



1. **工作区 (Working Directory)**：这是你当前工作的目录，也就是文件的实际存在之处。在这里，你可以编辑、添加或删除文件。这个阶段的文件是未追踪或修改的。
2. **暂存区 (Staging Area/Index)**：当你运行 `git add` 命令后，修改过的文件会从工作区移动到暂存区。暂存区是一个中间区域，保存了即将提交到本地仓库的修改。这意味着你已经告诉 Git 要追踪这些修改，但是还没有正式保存它们到版本历史中。
3. **本地仓库 (Local Repository)**：当你运行 `git commit` 命令后，暂存区的修改将会提交到本地仓库中，成为一个新的提交。此时，修改已经被永久记录在 Git 的历史中，并存储在 `.git/objects` 中。

#### 操作流程：

- `git add`：将工作区的修改（新增、修改、删除的文件）放到暂存区。
- `git commit`：将暂存区的修改提交到本地仓库。

未跟踪

Untrack

未修改

Unmodified

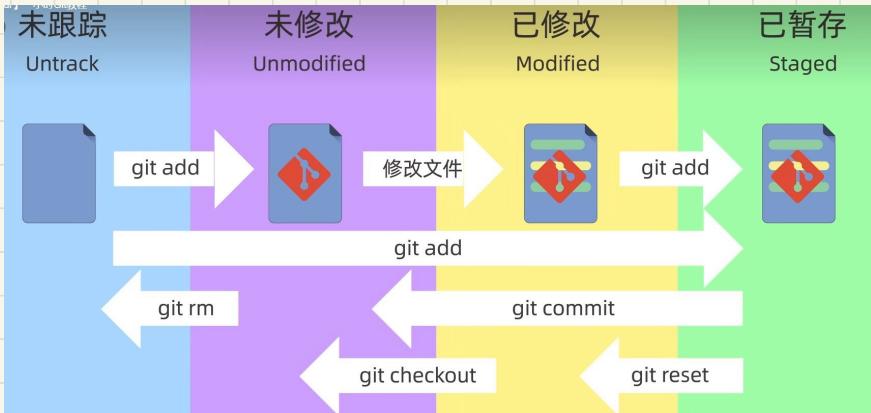
已修改

Modified

已暂存

Staged

1. 未跟踪 (Untrack) : 表示文件还没有被 Git 跟踪。通常是新创建的文件，Git 还没有记录它。此时，文件不会出现在版本控制中。
2. 未修改 (Unmodified) : 文件已被 Git 跟踪，并且当前没有任何修改。这个状态的文件与最近一次提交的版本完全一致。
3. 已修改 (Modified) : 文件已被 Git 跟踪，但自上次提交之后，文件的内容发生了修改。这意味着文件处于已更改但未保存到 Git 版本控制历史中的状态。
4. 已暂存 (Staged) : 文件的修改已经被 `git add` 命令添加到暂存区，准备好提交到本地仓库。此时，文件的变化还没有被真正提交，但已经标记为准备提交。



#### 1. 未跟踪 (Untrack) :

- 这个状态表示文件还没有被 Git 跟踪。
- 你可以使用 `git add` 命令将文件从未跟踪状态变为暂存状态。
- 如果你想删除这个文件，可以使用 `git rm` 命令。

#### 2. 未修改 (Unmodified) :

- 表示文件已经被 Git 跟踪且当前没有修改。
- 当你对文件进行修改时，文件状态会变成“已修改”。
- 如果你不想要保留修改，可以使用 `git checkout` 恢复文件到未修改状态。

#### 3. 已修改 (Modified) :

- 文件已经被更改，但还没有被放入暂存区。
- 你可以使用 `git add` 命令将这些更改添加到暂存区。
- 如果你不想要提交更改，可以使用 `git checkout` 恢复文件到未修改状态。

#### 4. 已暂存 (Staged) :

- 表示文件的更改已经被添加到暂存区，准备提交。
- 你可以使用 `git commit` 命令将这些更改提交到本地仓库。
- 如果你想撤销暂存的修改，可以使用 `git reset` 命令将文件从暂存区移回修改状态。

```
yiny@Ares ➤ ~/learn-git/my-repo ➤ main + echo "这是文件2的内容" > file2.txt
```

```
yiny@Ares ➤ ~/learn-git/my-repo ➤ main + git status  
On branch main
```

```
No commits yet
```

```
Changes to be committed:  
(use "git rm --cached <file>..." to unstage)  
      new file:   file1.txt
```

```
Untracked files:  
(use "git add <file>..." to include in what will be committed)  
      file2.txt
```



```
yiny@Ares ~ /learn-git/my-repo main + git commit -m "第一次提交"
[main (root-commit) 78de5c5] 第一次提交
 1 file changed, 1 insertion(+)
  create mode 100644 file1.txt
yiny@Ares ~ /learn-git/my-repo main git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file2.txt

nothing added to commit but untracked files present (use "git add" to track)
yiny@Ares ~ 这是因为我们已经把 file1 提交到仓库里面保管起来
```



```
to track)
yiny@Ares ~ /learn-git/my-repo main git add *.txt
yiny@Ares ~ /learn-git/my-repo main + git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   file2.txt
    new file:   file3.txt
    new file:   file4.txt
    new file:   file5.sh

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

通过命令 `git add *.txt`

→ `git commit -m "内容"`

→ Commit 只会执行  
将 staging index 中  
的文件提交到

Repository (绿色文)

件, 未 track 的文件  
不会被提交也就是  
红色 file2.txt

git add. ↓ 将所有文件都添加到 staging index

git log 查看提交纪录

git log --oneline 简单版提交纪录

git status      查看仓库的状态

git add          添加到暂存区

可以使用通配符，例如：git add \*.txt  
也可以使用目录，例如：git add .

git commit      提交

只提交暂存区中的内容，不会提交工作区中的内容

git log          查看仓库提交历史记录

可以使用 --oneline 参数来查看简洁的提交记录

# git reset的三种模式



## 1. `git reset --soft` :

- 作用：仅仅回退提交，不影响暂存区和工作区的内容。
- 暂存区：保留。
- 工作区：保留。
- 使用场景：当你想要回退提交但保留更改，以便稍后进行调整或重新提交。

## 2. `git reset --mixed` :

- 作用：回退提交，同时清除暂存区中的更改，但保留工作区的更改。
- 暂存区：清除。
- 工作区：保留。
- 使用场景：当你想回退提交并清空暂存区，但仍然保留修改的文件以便重新添加或修改。

## 3. `git reset --hard` :

- 作用：彻底回退提交，同时清除暂存区和工作区的更改。
- 暂存区：清除。
- 工作区：清除。
- 使用场景：当你想完全回滚到某个提交，不保留任何未提交的更改（包括工作区和暂存区中的更改）。

# git diff

工作区 VS 暂存区

git diff HEAD

工作区

+

暂存区

VS

本地仓库

git diff --cached /  
git diff --staged

暂存区

VS

本地仓库

git diff <commit\_hash> <commit\_hash> / 比较提交之间的差异  
git diff HEAD~ HEAD

git diff <branch\_name> <branch\_name> / 比较分支之间的差异

## 1. `git diff` :

- 比较工作区与暂存区之间的差异。
- 常用于查看哪些文件在工作区已被修改但还未暂存。

## 2. `git diff HEAD` :

- 比较工作区和暂存区的内容与本地仓库的最新提交（HEAD）之间的差异。
- 用于查看未提交的工作区修改和暂存区修改。

## 3. `git diff --cached` 或 `git diff --staged` :

- 比较暂存区和本地仓库的最新提交之间的差异。
- 常用于查看已暂存但还未提交的更改。

## 4. `git diff <commit_hash> <commit_hash>` 或 `git diff HEAD~ HEAD` :

- 比较两个提交之间的差异。
- 用于查看历史提交之间的修改情况，常用于代码审查。

## 5. `git diff <branch_name> <branch_name>` :

- 比较两个分支之间的差异。
- 用于查看不同分支的变化，特别是在合并或分支管理时非常有用。

# 删除

rm file; git add file	先从工作区删除文件, 然后再暂存删除内容
git rm <file>	把文件从工作区和暂存区同时删除
git rm --cached <file>	把文件从暂存区删除, 但保留在当前工作区中
git rm -r *	递归删除某个目录下的所有子目录和文件
	删除成功后不要忘记提交

-git ls-files ~已被追踪的文件

-ls -ltr

# 忽略

### 应该忽略哪些文件

.gitignore

忽略日志文件和文件夹	✓
忽略所有.class文件	✓
忽略所有.o文件	✓
忽略所有.env文件	✓
忽略所有.zip和tar文件	✓
忽略所有.pem文件	✓
.....	

- 系统或者软件自动生成的文件
- 编译产生的中间文件和结果文件
- 运行时生成日志文件、缓存文件、临时文件
- 涉及身份、密码、口令、秘钥等敏感信息文件

git commit -am  
||  
git add + git commit -m

只能忽略未跟踪文件

命令 `echo "modified" >> other.log` 的作用是将字符串 `"modified"` 追加到文件 `other.log` 的末尾。如果文件 `other.log` 不存在，系统会自动创建该文件并写入内容。具体解释如下：

- `echo "modified"` : `echo` 命令用于在终端输出一个字符串。在这个例子中，它会输出 `"modified"`。
- `>>` : 这是追加重定向操作符，表示将输出的内容追加到指定文件的末尾，而不是覆盖文件已有 的内容。
- `other.log` : 这是目标文件名。在这里，它表示我们要将字符串 `"modified"` 追加到 `other.log` 文件中。

如果 `other.log` 文件不存在，执行该命令会创建一个新文件 `other.log`，并且其内容为：`modified`  
这个命令用于将 `"modified"` 这一行文本追加到日志文件 `other.log` 中，不会覆盖文件的现有内容。

`git commit -a` 命令的作用是自动将已跟踪 ( tracked ) 的文件的修改进行提交，而不需要手动使用 `git add` 来将这些修改添加到暂存区。它会跳过对新文件的处理，即它不会自动添加未被 Git 跟踪的 ( 即新创建的 ) 文件或目录。

#### 具体解释：

- `git commit -a` 等同于 `git commit --all`，它会自动将所有已被跟踪文件的修改（包括修改和删除）添加到暂存区，然后提交。
- 这个命令不会处理新文件。对于新文件，仍然需要使用 `git add <file>` 命令将它们添加到暂存区。

#### 使用场景：

当你修改了一些已经被 Git 跟踪的文件，并且你不想手动使用 `git add` 来逐个暂存这些修改时，你可以直接使用 `git commit -a` 来一步完成暂存和提交的操作。

#### 示例：

##### 1. 有修改的已跟踪文件：

- 假设项目中已经有一个文件 `app.js` 被 Git 跟踪。你修改了 `app.js`，然后执行 `git commit -a -m "Updated app.js"`，这会自动将对 `app.js` 的修改提交，而无需手动 `git add app.js`。

##### 2. 有新添加的文件：

- 如果你新创建了一个文件 `new_file.js`，并且还没有使用 `git add` 将它添加到 Git 的跟踪中，那么 `git commit -a` 不会将该文件包含在提交中。你需要先运行 `git add new_file.js`，然后才能提交它。

当 `other.log` 已在 repository by  
`git ignore` 不会起作用

```

~/learn-git/my-repo [main] echo "hello" > temp/hello.txt
~/learn-git/my-repo [main] git status -s
?? temp/
~/learn-git/my-repo [main] vi .gitignore
~/learn-git/my-repo [main ±] git status -s
M .gitignore
~/learn-git/my-repo [main ±] git commit -am "test ignore folder"
[main 730b5bc] test ignore folder
1 file changed, 1 insertion(+)
~/learn-git/my-repo [main] git ls-files
.gitignore
file3.txt
file4.txt
file5.sh

```



?? temp

暂存区  
↓  
工作区

在 Git 状态中，?? temp/ 和 M .gitignore 表示不同的文件状态：

#### 1. ?? temp/ :

- ?? 表示未跟踪的文件。在这个例子中，temp/ 目录是未被 Git 跟踪的文件或文件夹。Git 发现这个目录存在，但它还没有被添加到暂存区，也就是说，这个文件夹并没有被纳入版本控制中。
- 要开始跟踪 temp/ 目录，你可以运行 git add temp/。不过，由于 .gitignore 可能已包含 temp/，所以这个文件夹被忽略，不会再被跟踪。

#### 2. M .gitignore :

- M 表示已修改的文件。在这里，.gitignore 文件被修改了，但尚未提交到仓库中。这意味着 Git 侦测到 .gitignore 文件的更改。
- 你可以通过 git add .gitignore 命令将修改暂存，或者使用 git commit -a 来自动暂存并提交所有已被跟踪的文件的更改。

总结：

- ?? temp/ : temp/ 目录是一个新创建且未被 Git 跟踪的目录。
- M .gitignore : .gitignore 文件已经被修改，但尚未提交。

#### 1. echo "hello" > temp/hello.txt :

- 这个命令在 temp 目录下创建了一个名为 hello.txt 的文件，文件内容为 "hello"。如果 temp 目录不存在，系统会自动创建它。

#### 2. git status -s :

- 这个命令以简洁模式显示 Git 仓库的当前状态。
- ?? temp/ 表示 temp 目录是一个未被 Git 跟踪的文件夹。

#### 3. vi .gitignore :

- 这个命令使用 vi 编辑器打开 .gitignore 文件进行修改。用户可能在 .gitignore 中添加了 temp/ 目录，以忽略它。

#### 4. git status -s :

- 再次运行这个命令后，输出显示 .gitignore 文件状态为 M，这意味着 .gitignore 文件已经被修改。

#### 5. git commit -am "test ignore folder" :

- a 选项会自动暂存所有已被 Git 跟踪的文件的修改（例如 .gitignore 文件），-m 选项用在单一命令中添加提交消息（这里的消息是 "test ignore folder"）。
- 这个命令提交了对 .gitignore 文件的更改，但不会提交未被跟踪的 temp/ 目录，因为它已经被忽略。

#### 6. git ls-files :

- 这个命令列出了当前被 Git 跟踪的文件。输出显示 .gitignore、file3.txt、file4.txt 和 file5.sh 文件被跟踪，而 temp/ 目录没有被跟踪（因为 .gitignore 中忽略了它）。

总结：用户创建了一个未被跟踪的目录 temp，并将其添加到 .gitignore 文件中以忽略该目录，然后提交了对 .gitignore 文件的更改，最后通过 git ls-files 验证哪些文件被 Git 跟踪。

# ignore 匹配 rule

—— 以上到下逐行匹配，每行表示一个忽略模式

```
# 忽略所有的 .a 文件  
*.a  
  
# 但跟踪所有的 lib.a, 即便你在前面忽略了 .a 文件  
!lib.a  
  
# 只忽略当前目录下的 TODO 文件, 而不忽略 subdir/TODO  
/TODO  
  
# 忽略任何目录下名为 build 的文件夹  
build/  
  
# 忽略 doc/notes.txt, 但不忽略 doc/server/arch.txt  
doc/*.txt  
  
# 忽略 doc/ 目录及其所有子目录下的 .pdf 文件  
".gitignore" 17L, 383B
```

可以去 [github](#) 查 gitignore  
裡面会有常见的 rule 直接拿  
来用。

# SSH 設置

```
User@DESKTOP-RIKE5S4 MINGW64 ~
$ ssh-keygen -t ed25519
Generating public/private key pair.
Enter file in which to save the key (/c/Users/User/.ssh/id_ed25519): githubkey
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in githubkey
Your public key has been saved in githubkey.pub
The key fingerprint is:
SHA256:sTvaKraAkhOcnUGu8ftjTnVA828ATH2FEBwnQwdjaVA User@DESKTOP-RIKE5S4
The key's randomart image is:
+--[ED25519 256]---+
|   . o**E*oo. |
|   o ..==o   |
|   . o   .oo. |
| . * o   .oo  |
|. +     .S. o  |
| +   .   .   . |
|= ..   . o    |
|.. .=o o .    |
|   .+=+.    |
+---[SHA256]-----+
```

↓ 按兩次  
enter

生成两个新文件 \*

github.pub → 公钥文件，上传到github

githubkey → 私钥文件 谁也不要给

```
~/.ssh tail -5 config  
# github  
Host github.com  
HostName github.com  
PreferredAuthentications publickey  
IdentityFile ~/.ssh/test
```

→如沒有 config 檔自己手動新建一下

當我訪問 github 時 指定使用 SSH 下的 githubkey 這個密鑰

```
x ~  
~/.ssh cd .ssh  
ssh-keygen -t rsa -b 4096
```

也可以用此命令設置 ssh

# 完整SOP(在ssh設置完後)

```
User@DESKTOP-RIKE554 MINGW64 ~/iris  
$ git init my-repo  
Initialized empty Git repository in C:/Users/User/iris/my-repo/.git/
```

```
User@DESKTOP-RIKE554 MINGW64 ~/iris  
$ cd my-repo
```

```
User@DESKTOP-RIKE554 MINGW64 ~/iris/my-repo (master)  
$ git clone git@github.com:temonms1/Remote-iris-flask.git → git clone  
Cloning into 'Remote-iris-flask'...  
warning: You appear to have cloned an empty repository.
```

```
User@DESKTOP-RIKE554 MINGW64 ~/iris  
$ ls  
Remote-iris-flask/ my-repo/
```

```
User@DESKTOP-RIKE554 MINGW64 ~/iris  
$ cd Remote-iris-flask
```

```
User@DESKTOP-RIKE554 MINGW64 ~/iris/Remote-iris-flask (main)  
$ git status  
On branch main  
  
No commits yet  
  
Untracked files:  
  (use "git add <file>..." to include in what will be committed)  
    __pycache__/  
    app.py  
    iris_model.pth  
    static/  
    templates/  
    train_model.py  
  
nothing added to commit but untracked files present (use "git add" to track)
```

```
User@DESKTOP-RIKE554 MINGW64 ~/iris/Remote-iris-flask (main)  
$ git add .
```

```
User@DESKTOP-RIKE554 MINGW64 ~/iris/Remote-iris-flask (main)  
$ git status  
On branch main  
  
No commits yet  
  
Changes to be committed:  
  (use "git rm --cached <file>..." to unstage)  
    new file:   __pycache__/train_model.cpython-310.pyc  
    new file:   __pycache__/train_model.cpython-312.pyc  
    new file:   app.py  
    new file:   iris_model.pth  
    new file:   static/button.js  
    new file:   static/particles.js  
    new file:   static/style.css  
    new file:   templates/index.html  
    new file:   train_model.py
```

```
User@DESKTOP-RIKE554 MINGW64 ~/iris/Remote-iris-flask (main)  
$ git commit -m "第一次提交"  
[main (root-commit) a171918] 第一次提交  
 9 files changed, 457 insertions(+)  
create mode 100644 __pycache__/train_model.cpython-310.pyc  
create mode 100644 __pycache__/train_model.cpython-312.pyc  
create mode 100644 app.py  
create mode 100644 iris_model.pth  
create mode 100644 static/button.js  
create mode 100644 static/particles.js  
create mode 100644 static/style.css  
create mode 100644 templates/index.html  
create mode 100644 train_model.py  
  
User@DESKTOP-RIKE554 MINGW64 ~/iris/Remote-iris-flask (main)  
$ git push → push  
Enumerating objects: 14, done.  
Counting objects: 100% (14/14), done.  
Delta compression using up to 16 threads  
Compressing objects: 100% (13/13), done.  
Writing objects: 100% (14/14), 11.43 KiB | 1.43 MiB/s, done.  
Total 14 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)  
To github.com:temonms1/Remote-iris-flask.git
```

# 將本地倉庫關連到 remote repo

```
User@DESKTOP-RIKE5S4 MINGW64 ~/iris  
$ cd my-repo  
  
User@DESKTOP-RIKE5S4 MINGW64 ~/iris/my-repo (master)  
$ git remote add origin git@github.com:temonmsl/Remote-iris-flask.git  
  
User@DESKTOP-RIKE5S4 MINGW64 ~/iris/my-repo (master)  
$ git remote -v  
origin git@github.com:temonmsl/Remote-iris-flask.git (fetch)  
origin git@github.com:temonmsl/Remote-iris-flask.git (push)  
  
User@DESKTOP-RIKE5S4 MINGW64 ~/iris/my-repo (master)  
$ ls  
app3-bg/  
  
User@DESKTOP-RIKE5S4 MINGW64 ~/iris/my-repo (master)  
$ git remote add origin git@github.com:temonmsl/Remote-iris-flask-2024.git  
error: remote origin already exists.  
  
User@DESKTOP-RIKE5S4 MINGW64 ~/iris/my-repo (master)  
$ git remote set-url origin git@github.com:temonmsl/Remote-iris-flask-2024.git  
  
User@DESKTOP-RIKE5S4 MINGW64 ~/iris/my-repo (master)  
$ git remote -v  
origin git@github.com:temonmsl/Remote-iris-flask-2024.git (fetch)  
origin git@github.com:temonmsl/Remote-iris-flask-2024.git (push)  
  
User@DESKTOP-RIKE5S4 MINGW64 ~/iris/my-repo (master)  
$ git branch -M main  
  
User@DESKTOP-RIKE5S4 MINGW64 ~/iris/my-repo (main)
```

→ git remote add origin [url]  
→ 查看當前配置的遠程 repository

× 因為配置過了

→ 要修改 remote repo 需用 set-url

github 默認分支是 main  
而 git 是 master, 所以要將分支  
改名為 main, 用 git branch  
-M main

```
User@DESKTOP-RIKE554 MINGW64 ~/iris/my-repo (main)
$ git push -u origin main
Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 16 threads
Compressing objects: 100% (13/13), done.
Writing objects: 100% (15/15), 11.46 KiB | 1.04 MiB/s, done.
Total 15 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To github.com:temonmsl/Remote-iris-flask-2024.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

接下來就可以將本地的 repo 中的檔案 push 到遠程倉庫  
☆注意：如果失敗的話，一定要先做初始化（也就是 add 跟 commit）

git pull ↓：如果在 remote repo 新增文件，可以用 git pull 拉取到本地

```
User@DESKTOP-RIKE554 MINGW64 ~/iris/my-repo (main)
$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 1.32 KiB | 224.00 KiB/s, done.
From github.com:temonmsl/Remote-iris-flask-2024
  6b03c09..32b78ee main      -> origin/main
Updating 6b03c09..32b78ee
Fast-forward
 README.md | 12 ++++++-----
 1 file changed, 12 insertions(+)
 create mode 100644 README.md
```

把 remote repo 的指定分支拉取到本地，再進行合併

```
User@DESKTOP-RIKE554 MINGW64 ~/iris/my-repo (main)
$ ls
README.md  app3-bg/
```

# 系统 整合

## 1. git remote add origin git@github.com:temonmsl/Remote-iris-flask.git

- 功能：将远程仓库的地址添加到你的本地 Git 仓库中，并命名为 `origin`。
- 详细解释：
  - `git remote add`：用于添加远程仓库。
  - `origin`：这是远程仓库的默认名称，表示你要给这个远程仓库起的名字。在大多数情况下，`origin` 被用作默认的远程仓库名称。
  - `git@github.com:temonmsl/Remote-Iris-flask.git`：这是远程仓库的 SSH 地址。它表示你想将这个 GitHub 仓库链接到本地的 Git 仓库。

## 2. git branch -M main

- 功能：将当前分支重命名为 `main`。
- 详细解释：
  - `git branch`：用于查看、创建或管理分支。
  - `-M`：表示强制重命名当前分支。如果分支已存在，它会覆盖旧的分支名称。
  - `main`：这是新的分支名称。Git 默认的分支名称曾经是 `master`，但近年来，许多项目将默认分支名称改为 `main`。

## 3. git push -u origin main

- 功能：将本地 `main` 分支的内容推送到远程仓库的 `main` 分支，并将该远程分支设为默认的上游分支。
- 详细解释：
  - `git push`：用于将本地的更改推送到远程仓库。
  - `-u`：这个选项告诉 Git 记录这个推送的远程分支，并将其设为上游分支。这样你在以后推送时，可以只使用 `git push` 而不需要指定分支和远程仓库。
  - `origin`：这是你之前设置的远程仓库名称。
  - `main`：这是你推送的目标分支，表示你将本地的 `main` 分支推送到远程仓库中的 `main` 分支。

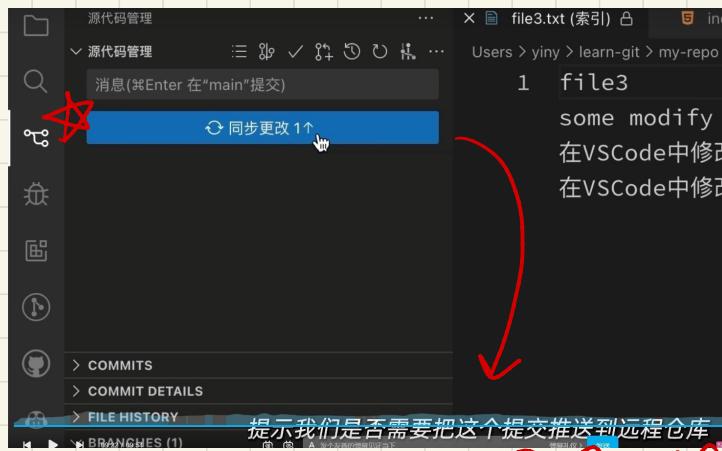
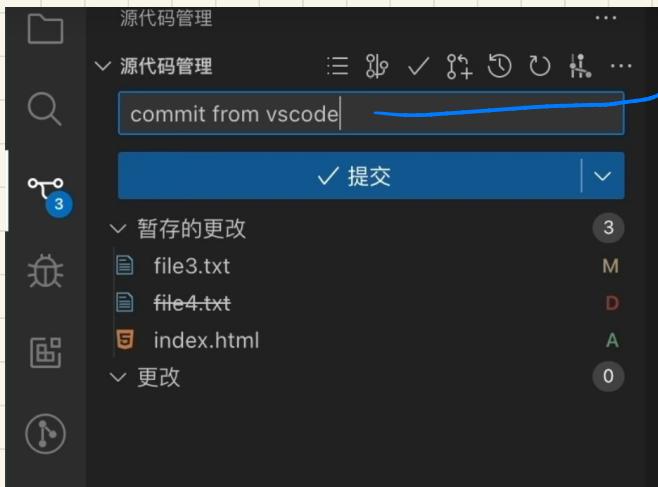
### 总结：

- 第一个命令将远程仓库链接到本地仓库。
- 第二个命令将当前分支命名为 `main`。
- 第三个命令将本地 `main` 分支推送到远程仓库的 `main` 分支，并将 `main` 分支设置为默认的上游分支，用于后续的推送和拉取操作。

这些步骤可以让你的本地代码与 GitHub 上的远程仓库保持同步。

# Vscode

?? (Untracked) : 未跟踪  
M (Modified) : 已修改  
A (Added) : 已添加暂存  
D (Deleted) : 已删除  
R (Renamed) : 重命名  
U (Updated) : 已更新未合并



查看分支列表: \$ git branch  
创建分支: \$ git branch branch-name  
切换分支: \$ git checkout branch-name  
【推荐】\$ git switch branch-name  
合并分支: \$ git merge branch-name  
删除分支: 【已合并】\$ git branch -d branch-name  
【未合并】\$ git branch -D branch-name

假如有 main 和 feat 分支

git merge feat

代表将 feat 分支合并到 main 分支

Vim 中删除一行的命令是 dd

git commit -a -m

“first commit”

可以一次完成 add 和 commit

只对已经添加过的文件有效, 比如要修改文件内容·如果是新文件会没作用



两个分支未修改同一个文件的同一处位置：Git 自动合并

两个分支修改了同一个文件的同一处位置：产生冲突

解决方法：

Step1 - 手工修改冲突文件，合并冲突内容

Step2 - 添加暂存区 \$ git add file

Step3 - 提交修改 \$ git commit -m "message"

中止合并：当不想继续执行合并操作时可以使用下面的命令来中止合并过程：

\$ git merge --abort

\$ git switch dev  
\$ git rebase main

# Rebase

\$ git switch main  
\$ git rebase dev

dev:2  
dev:1  
main:5  
main:4  
main:3  
main:2  
main:1

HEAD | main:5  
main:4  
main:3  
main:2  
main:1  
HEAD | dev:2  
dev:1

如果我们在 dev 分支上执行 rebase 操作

main:5  
main:4  
dev:2  
dev:1  
main:3  
main:2  
main:1

The screenshot shows a Git commit history with the following commits:

- merge conflict (between main and dev)
- feat:1
- Merge branch 'dev'
- main:6
- main:5
- main:4
- main:3
- main:2
- main:1
- dev:2
- dev:1
- main:3
- main:2
- main:1

Terminal commands shown:

```
~/learn-git/branch-demo % git branch -d feat
Deleted branch feat (was 2436710).
~/learn-git/branch-demo % git checkout -b dev 244d3
```

刪除分支

→ 復原分支 -b branch-name id

在截图中，命令 `git log --oneline --graph --decorate --all` 的功能如下：

- `git log`：显示 Git 仓库的提交历史。
- `--oneline`：将每个提交的日志信息显示在一行上，通常显示的是提交的哈希值和提交信息的简短版本。
- `--graph`：显示分支和提交历史的图形化表示，使你可以更直观地看到分支的分叉和合并。
- `--decorate`：为提交信息添加标签或分支名的装饰，方便了解哪些提交关联了标签或分支。
- `--all`：显示所有的分支和标签的提交记录，而不仅仅是当前分支。

这个命令的作用：

它以简洁的格式显示项目的所有提交历史，并用图形展示分支和合并信息，同时标注出当前提交在哪些分支或标签上。非常适合用于查看项目的整体提交情况和分支结构。

```
|/  
* ae24b56 Merge branch 'dev'  
|\  
| * 244d356 (HEAD -> dev) dev:2  
| * 19570e0 dev:1  
* | b4d139d main:5  
* | 41daeb9 main:4
```

输出结果

也可用 `git log` 查看 ID

用 alias 取别名

```
~/learn-git/branch-demo % dev git log --oneline --graph  
--decorate --all  
~/learn-git/branch-demo % dev alias graph="git log --one  
line --graph --decorate --all"  
这样以后我们就可以直接使用graph命令来查看图形化的提交记录了
```

# Merge

优点：不会破坏原分支的提交历史，方便回溯和查看。

缺点：会产生额外的提交节点，分支图比较复杂。

## Rebase

优点：不会新增额外的提交记录，形成线性历史，比较直观和干净；

缺点：会改变提交历史，改变了当前分支branch out的节点。  
避免在共享分支使用。

```
cp -rf branch-demo rebase1
```

解释：

- `cp`：这是 Linux/Unix 下用于复制文件或目录的命令。
- `-r`：表示递归复制 ( recursive )，即如果 `branch-demo` 是一个目录，会复制该目录及其所有子目录和文件。
- `-f`：表示强制复制 ( force )，如果目标位置已经存在同名文件或目录，它会强制覆盖而不提示。
- `branch-demo`：这是源文件或目录的名称，即你想要复制的文件或目录。
- `rebase1`：这是目标文件或目录的名称，复制的文件或目录会被重命名为 `rebase1` 或复制到 `rebase1` 目录下 ( 如果 `rebase1` 已经存在 )。

总结：

这条命令会将 `branch-demo` 目录及其所有内容递归地复制到 `rebase1` 目录。如果 `rebase1` 已经存在，命令会强制覆盖。