 日常开发分支,功能的集成分支
  + Feature: 具体功能开发分支,只与develop分支交互，为每个新功能新建一个feature分支
  + Release: 预发布分支,基于develop分支创建,用于发布前的测试,测试成功后将其合并到master和develop并打上版本标签
  + Hotfix: 用于线上bug的热修复,该分支是基于线上master分支,bug修复完后合并到master和dev,并打上版本标签



#### **GitFlow**

* GitFlow是由Vincent Driessen 推出一种成功的git分支模型
* GitFlow工作流程图
* 功能驱动式开发: 需求是开发的起点,先有需求再有功能分支,完成开发后功能分支就被合并和删除
  + 项目存在两个长期分支:master(稳定发布版)+develop(最新开发版)
  + 三个短期分支:feature+hotfix+release,一旦完成开发就被合并进develop或master和被删除



* Gitflow和各个开发环境的对应---分支流程对应环境流程---若每个环境进行严格测试则代码上线的bug率会很低

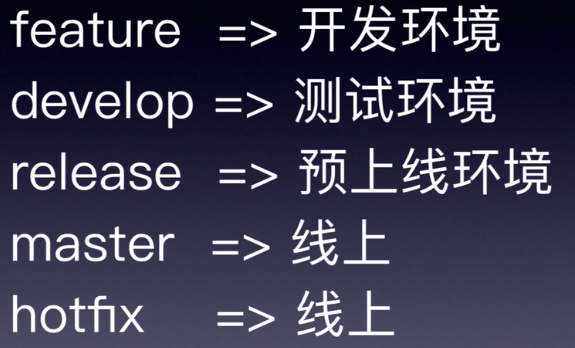
测试环境：团队集体进行功能测试环境

线上环境:与真实环境一致

小团队直接省略release分支,预发布环境用master---

建议:保持dev和feature同步---因我们在开发feature时,别人会在dev上合并代码,当你开发完feature合并到dev时可能会发生很多冲突(因别人的提交同一个文件)---故要时刻保持dev和featur差距不能太大,若团队dev分支代码变动频繁则每天上班前将dev代码合并到当前开发的feature,好习惯---可保证最后feature提交到dev后冲突基本都解决了

Feature分支虽然是个人独立的,但每天的代码也要提交1次到feature,预防突发事件可追回之前提交过的代码.避免几天的工作白费---故每完成1个小功能则提交代码到feature是个好习惯

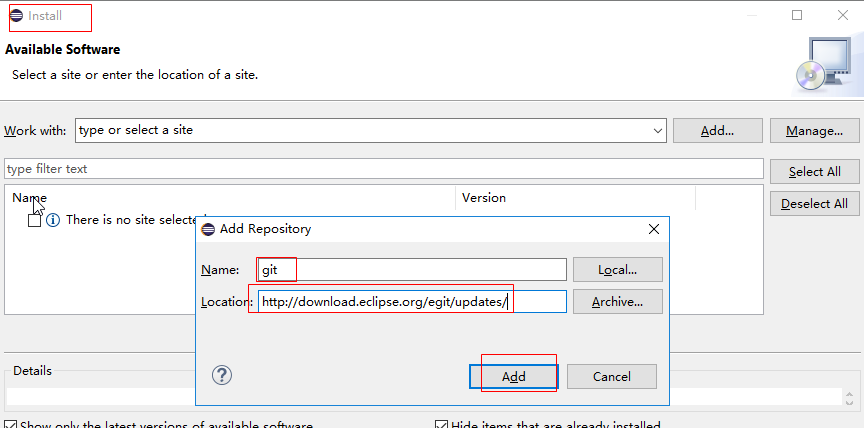


## **Git图形化操作**

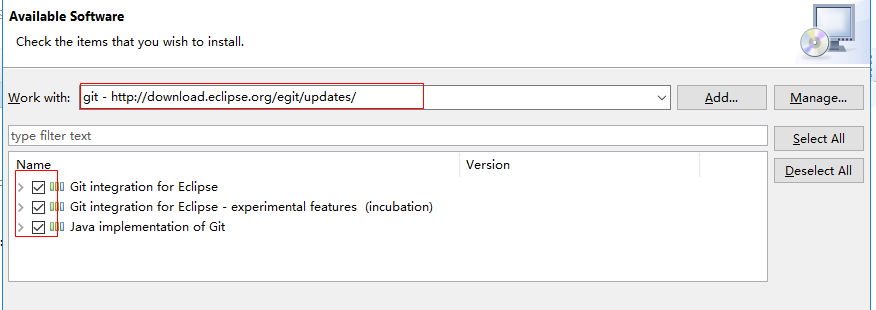
### Eclipse集成Git

#### Eclipse安装Git插件

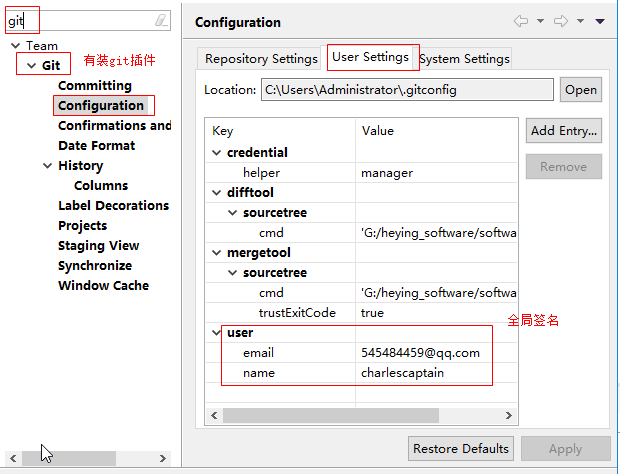
* 查看Eclipse是否已装git插件,若无则需先安装



* 3个勾选后点next做安装即可



* 说明已成功安装git插件



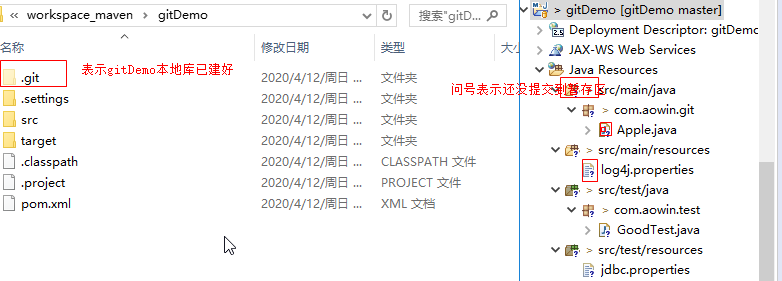
### **Eclipse的Git操作**

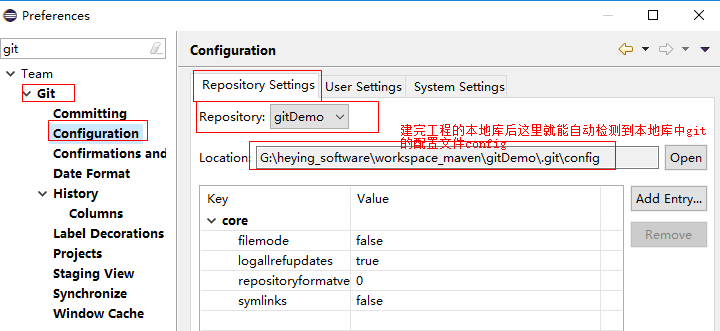
#### Eclipse本地库操作

* 对Eclipse工作区做git初始化,建立git仓库
* 工程--->右键--->team--->share project--->git--->

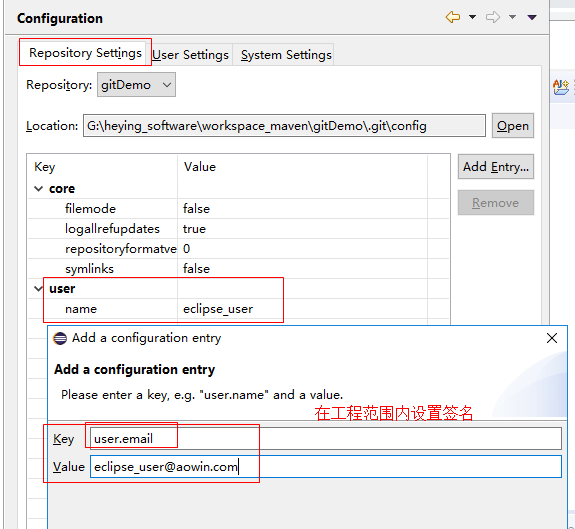




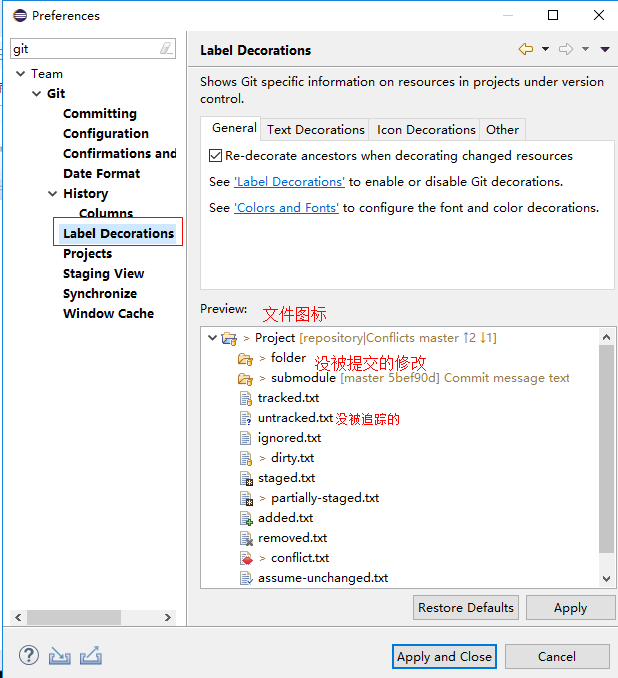




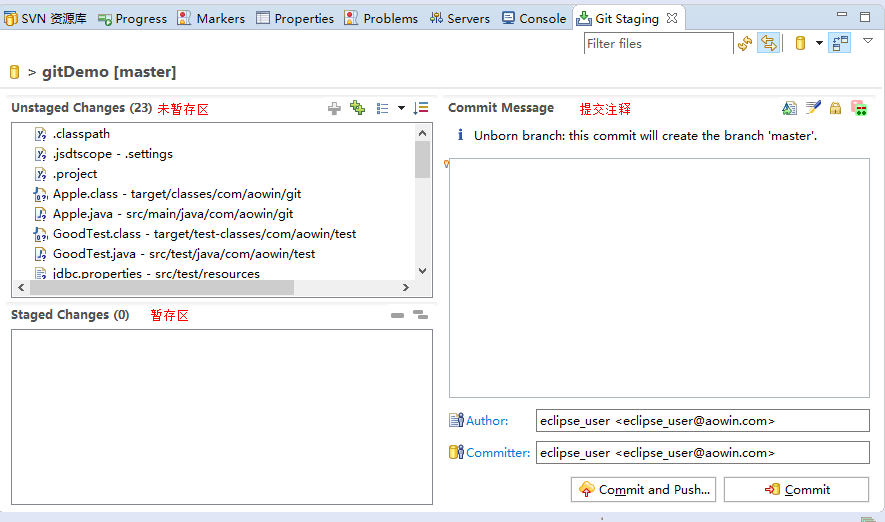
* 设置工程范围的签名



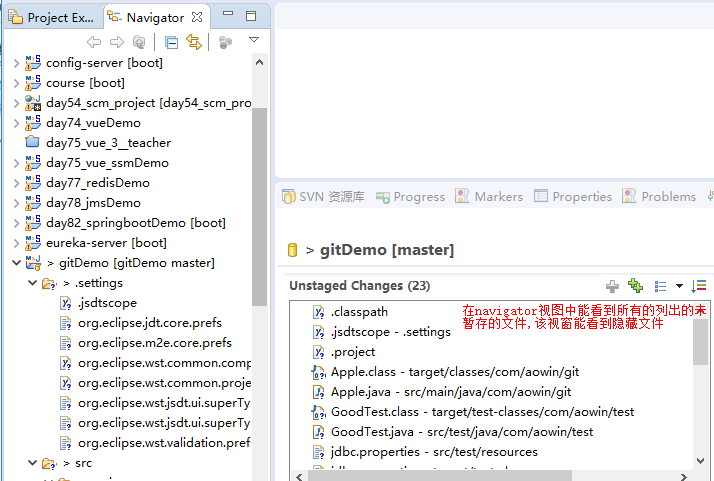
* 文件图标的含义



* Eclipse 对文件的add 和commit操作
  + 选中文件右键--->team--->commit

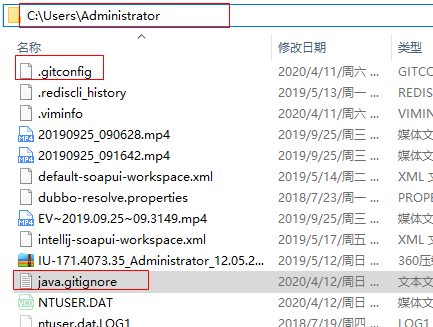


* 用.gitignore文件来忽略Eclipse的特定文件(不想提交的文件)
  + Eclipse特定文件
    - classpath+.project+.settings目录下的所有文件,这些都是Eclipse为了管理工程而建出的维护文件,和开发代码无直接关系,最好不要在git进行追踪,最好将它们忽略
  + 为什么要忽略特定文件?
    - 因不同版本的Eclipse特定文件有所不同,若都提交到同一个远程库,可能会产生冲突,解决这些文件的冲突,对开发并无意义,麻烦,故要忽略
  + Navigator视窗能看到隐藏文件

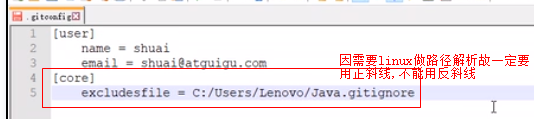


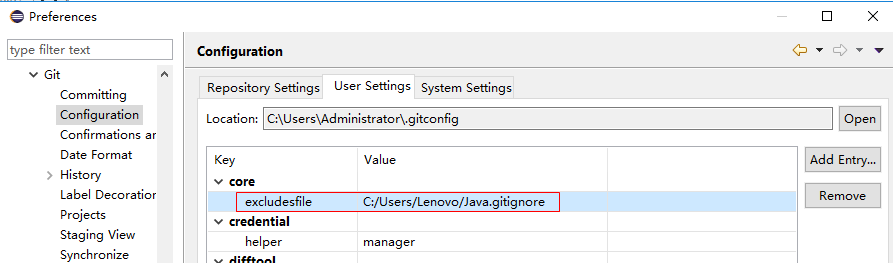
* + 配置忽略文件
    - https://github.com/github/gitignore---列出了各种名称的忽略文件



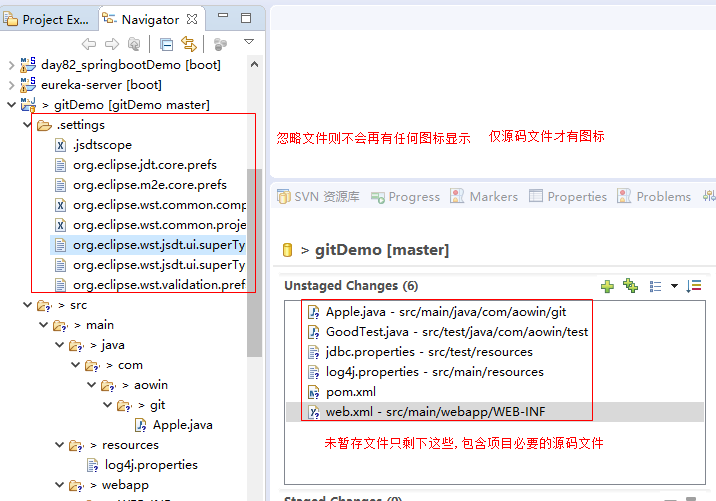


* + - 配置.gitconfig文件(git的全局的配置文件),只要配置的路径是能找到Java.gitignore即可

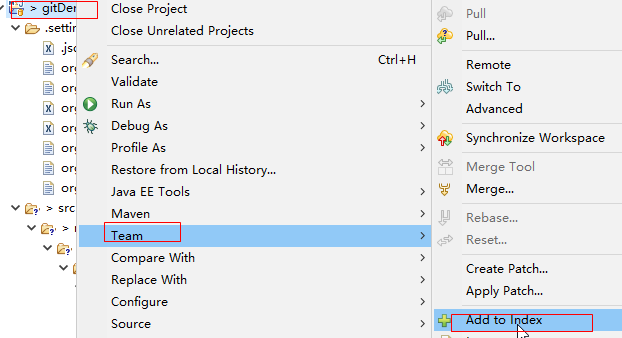


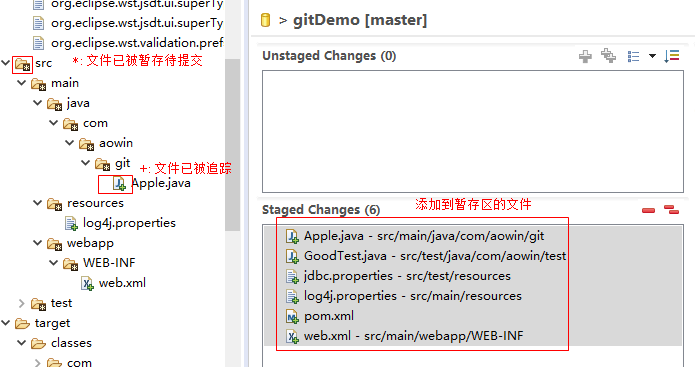


* + - 重启Eclipse后的效果

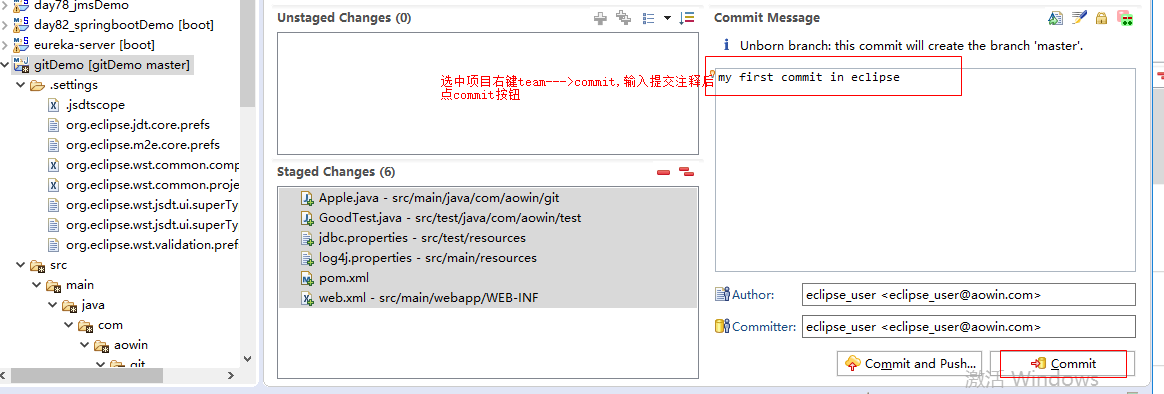


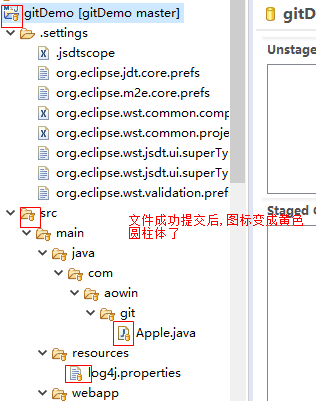
* Eclipse中的git基本操作
  + 将文件添加到暂存区

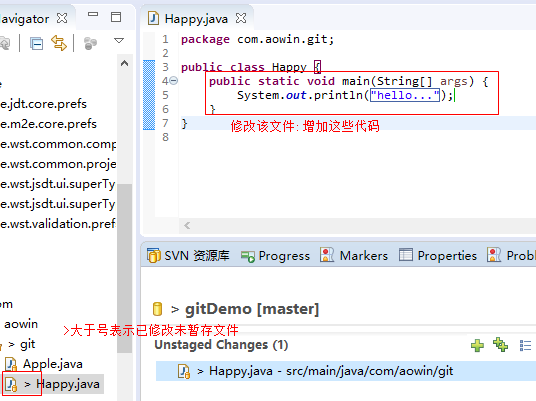




* + 提交文件:add to index+commit或直接commit+拖拽未暂存区文件到暂存区再点commit

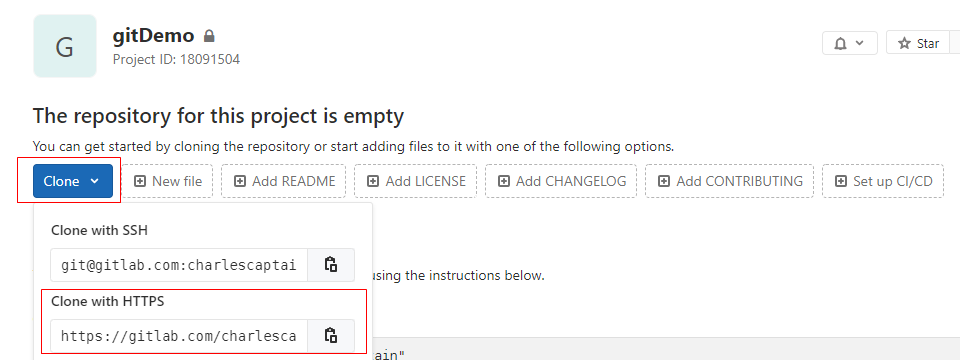


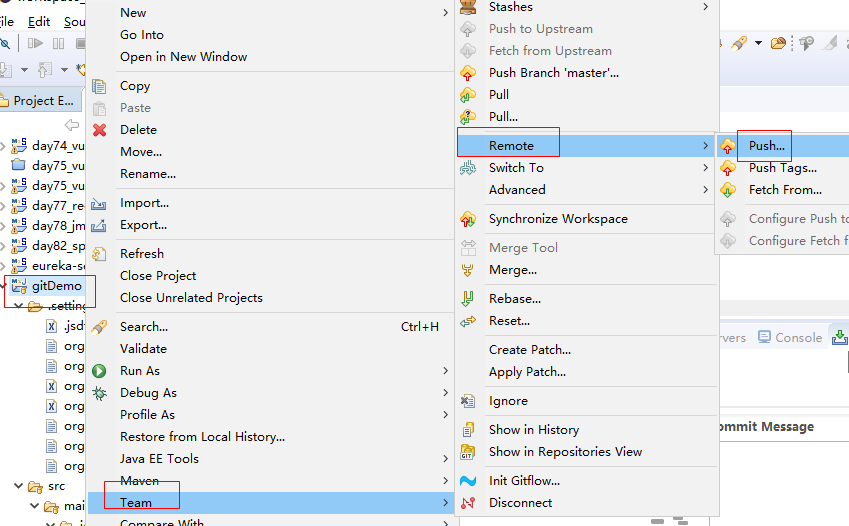


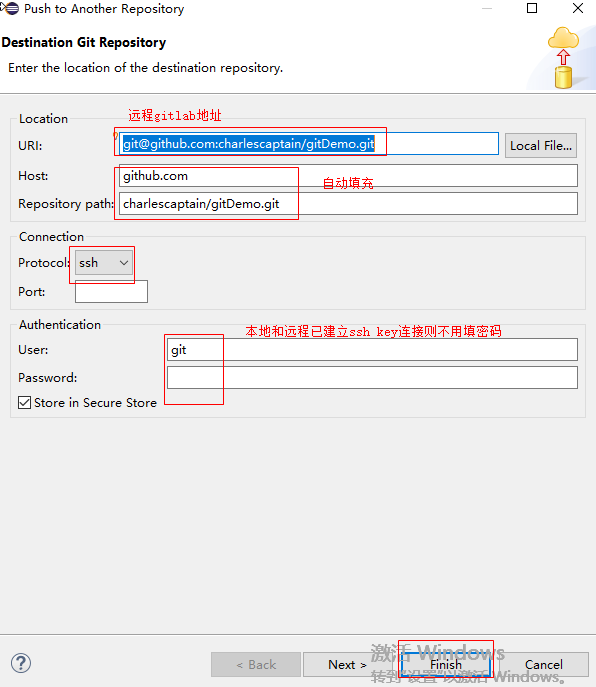


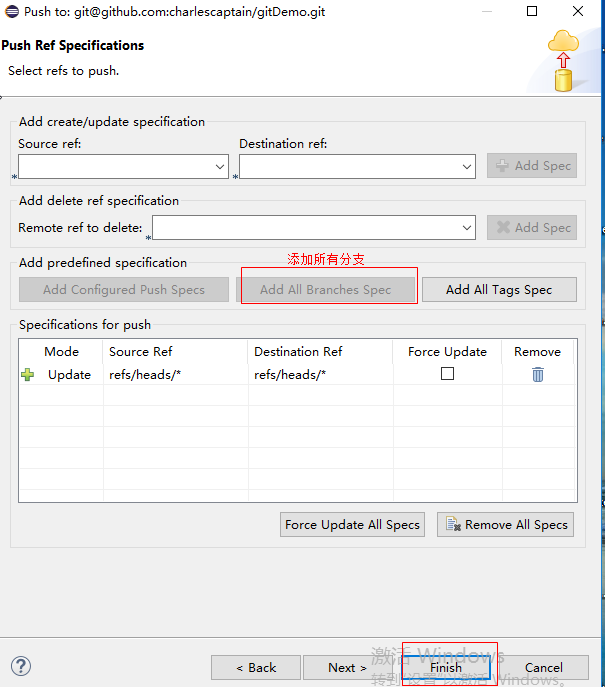
#### **Eclipse远程操作**

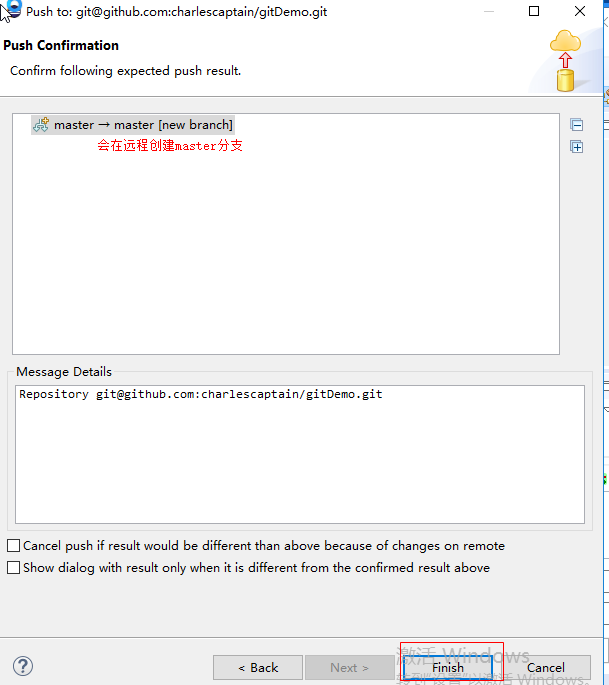
* Eclipse本地库推送到远程库
  + 先在gitlab中新建1个项目并克隆远程地址,因推送时需要指定远程库的地址

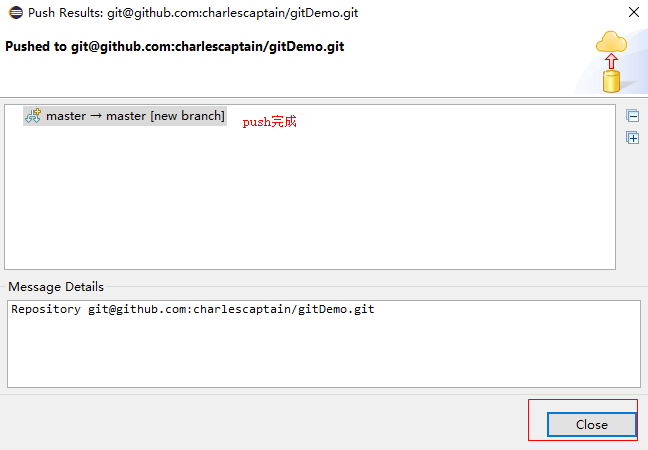


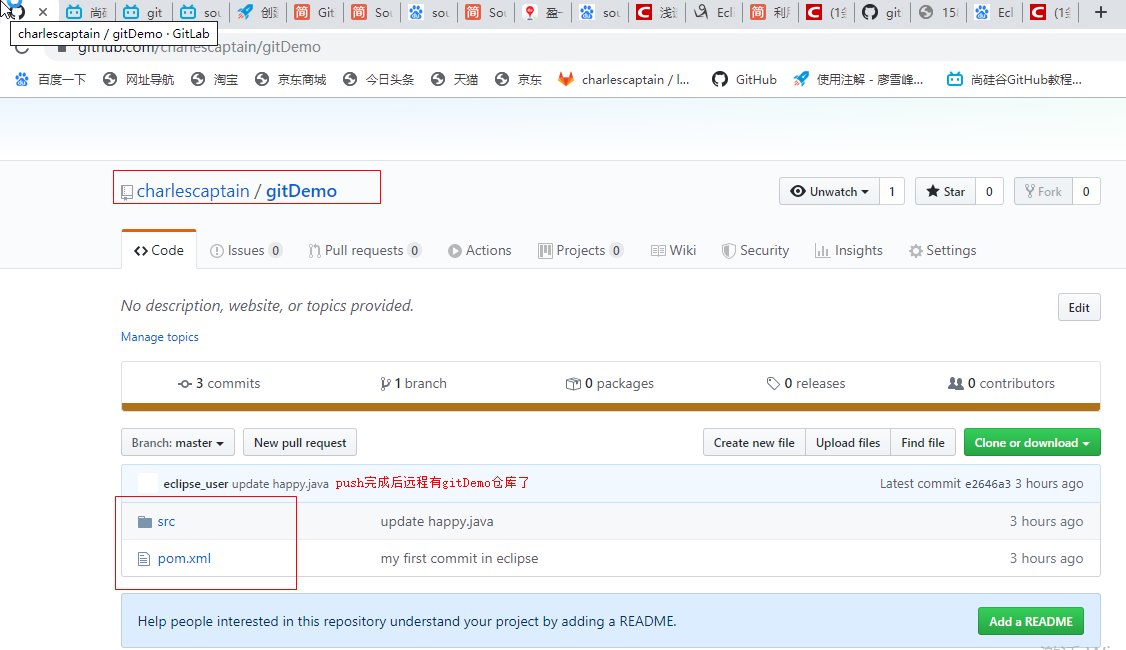




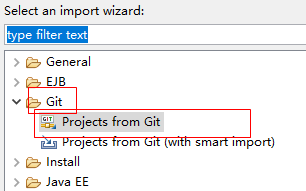


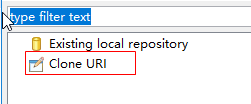




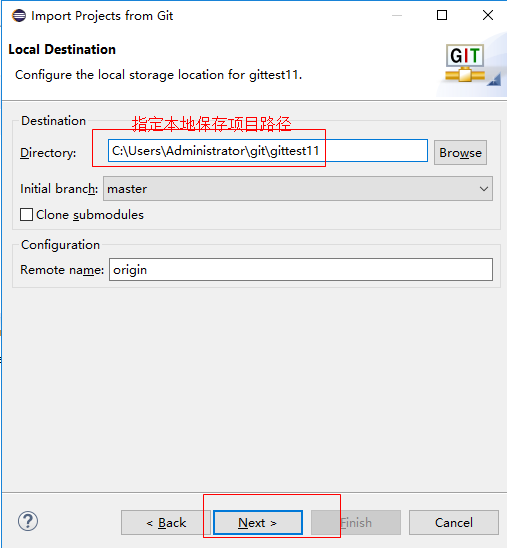


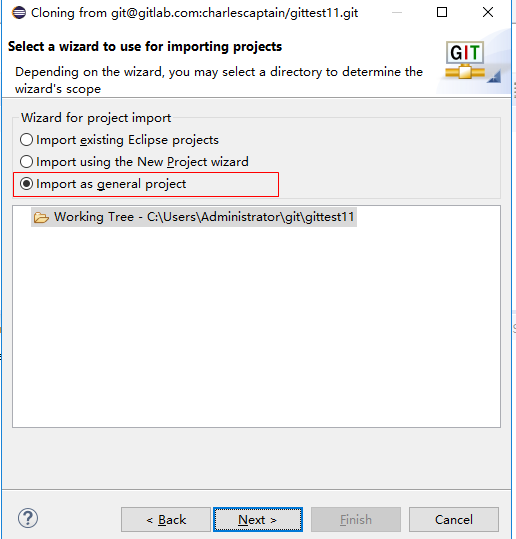
* + Eclipse远程库拉取到本地库
  + 远程项目导入到本地
    - File--->import--->

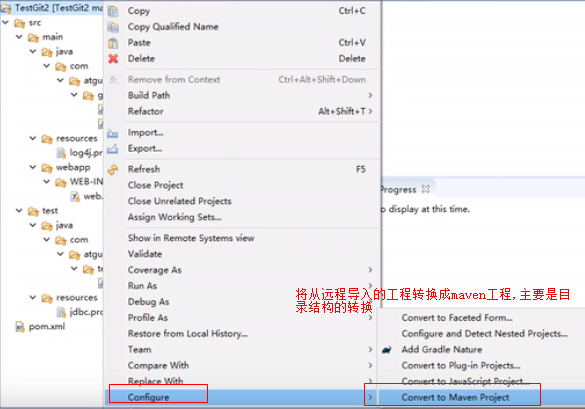






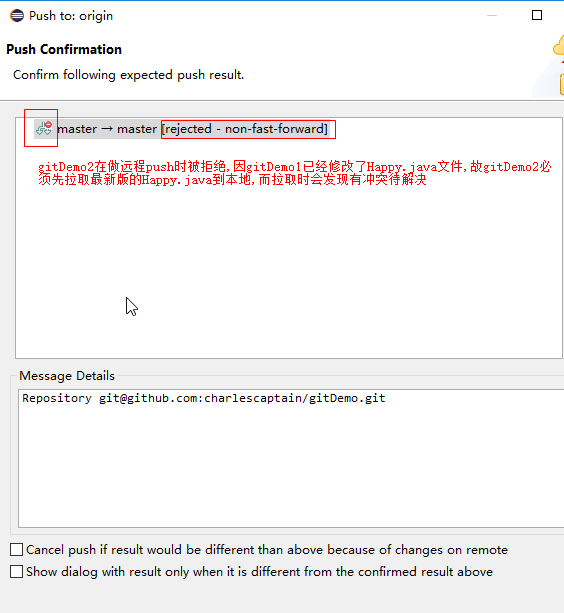




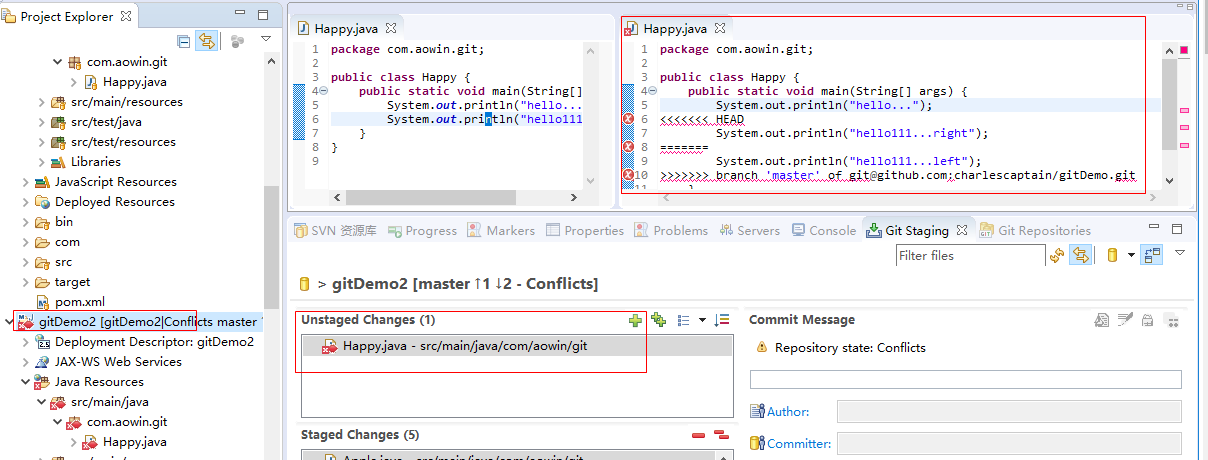


#### **Eclipse解决冲突**

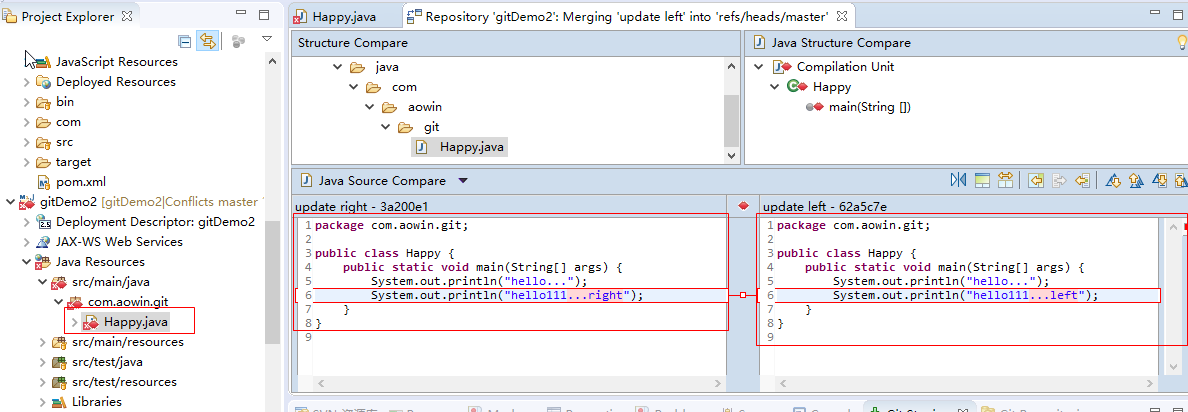
* 本地准备2个项目gitDemo+gitDemo2,这2个项目导入的都是同一个远程项目gitDemo,演示2个项目同时修改同一个文件Happy.java并提交到远程库时会发生冲突
* 对gitDemo和gitDemo2的Happy.java都做修改并提交,推送到远程,发现若先推送了gitDemo则gitDemo2推送时会有冲突

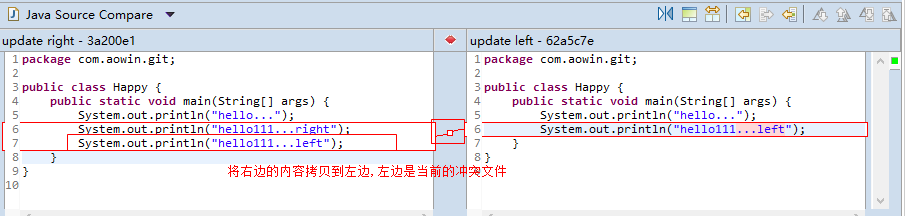


* 对gitDemo2执行pull操作发现有冲突



* 解决冲突(用eclipse的图形化工具来解决,不用手动合并文件)
* 选中冲突文件右键--->team--->merge tool





* 解决冲突后提交Happy.java,并推送到远程库

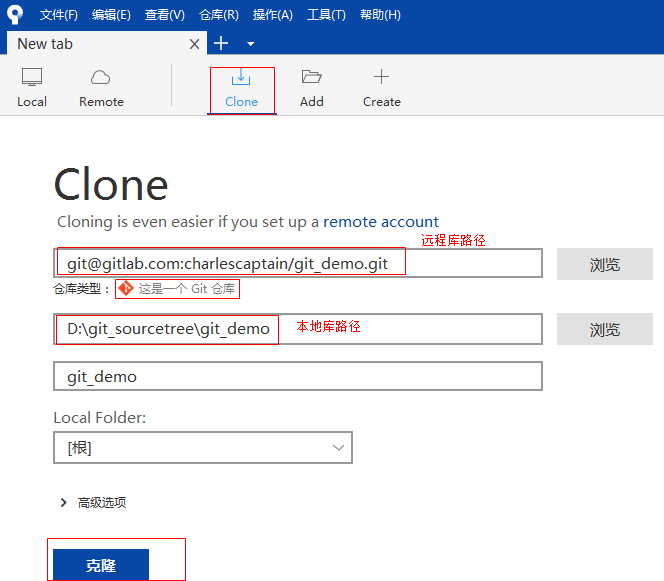
# .SourceTree

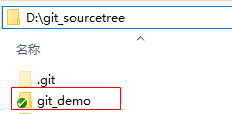
## SourceTree是什么

* SourceTree是一款免费的Git可视化客户端工具,简化了git仓库间的交互

## SourceTree基本使用

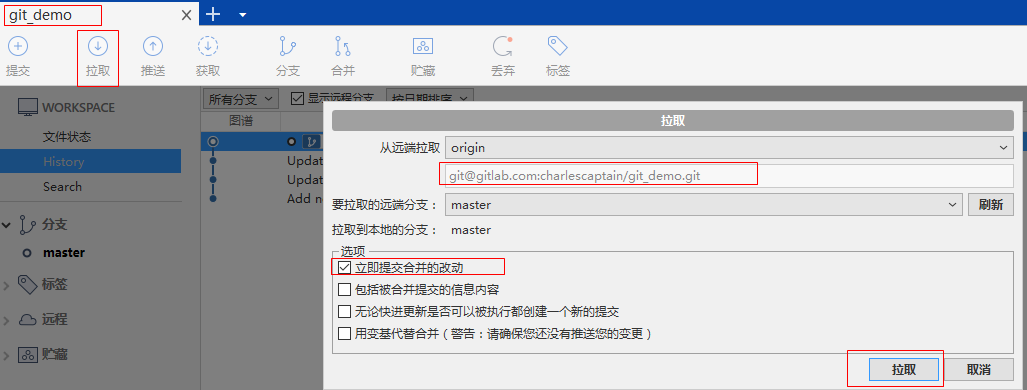
* 克隆远程库git\_demo到本地仓库:D:\git\_sourcetree

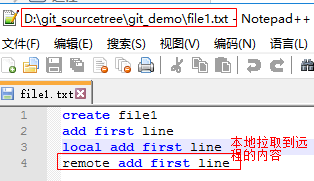




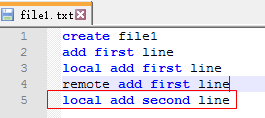
* 远程库修改file1.txt,并拉取到本地库

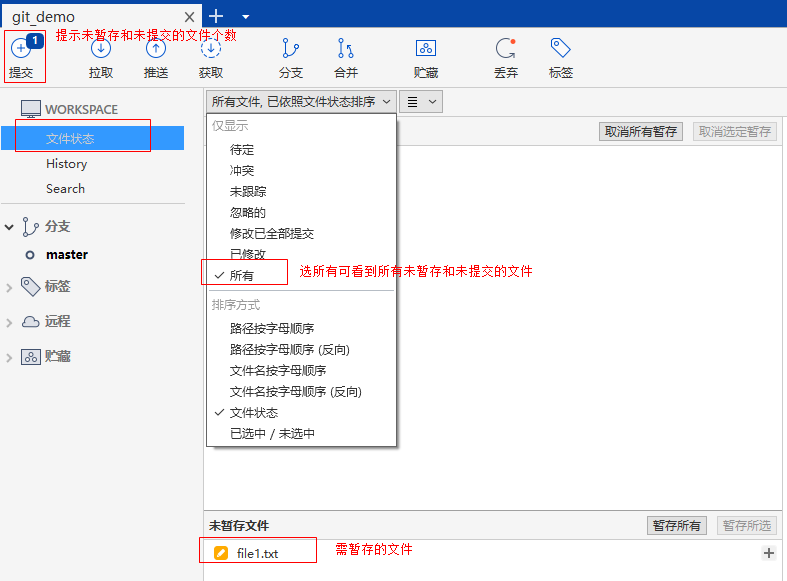


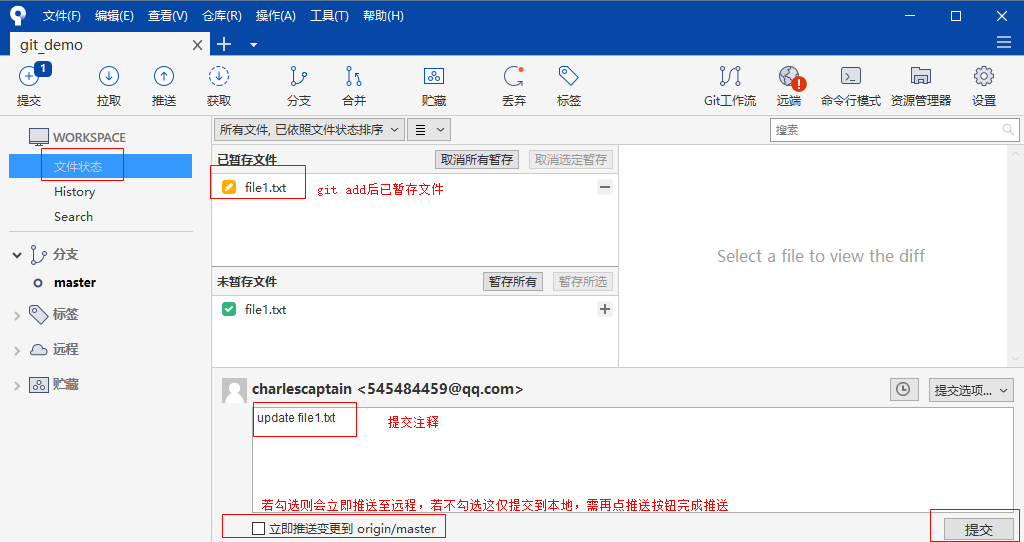


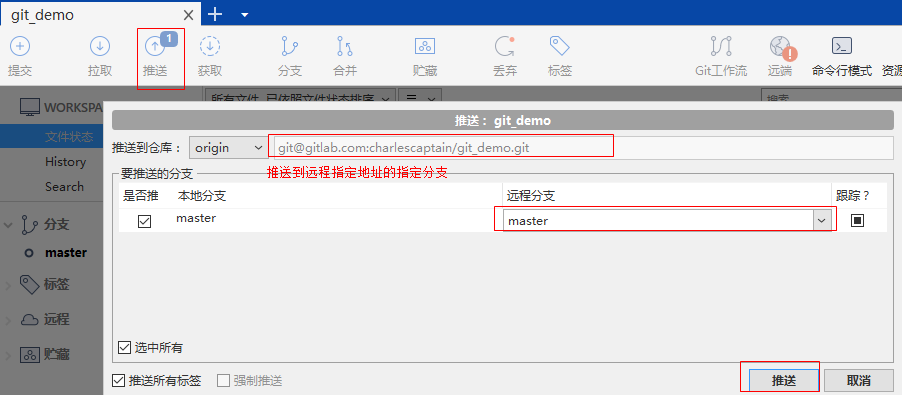


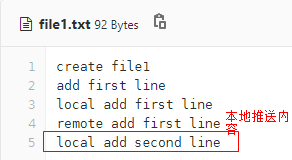
* 本地库修改file1.txt,并推送到远程库



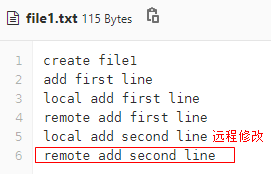


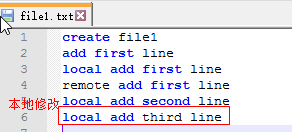


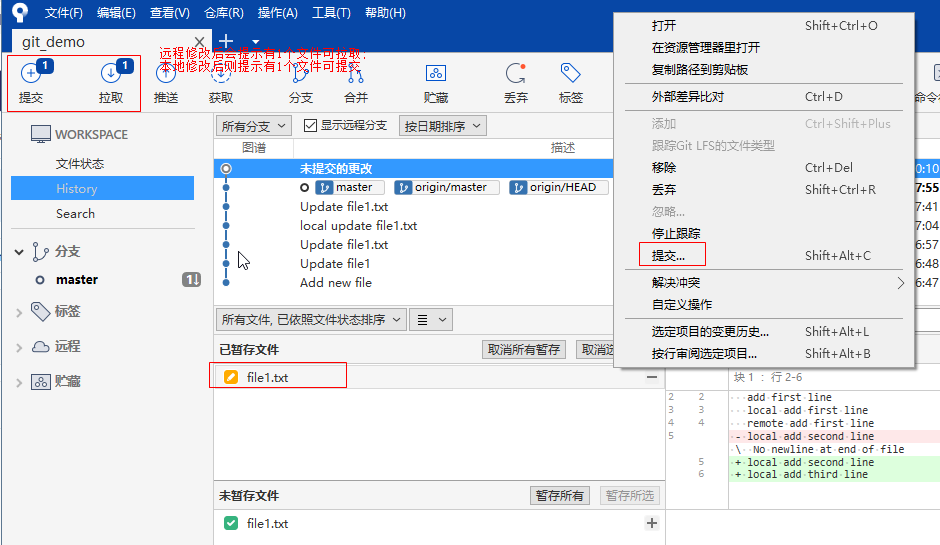


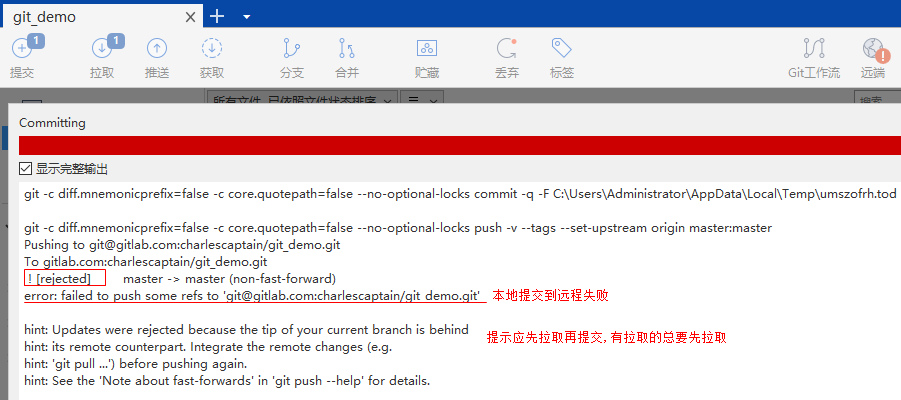


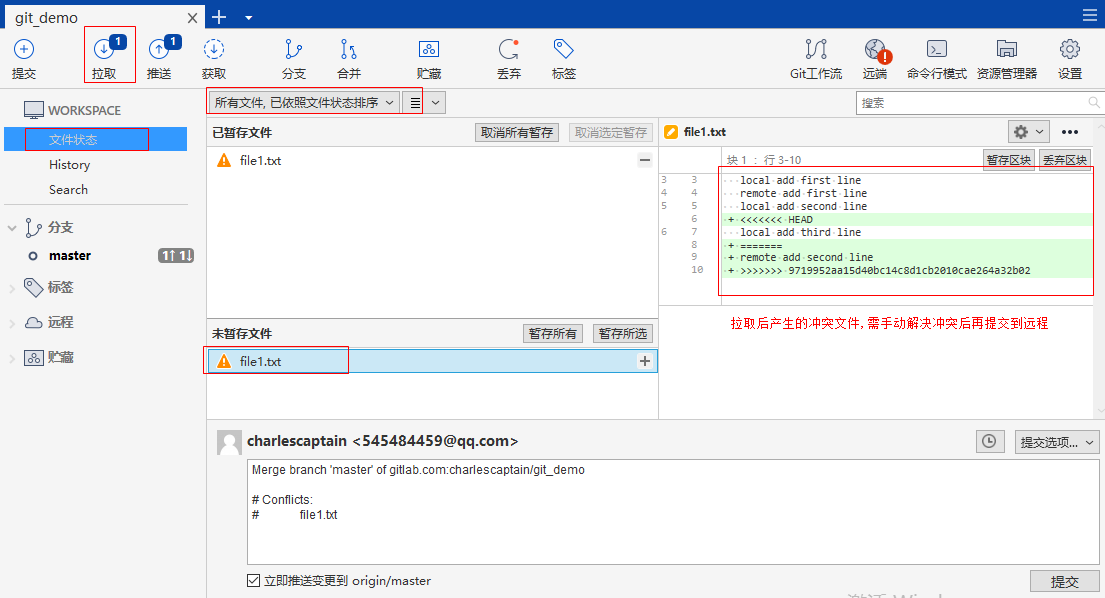
* 解决冲突: 远程和本地都对file1.txt做修改后,由本地推送到远程报错,需先拉取,本地产生冲突文件,解决冲突后再推送到远程

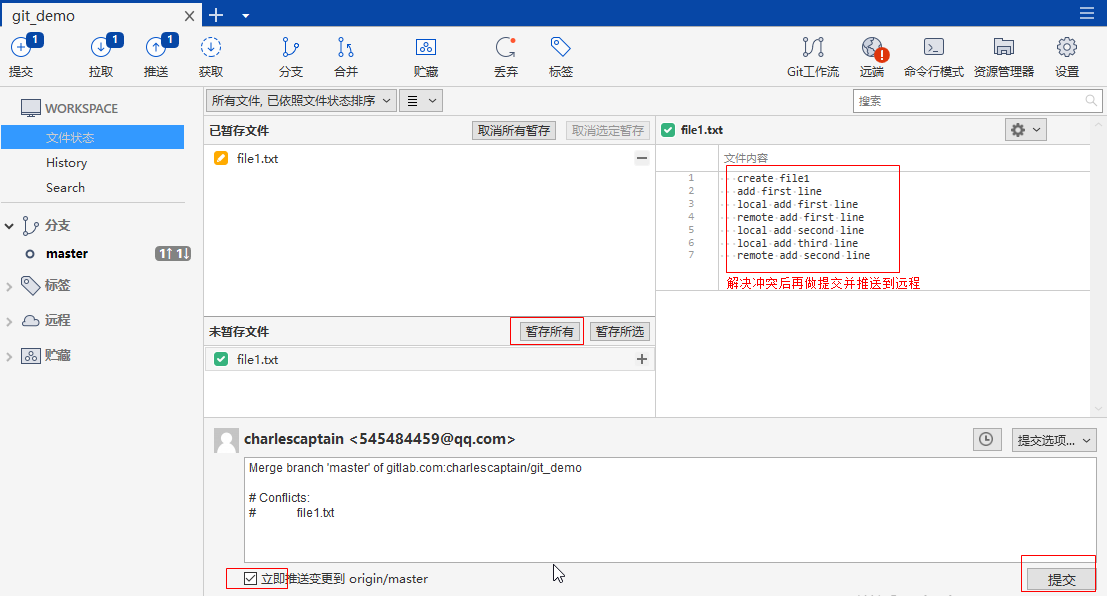






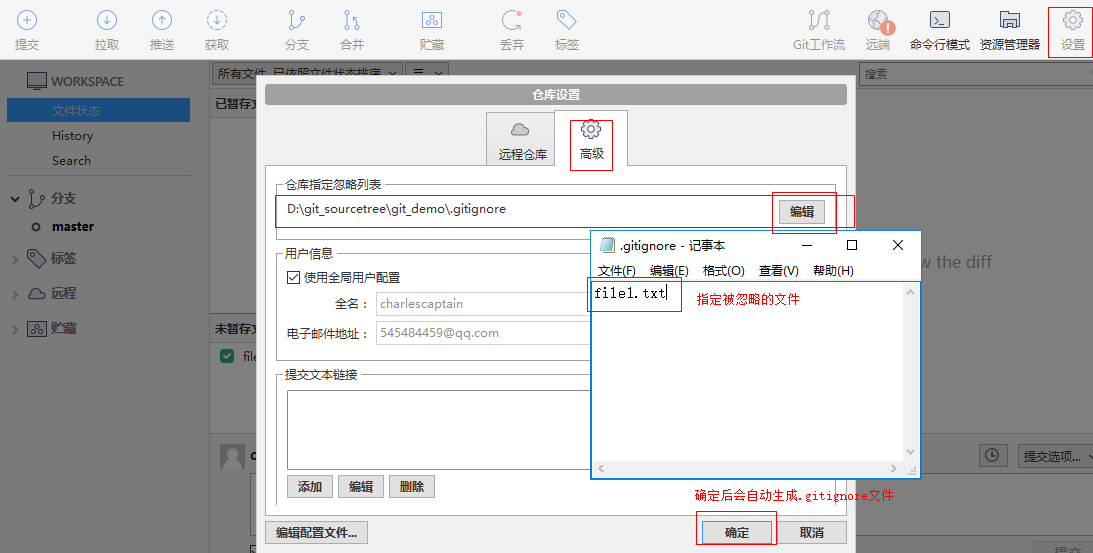


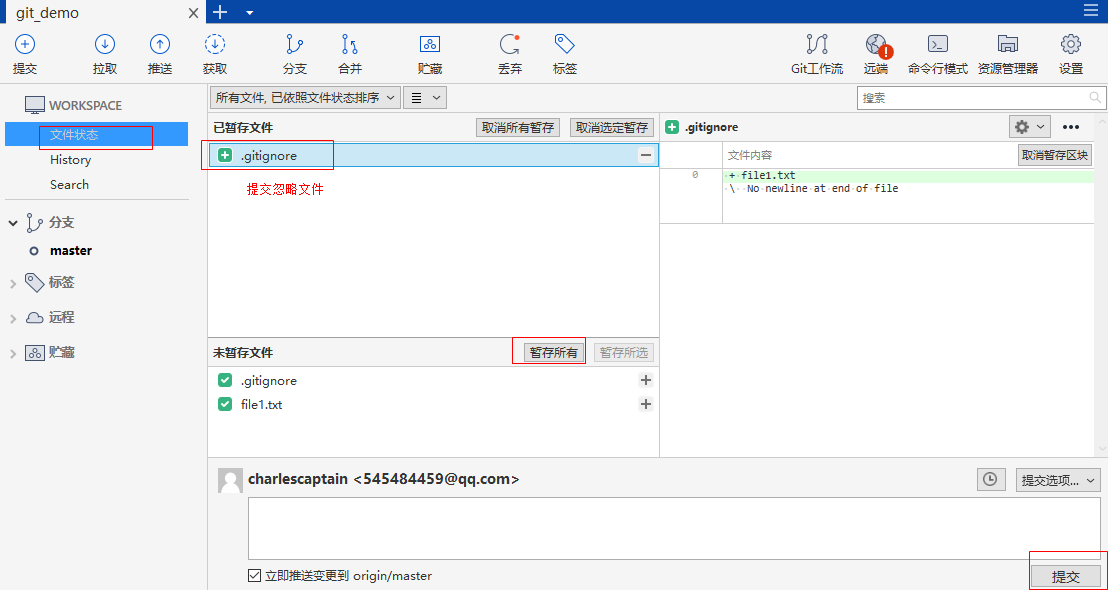




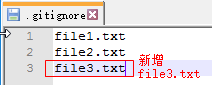
### SourceTree配置忽略文件

* 利用.gitignore过滤文件,指定哪些文件不会被提交到git本地仓库(创建项目时就建好忽略文件,因忽略文件只对没被git管理过的文件起效,若file1.txt之前提交过本地库,此时再将其列入忽略文件是无效的)

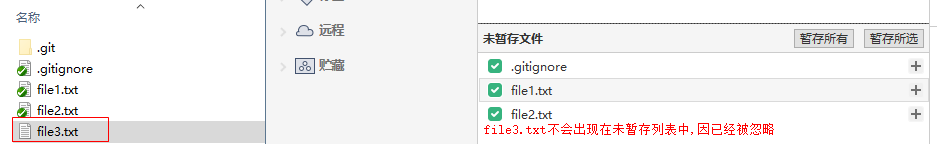




* 先往忽略文件添加file3.txt并提交,再建file3.txt并对其修改,此时git status显示目录仍是干净的,表示file3.txt文件不被git管理

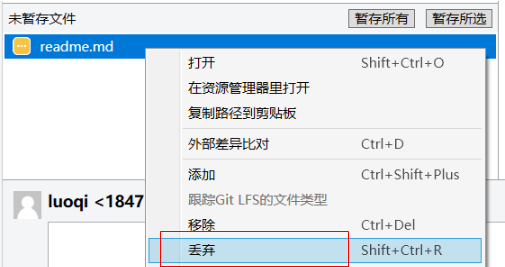




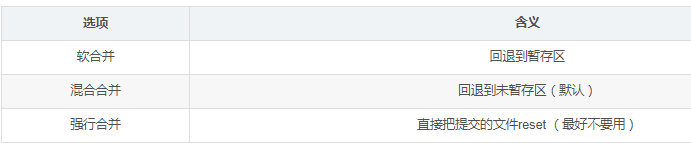


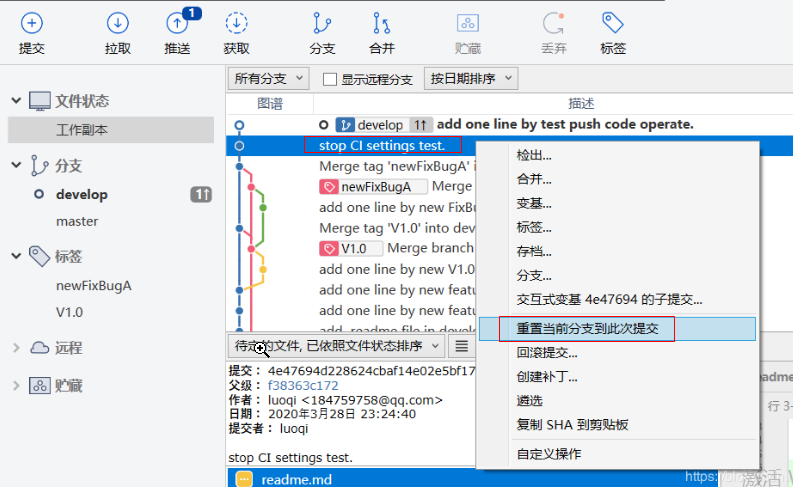
### **SourceTree回滚提交**

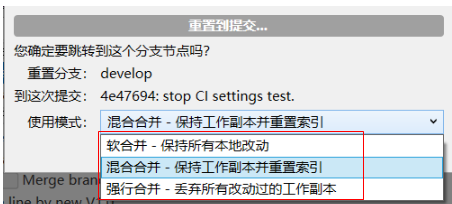
* 三种回滚方式
  + 回滚本地修改(此时还未commit)
* 未暂存区选中文件右键--->丢弃

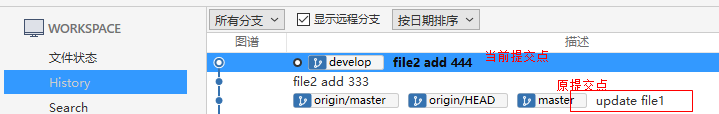


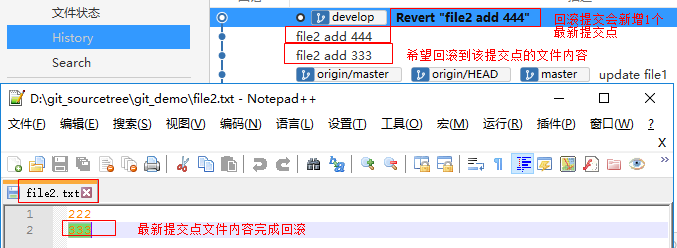
* 回滚本地commit
  + 选中某次提交点右键--->重置当前分支到此次提交--->选混合合并(效果:回滚掉commit,可再次修改,不会增加新的提交点)



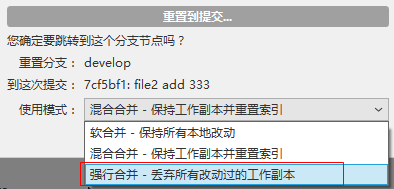




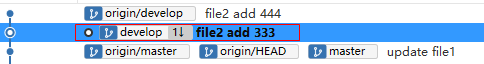
* 回滚远程commit
  + **撤销回滚(git revert): 回滚到上一次提交点的文件内容**--->选中要进行回滚的提交点”update file1”右键--->回滚提交--->文件内容回滚到上一次提交点内容,同时会新增1次提交
    - git只会增加提交点,不会删除提交点--->永远是内容的回退,而非提交点的回退
    - 操作示例: 在当前原提交点”update file1”对file2.txt文件内容添加一行333后提交,再添加一行444后提交,此时最新提交点”file2 add 444”领先原提交点2个版本
* 选中最新提交点右键选回滚提交: 从最新提交点回滚到上一次提交点”file2 add 333”



* 最后将最新提交点内容推送到远程,远程也完成回滚
* 回滚到指定历史版本(git resest)
  + 操作示例: 需求:从最新版本回滚到指定版本”file2 add 333”
    - 选中”file2 add 333”右键--->**重置当前分支到此次提交**--->选强行合并



此时develop内容已回退到”file2 add 333”,但远程并未回退

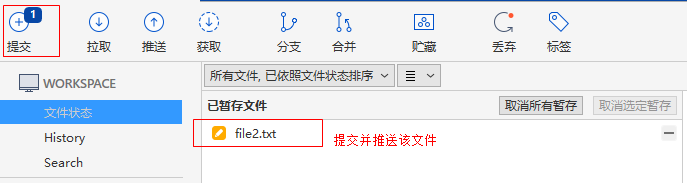


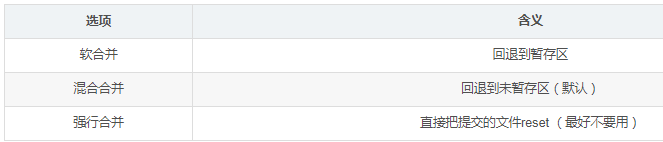
* + - 再选中”file2 add 444”右键--->重置当前分支到此次提交--->选软合并



此时develop回到”file2 add 444”,因是软合并,工作区文件内容没变仍是333



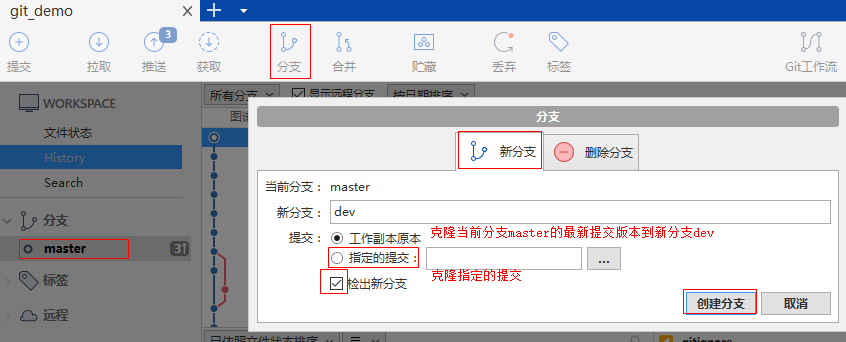
* 回滚远程origin/develop: 软合并后会提示有1个文件待提交,提交并推送该文件,则完成远程内容回滚,此时不会新增提交点,且develop和origin/develop处于同一个提交点,相当于是同一个提交点重新提交1次
* 软合并+混合合并+强行合并的区别
* 混合合并: 保留原提交点文件的工作区内容,不保留暂存区内容,丢弃目标提交点文件内容
* 强行合并: 丢弃原提交点文件所有内容,保留目标提交点所有内容
* 软合并:保留原提交点文件所有内容,丢弃目标提交点所有内容



### **SourceTree分支管理**

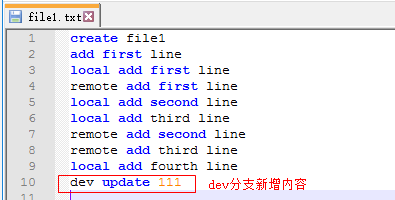
#### SourceTree分支操作

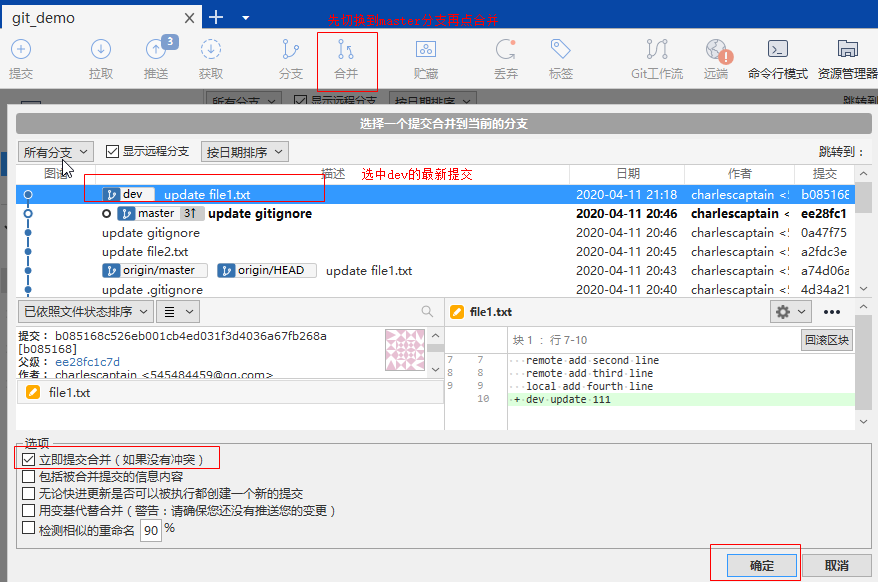
* 创建分支
  + 基于master分支创建dev分支

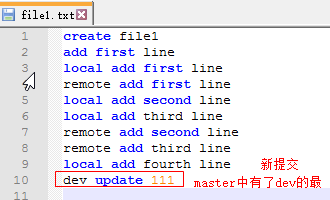




* 合并分支
  + dev分支修改file1.txt并提交,切换到master合并dev



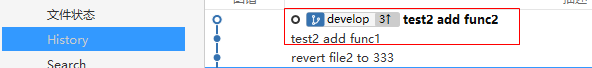




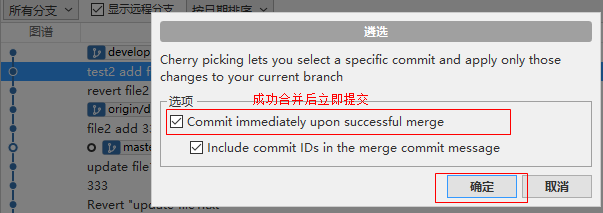
* 删除分支
  + 选中分支右键删除
    - 不能直接删除当前正处于检出状态的分支,切换分支后再删除

#### **SourceTree合并提交**

* 如何用SourceTree合并某次或某几次(不连续)提交到主分支
  + master合并develop某次提交
* 操作示例: 新建test2.txt,添加一行add func1提交--->再添加一行add func2提交



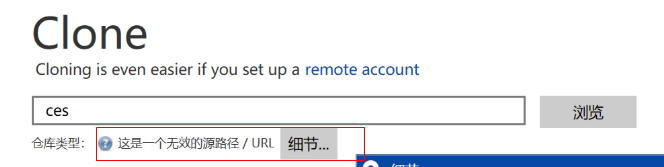
* 若func1功能测试通过需合并到master,而func2不合并
* 切换到master,选中提交点”test2 add func1”右键--->遴选(cherry picking)--->若合并没冲突则会直接提交到master本地库





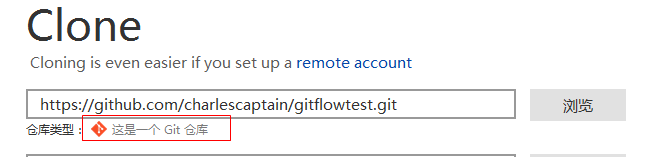
### SourceTree连接异常

* 若克隆时输入地址提示:这是个无效的源路径的解决方案



* + 删除windows保存的凭证后重试则成功:提示:这是个Git仓库





## **SourceTree GitFlow工作流使用**

* SourceTree利用GitFlow管理项目开发
  + 操作示例:若当前master版本为0.1,现要开发版本1.0--->需求发布--->建立feature分支--->合并到develop分支--->建立release分支--->合并到master分支--->hotflix修复master分支bug
    - GitFlow初始化本地库
      * 远程建好gittest2项目并克隆到本地
      * 点击GitFlow完成本地库初始化,会自动切换到develop分支





* 将本地develop分支推送到远程develop

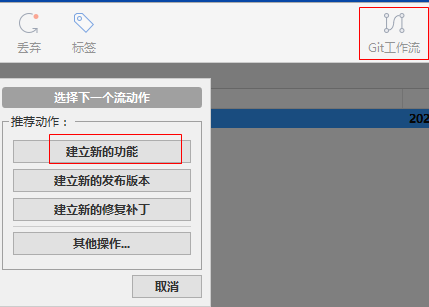


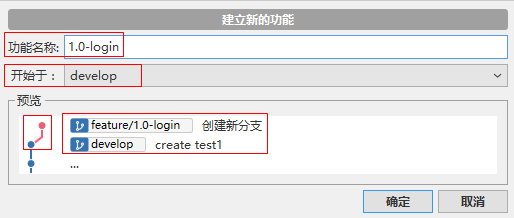


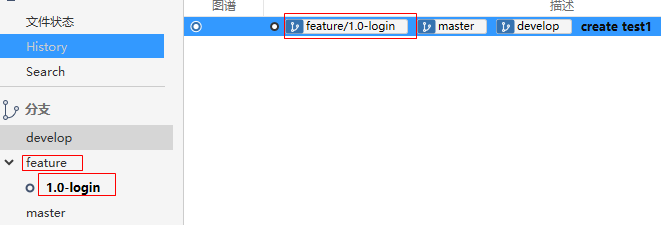
* 远程有develop分支



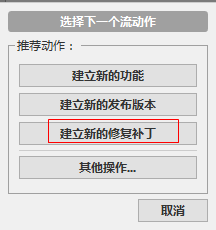
* 现有2个功能需求:login+register,login是1.0版本上线,register是2.0版本上线,此时需从develop拉取2个feature分别做login和register功能开发

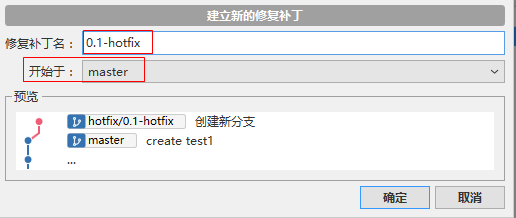


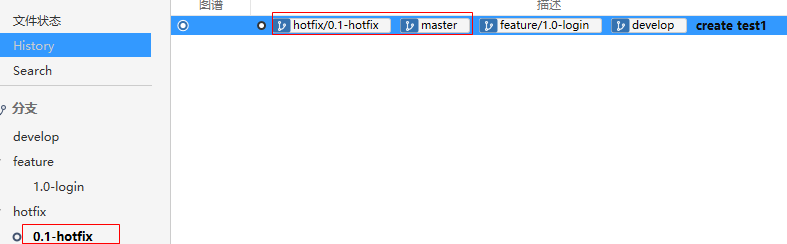


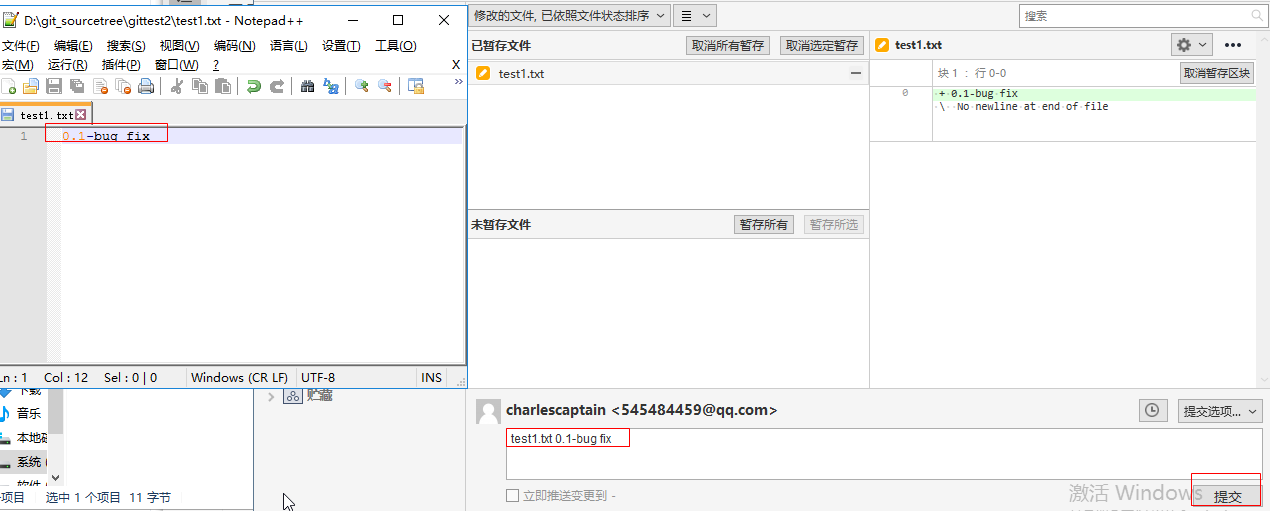


* 此时其他同事也在开发1.0会促使develop不断增加新的提交点
* 此时在开发feature时若在线上发现bug需修复,代号:issue-101,因手头工作没完成可先git stash一下,再立刻从master拉取一个hotfix分支:hotfix/issue-101--->修改test1.txt添加一行”0.1-bug fix”提交并合并到master--->修复完成后再git stash pop回到工作现场,继续未完成的工作

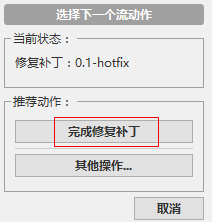


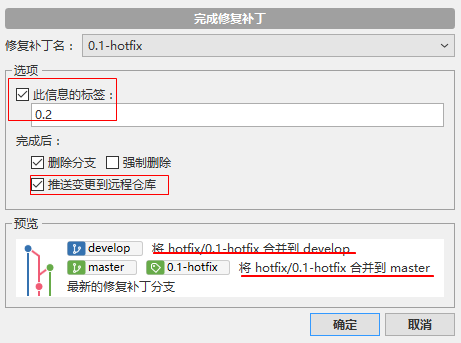






* Hotfix修复完bug后合并到master,保证线上bug的修复,并在master打了小版本标签0.2,标记版本升级,同时也合并到develop,保证了下次上线从dev到master不再出现该bug,若存在release分支也要合并到release

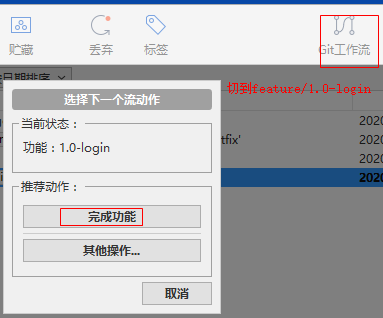




* 回到feature/1.0-login--->新建login.txt提交+添加一行”do login”提交,此时login功能已开发完成--->将login合并到develop提交给测试人员做功能测试



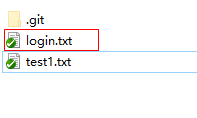
* + 切到feature/1.0-login--->点完成功能



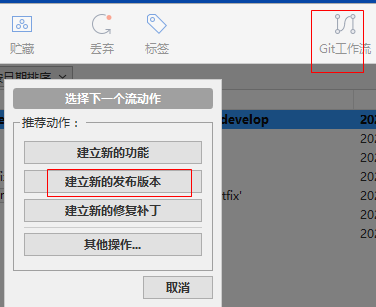
删除feature/1.0-login分支,合并并切换到develop



Develop中有login.txt



* Develop功能测试若产生bug则拉取一个功能分支feature/1.0-login-bug fix,修复完成后选完成功能分支合并到dev并推送到远程develop
* 当测试人员在进行login功能测试时,开发人员仍可以同时开发register功能,做到开发和测试的隔离
* Develop功能测试完成后需进行版本发布,此时从develop拉取release放到预上线环境提供测试

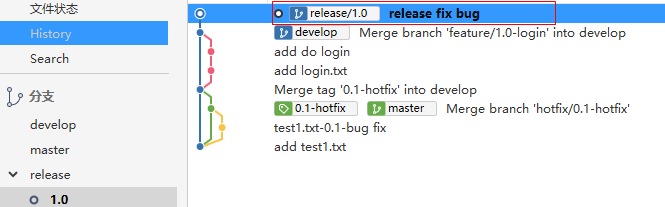




* 将release分支推送到远程提供给测试

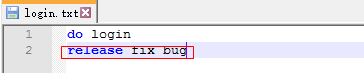


* 若测试发现release有bug且需在1.0上线前修复,则直接在release上修复bug: 在login.txt添加一行”release fix bug”提交推送到远程



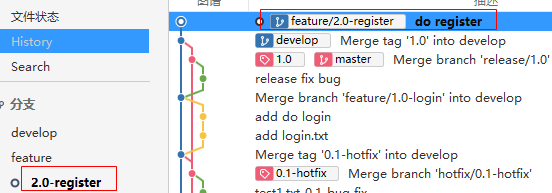
* 此时若release bug修复完成则合并到master打上标签1.0(正式发布版)+每次release改动都要合并到develop,保证下次该bug不会重现,若暂时不需要上生产则合并到develop+保留release分支等待发布到生产

Develop和master都包含”release fix bug”

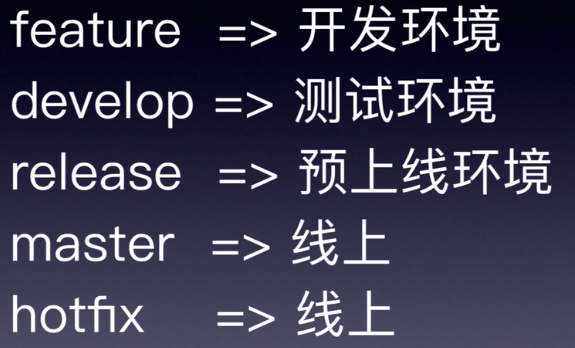


* 此时develop上2.0版本还在开发,包含一个正在开发的feature/2.0-register和一个正在上线的release的bug修复





* 之后就到了2.0版本上线,再发起1次从develop--->release--->master的上线流程,整个版本发布的循环就结束了
* GitFlow各个分支和开发环境的对应



* 如每个环境进行严格的测试则代码上线的bug率会很低
* 建议: 保持feature和develop分支代码同步,因当A在开发feature时,BCD会在develop上合并代码,当A完成feature合并到develop时会产生很多冲突,故要时刻保持develop和feature差距不能太大,若团队develop分支代码变动频繁则需在每天上班前将develop代码合并到当前开发的feature,这样保证了最后feature提交到develop时冲突都解决了;feature分支虽然是独立的,最好还是将每天的代码提交1次到feature,保留代码历史,预防突发事件