

# Git与GitHub使用说明文档

---

## Git与GitHub使用说明文档

### 引言

- Git和GitHub概述

- 为什么要进行版本控制？

### 准备工作

- 注册你的Github账号

- Git的安装

- Visual Studio Code的安装

  - 为什么要安装vscode

  - 在VSCode中安装Git Graph插件

### CMD指令入门

### Git初始配置

### 相关概念解惑

- 工作区域

- 文件状态

### 开始创建本地仓库

### 添加与提交更改

- 工作区→暂存区 (add)

- 暂存区→本地仓库 (commit)

### 查看差异

### 回退版本

### 分支

- 举个例子

- 分支概念

- 分支策略

### 远程仓库与GitHub

- 创建远程仓库

- 在GitHub上配置SSH

- 将本地仓库与远程仓库关联

- 创建本地main分支

- 推送本地分支到远程仓库

- 更简单的办法

- 使用命令行查看 GitHub 上的远程仓库

### 从Git Bash到Git Graph

- 用GitGraph完成一些基本操作

  - 创建本地仓库

  - 添加与提交更改

  - 查看差异

  - 回退版本

  - 分支操作

## 引言

---

### Git和GitHub概述

Git是一种分布式版本控制系统，用于跟踪、管理和协调项目中的变更。GitHub是一个基于云端的代码托管平台，它提供了许多与Git相关的功能，如代码托管、团队协作、问题跟踪等。本文将引导初学者使用Git和GitHub来管理项目。

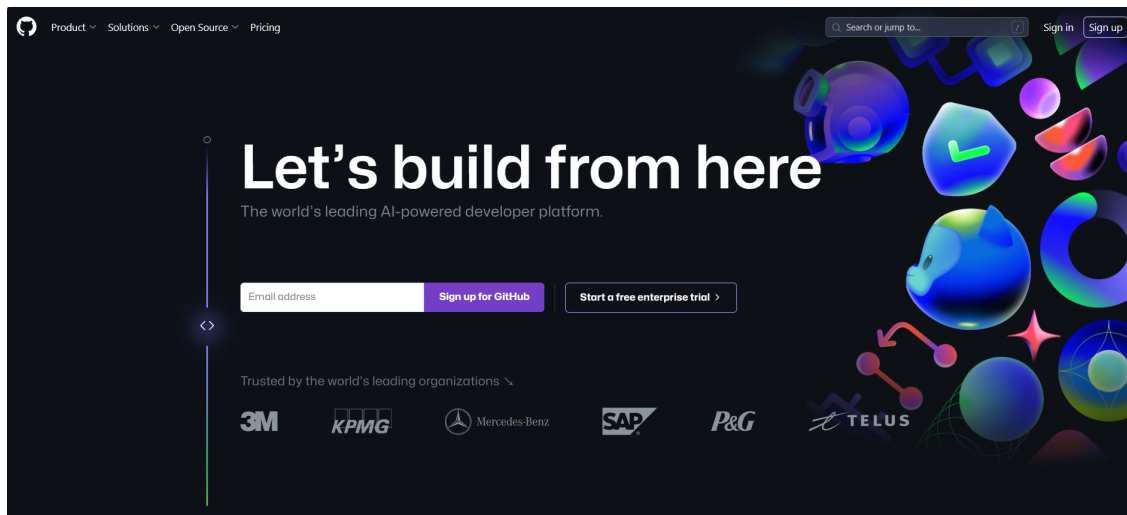
# 为什么要进行版本控制？

版本控制是一种记录文件更改历史的系统，以便将来查看和回溯。对于软件开发来说，版本控制是至关重要的，因为它允许团队成员同时工作在同一个项目上，而不会互相干扰。它还使得跟踪和修复错误、回退到旧版本和理解项目历史变得更加容易。

## 准备工作

### 注册你的Github账号

1. 首先进入github官网 <https://github.com/>



2. 点击右上角的sign up(注册)，sign in是登录。
3. 填写自己的邮箱、密码、用户名等信息，然后用邮箱验证即可完成。

## Git的安装

我们需要在计算机上安装Git。以下是不同操作系统的安装方法：

### • Windows

1. 访问Git官方网站：<https://git-scm.com/downloads>
2. 下载适用于Windows的Git安装程序。
3. 执行安装程序并按照指示完成安装。保留默认设置即可。

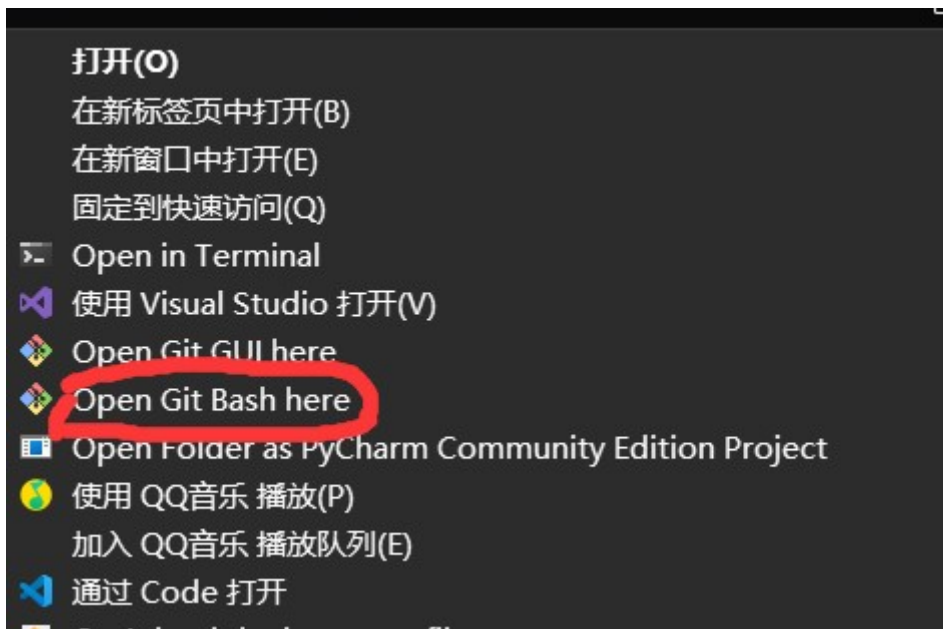
### • macOS

1. 打开终端。
2. 使用Homebrew包管理器安装Git，输入命令：`brew install git`

### • Linux (Ubuntu为例)

1. 打开终端。
2. 输入以下命令安装Git：`sudo apt-get update && sudo apt-get install git`

在windows环境下，具体安装步骤可以参考如下网址，一步一步来：<https://blog.csdn.net/mukes/article/details/115693833>。按照步骤完成后，右键任意一个文件夹，得到菜单中的以下内容，说明你已经安装成功了！！



## Visual Studio Code的安装

Visual Studio Code (VSCode) 是一个流行的代码编辑器，支持Git和许多其他开发工具。

直接搜索Microsoft VSCode官网，从官网下载，确保万无一失。

### 为什么要安装vscode

与传统的IDE相比，VSCode具有以下优点：

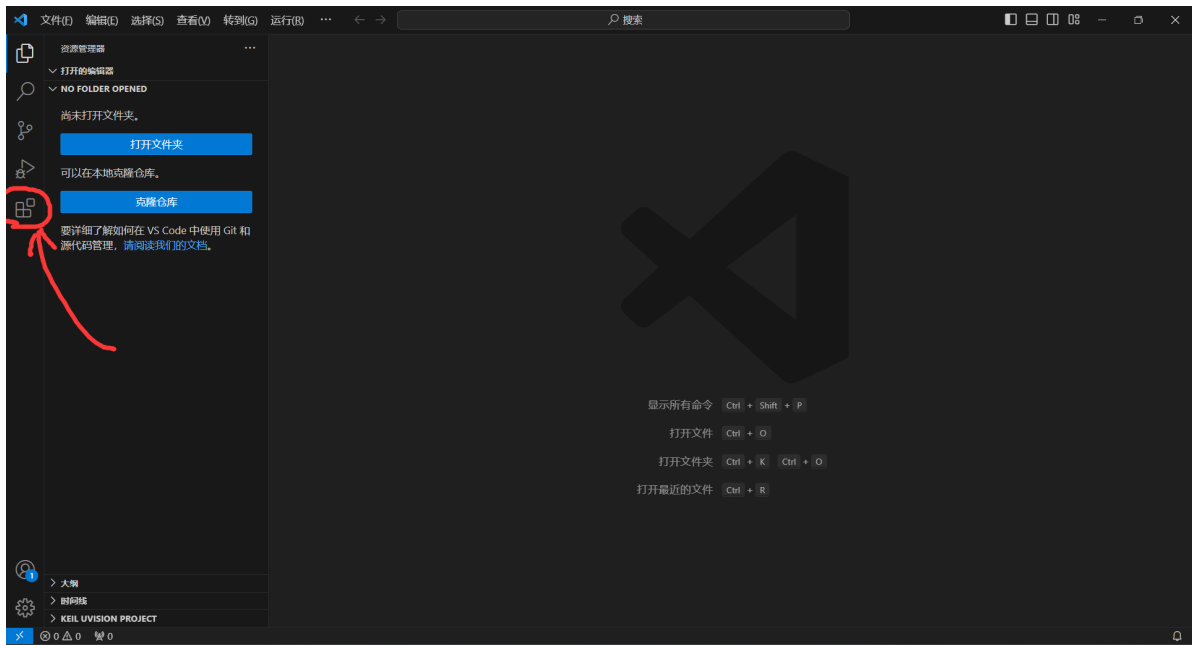
1. **轻量快速**：VSCode 启动迅速，占用资源相对较少，是一款轻量级的编辑器。
2. **丰富的扩展生态系统**：VSCode 支持丰富的扩展，几乎可以满足任何开发需求。你可以根据你的项目和语言选择安装相关的扩展，使得编辑器更符合你的工作流程。
3. **智能代码补全**：VSCode 提供强大的代码补全功能，可以根据你的代码上下文智能地补全代码，提高编码效率。
4. **内置 Git 支持**：VSCode 集成了 Git，你可以直接在编辑器中进行版本控制操作，查看提交历史，解决冲突等。

### 在VSCode中安装Git Graph插件

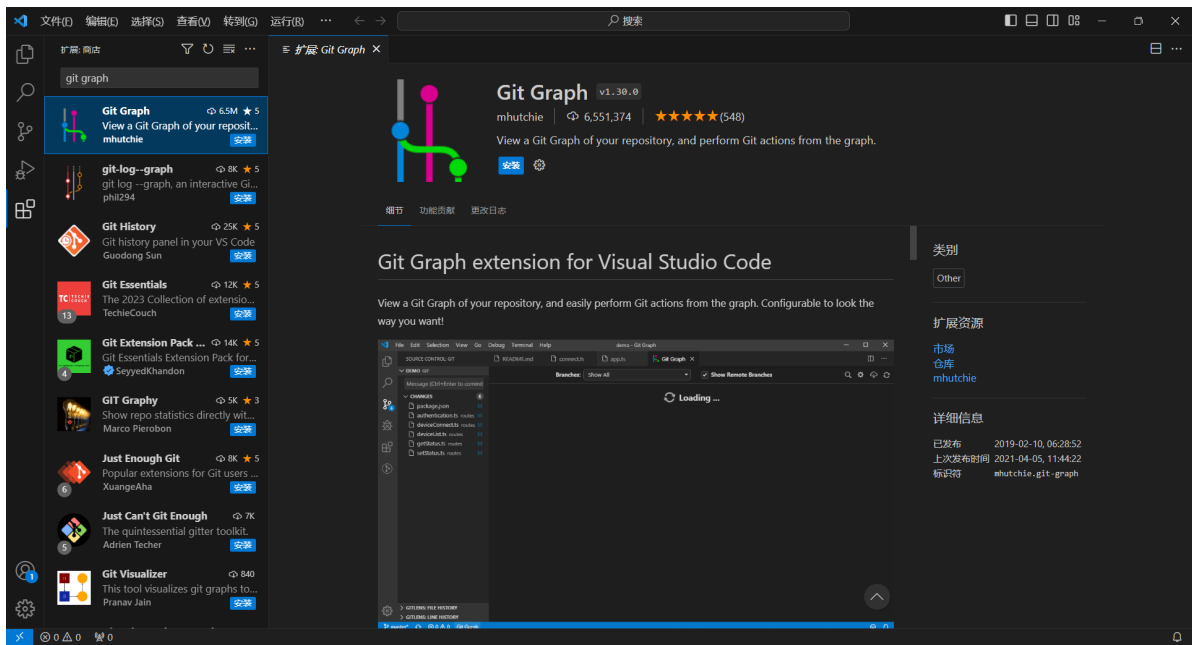
安装 Visual Studio Code (VSCode) 插件非常简单，下面是安装GitGraph的基本步骤，以后安装任何插件都同理。

打开 Visual Studio Code：启动你的 Visual Studio Code 编辑器。

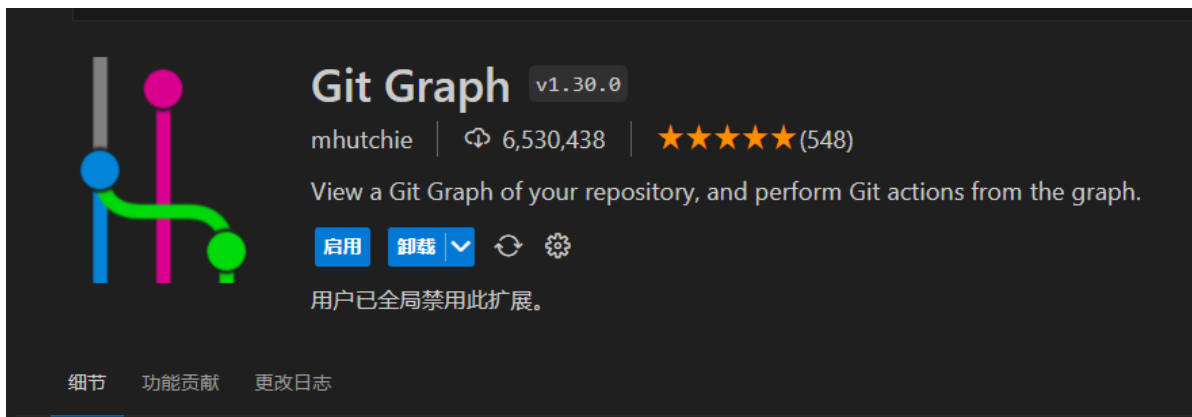
打开扩展视图：在 VSCode 的左侧边栏中，点击活动栏中的 "扩展" 图标（或者按 `Ctrl+Shift+X` 或 `Cmd+Shift+X`）。



搜索插件：在 Extensions 视图的搜索框中输入你想要安装的插件的名称或关键字，这里我们输入 Gitgraph。VSCode 将会显示相关的插件列表。



安装插件：在插件详细信息页面，你会看到一个安装按钮。点击该按钮，VSCode 将开始下载并安装插件。安装完毕后，插件一般会立即启用，若未启用，就在上面同意界面点击启用。



重启 VSCode：在插件安装完成后，VSCode 通常会提示你重新启动以应用插件的更改。重启后，随便打开一个文件夹，如果能在左下角看到如图所示GitGraph按钮，说明安装成功。



## CMD指令入门

在正式学习Git指令之前，我们需要先了解几个windows环境下常用的cmd指令。

### 1. cd (Change Directory):

- 切换到指定目录。

```
cd path to directory
```

### 2. dir (Directory Listing):

- 列出当前目录下的文件和子目录。

```
dir
```

### 3. copy:

- 复制文件或目录。

```
copy source destination
```

### 4. del (Delete):

- 删除文件。

```
del filename
```

### 5. rd (Remove Directory):

- 删除目录。

```
rd /s /q directoryname
```

### 6. ren (Rename):

- 重命名文件或目录。

```
ren oldname newname
```

### 7. move:

- 移动文件或目录到另一个位置。

```
move source destination
```

### 8. echo:

- 在命令行中显示文本。

```
echo Hello, world!
```

### 9. cls (Clear Screen):

- 清除屏幕内容。

```
cls
```

## 10. type:

- 显示文本文件的内容。（注意，type指令还可以用其他指令代替，代表打开文件的不同方式，如vi（用vim打开）、code（用vscode打开））

```
type filename.txt
```

```
vi filename.txt
```

```
code filename.txt
```

## Git初始配置

在开始使用Git之前，需要进行一些初始配置：

- 打开终端或命令提示符。
- 设置用户名：`git config <--global> user.name "Your Name"`
- 设置电子邮件地址：`git config <--global> user.email "youremail@example.com"`
- 查看设置结果：

查看用户名 `git config <--global> user.name`

查看电子邮件 `git config <--global> user.email`

或者 `git config --list` 列出所有配置项，包括用户信息、远程仓库等。

## 相关概念解惑

### 工作区域

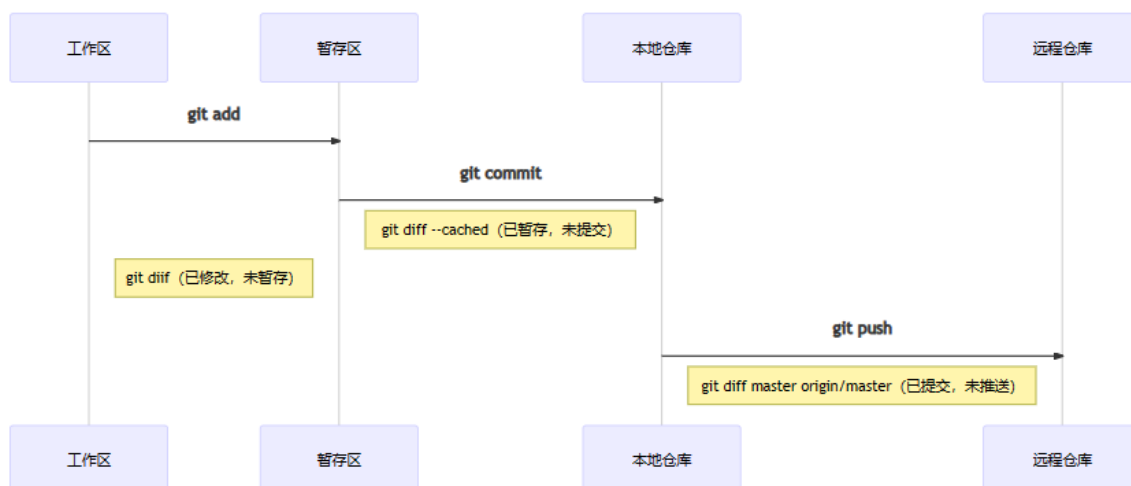
Git有四个工作区域，分别是工作区（Working Directory），暂存区（Staging Area/Index），本地仓库（Local Repository）和远程仓库（Remote Repository）。

**工作区（WorkSpace）：**存放项目文件的地方。

**暂存区（Stage/Index）：**临时存放项目文件的改动情况。

**本地仓库（Local Repository）：**本地库又叫版本库，本地安全存放数据的位置，里面存放着提交到所有版本的数据。

**远程仓库（Remote）：**托管代码的服务器，服务器安全存放数据的位置，里面存放着提交到所有版本的数据。



详细定义，以及各区域的作用（大家自己阅读）：

1. **工作区（Working Directory）** 是你当前正在进行编辑和修改的目录。它包含了项目的实际文件。

2. **暂存区（缓冲区）（Staging Area/Index）** 是一个介于工作目录和本地仓库之间的中间区域。当你对工作目录中的文件进行修改后，可以使用 `git add` 命令将这些修改添加到暂存区。在添加到暂存区后，这些修改就可以被提交到本地仓库。

1. 分离工作目录和本地仓库：通过将修改添加到暂存区，你可以选择性地某些修改纳入提交，而不是一次性将所有修改直接提交到本地仓库。这使得你可以更加细粒度地控制提交的内容，只保存你认为有意义或符合逻辑的修改。
2. 检查修改内容：在将修改添加到暂存区之前，你可以使用 `git diff` 命令查看工作目录中的修改与上次提交的差异。这可以让你仔细检查和审查所做的变更，确保准备好要提交的内容。
3. 执行多个提交：当你在工作目录中进行多个逻辑上独立的修改时，可以将每个修改分别添加到暂存区，并使用不同的提交来记录每个修改的目的和意图。这样，每个提交都会成为项目历史中的一个独立版本，使得追溯和回滚变得更加灵活和可控。
4. 撤销修改：如果你意识到工作目录中的某个修改是错误的或不需要的，你可以使用 `git restore --staged <file>` 命令将修改从暂存区移除，而不会影响工作目录中的实际文件。这让你可以撤销暂存的修改，重新整理提交内容。

3. **本地仓库（Local Repository）** 是保存项目历史记录的地方。它包含了你提交的所有版本以及每个版本的元数据信息。当你使用 `git commit` 命令提交暂存区中的修改时，这些修改就会被永久地保存在本地仓库中。

1. 版本控制：本地仓库保存了项目的完整历史记录，包括每次提交的快照和相关元数据（如作者、提交日期等）。这使得你可以回溯到任何时间点的特定版本，查看文件的状态、修改内容、比较差异或还原到先前的版本。
2. 工作区恢复：如果你在工作目录中删除了或修改了某些文件，本地仓库允许你通过检出（checkout）先前提交的版本来恢复工作目录的状态。这是一个强大的功能，可以帮助你纠正错误或恢复到稳定的工作状态。
3. 分支管理：本地仓库支持分支操作，允许你创建新的分支并在不同的分支上开展并行的工作。分支可以用于同时处理多个任务、开发新功能、修复问题等。本地仓库中的分支可以合并（merge）或重置（reset）到其他分支，以便将各个分支的修改整合到一起。
4. 单机协作：本地仓库使得在单个开发者工作中协作变得容易。你可以使用本地仓库创建和管理自己的分支，进行代码修改和提交，并与其他人共享本地仓库以实现团队协作。

4. **远程仓库（Remote Repository）** 是位于远程服务器上的仓库副本。它允许多个开发者之间协同工作，并可以用于共享代码。远程仓库通常用于集中存储和管理项目的代码，例如GitHub或Bitbucket等平台。你可以使用 `git push` 命令将本地仓库中的修改推送到远程仓库，或使用 `git pull` 命令从远程仓库获取最新的修改。

1. 共享和备份代码：远程仓库提供了一个中央化的代码存储和共享平台，使多个开发者能够访问、共享和备份项目的代码。即使本地仓库出现问题，你仍然可以从远程仓库中恢复项目。
2. 协同合作开发：远程仓库为团队成员提供了便利，他们可以在不同的计算机上共享相同的代码库，并通过拉取（pull）和推送（push）更改来协同工作。这样，团队成员可以同时在自己的分支上进行开发，并将更改合并到主干（或其他分支）上。
3. 版本控制和历史记录：远程仓库保存了项目的完整版本历史和提交记录，便于查看、管理和回溯代码的变化。你可以轻松地查看每个提交、比较不同版本之间的差异以及撤销或还原更改。
4. 部署和发布：远程仓库允许你将代码部署到生产环境或发布到特定的服务器。你可以通过将代码推送到远程仓库来触发自动部署流程或通知其他工具进行构建、测试和部署。



## 文件状态

**Untracked:** 未跟踪，此文件在文件夹中但并没有加入到git库，不参与版本控制，通过git add 状态变为Staged。

**Unmodify:** 文件已经入库，未修改，即版本库中的文件快照内容与文件夹中完全一致，这种类型的文件有两种去处，1. 如果被修改，而变成Modified, 2. 如果使用git rm移除版本库，则成为Untracked文件。

**Modified:** 文件已修改，仅仅时修改，并没有进行其他操作，这个文件也有两个去处，1. 通过git add可进入暂存Staged状态，2. 使用git checkout 则丢弃修改内容，返回Unmodify状态，这个git checkout 即从库中去除文件，覆盖当前修改。

**Staged:** 暂存状态，执行git commit 则将修改同步到库中，这时库中的文件和本地文件又变为一致，文件为Unmodify状态，执行git reset HEAD filename取消暂存，文件状态为Modified。

## 开始创建本地仓库

接下来，让我们创建一个新的Git仓库来管理项目的版本控制：

1. 打开终端或命令提示符。
2. 进入您想要创建仓库的文件夹：`cd path`
3. 输入以下命令初始化一个新的Git仓库：`git init`

```
PS E:\NIEZS\git文档\git培训相关资料\git_test> git init
Initialized empty Git repository in E:/NIEZS/git文档/git培训相关资料/git_test/.git/
PS E:\NIEZS\git文档\git培训相关资料\git_test> |
```

此时我们就创建好了一个 `.git` 文件夹。

`.git` 文件夹是Git版本控制系统用来存储仓库元数据和对象数据库的地方。该文件夹包含了整个仓库的配置信息、对象数据库（存储文件和目录的快照）、引用（指向提交的指针）、配置文件等。因此，可以说 `.git` 文件夹是本地Git仓库的核心。

**注意** `.git` 文件夹在文件资源管理器中是看不见的。你可以使用命令行中的 `ls -a`（在Unix/Linux系统下）或者 `dir /a`（在Windows系统下）来查看包含隐藏文件的目录。

## 添加与提交更改

在Git中，我们需要将更改先将更改添加到缓冲区（暂存区），然后提交更改到仓库。

### 工作区→暂存区（add）

1. 在所需工作区文件上进行更改。
2. 打开终端或命令提示符。
3. `git status` 命令将显示工作区和暂存区的状态信息，包括已修改但尚未暂存的文件、已暂存但尚未提交的变更等。在暂存区的部分，你将看到列出的文件，表示这些文件已被添加到下一次提交中。如下图，此时红色部分说明工作区内有两个文件，它们并未添加（add）到暂存区中，处于未跟踪状态（untracked）。



```

PS E:\NIEZS\git文档\git培训相关资料\git_test> git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        1.txt
        2.txt
        3.txt
        4.txt

nothing added to commit but untracked files present (use "git add" to track)

```

4. `git add <filename>` 此命令可将更改添加到暂存区：（可以使用通配符 `*` 来添加所有更改的文件）。如下图，此时绿色部分说明成功添加了“1.txt, 2.txt”到暂存区。同时“3.txt, 4.txt”尚处于未跟踪状态（untracked）。暂存区的“1.txt, 2.txt”尚未提交（commit）。

```

PS E:\NIEZS\git文档\git培训相关资料\git_test> git add 1.txt
PS E:\NIEZS\git文档\git培训相关资料\git_test> git add 2.txt
PS E:\NIEZS\git文档\git培训相关资料\git_test> git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   1.txt
        new file:   2.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        3.txt
        4.txt

```

## 暂存区→本地仓库（commit）

1. 继续使用终端或命令提示符。
2. 输入以下命令提交更改到仓库：`git commit -m "Commit message"`（请提供有意义的提交消息）。“commit message”表示此次提交的信息，最好是简单描述一下提交时，我们对项目修改了哪些内容。

```

PS E:\NIEZS\git文档\git培训相关资料\git_test> git commit -m "first"
[master (root-commit) ceb3cdf] first
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 1.txt
create mode 100644 2.txt
PS E:\NIEZS\git文档\git培训相关资料\git_test> |

```

3. 输入以下命令查看提交记录：`git log`

```

PS E:\NIEZS\git文档\git培训相关资料\git_test> git log
commit ceb3cdf18a41df95690e92cfd8eab8753eadb47c (HEAD -> master)
Author: Buzzzz <buzz210625@gmail.com>
Date:   Sat Dec 2 17:42:46 2023 +0800

    first
PS E:\NIEZS\git文档\git培训相关资料\git_test> |

```

## 查看差异

在 Git 中，有几个常用的命令用于查看文件之间的差异（diff）。以下是其中几个常见的 Git 查看差异的指令：

1. **git diff:**

- 用于比较工作目录中的文件和暂存区域（即将要提交的更改）之间的差异。

2. **git diff :**

- 查看指定文件的差异，包括工作目录和暂存区域的差异。

3. **git diff --cached (或 git diff --staged):**

- 查看已暂存（即将提交）和最后一次提交之间的差异。

4. **git diff :**

- 比较两个提交之间的差异。可以使用提交的 SHA-1 值、分支名或标签名。

5. **git log -p:**

- 查看提交历史，并显示每个提交的详细差异。

## 回退版本

在 Git 中，有几种方法可以用于回退版本，以下是一些常见的 Git 回退版本的指令：

1. **git reset:**

- `git reset` 可以用于撤销暂存区的更改或回退到之前的提交。有三种模式可以使用：`--soft`、`--mixed`（默认）和 `--hard`。



```
1 # 回退到前一个提交，保留更改在工作目录，但不在暂存区域中
2 git reset --soft HEAD~1
3
4 # 回退到前一个提交，取消暂存区域中的更改
5 git reset HEAD~1
6
7 # 回退到前一个提交，丢弃工作目录和暂存区域中的更改
8 git reset --hard HEAD #当前版本
9 git reset --hard HEAD^ #回退到上一个版本
10 git reset --hard HEAD^^ #回退到上上一个版本
11 git reset --hard HEAD~3 #回退到往上3个版本
12 git reset --hard HEAD~10 #回退到往上10个版本
```

HEAD~1 可以用提交编号代替，提交编号从 `git log` 找到，然后复制下来，当作参数。

```
git reset [--xxxx] <commit>
```

```
PS E:\NIEZS\git文档\git培训相关资料\git_test> git log
commit 92fd49aeefb56953e10f158f5eefa35e90a78b8 (HEAD -> master)
Author: Buzzz <buzzz10025@gmail.com>
Date: Sat Dec 2 18:06:24 2023 +0800

    commit2

commit ceb3cdf18a41df95690e92cfd8eab8753eadb47c
Author: Buzzz <buzzz10025@gmail.com>
Date: Sat Dec 2 17:42:46 2023 +0800
```

## 2. git revert:

- `git revert` 用于撤销指定提交引入的更改，并创建一个新的提交来应用这个撤销。

```
1 # 撤销前一个提交
2 git revert HEAD
3
4 # 撤销指定提交
5 git revert abc123
```

## 3. git checkout:

- `git checkout` 用于切换分支或还原文件到指定提交的状态。

```
1 # 切换到之前的提交，工作目录和暂存区域都会变为指定提交的状态
2 git checkout abc123
3
4 # 切换到分支并将工作目录和暂存区域还原为该分支的状态
5 git checkout branchname
```

这些指令允许你在项目中执行不同类型的版本回退操作，具体取决于你的需求和希望达到的效果。在执行这些操作之前，请确保理解其对项目历史和工作目录的影响。如果你要回退到较旧的提交，最好先备份或确保不会丢失重要的更改。

# 分支

在 Git 中，分支是用来表示项目不同版本、不同功能或不同任务的一种非常重要的概念。简要说，Git 分支可以理解项目代码的一个独立副本，它允许在不影响主要代码线的情况下进行修改和实验。

## 举个例子

假设有一个软件开发团队正在开发一个电子商务网站。团队决定使用 Git 分支来管理不同的功能和任务。以下是一个例子，鲜明地体现了分支在团队协作中的好处：

### 1. 主分支 (main) :

- 主分支是项目的稳定版本。在主分支上，团队只包含了已经通过测试和验证的代码，比如店铺的主界面。

### 2. 特性分支 (feature branches) :

- 假设团队需要添加一个新的功能或者增加一个新的商品，比如实现用户评论功能。每个开发人员可以为此创建一个特性分支，以便独立地进行开发，而不影响主分支的稳定性。

开发人员可以在这个分支上进行修改，测试新功能，并保持主分支的稳定。一旦新功能完成并通过测试，可以将这个特性分支合并回主分支。

### 3. 发布分支 (release branches) :

- 当团队准备发布新的版本时，可以创建一个发布分支。在这个分支上进行一些准备工作，如更新版本号、准备发布文档等。

在发布分支上的修改不会影响正在进行的其他功能开发。一旦准备就绪，可以将发布分支合并回主分支，并标记为发布版本。

通过使用分支，团队可以并行地进行多个任务，而不会相互干扰。每个分支都有其特定的目的和职责，从而提高了开发的灵活性和效率。分支还提供了对不同功能和任务的清晰可见性，有助于更好地组织和协调团队的工作。

## 分支概念

以下是关于 Git 分支的一些关键概念：

### 1. 主分支 (master/main) :

- 通常新项目的主要开发线就是主分支（通常叫做 `master` 或 `main`）。它是项目的默认分支，包含了最新的稳定代码。

### 2. 新建分支：

- 你可以通过创建新分支来在项目中进行新的开发工作。新分支的内容最初和创建它时的源分支相同。

```
1 # 创建新分支
2 git branch new-feature
```

### 3. 切换分支：

- 使用 `git checkout` 或 `git switch` 命令可以在不同分支之间切换。

```
1 # 切换到新创建的分支
2 git checkout new-feature
```

### 4. 合并分支：

- 将一个分支的更改合并到另一个分支。通常，你将新功能分支的更改合并到主分支。

```
1 # 在主分支上合并新功能分支
2 git checkout master
3 git merge new-feature
```

### 5. 查看分支：

- 使用 `git branch` 命令可以查看当前分支以及所有分支的列表。

```
1 # 查看所有分支
2 git branch
```

### 6. 删除分支：

- 删除已经完成工作的分支。

```
1 # 删除新功能分支
2 git branch -d new-feature
```

## 7. 远程分支：

- 在远程仓库上也可以有分支。你可以通过 `git push` 和 `git fetch` 命令与远程分支进行交互。

```
1 # 推送本地分支到远程仓库
2 git push origin new-feature
3
4 # 拉取远程分支到本地
5 git fetch origin new-feature
```

## 分支策略

分支策略是软件开发团队在使用版本控制系统（如 Git）时制定的一套规则和约定，用于管理和组织代码库中的分支。分支策略定义了在不同阶段、不同目的下应该如何创建、合并和维护分支。这有助于团队更有序、高效地协同开发，同时确保代码的稳定性和可维护性。

以下是一种常见的分支策略：

### 1. 主分支 (Main/Branch)：

- 主分支通常是代码库的主要分支，包含了最新的稳定版本。在 Git 中，主分支通常被称为 `main` 或 `master`。

### 2. 开发分支 (Develop/Branch)：

- 开发分支是用于整合所有功能的主要开发分支。在开发分支上进行的修改应该是相对较稳定的，每个功能或任务的开发通常都从这个分支开始。

### 3. 特性分支 (Feature Branches)：

- 特性分支用于开发单一功能或任务。每次添加新功能时，通常会创建一个新的特性分支。一旦开发完成，特性分支可以合并回开发分支。

### 4. 发布分支 (Release Branches)：

- 发布分支用于准备发布新的版本。在发布分支上进行一些准备工作，如更新版本号、解决最终性问题。一旦准备就绪，发布分支可以合并回主分支，并在合并后打上一个标签表示版本号。

### 5. 热修复分支 (Hotfix Branches)：

- 热修复分支用于紧急修复在发布后发现的生产问题。这个分支从主分支中创建，完成修复后合并回主分支和开发分支。

## 远程仓库与GitHub

将本地仓库与远程仓库关联，并使用GitHub托管您的代码。

## 创建远程仓库

1. 访问GitHub网站: <https://github.com> 并登录账户 (如果没有账户, 请先注册)。
2. 在页面右上角点击"New"按钮创建一个新的仓库。
3. 输入仓库名称、描述等信息, 并选择是否将该仓库设为私有。
4. 点击"Create repository"按钮创建远程仓库。

## 在GitHub上配置SSH

千万别忘了使用Github托管代码之前, 要先进行SSH的配置。SSH (Secure Shell) 是一种网络协议, 它通过加密的方式在网络上安全地传输数据, 常用于远程登录和文件传输。SSH可以提供一种安全的身份验证方式, 用于访问和操作 GitHub 仓库。配置 SSH后, 我们就可以使用公钥加密来进行身份验证, 这提供了以下几个重要的优势:

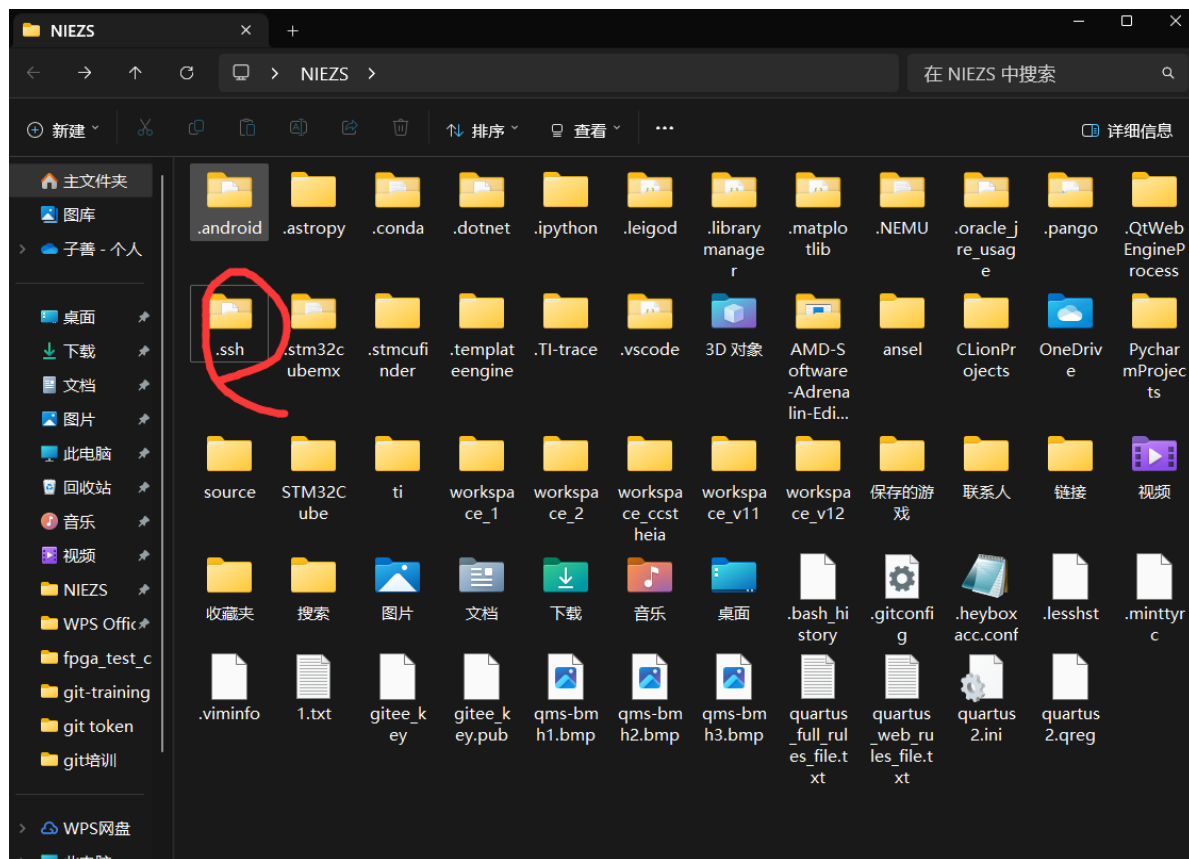
1. **安全性:** 使用 SSH 可以加密通信, 防止中间人攻击和窃听。这保护了用户与 GitHub 之间的数据传输的安全性。
2. **身份验证:** SSH 使用密钥对进行身份验证, 而不是像用户名和密码那样依赖于口令。这使得身份验证更为安全, 因为密钥对通常比密码更难以破解。
3. **便利性:** 一旦设置了 SSH, 就无需在每次与 GitHub 通信时都输入用户名和密码。这提供了更方便的工作流程, 尤其是在频繁进行 Git 操作时。

跟随下面步骤, 完成你的配置:

步骤一: 先用win+R打开cmd窗口, 输入 `cd .ssh`, 打开windows系统自带的.ssh文件。

```
C:\Users\NIEZS>cd .ssh  
C:\Users\NIEZS\.ssh>|
```

如果提示 `.ssh` 路径不存在, 在你的用户文件夹下新建一个名叫.ssh的文件夹。



步骤二：输入 `ssh-keygen -t rsa -b 4096`，此时它会提示你输入你准备创建的密钥文件的名称，因为我之前创建过叫做 `key` 的文件，所以这次我设置的是 `yourkey`，**注意！如果你下次创建的名称与之前创建过的重复，那么新文件会覆盖旧文件。**然后，系统还会让你输入两次 `passphrase`，不用管它，直接回车两次就好了。

```
C:\Users\NIEZS\.ssh>ssh-keygen -t rsa -b 4096
Generating public/private rsa key pair.
Enter file in which to save the key (C:\Users\NIEZS\.ssh/id_rsa): yourkey
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in yourkey
Your public key has been saved in yourkey.pub
The key fingerprint is:
SHA256:QpvLWp1HYVocBQNTww3z1Kt5F4vyt9z8l5TR4+4kzyY niezs@LAPTOP-NQ4JUF4M
The key's randomart image is:
+---[RSA 4096]-----+
|
| o+O=o. |
| o.B. . |
| . = . ..|
| . o + . .+.|
| + S . oo *|
| . + o .o..=|
| + o . oooo.|
| o . E**o|
| . =*B|
+----[SHA256]-----+
```

步骤三：不要退出 `.ssh` 文件夹，输入 `dir`，如果能找到你刚才创建的一对公钥（`yourkey.pub`）和私钥（`yourkey`）。公钥可以用于GitHub的配置中，私钥不要动。

```
C:\Users\NIEZS\.ssh>dir
Volume in drive C is Windows-C
Volume Serial Number is 3225-6168

Directory of C:\Users\NIEZS\.ssh

2023/12/03  15:57    <DIR>          .
2023/12/03  14:16    <DIR>          ..
2023/11/30  17:53          99 config
2023/11/30  17:53    <DIR>          fpga_test_clock
2023/12/01  22:24        399 id_ed25519
2023/12/01  22:24         96 id_ed25519.pub
2023/11/30  17:32       3,389 id_rsa
2023/11/30  17:32       747 id_rsa.pub
2023/11/30  17:33       3,389 key
2023/11/30  17:33       747 key.pub
2023/11/30  17:54       828 known_hosts
2023/11/30  17:17         92 known_hosts.old
2023/12/03  15:57       3,389 yourkey
2023/12/03  15:57       748 yourkey.pub
               11 File(s)        13,925 bytes
               3 Dir(s)  142,836,617,216 bytes free
```

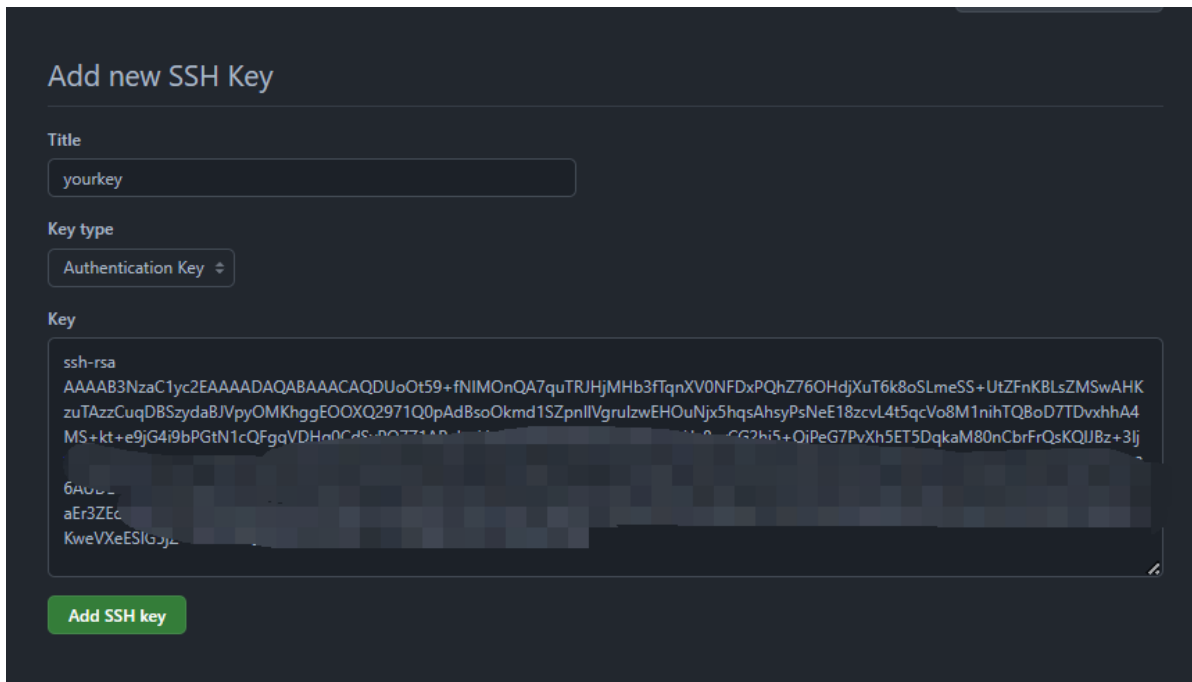
步骤四：复制公钥内容，输入 `code yourkey.pub` 用 `vscode` 打开公钥，然后把内容复制下来。

```
C:\Users\NIEZS\.ssh>code yourkey.pub|
```



```
yourkey.pub X
C: > Users > NIEZS > .ssh > yourkey.pub
1 ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQAUoT59+fNIMOnQA7quTRJHjMHb3fTqnXV0NFDxPQhZ76OHdjXuT6k8oSLmeSS+UtZFnKBLsZMSwAHKzuTAzzCuqD
2
```

步骤五：打开GitHub，进入个人首页，点击setting->SSH and GPG keys->New SSH key，把复制的内容放入key文本框内，安全起见，这里打了马赛克。



步骤六：修改config文件，指定GitHub每次传输数据时都使用此密钥

输入 `code config`，打开config文件，然后把里面的内容改成如下，`yourkey` 就是你创建密钥的名字。

```
1 # github
2 User git
3 HostName github.com
4 PreferredAuthentications publickey
5 IdentityFile ~/.ssh/yourkey
```

步骤七：用 `git clone git@github.com:Buzz2Z/aboutGit.git` 验证是否成功。

## 将本地仓库与远程仓库关联

在本地仓库所在的文件夹中打开命令行或终端，执行以下操作：

```
1 # 进入本地仓库目录
2 cd /path/to/your/local/repository
3
4 # 关联本地仓库与远程仓库
5 git remote add origin git@github.com:<your-username>/<your-repository>.git
```

确保将 `your-username` 和 `your-repository` 替换为你的 GitHub 用户名和仓库名称。

## 创建本地main分支

```
1 | git branch <branch-name>
```

<branch-name> 填 main

## 推送本地分支到远程仓库

```
1 | # 将本地分支推送到远程仓库
2 | git push -u origin main
```

这将把本地的 `main` 分支推送到远程仓库。如果你在本地使用了其他分支，可以将它们推送到远程仓库，例如：

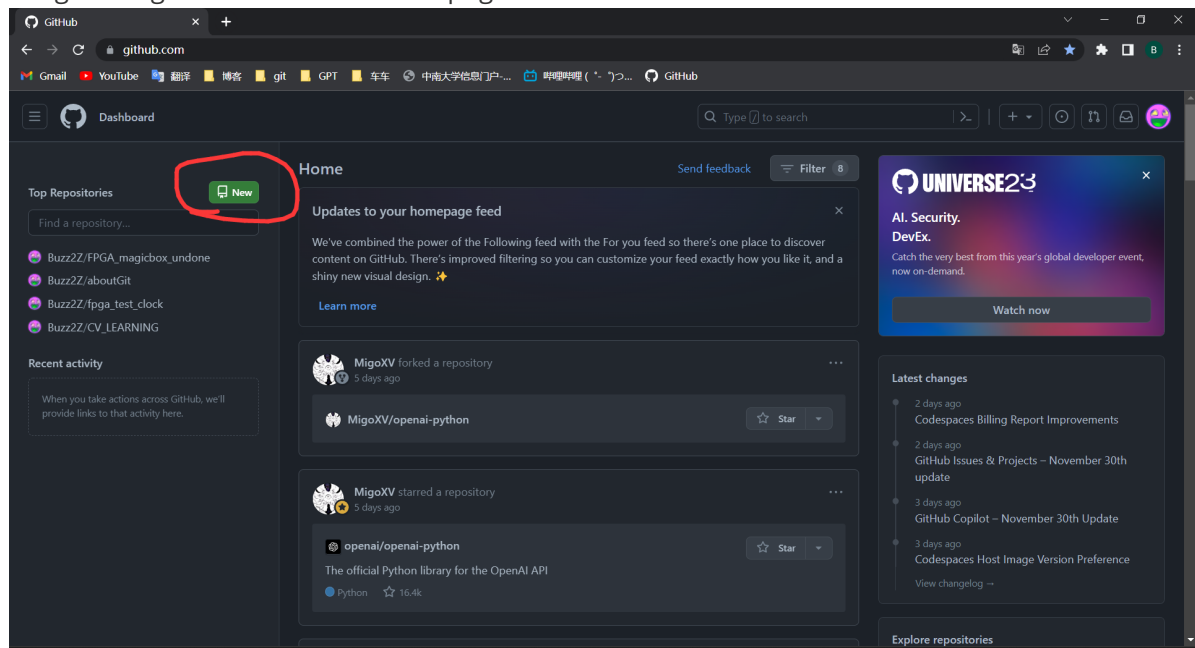
```
1 | git push -u origin your-local-branch-name
```

## 更简单的办法

这一部分有更简单的办法，确保你不会出错，那就是直接根据在GitHub中创建新的远程仓库时，Github给你的提示来操作。

**步骤一：打开GitHub首页，点击NEW，新建一个repository**

![image-20231203150655421](C:\Users\NIEZS\AppData\Roaming\Typora\typora-user-images\image-20231203150655421.png)



**步骤二：填写repository name, description, 后面的选项自己选，但是如果点了添加readme，那么后面一步的页面不会显示，不过没有关系，在下一步直接输入命令，效果是一样的。点击create repository.**


## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

---

*Required fields are marked with an asterisk (\*).*

Owner \*

 Buzz2Z

 /

Repository name \*

test


✔ test is available.

Great repository names are short and memorable. Need inspiration? How about [bookish-barnacle](#) ?


Description (optional)

your description

---

☒  **Public**

Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**

You choose who can see and commit to this repository.

---

Initialize this repository with:

☐ **Add a README file**

This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: None


Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: None

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

---

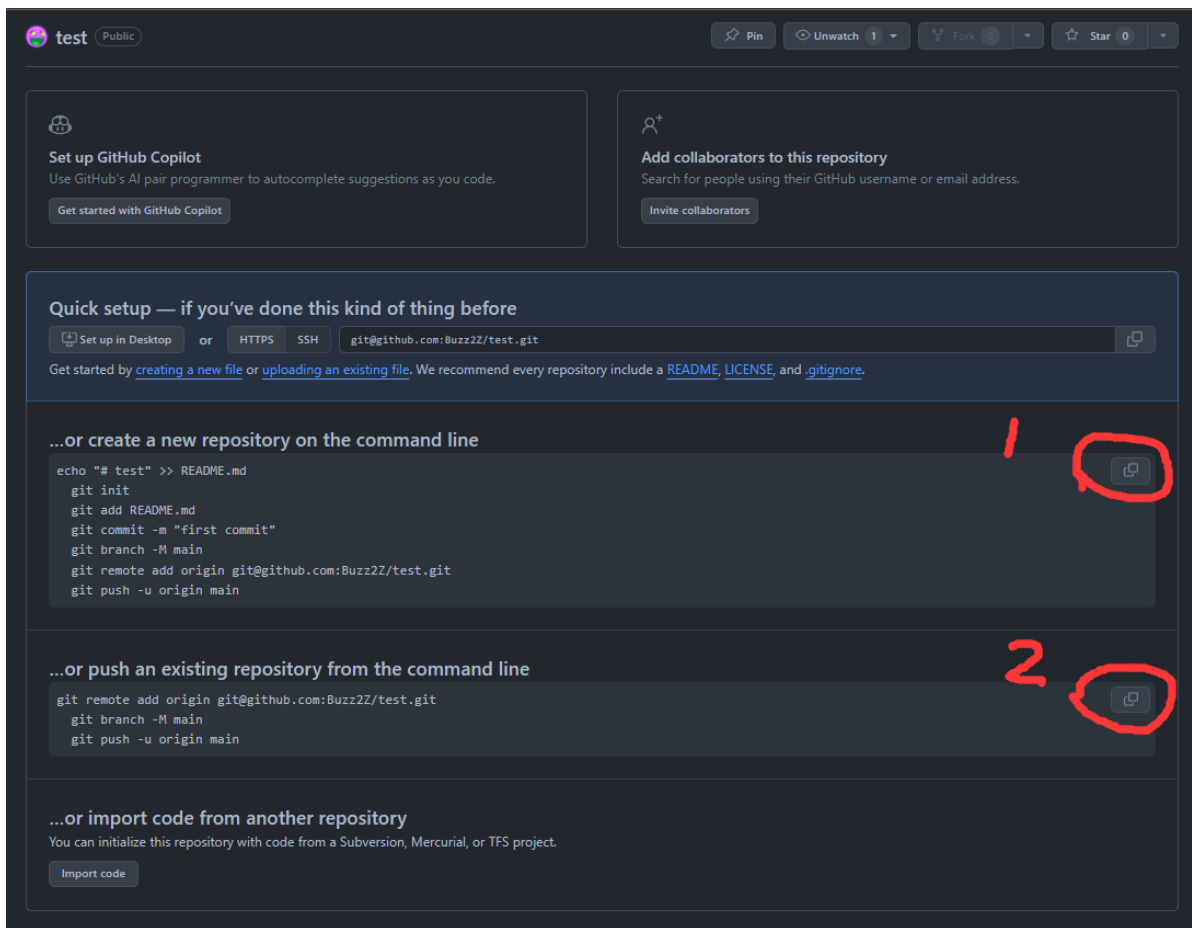
 You are creating a public repository in your personal account.

---

Create repository

**步骤三：如果前一步没勾选readme，就会出现下面的界面，这个界面引导你创建本地仓库，并且关联本地仓库到远程仓库。分为三种不同的方式。**

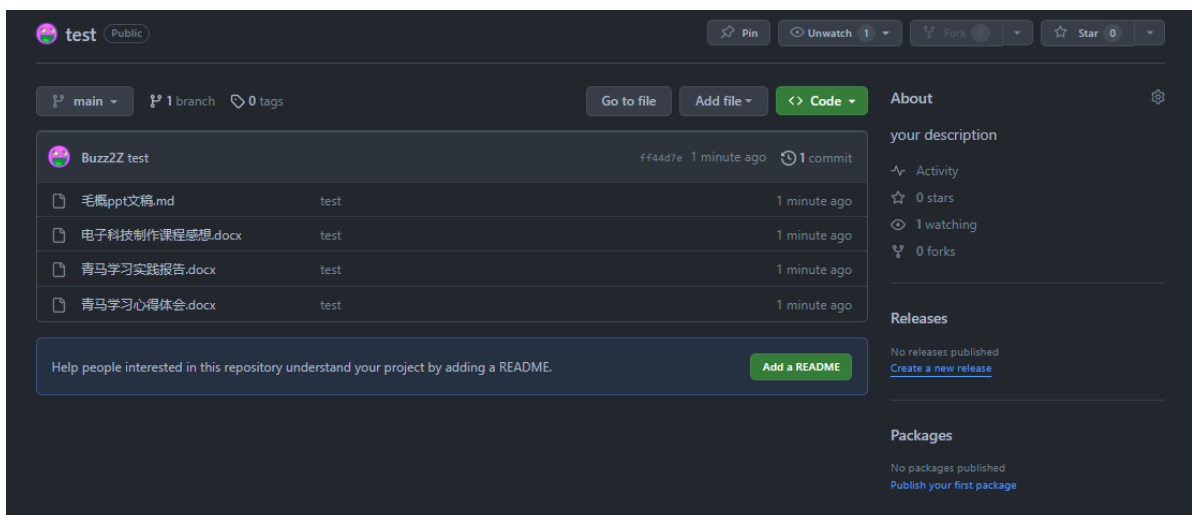
- 方式一：如果你还没有用git init创建自己的本地仓库，那么就用这种方式，先用git bash打开工作区文件，然后直接复制红圈一的指令到命令行执行。
- 方式二：如果你已经用git init创建了自己的本地仓库，那么就用第二种方式，先用git bash打开工作区文件，然后直接复制红圈二的指令到命令行执行。
- 方式三：用其他方式导入，这个方法很少用，可以自己研究。



我直接随便找了个文件夹关联：

```
PS E:\NIEZS\可套用文章> git remote add origin git@github.com:Buzz2Z/test.git
>> git branch -M main
>> git push -u origin main
```

刷新GitHub界面，下面是成功执行的结果：



## 使用命令行查看 GitHub 上的远程仓库

刷新 GitHub 页面，你应该能够看到已经成功推送的仓库。

一些关于远程仓库状态的指令：

1. 查看远程仓库列表：

```
1 | git remote
```

这会显示你当前 Git 仓库配置的所有远程仓库的简写名称。

#### 2. 查看远程仓库详细信息：

```
1 | git remote -v
```

这会显示远程仓库的详细信息，包括 URL。

#### 3. 重命名远程仓库：

```
1 | git remote rename <旧远程仓库名> <新远程仓库名>
```

这会将远程仓库的简写名称从旧的名字改为新的名字。

#### 4. 删除远程仓库：

```
1 | git remote remove <远程仓库名>
```

这会移除你的 Git 仓库配置中的某个远程仓库。

#### 5. 查看远程仓库信息：

```
1 | git remote show <远程仓库名>
```

这会显示关于指定远程仓库的更详细的信息，包括跟踪的分支、本地和远程分支之间的关系等。

## 从Git Bash到Git Graph

传统的 Git，主要使用命令行来进行配置，得到的项目主干、分支、提交结果、差异、时间非常不直观。

- **信息可视化较差：** Git 命令行输出的信息是文本形式的，不够直观。查看分支结构、提交历史等信息需要通过命令行输出，对于初学者或者需要频繁查看项目结构的用户来说，可能不够直观。
- **复杂的操作：** 有些 Git 操作涉及到较长的命令，特别是需要记忆一些参数和选项。对于初学者而言，记忆这些命令并正确使用它们可能需要一些时间。
- **代码审查困难：** 在命令行中进行代码审查可能相对繁琐，需要查看具体的文件路径、差异等。对于团队协作中的代码审查流程，可能希望能够更轻松地进行可视化的审查。
- **实时图形展示：** Git 命令行不提供实时的图形展示，不能动态地展示分支合并、提交历史的变化。有时候，这种动态的展示可以更好地帮助用户理解项目的演进。

Git Graph 插件的优势在于可视化展示。

使得项目的结构和提交历史更加清晰易懂。这对于初学者来说，尤其是对 Git 概念和操作不够熟悉的人来说，更容易理解和使用。同时，通过图形界面的交互，可以更直观地执行一些 Git 操作，减少了记忆和输入命令的负担。

👤 欢迎

🔍 .gitignore

🔍 COMMIT\_EDITMSG

📁 1 概论.md

🔗 Git Graph

✕

📅 ...

Branches: 

Show All

☒ Show Remote Branches

🔍

📁

⚙️

🔄

🔄

Graph

🔗 master

origin

🔗 origin/HEAD

Merge branch 'dev'

🔗 origin/dev

所有总集篇微调

第四章总集篇生成

第二章总集生成

Merge branch 'fix/chap4' into dev

第四章精校完成

Merge remote-tracking branch 'origin/fix' into dev

🔗 origin/fix

第二章精校

2.4 精校

2.3 精校

2.2 精校

第一章重新校订修改: 标题重新设置 无序列表修订

上传本地图片 删除多余的缩进

🔗 1.0.0

Merge branch 'dev'

更新readme.md

删除多余文件 笔记.md

总集篇重命名

删除空间域低通滤波.md

Merge branch 'feature/chapter6' into dev

规范文件架构 规范字符 公式加编号 补齐缺失的内容

删除一个多余文件

Merge branch 'feature/chapter2' into dev

初步校对完成 规范部分字符串 文件改名 文件架构规范

Merge branch 'feature/chapter5' into dev

规范文件格式

Merge branch 'feature/chapter5' into dev

规范文件格式

添加5.7

公式加编号 补齐部分章节缺失的内容 规范化某些行内公式

去除\mathrm{} 去除汉字之间的空格

Merge branch 'feature/chapter7' into local-dev

规范文件架构

Merge branch 'feature/chapter4' into local-dev

规范文件架构

Merge branch 'feature/chapter3----local' into local-dev

Merge branch 'feature/chapter8----local' into local-dev

文件夹重命名

Merge branch 'feature/chapter8----local' into local-dev

规范文件架构

总集去掉多余的目录

Merge branch 'feature/chapter1' into local-dev

Merge branch 'feature/chapter3----local' into local-dev

规范文件架构

规范文件架构

Merge branch 'feature/chapter8-local' into dev

8.5 代码字符替换

8.1 优化目录

8.5 上传一张图片

8.5改名

去除所有汉字之间的空格

去除所有的\mathrm{}

文件重命名

去除\mathrm{} 去除汉字之间的空格

去除\mathrm{} 去除汉字之间的空格 把所有的中文括号改为英文

Merge commit '6b644a02fea14cb1bc2e6ad4f3566c42771c0edf' into feature/chapter8

8.2 8.3微调

4.1-4.4补齐内容

Merge pull request #5 from MigoXV/feature/chapter3

去除所有两个汉字之间的括号 把所有中文括号替换为英文括号 修复小波变换的错别字

去掉所有的\mathrm{}

表格格式化

第三章粗加工完成

8.5 图片微调

3.1 完成

8.5 精加工

8.4 精加工

8.3 精加工

8.2 精加工

8.1改名

到2.4中间

第八章初步完成, 但无图无代码

第七章初步完成

7.1 开始

origin

微调

第四章初步完成, 未校对

第六章扫描完成, 未校对

6.1 半成品

第五章初步完成

5.1 粗整理完成

Date

Author

Commit

25 May 2023 15:20

MigoXV

73bc05d8

25 May 2023 15:20

MigoXV

34a04c00

25 May 2023 15:13

MigoXV

10a012de

25 May 2023 15:11

MigoXV

dc6517c9

25 May 2023 15:05

MigoXV

0916b676

25 May 2023 15:05

MigoXV

119f47cb

23 May 2023 14:54

MigoXV

5d7f0b32

3 May 2023 10:06

MigoXV

e4265629

3 May 2023 09:41

MigoXV

134f1af1

3 May 2023 09:34

MigoXV

97979028

1 May 2023 10:08

MigoXV

3e6d2be0

1 May 2023 09:37

MigoXV

afbe7971

30 Apr 2023 22:43

MigoXV

3ea2356a

28 Apr 2023 10:06

MigoPro4

04e66892

28 Apr 2023 10:05

MigoPro4

04d596e7

28 Apr 2023 10:03

MigoPro4

a67796ab

28 Apr 2023 10:02

MigoPro4

01365957

28 Apr 2023 10:00

MigoPro4

1fd38d51

28 Apr 2023 09:59

MigoPro4

dc2059d4

28 Apr 2023 09:58

MigoPro4

86282821

27 Apr 2023 16:28

MigoXV

809984d8

27 Apr 2023 16:21

MigoXV

a6ca280b

27 Apr 2023 16:20

MigoXV

f494428a

27 Apr 2023 09:21

MigoPro4

9ea056fe

27 Apr 2023 09:21

MigoPro4

a7ce62ea

27 Apr 2023 09:19

MigoPro4

51356b00

27 Apr 2023 09:18

MigoPro4

ed40bf27

27 Apr 2023 09:16

MigoPro4

4469a893

26 Apr 2023 16:54

MigoXV

fa23f04e

26 Apr 2023 10:59

MigoPro4

90f9ee4e

26 Apr 2023 10:33

MigoPro4

f94e9557

26 Apr 2023 10:32

MigoPro4

7582118c

26 Apr 2023 10:26

MigoPro4

c7419433

26 Apr 2023 10:26

MigoPro4

89217d4f

26 Apr 2023 10:22

MigoPro4

a2989a60

26 Apr 2023 10:21

MigoPro4

96a3584b

26 Apr 2023 10:21

MigoPro4

c00dba43

26 Apr 2023 10:20

MigoPro4

f4e48dc5

26 Apr 2023 10:19

MigoPro4

f7d94343

26 Apr 2023 10:15

MigoPro4

faf41dad

26 Apr 2023 10:09

MigoPro4

6981bc60

26 Apr 2023 10:06

MigoPro4

a5534818

26 Apr 2023 10:06

MigoPro4

02f4ea34

26 Apr 2023 10:01

MigoPro4

6ae335fc0

25 Apr 2023 20:21

MigoXV

55122b52

25 Apr 2023 20:15

MigoXV

c81c1377

25 Apr 2023 20:15

MigoXV

0f2a5126

25 Apr 2023 20:11

MigoXV

5765ed7f

25 Apr 2023 20:10

MigoXV

ac1f90f6

25 Apr 2023 20:01

MigoXV

7e57b84c

25 Apr 2023 19:56

MigoXV

b8348ba1

25 Apr 2023 19:52

MigoXV

6c39ba2c

25 Apr 2023 15:41

MigoPro4

c07757ce

25 Apr 2023 15:38

MigoPro4

2e3b5450

25 Apr 2023 15:26

MigoPro4

4184658f

25 Apr 2023 15:25

MigoPro4

f1371b19

23 Apr 2023 20:30

MigoXV

ea97278a

23 Apr 2023 15:35

MigoXV

02547654

23 Apr 2023 15:28

MigoXV

f641bd69

23 Apr 2023 15:20

MigoXV

4d3875f8

23 Apr 2023 15:18

MigoXV

ad221540

23 Apr 2023 14:04

MigoXV

dd99aa5f

19 Apr 2023 19:38

MigoXV

6b644a02

4 Apr 2023 17:18

MigoPro4

29f08f0a

4 Apr 2023 15:24

MigoPro4

8cd7428f

4 Apr 2023 15:08

MigoPro4

ba6c87a3

4 Apr 2023 14:53

MigoPro4

991b97c9

4 Apr 2023 14:38

MigoPro4

fe6366d0

31 Mar 2023 10:18

MigoPro4

c384c644

31 Mar 2023 10:16

MigoPro4

1757a410

29 Mar 2023 19:57

MigoPro4

69094723

29 Mar 2023 16:59

MigoPro4</

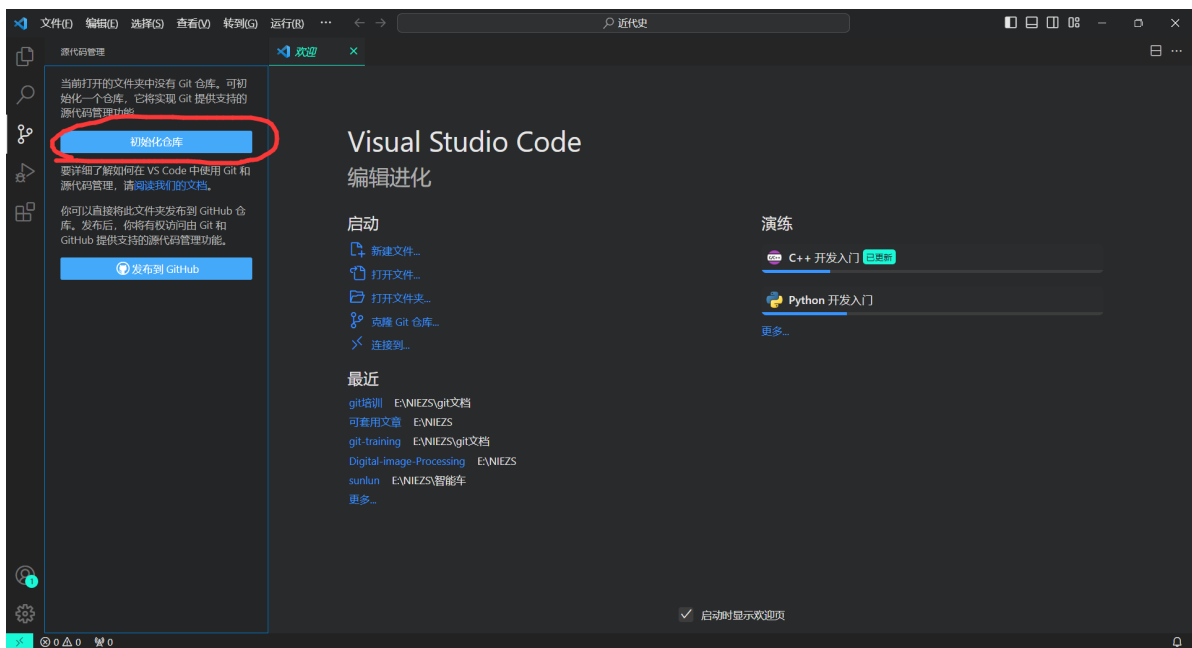


## 用GitGraph完成一些基本操作

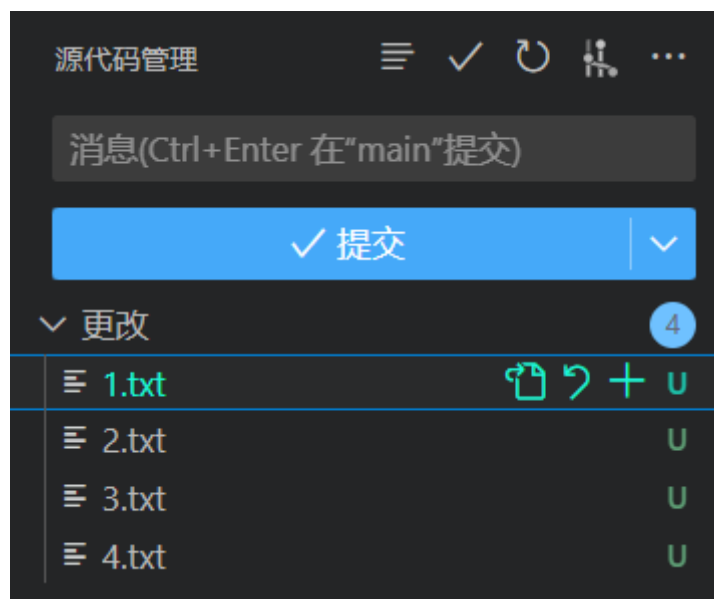
### 创建本地仓库

用vscode打开一个未创建过仓库的文件夹，然后点击左边栏的源代码管理选项，在这里进行所有关于git的操作。

点击初始化仓库，这个按钮帮助你完成 `git init` 的操作。



### 添加与提交更改

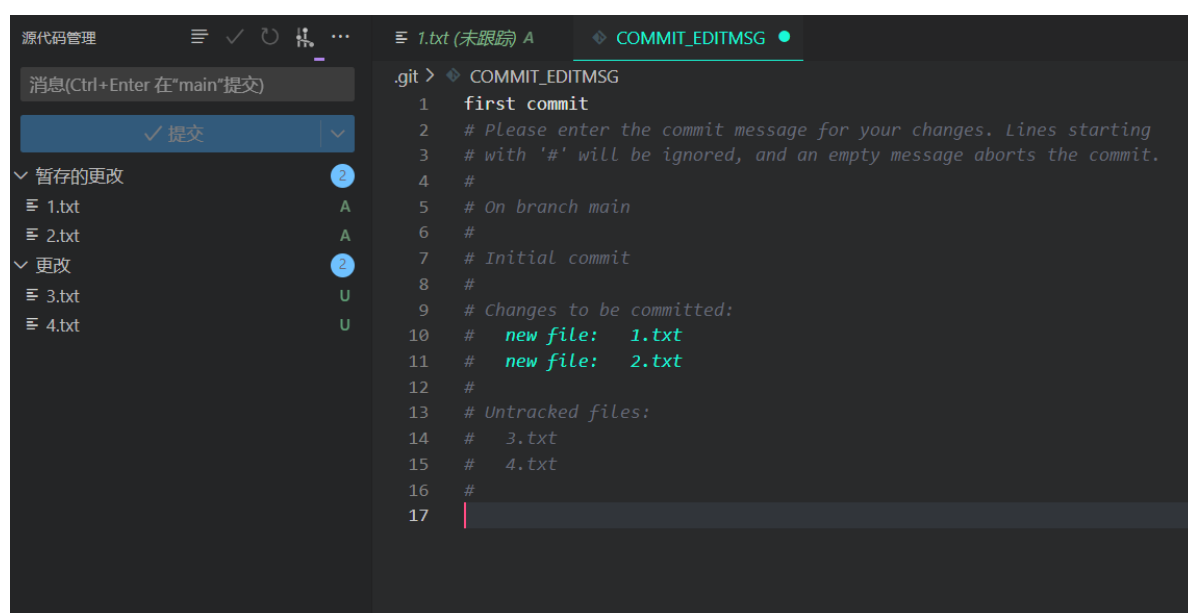


如果你在工作区中刚刚创建了新的文件，并且还未放入暂存区，那么默认处于未跟踪（untracked）状态，此时会显示绿色的“U”，选中此文件，点击“+”按钮，可以将其放入暂存区，完成文件的跟踪，这相当于 `git add` 指令，下图中，已经将“1.txt”“2.txt”放入了暂存区。

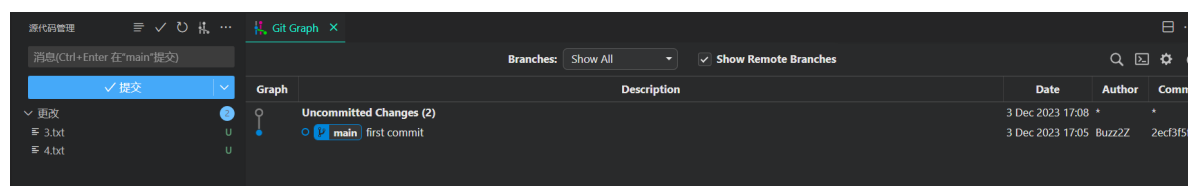




然后点击提交按钮，将暂存区的更改放入本地仓库，这相当于 `git commit` 指令，然后GitGraph会让你输入本次提交的信息，相当于 `git commit -m "Commit message"` 中 `Commit message` 的内容。

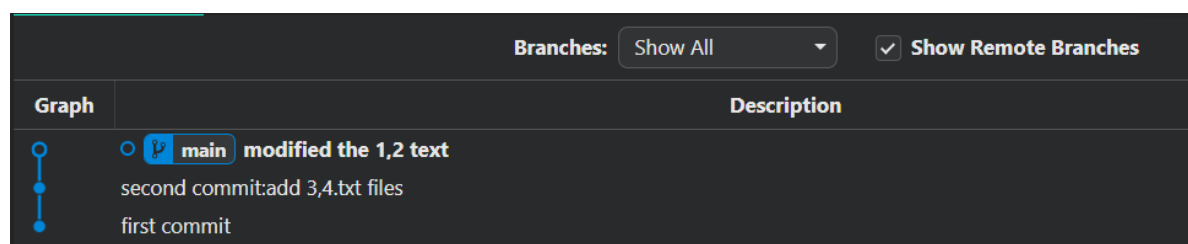


保存后退出“COMMIT\_EDITMSG”，完成一次提交，这时我们查看gitgraph树，发现已经完成一次提交。



## 查看差异

假设我已经完成了三次提交，git graph树变成了下面的样子。



其中，第一次提交：增加了1，2文本

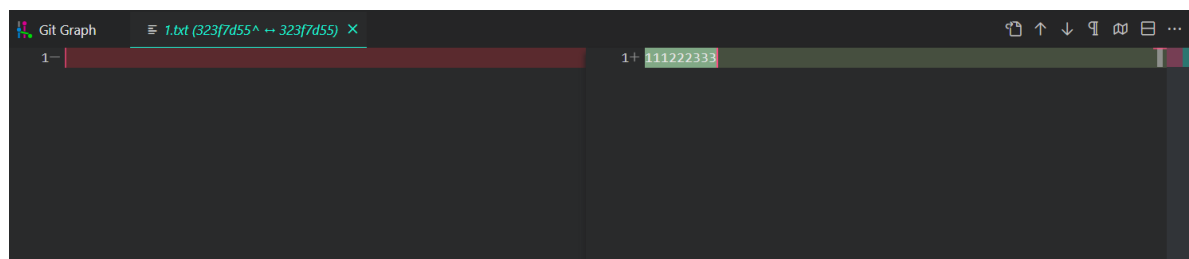
第二次提交：增加了3，4文本

第三次提交：修改了1, 2文本

我们要查看每次修改的差异，可以直接点击每次提交的标签，如图，我要查看第三次提交的更改，会显示以下内容。



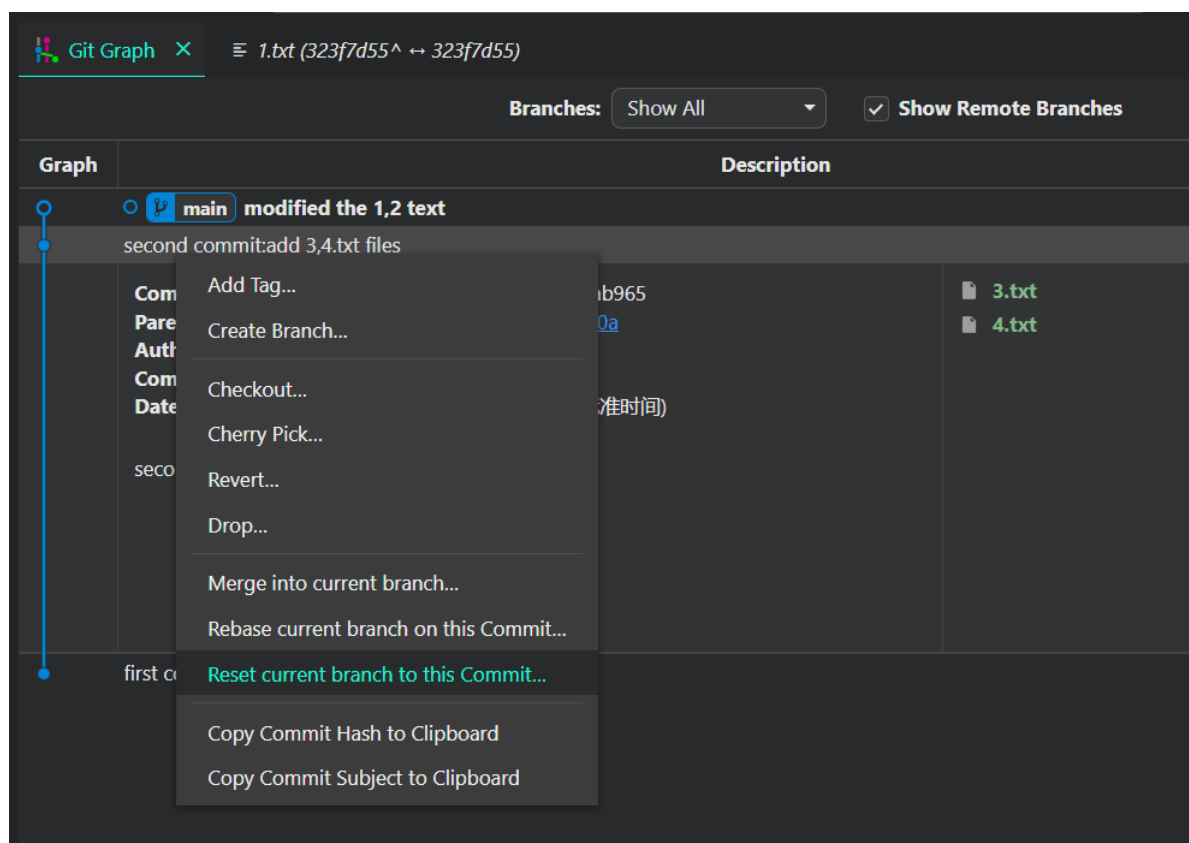
双击每一个修改，可以看到具体修改内容，比如双击图中1.txt，显示如下内容，这就完成了 `git diff` 的功能。

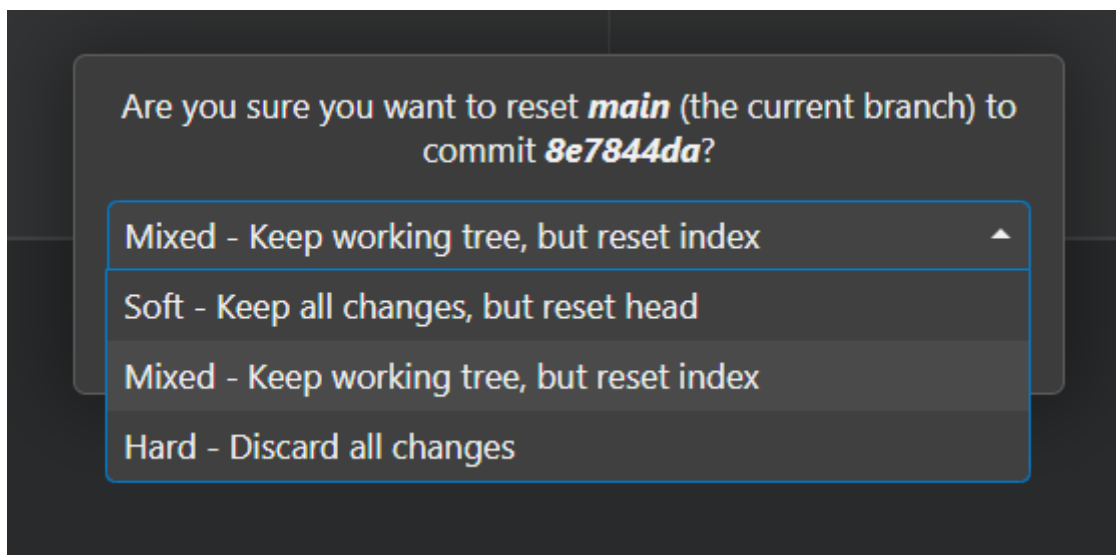


这表示我们添加了一行数字，内容为“111222333”。

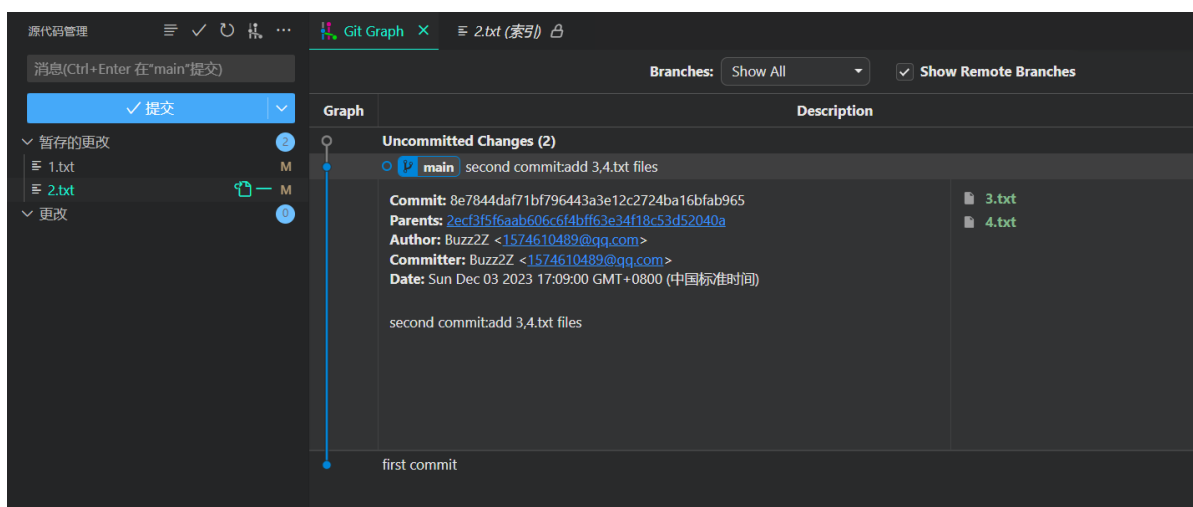
## 回退版本

直接右键每次提交的标签，选择Reset current branch to this Commit，选择回退方式，这相当于完成 `git reset` 指令。





这里我们选择soft方式回到第二次提交，可见，暂存区和工作区的更改都被保留了下来。



## 分支操作



。。。有待完善

