Git

简介

Linus花了两周时间自己用C写了一个`分布式`版本控制系统，就是Git！

Git是目前世界上最先进、最流行的分布式版本控制系统。

分布式vs集中式

分布式	集中式
Git,Mercurial,Bazaar	Svn,Cvs
不需要中心服务器	需要中心服务器
不需要联网	需要联网
速度快	速度慢
SourceTree，TortoiseGit，Gitlens	TortoiseSVN，SmartSVN

为什么要进行版本控制?

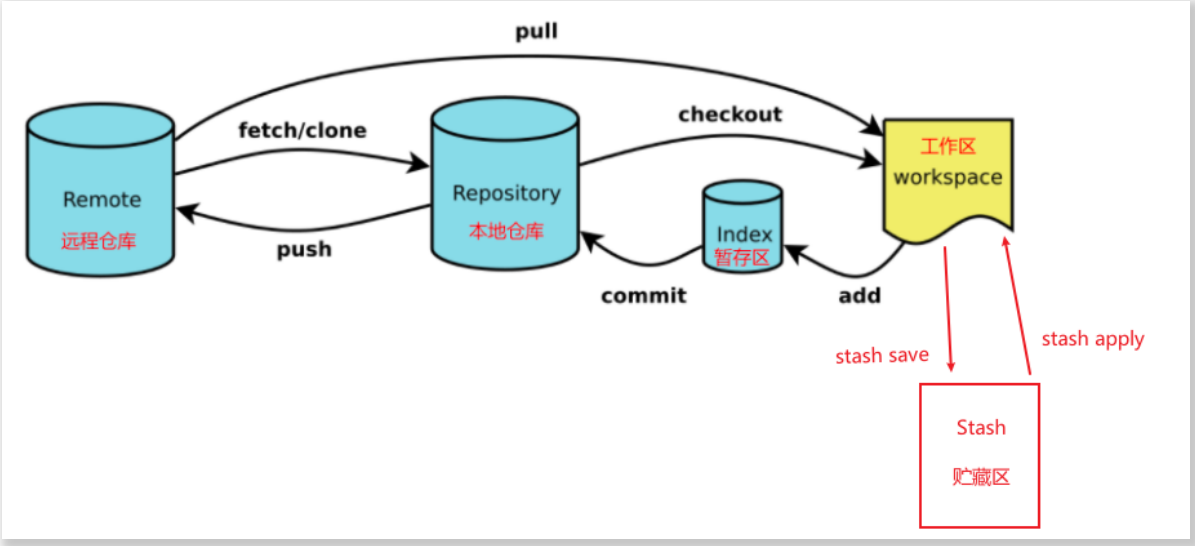
没有版本控制的时候

- 记录文件变更
- 多人协同
- 维护多个版本

什么文件不适合版本控制?

- 日志文件
- 临时文件
- 生成文件
- 编辑器配置文件
- 环境配置文件
- 二进制文件（图片、音频、视频），推荐使用git lfs

基本概念



Workspace: 工作区，当前正在编辑的文件

Index/stage: 暂存区，添加到暂存区的文件将被提交到版本库

Repository: 本地仓库，接受暂存区提交的文件

Remote: 远程仓库

Stash: 贮藏区，临时存放修改的文件

安装Git

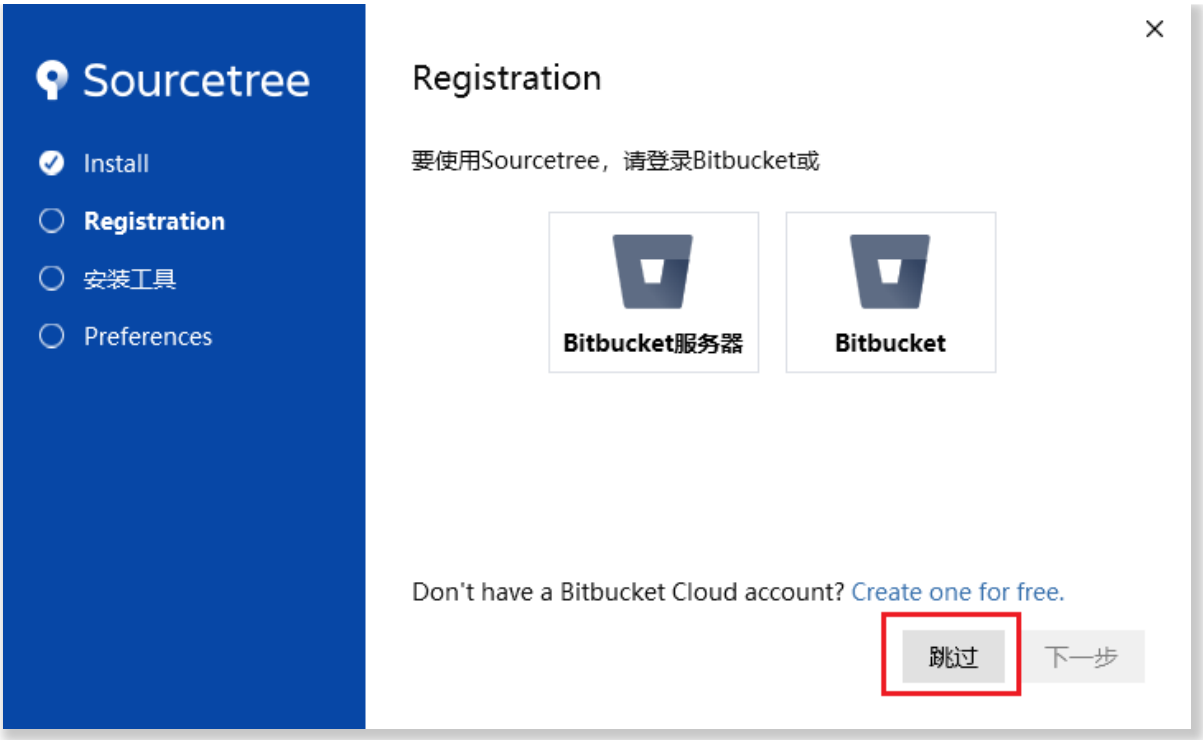
https://git-scm.com

配置Git

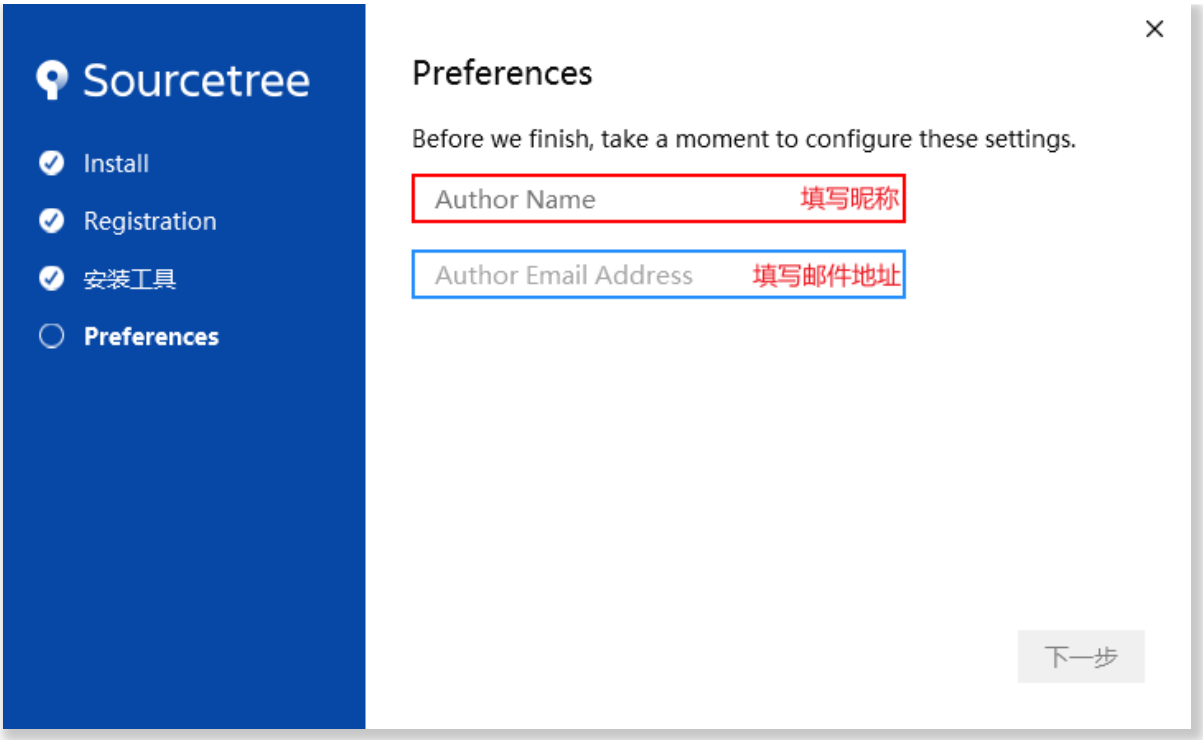
```
//配置用户名
 git config --global user.name "你的用户名"
 //配置邮箱
 git config --global user.email "你的邮箱"
```

安装Sourcetree

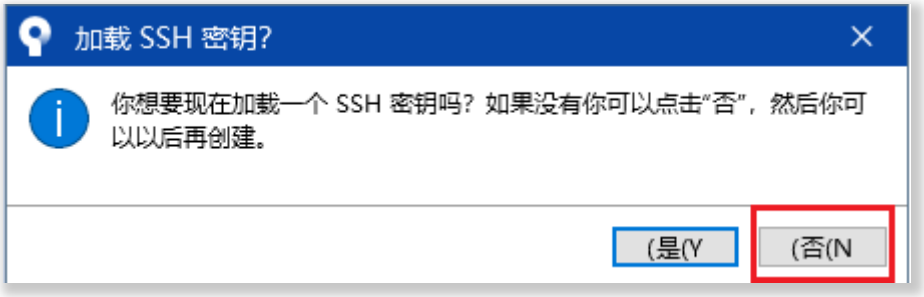
https://www.sourcetreeapp.com/



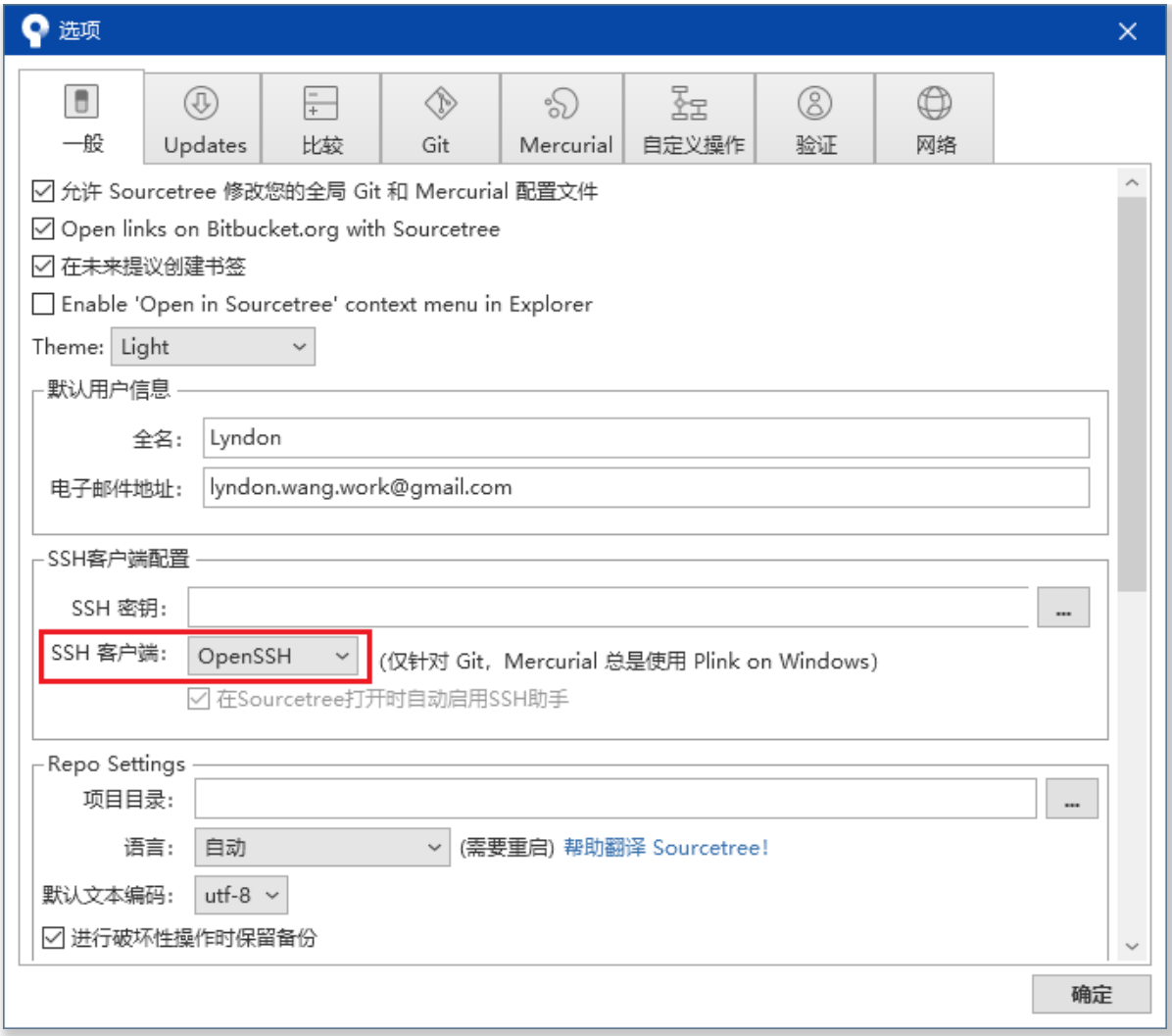
不勾选Mercurial



第一次打开提示是否加载ssh密钥，点击否即可



进入《工具》-《选项》，修改ssh客户端为`OpenSSH`



Git常用功能

基本使用流程

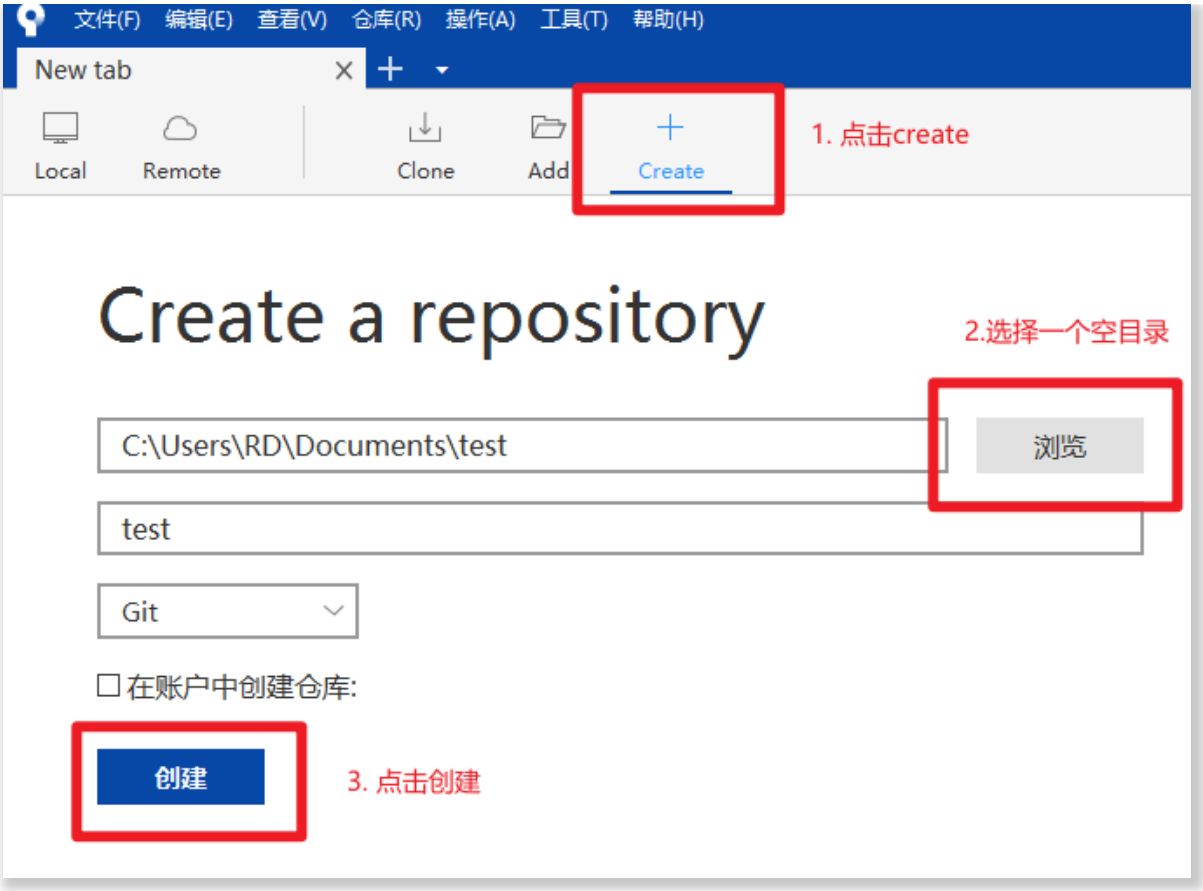
初始化

使用命令行

```
#进入工作目录
cd d:\code\git
#初始化git仓库
git init
```

使用sourcetree

注意：sourcetree不能将已有文件的目录初始化，只能从空目录开始初始化



克隆远程仓库

使用命令行

```
#进入工作目录
cd d:\code\git
#克隆
git clone https://gitee.com/LyndonWang/gitstudy.git
#或
git clone git@gitee.com:LyndonWang/gitstudy.git
```

https协议需要输入账户密码，如果要无密操作，需要使用git协议

使用git协议克隆仓库

git协议需要使用ssh密钥，ssh密钥分为公钥和私钥，公钥可以公开，私钥不能公开

- 生成密钥对

  ```
  ssh-keygen -t ed25519 -C "密钥名称"
  #一路回车继续，注意公钥保存地址
  ```

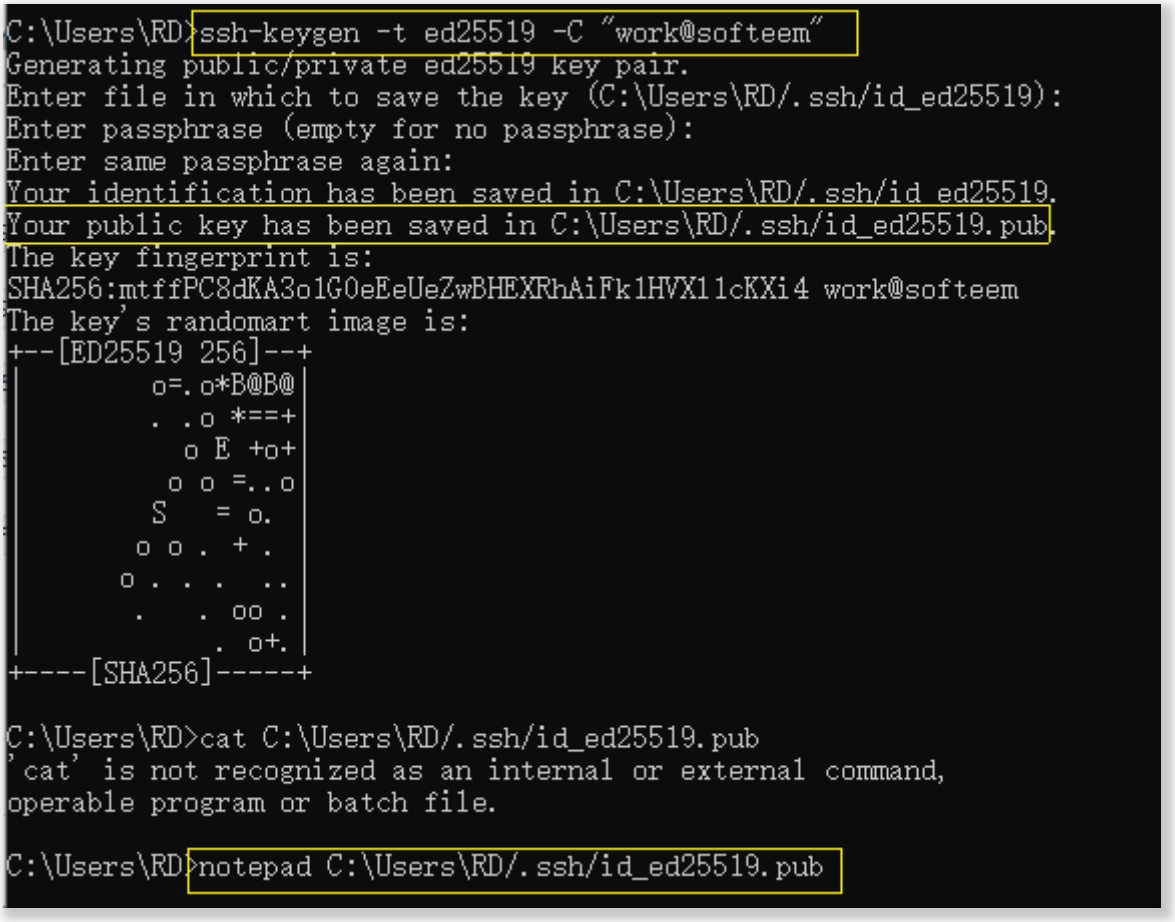
- 打开后缀为`.pub`的公钥文件，复制公钥

  ```
  notepad 公钥文件
  ```

- 打开gitee公钥管理页面

  粘贴公钥到公钥输入框，标题如未自动输入可随意填写

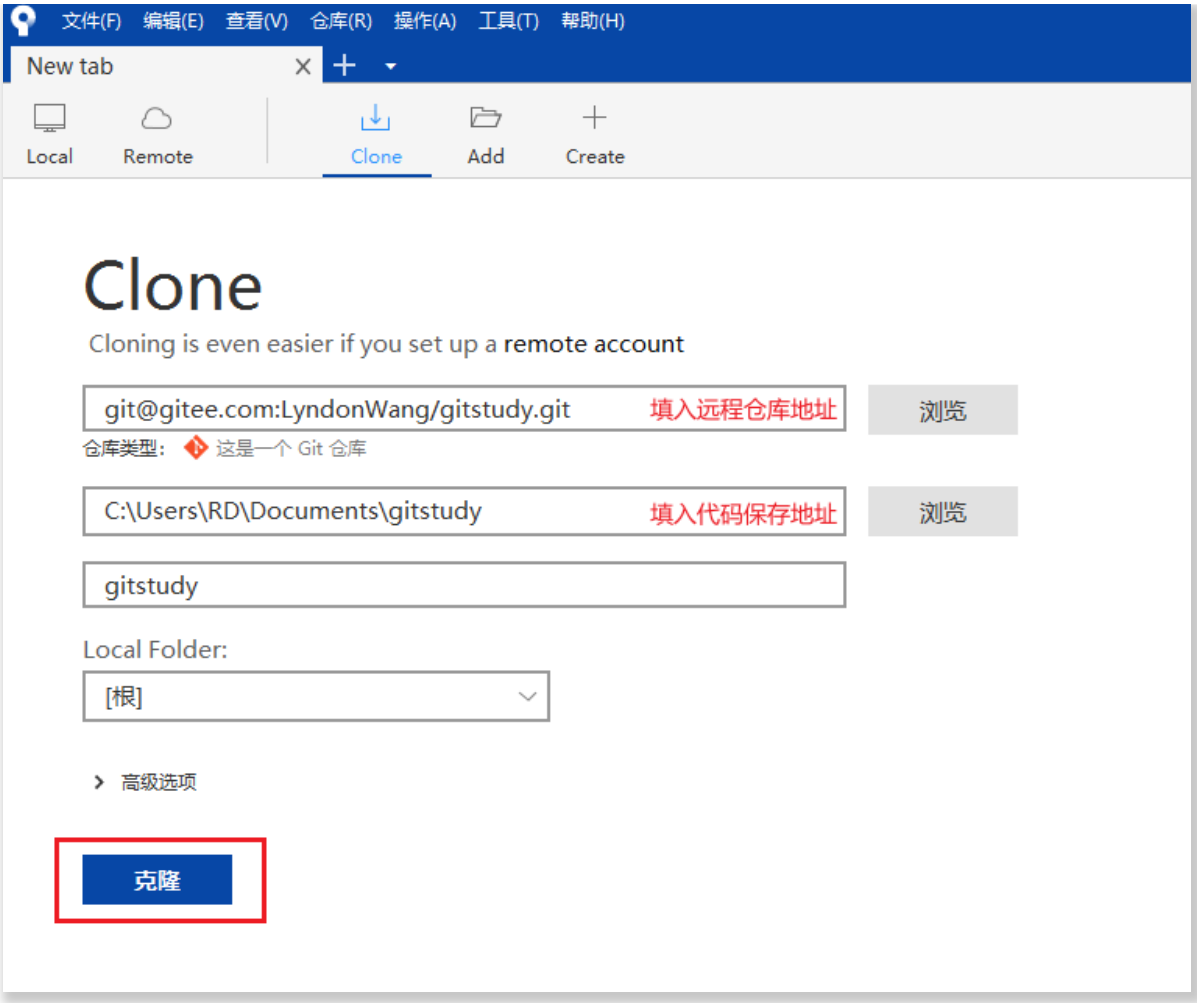
- 确定，提交

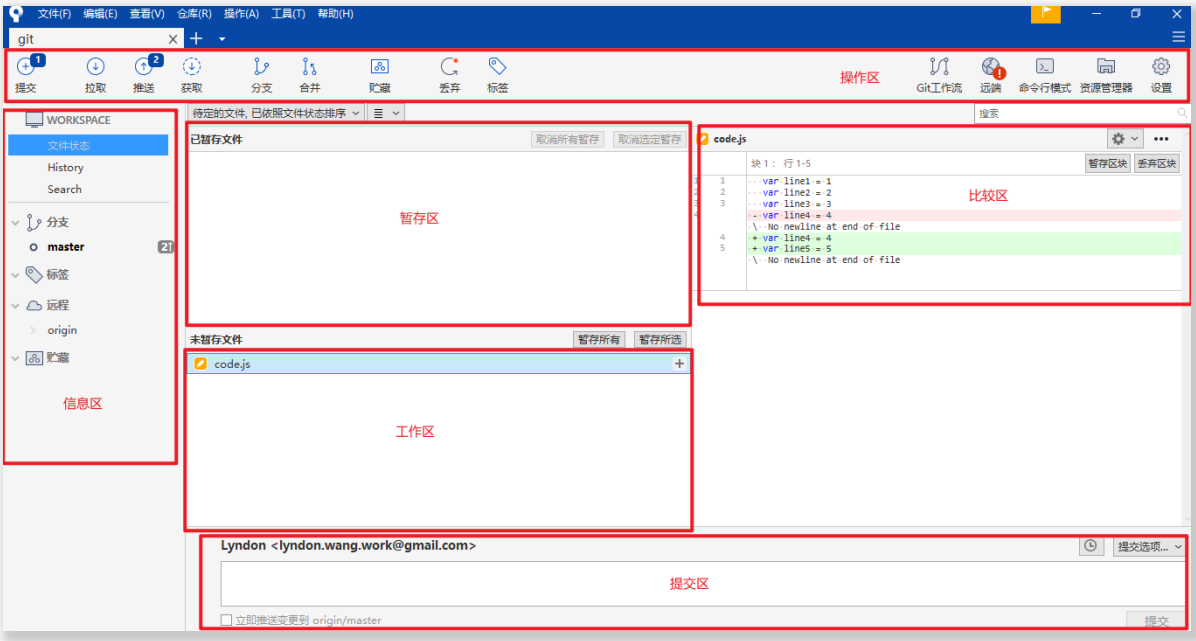


参考： Gitee生成/添加SSH公钥

使用sourcetree克隆远程仓库

点击clone克隆远程仓库



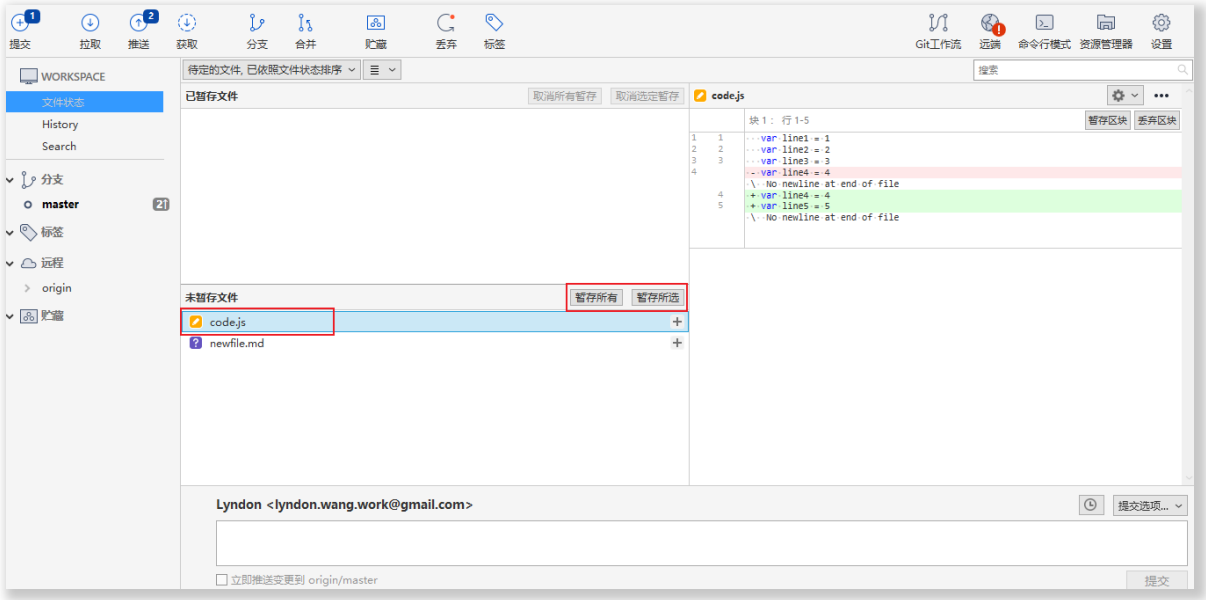


添加到暂存区

使用命令行

```
#添加全部文件
git add .
#添加单个文件
git add file_name
```

使用sourcetree，在未暂存文件列表中选择文件，然后点击《暂存所选》或文件后`+`号按钮，将文件加入已暂存文件列表

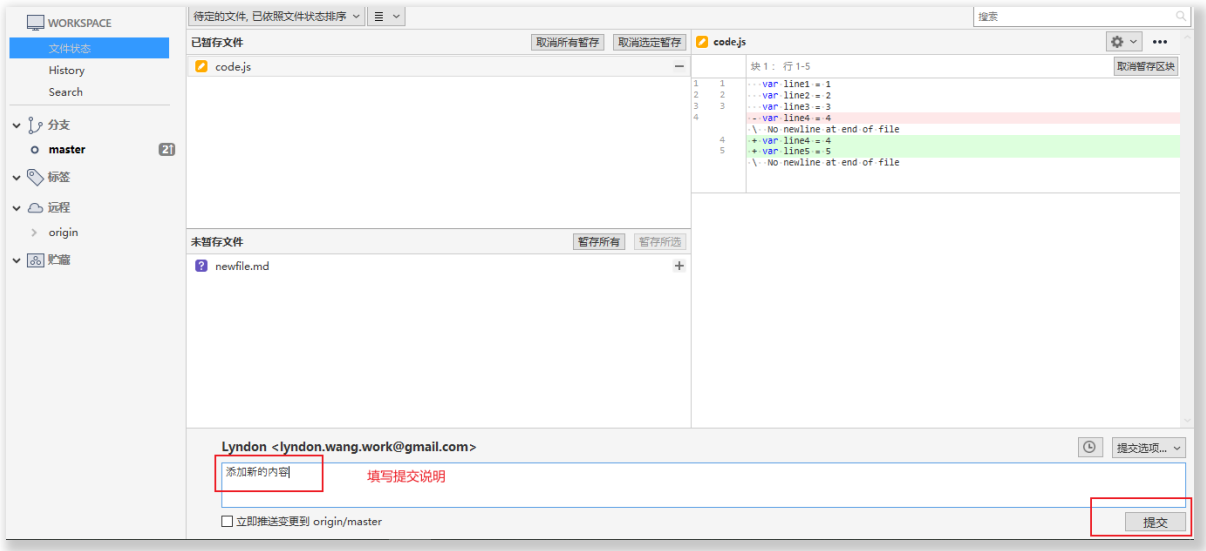


提交到本地仓库

使用命令行

```
git commit -m "提交记录"
```

使用sourcetree



添加远程仓库地址

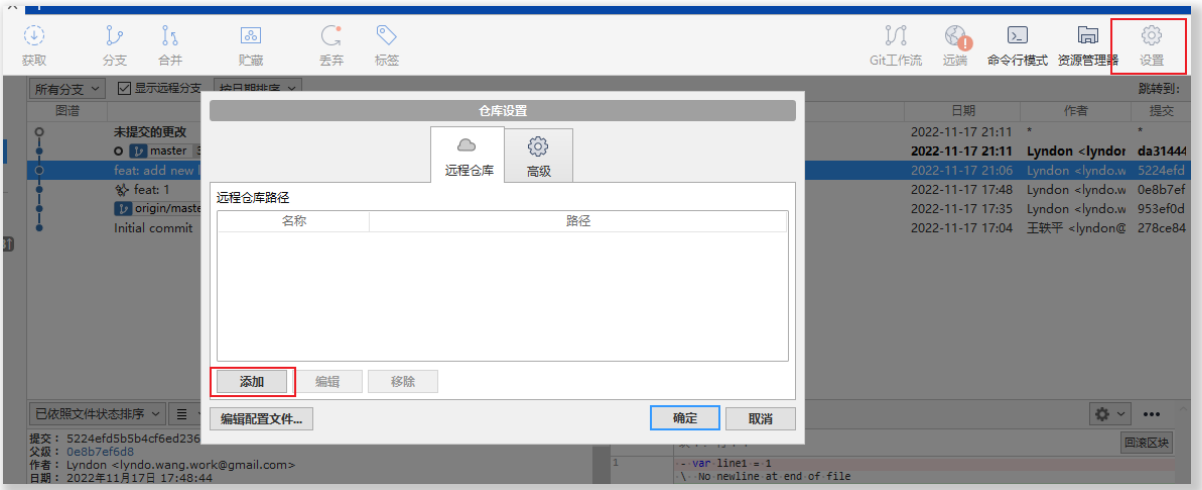
先有本地仓库，后有远程仓库的情况下需要添加远程仓库地址以便推送到远程服务器。

直接从服务器克隆的情况下，不需要添加远程仓库地址，默认即克隆的地址。但一个仓库可以添加多个远程仓库地址。

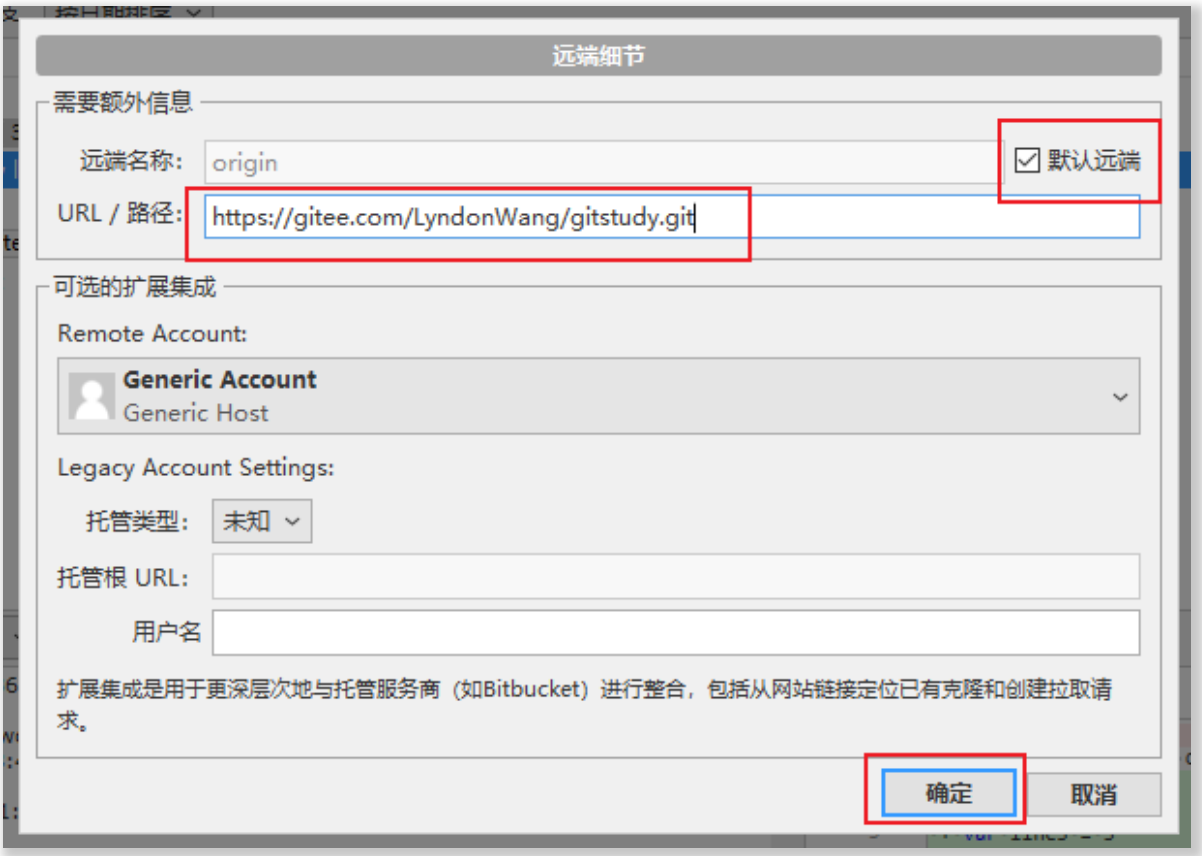
使用命令行

```
git remote add origin https://www.gitee.com/youname/yourepo
```

使用sourcetree，点击《设置》-《添加》



填写远端名称及远程仓库地址，确定即可

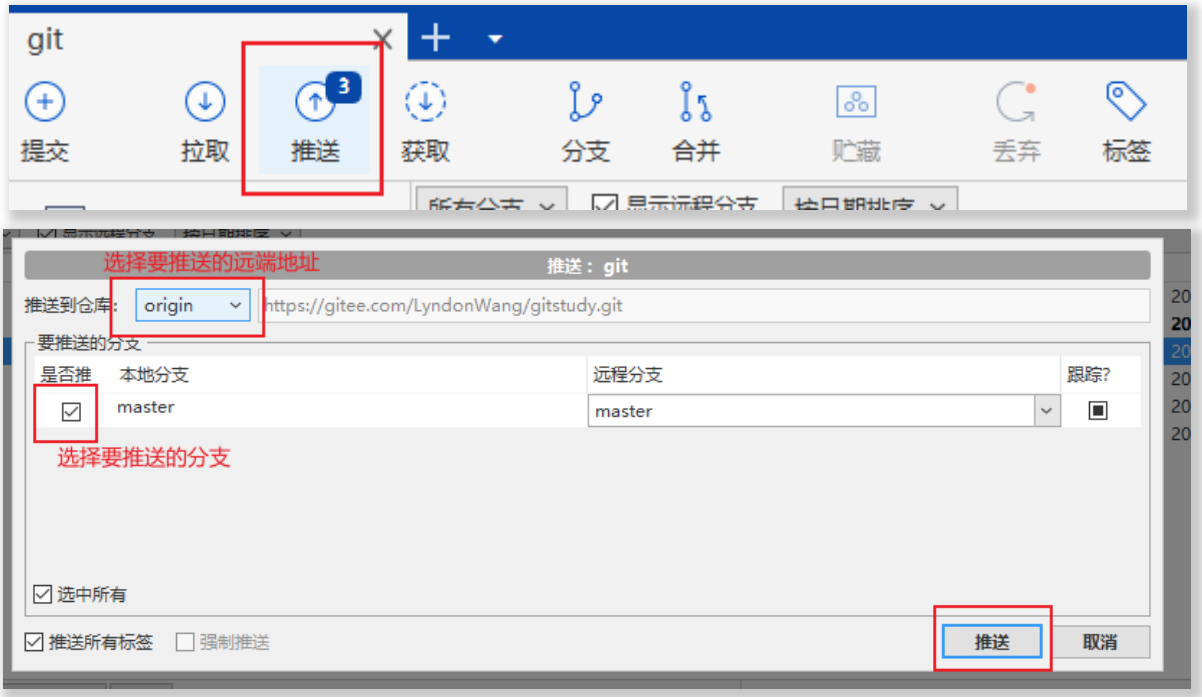


推送到远程仓库

使用命令行

```
git push origin master
```

使用sourcetree，点击《推送》按钮，选择要推送的远端地址及分支

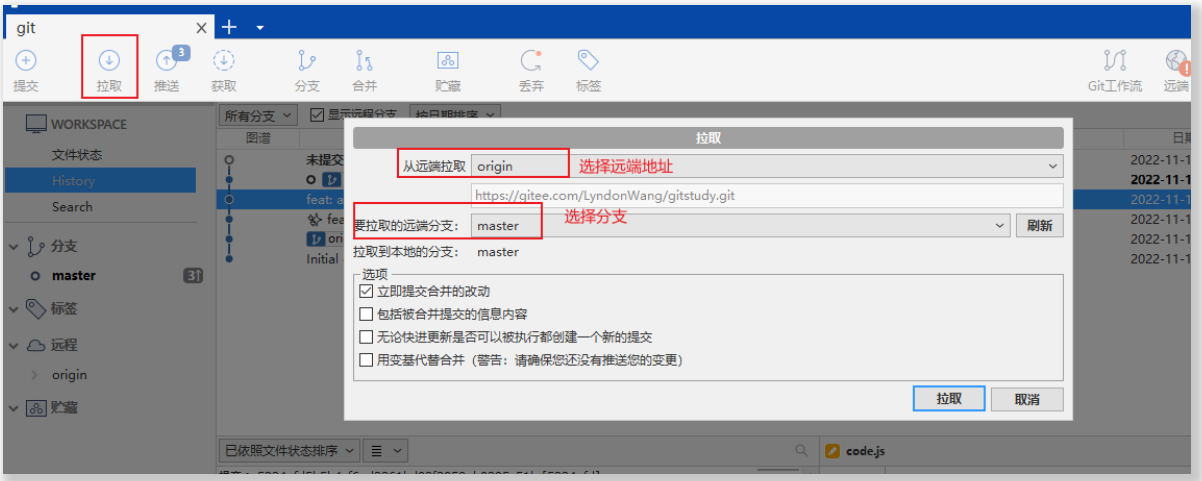


从远程仓库拉取

使用命令行

```
git pull origin master
```

使用sourcetree，点击《拉取》按钮，选择要拉取的远端地址及分支

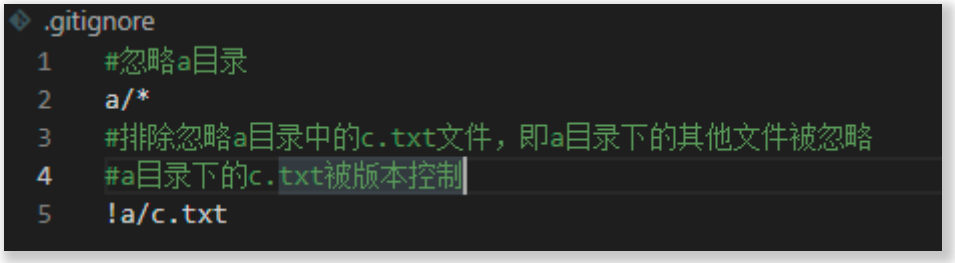


.gitignore文件

记录需要忽略的文件及目录，支持通配符。每个目录下都可以有自己的.gitignore文件。

忽略排除

先忽略目录，然后使用!排除忽略目录中的部分文件



保留目录，忽略所有文件

```
*
!.gitignore
```

标签Tag

每一次提交都会有一个commit id，标签就是commit id的别名，用来标记某一次提交。

标签分类

- 轻量标签

  仅仅只是一个别名

- 附注标签

  有标注者的名字，邮件，日期，标注信息等内容。

创建标签

使用命令行

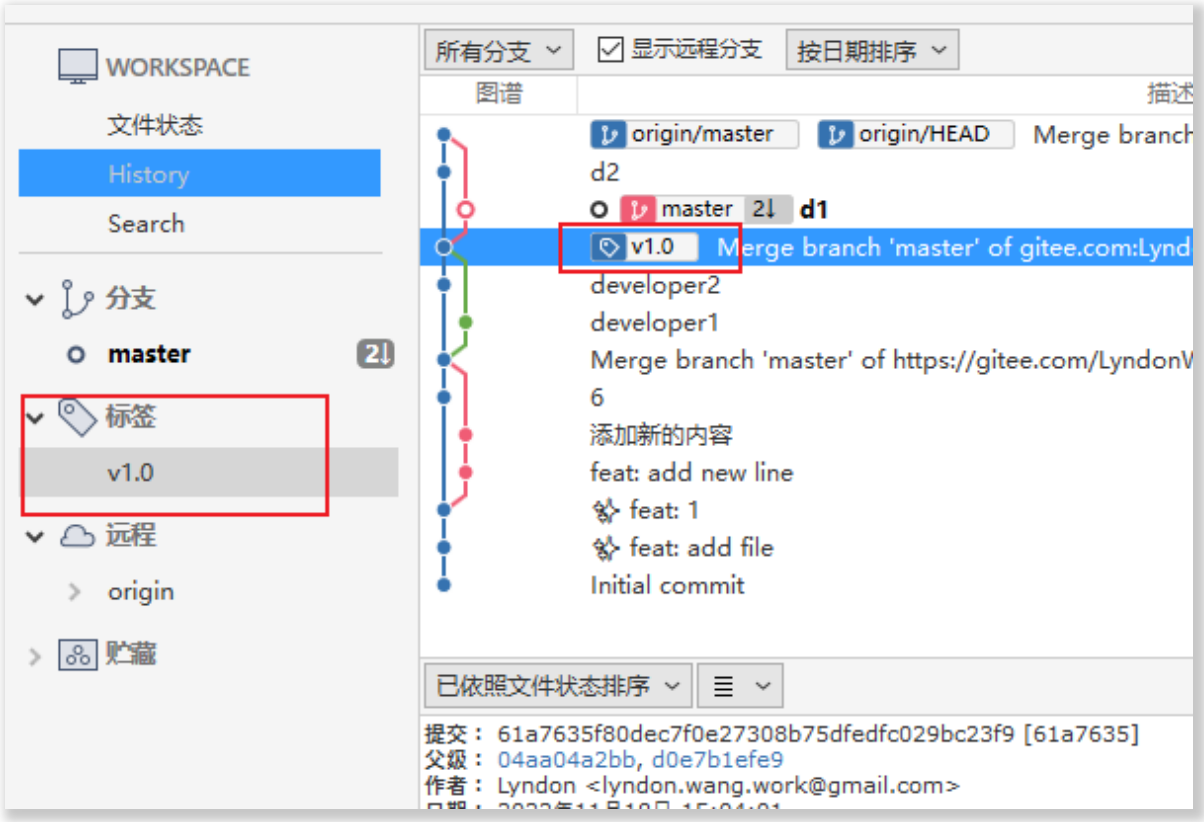
```
#创建轻量标签
git tag 标签名 [提交id]
#创建附注标签
git tag -a 标签名 [提交id] -m "附注信息"
```

查看所有标签

使用命令行

```
#查看标签列表
git tag

#查看单个标签信息
git show 标签名
```

使用sourcetree，在左侧信息区有标签列表，可查看所有标签



推送标签到远端

使用命令行

```
git push orgin 标签名
```

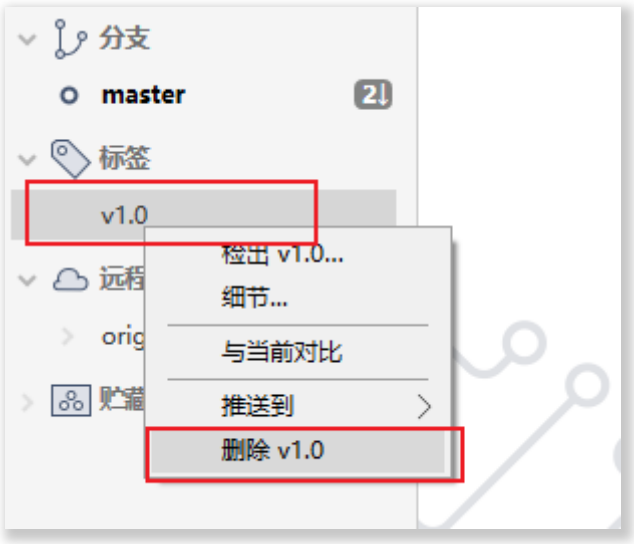
使用sourcetree，在标签名上点击右键，选择推送到...

删除标签

使用命令行

```
#删除本地标签
git tag -d 标签名

#删除远程标签
git push orgin --delete 标签名
```

使用sourcetree，在标签名上点击右键，选择删除



多人协作

提交的原则

- 提交的修改要尽量小，不要多个不同的修改一起提交
- 提交的修改要完整，未完成的工作不可提交到远端
- 提交的说明要规范清晰
- 尽量早的提交，尽量多的提交
- 不要随意推送你的分支到远端，除非你要与人分享
- 不要修改远端的提交历史(不要在远程分支上进行rebase)

Git commit message提交说明规范

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

- **type用于说明 commit 的类别。**
  feat：新增功能
  fix：bug 修复style：不影响程序逻辑的代码修改(修改空白字符，格式缩进，补全缺失的分号等，没有改变代码逻辑)
  docs：文档更新
  refactor：重构代码(既没有新增功能，也没有修复 bug)
  perf：性能, 体验优化
  test：新增测试用例或是更新现有测试
  build：主要目的是修改项目构建系统(例如 glup，webpack，rollup 的配置等)的提交
  ci：主要目的是修改项目继续集成流程(例如 Travis，Jenkins，GitLab CI，Circle等)的提交
  chore：不属于以上类型的其他类，比如构建流程, 依赖管理
  revert：回滚某个更早之前的提交

  wip: 工作中，未完成

- **scope（可选）**
  scope用于说明 commit 影响的范围，比如数据层、控制层、视图层等等，视项目不同而不同。也可以填写文件名。

- **subject（必填）**
  subject是 commit 目的的简短描述，
  - 以动词开头，使用第一人称现在时，比如change，而不是changed或changes
  - 第一个字母小写
  - 结尾不加句号（.）
- **Body（可省）**
  Body 部分是对本次 commit 的详细描述，可以分成多行。
  - 使用第一人称现在时，比如使用change而不是changed或changes。
  - 应该说明代码变动的动机，以及与以前行为的对比。
- **Footer（可省）**
  Footer 部分只用于两种情况。
  1）不兼容变动
  如果当前代码与上一个版本不兼容，则 Footer 部分以BREAKING CHANGE开头，后面是对变动的描述、以及变动理由和迁移方法。
  2）关闭 Issue
  如果当前 commit 针对某个issue，那么可以在 Footer 部分关闭这个 issue 。

  ```
  Closes #234
  ```

  也可以一次关闭多个 issue 。

  ```
  Closes #123, #245, #992
  ```

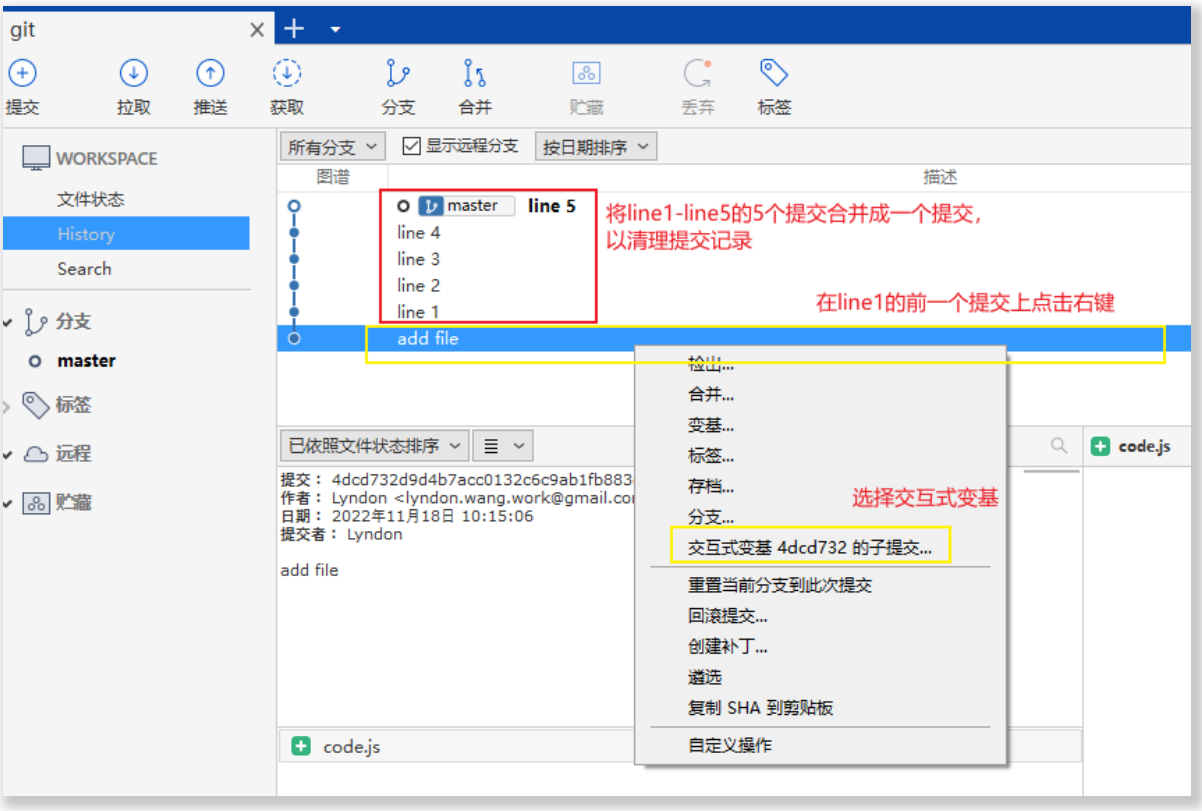
- **Revert**
  还有一种特殊情况，如果当前 commit 用于撤销以前的 commit，则必须以 revert: 开头，后面跟着被撤销 Commit 的 Header。

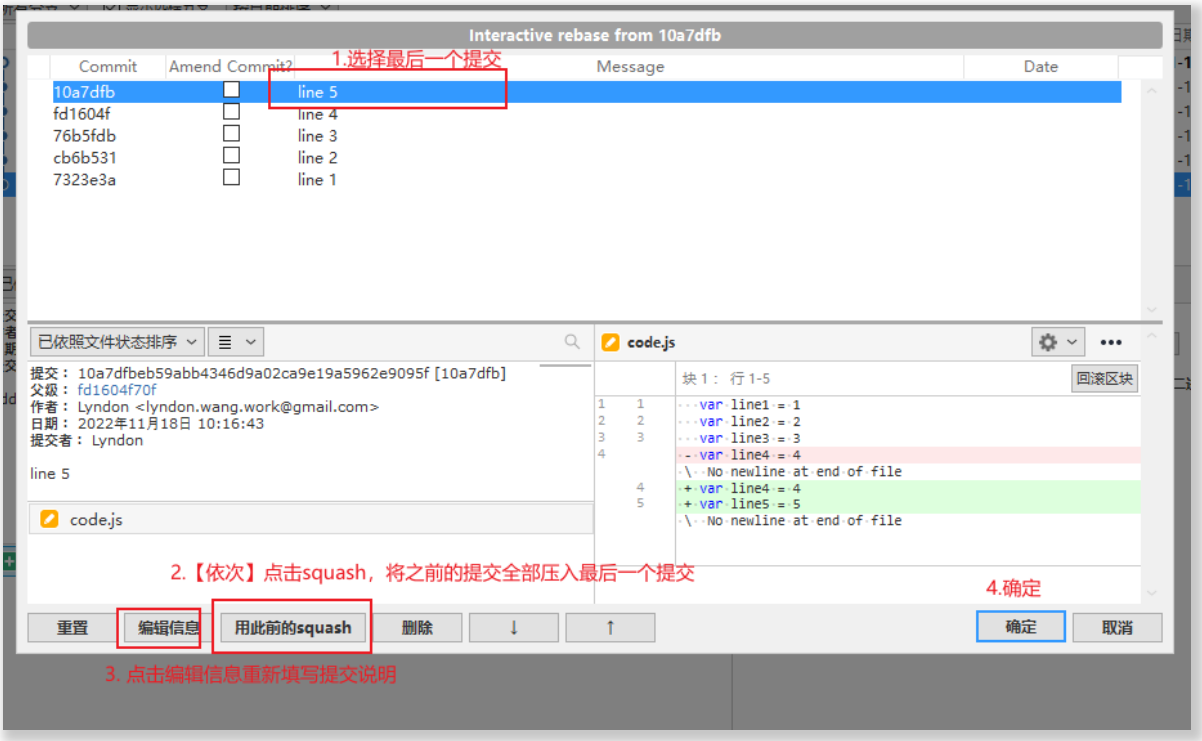
  ```
  revert: feat(pencil): add 'graphiteWidth' option

  This reverts commit 667ecc1654a317a13331b17617d973392f415f02.
  ```

Body部分的格式是固定的，必须写成 This reverts commit <hash>.，其中的 hash 是被撤销 commit 的 SHA 标识符。

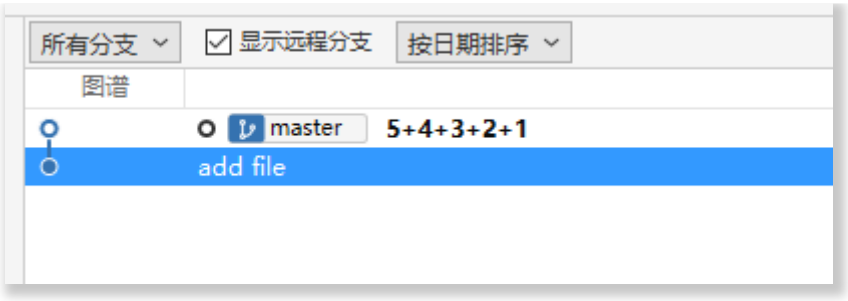
合并多个提交

将多个提交合并成一个提交，主要用于简化提交历史





合并之后



注意： 合并的提交应该是本地尚未push的提交。

分支管理

- 什么是分支?

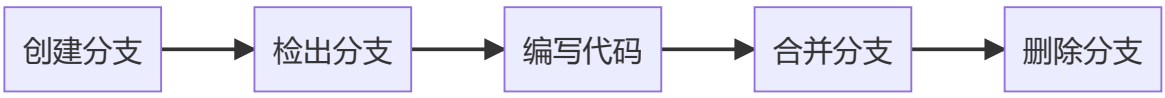
  分支是当前代码状态的平行宇宙。

  分支是指向提交的指针。

- 为什么要使用分支?

  防止正在进行的工作被影响。

基本使用流程

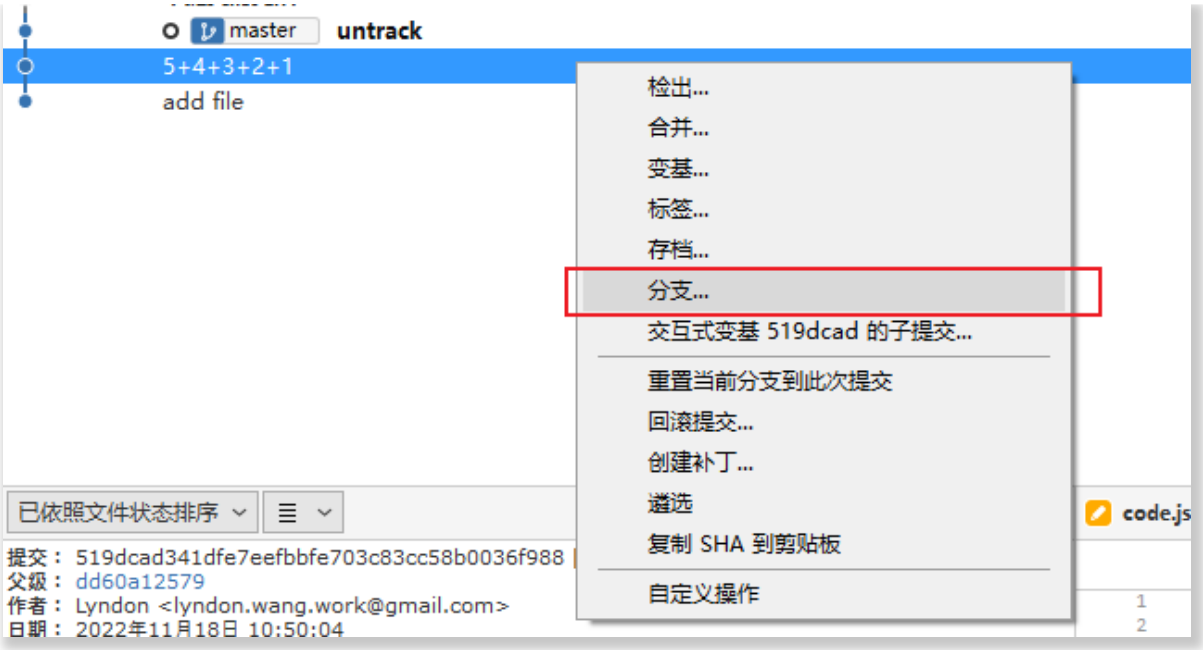


创建分支

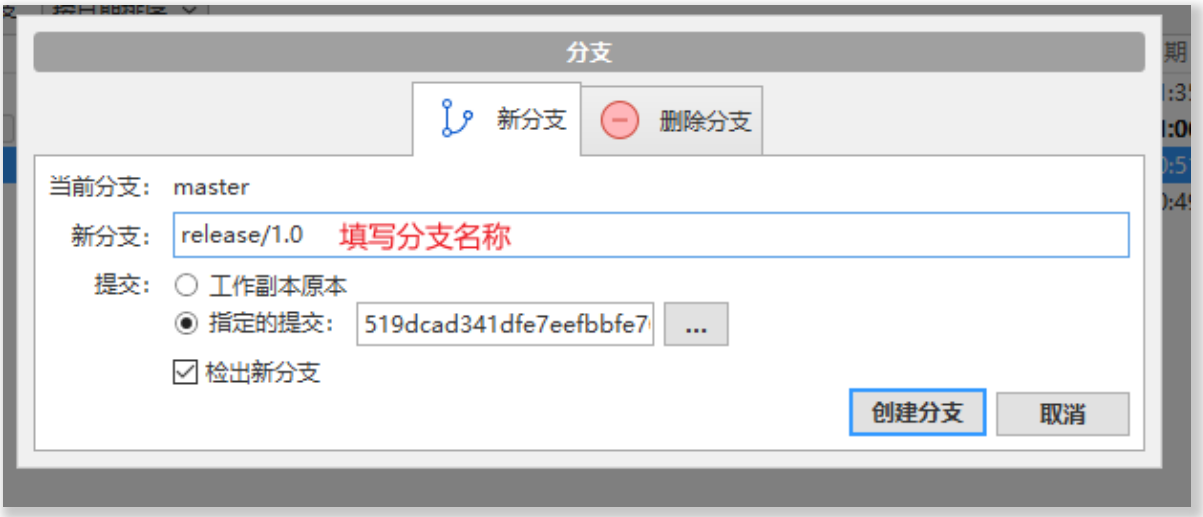
使用git命令行

```
git branch 分支名称
```

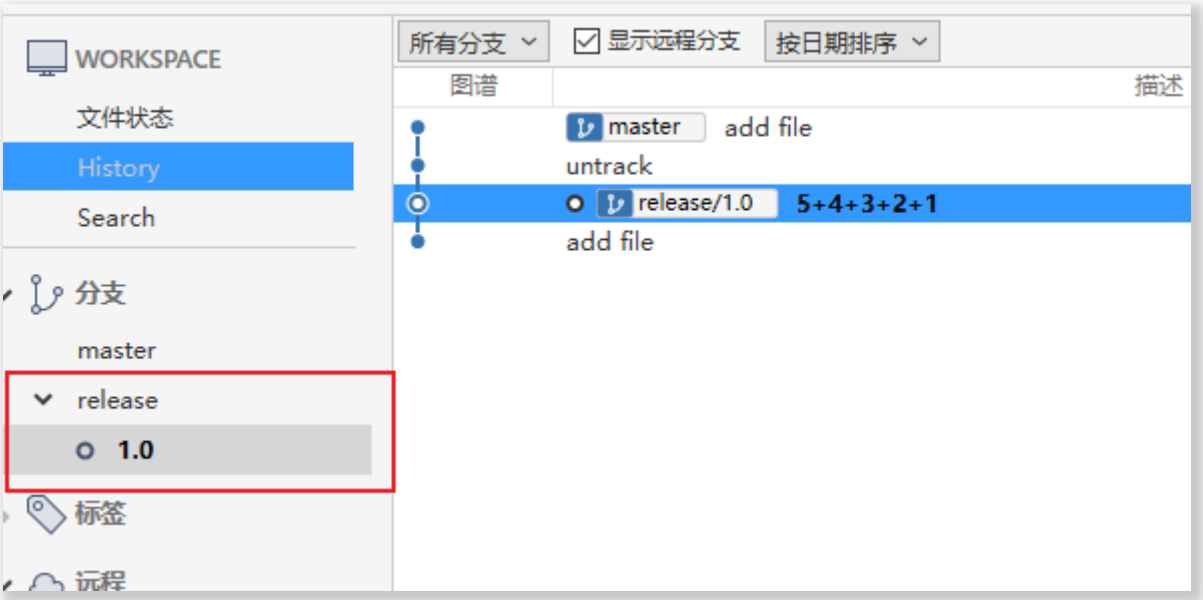
使用sourcetree，在需要分支的提交上点击右键，选择分支...



添加分支名称，点击创建分支即可



完成后，左侧出现分支信息



查看分支

使用命令行

```
#查看本地分支
git branch
#查看远程分支
git branch -r
#查看本地及远程分支
git branch -a
```

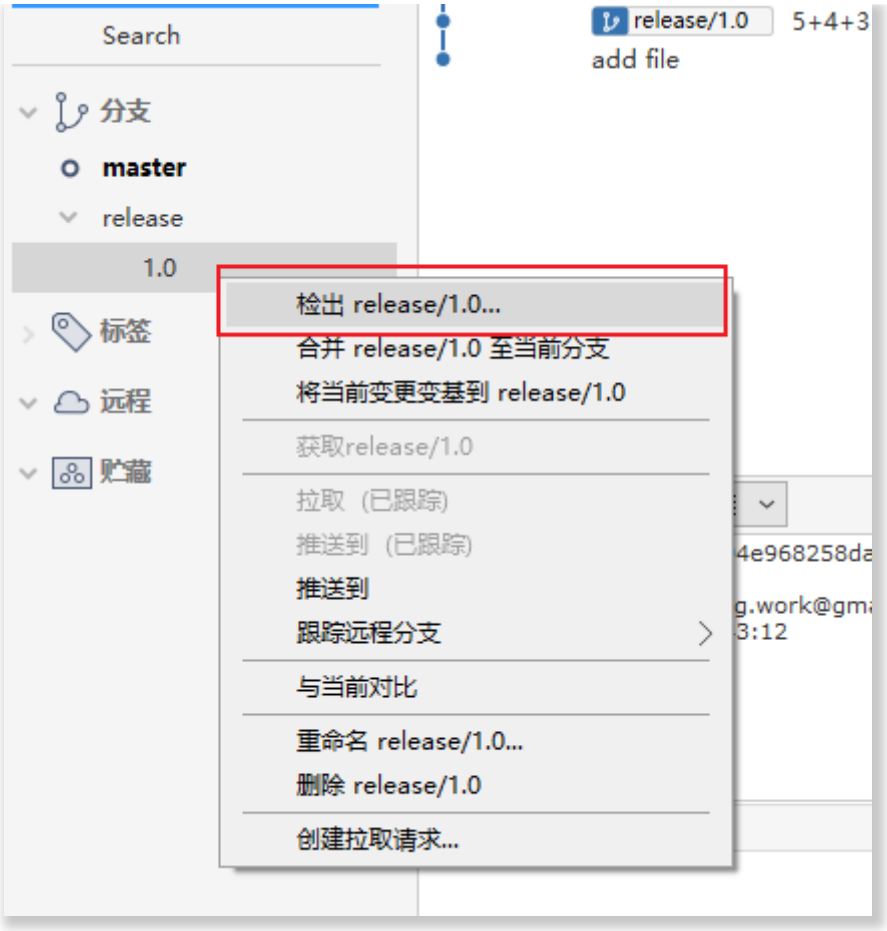
使用sourcetree，左侧信息区可以显示所有本地及远程分支

检出分支

使用命令行

```
git checkout 分支名称
```

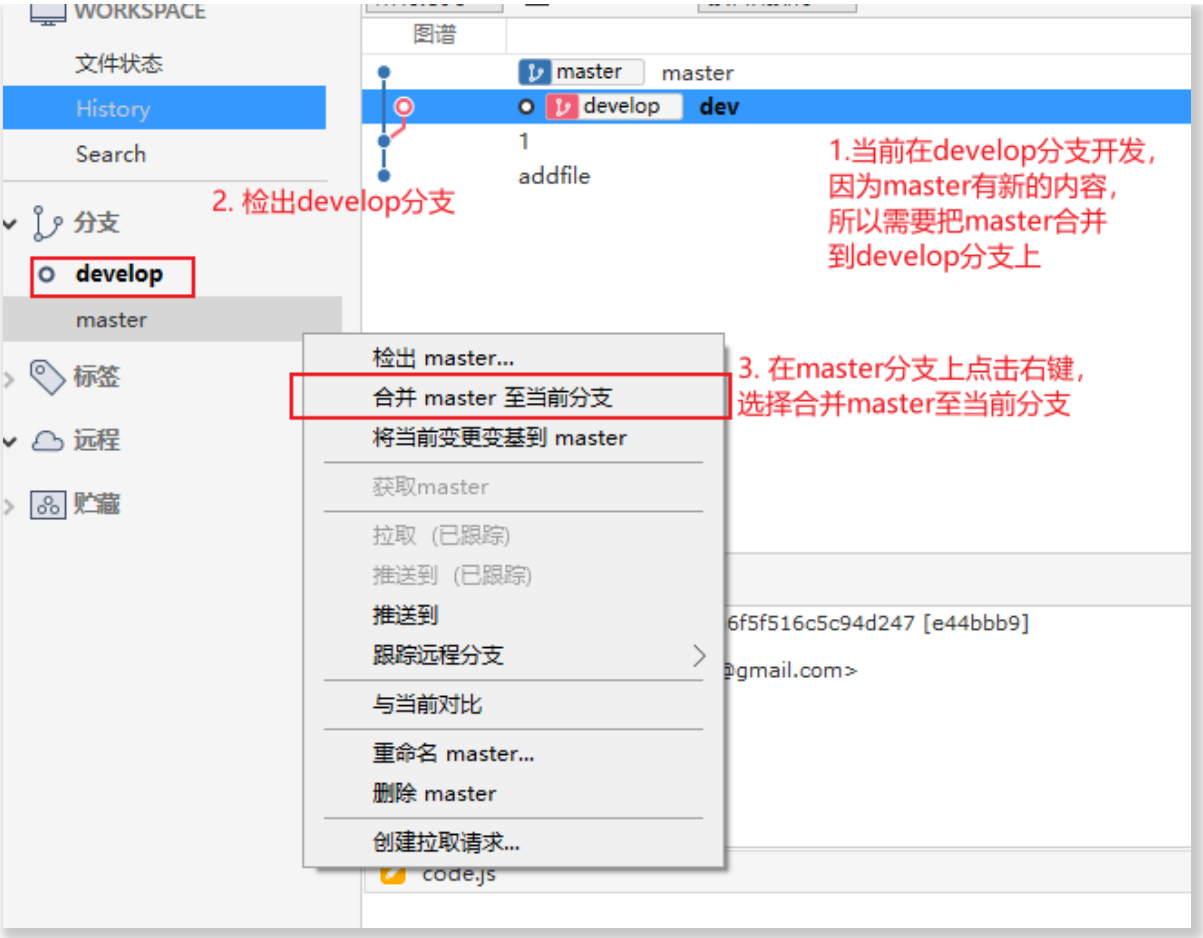
使用sourcetree，在分支上点击右键，选择检出...



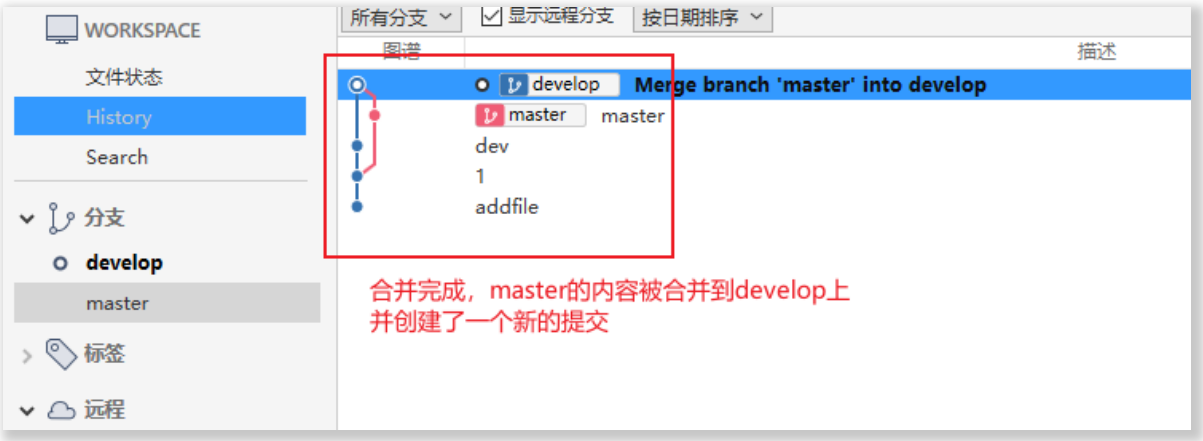
合并分支

合并分支可以使用合并`merge`或变基`rebase`命令

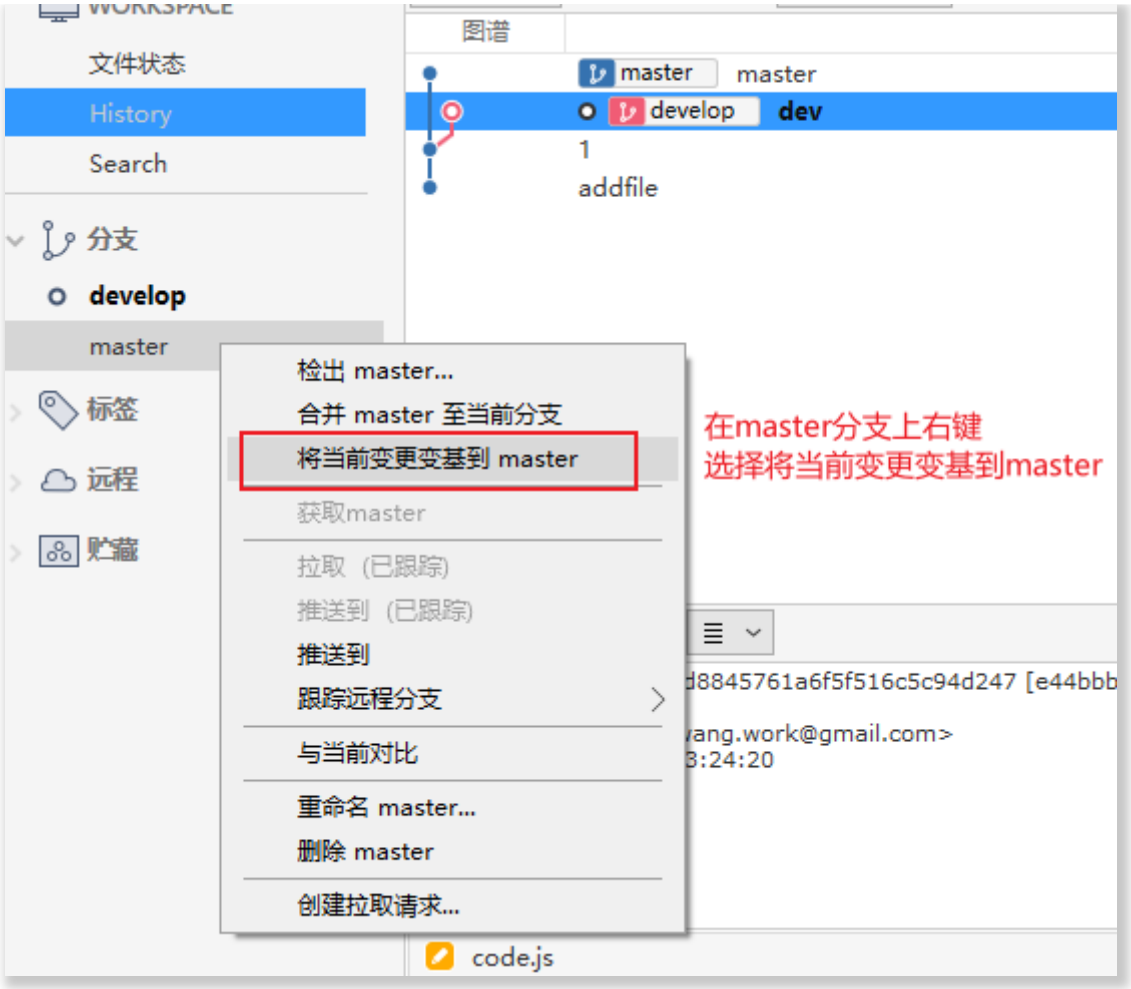
使用merge



合并完成之后，创建了新的提交



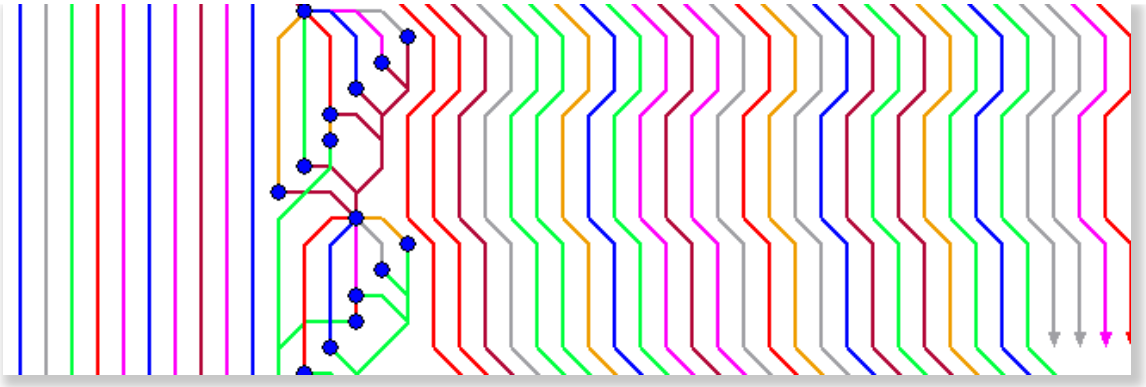
使用rebase



合并完成之后，develop的起点发生变化，即改变了基点，简称变基。

merge优缺点：

- 使用简单，易于理解。
- 保持源分支的原始上下文。
- 源分支上的提交与其他分支的提交是分开的。
- 可以保留提交历史。
- 乱



rebase优缺点：

- 代码历史是简化的、线性的、可读的。
- 与许多独立的特性分支的提交历史相比，操作单个提交历史更容易。
- 干净、清晰的提交信息可以更好地跟踪一个bug或何时引入的某个功能。可以避免众多的单行提交污染历史。
- 会更改历史提交时间，可能会丢失上下文。

如何选择?

推荐使用合并，只在本地分支上使用变基。

> 例如现有上游分支 master，基于 master 分支拉出来一个开发分支 dev，在 dev 上开发了一段时间后要把 master 分支提交的新内容更新到 dev 分支，此时切换到 dev 分支，使用 `git rebase master`
>
> 即：本地分支合并上游分支时，使用rebase

> 等 dev 分支开发完成了之后，要合并到上游分支 master 上的时候，切换到 master 分支，使用 `git merge dev`
>
> 即：上游分支合并本地分支时，使用merge

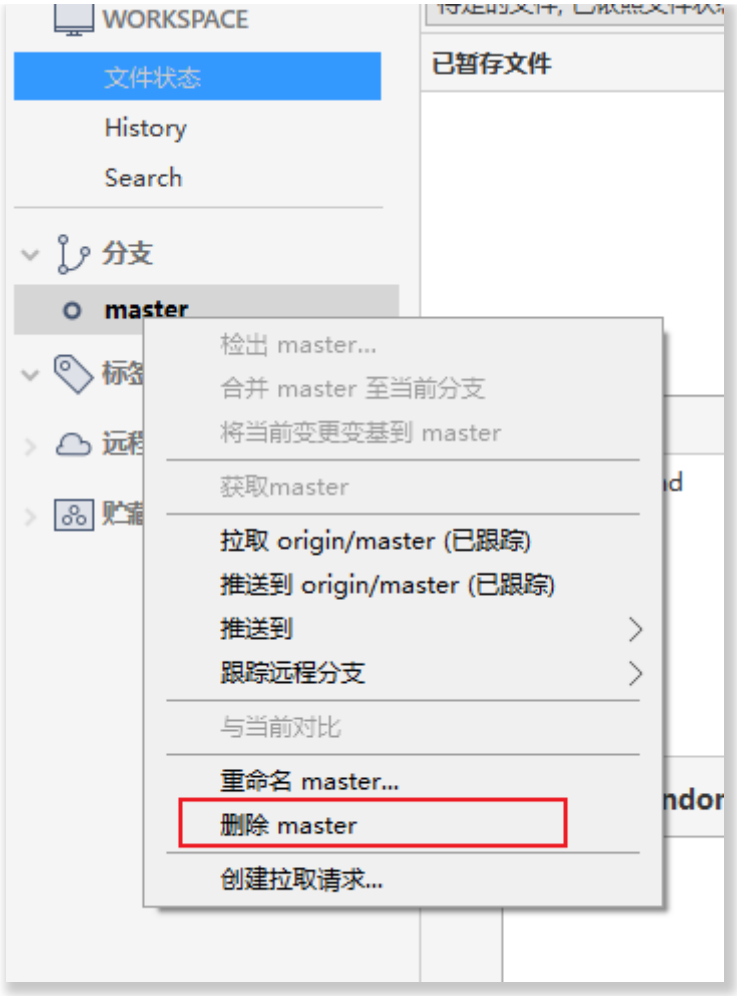
参考：Merge vs Rebase

删除分支

使用命令行

```
#删除本地分支
git branch --delete 分支名称
#强制删除本地分支
git branch --delete-force 分支名称
#删除远程分支
git push origin --delete 分支名称
```

使用sourcetree，在分支上点击右键，选择删除

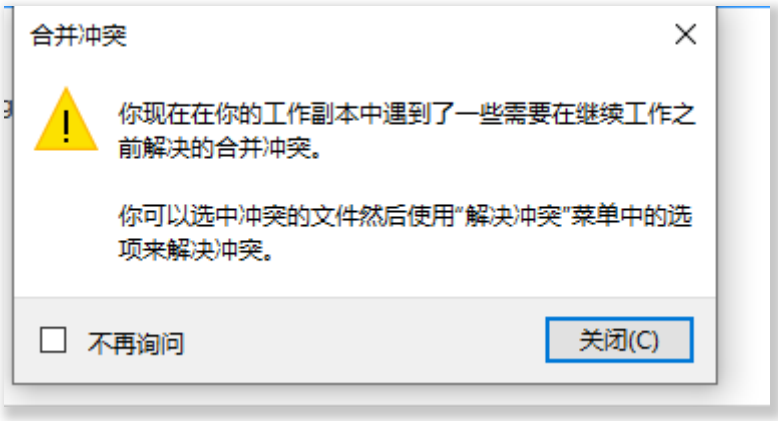


冲突解决

当本地和远程代码在同一个地方不一致时，git不知道应该使用哪一个代码，这个时候就产生了冲突。

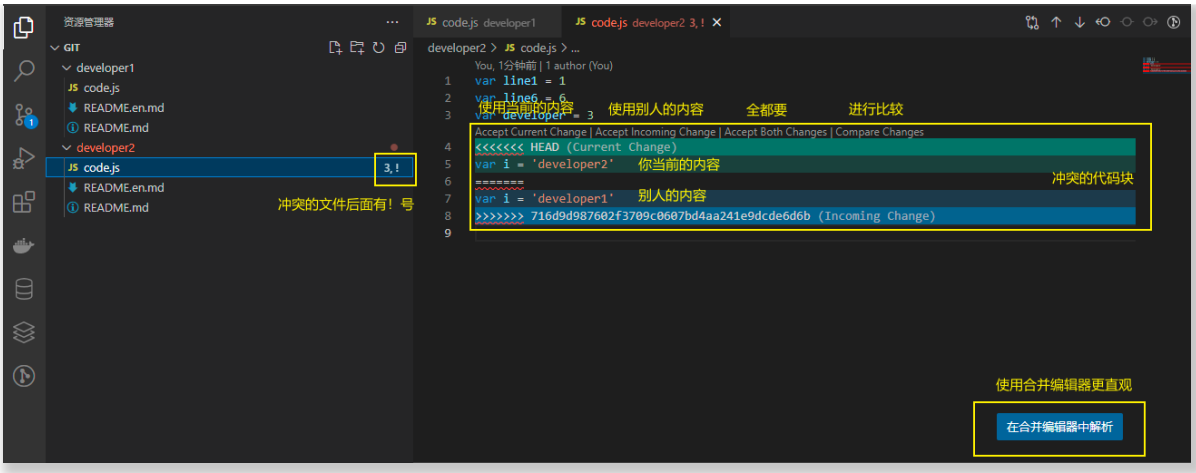
冲突在pull，push，merge，rebase等环节都有可能产生。

发生冲突时sourcetree会弹出提示，关闭对话框，打开vscode，查看冲突的文件

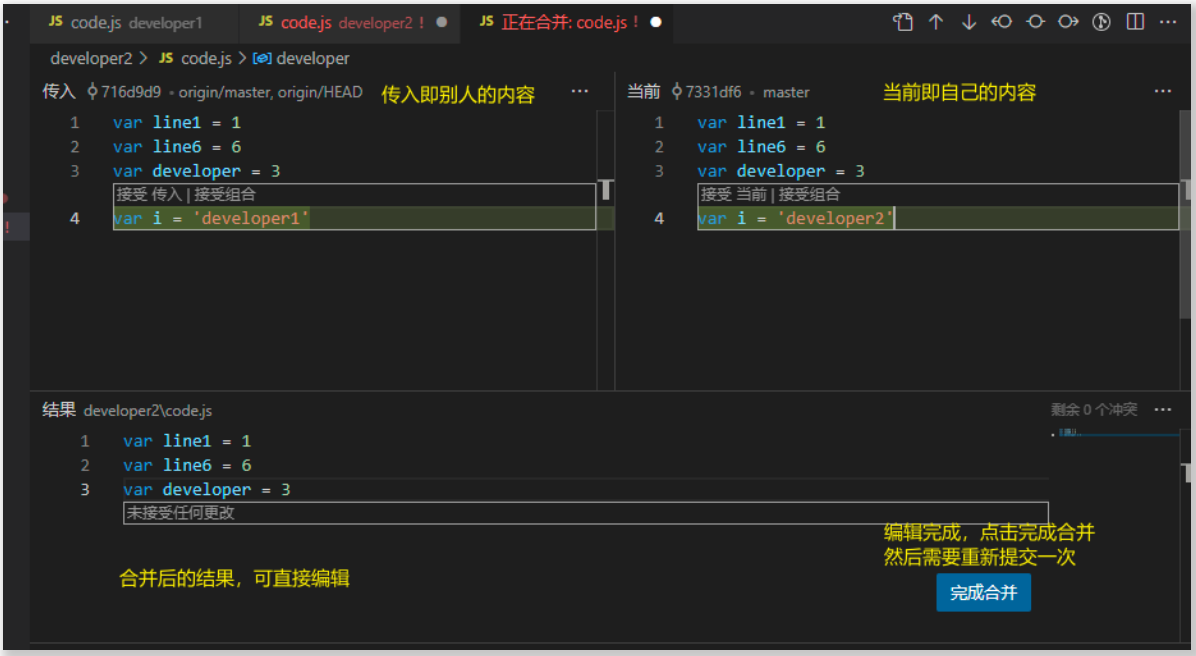


在vscode中，冲突的文件最后有[!]感叹号，表示此文件处于冲突状态。

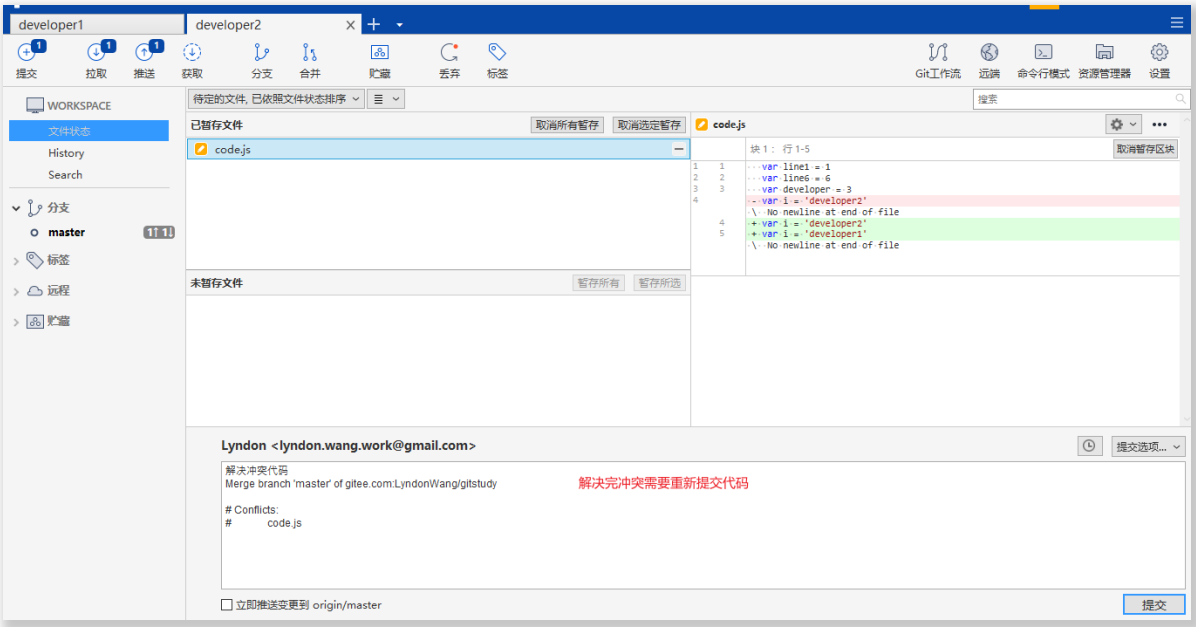
文件中，冲突的代码块以[<<<<<<HEAD]开头，表示冲突前当前文件的内容，中间以====分割，下面是其他人在此处修改的内容，然后以[>>>>>>对方的提交id]结束。



合并编辑器界面



解决冲突后需要重新提交一次。然后再次推送。

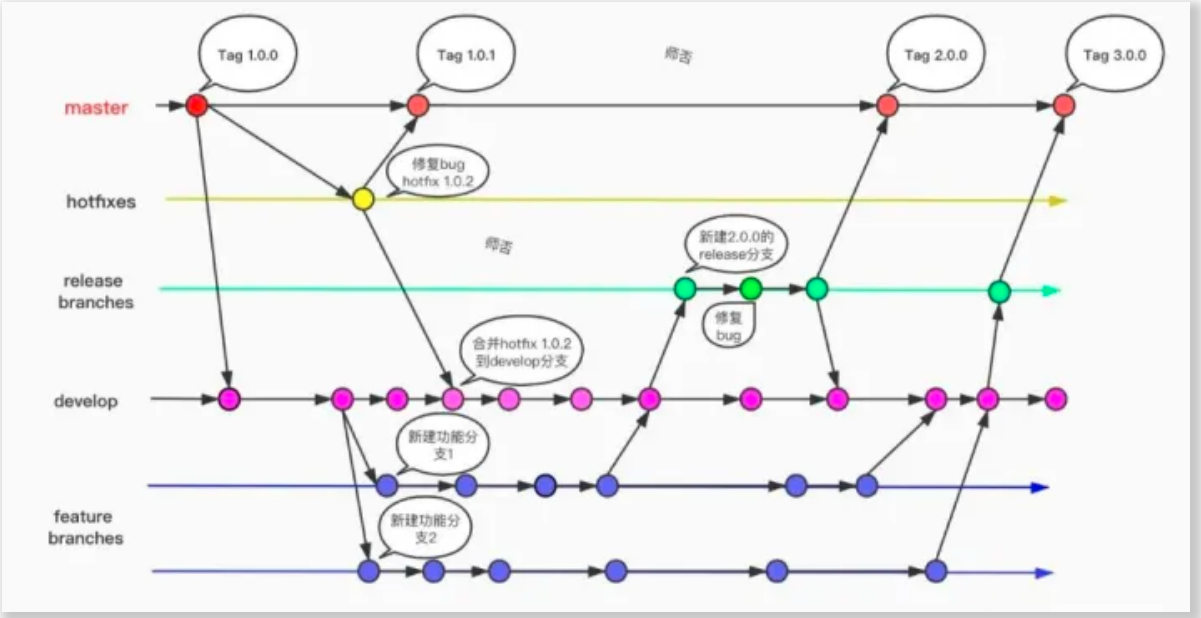


工作流

在实际工作中，围绕git产生了一系列工作流程，称为工作流。主要的工作流方式有：gitflow，githubflow，gitlabflow等等。

GitFlow

常用于非开源项目，适合于具有计划性发布周期的项目。 如公司内部项目。



分支说明

master 用于存档，不用于直接提交

develop 用于开发集成

feature 用于开发

release 用于发布

hotfixes 用于线上修复

流程

1. 从master拉取develop分支
2. 开发人员开发时，从develop分支拉取feature分支进行开发
3. 开发结束后合并到develop分支
4. 从develop分支拉取release分支用于发布
5. release分支测试、修复完毕后上线，并合并到master、develop分支
6. 在master分支为当前发布版本做标签
7. 线上修复时，从master对应线上版本拉取hotfixes分支
8. 修复后合并到master、develop
9. 重新生成release发布，或生成patch补丁线上应用

优缺点

结构清晰

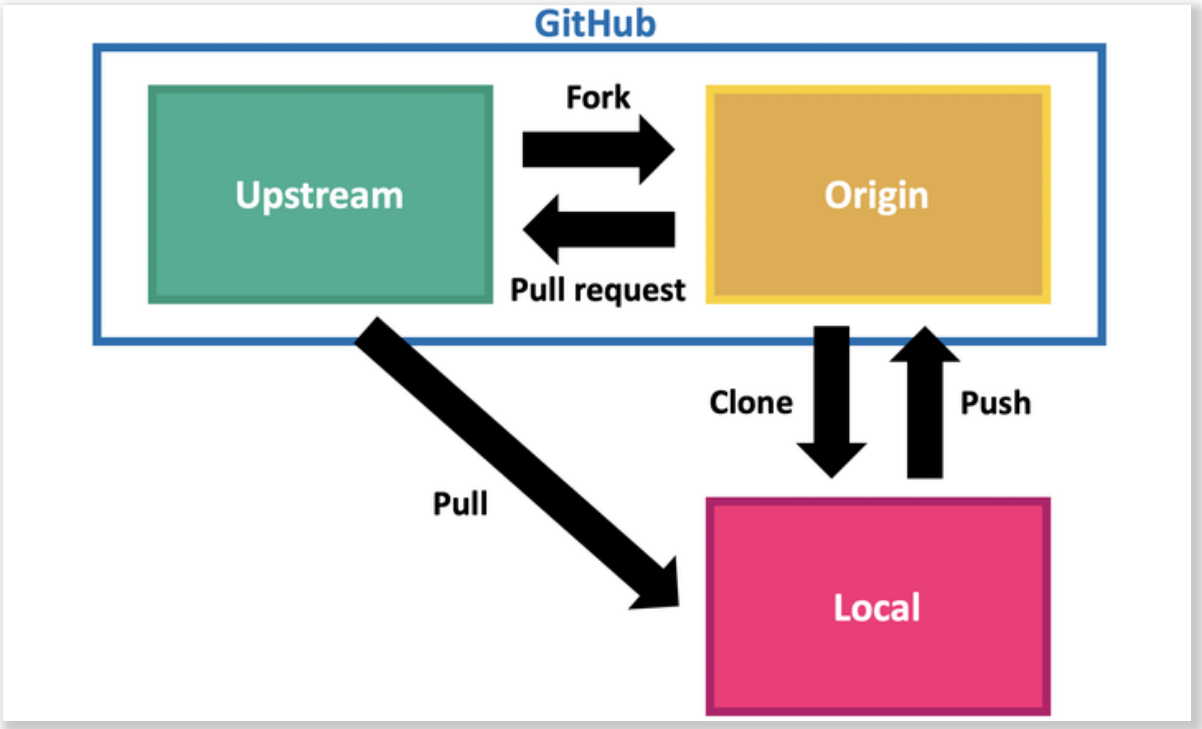
需要维护master，develop两个长期分支，较复杂

分支较多，需要注意切换分支，避免出错

发布比较频繁的项目中，master和develop区别不大，重复维护

GitHub Flow

多用于开源项目，github上的主要流程



术语

- Fork 复制一份上游仓库upstream到自己的远程仓库
- Pull request 拉取请求，请求上游仓库合并自己的修改
- Code review 代码审查，检查代码是否符合规范
- CI 持续集成，主要用于代码自动化测试

流程

1. Fork上游代码仓库到自己的远程仓库
2. 将自己的远程仓库拉取到本地开发、修复
3. 开发完成后推送到自己的远程仓库
4. 如果过程中上游仓库有修改，可直接拉取到本地
5. 向上游仓库发起PR，等待Code review
6. CI通过后，上游仓库合并

优缺点

相对简单，只有一个master分支

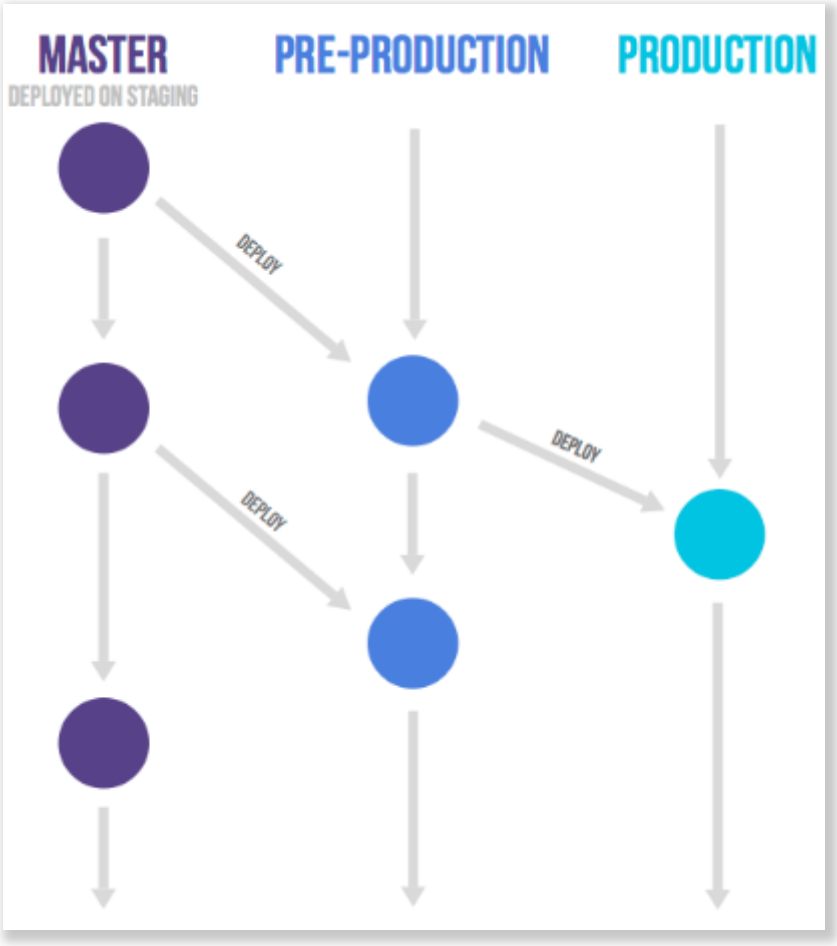
适合持续发布的产品

对于较复杂的发布环境，一个master分支略显不足

GitLab Flow

其他流程和GitHub Flow一致，区别只是在于发布方面，弥补了GitHub Flow的不足。

确定了”上游优先“的原则，只有上游版本更新后，才发布到其他分支。



练习网站

https://learngitbranching.js.org/?locale=zh_CN

学习学习

- 阅读官方文档

  官方文档最新，最全面，最完整，**第一查询对象应该是官方文档**

  先粗读一遍了解大致内容，再根据需要细读具体内容

- 刻意练习

  一万小时定律: 1万小时的锤炼是任何人从平凡变成世界级大师的必要条件

  但1万小时的简单重复并不能让你成为大师。你需要刻意的练习

  离开舒适区，不断提高练习难度，尝试没有做过的事，做过的事做得更好

  可以尝试的方向：

  - 如何对用户更友好
  - 如何减少对客户端/服务器的压力
  - 如何提高扩展性，健壮性，可用性，安全性

    - 如何提高效率
- 问题解决

  正确的提问

  将问题分解至你能解决的程度

  不知道不知道怎么办?
- 正确使用搜索引擎

  优先级：谷歌、bing、百度

  搜索内容：问题领域 关键词

  指定域名搜索： site:域名 关键词

  完全匹配搜索： “关键词”

  通配符： 关键*词

  排除： -关键词

  指定文件类型： filetype:pdf 关键词