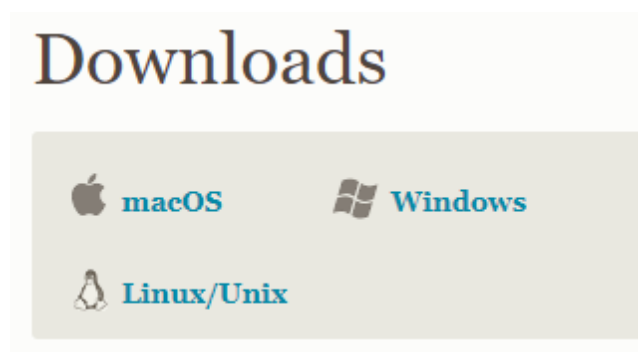
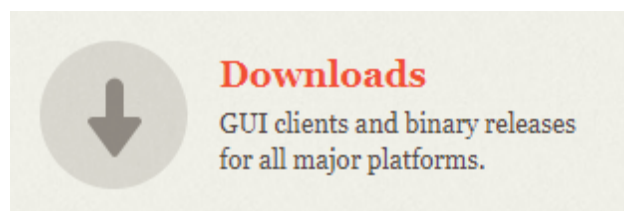


官网下载

<https://git-scm.com>



git -v 检查安装是否完成



git的三种使用方式

Git的使用方式



命令行



图形化界面（GUI）



IDE插件/扩展

推荐使用命令行来加深理解。

常用命令如下

1. 安装好 git 之后要先配置用户名和邮箱，用来在提交的时候分辨提交的用户

```
1 # 1. 配置用户名"Jasper Yang", 之所以用双引号括住因为中间有空格, 如果没有空格则可以省略
2 git config --global user.name "Jasper Yang"
3
4 不写这个参数则表示本地配置, 只对本地仓库有效
5 填写参数 --global 表示全局配置, 所有仓库生效 (使用最多)
6 填写参数 --system 系统配置, 对所有用户生效 (一般不会使用这个)
7
8 # 2. 配置邮箱
9 git config --global user.email geekhall.cn@gmail.com
10
11 # 3. 可以选择保存用户名和密码, 省去每次的输入
12 git config --global credential.helper store
13
14 # 4. 使用下面命令来查看配置了的用户名和邮箱和其他信息
15 git config --global --list
```

配置用户名和邮箱只需要执行一次, 如果之前配置过则可以不用再配置。

--list 的结果如下

如何新建版本库/仓库来进行版本管理

创建仓库十分简单, 只需把一个目录变成 git 可以管理的仓库即可。

两种方式创建仓库

一种是直接创建, 另一种是克隆云端

直接创建

蓝色括号 master 就是当前所在的分支的名称

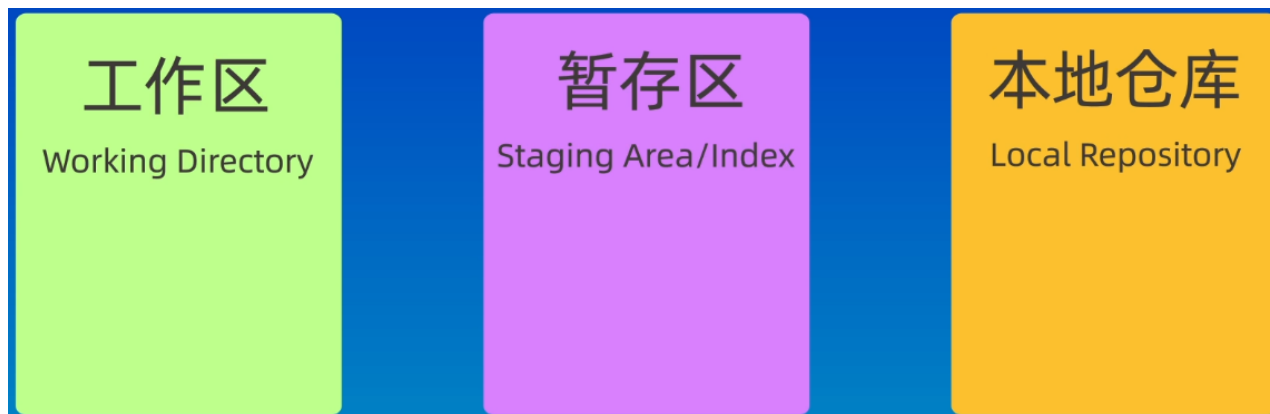
.git 是一个隐藏目录, 需要 ls -a 显示所有文件才能查看到

git init 后面可以指定目录的名称, 这是会在当前目录下创建一个新的目录作为 git 仓库。

第二种方式是直接clone远程仓库到本地，完成一个仓库的生成

Git 的工作区域和文件状态

Git的本地数据管理分为三个区域



Working Directory 工作区

就是我们自己电脑上的目录

Staging Area / Index 暂存区

用于保存即将提交到Git仓库的修改内容

Local Repository 本地仓库

本地仓库就是我们上一节课通过 `git init` 命令创建的那个仓库

它包含了完整的项目历史和元数据

修改完工作区的文件后，需要添加到暂存区，再将暂存区的修改提交到本地仓库。



添加和提交文件

上一节课我们已经通过 `git init` 命令创建了一个本地仓库

`git init` 创建仓库

`git status` 查看仓库的状态

`git add` 添加到暂存区

`git commit` 提交

下面就来看一下怎样将文件添加到仓库里面

```
贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/learn-git (master)
$ git status
On branch master

No commits yet

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

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/learn-git (master)
$
```

首先来创建一个文件

```
贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/learn-git (master)
$ pwd
/d/developer_tools/Git/workbench/learn-git

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/learn-git (master)
$ echo "this is the first file" > file1.txt

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/learn-git (master)
$ ll -rt
total 1
-rw-r--r-- 1 贝爷HelloWorld 197121 23 Sep 29 08:19 file1.txt

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/learn-git (master)
$
```

查看一下仓库的状态 `git status`

```
贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/learn-git (master)
$ ll -rt
total 1
-rw-r--r-- 1 贝爷HelloWorld 197121 23 Sep 29 08:19 file1.txt

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/learn-git (master)
$ cat file1.txt
this is the first file

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/learn-git (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file1.txt

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

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/learn-git (master)
$ |
```

下面这个提示告诉我们可以使用这个命令把添加到暂存区的文件再取消暂存

`git commit`

下面是没有指定 `-m` 时会进入的交互界面

可以使用 `git log` 命令来查看提交记录

`git log --oneline` 来查看简洁的提交记录

`git commit -am`

将所有已跟踪文件的更改添加到暂存区。

使用提供的提交信息创建一个新的提交。

git reset 回退版本

在日常开发的时候

我们经常会需要撤销之前的一些修改内容

或者回退到之前的某一个版本

这个时候 `git reset` 这个命令就派上用场了

`reset` 命令用于回退版本，可以退回到之前的某一个提交的状态。

`git reset` 的三种用法

```
1 git reset --soft <versionId>
2 # soft 表示回退到某一个版本，并且保留工作区和暂存区的所有修改内容
3
4
5 git reset --hard <versionId>
6 # hard 表示回退到某一个版本，丢弃工作区和暂存区的所有修改内容。（谨慎使用）
7
8
9 git reset --mixed <versionId>
10 # mixed 则介于 soft 和 hard 两个参数之间，回退到某一个版本，并且只保留工作区的修改内容，丢弃暂
    存区的修改内容。它是 reset 命令的默认参数。
11
12 eg.
13 # 用于将当前分支的 HEAD 指针重置到其父提交（即上一个提交）
14 git reset HEAD^
15 # 回退到指定版本(不指定参数则默认 mixed)
16 git reset 49c7e4a
17
18 #####
19
20 # 通过 git log 或者 git log --oneline 来获得版本号id
21 $ git log
22    commit 49c7e4a5aebbb1b6f0834187969872cfe6fb5df3 (HEAD -> master)
23    Author: Curtis Chen <chw_1118@outlook.com>
24    Date:    Sun Sep 29 08:30:40 2024 +0800
25
26        second commit
27
28    commit 0422638931e90fa9ce55c1bce113c5f3780bd322
29    Author: Curtis Chen <chw_1118@outlook.com>
30    Date:    Sun Sep 29 08:28:28 2024 +0800
31
32        first commit
33
34 $ git log --oneline
35 49c7e4a (HEAD -> master) second commit
36 0422638 first commit
```


git ls-files 可以查看暂存区状态，列出暂存区有哪些文件。

一般当我们连续提交了多个版本，但觉得这些提交没有太大作用和意义，其实可以进一步合并成一个版本时，就可以通过

git reset --soft 或 git reset --mixed 来进行回退到这些多个提交之前的某一个版本，再进行重新提交。

注意，每次文件发生修改之后，都要进行 add，然后才能 commit。

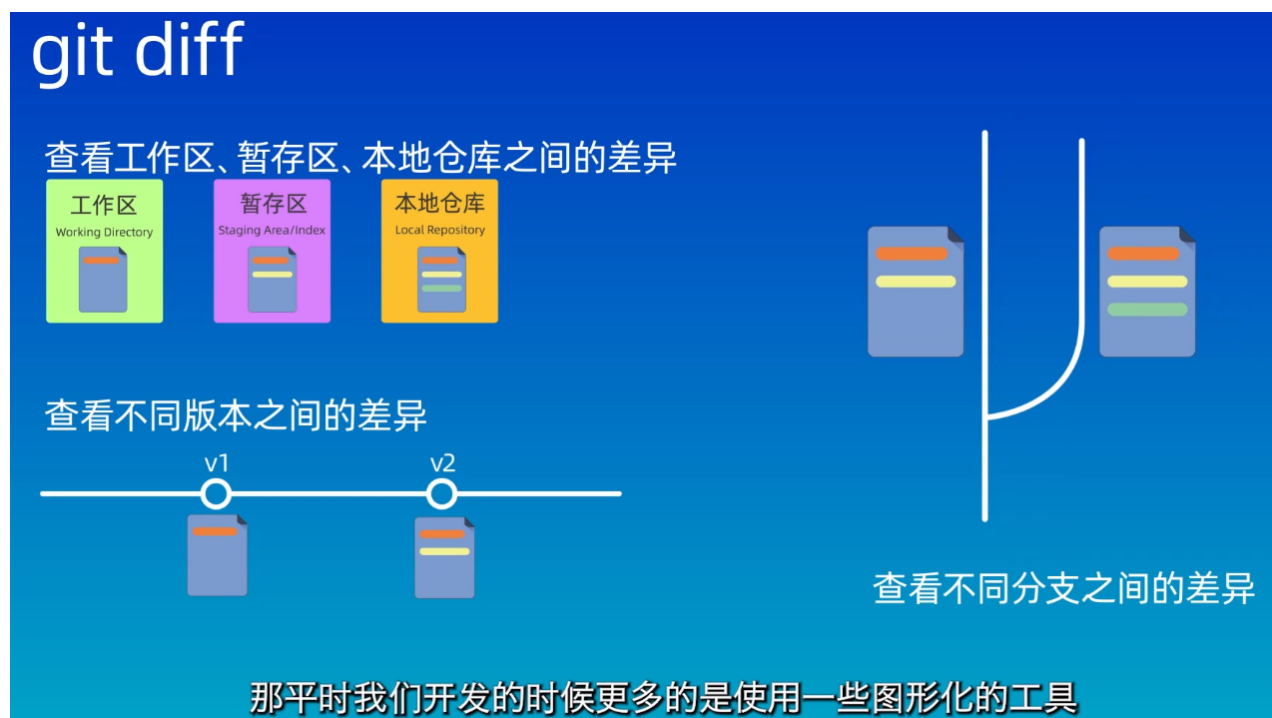
git 中的所有操作都可回溯，可以使用 git reflog 命令查看操作的历史纪录。

如果不小心误操作，可以通过 git reflog 命令找到想要的版本号，再使用 reset 把版本置过去。

假设按上图来说，最后一次 commit 是回退到了 Head 指针的父提交，完成回退之后的版本号是 0422638。

那完成这次回退之前的版本号就是下面的这个 49c7e4a，想回到这个版本就拿它去 reset 就好了

git diff 查看差异



git diff 默认比较工作区和暂存区之间的差异内容，显示发生更改的文件以及更改的详细信息

目前分别有3次commit

```
贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/repo (master)
$ git log --oneline
f486336 (HEAD -> master) 3rd commit
af10768 2nd commit
aa11d5b 1st commit
```

```
贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/repo (master)
$ cat file1.txt
111
```

```
贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/repo (master)
$ cat file2.txt
222
```

```
贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/repo (master)
$ cat file3.txt
333
```

现在修改 file3.txt 的内容并且 wq

直接使用 git diff 命令查看

第一行提示发生变更的文件。

第二行：git 将文件内容使用哈希算法生成一个 40 位哈希值，这里只显示哈希值的前7位 55bd0ac。后面的数字 100644 则表示文件的权限。

再往下，红色表示删除的内容，绿色表示添加的内容。

这里现在比较的是工作区和暂存区的差异。

下面对这个 file3.txt 进行添加到暂存区，然后再 git diff 查看区别

发现没有输出，工作区和暂存区现在是保持一致的。

以上是比较工作区和暂存区。

下面来比较工作区和版本库之间的差异。git diff HEAD

目前我们给 file3.txt 添加的内容还没有 commit 到仓库，那工作区和仓库理应存在差异

下面比较暂存区和版本库的差异。git diff --cached

注意我们已经把 file3.txt 提交到暂存区了，但是暂存区还没 commit 到仓库，所以应该是有差异的。

下面我们 commit 这个 file3.txt，然后再次执行 git diff HEAD 和 git diff --cached。会发现已经不存在差异了

git diff 除了比较工作区，暂存区和版本库之间的差异外，还可以用来比较两个特定版本之间的差异。

用法就是后面加上两个版本的提交 id 即可。git diff <oldVersionId> <newVersionId>

HEAD 可以表示当前分支的最新提交。

下面是上一个版本和 HEAD 的比较。git diff f486336 HEAD

由于每次都要查看提交 id 会很麻烦，对于常用的比较当前版本和上一版本之前的差异，git 提供了便捷方式，用 HEAD~ 表示上一个版本。HEAD 则是当前版本。

git diff HEAD~ HEAD

HEAD 表示当前已提交的最新版本。

HEAD^ 等同于 HEAD~，表示当前已提交版本的上1个版本。

HEAD~2 表示当前已提交版本的上2个版本。

HEAD~3 表示当前已提交版本的上3个版本。

git diff 后可加文件名，如此则只会查看这个文件的差异内容。git diff HEAD~3 HEAD file3.txt

git diff 后加分支名字则可以查看分支之间的差异。

从版本库中删除文件 git rm

普通方法：

1. 从工作区中删除文件，rm -f file4.txt
 2. 此时暂存区中file4.txt 还存在，所以要 git add . 将这个变化添加到暂存区。
 3. 此时工作区和暂存区都不存在这个文件了。
 4. 从仓库中删除这个文件就要把这个变化 commit。git commit -m "delete file4.txt"
- 到这里一个文件的删除就结束了。

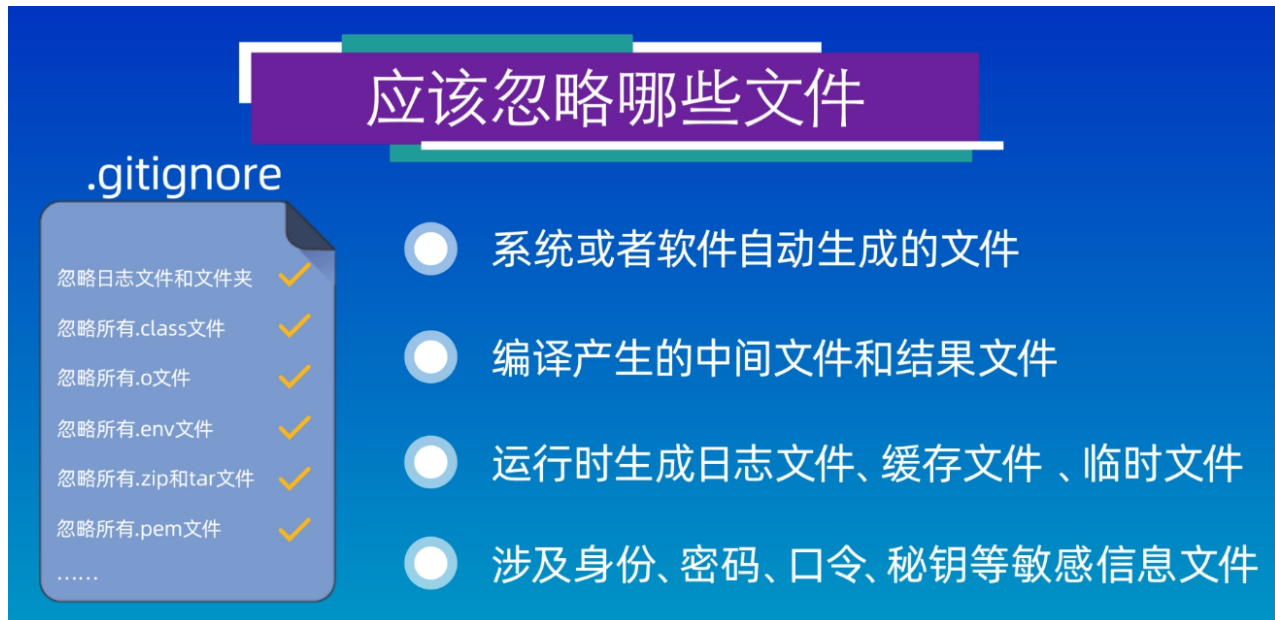
使用 git rm 的方法

1. 从工作区和暂存区中同时删除文件。git rm file4.txt
2. 这是工作区和暂存区都不存在这个文件了。git status 查看到这个 change 已经是等待 commit 的状态了。

3. 用 `git ls-files` 可以查看暂存区中的文件。

4. 通过 `commit` 从仓库中删除这个文件。 `git commit -m "delete file4.txt"`

.gitignore 忽略文件



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

仓库的根目录的配置文件 `.gitignore`，其中的每一行都代表一个需要忽略的文件或文件类型。

例如工作区中有两个文件分别是 `access.log` 和 `other.log` 文件。

`.gitignore` 中有且仅有一行 `access.log`

那么进行 `git add` 和 `git commit` 时，这个 `access.log` 都会被忽略掉。

一般我们在 `.gitignore` 中会使用 `*.log` 这个通配形式来忽略掉所有的 `log` 文件。

假设在配置 `.gitignore` 之前，就已经将会被忽略的文件放到暂存区和版本库中去了，设这个文件为 `other.log`。

此时我们在工作区中修改 `other.log`，再执行 `git status`，则仍然会被提醒这个文件被修改了，因为它已经在暂存区了，是个被跟踪的文件。

`.gitignore` 文件生效的一个前提是被 `ignore` 的文件不能已经存在于暂存区和版本库。

此时我们想到用 `git rm` 来删除这个文件，但注意这个命令会把工作区的 `other.log` 也删除掉，这是我们不想要的。

因此我们使用 `git rm --cached other.log` 来仅仅删除暂存区中的这个文件，再进行 `commit` 让版本库也删掉这个文件。

`.gitignore` 也可以配置文件夹。如 `.gitignore` 文件中仅有 2 行内容分别是

`*.log`

`temp/`

则所有 log 类型文件都被忽略，temp 文件夹下的所有内容也会被忽略。

```
1 git status -s
2
3 # git status -s 或
4 git status --short 是 Git 中的一个命令，它用来显示工作目录和暂存区的状态，但以一种更加简洁的形式呈现。
5
6 第一列字符（在左边）表示从最后一次提交到当前索引（即暂存区）的状态。（暂存区状态）
7 第二列字符（在右边）表示从索引（暂存区）到工作目录的状态。（工作区状态）
8
9 ?? ：新文件，尚未被 Git 跟踪。
10 A ：文件已被添加到暂存区。
11 M ：文件已被修改但未被暂存。
12 MM ：文件已修改，并且修改已暂存。
13 D ：文件已被删除但未从暂存区移除。
14 DD ：文件已删除，并且删除操作已暂存。
15 R ：文件已被重命名或移动，但更改尚未被暂存。
16 RM ：文件已被重命名或移动，并且更改已暂存。
```

.gitignore 文件的匹配规则

github 上提供了常用语言的忽略文件的模板

<https://github.com/github/gitignore>

<https://github.com/github/gitignore/blob/main/Java.gitignore>

注册 GitHub 账号

SSH 配置和克隆仓库

创建仓库

Top repositories

New

Find a repository...

 dundunEatnuts/git-testdemo

 dundunEatnuts/git-test

<https://github.com/new>

Home

[Send feedback](#)



immersive-translate/immersive-translate released

14 hours ago

v1.9.8

See [ChangeLog](#)

Note: 由于各商店审核时间不一致，所以对应的商店可能并不是最新版本，请耐心等待商店审核，或[手动下载安装包安装最新版](#)（大多数用户不推荐这样做）。

• [App Store for Safari](#)

• [Firefox Addon](#)

• [Chrome Store](#)

• [Edge Store](#)

• [Userscript](#)



 1

 2



immersive-translate/immersive-translate released

last week

v1.9.7

See [ChangeLog](#)

Note: 由于各商店审核时间不一致，所以对应的商店可能并不是最新版本，请耐心等待商店审核，或[手动下载安装包安装最新版](#)（大多数用户不推荐这样做）。


• [App Store for Safari](#)

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 *

 dundunEatnuts ▾

Repository name *



remote-repo

✔ remote-repo is available.

Great repository names are short and memorable. Need inspiration? How about **studious-octo-meme** ?

Description (optional)

My first repo.

- ☒  **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

HTTPS 和 SSH，这是远程仓库地址的两种方式

HTTPS 开头的方式在我们把本地代码 push 到远程仓库时，需要验证用户名和密码。

git 开头的这种方式使用 SSH 协议，这种方式在 push 时不需要验证用户名和密码。但是需要在 github 上添加 SSH 公钥的配置。这种是比较推荐的方式

在2021年8月13日以后，https 的方式已经被 github 停止使用。

下面我们尝试将这个远程仓库克隆到本地。

显示报错，要我们确认是否有正确的权限。

这是因为尚未配置 SSH 密钥。使用 SSH 方式必须配置 SSH 密钥。

如何配置？

1. 回到用户根目录。cd ~

2. cd .ssh

3. ssh-keygen -t rsa -b 4096

回车之后会提示输入密钥的文件名称，如果第一次使用这个命令则直接回车即可。它会在用户根目录的 .ssh 下生成一个 id_rsa 的私钥文件以及 id_rsa.pub 公钥文件。

如果不是第一次使用，那么最好设置别名否则会直接覆盖已经存在的 id_rsa 和 id_rsa.pub，并且操作不可逆。

这里我们已经生成了一个私钥公钥对 id_rsa 和 id_rsa.pub，没有为私钥设置密码。

4. 打开公钥复制其内容。

5. 回到 github，打开右上角头像弹出来的 settings，把公钥内容填入下面最后一张图的 key 里。

然后 Add SSH key 就可以了。如果是第一次创建密钥对，那么到这里就结束了，因为没有改密钥别名。

如果不是第一次创建密钥对，改了密钥别名为 test，那么最后还要在 .ssh 下创建一个 config 文件并添加下面5行

这个配置文件的意思是当我们访问 github.com 时，指定使用 .ssh/ 下的 test 这个密钥。

6. 再进行一次 git clone 这个命令。

cd 进这个 clone 下来的仓库发现已经出现括号main了，表示这个是一个 git 仓库

添加文件到这个本地仓库。echo hello > hello.txt

添加到暂存区。git add .

提交到本地仓库。git commit -m "first commit"

查看暂存区和仓库状态。git ls-files。看到本地仓库确实多了一个文件。但是打开远程仓库发现还是没有发生变化。

这个只是本地仓库的状态，只有使用 git push 才能将本地仓库的修改推送给远程仓库。

那么将本地仓库推送到远程仓库。git push

打开远程仓库页面刷新，发现确实已经被同步了。点击文件进去查看确实也更新了内容。

番外：ssh-keygen -t rsa -b 4096 以及公钥私钥的一些知识

`ssh-keygen -t rsa -b 4096` 是一个用于生成 SSH 密钥对的命令。这个命令的具体参数含义如下：

- `ssh-keygen`：这是用于创建、管理及转换认证密钥的工具，通常用于通过 SSH 协议进行安全登录。
- `-t rsa`：指定要创建的密钥类型为 RSA（Rivest–Shamir–Adleman）。RSA 是一种非对称加密算法，广泛用于互联网安全协议中，如 HTTPS 和 SSH。
- `-b 4096`：指定生成的 RSA 密钥的位数为 4096 位。位数越大，密钥越安全，但同时也意味着计算更复杂，可能会稍微影响性能。常见的选择有 2048 位和 4096 位。4096 位提供了更强的安全性，适用于需要更高安全级别的场景。

命令执行流程

当你运行 `ssh-keygen -t rsa -b 4096` 时，会发生以下步骤：

1. **生成密钥对**：系统会生成一对密钥——一个公钥和一个私钥。
2. **指定文件名**：系统会提示你输入保存密钥的文件路径，默认通常是 `~/.ssh/id_rsa` 对于私钥，以及 `~/.ssh/id_rsa.pub` 对于公钥。
3. **设置密码**：你可以选择给私钥设置一个密码（passphrase），这增加了额外的安全层，因为即使有人获得了你的私钥文件，他们也需要知道密码才能使用它。

示例

Sh浅色版本📄

```
$ ssh-keygen -t rsa -b 4096
Generating public/private rsa key pair.
Enter file in which to save the key (/home/yourusername/.ssh/id_rsa):
Created directory '/home/yourusername/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/yourusername/.ssh/id_rsa.
Your public key has been saved in /home/yourusername/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx your_username@hostname
The key's randomart image is:
+---[RSA 4096]---+
|                 |
|                 |
|                 |
|                 |
|                 |
|                 |
|                 |
|                 |
+-----[SHA256]-----+
```

在这个过程中，如果你不希望为私钥设置密码，可以直接按回车跳过该步骤。不过，出于安全考虑，建议设置一个强密码。生成的公钥可以被添加到远程服务器的 `~/.ssh/authorized_keys` 文件中，以实现免密码登录。

公钥和私钥是公钥加密系统（也称为非对称加密系统）中的两个重要组成部分。这种加密方式使用一对密钥来加密和解密数据，从而提供了一种安全的方式来交换信息。下面是公钥和私钥各自的作用及其工作原理：

公钥 (Public Key)

- **作用：**公钥是用来加密数据的钥匙。任何人都可以使用你的公钥来加密发送给你的消息。
- **特点：**公钥是可以公开分享的，不需要保密。你可以将它发布在网络上、通过电子邮件发送给他人或放在服务器上供其他人使用。

私钥 (Private Key)

- **作用：**私钥是用来解密数据的钥匙。只有拥有私钥的人才能解密用对应的公钥加密的数据。
- **特点：**私钥必须严格保密，不应与任何人共享。如果私钥泄露，那么任何拥有你私钥的人都能够冒充你进行通信或访问你的资源。

工作原理

1. **加密过程：**
 - 当有人想给你发送一条加密的消息时，他们会使用你的公钥来加密这条消息。
 - 加密后的消息只能用对应的私钥来解密。
2. **解密过程：**
 - 收到加密消息后，你使用自己的私钥来解密这条消息。
 - 只有持有正确私钥的人才能读取加密的内容。
3. **数字签名：**
 - 除了加密和解密，公钥和私钥还可以用于生成和验证数字签名。
 - 如果你想确保一条消息是你发送的，并且在传输过程中没有被篡改，你可以使用你的私钥对消息进行签名。
 - 接收者可以使用你的公钥来验证这个签名，以确认消息确实来自你并且未被修改。

应用场景

- **SSH 登录：**用户将自己的公钥添加到远程服务器的 `~/.ssh/authorized_keys` 文件中，这样就可以使用私钥进行身份验证而无需输入密码。
- **HTTPS：**网站使用 SSL/TLS 证书（其中包含公钥），客户端浏览器可以通过这些证书来建立一个安全连接。
- **电子邮件加密：**如 PGP 或 S/MIME，允许用户使用公钥加密邮件内容，只有接收者使用他们的私钥才能解密阅读。
- **软件分发：**开发者可以使用私钥签署发布的软件包，用户可以使用开发者的公钥来验证软件包的真实性和完整性。

总之，公钥和私钥共同构成了一个强大的安全机制，它们保证了数据的安全传输和身份验证，在许多网络安全协议和技术中都发挥着核心作用。

关联本地仓库和远程仓库

如何将本地仓库放到远程仓库？

1. 还是先要在 github 上创建一个新的仓库。


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 *

 dundunEatnuts ▾

Repository name *

first-repo

✔ first-repo is available.

Great repository names are short and memorable. Need inspiration? How about **cuddly-system** ?

Description (optional)

☒  **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

2. 复制 ssh 的 url

Quick setup — if you've done this kind of thing before

 Set up in Desktop or ☐ HTTPS ☒ SSH `git@github.com:dundunEatnuts/first-repo.git`



Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

3. 看下哪个本地仓库，现在要把 my-repo 这个本地仓库与 github 的这个 first-repo 进行关联。

```
贝爷HelloWorld@DESKTOP-3UBM31M MINGW64 /d/developer_tools/Git/workbench/my-repo (master)
$ pwd
/d/developer_tools/Git/workbench/my-repo

贝爷HelloWorld@DESKTOP-3UBM31M MINGW64 /d/developer_tools/Git/workbench/my-repo (master)
$
```

添加一个远程仓库 `git remote add <shortname> <url>`

```
1 # git remote add <shortname> <url>
2 git remote add origin git@github.com:dundunEatnuts/first-repo.git
3
4 # origin 是默认的别名，一般都是这个。
```

执行这个。 **git remote add origin git@github.com:dundunEatnuts/first-repo.git**

执行之后使用 `git remote -v` 查看

```
1 $ git remote -v
2 origin  git@github.com:dundunEatnuts/first-repo.git (fetch)
3 origin  git@github.com:dundunEatnuts/first-repo.git (push)
4
5 # 返回当前仓库锁对应的远程仓库的别名和地址。
6 # 上面返回的结果中，别名是 origin，后面就是所对应的远程仓库的地址。
```

然后指定本地仓库当前分支的名称为 main。 **git branch -M main**

最后是把本地仓库的 main 分支和远程的 origin 仓库的 main 分支关联起来。 **git push -u origin main**

```
1 git push -u origin main
2
3 # -u 是 upstream
4 # 命令全称应该为 git push -u origin main:main
5 # 把本地仓库和别名为 origin 的远程仓库关联起来，main:main --> 本地仓库的main分支:远程仓库的main分支
6 # 而如果本地分支的名称与远程分支的名称相同的话，那么就只用写一个 main 即可。
```

执行 `git push -u origin main`

页面上刷新就发现已经成功了。

模拟远程仓库发生变化再通过 `pull` 拉到本地。

现在从github frontend add 一个 readme 文件

查看本地仓库是还没有这个文件的。

使用 pull 命令拉取远程仓库的修改内容。git pull <远程仓库> <远程分支名>:<本地分支名>
如果省略远程仓库名称和远程分支名称，则默认为 origin 仓库的 main 分支。
作用是拉取远程仓库的指定分支到本地再进行合并。

关于 git pull 的注意：

1. 执行 git pull 后，git 会自动执行1次合并操作。如果远程仓库的修改内容跟本地仓库中的修改内容没有冲突，那么merge会自动成功。如果有冲突则会 merge 失败，就会被要求手动 merge 来解决冲突。

从远程仓库获取内容也可使用 fetch 命令 git fetch。

区别在于 fetch 命令只是获取远程仓库的修改内容，而不会自动 merge，必须手动 merge。

分支简介和基本操作

查看当前仓库所有分支。git branch

我们创建了三个文件 main1.txt main2.txt main3.txt，创建完1个就 add 和 commit 1次，最后一共是3次 commit。

```
贝壳HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (master)
$ git log --oneline
f0e2254 (HEAD -> master) main:3
8f6ee4d main:2
5d1629e main:1

贝壳HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (master)
$ git branch
* master

贝壳HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (master)
$ |
```

使用 git branch 可以查看到，带有 * 的就是当前所处分支。

创建一个新的分支。git branch dev

```

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (master)
$ git branch dev

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (master)
$ ll -rt
total 3
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:11 main1.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:12 main2.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:13 main3.txt

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (master)
$ git branch
  dev
* master

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (master)
$ |

```

可以看到分支 dev 已经创建好了。并且当前所处分支还是 master。

git branch dev 只是创建了这个分支，并没有将当前分支切换过去。

那么我们要切换到这个新建的 dev 分支上。git checkout <name>，即 git checkout dev 去切换到这个分支上。

```

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (master)
$ git checkout dev
Switched to branch 'dev'

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (dev)
$ git branch
* dev
  master

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (dev)
$

```

使用 git branch -m <new-branch-name> 来将当前分支的命名改为新的。

```

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (master)
$ git branch -m main

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (main)
$

```

****注意

git checkout 命令即接收分支名，也可以接收目录名和文件名。

所以当分支名和目录名或者文件名重复时，将会发生歧义，而默认将以其为分支进行处理。

所以 git 提供了 switch 命令来仅为切换分支服务。

所以切换分支也可以使用 git switch <branchname>

```

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (main)
$ git switch dev
Switched to branch 'dev'

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (dev)
$ |

```

进到 dev 分支。

echo dev1 > dev1.txt

git add .

git commit -m "dev:1"

echo dev2 > dev2.txt

git add .

git commit -m "dev:2"

到这里来回切换一下 main 和 dev，观察工作区文件状态

```
贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (dev)
$ ll -rt
total 5
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:11 main1.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:12 main2.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:13 main3.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:32 dev1.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 5 Sep 29 13:33 dev2.txt

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (dev)
$ git ls-files
dev1.txt
dev2.txt
main1.txt
main2.txt
main3.txt

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (dev)
$ |
```

```
贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (dev)
$ git switch main
Switched to branch 'main'

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (main)
$ ll -rt
total 3
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:11 main1.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:12 main2.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:13 main3.txt

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (main)
$ git ls-files
main1.txt
main2.txt
main3.txt

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (main)
$
```

可以看到 main 和 dev 的工作区中的文件确实是有区别的。

现在再去 main 中 commit 两个变化。

git switch main

echo main4 > main4.txt

git add .

git commit -m "main:4"

echo main5 > main5.txt

git add .

git commit -m "main:5"

再去观察 main 和 dev 的工作区，返现也确实有区别。

```
贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (main)
$ ll -rt
total 5
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:11 main1.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:12 main2.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:13 main3.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 7 Sep 29 13:38 main4.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 7 Sep 29 13:38 main5.txt

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (main)
$
```

```
贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (dev)
$ ll -rt
total 5
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:11 main1.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:12 main2.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:13 main3.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:37 dev1.txt
-rw-r--r-- 1 贝爷HelloWorld 197121 6 Sep 29 13:37 dev2.txt
```

那么现在我们要将 dev 分支合并到 main 分支。使用 `git merge <branchname>`

```
1 ~/branch-demo (main)$ git merge dev
2
3 # 上方意为，将 dev 分支合并到当前的 main 分支。
4 # 必须先切换到 main 分支，然后再使用该命令对 dev 分支进行合并。
```

git merge dev 执行后，git 会自动为我们产生一次提交，我们要输入提交的消息。commit 的 msg。

直接使用默认的提交信息，那么直接 wq 保存即可。

使用命令查看分支图。**git log --graph --oneline --decorate --all**

将 dev 合并到 main 之后，这个 dev 的分支还是存在的，并不会直接被删除。git branch

如何手动删除分支？

如果一个分支已经被用来 merge 完了不再被需要了，那么使用下方命令删除

`git branch -d dev`

这是删除已经完成合并的分支，如果分支还没有完成合并，则无法删除。要想强制删除则使用下方

`git branch -D dev`

解决合并冲突

小技巧：一个命令完成 add 和 commit 的操作。git commit -am "msg"

下面制造一个冲突。

1. 创建一个feat分支。修改其中的main1.txt，并且 commit

```
git branch feat
```

```
vi main1.txt
```

```
git commit -am "feat:1"
```

2. 切换回 main 分支，修改 main1.txt，并且 commit

```
git switch main
```

```
vi main1.txt
```

```
git commit -am "main:6"
```

3. 在 main 分支执行 merge 指令 merge feat，然后失败了

```
贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (main)
$ git merge feat
Auto-merging main1.txt
CONFLICT (content): Merge conflict in main1.txt
Automatic merge failed; fix conflicts and then commit the result.

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (main|MERGING)
$ |
```

需要解决冲突后提交。

git status 查看冲突文件列表

```
贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (main|MERGING)
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   main1.txt

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

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (main|MERGING)
$
```

git diff 查看冲突的具体内容。

```
贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (main|MERGING)
$ git diff
diff --cc main1.txt
index 6a86e1c,af12842..0000000
--- a/main1.txt
+++ b/main1.txt
@@@ -1,2 -1,2 +1,6 @@@
    main1
++<<<<<< HEAD
+learn git after OldY
++=====
+ this change is from feat branch
++>>>>>> feat

贝爷HelloWorld@DESKTOP-3UBMJ1M MINGW64 /d/developer_tools/Git/workbench/branch-demo (main|MERGING)
$
```

现在我们要解决冲突，则需要手工编辑这个文件留下我们想要的内容，然后再次 commit

vi main1.txt

修改后

然后提交

如果提交之前想要中断这次合并，则使用命令。 `git merge --abort`

回退和 rebase

如果现在有两个分支，分别是 main 和 dev。

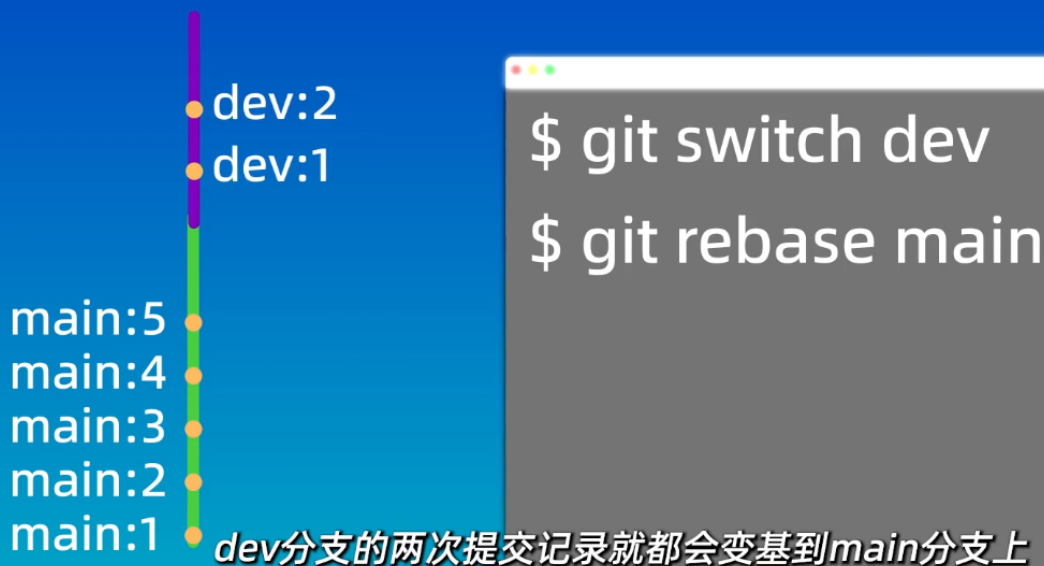
如果先 switch 到 dev 上，然后执行 git rebase main，那么 dev 分支将会被直接拼接上 main 分支上。

如下



rebase 后 dev 分支的两次 commit 记录会被变基到 main 分支上。

Rebase



但是如果先 switch 到 main 分支上，在执行 git rebase dev，那么 main 分支将会被拼接到 dev 分支上。



那么 git rebase branch-name 的意思就是将我当前所处的分支拼接到目标分支 branch-name 上。

Rebase和Merge有什么区别 该如何区分使用？

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。

分支管理和工作流模型