



南開大學
Nankai University

网络空间安全学院

软件工程实验

实践课题：项目管理

学院：网络空间安全学院

年级：2021 级

班级：信息安全一班

学号：2111408

姓名：周钰宸

指导教师：刘健

2024 年 6 月 24 日

目录

1 实验要求	2
1.1 Git 原理简介	2
1.1.1 主要组件和术语	2
1.1.2 Git 支持软件配置管理的核心机理	2
1.2 实验目标	3
1.3 实验环境以及工具	4
1.4 实验完成情况	4
2 安装 Git	4
2.1 本地机器上安装 git	4
2.2 申请 github 帐号	6
2.3 git 本次默认分支重命名	10
3 Git 操作过程	13
3.1 实验场景一：仓库创建与提交	13
3.2 实验场景二：分支管理	31
3.3 实验场景三：远程分支管理	51
4 小结	61
4.1 实验思考	61
4.1.1 git 的优点	62
4.1.2 git 在个人开发和团队开发中作用的差异	63
4.1.3 其它版本控制软件对比	64
4.1.4 git 的适用情况	65
4.1.5 其它思考	66
4.2 实验总结	67
致谢	68

1 实验要求

1.1 Git 原理简介

Git 是一个分布式版本控制系统，用于跟踪文件的更改，并促进多个用户之间的协作开发。Git 的核心是一个支持分支、合并和版本管理的系统，确保代码库的一致性和完整性。

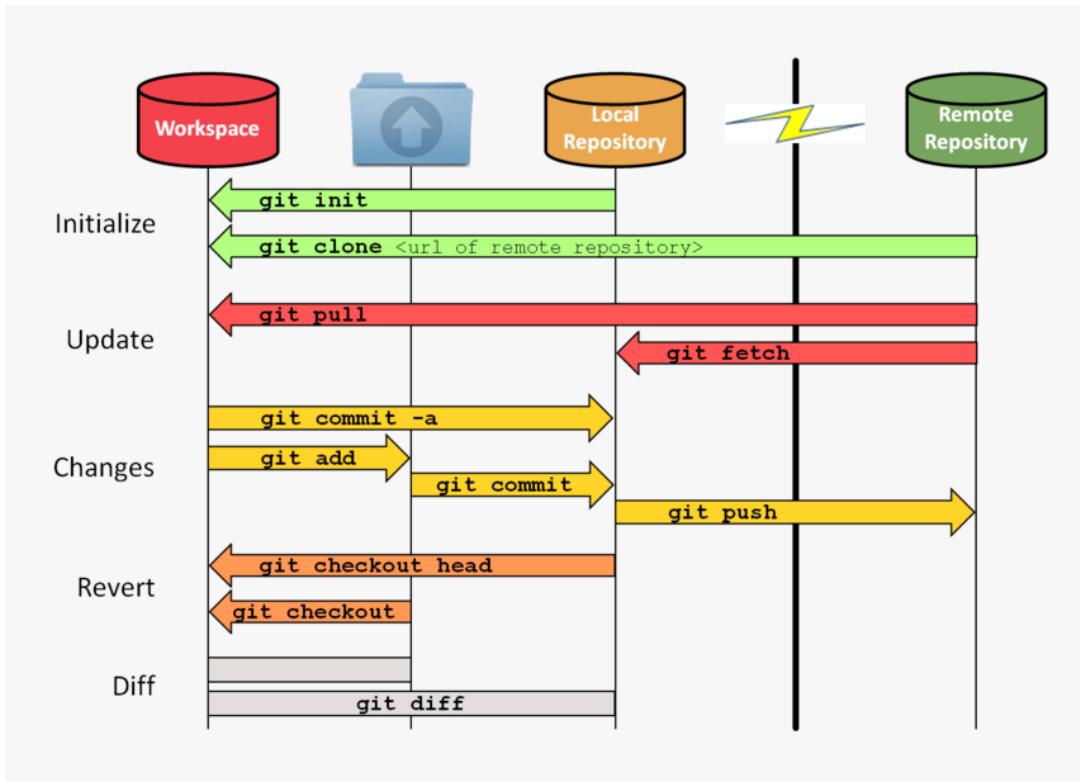


图 1.1: Git 的主要操作和流程

1.1.1 主要组件和术语

- **Workspace (工作区)**：用户实际进行文件修改的地方。
- **Local Repository (本地仓库)**：包含所有分支、标签以及版本历史的本地存储库。
- **Remote Repository (远程仓库)**：一个托管在服务器上的仓库，用于与其他开发者共享代码。

1.1.2 Git 支持软件配置管理的核心机理

1. **分布式架构**: 每个开发者都有一个完整的仓库副本，这确保了即使在没有网络连接的情况下也能进行版本控制和代码管理。
2. **分支和合并**: Git 允许用户创建和管理多个分支，使得不同的开发任务可以并行进行。分支的合并功能使得开发者可以将不同的工作流合并到主分支，确保代码的一致性。
3. **快照存储**: Git 通过保存项目在每次提交时的快照（而不是文件的差异）来跟踪文件的变化。这种方法不仅提高了效率，还提供了更可靠的历史记录。

4. **高效的差异比较:** Git 的 `diff` 功能使得开发者可以轻松查看不同版本之间的差异，从而进行代码审查和质量控制。
5. **冲突解决:** 在多个开发者同时修改同一文件时，Git 提供了冲突检测和解决机制，确保最终版本的完整性和一致性。
6. **版本回退:** Git 允许用户在必要时回退到以前的版本，这对于错误修复和代码审查非常有用。

通过这些核心机理，Git 提供了一个强大的框架来支持软件的配置管理，使得开发团队可以高效、协作地进行开发工作。

如图1.1所示，展示了 Git 的主要操作和流程，包括初始化 (Initialize)，更新 (Update)，提交更改 (Changes)，回退 (Revert) 和差异比较 (Diff)。后续实践实验中会依次通过三个不同的场景：仓库创建与提交，分支管理以及远程分支管理，对上述 Git 的原理，使用方法，以及其支持软件配置管理的核心机理进行更加深刻的体会与理解。

1.2 实验目标

本次软件工程实践是要实现项目中的管理任务，主要通过 **Git** 以及 **Github** 工具完成。

项目管理的重要性

1. **确保项目成功交付:** 项目管理在软件工程中扮演着至关重要的角色，它确保项目能够在**规定时间**、**预算**和**质量**范围内成功交付。通过有效的项目管理，可以避免项目延期、成本超支和质量问题。
2. **资源优化配置:** 通过项目管理，可以优化资源配置，确保每个团队成员的技能和时间被合理利用，从而提高项目的**效率**和**生产力**。
3. **风险管理:** 项目管理有助于识别、分析和应对风险。通过提前规划和风险管理，可以减少项目过程中可能遇到的不确定性和挑战。
4. **提高团队协作:** 项目管理强调沟通和协作，促进团队成员之间的信息共享和协调工作，从而提高整体工作效率和团队凝聚力。
5. **客户满意度:** 通过严格的项目管理流程，可以确保项目按客户的需求和期望交付，提高客户的**满意度**和**忠诚度**。

具体实验目标可以表述为：

- 熟练掌握 git 的基本指令和分支管理指令
- 掌握 git 支持软件配置管理的核心机理
- 在实践项目中使用 git/github 管理自己的项目源代码
- 本次实验由个人单独完成（使用团队作业的文件，但仓库需要个人新建）

1.3 实验环境以及工具

实验平台	Windows 11 家庭中文版
Git 版本	git version 2.42.0.windows.2
GitHub 主页 URL	https://github.com/ErwinZhou
实验场景一和三涉及的项目 URL 地址 1	Software-Engineering-project-management-test
实验场景二和三涉及的项目 URL 地址 2	SE2024-ProjectManagement-test
Git 本地默认分支名	main

表 1: 本次实验环境及工具

1.4 实验完成情况

作业要求	是否要求	个人完成情况
完成三个实验场景下的 R0-R18	√	√
熟练掌握 git 的基本指令和分支管理指令	√	√
掌握 git 支持软件配置管理的核心机理	√	√
在实践项目中使用 git/github 管理自己的项目源代码	√	√
个人单独完成 (使用团队作业的文件, 但仓库由个人新建)	√	√
命令行用 CMD 和 GitBash 交替使用		√
git 本地配置查看与修改		√
git 本地默认分支重命名		√
Github Desktop 的使用		√
与 Gitlab 使用对比		√
其它版本控制软件使用并对比		√

表 2: 作业要求与个人完成情况

2 安装 Git

2.1 本地机器上安装 git

本地安装要求

1. 实验平台: Linux 和 Windows 下的 Git 安装操作二选一即可
2. 操作要求: 使用命令行方式完成实验, 避免图形界面下的操作

由于我之前在 Windows 操作系统上就安装过 Git 了, 这里就不再重新安装一遍了。我直接展示安装后的结果:

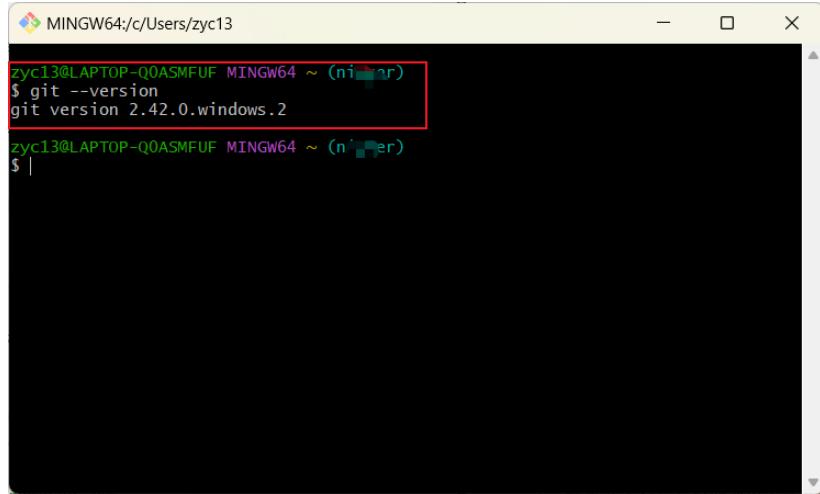


图 2.2: Git Version

如图2.2所示，打开 Git Bash 后通过命令：

```
1 zyc13@LAPTOP-QOASMFUF MINGW64 ~ (nixxer)
2 $ git --version
3 git version 2.42.0.windows.2
```

这里可以看到结果显示 git 版本号为：2.42.0.windows.2。证明了在本地上是安装成功的。

默认分支情况说明

这里我想对自己的分支名”nixxer“进行一个解释。由于我不是第一次安装 Git，我已经使用 Git 长达 2-3 年了，最初安装的时候我在阅读一个安装指南上介绍：

“Git 将默认的分支从 master 更改为 main 的原因是为了消除种族歧视，因为 master-slave（主从）架构在当今社会被认为是不恰当的，而 main（主要）作为默认分支名更加中立和包容。”

那时候的我更多的是一种抱着梗小鬼的心态，所以我逆其道而行之，选了一个更加地獄笑话的默认分支名。之后因为用了很久和远程分支进行联动，一直没有改。不过本次实验在操作之前我会纠正这个遗留已久的问题（尴尬）。

配置好了后可以通过命令 `git config -list` 查看全部的配置信息

```

zyc13@LAPTOP-QOASMFUF MINGW64 ~ (nixxer)
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=D:/Tools/Git/mingw64/etc/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
core.fsmonitor=true
pull.rebase=false
credential.helper=manager
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
user.email=ErwinZhou@outlook.com
user.name=Yuchen Zhou
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
core.protectntfs=false
core.repositoryformatversion=0
....skipping...
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=D:/Tools/Git/mingw64/etc/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
core.fsmonitor=true
pull.rebase=false
credential.helper=manager
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
user.email=ErwinZhou@outlook.com
user.name=Yuchen Zhou
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
core.protectntfs=false
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
remote.origin.url=https://gitlab.eduxiji.net/nku20233/ImplementationOfCompiler
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
gui.wmstate=normal

```

图 2.3: git config 配置

如图2.3所示，可以看到显示了完整的 git 配置信息，其中和远程 git 相关的配置信息主要有：

```

1 zyc13@LAPTOP-QOASMFUF MINGW64 ~ (nixxer)
2 $ git config --list
3
4 user.email=ErwinZhou@outlook.com
5 user.name=Yuchen Zhou
6
7 remote.origin.url=https://gitlab.eduxiji.net/nku20233/ImplementationOfCompiler
8 remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*

```

可以看到主要有我们本地的邮件和名字分别为我的 Outlook 邮箱和英文名字。另外还有一个配置信息，是因为之前在本地该根目录下和上学期编译原理课程使用的 gitlab 建立了对应的仓库。

2.2 申请 github 帐号

接下来主动申请 github 帐号，由于我之前很久前就申请了 Github 帐号，因此此处不再重复申请，而直接展示申请后的 GitHub 帐号信息。我申请后的 Github 信息如下：

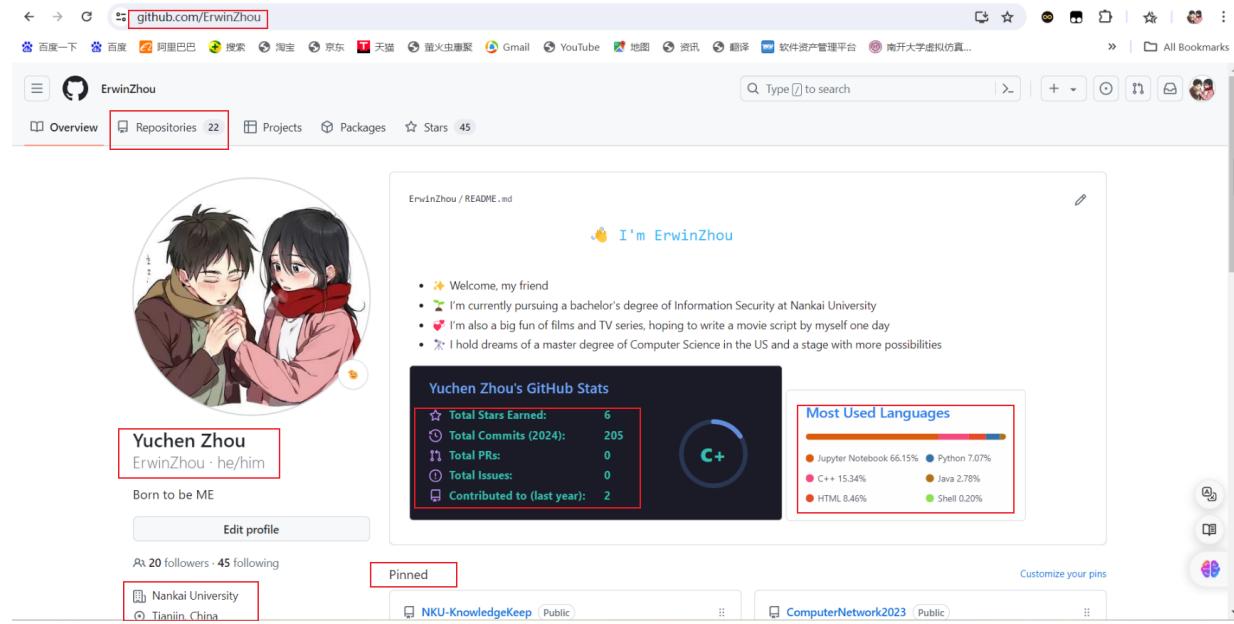


图 2.4: Github 主页 1

如图2.4所示，我的主页 URL 为 <https://github.com/ErwinZhou>。另外还有很多我的主页设置，其中详细包括：

- **用户名与昵称：**由于看到很多外国友人在 Github 都是实名上网的，因此我这里也采用我的真实中文汉语拼音名以及英文名字 **Erwin Zhou**。
- **已创建仓库：**由于我已经使用 Github 管理仓库有一段时间了，可以看到我已经有相当数量的 GitHub 仓库了，代表了我先前的 Git 操作经验。
- **Github 个性化主页**

可以看到我自己设计的 Github 个性化主页，关于我现在就读的学校专业以及未来对自己的一些规划。上面有一个可以变化的动图，显示“**I'm ErwinZhou**” 和 “**Welcome to my Rivendell**”。

- **Github Stats：**在右侧首先可以看到我的 Github 提交数据，其中包括一共获得过 6 颗星星（可能因为我的 Github 是全英文的缘故，不太容易被南开人发现），2024 年一共有 219 次提交，总共的 Pull Request 次数，总共发现的 Issues 以及股哦去一年中的贡献次数为 2。这些是我 Github 的活跃以及良好的使用习惯的证明，总评为 C+。
- **Most Used Languages：**其中 Python 语言占据打多少，C++ 和 Java 次之。代表了我上传过的项目语言很具丰富性。
- **Profile：**可以看到我的部分 Profie，包括我现在已经有 20 个 followers，45 个 following 的用户。以及我的学校信息 Nankai University。我现在所处的地址 Tianjin, China。

Rivendell Easter Egg

幽谷 (Rivendell)，辛达语称伊姆拉缀斯 (Imladris)，也译作林谷，是 J.R.R. 托尔金在其小说《魔戒》(Lord of the Rings) 中的虚构名。在他所创造的世界中，Rivendell 是半精灵埃尔隆德的领地。也是第三纪元精灵在中洲仅存的领地之一。这里保存着精灵族大部分历史和学说，是古老的智慧与知识的象征。幽谷是一块美丽与平静之地，旅人们在这里思索和歇息，也可以得到睿智的指点。

我之所以用 Rivendell 来将我的 Github 仓库与精灵的圣地比喻，因为我在其中保存了我大部分的代码工作，学习生活还有一些留学出国等考试经验与资料，不仅是为了我自己留存，更是为了与他人分享，争取帮到后面更多的人。

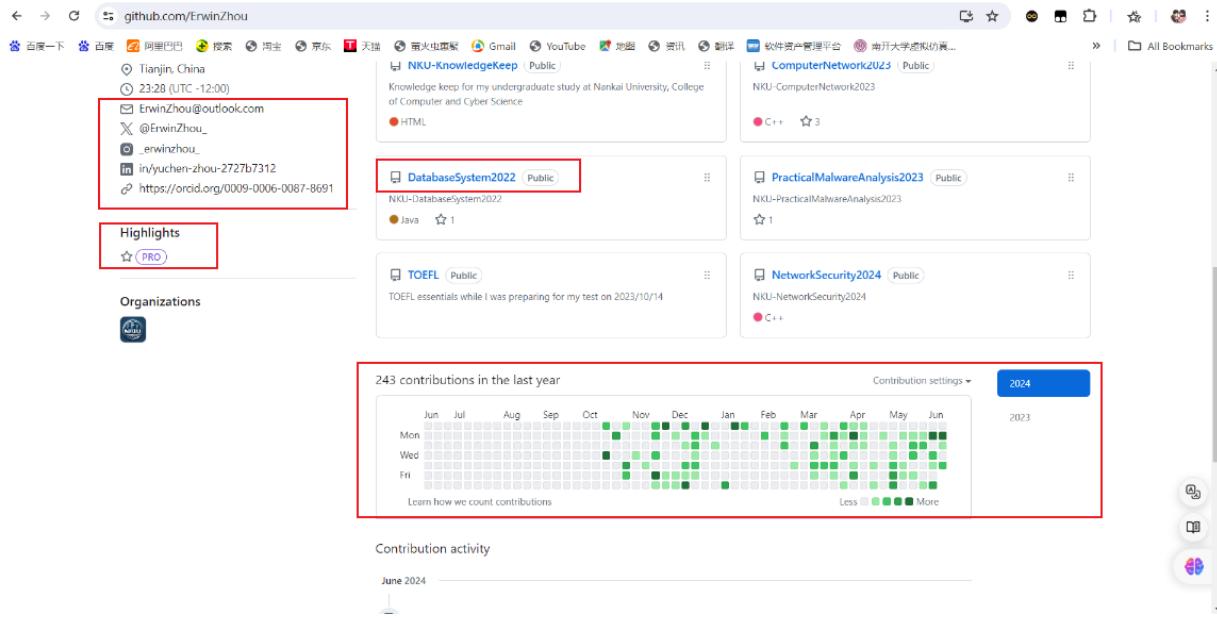


图 2.5: Github 主页 2

如图2.5所示，还可以看到更多我的 Github 主页信息，其中值得注意的有：

- Profile:** 可以看到我的另一部分 Profie，包括**我的邮箱**，**Twitter**，**Instagram**，**Linkedin** 以及 **ORCID 帐号**。
- Highlights:** 在一年前就已经使用的，包括本学期第一次作业的 **Github Pro**，使用 **Github Copilot**。
- Pinned Repositories:** 下面显示了我展示的几个最满意的仓库，包括恶意代码，计算机网络以及数据库，甚至包括我的托福资料和经验。
- Contributions last year:** 下面可以看到我的 **Github 贡献记录**，证明了我 git 使用的活跃。

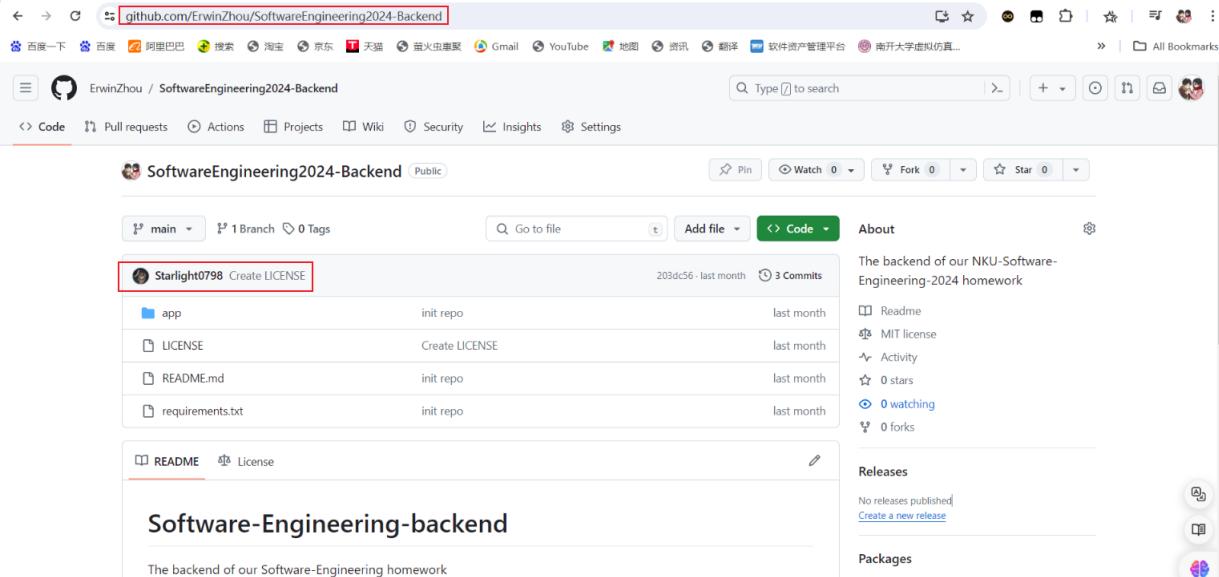


图 2.6: Github-Project

如图2.6所示，由于本次实验的两个仓库都需要新建，因此这里随便挑选了一个仓库进行展示和简单介绍，项目名为 SoftwareEngineering2024-Backend，代表的是我们软件工程项目大作业的后端。是我个人创建的我们后端仓库，并不是我们项目最大的后端仓库，只是留给我个人进行记录。创建仓库 License 的也是我们的组员之一。实际实验中涉及的新建仓库 URL 地址详见表1。

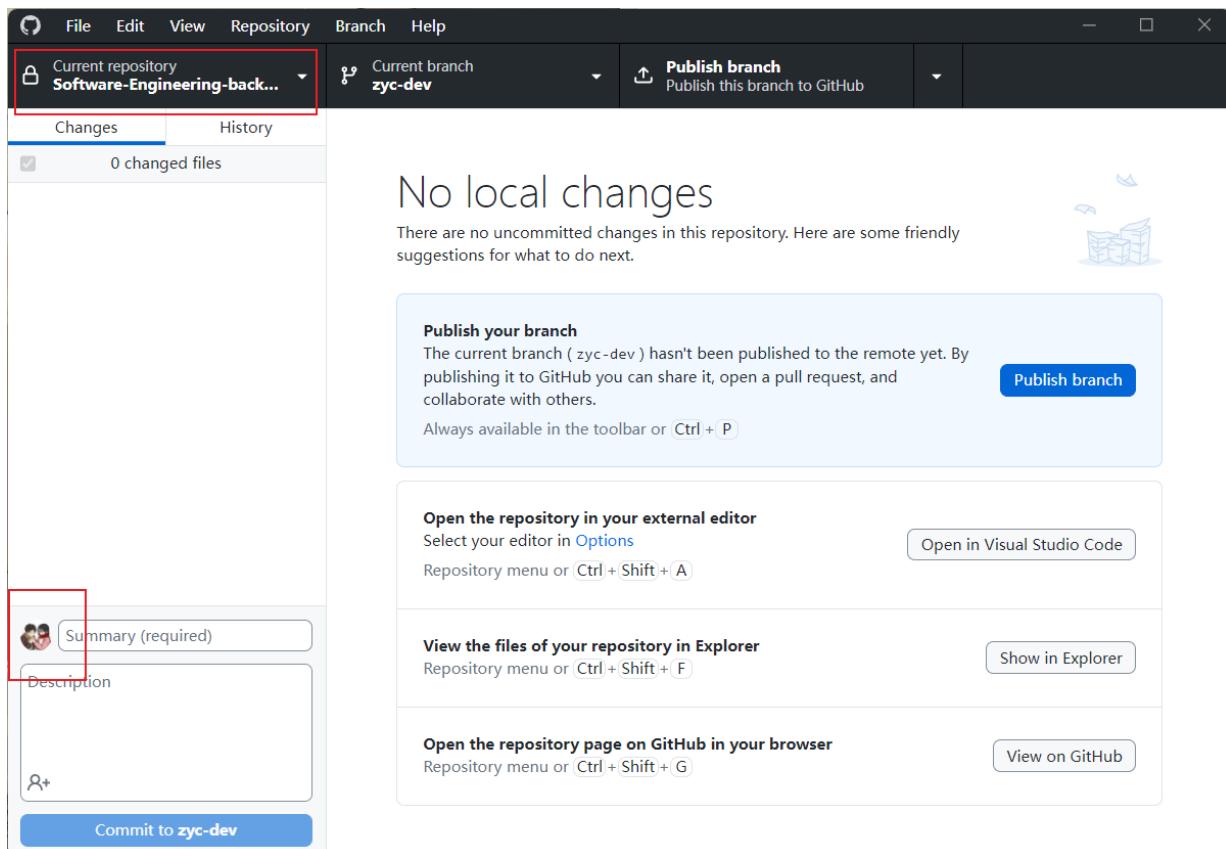


图 2.7: Github-Project Desktop

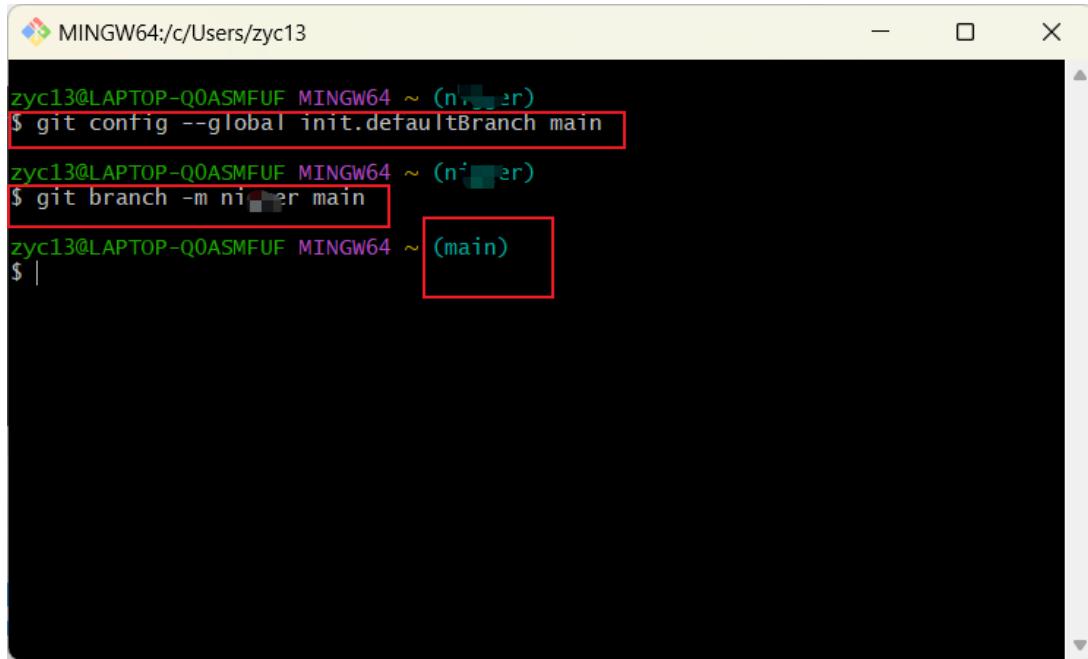
如图2.7所示，可以看到我的 Github Desktop。还是那个 SoftwareEngineering2024-Backend。

2.3 git 本次默认分支重命名

就像前面说的，为了修正这个历史遗留问题，我现在在这里进行本地默认分支重命名的更改，更改为更加正式的 main 分支名。

包括我未来还要申请留学的，一定得把这个坏习惯改掉，不然如果有一天，咳咳后果不堪设想（哭）。

首先打开 Git Bash 可以看到此时的默认分支名是错误的，首先利用 `init.defaultBranch` 更改为 main，然后再将本地此目录下已经创建好的 git 仓库的分支名重命名，即 `branch -m`。



```
MINGW64:/c/Users/zyc13
zyc13@LAPTOP-QOASMFUF MINGW64 ~ (nixxer)
$ git config --global init.defaultBranch main
zyc13@LAPTOP-QOASMFUF MINGW64 ~ (nixxer)
$ git branch -m nixxer main
zyc13@LAPTOP-QOASMFUF MINGW64 ~ (main)
$ |
```

图 2.8: Git rename default local branch

```
1 zyc13@LAPTOP-QOASMFUF MINGW64 ~ (nixxer)
2 $ git config --global init.defaultBranch main
3
4 zyc13@LAPTOP-QOASMFUF MINGW64 ~ (nixxer)
5 $ git branch -m nixxer main
6
7 zyc13@LAPTOP-QOASMFUF MINGW64 ~ (main)
8 $
```

如图2.8所示，可以看到本地默认分支名和该目录的仓库名字已经改过来了，，接下来查看所有的配置以及它们所在的文件验证修改: `git config -list -show-origin`

```

file:D:/Tools/Git/etc/gitconfig init.defaultbranch=nixxer
file:C:/Users/zyc13/.gitconfig user.email=ErwinZhou@outlook.com
file:C:/Users/zyc13/.gitconfig user.name=Yuchen Zhou
file:C:/Users/zyc13/.gitconfig filter.lfs.clean=git-lfs clean -- %f
file:C:/Users/zyc13/.gitconfig filter.lfs.smudge=git-lfs smudge -- %f
file:C:/Users/zyc13/.gitconfig filter.lfs.process=git-lfs filter-process
file:C:/Users/zyc13/.gitconfig filter.lfs.required=true
file:C:/Users/zyc13/.gitconfig core.protectntfs=false
file:C:/Users/zyc13/.gitconfig init.defaultbranch=main
file:.git/config core.repositoryformatversion=0
file:.git/config core.filemode=false
file:.git/config core.bare=false
file:.git/config core.logallrefupdates=true

```

图 2.9: Git 配置文件溯源

```

1 zyc13@LAPTOP-QOASMFUF MINGW64 ~ (main)
2 $ git config --list --show-origin
3
4 file:D:/Tools/Git/etc/gitconfig init.defaultbranch=nixxer
5 file:C:/Users/zyc13/.gitconfig user.email=ErwinZhou@outlook.com
6 file:C:/Users/zyc13/.gitconfig user.name=Yuchen Zhou
7
8 file:C:/Users/zyc13/.gitconfig init.defaultbranch=main
9
10 file:.git/config init.defaultbranch=main

```

如图2.9所示，可以看到目录`.git/config` 即当前仓库`zyc13/.gitconfig` 当前用户的默认分支已经变成了`main`，修改成功。但是`Git/etc/gitconfig` 下的默认分支还是错误的。因此需要到该目录的文件下直接修改。

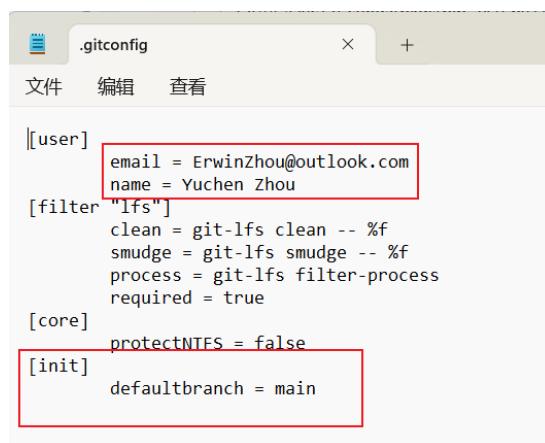


图 2.10: zyc13/.gitconfig

如图2.10所示，可以看到`zyc13/.gitconfig` 该目录下的配置文件的默认分支是正确的，同时还保留着**对应的后续邮箱和姓名**。

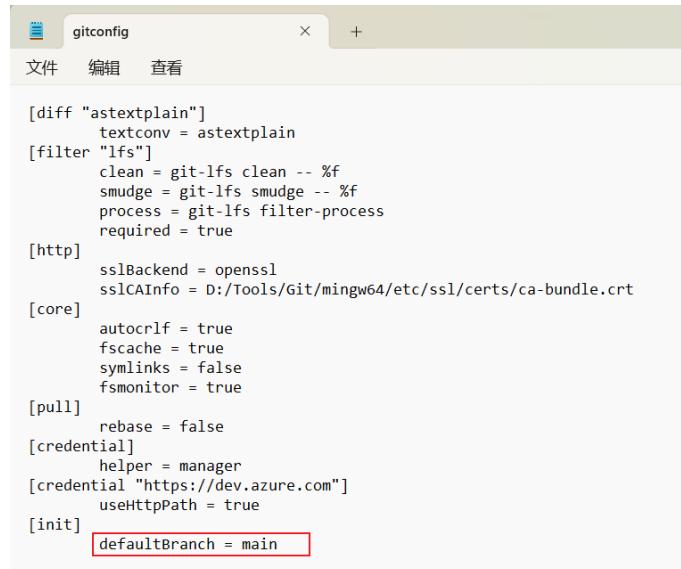


图 2.11: Git/etc/gitconfig

如图2.11所示，Git/etc/gitconfig 文件中将下面的 init 项中的 defaultBranch 改为 main 并保存，此时才修改完毕。

最后通过 `git config -list` 命令再次查看 git 配置信息：

```
zyc13@LAPTOP-QQASMEUF MINGW64 ~ (main)
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=D:/Tools/Git/mingw64/etc/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
core.fsmonitor=true
pull.rebase=false
credential.helper=manager
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=main
user.email=ErwinZhou@outlook.com
user.name=Yuchen Zhou
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
core.protectntfs=false
init.defaultbranch=main
:...skipping...
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=D:/Tools/Git/mingw64/etc/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
```

图 2.12: Git 最终查看

如图2.12所示，现在所有默认分支名都是 main 了！我的罪孽终于结束了！下面正式开始实验。

3 Git 操作过程

3.1 实验场景一：仓库创建与提交

实验场景一内容

- **R0: 查看状态**

- 在进行每次 git 操作之前，随时查看工作区、暂存区、git 仓库的状态，确认项目里的各文件当前处于什么状态。

- **R1: 初始化仓库**

- 本地初始化一个 git 仓库，将自己在团队作业中所创建项目的全部源文件加入进去，纳入 git 管理。

- **R2: 提交修改**

- 提交。
 - 手工对团队作业中的 3 个文件进行修改。

- **R3: 查看修改**

- 查看上次提交之后都有哪些文件修改、具体修改内容是什么（查看修改后的文件和暂存区域中相应文件的差别）。

- **R4: 重新提交**

- 重新提交。
 - 再次对这 3 个文件进行修改。

- **R5: 再次提交**

- 重新提交。

- **R6: 撤销提交**

- 把最后一次提交撤销。

- **R7: 查询提交记录**

- 查询提交记录。

基于实验场景一：首先完成 **R1-R7 的仓库创建与提交**。其中每次 git 操作之前，都会完成 R0 的要求，即随时查看工作区、暂存区、git 仓库的状态，确认项目里的各文件当前处于什么状态。**R0 主要通过以下命令：**

¹ # 查看当前工作区的状态，显示哪些文件被修改、哪些文件被添加到暂存区、哪些文件未被跟踪

² git status

³ # 查看工作区中的文件与最新提交之间的差异

⁴ git diff

```
5 # 查看暂存区中的文件与最新提交之间的差异  
6 git diff --cached  
7 # 查看工作区与暂存区之间的所有差异  
8 git diff HEAD  
9 # 查看提交记录  
10 git log
```

1. R1: 初始化仓库

首先选定本地初始化仓库的地址，我这里选择的是 D:\GitHub 地址，新建文件夹 Software-Engineering-project-management-test。因为还没有初始化任何 git 仓库，没有.git 文件，因此这里没必要先进行 R0 的要求，先初始化仓库后，再继续进行 R0 的要求。首先在该路径下如下 **R1 对应命令**：

```
1 # 初始化一个新的 git 仓库  
2 git init
```

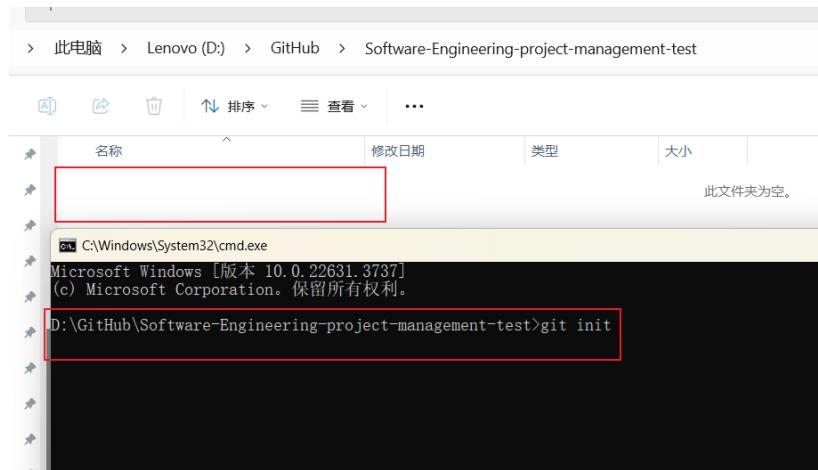


图 3.13: 本地初始化一个 git 仓库前

如图3.13所示，此时该目录下是空的，没有任何文件，也没有.git 文件夹，使用所需的命令后结果为：

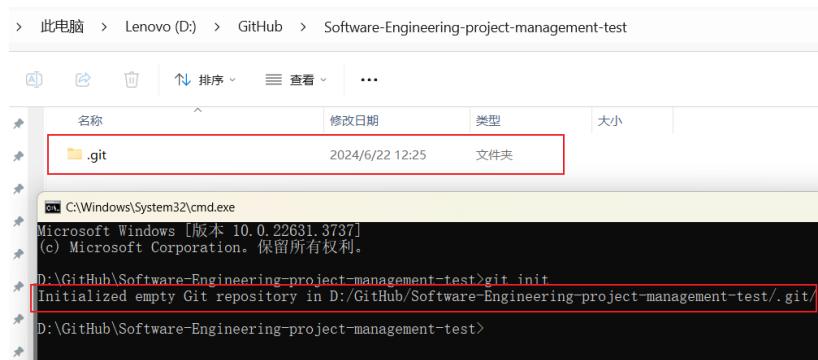


图 3.14: 本地初始化一个 git 仓库后

如图3.14所示，可以看到输出了提示 **Initialized empty Git repository in D:/GitHub/Software-Engineering-project-management-test/.git/**，证明本地 git 仓库创建成功，标志就是.git 文件夹。此时通过使用 R0 命令即代码块3.1查看下刚刚创建后的 git 仓库状态：

```

> 此电脑 > Lenovo (D:) > GitHub > Software-Engineering-project-management-test
    名称          修改日期      类型      大小
    .git          2024/6/22 12:29  文件夹

C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.22631.3737]
(c) Microsoft Corporation. 保留所有权利。

D:\GitHub\Software-Engineering-project-management-test>git init
Initialized empty Git repository in D:/GitHub/Software-Engineering-project-management-test/.git/
D:\GitHub\Software-Engineering-project-management-test>git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)

D:\GitHub\Software-Engineering-project-management-test>git log
fatal: your current branch 'main' does not have any commits yet

D:\GitHub\Software-Engineering-project-management-test>

```

图 3.15: 创建后查看 git 仓库状态

如图3.15所示，通过 `git status` 命令可以看到，此时处于分支 `main` 上，**证明我的默认分支名修改好了（长呼一口气）**。另外，对于暂存区和工作区的状态，`No commits yet` 且 `nothing to commit` 说明当前分支没有任何提交记录，并且没有任何文件被跟踪或修改。随后使用 `git log` 命令时，出现了 `fatal: your current branch 'main' does not have any commits yet` 的错误信息，**这进一步确认了当前分支还没有任何提交记录。**

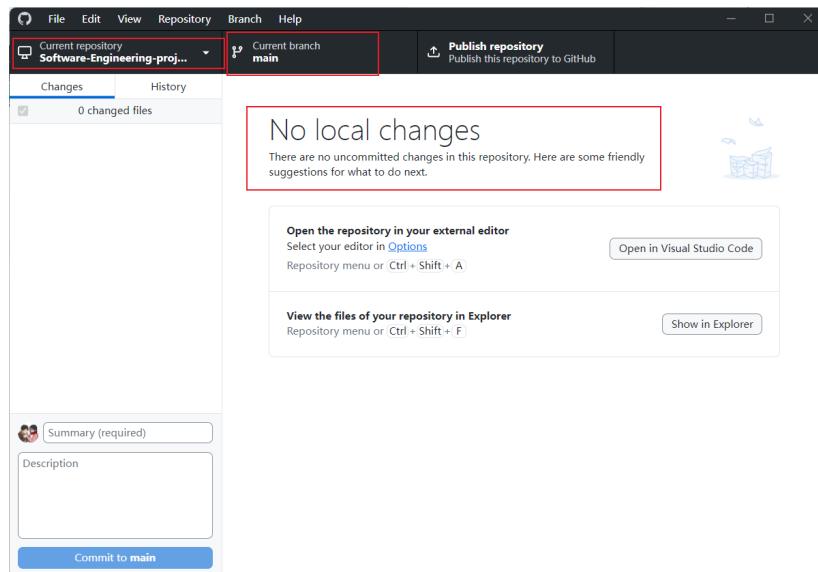


图 3.16: Github Desktop 显示初始化结果

如图3.16所示，现在在 Github Desktop 也可以看到刚创建的 Git 仓库是空的，没有任何提交记录。**现在我们将我们团队作业所创建项目的文件加入进来，具体而言：**

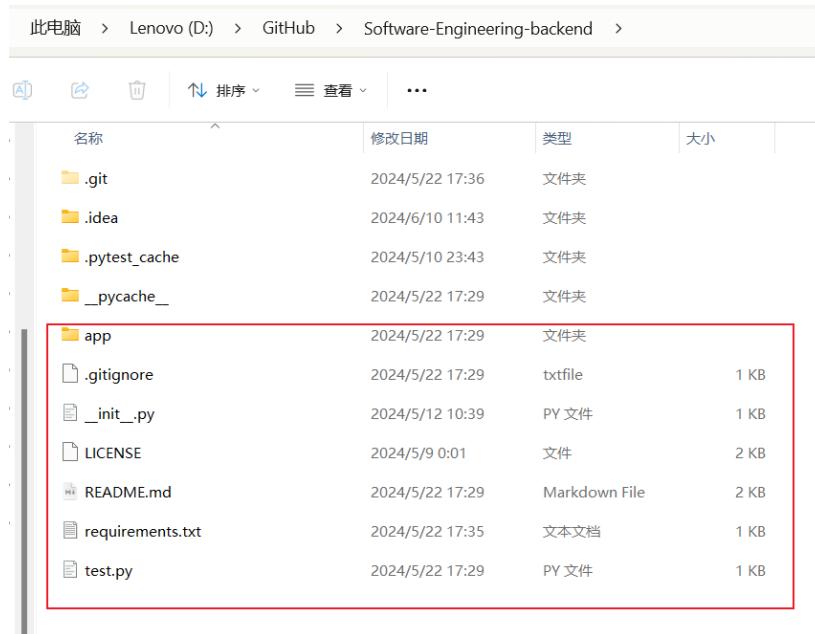


图 3.17: 加入的团队项目文件

如图3.17所示，将另一个项目目录下即 D:\GitHub\Software-Engineering-backend 即我们团队作业项目后端的实际代码，忽略 idea 和 vscode 配置文件等的源文件加入其中。

继续通过 R0 的代码块3.1中 git status 命令可以看到，此时处于分支 main 上，并且显示没有任何提交记录（No commits yet）。此外，状态信息显示有一些未被跟踪的文件，包括.gitignore、LICENSE、README.md、__init__.py、app/、requirements.txt 和 test.py。这些文件尚未被添加到暂存区，提示我们可以使用 git add <file>... 命令将这些文件添加到暂存区以进行跟踪。当前状态说明项目文件已经复制到工作区，但还没有被纳入版本控制。

接下来通过对命令，将当前目录下的全部源文件加入到本地 git 暂存区中：

1 # 加入全部源文件到暂存区

2 git add .

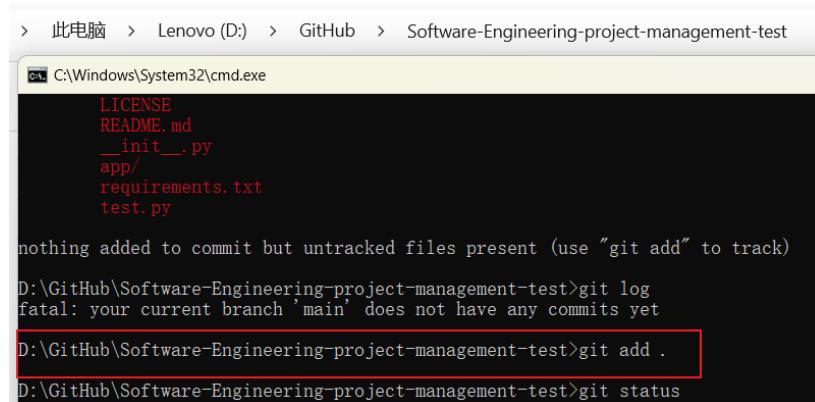


图 3.18: git add 加入到暂存区

如图3.18所示，可以看到通过`.`表示将所有文件都加入到暂存区，由于没有输出任何错误提示，其实是加入成功了。不过这里再利用R0的代码块3.1中`git status`命令可以看到：

The screenshot shows a Windows File Explorer window with the path: > 此电脑 > Lenovo (D:) > GitHub > Software-Engineering-project-management-test. Inside the folder, there are three items: .git (修改日期: 2024/6/22 12:51, 文件夹), app (修改日期: 2024/6/22 12:46, 文件夹), and .gitignore (修改日期: 2024/5/22 17:29, txtfile, 大小: 1 KB). Below the File Explorer is a terminal window titled 'cmd' with the path 'D:\GitHub\Software-Engineering-project-management-test'. The terminal output is:

```
nothing added to commit but untracked files present (use "git add" to track)
D:\GitHub\Software-Engineering-project-management-test>git log
fatal: your current branch main does not have any commits yet
D:\GitHub\Software-Engineering-project-management-test>git add .
D:\GitHub\Software-Engineering-project-management-test>git status
On branch main
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file: .gitignore
    new file: LICENSE
    new file: README.md
    new file: __init__.py
    new file: app/__init__.py
    new file: app/.pycache__config.cpython-312.pyc
```

图 3.19: git add 后查看 git 状态

如图3.19所示，仍然显示没有任何提交记录（`No commits yet`）。但是现在状态信息显示有一些变化即将被提交（`Changes to be committed`），这些文件包括`.gitignore`就是刚刚 add 进暂存区的文件，并且和之前颜色也不一样了，是绿色，之前 add 前是红色，说明 git 会将跟踪在暂存区中的文件标记为绿色，没有跟踪的标记为红色。这些文件已经被添加到暂存区，等待提交。表明这些文件现在被 Git 跟踪，可以在下一步进行提交操作。

另外还有（`use "git rm --cached <file>..." to unstage`）提示可以使用命令`git rm --cached <file>`来将文件从暂存区移除，但保留在工作区中。这意味着如果若误添加了某些文件到暂存区，可以通过这个命令将它们从暂存区中移除，而不会删除工作区中的实际文件。

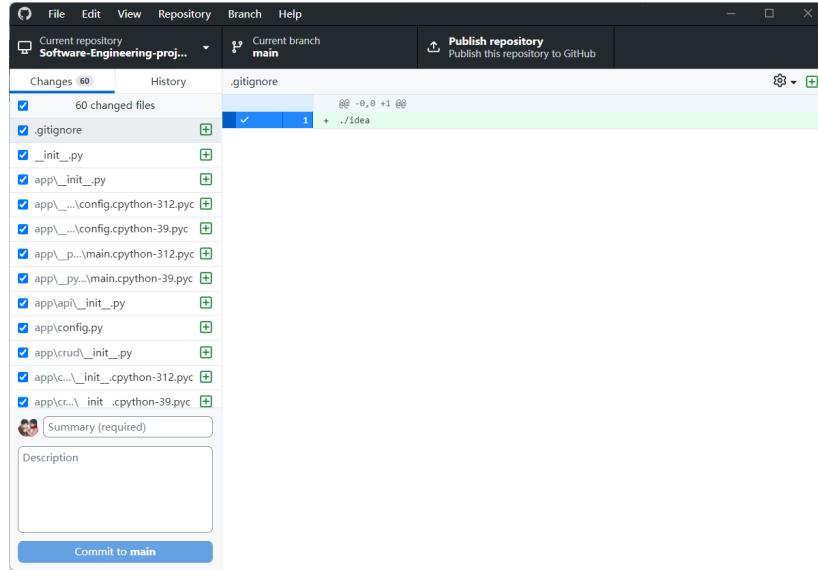


图 3.20: Github Desktop 在 add 后状态

如图3.20所示，可以看到此时 GitHub Desktop 中也显示了左侧的那些已经在暂存区中的文件。到此进一步的证明 **R1** 的操作已经全部完成，全部源文件已经纳入了 git 管理。

2. R2: 提交修改

接下来对 Git 进行提交，提交前的 git 状态如图3.19和3.20所示，也就是 add 后的状态，这里就不再重复展示了。现在通过以下命令对修改进行提交，即开始完成 **R1** 的操作：

```
1 # 将当前暂存区的文件提交到 Git 仓库:  
2 git commit -m "Initial commit of project files"
```

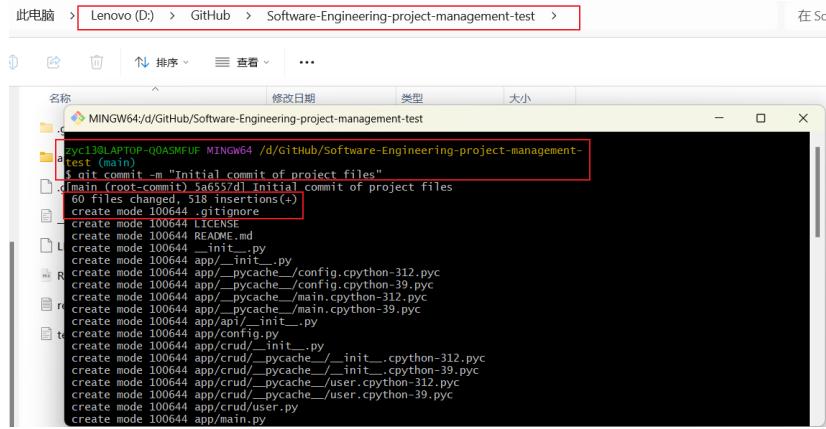
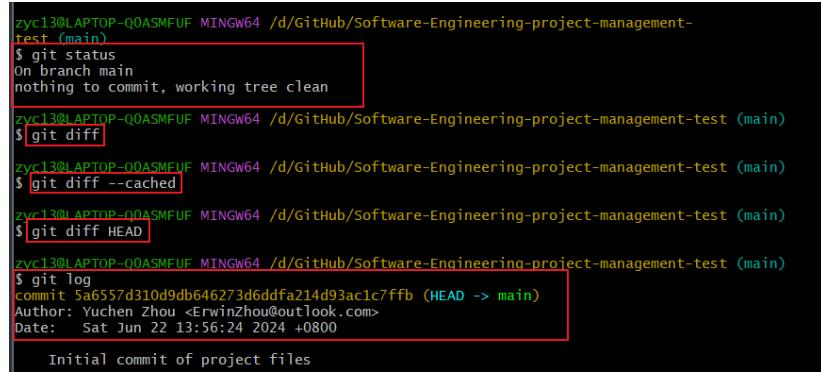


图 3.21: Git commit 后

如图3.21所示，接下来交替采用 **Git Bash** 进行操作。可以看到通过命令进行首次提交后，输出结果显示提交的信息：[main (root-commit) 5a6557d] Initial commit of project files。这里 main 表示当前分支，root-commit 表示这是该分支的第一次提交（根提交），5a6557d 是这次提交的哈希值，提交信息是 Initial commit of project files。

此次提交共涉及 60 个文件的变化，插入了 518 行内容。这些都与我们源文件数目和暂存数目一致。文件模式和文件名部分显示 `create mode 100644`，表示这些文件在此次提交中被创建。模式 100644 是文件权限模式，表示这是一个普通文件，并且其权限是可读可写，但只有所有者有写权限，组和其他用户只有读权限。这次提交将所有初始项目文件提交到了 `main` 分支，每个文件的模式为 100644，表示它们是普通文件且具有适当的读写权限。

接下来通过 R0 的代码块3.1中 `git status` 等命令查看 git 状态：



```

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git status
On branch main
nothing to commit, working tree clean

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git diff

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git diff --cached

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git diff HEAD

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git log
commit 5a6557d310d9db646273d6ddfa214d93ac1c7ffb (HEAD -> main)
Author: Yuchen Zhou <ErwinZhou@outlook.com>
Date:   Sat Jun 22 13:56:24 2024 +0800

Initial commit of project files
  
```

图 3.22: commit 提交后的 git 状态

如图3.22所示，通过 `git status` 命令可以看到，当前处于分支 `main` 上，并且显示 `nothing to commit, working tree clean`，这意味着工作区干净，没有任何未提交的更改。这和最开始显示的 `No commits yet` 不同，代表现在已经有了 commit 历史了，不过没有更新需要 commit 的了。

接下来运行 `git diff` 命令，没有输出，这表示工作区与暂存区之间没有差异。随后使用 `git diff --cached` 命令，同样没有输出，表明暂存区与上次提交之间没有差异。使用 `git diff HEAD` 命令也没有输出，这确认了工作区、暂存区和最新提交之间没有任何差异。这些都和我们的行为一致。

最后，通过 `git log` 命令，可以看到提交记录，其中显示唯一的提交是 `commit 5a6557d310d9db646273d6ddfa214d93ac1c7ffb`，这是分支 `main` 的根提交 (`HEAD -> main`)，提交信息是 `Initial commit of project files`。这表明所有初始项目文件已经成功提交，并且当前没有未提交的更改。

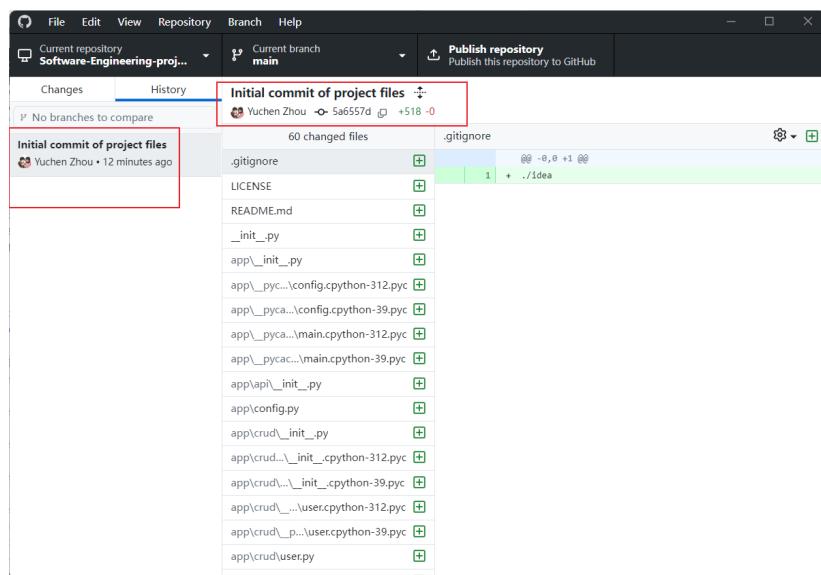


图 3.23: Github Desktop 查看 commit 提交后的状态

如图3.23所示，通过 Github Desktop 查看状态，左侧可以看到第一次也是唯一的一次提交记录，上面的显示的修改内容和刚才的 Git Bash 显示的一致。

接下来进行文件的修改，修改的三个文件分别属于不同的文件类型，并且路径也有所不同，都进行一定的增添和删除：

- README.md
- requirements.txt
- app\utils__init__.py

这里统一直接使用记事本打开：

```
1 # 直接使用记事本修改文件
2 notepad README.md
```

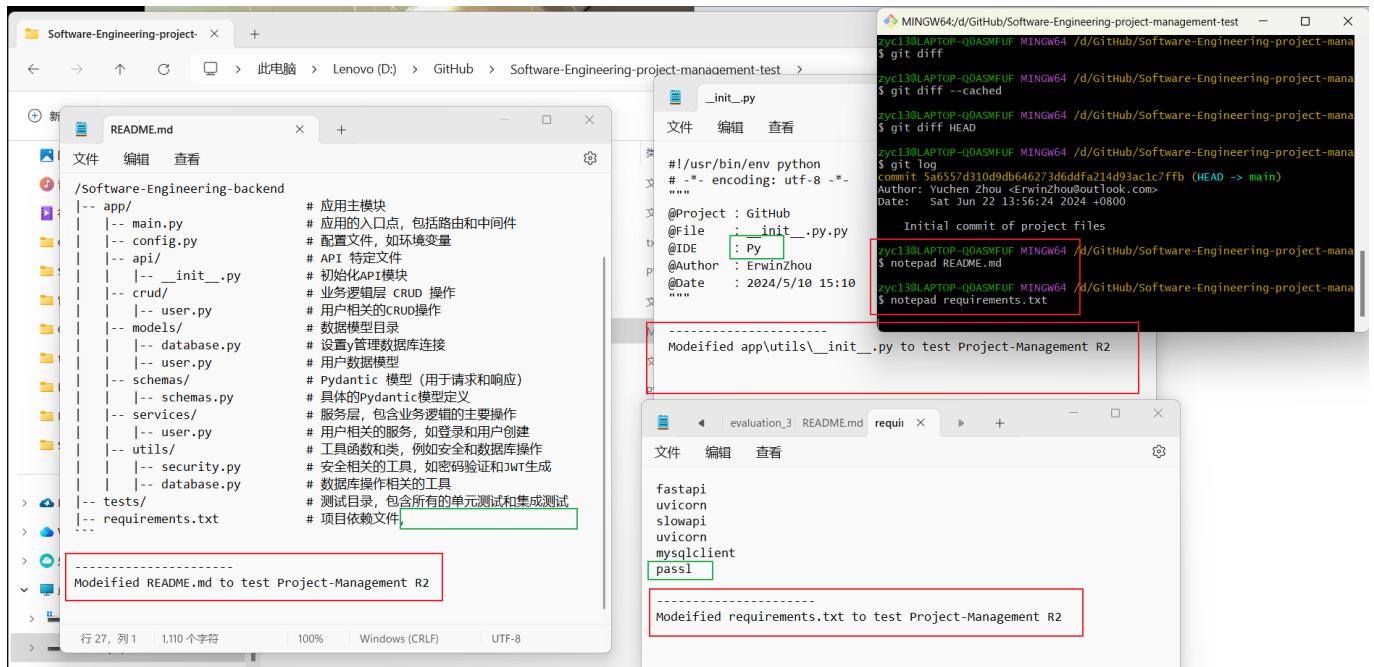


图 3.24: notepad 修改文件

如图3.24所示，**红框中**可以看到修改新增加的内容，每个文件都加入了一行分隔符和一句话：“Modeified FileName to test Project-Management R2”。**绿框中**可以看到修改新删除的内容，包括 README.md 去掉了一个文件描述的后半句，requirements.txt 把一个包 passlib 的名字就打了一半，还有 __init__.py 中。每个文件都进行了增加和删除，方便测试。

3. R3：查看修改

接下来进行 **R3**，查看上次提交之后都有哪些文件修改、具体修改内容是什么（查看修改后的文件和暂存区域中相应文件的差别），可以直接使用 R03.1中的命令：

```
1 # 查看当前工作区的状态，显示哪些文件被修改、哪些文件被添加到暂存区、哪些文件未被跟踪
2 git status
```

```

3 # 查看工作区中的文件与最新提交之间的差异
4 git diff
5 # 查看暂存区中的文件与最新提交之间的差异
6 git diff --cached
7 # 查看工作区与暂存区之间的所有差异
8 git diff HEAD

```

此电脑 > Lenovo (D:) > GitHub > Software-Engineering-project-management-test >

名称 修改日期 类型 大小

.git 2024/6/22 14:41 文件夹

C:\Windows\System32\cmd.exe

Microsoft Windows [版本 10.0.22631.3737]
(c) Microsoft Corporation。保留所有权利。

D:\GitHub\Software-Engineering-project-management-test>git status
On branch main
Changes not staged for commit:
 (use "git add <file>..." to update what will be committed)
 (use "git restore <file>..." to discard changes in working directory)
 modified: README.md
 modified: app/utils/__init__.py
 modified: requirements.txt

no changes added to commit (use "git add" and/or "git commit -a")

D:\GitHub\Software-Engineering-project-management-test>

图 3.25: git status

如图3.25, 通过 `git status` 命令可以看到, 当前处于分支 `main` 上, 并且显示有一些修改未暂存 (`Changes not staged for commit`)。这些文件包括 `README.md`、`app/utils/__init__.py` 和 `requirements.txt`。这些和我们刚才的修改完全一致。

状态信息提示可以使用 `git add <file>...` 命令将这些修改添加到暂存区, 或者使用 `git restore <file>...` 命令放弃这些修改。当前没有任何更改被添加到暂存区 (`no changes added to commit`), 可以使用 `git add` 和/或 `git commit -a` 命令来更新暂存区并提交更改。这些信息表明自上次提交以来, 这些文件已被修改, 但尚未被添加到暂存区。

C:\Windows\System32\cmd.exe - git diff

no changes added to commit (use "git add" and/or "git commit -a")

D:\GitHub\Software-Engineering-project-management-test>git diff
diff --git a/README.md b/README.md
index 77fb7a..5b607f7 100644
--- a/README.md
+++ b/README.md
@@ -23,6 +23,8 @@ content tree:
 | |-- security.py # 安全相关的工具, 如密码验证和JWT生成
 | |-- database.py # 数据库操作相关的工具
 | |-- tests/ # 测试目录, 包含所有的单元测试和集成测试
-|-- requirements.txt # 项目依赖文件, 列出了项目需要的所有Python库
+|-- requirements.txt # 项目依赖文件,

图 3.26: git diff-1

```

diff --git a/README.md b/README.md
index 77f1b7a..5b607f7 100644
--- a/README.md
+++ b/README.md
@@ -23,6 +23,8 @@ content tree:
    |   |-- security.py      # 安全相关的工具，如密码验证和JWT生成
    |   |-- database.py      # 数据库操作相关的工具
    |   -- tests/
    |   -- requirements.txt   # 测试目录，包含所有的单元测试和集成测试
+  -- requirements.txt      # 项目依赖文件，列出了项目需要的所有Python库
+  -- requirements.txt      # 项目依赖文件


+-----+
+Modified README.md to test Project-Management R2
diff --git a/app/utils/__init__.py b/app/utils/__init__.py
index 0cb913e..e63ccde 100644
--- a/app/utils/__init__.py
+++ b/app/utils/__init__.py
@@ -3,7 +3,10 @@
"""
@Project : GitHub
@File    : __init__.py.py
-@IDE    : PyCharm
+@IDE    : Py
@Author  : ErwinZhou
@Date    : 2024/5/10 15:10
"""


+
+-----+
+Modified app\utils\__init__.py to test Project-Management R2
\ No newline at end of file
diff --git a/requirements.txt b/requirements.txt
index e4fcbb48..599d3f3 100644
--- a/requirements.txt
+++ b/requirements.txt
@@ -3,4 +3,7 @@
uvicorn
slowapi
mysqlclient
-passlib
+passl
+
+-----+
+Modified requirements.txt to test Project-Management R2
\ No newline at end of file

```

图 3.27: git diff-2

如图3.26所示为使用 `git diff` 命令的截图，图3.27是叫未完整的输出结果图，可以看到，我们通过命令 `git diff` 后，实际上 git 是找到更改的文件，然后通过更加详细的命令挨个文件对比，比如 `diff --git a/app/utils/__init__.py b/app/utils/__init__.py`。显示了 `app/utils/__init__.py` 文件在工作区与上次提交之间的差异。输出的 `index 0cb913e blablabla 100644` 部分表示该文件在两次提交之间的差异索引及其权限模式。以下是具体的差异内容分析：

- a/app/utils/__init__.py 表示文件的旧版本。+++ b/app/utils/__init__.py 表示文件的新版本。
- @@ -3,7 +3,10 @@ 表示差异发生的位置。具体来说，这是从文件的第 3 行开始，旧版本和新版本的差异范围分别是 7 行和 10 行。

可以看到通过红色和绿色，后续显示的具体差异内容详细列出了哪些行被修改、添加或删除，这些都与我们之前修改的完全一致。

```

> 此电脑 > Lenovo (D) > GitHub > Software-Engineering-project-management-test >
    名称          修改日期      类型      大小
C:\Windows\System32\cmd.exe - git diff HEAD
Microsoft Windows [版本 10.0.22631.3737]
(c) Microsoft Corporation。保留所有权利。
D:\GitHub\Software-Engineering-project-management-test>git diff --cached
D:\GitHub\Software-Engineering-project-management-test>git diff HEAD
diff --git a/README.md b/README.md
index 77f1b7a..5b607f7 100644
--- a/README.md
+++ b/README.md
@@ -23,6 +23,8 @@ content tree:
 |   |-- security.py      # 安全相关的工具，如密码验证和JWT生成
 |   |-- database.py      # 数据库操作相关的工具
 |   |-- tests/           # 测试目录，包含所有的单元测试和集成测试
+- requirements.txt      # 项目依赖文件，列出了项目需要的所有Python库
+-- requirements.txt      # 项目依赖文件.

+-----+
+Modified README.md to test Project-Management R2
diff --git a/app/utils/_init_.py b/app/utils/_init__.py
index 0cb913e..e63ccde 100644
--- a/app/utils/_init_.py
+++ b/app/utils/_init_.py

```

图 3.28: git diff --cached 和 git diff HEAD

如图3.28所示，现在查看暂存区与上次提交的差异以及工作区和暂存区所有未提交的差异。可以看到由于本地的修改还没提交，因此暂存区和上次提交保持一致，没有任何输出。而工作区无需提交，已经可以比对出差异了，并且差异具体内容和 git diff 的输出内容一致，不再赘述。

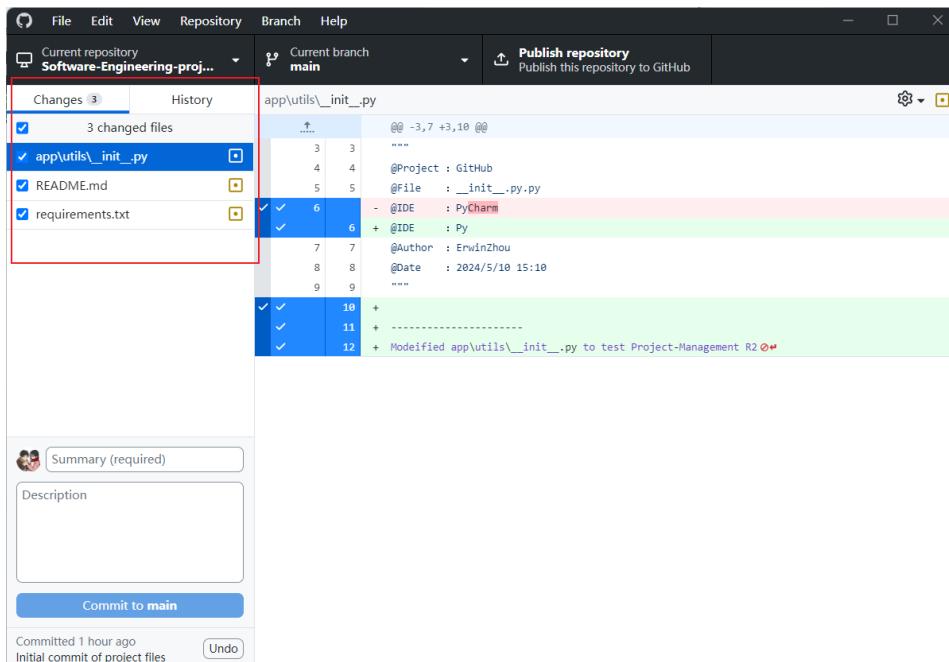


图 3.29: Github Desktop 查看修改

如图3.29所示，通过 Github Desktop 可以看到左侧非常清晰地列出来了文件的增添和删改位置，到此证明了 git 对暂存区以及工作区文件修改的追踪，都与我们操作的一致。

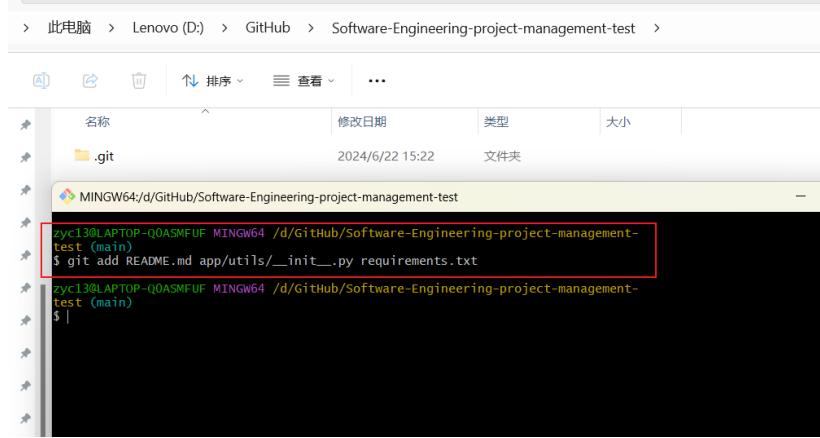
4. R4: 重新提交

接下来进行 R4，重新提交后再次对这 3 个文件进行修改。首先通过命令：

```

1 # 指定将当前未暂存的修改添加到暂存区
2 git add README.md app/utils/__init__.py requirements.txt

```



```

> 此电脑 > Lenovo (D:) > GitHub > Software-Engineering-project-management-test >
    名称          修改日期      类型      大小
    .git          2024/6/22 15:22  文件夹
MINGW64:/d/GitHub/Software-Engineering-project-management-test
zycl3@LAPTOP-QOASMFIU MINGW64 /d/Github/Software-Engineering-project-management-test (main)
$ git add README.md app/utils/__init__.py requirements.txt

```

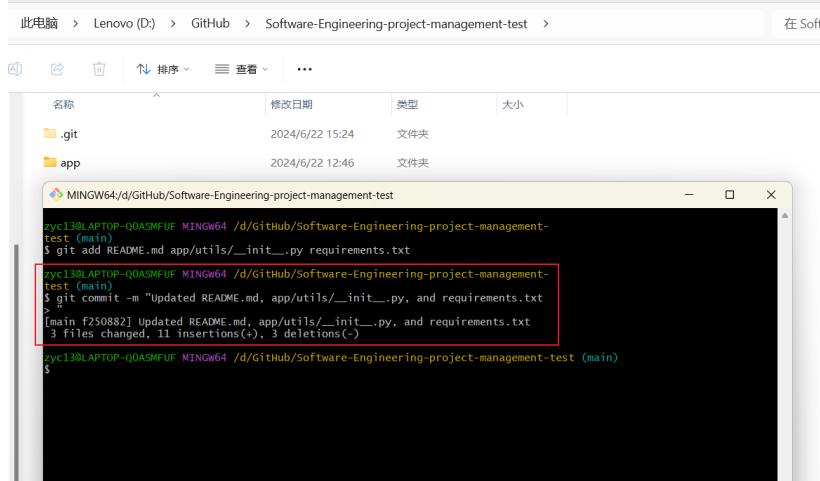
图 3.30: 重新 git add 加入到暂存区

如图3.30所示，只指定三个修改的文件加入到暂存区，没有任何输出信息表面正常加入。然后再进行提交：

```

1 # 将这些修改提交到 Git 仓库
2 git commit -m "Updated README.md, app/utils/__init__.py, and requirements.txt"

```



```

> 此电脑 > Lenovo (D:) > GitHub > Software-Engineering-project-management-test >
    名称          修改日期      类型      大小
    .git          2024/6/22 15:24  文件夹
    app          2024/6/22 12:46  文件夹
MINGW64:/d/GitHub/Software-Engineering-project-management-test
zycl3@LAPTOP-QOASMFIU MINGW64 /d/Github/Software-Engineering-project-management-test (main)
$ git add README.md app/utils/__init__.py requirements.txt
zycl3@LAPTOP-QOASMFIU MINGW64 /d/Github/Software-Engineering-project-management-test (main)
$ git commit -m "Updated README.md, app/utils/__init__.py, and requirements.txt"
[main f250882] Updated README.md, app/utils/__init__.py, and requirements.txt
  3 files changed, 11 insertions(+), 3 deletions(-)

```

图 3.31: 重新 commit

如图3.31所示，现在重新将三个文件 commmit 后，[main f250882] Updated README.md, app/utils/__init__.py, and requirements.txt。这里 main 表示当前分支，f250882 是这次提交的哈希值。此次提交涉及 3 个文件的变化，具体为 11 行内容的插入和 3 行内容的删除。

这表明对 README.md、app/utils/__init__.py 和 requirements.txt 的修改已成功提交到 Git 仓库。

```

yyc13@LAPTOP-QOASMIFU MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git add README.md app/utils/__init__.py requirements.txt
yyc13@LAPTOP-QOASMIFU MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git commit -m "Updated README.md, app/utils/__init__.py, and requirements.txt"
> [main f250882] Updated README.md, app/utils/__init__.py, and requirements.txt
  3 files changed, 11 insertions(+), 3 deletions(-)
yyc13@LAPTOP-QOASMIFU MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git status
On branch main
nothing to commit, working tree clean
yyc13@LAPTOP-QOASMIFU MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ 

```

图 3.32: git status 查看重新提交后

如图3.32所示，重新提交后，git status 查看没有更新的提交了，因此两次修改都已经传上去了。

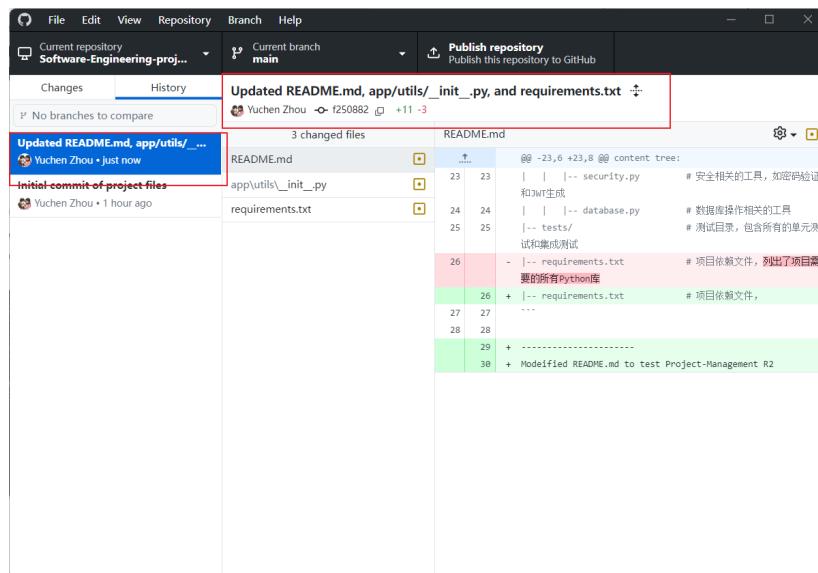


图 3.33: Github Desktop 查看重新 commit 后

如图3.33所示，接下来通过 Github Desktop 查看修改，可以看到此时左侧就有两个提交记录了，并且修改的描述也和刚刚提交的一致。

接下来再次对这 3 个文件进行修改，这次保证修改的内容和上次完全不同：

- README.md
- requirements.txt
- app\utils__init__.py

The terminal window shows the following command sequence:

```

$ git add README.md app/utils/__init__.py requirements.txt
$ git commit -m "Updated README.md, app/utils/__init__.py, and requirements.txt"
[main f250882] Updated README.md, app/utils/__init__.py, and requirements.txt
 3 files changed, 11 insertions(+), 3 deletions(-)

$ git status
On branch main
nothing to commit, working tree clean

```

The code editor shows the following changes made to three files:

- requirements.txt**: The first line was modified from "fastapi" to "fast". A red box highlights the change.
- app/utils/__init__.py**: The author information was modified. A red box highlights the change.
- README.md**: The text "I'm back again!" was added. A red box highlights the addition.

图 3.34: 重新修改三个文件

如图3.34所示，这次增加的新内容是 I'm back again!。进行的删除，对于 README.md，是上一个文件的描述后半句，对于 requirement.txt 是把第一个包 fastapi 删为 fast，对于 __init__.py 是删除了最后时间的除了年份部分。这次进行的修改是把上次添加的那句 R2 换为了 R4。

5. R5: 再次提交

接下来进行 R5 的操作，在进行 R5 操作前，还首先依据 R0 的代码块3.1中查看再次提交前的 git 状态，包括 git status, git diff 以及--cached, HEAD 等：

The terminal window shows the following command sequence:

```

$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified: README.md
    modified: app/utils/__init__.py
    modified: requirements.txt

no changes added to commit (use "git add" and/or "git commit -a")

$ git diff
diff --git a/README.md b/README.md
index 5b607f7..cblee6c 100644
--- a/README.md
+++ b/README.md

```

图 3.35: git 重新查看状态与差异等 1

```

MINGW64/d/GitHub/Software-Engineering-project-management-test
zcl3@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-
test (main)
$ git diff --cached
zcl3@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-
test (main)
$ git diff HEAD
diff --git a/README.md b/README.md
index 5b60f7...cb1ee6c 100644
--- a/README.md
+++ b/README.md
@@ -22,9 +22,12 @@ content tree:
 |   |-- utils/
 |   |   |-- security.py      # 工具函数和类，例如安全和数据库操作
 |   |   |-- database.py      # 安全相关的工具，如密码验证和JWT生成
 |   |   |-- tests/            # 数据库操作相关的工具
 |   |   |-- tests/            # 测试目录，包含所有的单元测试和集成测试
 |   |   |-- requirements.txt   # 测试目录，# 项目依赖文件,
image
ingine Modified README.md to test Project-Management R2

```

图 3.36: git 重新查看状态与差异等 2

如图3.35和3.36所示，可以看到输出详细的差异对比，都和我们的修改一致，这里不再赘述。

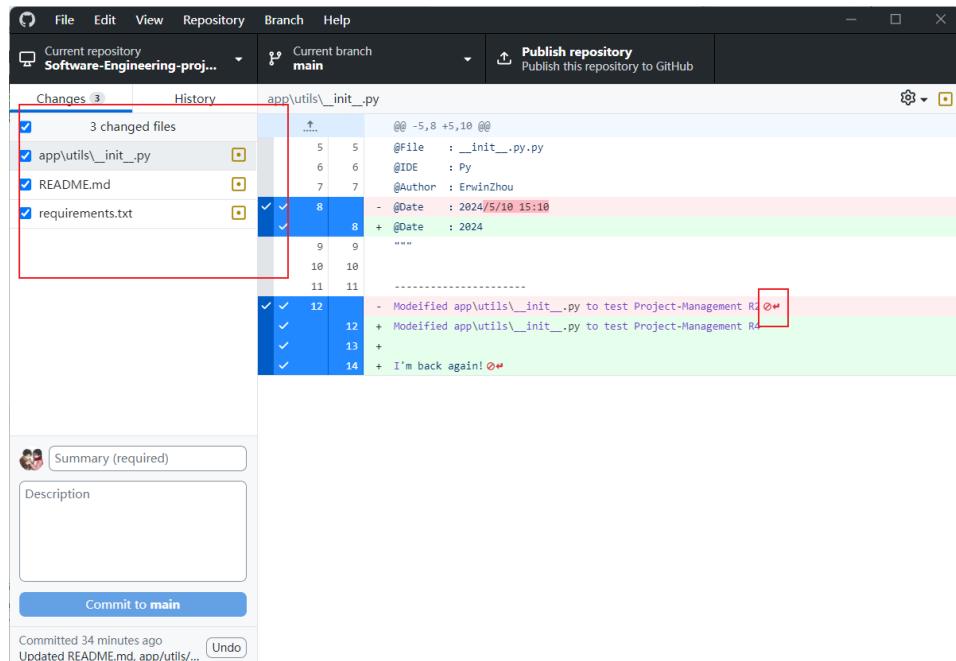


图 3.37: Github Desktop 重新查看状态与差异等

如图3.37所示，通过 Github Desktop 详细得出的差异也与图3.35和3.36相同，值得注意的是修改的位置有一个红色的箭头和弓靶子。

接下来通过以下命令，加入暂存区并提交最新的修改：

```

1 # 指定将当前未暂存的修改添加到暂存区
2 git add README.md app/utils/\_\_init\_\_.py requirements.txt
3 # 再次提交新的修改
4 git commit -m "Modified README.md, app/utils/\_\_init\_\_.py, and requirements.txt again"

```

```

MINGW64/d/GitHub/Software-Engineering-project-management-test
$ git add README.md app/utils/_init__.py requirements.txt
MINGW64/d/GitHub/Software-Engineering-project-management-test
$ git commit -m "Modified README.md, app/utils/_init__.py, and requirements.txt again"
[main 67a0c9f] Modified README.md, app/utils/_init__.py, and requirements.txt again
 3 files changed, 13 insertions(+), 6 deletions(-)

```

图 3.38: git 重新加入并提交最新的修改

如图3.38所示，重新加入并提交了最新的修改，输出信息与之前类似，不再赘述。

6. R6: 撤销提交

接下来进行 R6 的操作，操作前也是先依据 R0 的代码块3.1中查看撤销前的 git 状态，包括 `git status`, `git diff` 以及`--cached`, `HEAD` 等：

Commit	Author	Date	Message
Initial commit of project files	Yuchen Zhou	2 hours ago	
Modified README.md, app/utils/_init__.py	Yuchen Zhou	7 minutes ago	Modified README.md, app/utils/_init__.py
Updated README.md, app/utils/_init__.py, and requirements.txt	Yuchen Zhou	1 hour ago	Updated README.md, app/utils/_init__.py, and requirements.txt again

图 3.39: R6 操作前状态查看

如图3.39所示，可以看到此时工作区没有任何未提交的更改，并且通过 GitHub Desktop 能够看到往次的三次提交记录。

接下来进行撤销提交，把最后一次提交即”Modified README.md, app/utils/_init__.py, and requirements.txt again”撤销。具体而言，我这里的理解为，想要保留最后一次提交的修改，但将其从提交历史中移除。而不是将工作区恢复到上一个提交的状态，将修改保留在暂存区中。通过如下命令：

1 # 想要保留最后一次提交的修改，但将其从提交历史中移除

2 git reset --soft HEAD^

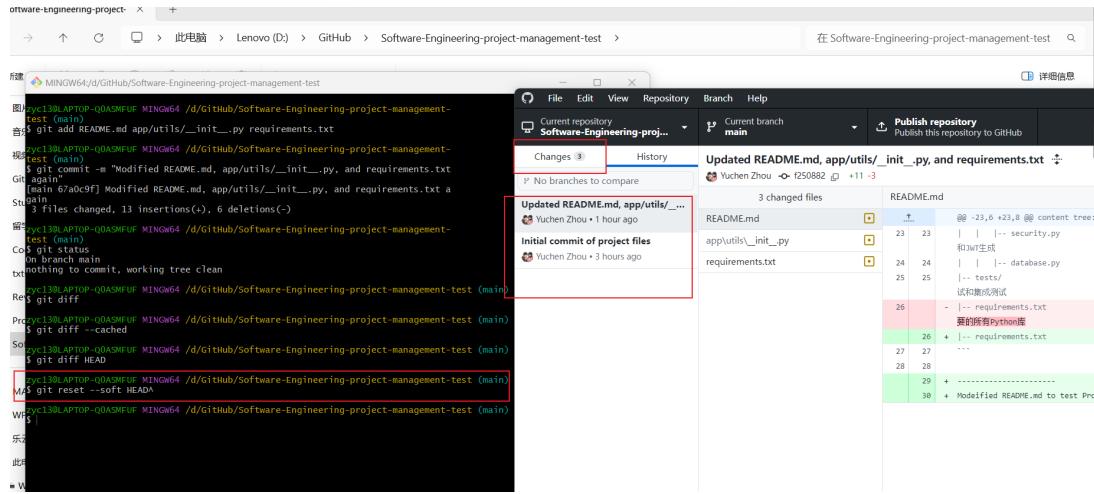


图 3.40: 软撤销后并用 Github Desktop 状态查看

如图3.40所示，软撤销后没有输出日志，代表成功。此时看 Github Desktop 能看到左侧又显示了 3 个 changes，并且提交记录只有两个了，最新的那个没了。

PS

这里实验场景一的 R6 要求为“把最后一次提交撤销”，我不知道这里撤销是要 soft 还是 hard，也就是说我不知道是要将最后一次提交的修改移除，并将工作区恢复到上一个提交的状态，还是移除就好，修改保留。我在上面实验中采用的是保留工作区的修改，若不保留工作区的修改，则通过如下命令：

```
1 git reset --hard HEAD^
```

我的理解是 R6 要求的是“撤销最后一次提交”，而不是要求“撤销对这些文件的暂存”。因此只展示了一个可能的后续操作，若实际理解为另一种，还请谅解。

同样现在依据 R0 的代码块3.1中查看撤销后的 git 状态：

```

MINGW64:/d/GitHub/Software-Engineering-project-management-test
$ git diff
zyc13@LAPTOP-QOASMUFU MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git diff --cached
zyc13@LAPTOP-QOASMUFU MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git diff HEAD
zyc13@LAPTOP-QOASMUFU MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git reset --soft HEAD
zyc13@LAPTOP-QOASMUFU MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   README.md
    modified:   app/utils/_init_.py
    modified:   requirements.txt

zyc13@LAPTOP-QOASMUFU MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git diff
zyc13@LAPTOP-QOASMUFU MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git diff --cached
diff --git a/README.md b/README.md
index 5b607f7..cb1ee6c 100644
--- a/README.md
+++ b/README.md
@@ -22,9 +22,12 @@ content tree:
 |  |-- utils/
 |  |  |-- security.py      # 工具函数和类，例如安全和数据库操作
 |  |  |-- database.py     # 安全相关的工具，如密码验证和JWT生成
 |  |  |-- tests/           # 数据库操作相关的工具
 |  |  |  |-- tests/         # 测试目录，包含所有的单元测试和集成测试
 |  |  |  |  |-- tests/       # 测试目录
 |  |  |  |  |  |-- requirements.txt  # 项目依赖文件,

```

图 3.41: 软撤销后 Git Status 查看状态

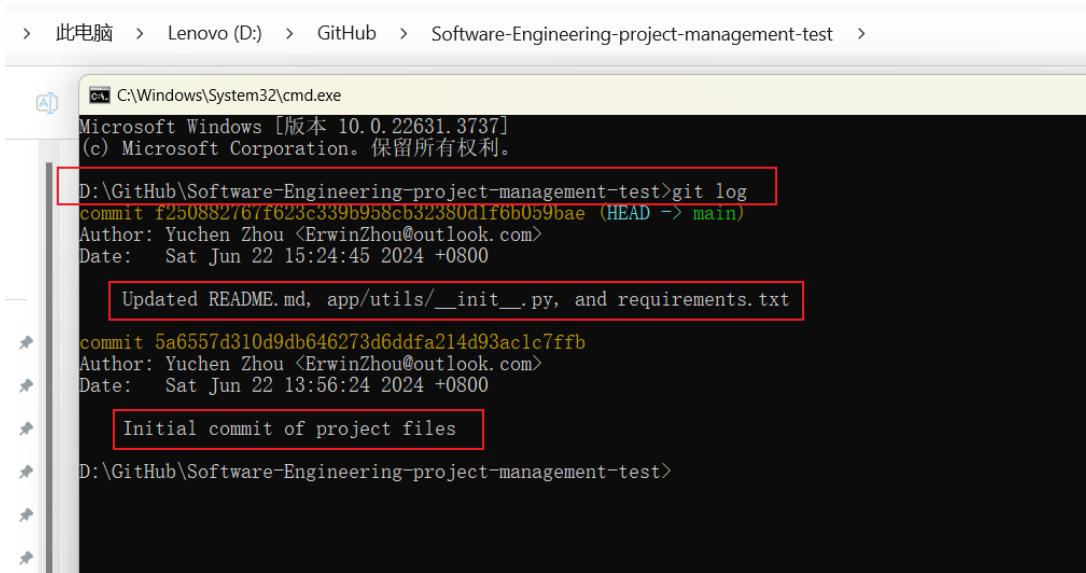
如图3.41所示，`git status` 又显示了“Changes to be committed:”代表了刚才的提交已经成功撤销了，另外有意思的是：这些文件标为绿色，表示它们虽然被撤销提交，但还是被添加到暂存区，等待提交。

总结一下上述操作即：

- 使用 `git reset --soft` 撤销了最后一次提交，但保留了修改在暂存区内。
- 由于使用了`--soft` 选项，修改保留在暂存区中。
- `git status` 显示这些文件为绿色，表示它们已暂存。

7. R7: 查询提交记录

现在进行 **R7 的操作要求**，实际上就是 R0 的要求之一，所以直接通过代码块3.1中的 `git log` 命令查看历史提交记录，结果如下：



```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.22631.3737]
(c) Microsoft Corporation. 保留所有权利。

D:\GitHub\Software-Engineering-project-management-test>git log
commit f25088276/f623c339b958cb32380d1f6b059bae (HEAD -> main)
Author: Yuchen Zhou <ErwinZhou@outlook.com>
Date:   Sat Jun 22 15:24:45 2024 +0800

    Updated README.md, app/utils/_init_.py, and requirements.txt

commit 5a6557d310d9db646273d6ddfa214d93ac1c7ffb
Author: Yuchen Zhou <ErwinZhou@outlook.com>
Date:   Sat Jun 22 13:56:24 2024 +0800

    Initial commit of project files

D:\GitHub\Software-Engineering-project-management-test>
```

图 3.42: git log 查看提交记录

如图3.42所示，可以清楚地看到此时只有两次提交记录，没有最后一次提交了，这是因为我在 R6 中将最后一次提交撤销了。同样地，之前通过 GitHub Desktop 查看的结果即图3.40也和 git log 查询结果保持一致。并且通过与图3.39在 R6 操作前的 Github Desktop 查看提交历史记录进行对比，能够很轻松地验证了 R5 中的最后一次提交已经通过 R6 撤销成功。

到此实验场景一，即仓库的创建与提交的实验内容全部完成。

3.2 实验场景二：分支管理

实验场景二内容

- 在你的 Github 网站上，通过 web 界面建立一个 project，将不少于 10 个文件（程序代码、文档等）加入进去，形成初始分支 B1；在 B1 基础上建立两个并行的分支 B2、B3，手工对 B2 和 B3 上的某些文件进行不同程度的修改并提交。
- **R8：克隆仓库**
 - 从 Github 上 (URL) 克隆一个已有的 git 仓库到本地。
- **R9：获取分支**
 - 获得该仓库的全部分支。
- **R10：创建新分支**
 - 在 B2 分支基础上创建一个新分支 C4。
- **R11：修改并提交**
 - 在 C4 上，对 4 个文件进行修改并提交。
- **R12：修改并提交**

- 在 B3 分支上对同样的 4 个文件做不同修改并提交。

- **R13: 合并分支**

- 将 B3 和 C4 分支合并，若有冲突，手工消解。

- **R14: 查看分支状态**

- 查看目前哪些分支已经合并、哪些分支尚未合并。

- **R15: 删除和合并分支**

- 将 C4 和 B3 合并后的分支删除，将尚未合并的分支合并到一个新分支上，分支名字为你的学号。

首先在进行 R8-R15 的操作前，按照要求通过 Github 的 web 界面创建仓库（项目）并添加对应的文件，这里首先创建一个仓库为 **SE2024-ProjectManagement-test**。

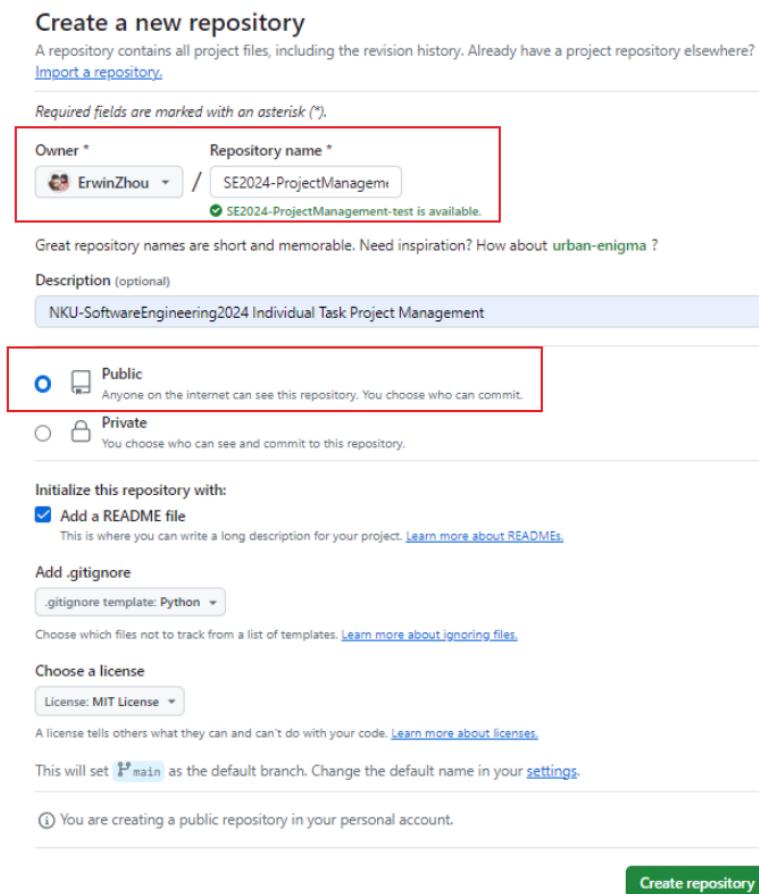


图 3.43: Github 的 web 界面创建新仓库

如图3.43所示，可以看到仓库的一些基本信息设置：仓库名为 SE2024-ProjectManagement-test，设置了基本描述，README，.gitignore 和 License 等。

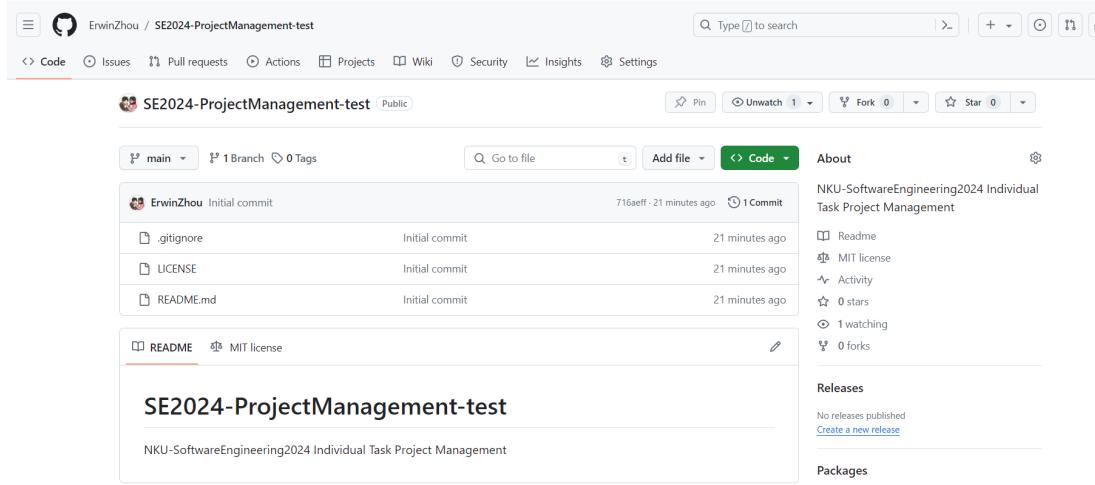


图 3.44: SE2024-ProjectManagement-test 初始仅有几个文件

如图3.44所示，SE2024-ProjectManagement-test 最开始仅有 README 几个文件，因此接下来要按照要求加入不少于 10 个文件（程序代码、文档等）。

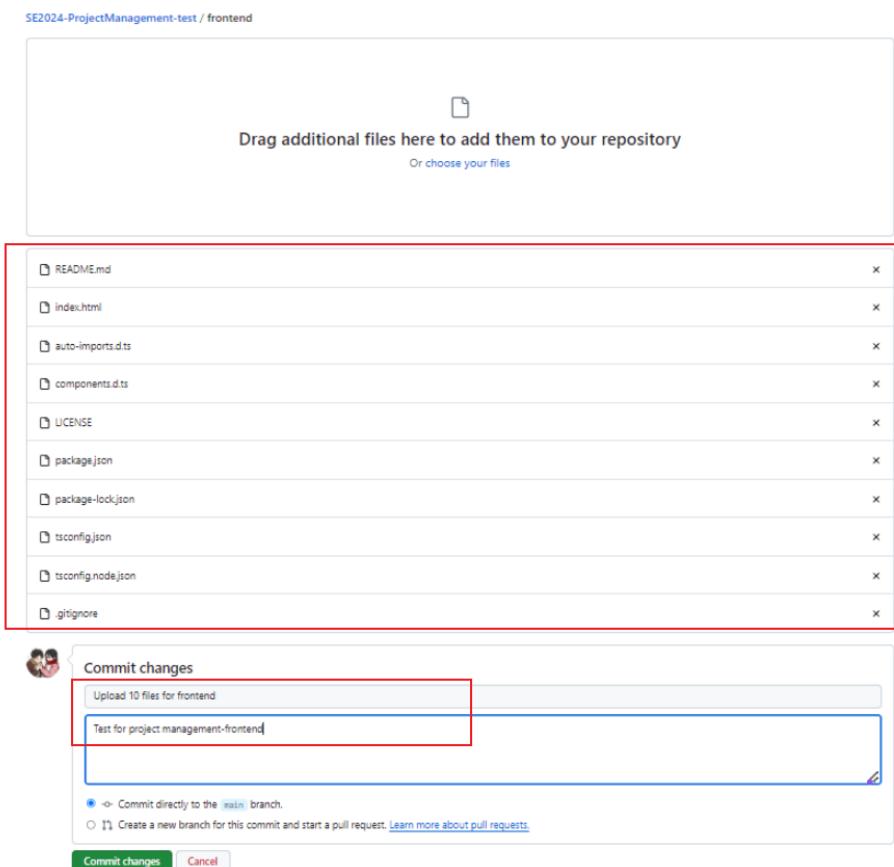


图 3.45: 上传 frontend

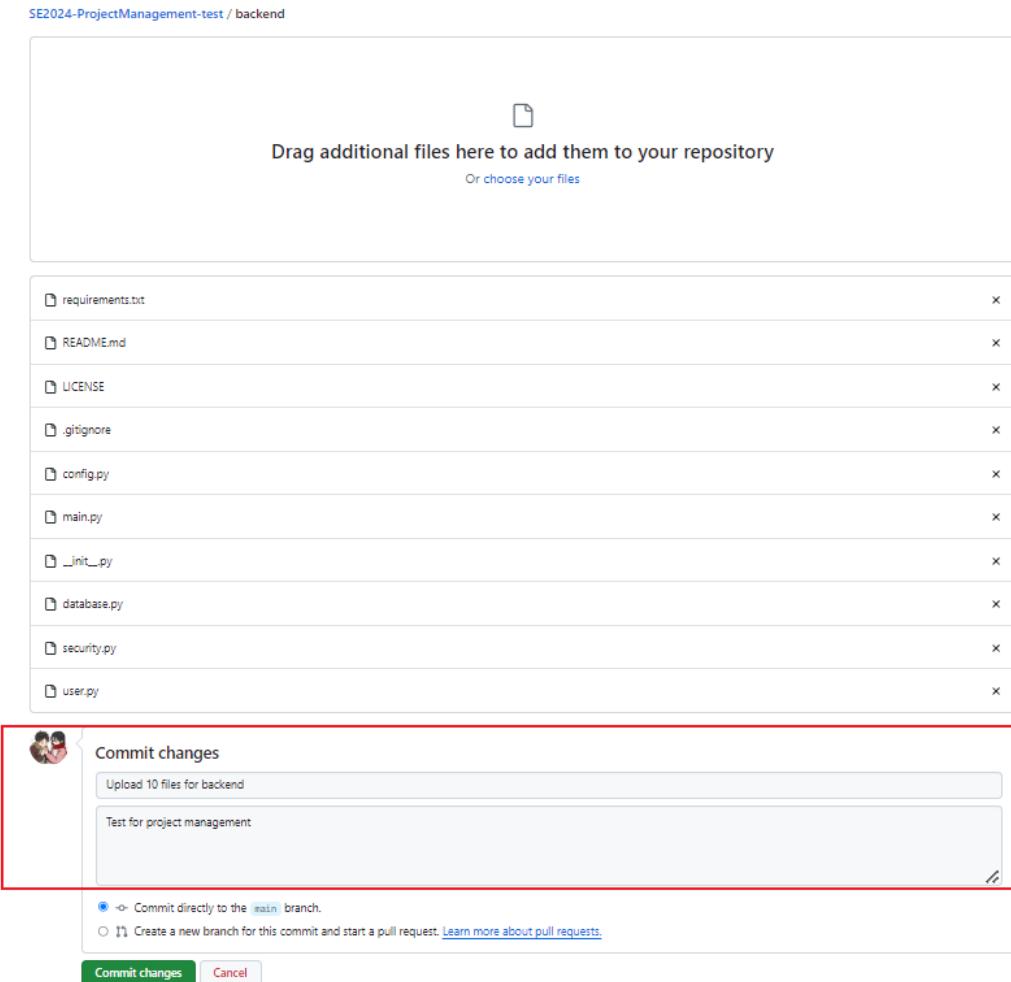


图 3.46: 上传 backend

如图3.45和3.46所示，我在根目录下创建了 frontend 和 backend 两个子目录，上传了各自十个文件，包括程序代码、文档等。他们都在初始分支 main 上。

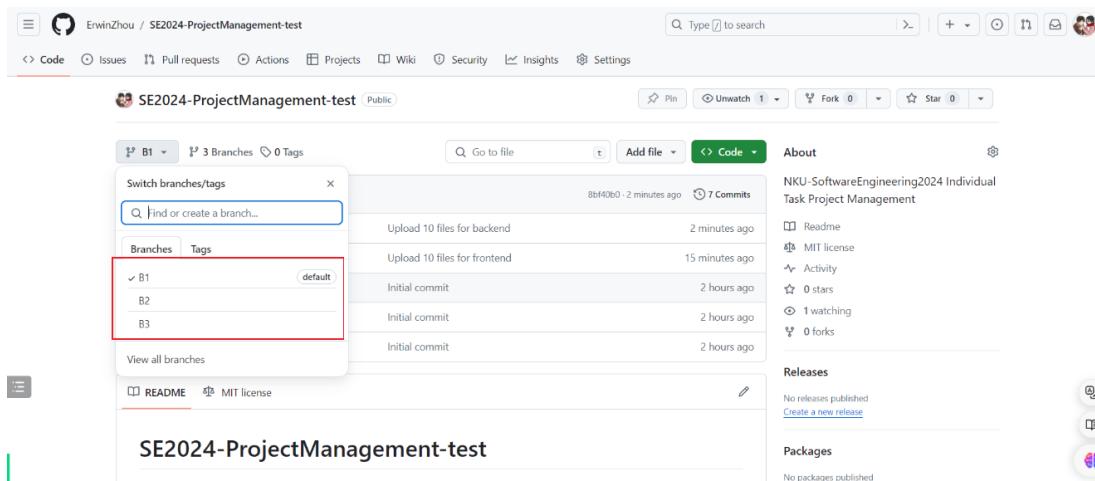


图 3.47: B1, B2 和 B3 分支

如图3.47所示，首先将 main 分支重命名为 B1，然后从 B1 的分支为源分支建立两个并行的分支 B2 和 B3。

下来对 B2 和 B3 分支上的文件进行不同程度的修改：

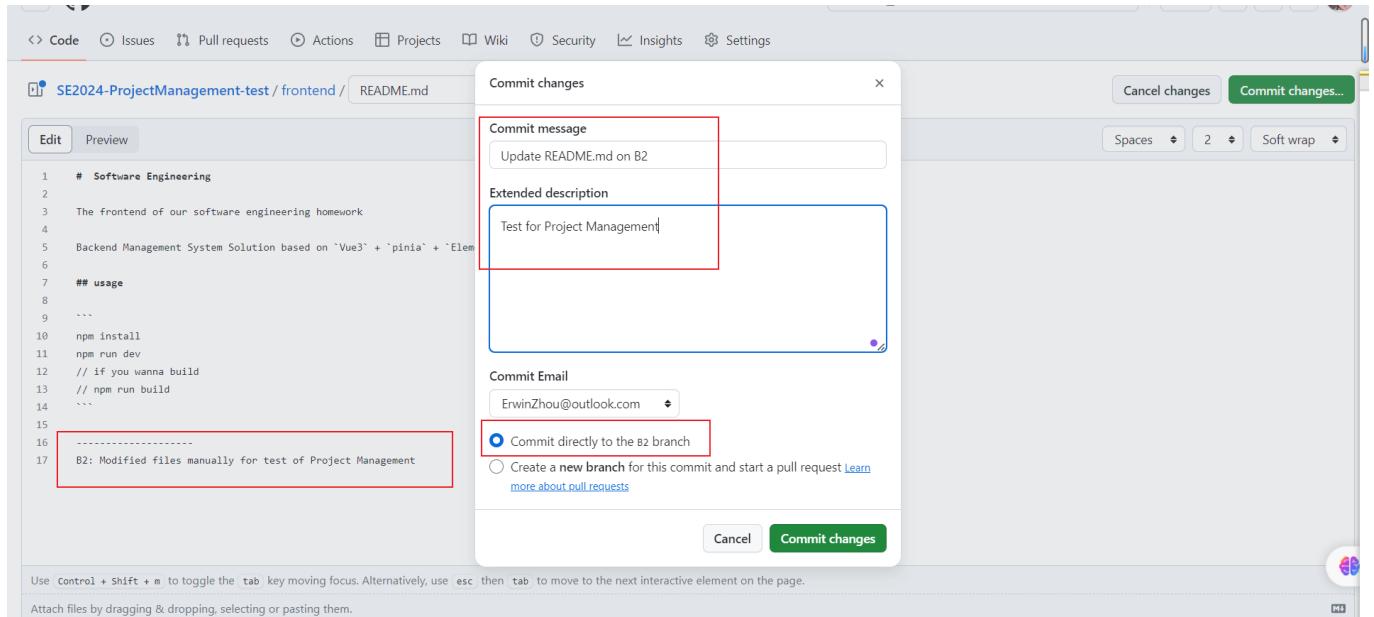


图 3.48: B2 上进行修改

如图3.48所示，首先在 B2 分支上的 frontend 文件夹下的 README.md 上加入一句对应的话，然后提交直接到 B2 分支上，同理对 B3 分支：

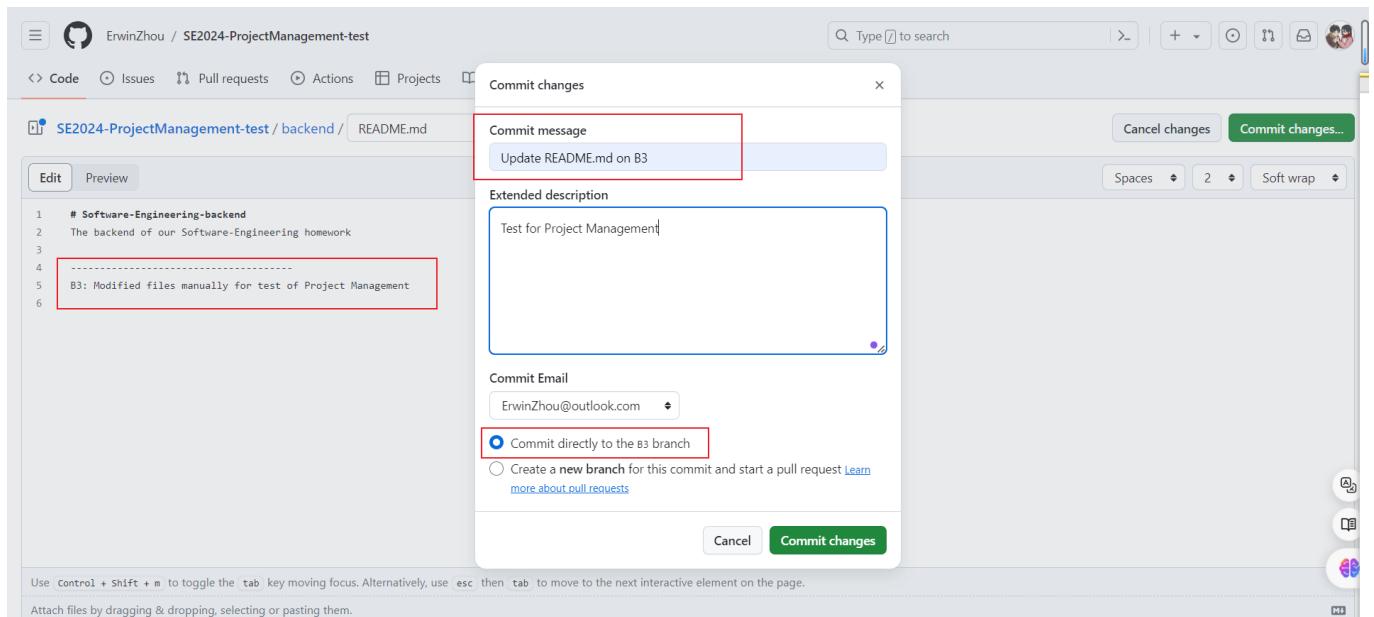


图 3.49: B3 上进行修改

The screenshot shows a GitHub repository page for 'SE2024-ProjectManagement-test'. At the top, there are two notifications: 'B2 had recent pushes 9 minutes ago' and 'B3 had recent pushes 11 seconds ago', both highlighted with red boxes. Below the notifications, there are buttons for 'Pin', 'Unwatch', 'Fork', and 'Star'. On the right side, there's an 'About' section with the repository name 'NKU-SoftwareEngineering2024 Individual Task Project Management' and links to 'Readme', 'MIT license', 'Activity', '0 stars', '1 watching', and '0 forks'. There are also sections for 'Releases' (no releases published) and 'Packages' (no packages published). The main content area shows a list of 7 commits, each with a file icon, commit message, and timestamp.

图 3.50: B2 和 B3 修改结果

如图3.49所示，对 B3 上的文件也进行了修改后。如图3.50，此时完成了预先任务，**可以正式开始实验场景二 R8-R15 的内容了。**

1. R8: 克隆仓库

R8 的任务是从 Github 上 (URL) 克隆一个已有的 git 仓库到本地，这里选择刚刚创建的仓库 **SE2024-ProjectManagement-test**。直接通过 `git clone` 加上对应的 URL 即可，命令为：

```
1 # 克隆 Github 远程仓库到本地
2 git clone https://github.com/ErwinZhou/SE2024-ProjectManagement-test
```

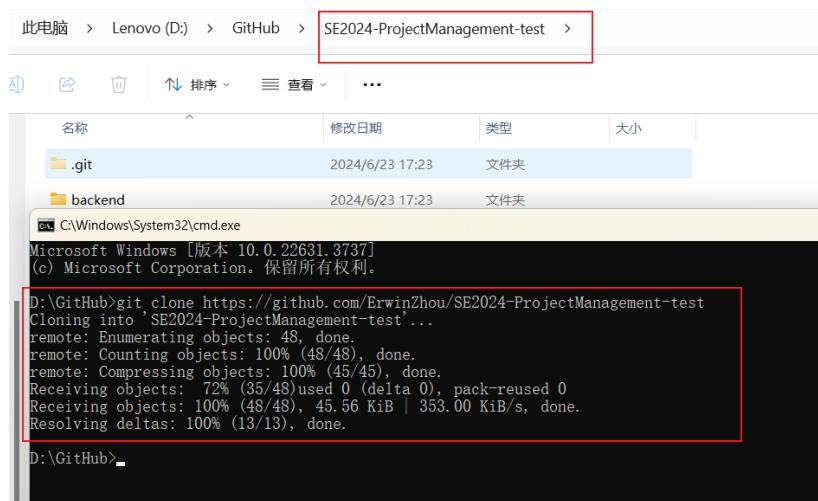


图 3.51: git clone 结果

如图3.51所示，可以看到远程服务器开始枚举对象 (`remote: Enumerating objects: 48, done.`)，并完成对象计数 (`remote: Counting objects: 100% (48/48), done.`)。随后，远程服务器压缩对象 (`remote: Compressing objects: 100% (45/45), done.`)。

Compressing objects: 100% (45/45), done.), 并开始接收对象。在接收过程中, 可以看到进度信息, 如 Receiving objects: 72% (35/48), 最终显示接收完成 (Receiving objects: 100% (48/48), 45.56 KiB | 353.00 KiB/s, done.)。最后, 分解增量完成 (Resolving deltas: 100% (13/13), done.)。这些信息表明仓库已成功克隆到本地, R8 完成。

2. R9: 获取分支

R9 要求获得该仓库的全部分支, 这里首先可以通过以下命令检测本地和远程的分支对应情况:

```

1 # 只显示远程分支, 适用于查看远程仓库中的所有分支
2 git branch -r
3 # 显示所有本地和远程分支, 适用于全面查看当前仓库中的所有分支
4 git branch -a

```

```

此电脑 > Lenovo (D:) > GitHub > SE2024-ProjectManagement-test >

名称 修改日期 类型 大小
.git 2024/6/23 17:23 文件夹

MINGW64:/d/GitHub/SE2024-ProjectManagement-test
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B1)
$ git branch -r
origin/B1
origin/B2
origin/B3
origin/HEAD -> origin/B1

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B1)
$ git branch -a
* B1
  remotes/origin/B1
  remotes/origin/B2
  remotes/origin/B3
  remotes/origin/HEAD -> origin/B1

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B1)
$ 

```

图 3.52: R9 查看分支获取情况

如图3.52所示, 可以看到通过上述两个命令获取本地对远程分支的克隆情况, 通过 `git branch -a` 命令查看所有本地和远程分支后, 输出结果显示当前本地分支和远程分支的信息。当前分支为 B1 (用 * 标记), 并且只有这么一个 B1。这是因为最开始克隆后, 所有远程分支的信息会被复制到本地, 但这些远程分支不会自动创建对应的本地分支。要想对这些远程分支进行操作, 必须进行“检出”(checkout)操作, 检出分支的目的是在本地创建一个与远程分支对应的分支, 并切换到该分支上。只有这样, 才能在本地对这些分支进行修改、提交等操作。我认为只有这样才是真正“获得”全部分支。

于是接下来依次检出三个分支, 通过以下命令:

```

1 git checkout -b B1 origin/B1
2 git checkout -b B2 origin/B2
3 git checkout -b B3 origin/B3

```

```

> 此电脑 > Lenovo (D:) > GitHub > SE2024-ProjectManagement-test >
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
排序 查看 ...
名称 修改日期 类型 大小
.git MINGW64:/d/GitHub/SE2024-ProjectManagement-test
|   origin/B2
|   origin/B3
|   origin/HEAD -> origin/B1
back
front
|.gitignore
LICENSE
README.md
nana
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B1)
$ git branch -a
* B1
  remotes/origin/B1
  remotes/origin/B2
  remotes/origin/B3
  remotes/origin/HEAD -> origin/B1

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B1)
$ git checkout -b B1 origin/B1
git checkout -b B2 origin/B2
git checkout -b B3 origin/B3
fatal: a branch named 'B1' already exists
Switched to a new branch 'B2'
branch 'B2' set up to track 'origin/B2'.
Switched to a new branch 'B3'
branch 'B3' set up to track 'origin/B3'.

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B3)
$ git branch -a
  B1
  B2
* B3
  remotes/origin/B1
  remotes/origin/B2
  remotes/origin/B3
  remotes/origin/HEAD -> origin/B1

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B3)
$ 

```

图 3.53: 检出 B2, B3

如图3.53所示，再次通过 `git branch -a` 命令查看所有分支后，输出结果显示当前仓库的本地和远程分支信息。本地分支包括 B1、B2 和当前所在分支 B3（用 * 标记）。远程分支包括 `remotes/origin/B1`、`remotes/origin/B2`、`remotes/origin/B3` 以及 `remotes/origin/HEAD -> origin/B1`。这些信息表明已经在本地成功创建并检出了与远程分支对应的分支 B1、B2 和 B3，并且当前位于 B3 分支上。这证明已经成功获得了该仓库的全部分支，并在本地可以进行相关操作。

3. R10: 创建新分支

接下来进行 R10 的任务，由于我已经在 R8 和 R9 上克隆并获取全部的分支，因此此处可以通过如下命令在 B2 分支的基础上创建一个新分支 C4：

```

1 # 切换到 B2 分支
2 git checkout B2
3 # 基于 B2 分支创建一个新分支 C4
4 git checkout -b C4
5 # 再次查看分支
6 git branch -a

```

```
MINGW64:/d/GitHub/SE2024-ProjectManagement-test
branch 'B3' set up to track 'origin/B3'.
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B3)
$ git branch -a
B1
B2
* B3
remotes/origin/B1
remotes/origin/B2
remotes/origin/B3
remotes/origin/HEAD -> origin/B1

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B3)
$ git checkout B2
Switched to branch 'B2'
Your branch is up to date with 'origin/B2'.

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B2)
$ git checkout -b C4
Switched to a new branch 'C4'

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (C4)
$ git branch -a
B1
B2
B3
* C4
remotes/origin/B1
remotes/origin/B2
remotes/origin/B3
remotes/origin/HEAD -> origin/B1
```

图 3.54: create C4

如图3.54所示，通过上述两个命令先从 B3 切换到 B2 再创建了新的分支。此时再通过 `git branch -a` 查看，可以看到当前已经存在了新的本地分支 C4 并且处在其上，R10 完成。

4. R11：修改并提交

现在是 R11，要求完成在 C4 上对 4 个文件进行修改并提交。首先通过 R0 的代码3.1查看 `git status`。

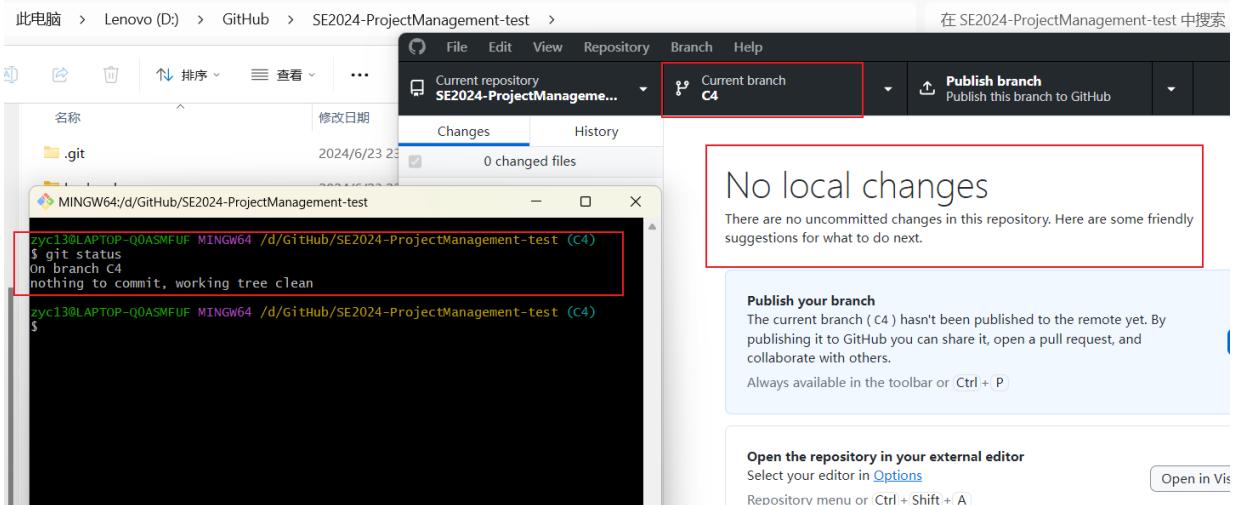


图 3.55: R11 修改前 git status 和 Github Desktop

如图3.55所示，在修改四个文件前，**本地没有任何更改，work tree 是干净的**。接下来手动对四个文件进行修改，修改后再次通过 git status, git diff 以及 Github Desktop 查看：

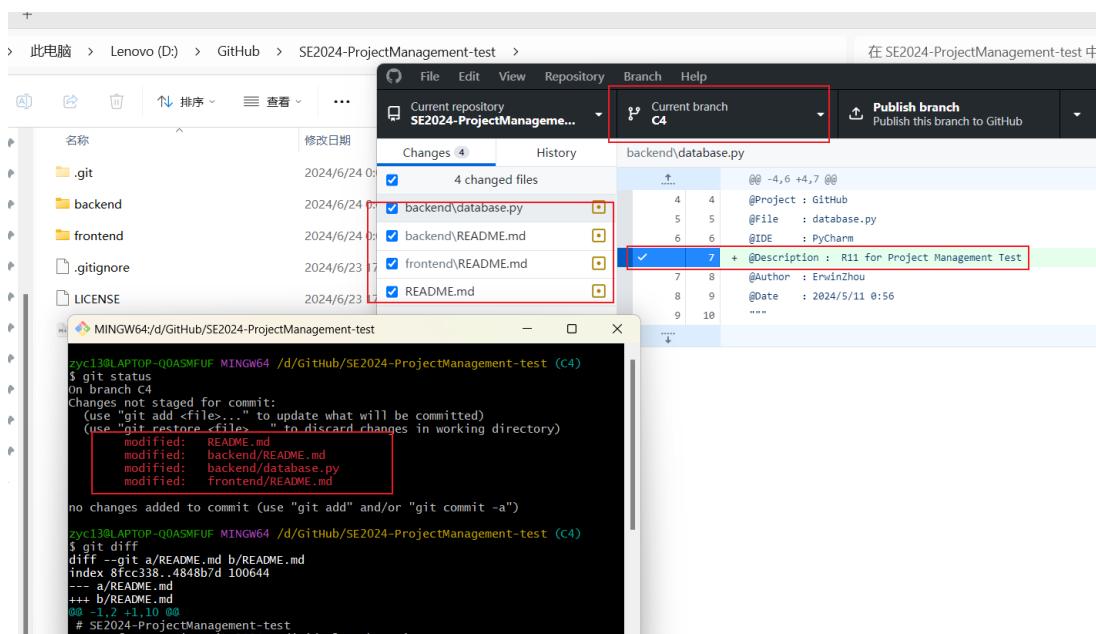


图 3.56: R11 修改后 git status 和 Github Desktop

如图3.56所示，可以看到我们还是在一些文件中加入了**R11 有关的文字注释标注修改**，在左面的 git diff 和 Github Desktop 能具体看到修改内容。现在让我们将文件修改提交，通过如下命令：

```

1 # 把修改加入到暂存区
2 git add README.md backend/database.py backend/README.md frontend/README.md
3 # 把修改提交
4 git commit -m "Updated README.md, backend/README.md, backend/database.py, and
   ↪ frontend/README.md for R11"

```

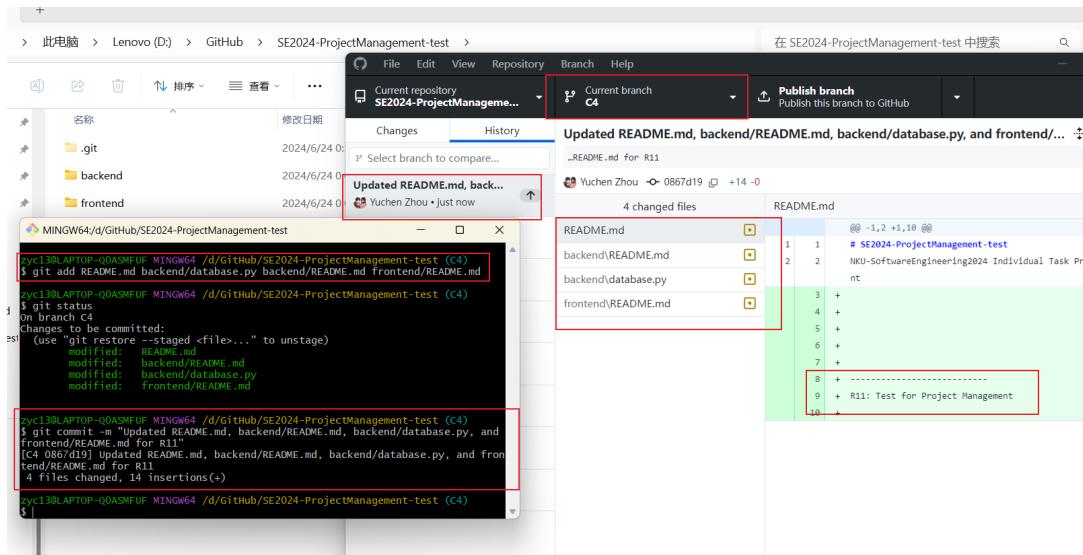


图 3.57: R11 提交修改后 git status 和 Github Desktop

如图3.57所示，现在已经成功在 C4 分支上完成了对 4 个文件进行修改并提交，到此 R11 完成。

5. R12: 修改并提交

现在完成 R12，即在 B3 分支上对同样的 4 个文件做不同修改并提交。首先切换到 B3 分支并查看状态：

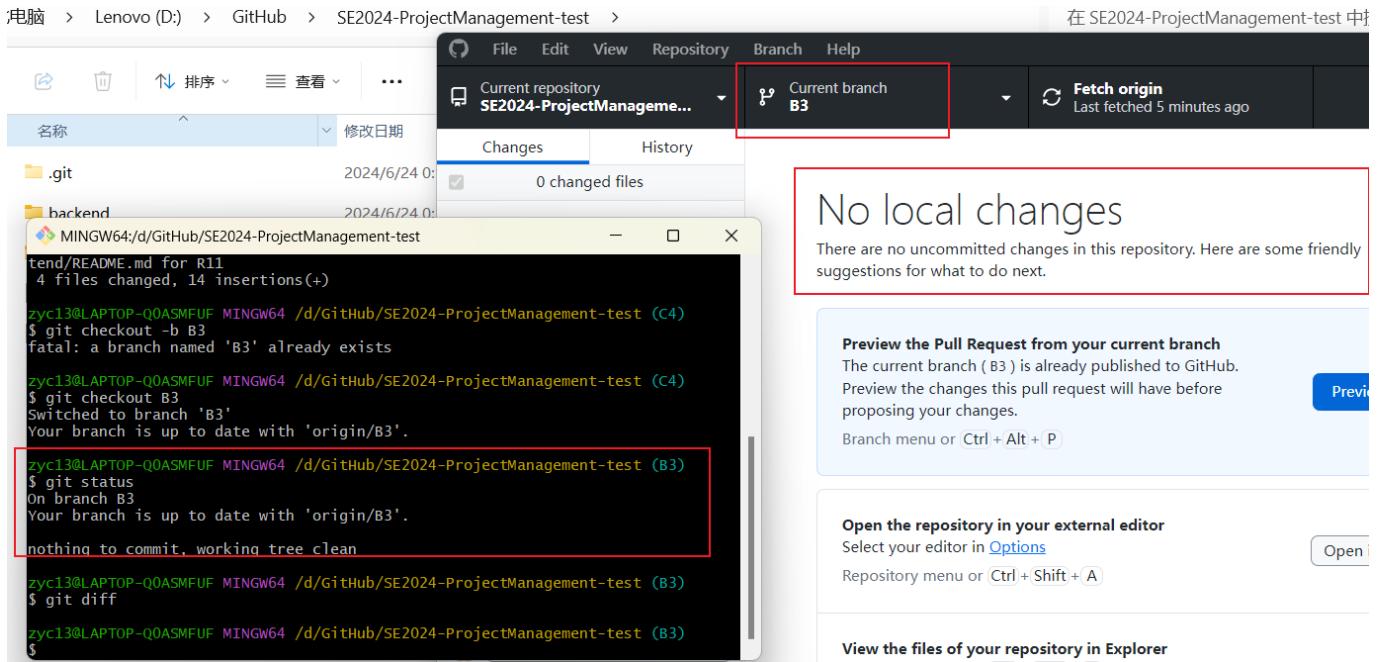


图 3.58: R12 修改前 git status 和 Github Desktop

如图3.58所示，此时还没有任何更改的痕迹，B3 分支上保持着 working tree clean。接下来同样地对 B3 分支上的 README.md, backend/database.py, backend/README.md, frontend/README.md 四个文件进行不同的更改。

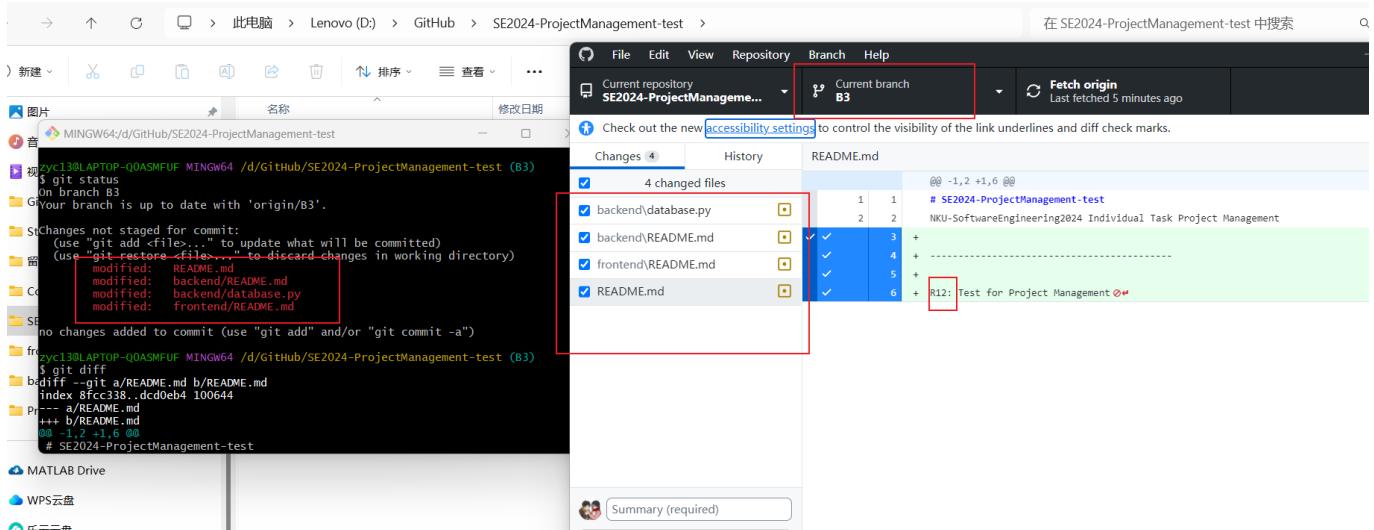


图 3.59: R12 修改后 git status 和 Github Desktop

如图3.59所示，现在已经修改后再次通过 `git status` 和 Github Desktop 查看状态，结果能看到这次的修改已经完成了，主要和上次的修改差异就在 R11 变成了 R12，所以就是不同的修改了。现在将刚刚的修改提交，通过命令：

```

1 # 把修改加入到暂存区
2 git add README.md backend/database.py backend/README.md frontend/README.md
3 # 把修改提交
4 git commit -m "Updated README.md, backend/README.md, backend/database.py, and
   → frontend/README.md for R12"

```

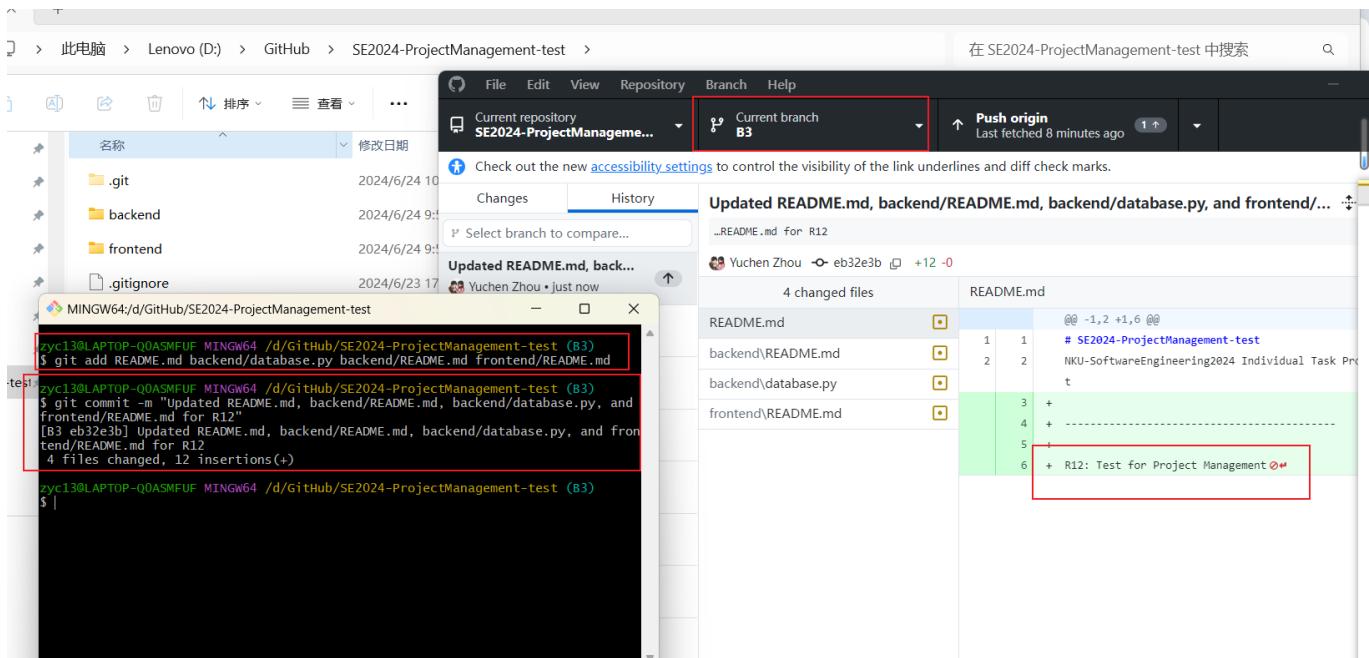


图 3.60: R12 提交后 git status 和 Github Desktop

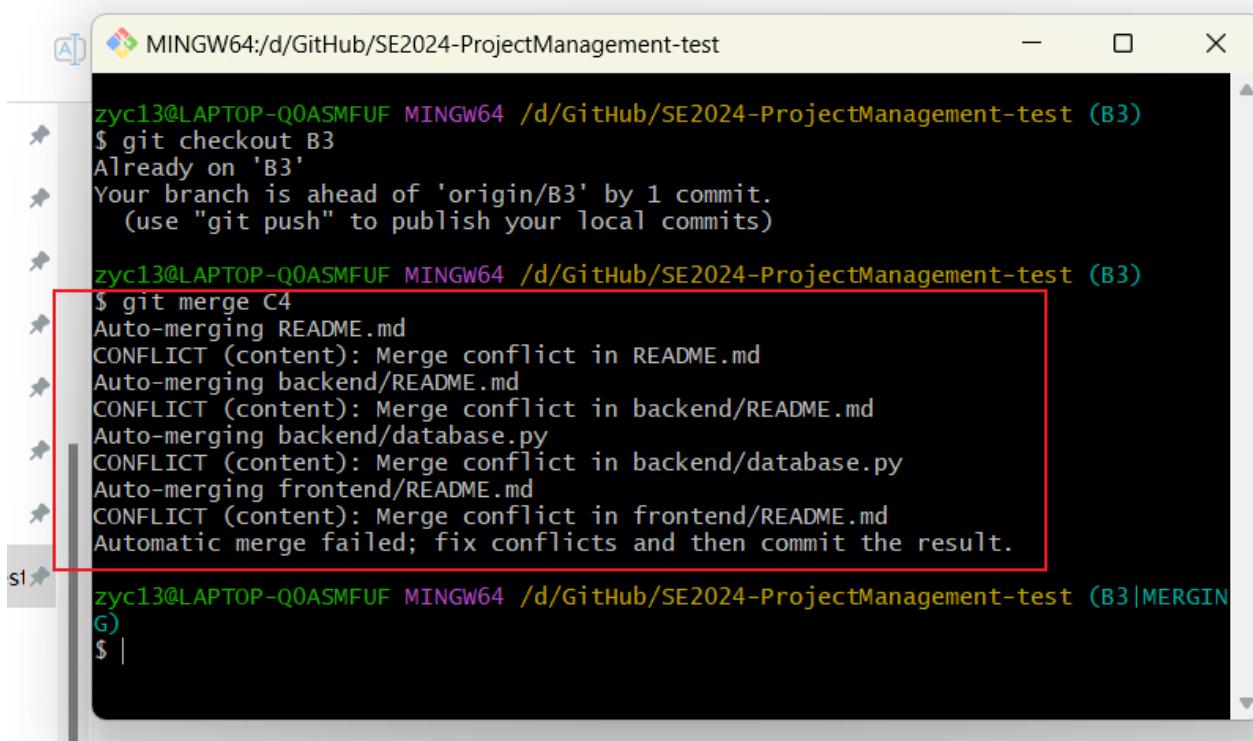
如图3.60所示，已经将 R12 中不同的修改提交到了 B3 分支，**R12 完成**。

6. R13: 合并分支

来完成 R13，R13 要求将 C4 和 B3 分支合并，若有冲突，手工消解。由于我们在 R11 和 R12 中通过不同的修改方式修改了四个相同文件，因此可以预料到一定会发生冲突，需要手工消解。并且实验要求没有标注将这两个分支中某个合并到另一个，后面默认将 C4 分支合并到 B3。首先通过如下命令：

```
1 # 切换到 B3 分支  
2 git checkout B3  
3 # 合并 C4 分支到 B3 分支  
4 git merge C4
```

> 此电脑 > Lenovo (D:) > GitHub > SE2024-ProjectManagement-test >



```
MINGW64:/d/GitHub/SE2024-ProjectManagement-test  
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B3)  
$ git checkout B3  
Already on 'B3'  
Your branch is ahead of 'origin/B3' by 1 commit.  
(use "git push" to publish your local commits)  
  
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B3)  
$ git merge C4  
Auto-merging README.md  
CONFLICT (content): Merge conflict in README.md  
Auto-merging backend/README.md  
CONFLICT (content): Merge conflict in backend/README.md  
Auto-merging backend/database.py  
CONFLICT (content): Merge conflict in backend/database.py  
Auto-merging frontend/README.md  
CONFLICT (content): Merge conflict in frontend/README.md  
Automatic merge failed; fix conflicts and then commit the result.  
  
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B3|MERGING)  
$ |
```

图 3.61: 初 git merge 结果

如图3.61所示，该输出结果显示合并过程中发生了一些冲突：

```
1 zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B3)  
2 $ git merge C4  
3 Auto-merging README.md  
4 CONFLICT (content): Merge conflict in README.md  
5 Auto-merging backend/README.md  
6 CONFLICT (content): Merge conflict in backend/README.md
```

```
7 Auto-merging backend/database.py
8 CONFLICT (content): Merge conflict in backend/database.py
9 Auto-merging frontend/README.md
10 CONFLICT (content): Merge conflict in frontend/README.md
11 Automatic merge failed; fix conflicts and then commit the result.
```

在合并过程中，Git 尝试自动合并以下文件，正是我们修改的几个文件：

- README.md
- backend/README.md
- backend/database.py
- frontend/README.md

然而每个文件都出现了内容冲突，分别显示为：

- CONFLICT (content): Merge conflict in README.md
- CONFLICT (content): Merge conflict in backend/README.md
- CONFLICT (content): Merge conflict in backend/database.py
- CONFLICT (content): Merge conflict in frontend/README.md

最后，Git 提示 Automatic merge failed; fix conflicts and then commit the result.，说明自动合并失败，需要手动解决冲突并提交结果。接下来需要通过如下命令：

```
1 # 查看冲突文件
2 git status
3 # 打开每个冲突文件，手动编辑解决冲突，移除冲突标记并保留正确的内容
4 <<<<< HEAD
5 当前分支 (B3) 的内容
6 ======
7 合并进来的分支 (C4) 的内容
8 >>>>> C4
9 # 将解决冲突的文件添加到暂存区
10 git add README.md backend/README.md backend/database.py frontend/README.md
11 # 完成合并
12 git commit
```

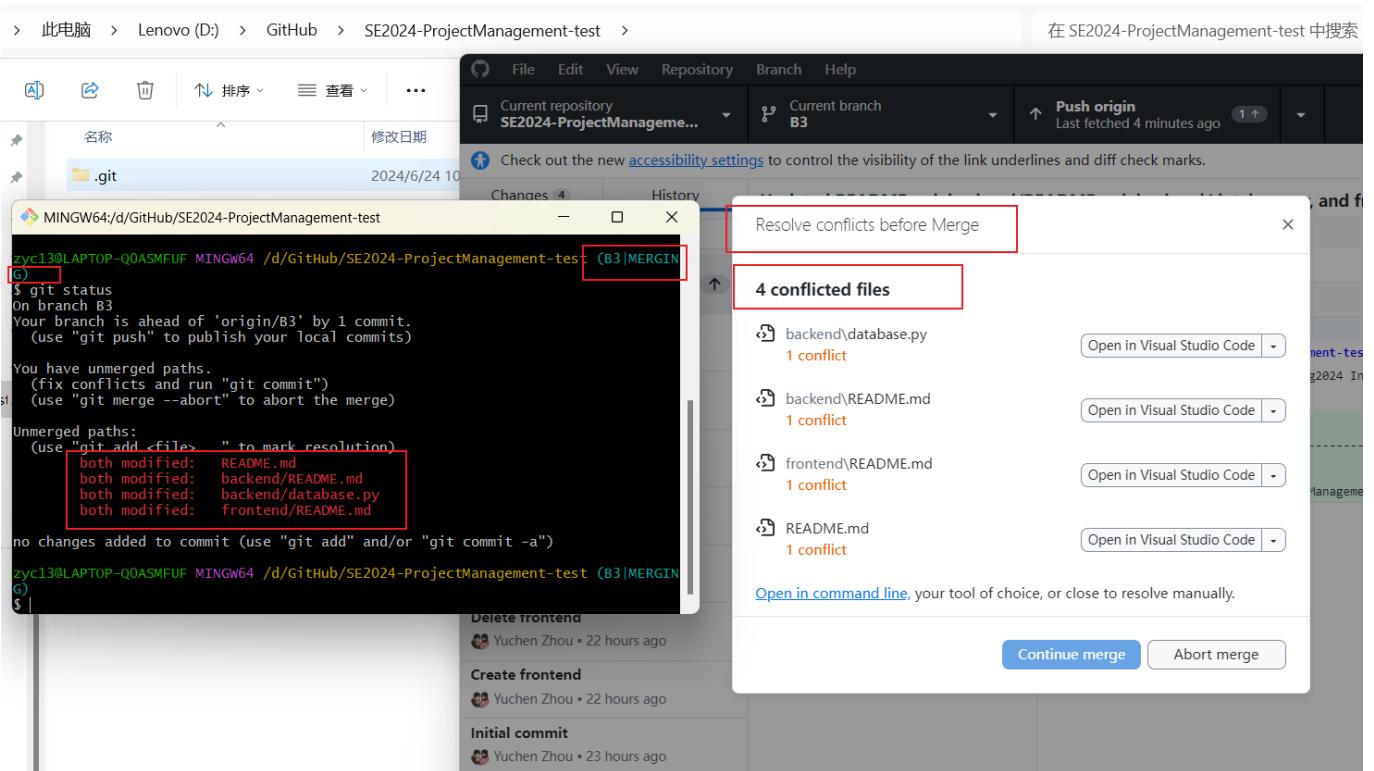


图 3.62: git status 和 Github Desktop 查看冲突状态

如图3.62所示，状态信息显示有一些未合并的路径（**You have unmerged paths**），需要手动解决冲突并运行 `git commit` 来完成合并。也可以使用 `git merge --abort` 来中止合并。这些文件在两个分支中都有修改（**both modified**），需要手动解决冲突。当前没有任何更改被添加到提交中（**no changes added to commit**），可以使用 `git add <file>...` 命令来标记冲突已解决并准备提交。接下来打开每个冲突文件，手动编辑解决冲突，移除冲突标记并保留正确的内容。

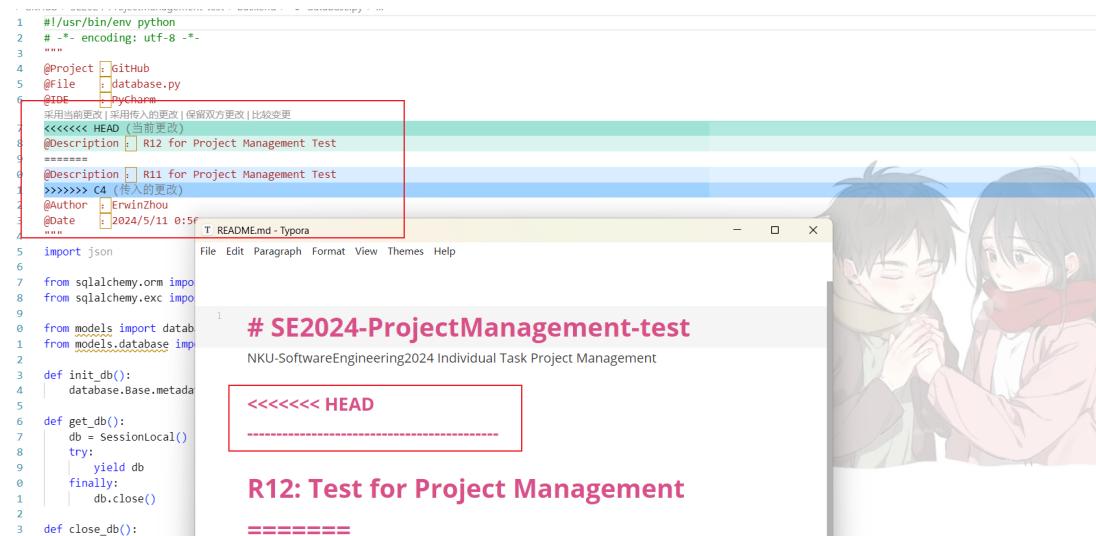


图 3.63: 查看冲突并解决

如图3.63所示，能清楚看到不论是 VSCode 中，还是 Typora 打开的 md 文件，都有冲突的标记。其中 VSCode 的标记更为清晰，我这里解决冲突选择对于 README.md, backend/database.py, backend/README.md,

frontend/README.md 的前两个文件保留 B3 更改，后两个文件保留 C4 更改。

冲突消解后再次加入暂存区并提交，结果为：

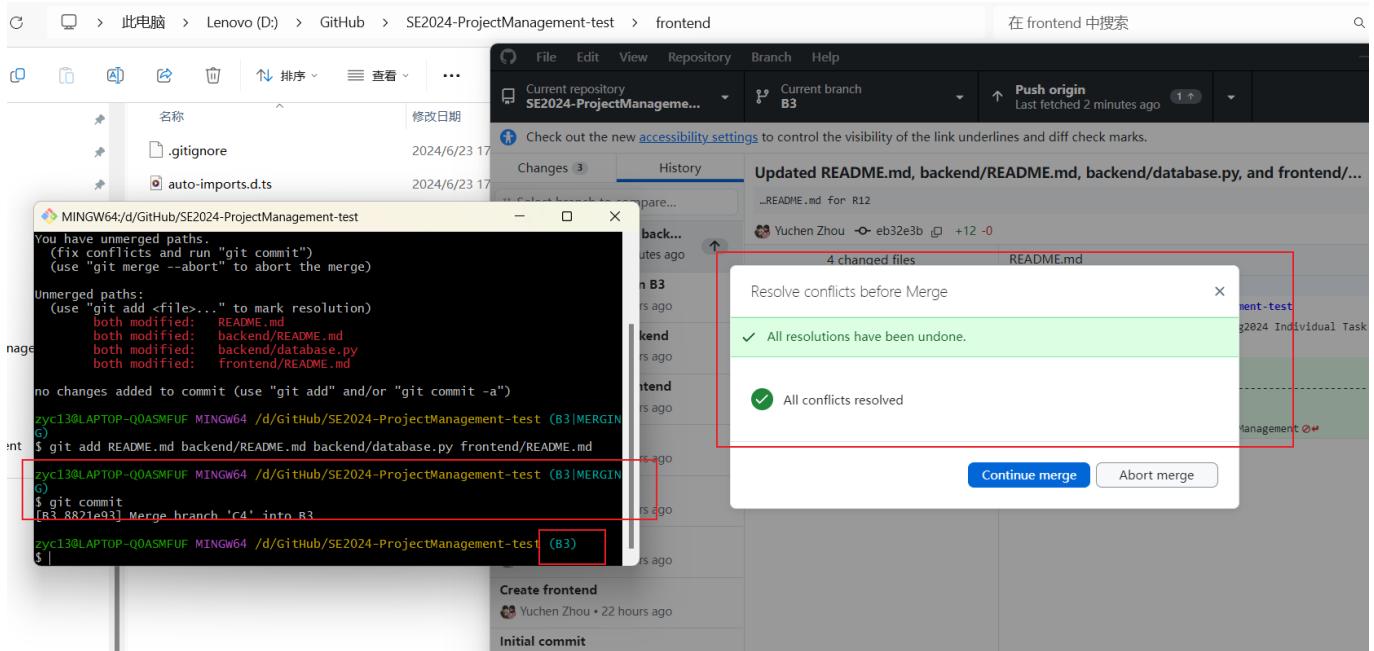


图 3.64: 冲突解决并合并完成

如图3.64所示，此时 `git commit` 结束后，对应的右侧 Github Desktop 已经显示了成功消解冲突，合并成功的提示，证明合并成功，R13 完成。

7. R14: 查看分支状态

R14 要求查看目前哪些分支已经合并、哪些分支尚未合并。这里可以直接通过命令查看：

```

1 # 查看已合并的分支
2 git branch --merged
3 # 查看未合并的分支
4 git branch --no-merged
5 # 查看详细合并历史
6 git log --graph --oneline --decorate

```

```

此电脑 > lenovo (D:) > GitHub > SE2024-ProjectManagement-test >
MINGW64:/d/GitHub/SE2024-ProjectManagement-test

configuration
error: no Jamfile in current directory found, and no target references specified
.

zyc13@LAPTOP-QOASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B3)
$ git branch --merged
bash: S: command not found

zyc13@LAPTOP-QOASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B3)
$ git branch --merged
  B1
* B3
  C4

zyc13@LAPTOP-QOASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B3)
$ git branch --no-merged
  B2

zyc13@LAPTOP-QOASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B3)
$ git log --graph --oneline --decorate
* 8821e93 (HEAD -> B3) Merge branch 'C4' into B3
|\ 
| * 0867d19 (C4) Updated README.md, backend/README.md, backend/database.py, and
|   frontend/README.md for R11
| * eb32e3b Updated README.md, backend/README.md, backend/database.py, and front
|   end/README.md for R12
| * b04661b (origin/B3) Update README.md on B3
|/
* 8bf40b0 (origin/HEAD, origin/B1, B1) Upload 10 files for backend
* 94a28e3 Upload 10 files for frontend
* f53f9f0 Create README.md
* b8c2b7d Create README.md
* 01b33f9 Delete frontend
* c6657b7 Create frontend
* 716aeff Initial commit

```

图 3.65: 查看合并分支情况

如图3.65所示，通过 `git branch --merged` 命令查看已合并到当前分支的所有分支，输出显示 B1、B3 和 C4 已经合并。这意味着当前分支 B3 包含了这些分支的提交历史。

通过 `git branch --no-merged` 命令查看尚未合并到当前分支的所有分支，输出显示 B2 尚未合并。这表明当前分支 B3 中没有包含 B2 的提交历史，B2 中的更改未被合并到 B3。

B1 为什么合并了？

这里第一次查看状态时，看到 B1 也被合并我是很奇怪的，经过查阅大量资料我发现：B1，它是最初的默认分支 `main` 改名而来的，因为在创建其他分支 B2 和 B3 时，都是基于 B1，所以其提交历史被其他分支继承。当 C4 在 B2 的基础上创建并进行合并操作时，B1 的提交历史也随之被包含在 C4 和 B3 的合并中，因此显示为已合并。

再如图3.65所示，看看详细的合并历史。

- ¹ zyc13@LAPTOP-QOASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B3)
- ² \$ git log --graph --oneline --decorate
- ³ * 8821e93 (HEAD -> B3) Merge branch 'C4' into B3
- ⁴ |\
- | * 0867d19 (C4) Updated README.md, backend/README.md, backend/database.py, and
→ frontend/README.md for R11

```

6 * | eb32e3b Updated README.md, backend/README.md, backend/database.py, and
   ↵ frontend/README.md for R12
7 * | b04661b (origin/B3) Update README.md on B3
8 |
9 * 8bf40b0 (origin/HEAD, origin/B1, B1) Upload 10 files for backend
10 * 94a28e3 Upload 10 files for frontend
11 * f53f9f0 Create README.md
12 * b8c2b7d Create README.md
13 * 01b33f9 Delete frontend
14 * c6657b7 Create frontend
15 * 716aeff Initial commit

```

- 当前分支 B3: HEAD → B3 指向提交 8821e93, 这是一个合并提交, 合并了 C4 分支到 B3, 合并消息为 Merge branch 'C4' into B3。
- C4 分支的提交: 提交 0867d19 是在 C4 分支上的修改, 描述为 Updated README.md, backend/README.md, backend/database.py, and frontend/README.md for R11。此提交已经合并到 B3。
- B3 分支的提交: 提交 eb32e3b 是在 B3 分支上的修改, 描述为 Updated README.md, backend/README.md, backend/database.py, and frontend/README.md for R12。提交 b04661b 是在远程 origin/B3 上的修改, 描述为 Update README.md on B3。
- 共同祖先 B1: 提交 8bf40b0 显示为 origin/HEAD, origin/B1, B1, 描述为 Upload 10 files for backend。这是最初的默认分支 main 改名为 B1 后的一个提交点。这个提交历史也被包含在后续分支的合并操作中, 因此 B1 显示为已合并。
- 其他初始提交: 提交 94a28e3、f53f9f0、b8c2b7d 等描述为 Upload 10 files for frontend、Create README.md 等, 显示了初始提交历史。

因此总的来说:

- (a) B1 显示为已合并是因为它是最初的默认分支, 所有后续创建的分支都是基于它, 并且它的提交历史被其他分支继承和合并。
- (b) C4 在 B2 基础上创建, 并最终合并到 B3, 所以 C4 显示为已合并。
- (c) B2 尚未合并到 B3, 所以显示为未合并。

当前的分支和提交历史显示, 所有操作步骤符合预期, B1、B3 和 C4 已经合并, 而 B2 尚未合并。通过这些分析, 可以清晰地理解分支的合并状态和提交历史。到此 R14 操作完成。

8. R15: 删除和合并分支

R15 要求将 C4 和 B3 合并后的分支删除, 将尚未合并的分支合并到一个新分支上, 分支名字为我自己的学号。本步操作比较多, 我这里先在这里将操作进行总结, 我的学号是 2111408, 尚未合并的分支根据之前的分析就是 B2:

```

1 # 切换到一个安全的分支 (B1)
2 git checkout B1

```

```

3
4 # 删除本地分支 B3
5 git branch -d B3
6
7 # 创建一个新的分支 2111408
8 git checkout -b 2111408
9
10 # 合并 B2 分支到新的分支 2111408
11 git merge B2
12
13 # 如果出现冲突，解决冲突并完成合并提交
14 # git add <resolved files>
15 # git commit

```

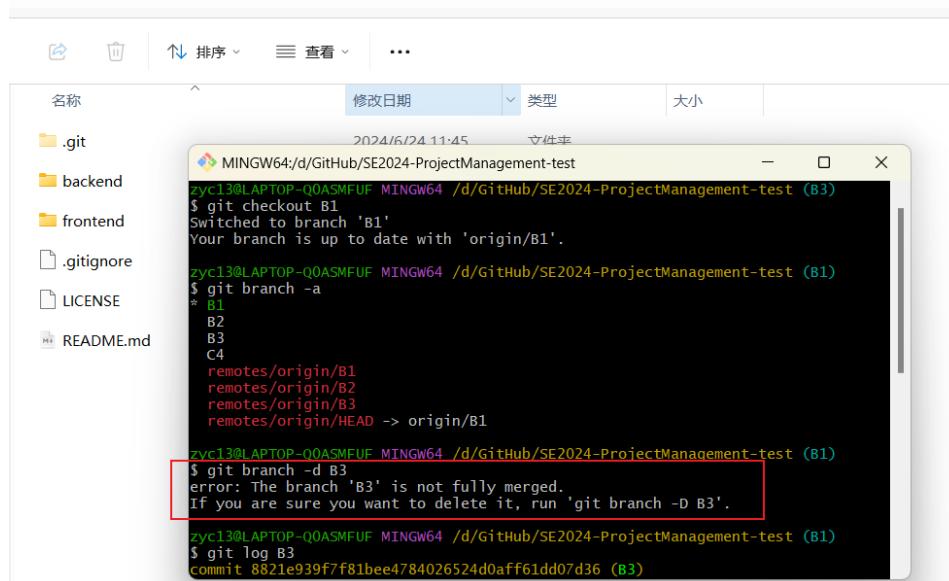


图 3.66: 删除时发现问题

如图3.66所示，在直接对 B3 进行删除时，提示 The branch 'B3' is not fully merged..。回顾图3.65，经过分析这是因为

- Git 检测到 B3 分支上有一些提交（例如 eb32e3b 和 b04661b）尚未合并到任何其他分支。这些提交代表了在 B3 分支上的独特更改。分别对应在实验场景二最开始 Web 进行 B3 分支更改时 b04661b 和 R12 即在 B3 分支上对同样的 4 个文件做不同修改并提交的 eb32e3b。
- 虽然 C4 已经合并到 B3，但 B3 上自己的独立提交并未合并到其他分支（例如 B1 或 B2）。

由于没有要求将 B3 上自己的独立的更改进行合并，这里无伤大雅，不需要将 B3 上的更改合并到 B1 或其他分支，正常直接继续删除即可，只需要将删除命令换为：

```
1 # 强制删除 B3 分支
```

```
2 git branch -D B3
```

The terminal window shows the following command and its output:

```
MINGW64:/d/GitHub/SE2024-ProjectManagement-test
$ git branch -D B3
Deleted branch B3 (was 8821e93).
```

After the deletion, the branch list shows B1, B2, and C4.

```
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B1)
$ git branch -a
* B1
  B2
  C4
```

Then, the user runs commands to find merged branches:

```
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B1)
$ git branch --merged
* B1
```

```
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B1)
$ git branch --no-merged
  B2
  C4
```

图 3.67: 删除分支后

如图3.67已经成功删除了 B3 和 C4 分支合并后的 B3，下面创建一个新的分支 2111408，并合并 B2 分支到新的分支 2111408。

The terminal window shows the following steps:

```
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B1)
$ git branch --no-merged
  B2
  C4
```

The user creates a new branch 2111408 and switches to it:

```
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (B1)
$ git checkout -b 2111408
Switched to a new branch '2111408'
```

Then, the user merges the B2 branch into 2111408:

```
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (2111408)
$ git merge B2
Updating 8bf40b0..b4e3fe4
Fast-forward
  Frontend/README.md | 2 ++
  1 file changed, 2 insertions(+)
```

Finally, the user checks the branch list again:

```
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/SE2024-ProjectManagement-test (2111408)
$ git branch -a
* 2111408
  B1
  B2
  C4
```

图 3.68: 合并学号分支后查看

如图3.68所示，通过 `git checkout -b 2111408` 命令创建一个新分支 `2111408` 并切换到该分支后，输出显示 **Switched to a new branch '2111408'**。这表明已经成功创建并切换到了新分支 `2111408`。

接下来，通过 `git merge B2` 命令将 `B2` 分支合并到 `2111408` 分支，输出显示合并操作为 **Fast-forward**，更新范围为 `8bf40b0..b4e3fe4`。在这次合并中，只有一个文件 `frontend/README.md` 发生了变化，进行了 2 处插入 (2 insertions)。由于这是一个快速合并 (**Fast-forward**)，表示合并过程中没有冲突，且直接将 `B2` 的提交历史移动到了 `2111408` 的头部。

最后查看分支状态，当前分支为 `2111408`，表明已经切换并在该分支上工作。本地分支列表显示仍然存在 `B1`、`B2` 和 `C4`，但 `B3` 已被删除。远程分支列表显示远程仓库的所有分支，包括 `origin/B1`、`origin/B2`、`origin/B3`，以及指向 `origin/B1` 的 `origin/HEAD`。

到此说明已经成功创建了以学号命名的新分支 `2111408`，并将尚未合并的 `B2` 分支合并到这个新分支上。合并过程中没有冲突，且成功应用了 `B2` 的更改。当前本地和远程分支状态清晰，显示了所有分支的情况，证明已经完成了 R15 的要求。

到此实验场景二，即分支管理的实验内容全部完成。

3.3 实验场景三：远程分支管理

在完成了实验场景一和二后，最后来完成实验场景三，其中涉及步骤 R16-R18。

实验场景三内容

- **R16: 推送分支**
 - 将本地以你的学号命名的分支推送到 Github 上。
- **R17: 推送结果**
 - 将 R1 到 R7 各步骤得到的结果推送到 Github 上。
- **R18: 查看状态**
 - 在 Github 网站以 web 页面的方式查看你的两个仓库的当前状态。

1. R16: 推送分支

首先在实验场景二的 R15 中，我创建了以我自己的学号即 `2111408` 命名的分支，并将其与 `B2` 进行了合并，接下来进行将本地的学号命名的分支推送到 Github 上，对应的仓库 URL 地址应该为 [SE2024-ProjectManagement-test](#)。

¹ # 将本地的学号命名的分支推送到 Github 上

² git push origin 2111408

The screenshot shows a GitHub repository page for 'SE2024-ProjectManagement-test'. A yellow box highlights the message '2111408 had recent pushes 24 seconds ago'. The terminal window shows the command history for pushing the branch:

```

$ git remote -v
origin https://github.com/ErwinZhou/SE2024-ProjectManagement-test (fetch)
origin https://github.com/ErwinZhou/SE2024-ProjectManagement-test (push)

$ git branch -a
* 2111408
  B1
  B2
  B3

$ git push origin 2111408
remote: unable to access 'https://github.com/ErwinZhou/SE2024-ProjectManagement-test': SSL certificate problem: self signed certificate in certificate chain
Initial sync: 100% (0/0), done.
remote: Create a pull request for "2111408" on GitHub by visiting:
remote:   https://github.com/ErwinZhou/SE2024-ProjectManagement-test/pull/new/2111408
remote: 
remote:   or
remote:   https://github.com/ErwinZhou/SE2024-ProjectManagement-test
remote: [new branch] 2111408 -> 2111408

```

The terminal output indicates a SSL certificate error during the push operation. The message also suggests creating a pull request for the new branch.

图 3.69: R16 操作后结果

如图3.69所示，通过 `git push origin 2111408` 命令将本地分支 2111408 推送到远程仓库，输出显示推送操作成功，远程仓库上创建了一个新分支 2111408。推送过程中没有任何对象被传输（`Total 0 (delta 0)`, `reused 0 (delta 0)`, `pack-reused 0`），这意味着本地和远程的提交记录已经同步。远程仓库提供了一个链接，提示可以通过访问该链接在 GitHub 上为 2111408 分支创建一个拉取请求（Pull Request）。最终输出确认分支 2111408 已经成功推送到远程仓库 ErwinZhou/SE2024-ProjectManagement-test。

这说明 2111408 分支已经被成功推到了 Github 上，我通过 web 界面和 `git push` 的成功消息都可以验证成功。期间还遇到了 Github 经典的断网问题 hhh。R16 完成。

2. R17: 推送结果

接下来进行 R17，要求将 R1 到 R7 各步骤得到的结果推送到 Github 上。由于我 R1-R7 都是在本地仓库进行的操作，因此首先第一步先是要建立一个和本地仓库 Software-Engineering-project-management-test 同名的远程仓库即 Software-Engineering-project-management-test。

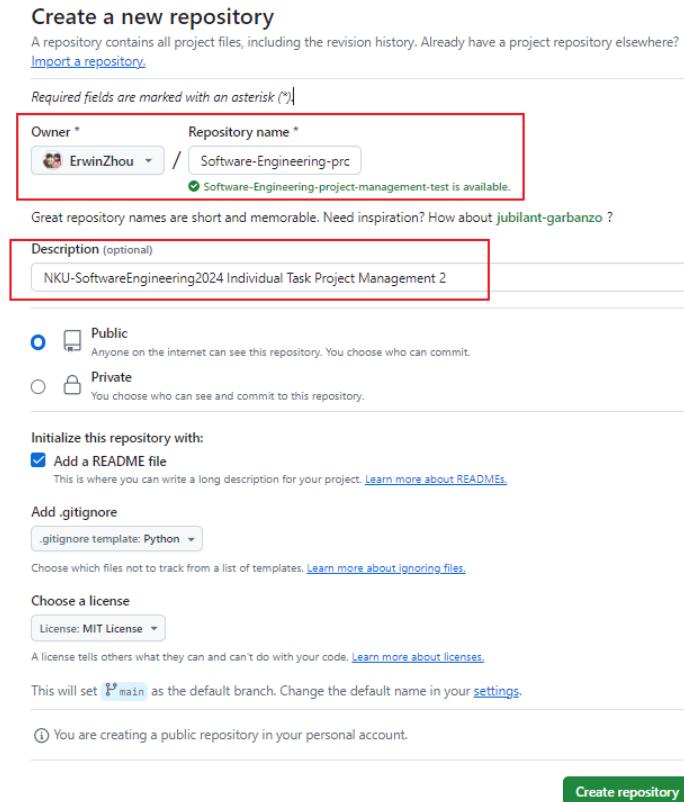


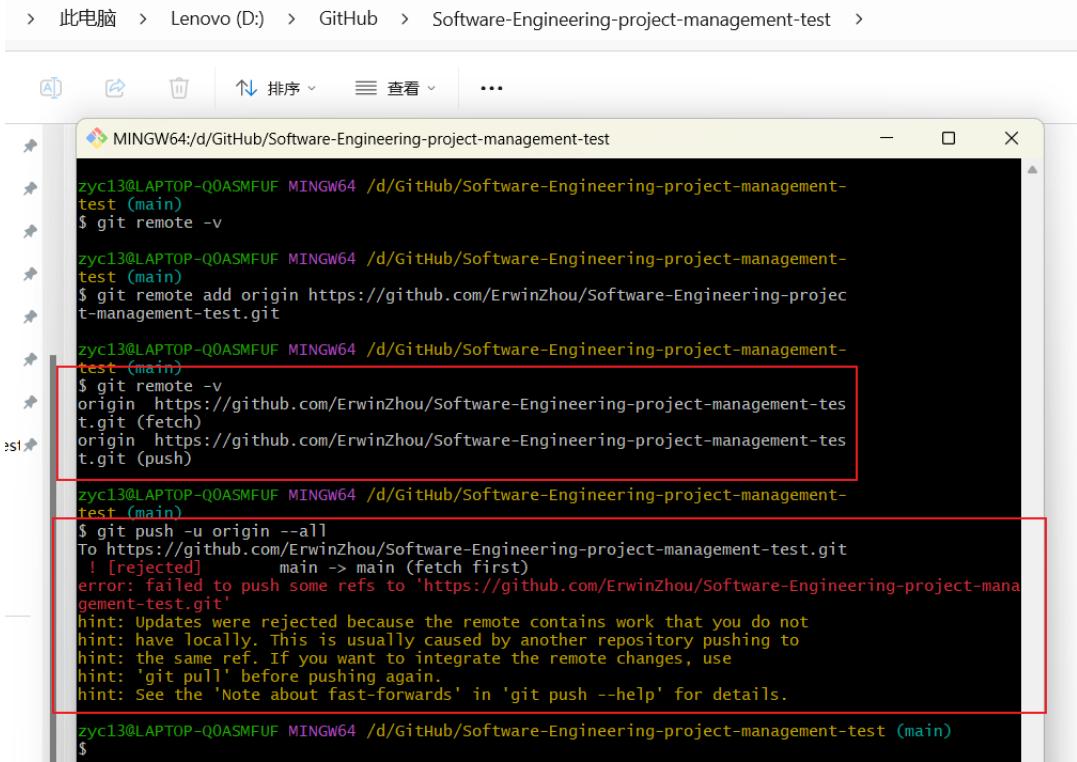
图 3.70: 创建远程仓库 Software-Engineering-project-management-test

如图3.70所示，远程同名仓库已经建立完成，现在需要将本地该仓库与远程仓库建立关联，并进行具体的推送，总的来说命令为：

```

1 # 0. 查看本地仓库与远程仓库的关联情况
2 git remote -v
3
4 # 1. 在本地仓库的根目录中，关联远程仓库
5 git remote add origin
   ↳ https://github.com/ErwinZhou/Software-Engineering-project-management-test.git
6
7 # 2. 推送所有分支和提交到远程仓库
8 git push -u origin --all
9
10 # 3. 如果有标签，推送所有标签到远程仓库
11 git push --tags

```



The screenshot shows a terminal window titled 'MINGW64:/d/GitHub/Software-Engineering-project-management-test'. The user has run several commands:

```
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git remote -v
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git remote add origin https://github.com/ErwinZhou/Software-Engineering-project-management-test.git
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git remote -v
origin https://github.com/ErwinZhou/Software-Engineering-project-management-test.git (fetch)
origin https://github.com/ErwinZhou/Software-Engineering-project-management-test.git (push)
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git push -u origin --all
To https://github.com/ErwinZhou/Software-Engineering-project-management-test.git
 ! [rejected]          main -> main (fetch first)
error: failed to push some refs to 'https://github.com/ErwinZhou/Software-Engineering-project-management-test.git'
hint: Updates were rejected because the remote contains work that you do not
hint: have locally. This is usually caused by another repository pushing to
hint: the same ref. If you want to integrate the remote changes, use
hint: 'git pull' before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
```

The last command, `git push -u origin --all`, fails with a rejection message. A red box highlights the error output:

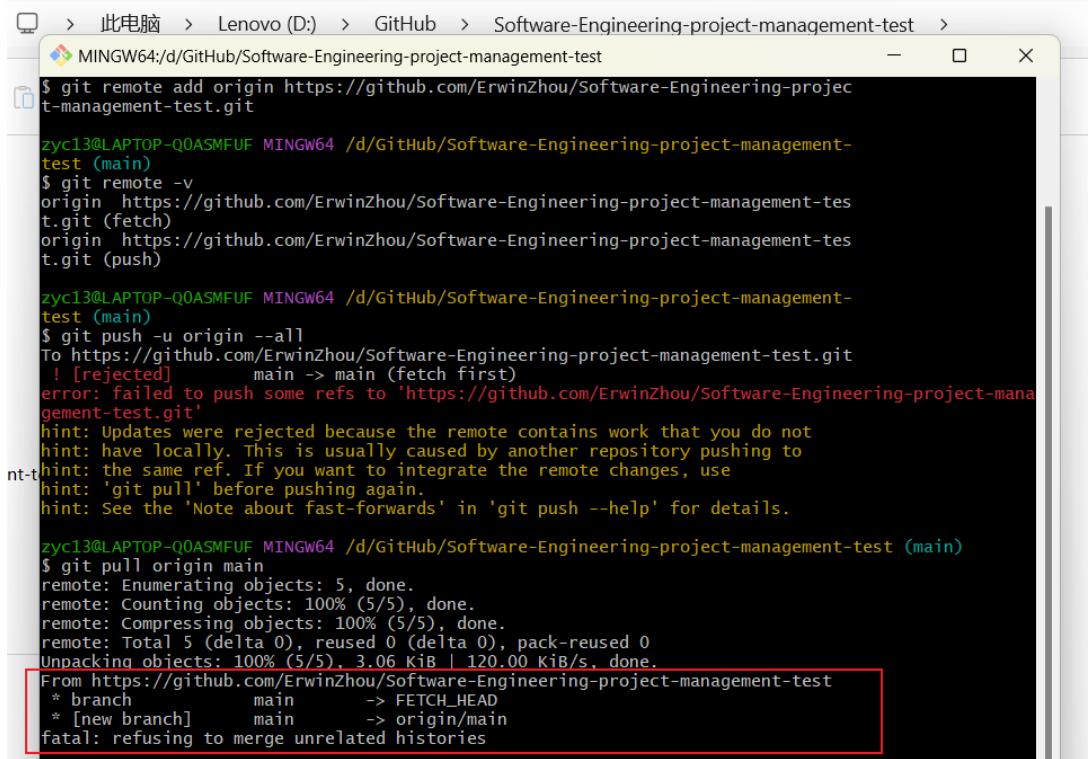
```
! [rejected]          main -> main (fetch first)
error: failed to push some refs to 'https://github.com/ErwinZhou/Software-Engineering-project-management-test.git'
hint: Updates were rejected because the remote contains work that you do not
hint: have locally. This is usually caused by another repository pushing to
hint: the same ref. If you want to integrate the remote changes, use
hint: 'git pull' before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

图 3.71: 出现错误

如图3.71所示，通过 `git push -u origin --all` 命令推送到远程仓库时，出现错误提示：由于远程仓库中的 `main` 分支包含本地仓库没有的更新，导致推送被拒绝。这个错误通常是因为其他人在远程仓库中进行了推送操作，而本地仓库未同步这些更改。但是刚刚创建好仓库，因此我推测原因是因为我刚新建的该仓库中的 `README.md`, `License` 等都和本地的仓库不同。

解决此问题的方法是先通过 `git pull` 命令将远程仓库的更改拉取到本地并合并，如果有冲突需要手动解决，完成合并后再重新进行推送。

```
1 # 拉取远程仓库的更改并合并
2 git pull origin main
3
4 # 如果有冲突，解决冲突并完成合并提交
5 # git add <resolved files>
6 # git commit
7
8 # 重新推送到远程仓库
9 git push -u origin --all
```



```

MINGW64:/d/GitHub/Software-Engineering-project-management-test>
$ git remote add origin https://github.com/ErwinZhou/Software-Engineering-project-management-test.git
zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git remote -v
origin https://github.com/ErwinZhou/Software-Engineering-project-management-test.git (fetch)
origin https://github.com/ErwinZhou/Software-Engineering-project-management-test.git (push)

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git push -u origin --all
To https://github.com/ErwinZhou/Software-Engineering-project-management-test.git
 ! [rejected]      main -> main (fetch first)
error: failed to push some refs to 'https://github.com/ErwinZhou/Software-Engineering-project-management-test.git'
hint: Updates were rejected because the remote contains work that you do not
hint: have locally. This is usually caused by another repository pushing to
hint: the same ref. If you want to integrate the remote changes, use
hint: 'git pull' before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

zyc13@LAPTOP-Q0ASMFUF MINGW64 /d/GitHub/Software-Engineering-project-management-test (main)
$ git pull origin main
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (5/5), 3.06 KiB | 120.00 KiB/s, done.
From https://github.com/ErwinZhou/Software-Engineering-project-management-test
 * branch            main      -> FETCH_HEAD
 * [new branch]      main      -> origin/main
fatal: refusing to merge unrelated histories

```

图 3.72: 又次出现错误

如图3.72所示，通过 `git pull origin main` 命令从远程仓库拉取更改时，再次出现了错误提示：`refusing to merge unrelated histories`。这意味着本地仓库和远程仓库的历史记录没有共同的基础提交。通常发生在分别创建了本地和远程仓库，并在没有共同历史的情况下尝试合并它们。这也与我们的情况相符合，因此直接拉取并允许合并不相关的历史。

```

1 # 暂存本地更改
2 git stash
3
4 # 拉取远程更改并允许合并不相关的历史
5 git pull origin main --allow-unrelated-histories
6
7 # 恢复暂存的更改
8 git stash pop
9
10 # 如果有冲突，解决冲突并完成合并提交
11 # git add <resolved files>
12 # git commit
13
14 # 重新推送到远程仓库
15 git push -u origin --all

```

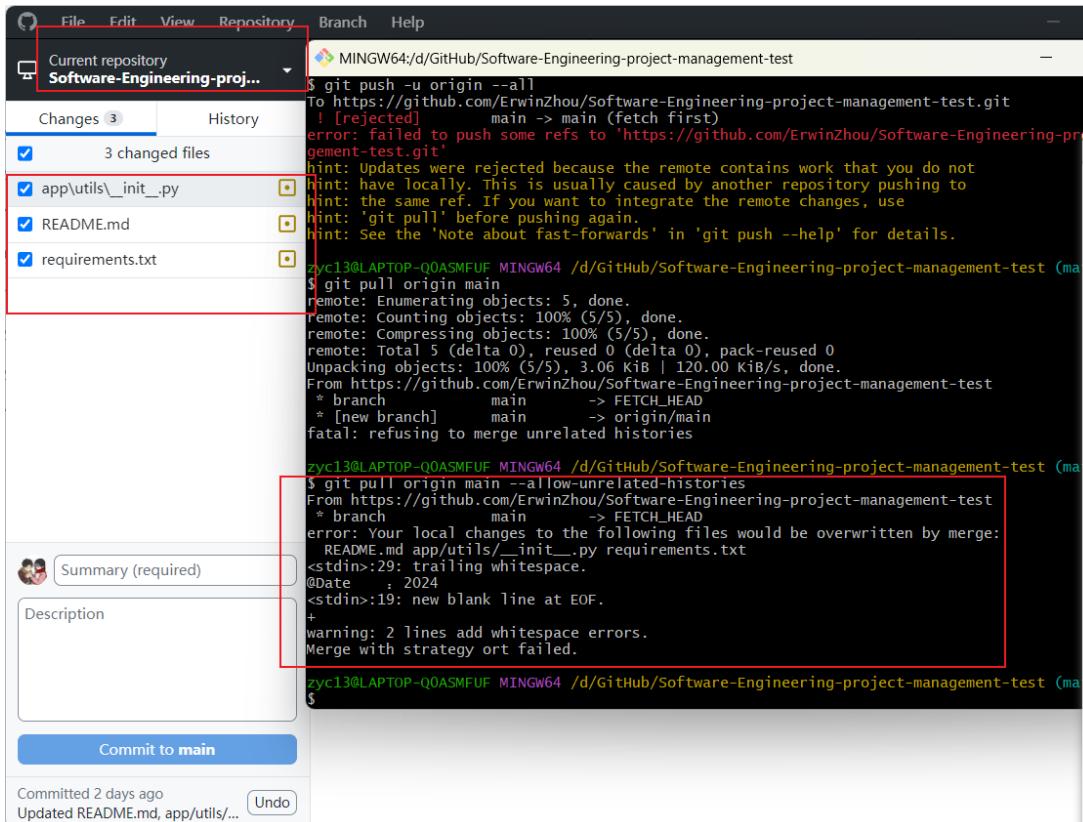


图 3.73: 又双囧囧出现了错误

如图3.73所示，又双囧囧出现了错误，通过 `git pull origin main --allow-unrelated-histories` 命令拉取远程更改时，出现了错误提示：**Your local changes to the following files would be overwritten by merge**。这意味着本地文件的更改将被合并操作覆盖。要解决这个问题，需要先处理本地未提交的更改，然后再进行拉取和合并。这是因为我们在 R6 中将本地 R5 的提交撤销了，还没有提交上去，因此遗留到这里的问题，因此真正完整的命令为：

```

1 # 暂存本地更改
2 git stash
3
4 # 拉取远程更改并允许合并不相关的历史
5 git pull origin main --allow-unrelated-histories
6
7 # 恢复暂存的更改
8 git stash pop
9
10 # 如果有冲突，解决冲突并完成合并提交
11 # git add <resolved files>
12 # git commit -m "Push R1-R7 to remote repository"
13
14 # 重新推送到远程仓库

```

15 git push -u origin --all

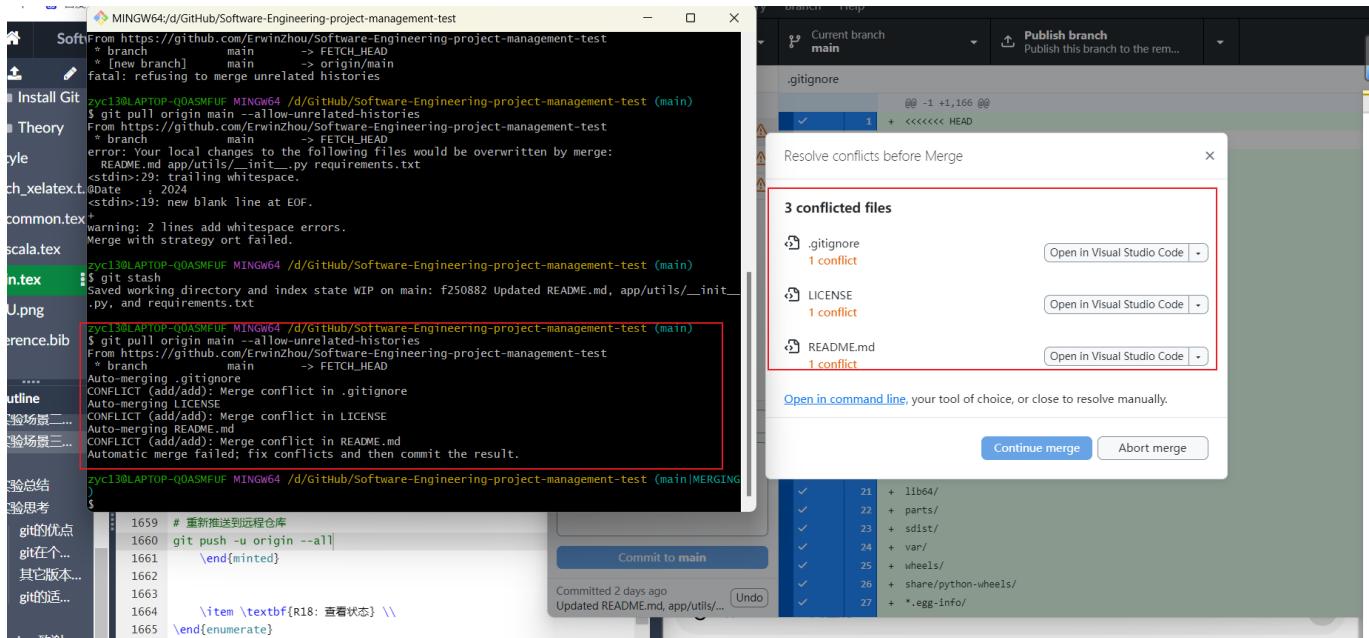


图 3.74: 如同预期出现冲突

如图3.74所示，如同预期出现了冲突，下面依次解决，解决后添加到暂存区，提交并最后重新推送，结果如下：

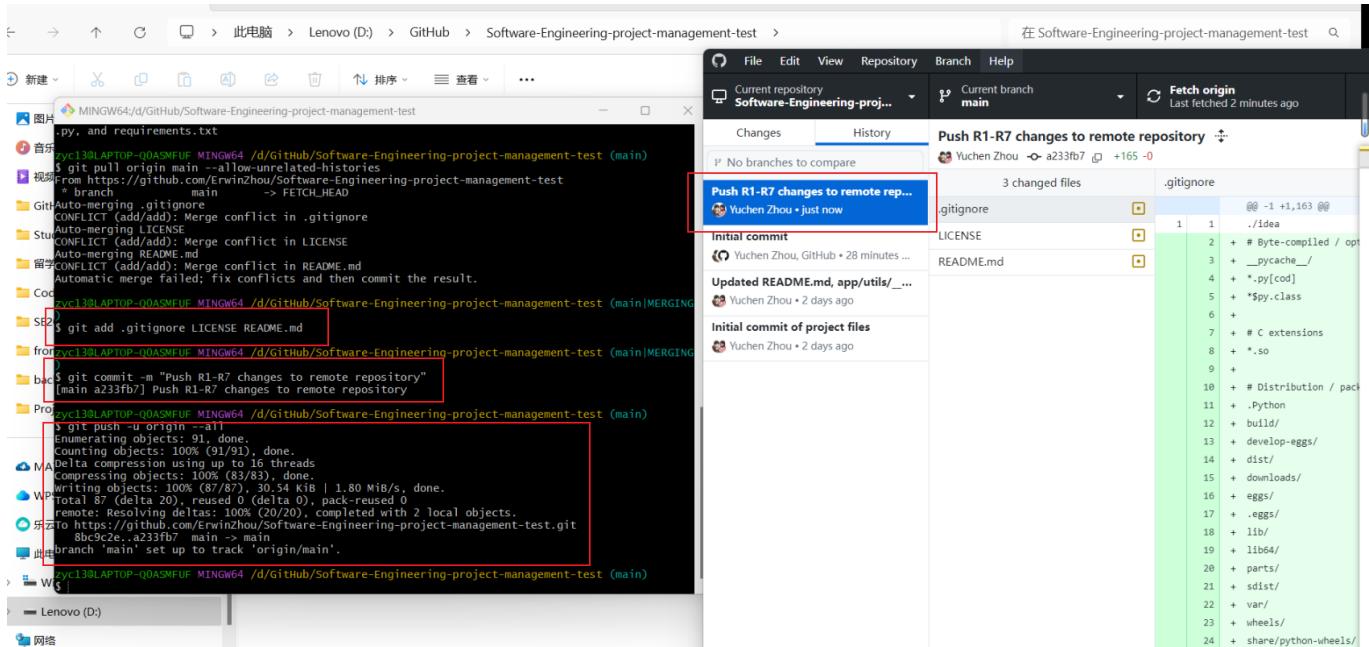


图 3.75: 终于成功了哭哭 1

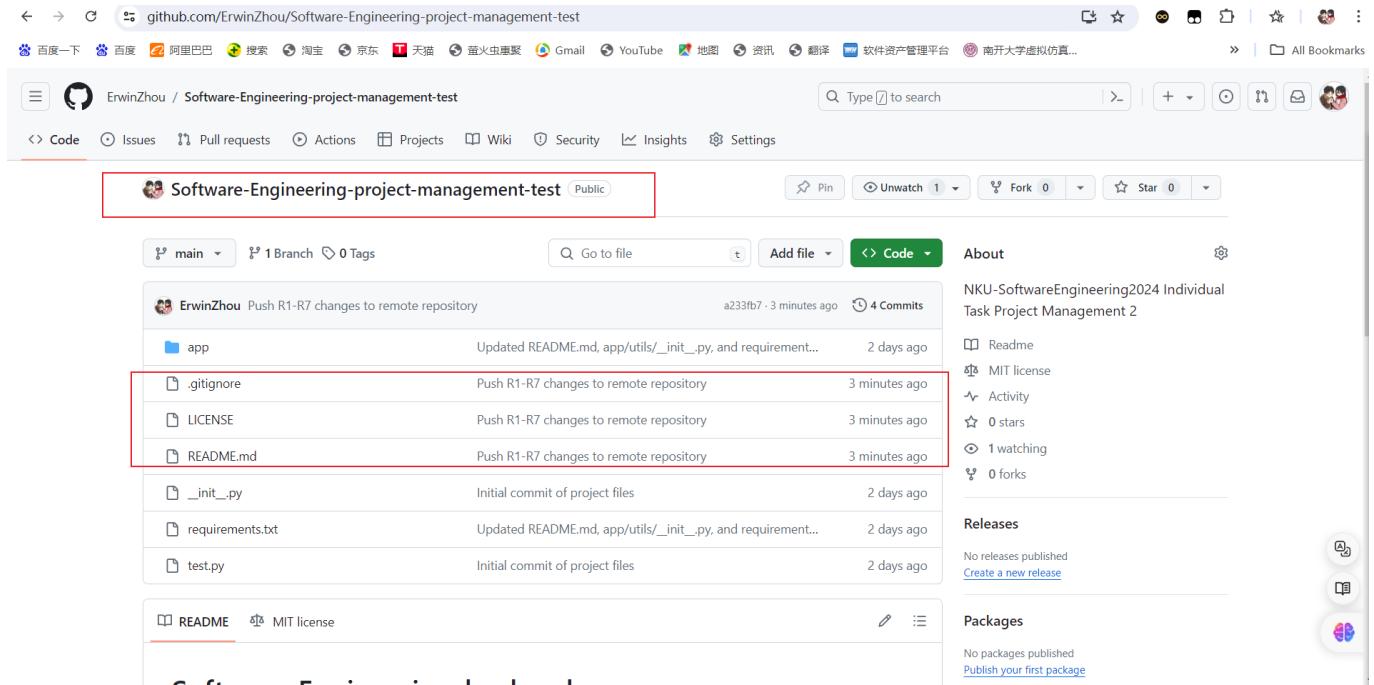


图 3.76: 终于成功了哭哭 2

如图3.75和3.76所示, `git push` 能够看到成功的信息, Github 远程仓库也能够看到刚推送的信息。终于完成了将 R1 到 R7 各步骤得到的结果推送到 Github 的远程仓库SE2024-ProjectManagement-test上。到此 R17 结束!

3. R18: 查看状态

最后的最后, 让我们来到 R18, 只需在 Github 网站以 web 页面的方式查看两个仓库的当前状态。两个仓库分别为实验场景一涉及的Software-Engineering-project-management-test, 实验场景二涉及的SE2024-ProjectManagement-test, 如表1所述。现在直接利用 Web 界面查看下它们各自的状态:

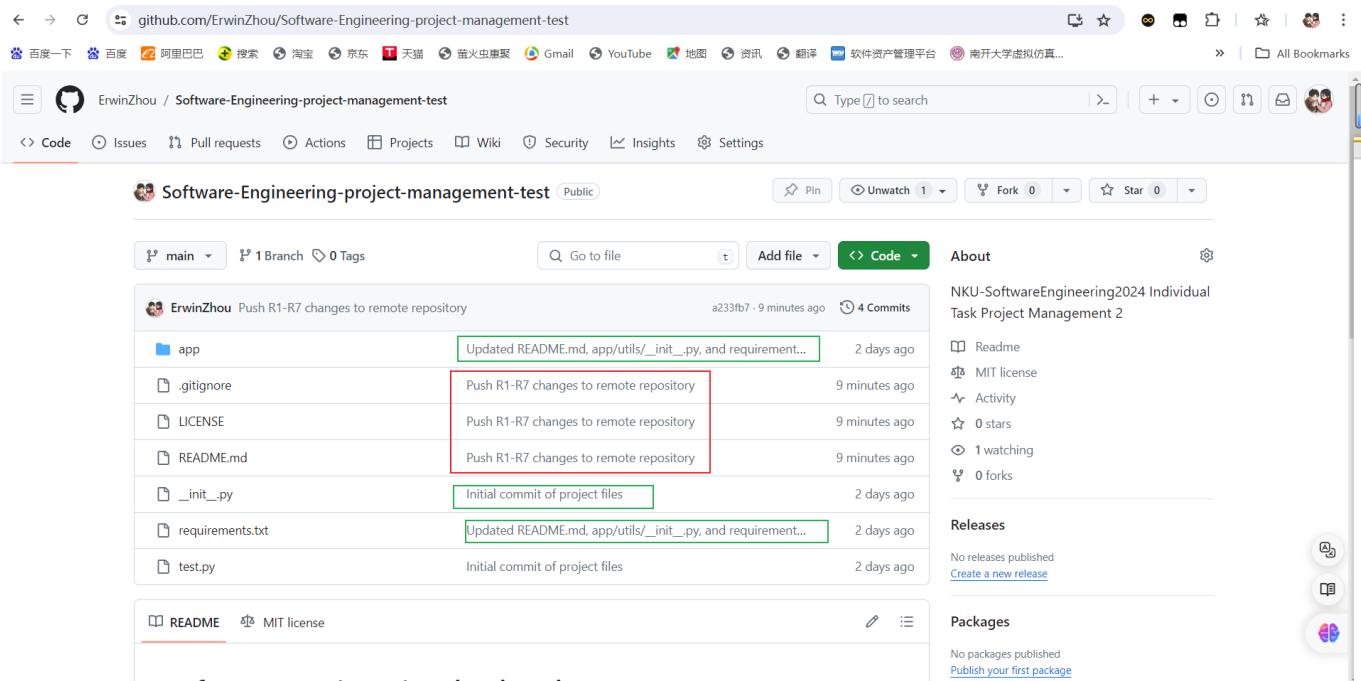


图 3.77: Software-Engineering-project-management-test 仓库状态

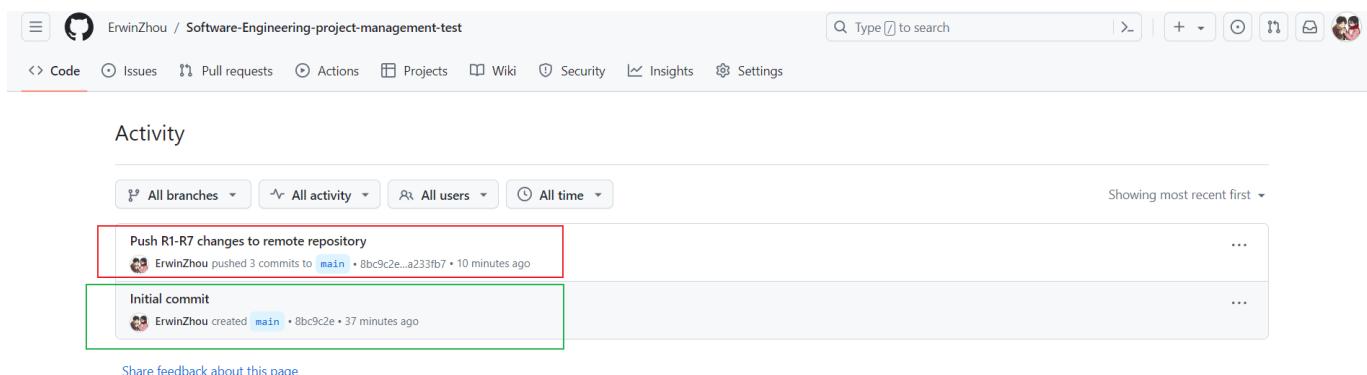


图 3.78: Software-Engineering-project-management-test 仓库活动

如图3.77和3.78所示，能看到实验场景一涉及的仓库Software-Engineering-project-management-test的状态和活动，与我们的操作完全一致，可以得到验证。

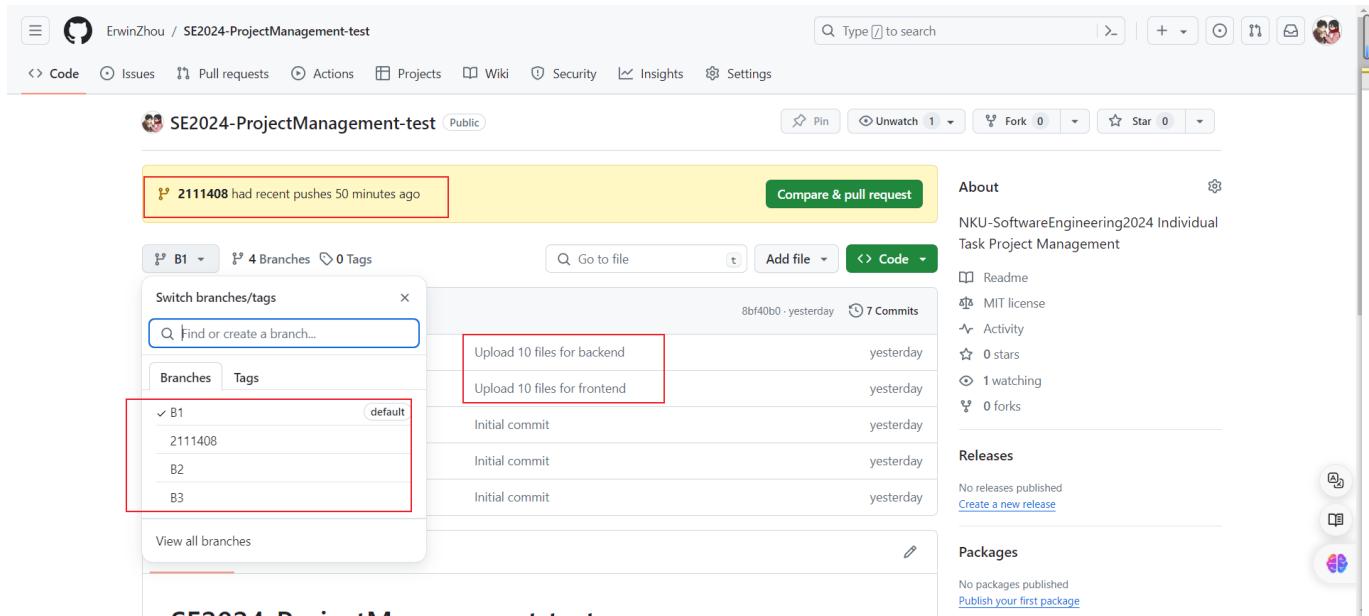


图 3.79: SE2024-ProjectManagement-test 仓库状态

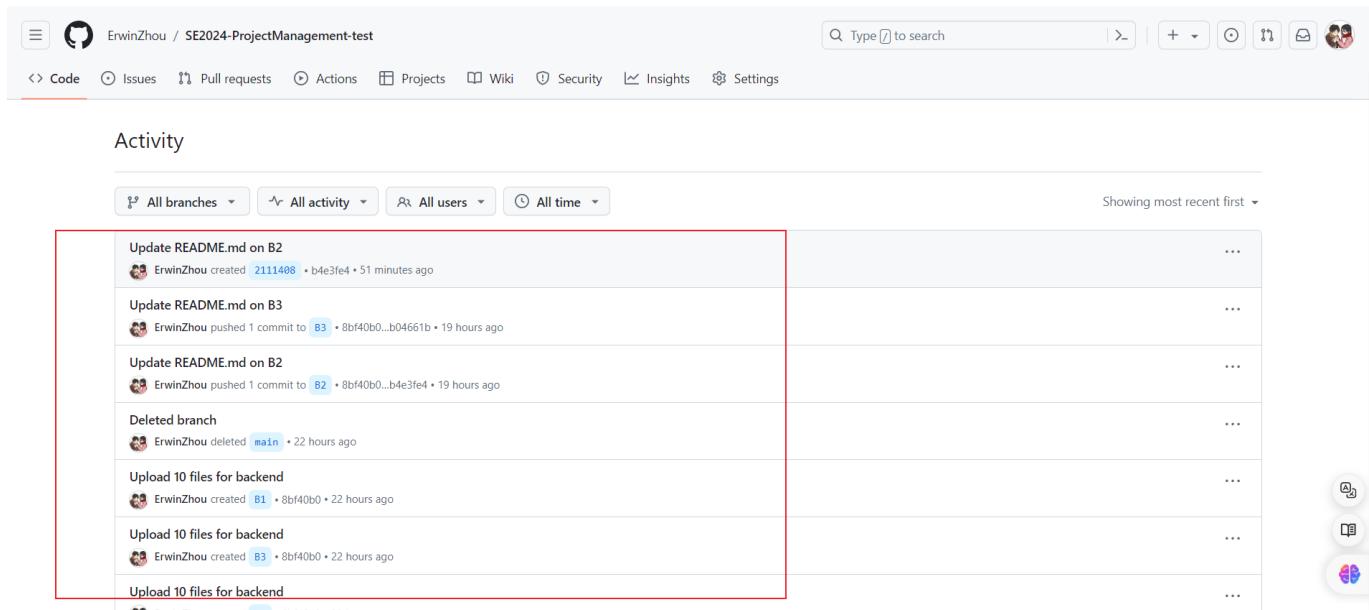


图 3.80: SE2024-ProjectManagement-test 仓库活动

如图3.79和3.80所示，能看到实验场景二涉及的仓库SE2024-ProjectManagement-test的状态和活动，也与我们的操作完全一致，可以得到验证。

到此实验场景三，即远程分支管理的实验内容全部完成。并且三个实验场景均已完成，实验完成度较高，非常成功！

4 小结

4.1 实验思考

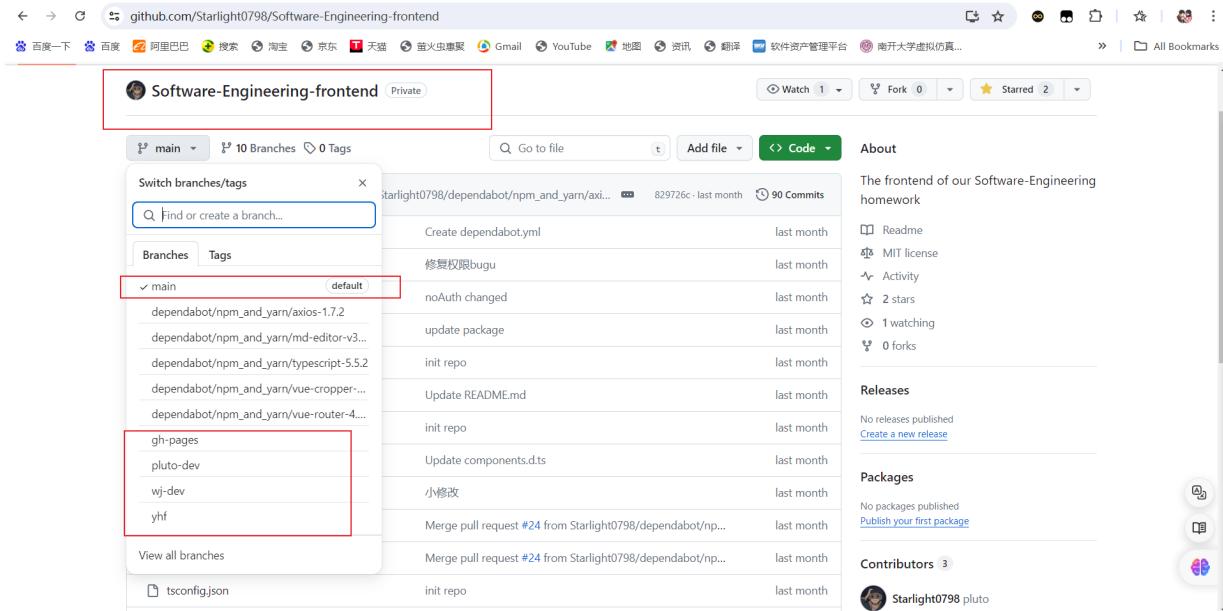


图 4.81: 团队项目前端 Github 仓库

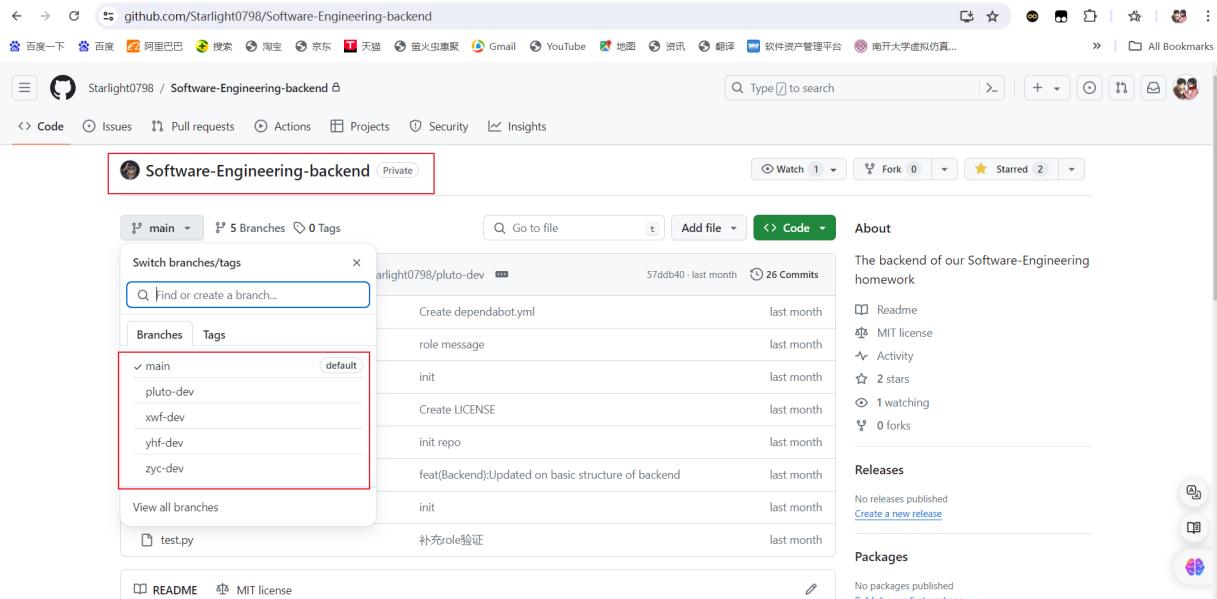


图 4.82: 团队项目后端 Github 仓库

这里我会对实验过程和结果进行思考，反思其中出现的一些情况，主要通过回答一些问题的形式。

4.1.1 git 的优点

思考问题一

Q: 比较之前的开发经验，使用 git 的优点？

问题一回答

A: 在大一刚入学时我也是一个计算机小白，作为一个初高中只会用电脑打游戏的男生来说，最开始写代码是困难的，繁琐的，没有条理的。更别提使用 git 这样的版本控制工具有组织地对代码进行管理。因此和最开始写代码时没有版本控制工具相比，并结合本次实验，Git 有一些明显的优势，可以极大地提升了开发效率和协作体验：

1. **分布式版本控制**: Git 是一种分布式版本控制系统，这意味着每个开发者的本地仓库都是一个完整的代码库副本。本次实验可能看不出来，但是我们大作业的实现时，如图4.81和4.81所示，我们小组每个人都拥有一个独立的分支，比如我的是 zyc-dev。这样的方法允许我们每个人独立地在本地进行更改和并行开发，提高了我们的效率和协作能力。
2. **高效的分支管理**: Git 提供了强大的分支管理功能，允许开发者轻松创建、切换、合并和删除分支。在实验过程中，我创建了多个分支（如 B2、B3 和 C4），并在不同分支上进行了独立的开发和修改。我们团队作业也有着个人单独的开发分支。这种分支管理模式使得团队协作变得更加方便，开发者可以并行工作而不会互相干扰。
3. **方便的合并和冲突解决**: 在合并分支时，Git 提供了强大的合并工具和冲突解决机制。在实验中，我合并 c4 和 B3 分支并处理冲突，通过 Git 的合并功能，有效地解决代码冲突并保证代码的完整性。
4. **完整的历史记录**: Git 保存了所有提交的详细历史记录，包括每次提交的作者、时间和修改内容。在实验中，我多次通过 git log 命令查看提交历史，清晰地了解项目的开发历程和各个分支的变化。这有助于代码审查和问题追踪，提高了项目的可维护性。
5. **强大的社区支持和工具生态**: Git 拥有广泛的社区支持和丰富的工具生态，如 GitHub、GitLab 等平台。这些平台提供了丰富的协作功能，如代码托管、Pull Request、Issue 跟踪等，进一步增强了团队协作效率。在实验中，我使用了 GitHub 进行远程仓库的管理和协作，使得项目的管理更加方便和系统化。
6. **增量式备份和恢复**: Git 的版本控制机制允许开发者随时回到任意一个历史版本，这使得在出现问题时可以方便地进行版本回滚和恢复。我在实验过程中，利用 Git 撤销最后一次提交和恢复修改，体验了其强大的版本恢复能力。

综上所述，Git 的这些优点显著提升了开发效率、协作体验和代码管理的可靠性，使其成为现代软件开发中不可或缺的工具。

4.1.2 git 在个人开发和团队开发中作用的差异

思考问题二

Q: 在个人开发和团队开发中, git 起到的作用有何主要差异?

问题二回答

A: 在个人开发和团队开发中, Git 的作用有显著的差异, 虽然本次实验没有涉及团队协作, 但在大作业的完成中, Git 帮助了我们很多, 在这两次任务中, 我也体会到了 git 起到作用的差异, 具体体现在以下几个方面:

1. **版本控制的独立性和协作性:** 在个人开发中, Git 主要用于本地版本控制, 开发者可以独立管理自己的代码版本。在团队开发中, Git 提供了分布式版本控制, 团队成员可以在各自的分支上工作, 并在需要时进行合并和同步, 提升协作效率。如图4.81和4.81所示, 我们的大作业项目就是通过以不同人命名的独立的 dev 分支实现的并行开发, 最终合并在 main 中。
2. **冲突解决的频率和复杂性:** 在个人开发中, 冲突很少发生, 因为只有一个人对代码进行更改。而在团队开发中, 多个开发者可能会对相同的文件进行更改, 导致冲突频率和复杂性增加, Git 提供了强大的工具来处理这些冲突。如图4.81和4.81所示, 我们的团队作业, 当一个人修复另一个人 bug 时候, 即使改乱了代码, 也可以避免冲突, 在不同分支上独立开发。
3. **代码审查和历史追踪:** 在个人开发中, 代码审查由开发者自己完成, 历史记录主要用于自我参考。在团队开发中, 详细的提交记录和变更历史对于代码审查和问题追踪非常重要, 有助于保持代码质量和项目透明度。
4. **协作工具的使用:** 在个人开发中, 使用 GitHub、GitLab 等平台的频率较低。而在团队开发中, 这些平台提供的 Pull Request、Issue 跟踪等功能极大地增强了团队协作效率, 方便了远程协作和项目管理。如图4.81和4.81所示, 我们的团队项目就分为前后两端各一个仓库, 在 Github 上进行管理。
5. **备份和恢复机制:** 在个人开发中, Git 的备份和恢复功能帮助开发者应对本地代码问题。在团队开发中, 这一机制更为重要, 确保团队成员可以随时恢复到历史版本, 保证项目的稳定性和可靠性。这一点如果只是在 VSCode 本地写代码, 如果为了改代码将过去的一些改动抹除就无法恢复, 除非每次改动都提前把全部代码打包下来, 我相信每个人年轻时都干过...

综上所述, 在个人开发中, Git 主要用于本地版本控制和代码管理; 而在团队开发中, Git 的协作功能、冲突解决、代码审查和历史追踪等功能显著提升了团队开发的效率和质量。

4.1.3 其它版本控制软件对比

思考问题三

Q: 之前是否用过其他的版本控制软件？如果有，同 git 相比有哪些优缺点？如果没有查阅资料对比一下不同版本控制系统的差别。

问题三回答

A: 之前我是没有用过其它的版本控制软件的，不过为了更好地回答本问题，我尝试了 Mercurial，结合亲身体验并查阅资料对比不同版本控制系统的优缺点如下：

1. Git:

- **优点:** Git 是一种分布式版本控制系统，每个开发者的本地仓库都是完整的代码库副本，不依赖中央服务器。Git 的分支管理非常高效，分支和合并操作速度快，支持离线操作，有强大的社区支持和丰富的工具生态，如 GitHub 和 GitLab。
- **缺点:** Git 的学习曲线较陡，命令行操作相对复杂，新手需要时间熟悉其概念和使用方式。

2. Subversion (SVN):

- **优点:** SVN 是集中式版本控制系统，简单易用，适合小型团队。它有直观的目录结构和访问控制，适合需要严格管理的项目。SVN 的命令相对简单，适合初学者。
- **缺点:** 集中式管理依赖中央服务器，如果服务器出现故障，开发工作会受到影响。分支和合并操作相对笨重，速度慢，无法离线操作。

3. Mercurial:

- **优点:** Mercurial 是分布式版本控制系统，类似于 Git，操作简单，性能高效，支持大规模项目。它的命令行工具设计简洁，学习曲线较平缓。**对于我们这种平时的课程作业等代码量还犯不上动用这尊大佛。**
- **缺点:** 社区和生态系统不如 Git 丰富，特别是在企业和开源社区中，工具和插件支持相对较少。

4. Perforce:

- **优点:** Perforce 是企业级版本控制系统，适合管理大型项目和文件。它支持强大的访问控制和大规模协作，有丰富的企业级功能，如代码审查和集成工具。**我认为在我们未来的职场工作中会适合使用。**
- **缺点:** Perforce 的复杂性和成本较高，适合大公司和复杂项目，不适合小型团队和个人开发者。

总的来说，Git 作为分布式版本控制系统，相比于 SVN 的集中式管理，更适合现代化团队协作和开源项目开发，具有高效的分支管理和强大的社区支持。Mercurial 类似于 Git，但在工具和社区支持上不及 Git。Perforce 则适合企业级的复杂项目管理，但其复杂性和成本较高。**因此根据项目需求和团队规模选择合适的版本控制系统非常重要。**

4.1.4 git 的适用情况

思考问题四

Q: 在什么情况下适合使用 git、什么情况下没必要使用 git?

问题四回答

A: 这个问题十分有趣，虽然我已经使用了 Git 长达两年多之久，却没有仔细思考过这个问题，我一直严格要求自己任何代码都用 git 管理，但有些时候十分繁琐和费时。仔细想来，Git 作为一个强大的分布式版本控制系统，在很多场景下都是非常适合使用的，**但也有一些场景可能不需要使用 Git**。

1. 适合使用 Git 的情况：

- **开源项目：** Git 是开源项目的首选版本控制工具，尤其是结合 GitHub、GitLab 等平台，可以方便地进行代码托管、协作开发、代码审查和 issue 跟踪，促进社区贡献和项目发展。**这也是我使用 Github 的最核心原因之一，我认为计算机领域最伟大的思想就是开源，正是因为有了开源，才有了越来越多人能站在巨人的肩膀上做出更伟大的事业和工作。我努力地管理我的 Github，也是为了做好开源，为社区做出一份贡献。**
- **团队协作开发：** Git 的分布式特性和强大的分支管理功能，使其非常适合团队协作开发。
- **复杂项目：** 对于大型和复杂项目，Git 的分支和合并功能可以帮助开发团队高效管理多个开发流程和特性开发，确保代码的稳定性和一致性。
- **需要详细版本历史记录：** Git 的每次提交都会记录详细的历史信息，包括作者、时间和更改内容，有助于代码审查、问题追踪和版本回滚。

2. 没必要使用 Git 的情况：

- **小型和单人项目：** 对于一些简单的大型项目或单人开发项目，如果没有复杂的版本控制需求，使用 Git 可能会显得过于复杂。此时可以选择更简单的版本控制方式，甚至直接手动备份文件。
- **短期和临时性项目：** 如果项目是短期的或临时性的，且没有复杂的协作需求，使用 Git 进行版本控制可能会增加不必要的工作量。
- **静态文件管理：** 对于一些纯粹的文档管理或静态文件管理项目，如果不涉及代码开发和版本控制，使用 Git 的优势不明显，可以选择更适合的文件管理工具。
- **初学者项目：** 对于刚接触版本控制的新手，可能先从简单的工具开始，以便更好地理解版本控制的基本概念，然后再过渡到使用 Git 这样的强大工具。

综上所述，如果是为了开源项目等情况则 Git 十分适用，而对于小型单人项目、短期临时性项目、纯文档管理和初学者项目，则没必要使用 Git，可以选择更简单的版本控制或文件管理方式。

4.1.5 其它思考

其它 git 协作工具

Q: 我是否用过其它 git 协作工具，除了 Github 以外？它们之间的体验有何区别？

Gitee 和恶心的 Gitlab

A: 事实上，很久我都一直用 Github，不过因为一些众所周知的原因因为国产就是好！大一的高级语言程序设计课程上曾让我们使用过 **Gitee**，不过并没有给我留下太深的印象，只是网络连接更加流畅，一般不会出现 Github 那种断网的情况。更适合种花宝宝体质的 Git 协作工具捏。

而最重量级的，莫过于 **Gitlab**。这是上学期的编译原理课程中要求我们管理代码的版本控制工具。如图4.83所示，我们小组成员在其中将每次的代码都上传上去了。整体虽然看着和 Github 类似，但其实有很多恶心地方：比如每次我 push 新代码，都得重新输入用户名和密码，Gitlab 的 SSH 做的极其不稳定，每次一动用 SSH 网页直接错误了。另外看起来很 low，还经常出现网站崩溃的情况。

千言万语化成一句话，Github YYDS！

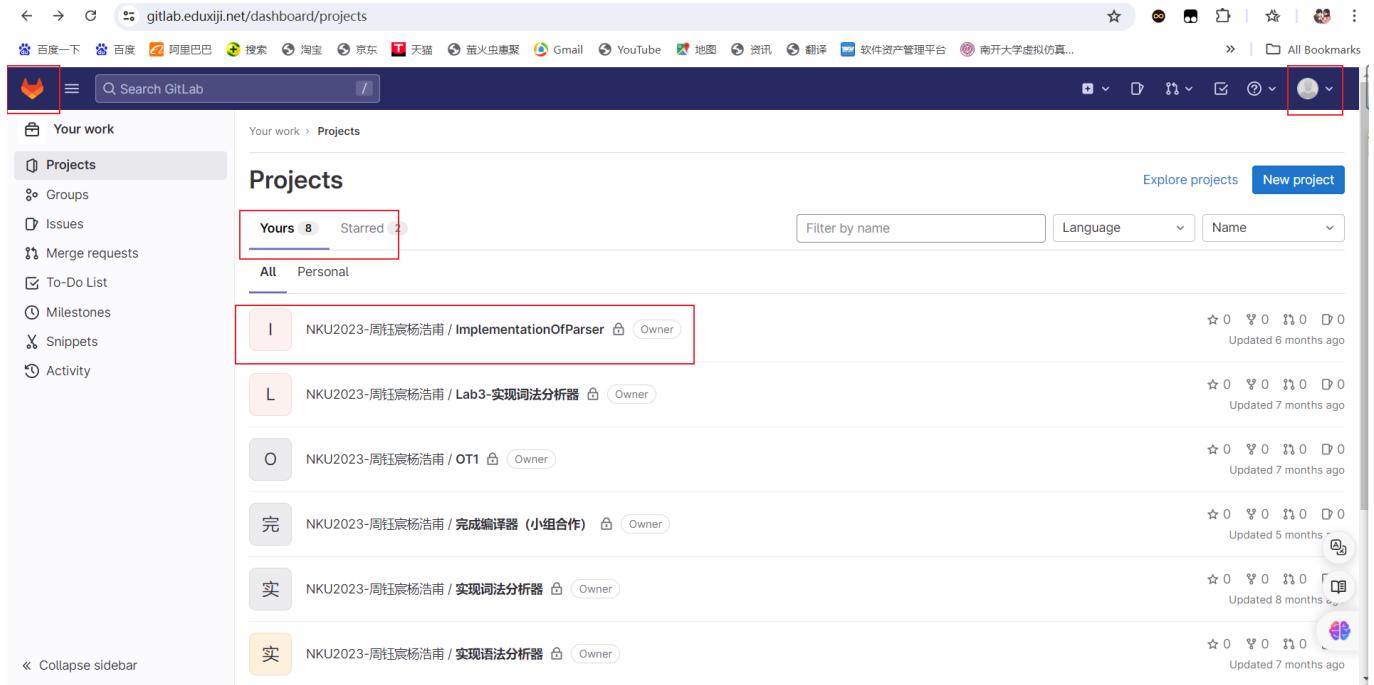


图 4.83: 恶心的 Gitlab

4.2 实验总结

本次实验主题是项目管理，这不仅是软件工程该课程的重点，更是我们未来作为一名计算机人参加职场和工作中，进行软件开发，团队协作必不可少的能力。最开始的实验准备，三个不同的实验场景，从仓库的创建与提交到分支管理，尽管我在之前就有过使用 git 和 Github 管理代码的经历，但这项实践任务做下来还是让我收获了很多。**我在表 1 的实验环境下完成了表 2 中的实验内容**，具体而言：

1. 我通过不同的实验场景，**熟练掌握 git 的基本指令和分支管理指令**。包括最基本的拉取，推送，提交，撤销，到分支管理的合并解决冲突等等。
2. 在这个过程中，我经历了许多困难，遇到了许多实验手册中没有预先料到的问题。在面临**不同的理解时，我也加入了自己**的思考。对于超出预期的情况和现象，**我也通过查阅资料等方式去刨根问底**。这些都让我对**git 支持软件配置管理的核心机理有了更明晰的理解**。
3. 实验全程，我对每一个 Git 操作前都进行了状态查看，确保每一步都在计划中进行，**这让我对项目和分支的管理更加有组织并且高效**。
4. **实验中我避免了图形界面操作，完全利用命令行**，感受着作为程序员，shell 的强大。同时结合 git/github 管理自己的项目源代码，利用 **Github Desktop 以更方便的形式对一些状态进行查看**。这些都让我对 git 相关生态有了完整的认识。
5. 最后，我根据实验过程，自己的多年使用经历，团队协作经历，以及广泛地搜索资料，对 Git 的优势，使用场景等进行了全方面的思考，对实验问题进行了回答。

总的来说，在一次次的报错解决和查找资料之间，**我感受到了 Git 的强大，又因为 Github 的开源精神，我作为一个苦逼的工科人竟然感到了一些自豪**。本次实验我收获颇丰，我会未来继续努力学习 Git 的更多操作，将本次实践中使用 Git 的经验用在未来的软件开发，代码管理的学习工作之中。

致谢

本学期软件工程课程到此结束了，感谢刘老师一直以来的讲解，刘老师一直在课上耐心的为我们传递这一门十分实用的课程知识。其中最令我印象深刻的是最后几节课刘老师实验室的学长将软件开发的思想融合进了一项生物细胞学的研究中，让我感叹其精妙。

我还想要感谢助教韩学姐，学姐一直对我耐心协调和答疑，组织实验课程。让我本学期的课程学习十分轻松愉快。

最后想要感谢我自己，在本学期的多次实验中，我努力地在实验要求中进一步探索和思考。尤其是本次实验，我们遇到了诸多问题，也思考了很多解决方案，提出了很多自己的见解。坚持不懈的努力，对完美的追求，加上求知的探索，让这段经历格外可贵。

感谢老师的耐心指导，收获颇丰！希望老师学姐们未来生活工作一切顺利。

学姐人美心善！毕业快乐呀！