# Git 入门

**git和github是两个完全不同的概念。**git是一个版本管理工具，是可以在你电脑不联网的情况下，只在本地使用的一个版本管理工具，其作用就是可以让你更好的管理你的程序，比如你原来提交过的内容，以后虽然修改了，但是通过git这个工具，可以把你原来提交的内容重现出来，这样对于你后来才意识到的一些错误的更改，可以进行还原。github是一个网站，就是每个程序员自己写的程序，可以在github上建立一个网上的仓库，你每次提交的时候可以把代码提交到网上，这样你的每次提交，别人也都可以看到你的代码，同时别人也可以帮你修改你的代码，这种开源的方式非常方便程序员之间的交流和学习。

github是一个非常适合程序员交流的网站，很多国际上的技术大牛都在github上有自己的开源代码，其他人只要申请个账号就可以随意的看到这些大牛写的程序。同时国内的很多互联网公司如百度，阿里等，也在github上公布有开源的代码，感兴趣的程序员也可以自己查找着看一些。想看国内大公司的代码，可以看连接：https://www.zhihu.com/question/20194937总的来说，git可以认为是一个软件，能够帮你更好的写程序，github则是一个网站，这个网站可以帮助程序员之间互相交流和学习。

## Git 简介

**什么是“版本控制”？**版本控制是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。在 CODE CHINA 中，我们对保存着软件源代码的文件作版本控制，但实际上，你可以对任何类型的文件进行版本控制。

如果你是位图形或网页设计师，可能会需要保存某一幅图片或页面布局文件的所有修订版本（这或许是你非常渴望拥有的功能），采用版本控制系统（VCS）是个明智的选择。 有了它你就可以将选定的文件回溯到之前的状态，甚至将整个项目都回退到过去某个时间点的状态，你可以比较文件的变化细节，查出最后是谁修改了哪个地方，从而找出导致怪异问题出现的原因，又是谁在何时报告了某个功能缺陷等等。 使用版本控制系统通常还意味着，就算你乱来一气把整个项目中的文件改的改删的删，你也照样可以轻松恢复到原先的样子。 但额外增加的工作量却微乎其微。

**版本控制系统发展可以分为三个阶段：①本地版本控制系统、②集中式版本控制系统**

**、③分布式版本控制系统。**

**本地版本控制系统**

许多人习惯用复制整个项目目录的方式来保存不同的版本，或许还会改名加上备份时间以示区别。 这么做唯一的好处就是简单，但是特别容易犯错。 有时候会混淆所在的工作目录，一不小心会写错文件或者覆盖意想外的文件。

为了解决这个问题，人们很久以前就开发了许多种本地版本控制系统，大多都是采用某种简单的数据库来记录文件的历次更新差异。

其中最流行的一种叫做 RCS ，现今许多计算机系统上都还看得到它的踪影。 RCS 的工作原理是在硬盘上保存补丁集（补丁是指文件修订前后的变化）；通过应用所有的补丁，可以重新计算出各个版本的文件内容。

**集中化的版本控制系统**

接下来人们又遇到一个问题，如何让在不同系统上的开发者协同工作？ 于是，集中化的版本控制系统（CVCS）应运而生。 这类系统，诸如 CVS、Subversion 以及 Perforce 等，都有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。多年以来，这已成为版本控制系统的标准做法。

这种做法带来了许多**好处**，特别是相较于老式的本地 VCS 来说。 现在，每个人都可以在一定程度上看到项目中的其他人正在做些什么。 而管理员也可以轻松掌控每个开发者的权限，并且管理一个 CVCS 要远比在各个客户端上维护本地数据库来得轻松容易。

但这么做也有一个显而易见的**缺点**，那就是是中央服务器的单点故障。如果宕机一小时，那么在这一小时内，谁都无法提交更新，也就无法协同工作。如果中心数据库所在的磁盘发生损坏，又没有做恰当备份，毫无疑问你将丢失所有数据——包括项目的整个变更历史，只剩下人们在各自机器上保留的单独数据。本地版本控制系统也存在类似问题，只要整个项目的历史记录被保存在单一位置，就有丢失所有历史更新记录的风险。

**分布式版本控制系统**

于是分布式版本控制系统（DVCS）面世了。 在这类系统中，像 Git、Mercurial、Bazaar 以及 Darcs 等，客户端并不只提取最新版本的文件数据， 而是把代码仓库完整地镜像下来，包括完整的历史记录。这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。 因为每一次的克隆操作，实际上都是一次对代码仓库的完整备份。

不仅如此，许多这类系统都可以指定和若干不同的远端代码仓库进行交互。这样一来，你就可以在同一个项目中，分别和不同工作小组的人相互协作。 你可以根据需要设定不同的协作流程，比如层次模型式的工作流，而这在以前的集中式系统中是无法实现的。

**Git 诞生的背景**

同生活中的许多伟大事物一样，Git 诞生于一个极富纷争大举创新的年代。Linus 在1991年创建了开源的 Linux，从此，Linux 系统不断发展，已经成为最大的服务器系统软件了。在1991－2002年期间，世界各地的志愿者把源代码文件通过 diff 的方式发给 Linus，然后由 Linus 本人通过手工方式合并代码。

你也许会想，为什么 Linus 不把 Linux 代码放到版本控制系统里呢？不是有CVS、SVN这些免费的版本控制系统吗？因为Linus 坚定地反对CVS和SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比CVS、SVN好用，但那是付费的，和 Linux 的开源精神不符。

**Linus 两周完成 Git**

到 2002 年，Linux 系统已经发展了十年了，代码库之大让 Linus 很难继续通过手工方式管理了，整个项目组开始启用一个专有的分布式版本控制系统 BitKeeper 来管理和维护代码，BitKeeper 的东家 BitMover 公司也免费授权 Linux 社区使用这个版本控制系统。后来 BitMover 公司发现社区有人试图破解 BitKeeper 的协议，于是 BitMover 公司收了回 Linux 社区的免费使用权。

这就迫使 Linux 开源社区（特别是 Linux 的缔造者 Linus Torvalds）基于使用 BitKeeper 时的经验教训，开发出自己的版本系统。 他们对新的系统制订了若干目标：速度、简单的设计、对非线性开发模式的强力支持（允许成千上万个并行开发的分支）、完全分布式、有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）。于是，Linus 花了两周时间自己用 C 写了一个分布式版本控制系统，这就是 Git！一个月之内，Linux 系统的源码已经由 Git 管理了！

**Git 的发展壮大**

自 2005 年诞生以来，Git 日臻成熟完善，在高度易用的同时，仍然保留着初期设定的目标。它的速度飞快，极其适合管理大项目，有着令人难以置信的非线性分支管理系统。Git 迅速成为最流行的分布式版本控制系统，尤其是 2008 年，GitHub 网站上线了，它为开源项目免费提供 Git 存储，无数开源项目开始迁移至 GitHub ，包括jQuery，PHP，Ruby等等。

## Git安装

**在 Windows 上安装 Git 的方法如下**：

方式一使用官方版本安装，官方版本可以在 Git 官方网站下载。打开 https://git-scm.com/download/win，然后选择相应的版本即可。下载完成后，按照提示进行安装即可。

方式二Chocolatey 自动安装，如果要进行自动安装，你可以使用 Git Chocolatey 包。 注意 Chocolatey 包是由社区维护的。Chocolatey 官网：<https://chocolatey.org>。安装好 Chocolatey 后，执行如下命令即可：choco install git.install

**在 macOS 上安装Git** 有多种方式。 最简单的方法是安装 Xcode Command Line Tools。 Mavericks （10.9） 或更高版本的系统中，在 Terminal 里尝试首次运行 git 命令即可。例如：$ git –version 如果没有安装过命令行开发者工具，将会提示你安装。如果你想安装更新的版本，可以使用二进制安装程序。 官方维护的 macOS Git 安装程序可以在 Git 官方网站下载，网址为 https://git-scm.com/download/mac。

**在 Linux 上安装**：在 Linux 上用二进制安装程序来安装基本的 Git 工具，可以使用发行版包含的基础软件包管理工具来安装。 以 Fedora 为例，如果你在使用它（或与之紧密相关的基于 RPM 的发行版，如 RHEL 或 CentOS），你可以使用 dnf：$ sudo dnf install git-all。如果你在基于 Debian 的发行版上，如 Ubuntu，请使用 apt：$ sudo apt install git-all要了解更多选择，Git 官方网站上有在各种 Unix 发行版的系统上安装步骤，网址为 https://git-scm.com/download/linux。

**Git环境配置**

好了，当你当完成了 Git 的安装后，接下来我们就需要对 Git 进行一些必要的环境配置。通常情况下，每台计算机上只需要配置一次 Git，当 Git 程序升级时会保留配置信息。 你可以在任何时候再次通过运行 git config命令来修改它们。

Git 自带一个 **git config** 的工具来帮助设置控制 Git 外观和行为的配置变量。接下来，我们先学习如何通过 git config 命令来配置用户信息。

**配置用户名和邮件地址**

安装完 Git 之后，要做的第一件事就是设置你的用户名和邮件地址。 这一点很重要，因为每一个 Git 提交都会使用这些信息，它们会写入到你的每一次提交中，不可更改：

$ git config --global user.name "李老师"

$ git config --global user.email li@csdn.net

**再次强调**，如果使用了 **--global** 选项，那么该命令只需要运行一次，因为之后无论你在该系统上做任何事情， Git 都会使用那些信息。当你想针对特定项目使用不同的用户名称与邮件地址时，可以在那个项目目录下运行没有 --global 选项的命令来配置。通俗的讲—global是个全局变量标志，仅可设置一次之后默认用当前设置，如果某个子项目不是当前默认设置则用不带—global的命令重新设置一次。

**检查配置**

如果想要检查你的配置，可以使用 git config --list 命令来列出所有 Git 当时能找到的配置。例如：

$ git config --list

user.name= "李老师"

user.email=li@csdn.net

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

你可能会看到重复的变量名，因为 Git 会从不同的文件中读取同一个配置（例如：/etc/gitconfig 与 ~/.gitconfig）。 这种情况下，Git 会使用它找到的每一个变量的最后一个配置。

你可以通过输入 git config <key>： 来检查 Git 的某一项配置，例如：

$ git config user.name

李老师

**更多Git 配置**

既然已经在系统上安装了 Git，你会想要做几件事来定制你的 Git 环境。 每台计算机上只需要配置一次，程序升级时会保留配置信息。 你可以在任何时候再次通过运行命令来修改它们。Git 自带一个 git config 的工具来帮助设置控制 Git 外观和行为的配置变量。 这些变量存储在**三个不同的位置**：①/etc/gitconfig 文件: 包含系统上每一个用户及他们仓库的通用配置。 如果在执行 git config 时带上 --system选项，那么它就会读写该文件中的配置变量。（由于它是系统配置文件，因此你需要管理员或超级用户权限来修改它）②~/.gitconfig 或 ~/.config/git/config 文件：只针对当前用户。 你可以传递 --global 选项让 Git 读写此文件，这会对你系统上所有的仓库生效。③当前使用仓库的 Git 目录中的 config 文件（即 .git/config``）：针对该仓库。你可以传递--local 选项让 Git 强制读写此文件，虽然默认情况下用的就是它。（当然，你需要进入某个 Git 仓库中才能让该选项生效）

**每一个级别会覆盖上一级别的配置，即优先级：当前仓库>全部仓库>系统配置**，所以 .git/config 的配置变量会覆盖 /etc/gitconfig 中的配置变量。

在 Windows 系统中，Git 会查找 $HOME 目录下的 .gitconfig 文件（一般情况下是C:\Users\$USER）。Git 同样也会寻找 /etc/gitconfig 文件，但只限于msys的根目录下，即安装Git时所选的目标位置。 如果你在 Windows 上使用Git 2.x以后的版本，那么还有一个系统级的配置文件，Windows XP上在 C:\Documents and Settings\All Users\Application Data\Git\config，Windows Vista及更新的版本在 C:\ProgramData\Git\config。此文件只能以管理员权限通过 git config -f <file> 来修改。

你可以通过以下命令查看所有的配置以及它们所在的文件：$ git config --list --show-origin。

**Git 颜色配置**

到目前为止，我们已经配置了user.name和 user.email，实际上Git 还有很多可配置项。比如，让 Git 显示颜色，会让命令输出看起来更醒目：$ git config --global color.ui true。这样，Git 会适当地显示不同的颜色，比如使用`git status`命令后文件名就会标上颜色。

**Git忽略文件配置**

有些时候，你必须把某些文件放到 Git 工作目录中，但又不能提交它们，比如保存了数据库密码的配置文件等等，每次git status都会显示Untracked files ...，这种情况下，就可以使用忽略特殊文件.gitignore来解决这个问题。

首先我们在Git工作区的根目录下创建一个特殊的.gitignore文件，然后把要忽略的文件名填进去，Git 在每次进行提交的时候就会自动忽略这些文件。

**忽略文件的规则**

日常使用中，我们一般不需要从头开始编辑.gitignore文件，已经有各种现成的种配置文件，只需要组合一下就可以使用了。所有配置文件可以直接在线浏览：https://codechina.csdn.net/codechina/gitignore

**忽略文件的原则是**：①忽略操作系统自动生成的文件，比如缩略图等；②忽略编译生成的中间文件、可执行文件等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如Java编译产生的.class文件；③忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。

让我们来看一个例子：假设你在 Windows 下进行 Python 开发，Windows 会自动在有图片的目录下生成隐藏的缩略图文件，如果有自定义目录，目录下就会有Desktop.ini文件，因此你需要忽略 Windows 自动生成的垃圾文件，就要在.gitignore 文件中设置：

# Windows:

Thumbs.db

ehthumbs.db

Desktop.ini

然后，继续忽略Python编译产生的.pyc、.pyo、dist等文件或目录，就要在.gitignore 文件中设置：

# Python:

\*.py[cod]

\*.so

\*.egg

\*.egg-info

dist

build

加上你自己定义的文件（如下），最终得到一个完整的.gitignore文件（三部分加起来），内容如下：

# My configurations:

db.ini

deploy\_key\_rsa

最后一步就是把.gitignore文件也提交到Git就完成了！当然检验.gitignore的标准是git status命令会不会提示working directory clean。

**强制添加被忽略文件**

有些时候，你想添加一个文件到 Git，但发现添加不了，原因是这个文件被.gitignore忽略了。如果你确实想添加该文件，可以用-f强制添加到 Git：$ git add -f App.class

**检查忽略规则**

文件添加不上也可能是.gitignore写得有问题，需要找出来到底哪个规则写错了，可以用git check-ignore命令检查：

$ git check-ignore -v App.class

.gitignore:3:\*.class App.class

如上所示，Git会告诉我们.gitignore的第3行规则忽略了该文件，于是我们就可以知道应该修订哪个规则。

**添加例外规则**

还有些时候，当我们用正则表达式编写了规则，排除了某一类型的文件，例如：

.\* # 排除所有.开头的隐藏文件

\*.class # 排除所有.class文件

我们发现.\*这个规则把.gitignore也排除了，并且App.class需要被添加到版本库，但是被\*.class规则排除了。这个时候，虽然可以用git add -f强制添加进去，但我们也可以添加两条例外规则：**把指定文件排除在.gitignore规则外的写法就是!+文件名，所以只需把例外文件添加进去即可**。

.\* # 排除所有.开头的隐藏文件

\*.class # 排除所有.class文件

!.gitignore # 不排除.gitignore和App.class

!App.class

**Git 配置别名**

除了通过 配置忽略文件来提高git commit 时的便捷性外，Git中还有一种可以让大家在敲入Git命令时偷懒的办法——那就是配置Git别名。

（1）配置 git status/commit/checkout/branch

比如在使用git status命令时，我们可以通过配置别名的方式将其配置为git st，这样在使用时是不是就比输入 git status简单方便很多呢？我们只需要敲一行命令，告诉 Git以后st就表示status：$ git config --global alias.st status当然还有别的命令可以简写，很多人都用co表示checkout，ci表示commit，br表示branch，如下所示：

$ git config --global alias.co checkout

$ git config --global alias.ci commit

$ git config --global alias.br branch

配置完成以上别名后，以后提交就可以简写成：$ git ci -m "sth."

（2）配置 git reset HEAD file

再比如git reset HEAD file命令，他可以把暂存区的修改撤销掉（unstage），重新放回工作区。既然是一个unstage操作，就可以配置一个unstage别名：$ git config --global alias.unstage 'reset HEAD'当你敲入命令$ git unstage test.py，实际上 Git 执行的是$ git reset HEAD test.py。

（3）配置 git log -1

配置一个git last，让其显示最后一次提交信息：$ git config --global alias.last 'log -1'

这样，用git last就能显示最近一次的提交：

$ git last

commit 4aac6c7ee018f24d2dabfd01a1e09c77612e1a4e (HEAD -> master)

Author: Miykael\_xxm <xiongjiamu@gmail.com>

Date: Tue Nov 17 11:14:15 2020 +0800

branch test

（4）配置 git lg

git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"

**Git 配置文件**

**全局配置文件**

这些自定义的Git配置文件通常都存放在仓库的.git/config文件中。查看命令：$ cat .git/config，别名就在[alias]后面，要删除别名直接把对应的行删掉即可。

**用户配置文件**

而当前用户的 Git 配置文件放在用户主目录下的一个隐藏文件.gitconfig中。查看命令$ cat .gitconfig，配置别名也可以直接修改这个文件，如果改错了，可以删掉文件重新通过命令配置。

## Git使用

在开始 Git 的基础命令学习之前，我们先来认识一下版本库——Repository，接下来我们所有提到的 Git 基础命令，都是基于版本库的。那么什么是版本库呢？版本库又名仓库，英文名 repository，你可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。接下来，我们首先看看 Git 版本库创建的相关命令。

**在已存在目录中初始化仓库 —— git init**

在已存在目录中创建一个版本库的过程非常简单：

第一步，选择一个合适的地方，创建一个空目录：用mkdir创建文件夹\cd进入文件夹\pwd显示当前文件夹绝对路径。

第二步，通过git init命令把这个目录变成 Git 可以管理的仓库：

$ git init

Initialized empty Git repository in /Documents/CS\_learn/Git/Store\_Share/Ls1b

瞬间 Git 就把仓库建好了，而且告诉你是一个空的仓库（empty Git repository），同时在当前目录下多了一个.git的目录，这个目录是 Git 来跟踪管理版本库的，如果你没有看到 .git 目录，那是因为这个目录默认是隐藏的，用ls -ah命令就可以看到了。

**克隆现有的仓库 —— git clone**

如果你想拷贝一份已经存在了的 Git 仓库，比如你想为某个开源项目贡献自己的一份力，这时就要用到 git clone 命令，Git 克隆的是该仓库服务器上的几乎所有数据，而不是仅仅复制完成你的工作所需要文件。当你执行 git clone 命令的时候，默认配置下远程 Git 仓库中的每一个文件的每一个版本都将被拉取下来。

克隆仓库的命令是 git clone <url> 。 比如用下面的命令克隆龙芯代码仓库：git clone https://github.com/litianling/Ls1b这会在当前目录下创建一个名为 Ls1b 的目录，并在这个目录下初始化一个 .git 文件夹， 从远程仓库拉取下所有数据放入 .git 文件夹，然后从中读取最新版本的文件的拷贝。 如果你进入到这个新建的Ls1b文件夹，你会发现所有的项目文件已经在里面了，准备就绪等待后续的开发和使用。

自定义本地仓库名称：当然如果你想在克隆远程仓库的时候，自定义本地仓库的名字也是可以的，你可以通过额外的参数指定新的目录名：git clone https://github.com/litianling/Ls1b mydocs这会执行与上一条命令相同的操作，但目标目录名变为了 mydocs。

Git 支持多种数据传输协议。 上面的例子使用的是 https://协议，不过你也可以使用 git:// 协议或者使用SSH传输协议，比如user@server:path/to/repo.git。

**编辑并添加文件**

首先，我们来尝试在已经准备好的Git仓库中编辑一个readme.txt文件，用gedit readme.txt即可。

接下来，我们可以通过2个命令将刚创建好的readme.txt添加到Git仓库：

第一步，用命令git add readme.txt告诉 Git，把变动文件添加到仓库缓存。

第二步，用命令git commit -m "wrote a readme file"告诉 Git，把文件提交到仓库。

这里简单解释一下git commit命令，-m后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样你就能从历史记录里方便地找到改动记录。

为什么 Git 添加文件需要add，commit一共两步呢？因为commit可以一次提交很多文件，所以你可以多次add不同的文件，比如：

$ git add file1.txt

$ git add file2.txt file3.txt

$ git commit -m "add 3 files."

**查看Git仓库当前状态变化**

**git status**命令可以让我们时刻掌握仓库当前的状态，还会提示我们下一步操作。

**比较变动**

虽然git status会告诉我们readme.txt被修改了，但并没有告诉我们具体修改的内容是什么，假如刚好是上周修改的，等到周一来班时，已经记不清上次怎么修改的readme.txt，这个时候我们就需要用git diff这个命令查看相较于上一次暂存都修改了些什么内容了：

$ git diff readme.txt

diff --git a/readme.txt b/readme.txt

index 46d49bf..9247db6 100644

--- a/readme.txt

+++ b/readme.txt

@@ -1,2 +1,2 @@

-Git is a version control system.

+Git is a distributed version control system.

Git is free software.

(END)

git diff顾名思义就是查看 difference，显示的格式正是 Unix 通用的 diff 格式，可以从上面的输出看到具体修改。

**综合操作**

知道了对readme.txt作了什么修改后，再把它提交到仓库就放心多了，提交修改和提交新文件是一样的两步，第一步git add readme.txt同样没有任何输出。在执行第二步git commit之前，我们再运行git status看看当前仓库的状态：git status告诉我们，将要被提交的修改包括readme.txt，下一步就可以放心地提交了：git commit -m "add distributed"，提交后，我们再用git status命令看看仓库的当前状态：Git告诉我们当前没有需要提交的修改，而且，工作目录是干净（working tree clean）的。

**查看日志**

在 Git 中，我们可以通过**git log**命令查看全部的commit记录，git log命令显示从最近到最开始的所有提交日志。

如果嫌输出信息太多，看得眼花缭乱的，可以试试加上--pretty=oneline参数：git log --pretty=oneline。每提交一个新版本，实际上 Git 就会把它们自动串成一条时间线。如果使用可视化工具或者之前在git自定义配置中介绍的git lg命令（在安装中重命名了），就可以更清楚地看到提交历史的时间线。

作为一个优秀的版本控制系统，Git 能够让我们查看每一次提交的记录。在日常的工作中，我们可以随时对 Git 仓库中的内容进行修改，每当你觉得文件修改到一定程度的时候，就可以“保存一个快照”，这个快照在 Git中 被称为commit/提交。一旦你把文件改乱了，或者误删了文件，还可以从最近的一个commit恢复，然后继续工作，而不是把几个月的工作成果全部丢失。

**Git 回退**

这个时候，假设我们需要将 readme.txt 回退到上一个版本，我们需要怎么操作呢？首先，Git 必须知道当前版本是哪个版本，在 Git 中用HEAD表示当前版本也就是最新提交的，上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

现在，我们要把当前版本回退到上一个版本，就可以使用**git reset**命令：git reset --hard HEAD^，注意Git 的版本回退速度非常快，因为 Git 在内部有个指向当前版本的HEAD指针，当你回退版本的时候，Git 仅仅是把HEAD从指向对应版本。

\*\* HEAD 指针移动记录\*\*

┌────┐

│HEAD │

└────┘

│

└──> ○ add distributed

│

○ wrote a readme file

改为指向wrote a readme file：

┌────┐

│HEAD │

└────┘

│

│ ○ add distributed

│ │

└──> ○ wrote a readme file

然后顺便把工作区的文件更新了。所以你让HEAD指向哪个版本号，你就把当前版本定位在哪。

**Git重置**

现在，你回退到了某个版本，关掉了电脑，第二天早上就后悔了，想恢复到新版本怎么办？找不到新版本的commit id怎么办？

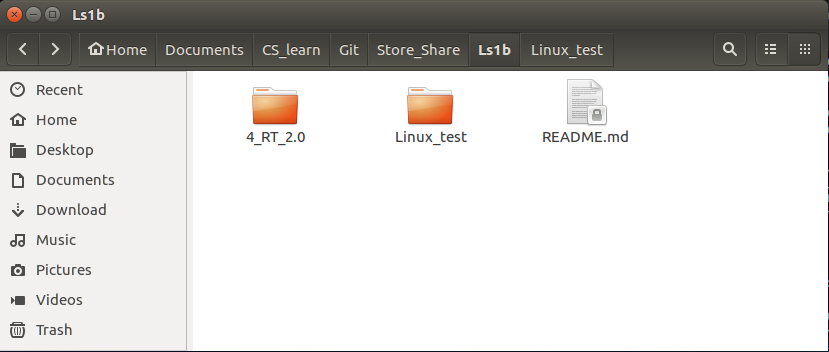
好在 Git 提供了一个命令git reflog用来记录你的每一次命令，当你用git reset --hard HEAD^回退到上一版本时，再想恢复到新版本，就可以通过git reflog命令找到新版本的commit id。

从输出可以看到，新版本的commit id是e55063a，现在，我们就可以通过 git reset --hard e55063a切换到最新的版本上了。

**工作区和暂存区**

Git 和其他版本控制系统如 SVN 的一个不同之处就是有暂存区的概念。

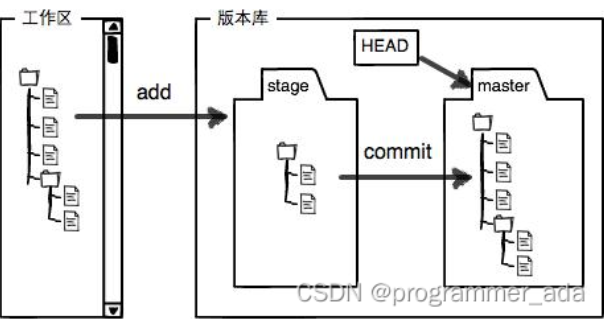
工作区（Working Directory）:就是你在电脑里能看到的目录，比如我的Ls1b文件夹就是一个工作区：



版本库（Repository）:工作区有一个隐藏目录.git，这个不算工作区，而是 Git 的版本库。Windos系统下可以直接看到版本库，linux系统必须用ls -ah命令才能看到隐藏的版本库。



Git 的版本库里存了很多东西，其中最重要的就是称为stage（或者叫index）的暂存区，还有 Git 为我们自动创建的第一个分支main，以及指向main的一个指针叫HEAD。



**main**

**Index**

上图中stage改名为index，master改名为main





前面讲了我们把文件往 Git 版本库里添加的时候，是分两步执行的：第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支。因为我们创建 Git 版本库时，Git自动为我们创建了唯一一个main分支，所以，现在，git commit就是往 master 分支上提交更改。你可以简单理解为，需要提交的文件修改都放到暂存区，然后一次性将暂存区的所有修改都提交到当前分支。

现在，我们来试一下，先对readme.txt做个修改，并在工作区新增一个LICENSE文本文件。先用git status查看一下状态：

$ git status

On branch main

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: readme.txt

Untracked files:

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

LICENSE

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

Git非常清楚地告诉我们，readme.txt被修改了，而LICENSE还从来没有被添加过，所以它的状态是Untracked。然后使用两次命令git add，把readme.txt和LICENSE都添加后，用git status再查看一下：

$ git status

On branch main

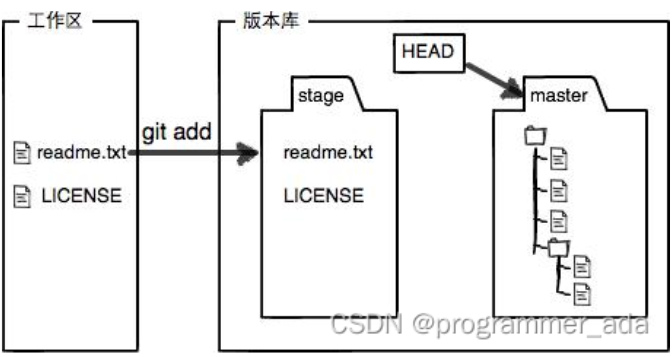
Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: LICENSE

modified: readme.txt

现在，暂存区的状态就变成这样了：



**main**

**Index**

所以，git add命令实际上就是把要提交的所有修改放到暂存区（Index），然后，执行git commit就可以一次性把暂存区的所有修改提交到分支。

$ git commit -m "understand how stage works"

[master 599dbdb] understand how stage works

2 files changed, 2 insertions(+)

create mode 100644 LICENSE

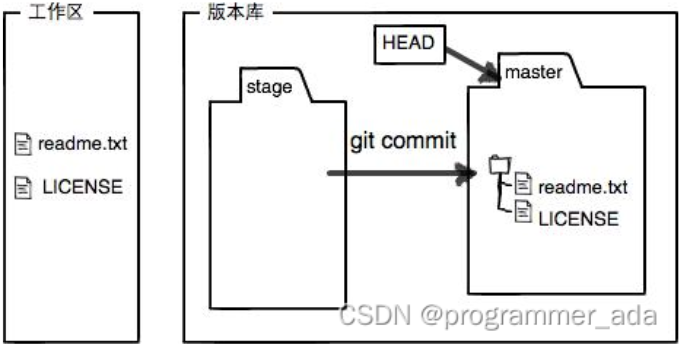
一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的：

$ git status

On branch main

nothing to commit, working tree clean

现在版本库变成了这样，暂存区就没有任何内容了：



**main**

**Index**

**小结**

**git init、git clone、git add、git commit、git status、git diff、git log、git reset、git reflog**

## 版本管理

**撤消对文件的修改(git --checkout)**

如果你并不想保留最后对README.md文件的修改怎么办？你该如何方便地撤消修改将它还原成上次的状态？幸运的是，git status也告诉了你应该如何做。 输入命令git status可以看到README.md文件被修改了，未暂存区域是这样：

$git status

On branch main

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README.md

它非常清楚地告诉了你如何撤消之前所做的修改。 让我们来按照提示执行：

$ git checkout – README.md

$git status

On branch main

Untracked files:

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

可以看到那些修改已经被撤消了。请务必牢记git checkout -- <file>是一个危险的命令。 你对那个文件在本地的任何修改都会消失——Git 会用最近提交的版本覆盖掉它。除非你确信不想要对那个文件的本地修改了，否则请不要使用这个命令。

**取消暂存的文件(git reset)**

接下来我们看看如何操作暂存区和工作目录中已修改的文件。这些命令在修改文件状态的同时，也会提示如何撤消操作。例如，你意外地输入git add xiugai.txt暂存了这个文件，但是你想要取消暂存继续修改该文件，如何取消暂存这个文件呢？git status命令提示了你：

$ git add xiugai.txt

$ git status

On branch main

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: xiugai.txt

在 “Changes to be committed” 文字正下方，提示使用 git reset HEAD <file>... 来取消暂存。 所以，我们可以这样来取消暂存xiugai.txt文件：

$ git reset HEAD xiugai.txt

$ git status

On branch main

Untracked files:

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

xiugai.txt

如上所示，xiugai.txt 文件已经是未暂存的状态了。git reset确实是个危险的命令，如果加上了 --hard 选项则更是如此。 然而在上述场景中，工作目录中的文件尚未修改，因此相对安全一些。

**修补上一次提交到分支的日志和文件(git commit --amend)**

在任何一个阶段，你都有可能想要撤消某些操作。接下来，我们将会学习几个撤消你所做修改的基本命令。 注意，有些撤消操作是不可逆的。 这是在使用 Git 的过程中，会因为操作失误而导致之前的工作丢失的少有的几个地方之一。

有时候我们提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。此时，可以运行带有--amend选项的提交命令来重新提交：git commit --amend这个命令会提交暂存区中的文件。

(1)**如果只想修改上次的提交信息**，那么不需要执行git add，只需要直接执行此命令即可，提交的文件内容会保持不变。例如首次提交xiugai.txt文件：

$ git add xiugai.txt

$ git commit -m "add xiugai.txt"

[main f90b9b2] add xiugai.txt

1 file changed, 2 insertions(+)

create mode 100644 xiugai.txt

$ git reflog

f90b9b2 HEAD@{0}: commit: add xiugai.txt

1b5a233 HEAD@{1}: commit: delete test.c

dd01084 HEAD@{2}: commit: add test.c

33c2399 HEAD@{3}: commit: add Linux\_test

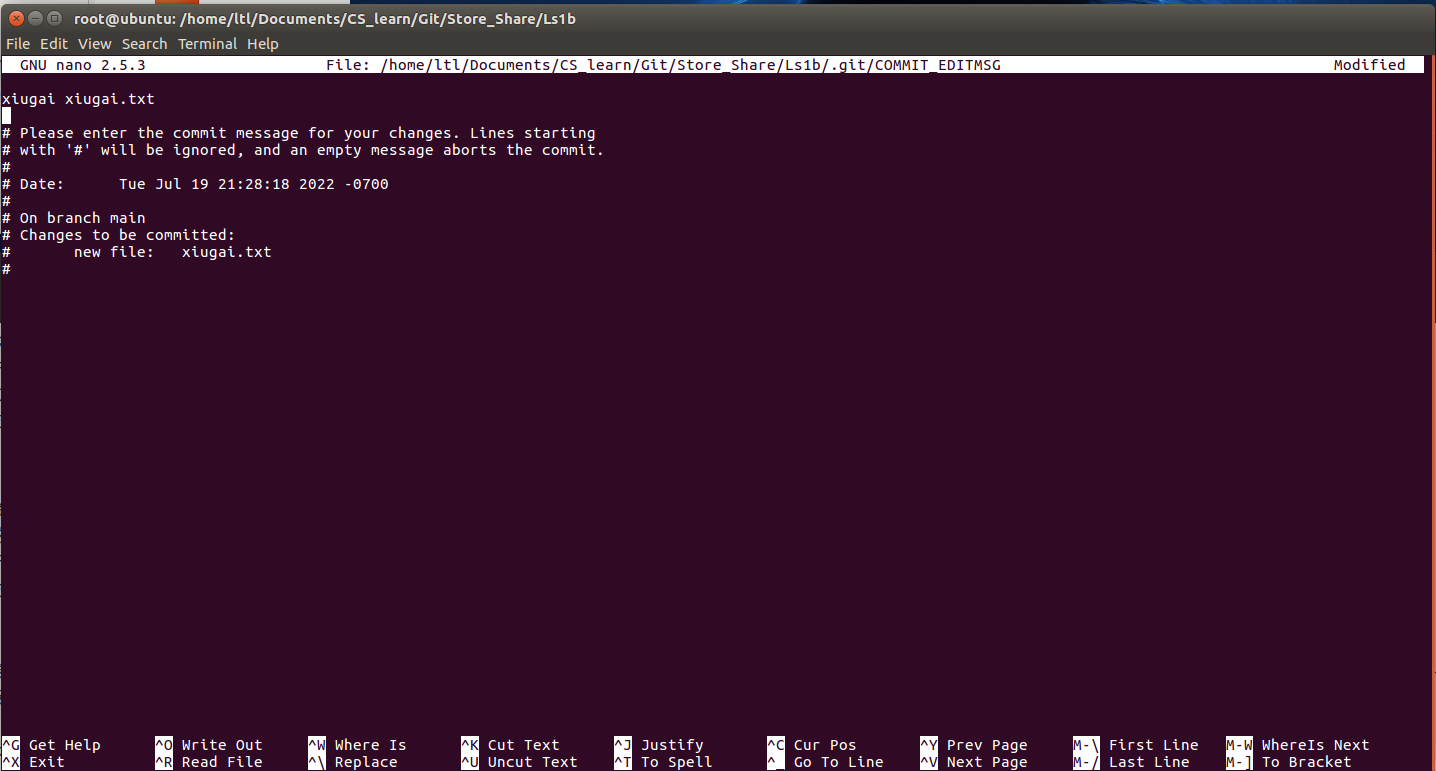
22f3371 HEAD@{4}: commit: deleted Linux\_test

a528c38 HEAD@{5}: commit: .

cf2b289 HEAD@{6}: clone: from <https://github.com/litianling/Ls1b>

可以看到上一次提交的说明是add xiugai.txt，现在我们想修改提交的说明，可直接运行命令git commit –amend：

$ git commit -amend



如上图所示修改信息后，按下Alt+X退出。

[main f40fb6f] xiugai xiugai.txt

Date: Tue Jul 19 21:28:18 2022 -0700

1 file changed, 2 insertions(+)

create mode 100644 xiugai.txt

然后我们可以查看修改后的日志：

$ git reflog

f40fb6f HEAD@{0}: commit (amend): xiugai xiugai.txt

f90b9b2 HEAD@{1}: commit: add xiugai.txt

1b5a233 HEAD@{2}: commit: delete test.c

dd01084 HEAD@{3}: commit: add test.c

33c2399 HEAD@{4}: commit: add Linux\_test

22f3371 HEAD@{5}: commit: deleted Linux\_test

a528c38 HEAD@{6}: commit: .

cf2b289 HEAD@{7}: clone: from <https://github.com/litianling/Ls1b>

$ git log

commit 80dc5c8fdce1113eb116ef952df4c4aaf91a1535

Author: litianling <2467280868@qq.com>

Date: Tue Jul 19 21:28:18 2022 -0700

xiugai xiugai.txt

commit 1b5a2332cb87bf67bbea3f2cfcced05fbe871789

Author: litianling <2467280868@qq.com>

Date: Tue Jul 19 06:16:52 2022 -0700

delete test.c

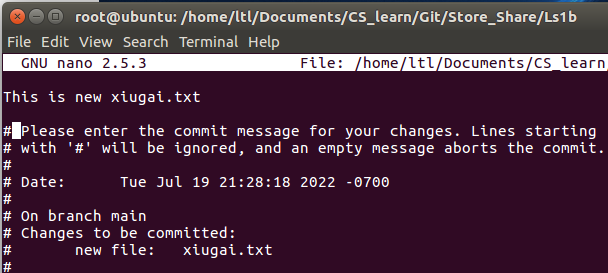
从上面可以看到，上一次提交的add xiugai.txt被修改成了xiugai xiugai.txt。注意，git reflog可以看到日志的所有修改过程，而git log只会看到最终的日志。

(2)如果不只想修改上次的提交信息，而是想修改或补充提交文件，例如修改上次提交的文件。可以像下面这样操作：

$ gedit xiugai.txt

$ git add xiugai.txt

$ git commit --amend



[main d1a2dfe] This is new xiugai.txt

Date: Tue Jul 19 21:28:18 2022 -0700

1 file changed, 2 insertions(+)

create mode 100644 xiugai.txt

$ git reflog

d1a2dfe HEAD@{0}: commit (amend): This is new xiugai.txt

80dc5c8 HEAD@{1}: commit (amend): xiugai xiugai.txt

f40fb6f HEAD@{2}: commit (amend): xiugai xiugai.txt

f90b9b2 HEAD@{3}: commit: add xiugai.txt

1b5a233 HEAD@{4}: commit: delete test.c

dd01084 HEAD@{5}: commit: add test.c

33c2399 HEAD@{6}: commit: add Linux\_test

22f3371 HEAD@{7}: commit: deleted Linux\_test

a528c38 HEAD@{8}: commit: .

cf2b289 HEAD@{9}: clone: from <https://github.com/litianling/Ls1b>

$ git log

commit d1a2dfec2283bc7196251cc696760102aadcfbc9

Author: litianling <2467280868@qq.com>

Date: Tue Jul 19 21:28:18 2022 -0700

This is new xiugai.txt

commit 1b5a2332cb87bf67bbea3f2cfcced05fbe871789

Author: litianling <2467280868@qq.com>

Date: Tue Jul 19 06:16:52 2022 -0700

delete test.c

最终你只会有一个提交——第二次提交将覆盖第一次提交的结果。从效果上来说，就像是原先的提交从未存在过一样，它并不会出现在仓库的日志中。

修补提交最明显的价值是可以稍微改进你最后的提交，而不会让“啊，忘了添加一个文件”或者 “小修补，修正笔误”这种提交信息弄乱你的仓库历史。

**删除文件**

在 Git 中，删除也是一个修改操作，之前我们添加并提交了xiugai.txt文件。一般情况下，直接在文件夹中把没用的文件用rm命令删除了：rm -rf xiugai.txt这个时候，Git知道你删除了文件，工作区和版本库就不一致了，git status命令会立刻告诉你哪些文件被删除了：

$ rm -rf xiugai.txt

$ git status

On branch main

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: xiugai.txt

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

(1)现在你有两个选择，一种情况是确实要从版本库中删除该文件，那就用命令git rm删掉，并且git commit：

$ git rm xiugai.txt

rm 'xiugai.txt'

$ git commit -m "remove xiugai.txt"

[main 3ecaa1c] remove xiugai.txt

1 file changed, 2 deletions(-)

delete mode 100644 xiugai.txt

现在，文件就从版本库中被删除了。小提示：先手动删除文件，然后使用git rm 和git add效果是一样的。

(2)另一种情况是本地文件夹中的文件删错了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到版本库里面的最新版本：git checkout -- xiugai.txt。git checkout其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

**注意：某文件从来没有添加进版本库就被删除的话是无法恢复的！**

**小结**

以上这些就是 git 中关于撤销以及删除文件的相关内容了，总结：

**场景1**：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令git checkout -- file

**场景2**：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令git reset HEAD <file>就回到了场景1，第二步按场景1操作

**场景3**：已经提交了不合适的修改到版本库时，想要修改上一次的提交，用git commit –amend修补上一次提交的文件和日志。

**场景4**：已经提交了不合适的修改到版本库时，想要撤销本次提交恢复到指定版本，可以用命令git reset --hard 版本ID，不过前提是没有推送到远程版本库。**一键将工作区回退到指定版本库状态，无需任何多余操作。**

## 分支管理

几乎所有的版本控制系统都以某种形式支持分支。使用分支意味着你可以把你的工作从开发主线上分离开来，以免影响开发主线。在很多版本控制系统中，常常需要完全创建一个源代码目录的副本，这个过程效率略低。对于大项目来说，这样的过程会耗费很多时间。

有人把 Git 的分支模型称为它的“必杀技特性”，也正因为这一特性使得 Git 从众多版本控制系统中脱颖而出。Git 处理分支的方式可谓是难以置信的轻量，创建新分支这一操作几乎能在瞬间完成，并且在不同分支之间的切换操作也是相当便捷。与许多其它版本控制系统不同，Git 鼓励在工作流程中频繁地使用分支与合并，哪怕一天之内进行许多次。

**git 分支简介**

为了真正理解Git处理分支的方式，我们需要回顾一下Git 是如何保存数据的。

前面我们了解到，Git保存的不是文件的变化或者差异，而是一系列不同时刻的快照。

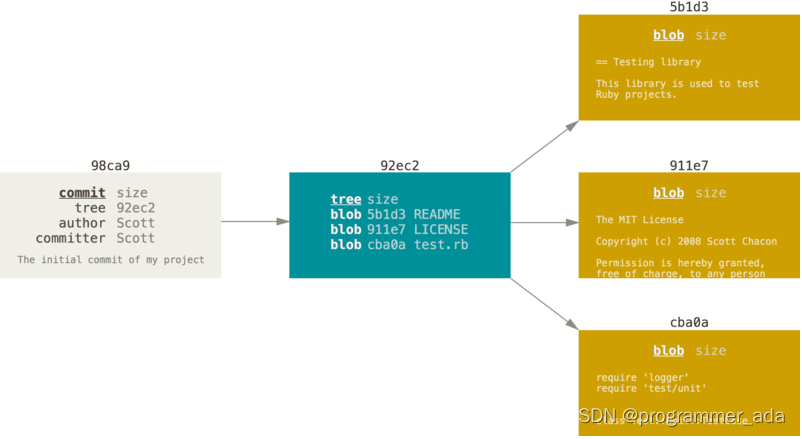
在进行提交操作时，Git 会保存一个提交对象（commit object）。知道了Git保存数据的方式，我们可以很自然的想到——该提交对象会包含一个指向暂存内容快照的指针。但不仅仅是这样，该提交对象还包含了作者的姓名和邮箱、提交时输入的信息以及指向它的父对象的指针。首次提交产生的提交对象没有父对象，普通提交操作产生的提交对象有一个父对象， 而由多个分支合并产生的提交对象有多个父对象，为了更加形象地说明，我们假设现在有一个工作目录，里面包含了三个将要被暂存和提交的文件。暂存操作会为每一个文件计算校验和，然后会把当前版本的文件快照保存到Git仓库中(Git使用blob对象来保存它们)，最终将校验和加入到暂存区域等待提交：

$ git add readme.txt test.md LICENSE

$ git commit -m 'The initial commit of my project'

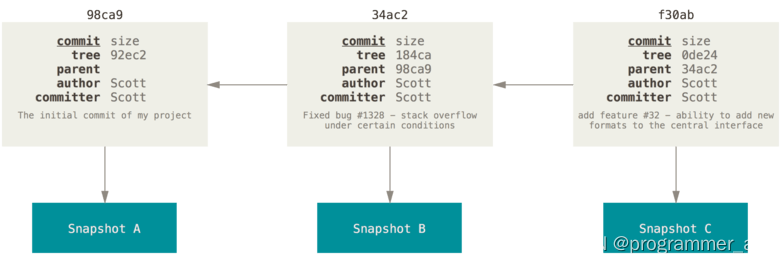
当使用git commit进行提交操作时，Git会先计算每一个子目录（本例中只有项目根目录）的校验和，然后在Git仓库中这些校验和保存为树对象。随后，Git 便会创建一个提交对象，它除了包含上面提到的那些信息外，还包含指向这个树对象（项目根目录）的指针。 如此一来，Git 就可以在需要的时候重现此次保存的快照。

现在，Git 仓库中有五个对象：三个blob对象（保存着文件快照）、一个树对象（记录着目录结构和blob对象索引）以及一个提交对象（包含着指向前述树对象的指针和所有提交信息）。



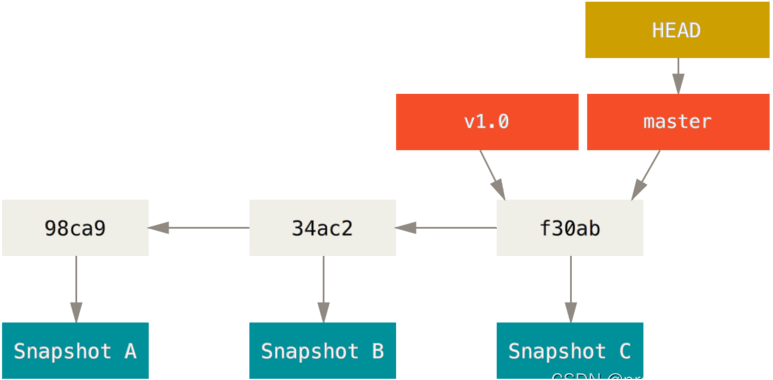
首次提交对象及其树结构

做些修改后再次提交，那么这次产生的提交对象会包含一个指向上次提交对象（父对象）的指针。



提交对象及其父对象

Git的分支，其实本质上仅仅是指向提交对象的可变指针。Git的默认分支名字是main。 在多次提交操作之后，你其实已经有一个指向最后那个提交对象的mian分支。main分支会在每次提交时自动向前移动。



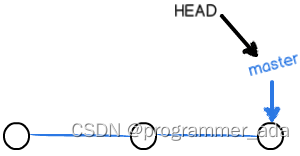
**main**

分支及其提交历史

**创建分支**

**Git会把仓库中的每次提交串成一条时间线，这条时间线就是一个分支**。在 Git 里，每个仓库都会有一个主分支，即main分支。HEAD严格来说不是指向提交，而是指向main，main才是指向提交的，所以HEAD指向的就是当前分支。

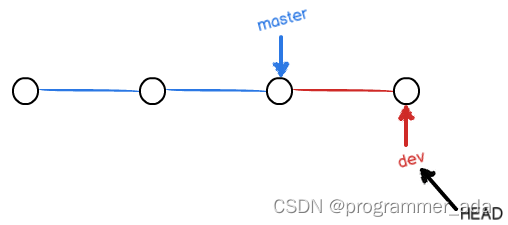
一开始的时候，main分支是一条线，Git用main指向最新的提交，再用HEAD指向main，就能确定当前分支，以及当前分支的提交点：



**main**

每次提交main分支都会向前移动一步，随着你不断提交main分支的线也越来越长。

当我们**创建**新的分支(dev)时，Git会新建一个指针dev，指向main相同的提交，再把HEAD指向dev，就表示当前分支在dev上：

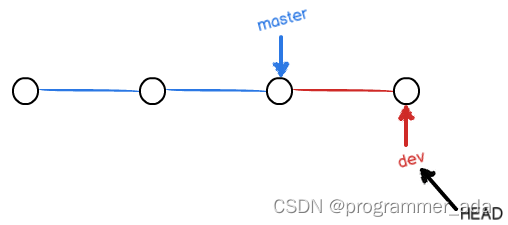


**main**

**main**

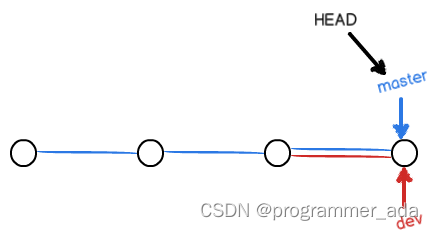
**如上图所示，Git创建一个分支很快，因为除了增加一个dev指针并修改HEAD的指向，工作区的文件都没有任何变化！**

不过，从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev指针往前移动一步，而main指针不变：



**main**

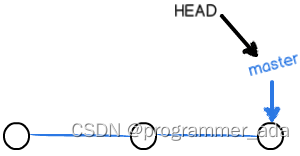
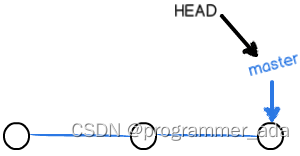
假如我们在dev上的工作完成了，就可以把dev合并到master上。Git 怎么**合并**呢？最简单的方法，就是直接把main指向dev的当前提交，就完成了合并：



**main**

**所以 Git 合并分支也很快！就改改指针，工作区内容也不变！**

合并完分支后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，删掉后，我们就剩下了一条main分支：



**main**

**下面开始实战**

首先，我们创建dev分支，然后切换到dev分支：

$ git branch dev

$ git checkout dev

Switched to branch 'dev'

git checkout命令加上-b参数表示创建并切换git checkout -b dev，相当于以上两条命令。然后，用git branch命令查看当前分支：git branch命令会列出所有分支，当前分支前面会标一个\*号。

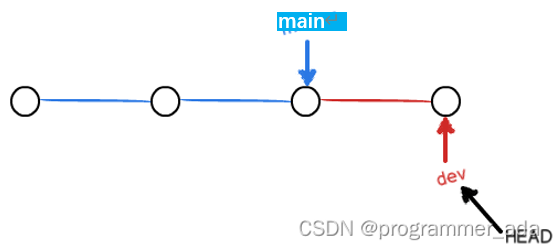
$ git branch

\* dev

main

然后，我们就可以在dev分支上正常修改(例如添加一个branch.txt文件)，然后用git add和git commit -m提交。现在，dev分支的工作完成，我们就可以用**git checkout main**切换回main分支。

切换回main分支后，可以发现刚才添加的文件不见了！因为那个提交是在dev分支上，而main分支此刻的提交点并没有变：



**branch.txt**

现在，我们把dev分支的工作成果合并到main分支上：

$ git merge dev

Updating 1b5a233..f440a50

Fast-forward

branch.txt | 2 ++

1 file changed, 2 insertions(+)

create mode 100755 branch.txt

**git merge**命令用于合并指定分支到当前分支。合并后，可以看到branch.txt文件和dev分支的最新提交是完全一样的。注意到上面的Fast-forward信息，Git 告诉我们这次合并是“快进模式”，也就是直接把main指向dev的当前提交，所以合并速度非常快。当然，也不是每次合并都能Fast-forward，我们后面会讲其他方式的合并。合并完成后，就可以放心地删除dev分支了：

$ git branch -d dev

Deleted branch dev (was 4aac6c7).

删除后，查看branch，就只剩下main分支了：

$ git branch

\* main

**因为创建、合并和删除分支非常快，所以 Git 鼓励你使用分支完成某个任务，合并后再删掉分支，这和直接在main分支上工作效果是一样的，但过程更安全。**

**git switch**

我们注意到切换分支使用git checkout <branch>，而Git中撤销修改则是git checkout -- <file>，同一个命令有两种作用确实有点令人迷惑。

实际上，切换分支这个动作用switch更科学。因此，最新版本的Git提供了新的git switch命令来切换分支。直接切换到已有的dev分支，可以使用：**git switch dev**。创建并切换到新的dev分支，可以使用：**git switch -c dev**。使用新的git switch命令，比git checkout要更容易理解。

**分支管理策略**

通常，Git合并分支时会用Fast forward模式，但这种模式下，删除分支后会丢掉分支信息。如果要强制禁用Fast forward模式，Git就会在合并时生成一个新的commit，这样，从分支历史上就可以看出分支信息。下面我们实战一下--no-ff方式的git merge：

首先，仍然创建并切换dev分支：

$ git branch dev

$ git checkout dev

Switched to branch 'dev'

修改branch.txt文件，并提交一个新的commit：

$ git add branch.txt

$ git commit -m "fix branch.txt"

[dev a64b4e7] fix branch.txt

1 file changed, 1 insertion(+), 1 deletion(-)

现在，我们切换回main：

$ git checkout main

Switched to branch 'main'

使用--no-ff参数合并dev分支，表示禁用Fast forward：

$ git merge --no-ff -m "merge with no-ff" dev

Merge made by the 'recursive' strategy.

branch.txt | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

因为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去。

合并后，我们用git lg看看分支历史(git lg是之前《Git安装中配置的别名》)：

$ git lg

\* 01cf6f1 - (HEAD -> main) merge with no-ff (12 minutes ago) <litianling>

|\

| \* a64b4e7 - (dev) fix branch.txt (16 minutes ago) <litianling>

|/

\* f440a50 - add branch.txt (10 hours ago) <litianling>

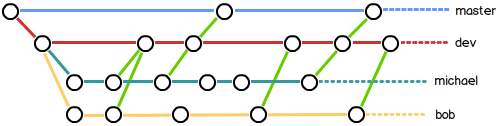
\* 1b5a233 - (origin/main) delete test.c (2 days ago) <litianling>

**上述过程可以看出，不使用Fast forward模式合并分分支才会显示这样的分支日志。**

**分支策略**

**在实际开发中，我们应该按照几个基本原则进行分支管理**：**首先**，main分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；**其次**，干活都在dev分支上，也就是说，dev分支是不稳定的，到某个时候，比如1.0版本发布时，再把dev分支合并到main上，并在main分支发布1.0版本；**总结**，你和你的小伙伴们每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并就可以了。

所以，团队合作的分支看起来就像这样：



**工作缓存**

**员工2**

**员工1**

**发布**

**main**

在日常的开发工作中，除了以上介绍的 main和 dev 两个常用分支外，我们还会有其他类型的分支使用策略：

**分支策略——bug分支**

在开发过程中，bug 就像家常便饭一样。有了bug就需要修复，在Git中每个bug都可以通过一个新的临时分支来修复，修复成功后合并分支，然后将临时分支删除。

当你接到一个修复一个代号101的bug的任务时，很自然地，你想创建一个分支issue-101来修复它，但是，你当前正在dev上进行的工作还没有提交：

$ git status

On branch dev

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: branch.txt

并不是你不想提交，而是当前工作只进行到一半还没法提交，预计完成还需1天时间。但是，这个bug比较急必须在两个小时内修复该 bug，怎么办？

幸好，Git还提供了一个功能(**git stash**)，可以把当前工作状态“储藏”起来，等以后恢复状态后继续工作：

$ git stash

Saved working directory and index state WIP on dev: a64b4e7 fix branch.txt

HEAD is now at a64b4e7 fix branch.txt

$ git status

On branch dev

nothing to commit, working directory clean

现在，用git status查看工作区就是干净的（除非有没有被Git管理的文件），因此可以放心地创建分支来修复bug。首先确定要在哪个分支上修复 bug，假定需要在main分支上修复，就从main创建临时分支：

$ git checkout main

Switched to branch 'main'

$ git branch issue-101

$ git checkout issus-101

Switched to branch 'issue-101'

现在修复bug，例如添加一个fix\_101.txt文件然后提交：

$ gedit fix\_101.txt

$ git add fix\_.txt

$ git commit -m "fix bug 101"

[issue-101 9538b8a] fix bug 101

1 file changed, 1 insertion(+)

create mode 100644 fix\_101.txt

修复完成后，切换到main分支，并完成合并，最后删除issue-101分支：

$ git checkout main

Switched to branch 'main'

$ Merge made by the 'recursive' strategy.

fix\_101.txt | 1 +

1 file changed, 1 insertion(+)

create mode 100644 fix\_101.txt

$ git branch -d issue-101

Deleted branch issue-101 (was 9538b8a).

$ git status

On branch main

nothing to commit, working directory clean

$ git lg

\* 135d982 - (HEAD -> main) merged bug fix 101 (2 minutes ago) <litianling>

|\

| \* 9538b8a - (issue-101) fix bug 101 (4 minutes ago) <litianling>

|/

\* 01cf6f1 - merge with no-ff (72 minutes ago) <litianling>

|\

| \* a64b4e7 - (dev) fix branch.txt (75 minutes ago) <litianling>

|/

\* f440a50 - add branch.txt (11 hours ago) <litianling>

……

bug 修复成功！现在，是时候接着回到dev分支继续干活了！

$ git switch dev

Switched to branch 'dev'

$ git status

On branch dev

nothing to commit, working tree clean

工作区是干净的，刚才的工作状态存到哪去了？用git stash list命令看看：

$ git stash list

stash@{0}: WIP on dev: a64b4e7 fix branch.txt

工作状态还在，Git 把stash内容存在某个地方了，但是需要恢复一下，有两个办法：一是用git stash apply恢复，但是恢复后，stash内容并不删除，你需要用git stash drop来删除；另一种方式是用git stash pop，恢复的同时把stash内容也删了，再用git stash list查看，就看不到任何stash内容了。

$ git stash pop

On branch dev

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: branch.txt

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

Dropped refs/stash@{0} (655b708481ad2adb768bd6ca5c92973550eb5309)

你可以多次stash，恢复的时候，先用git stash list查看，然后恢复指定的stash，用命令：**git stash apply stash@{0}**。

**在main分支上修复了bug后，我们要想一想，dev分支是早期从main分支分出来的，所以，这个bug其实在当前dev分支上也存在。那怎么在dev分支上修复同样的bug？**重复操作一次，提交不就行了？

在 Git 中还有比这更简单的方法可以实现。同样的 bug，要在dev上修复，我们只需要把fix bug 101这个提交所做的修改“复制”到dev分支。注意：我们只想复制fix bug 101这个提交所做的修改，并不是把整个main分支合并过来。为了方便操作，Git 专门提供了一个cherry-pick命令，让我们能复制一个特定的提交到当前分支（首先要查看对应某次提交的编号）：

$ git reflog

9538b8a HEAD@{5}: commit: fix bug 101

$ git cherry-pick 9538b8a

[dev b89eb66] fix bug 101

Date: Wed Jul 20 20:18:45 2022 -0700

1 file changed, 1 insertion(+)

create mode 100644 fix\_101.txt

Git 自动给dev分支做了一次提交，注意这次提交的commit编号是b89eb66，它并不同于main分支中的9538b8a，因为这两个commit只是改动相同，但确实是两个不同的commit。用git cherry-pick，我们就不需要在dev分支上手动再把修bug的过程重复一遍。

有些聪明的童鞋会想了，既然可以在main分支上修复bug后，在dev分支上可以“复现”这个修复过程，那么直接在dev分支上修复 bug，然后在main分支上“复现“行不行？当然可以，不过你仍然需要git stash命令保存现场，才能从dev分支切换到main分支。

**分支策略——feature分支**

在开发过程中，除了 bug 外也还会有无穷无尽的新的功能要不断添加进来。添加一个新功能时，你肯定不希望因为一些正在测试的代码把主分支搞乱了，所以每添加一个新功能，最好新建一个feature分支在上面开发，完成后合并最后删除该feature分支。

现在，你终于接到了一个新任务：开发代号为Vulcan的新功能，该功能计划用于下一代星际飞船。于是在main分支创建并切换到一个新的分支feature-vulcan准备开发，5分钟后，开发完毕并用git add/commit提交成功。最后切回dev分支准备合并，一切顺利的话，feature分支和bug分支是类似的，先合并再删除feature分支。

但是！就在此时，接到上级命令，因经费不足新功能必须取消！虽然白干了，但是这个包含机密资料的分支还是必须就地销毁：

$ git branch -d feature-vulcan

error: The branch 'feature-vulcan' is not fully merged.

If you are sure you want to delete it, run 'git branch -D feature-vulcan'.

Git 提醒你销毁失败，feature-vulcan分支还没有被合并，如果删除，将丢失掉修改，如果要强行删除，需要使用大写的-D参数。现在我们强行删除，终于删除成功！

$ git branch -D feature-vulcan

Deleted branch feature-vulcan (was d12cf23).

**Git 分支合并时存在冲突的解决办法**

人生不如意之事十之八九，合并分支往往也不是一帆风顺的。

在main分支创建feature1分支，并在feature1中在branch.txt文件最后添加This is feature1 branch并提交：

$ it branch feature1

$ git checkout feature1

Switched to branch 'feature1'

$ gedit branch.txt

$ git add branch.txt

$ git commit -m "feature1 branch"

[feature1 28cc59a] feature1 branch

1 file changed, 1 insertion(+)

然后切换到main分支，在branch.txt文件最后添加This is main branch并提交：

$ git checkout main

Switched to branch 'main'

$ gedit branch.txt

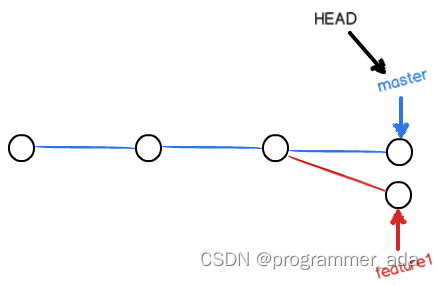
$ git add branch.txt

$ git commit -m "main branch"

[main 5905191] main branch

1 file changed, 1 insertion(+)

**现在，main分支和feature1分支各自都分别有新的提交，变成了这样**：



**main**

**这种情况下，Git 无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突**，我们试试看：

$ git merge feature1

Auto-merging branch.txt

CONFLICT (content): Merge conflict in branch.txt

Automatic merge failed; fix conflicts and then commit the result.

果然冲突了！Git 告诉我们，branch.txt文件存在冲突，必须手动解决冲突后再提交。现在git status也可以告诉我们冲突的文件：

$ git status

On branch main

You have unmerged paths.

(fix conflicts and run "git commit")

Unmerged paths:

(use "git add <file>..." to mark resolution)

both modified: branch.txt

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

我们可以直接查看branch.txt的内容：

$ gedit branch.txt

123456

fix this file

<<<<<<< HEAD

This is main branch

=======

This is feature1 branch

>>>>>>> feature1

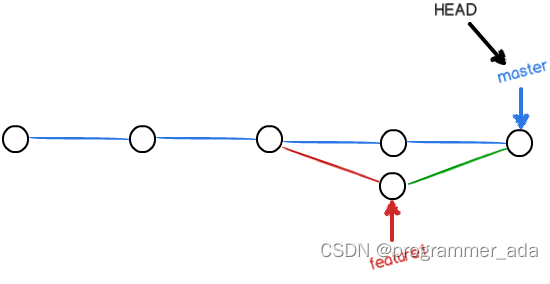
**Git在有冲突的文件中用<<<<<<<，=======，>>>>>>>标记出不同分支的内容**，我们修改为This is new main branch后保存并提交：

$ git add branch.txt

$ git commit -m "new main branch"

[main 806715a] new main branch

现在，master分支和feature1分支变成了下图所示：



用git lg也可以看到分支的合并情况：

$ git lg

\* 806715a - (HEAD -> main) new main branch (2 minutes ago) <litianling>

|\

| \* 28cc59a - (feature1) feature1 branch (18 minutes ago) <litianling>

\* | 5905191 - main branch (14 minutes ago) <litianling>

|/

\* 135d982 - merged bug fix 101 (4 hours ago) <litianling>

|\

| \* 9538b8a - fix bug 101 (4 hours ago) <litianling>

|/

\* 01cf6f1 - merge with no-ff (5 hours ago) <litianling>

|\

| \* a64b4e7 - fix branch.txt (5 hours ago) <litianling>

|/

\* f440a50 - add branch.txt (15 hours ago) <litianling>

最后，删除feature1分支工作完成。

$ git branch -d feature1

Deleted branch feature1 (was 28cc59a).

**git cherry-pick**

对于多分支的代码库，将代码从一个分支转移到另一个分支是常见需求。

这时分两种情况。一种情况是，你需要另一个分支的所有代码变动，那么就采用合并git merge。另一种情况是，你只需要部分代码变动（某几个commit提交），这时可以采用 cherry pick。

**git cherry-pick 基本用法**

git cherry-pick命令的作用，就是将指定的提交commit应用于其他分支。

$ git cherry-pick <commitHash>

上面命令就会将指定的提交commitHash，应用于当前分支。这会在当前分支产生一个新的提交，当然它们的哈希值会不一样。

举例来说，代码仓库有main和feature两个分支。

a - b - c - d main

\

e - f - g feature

现在如果要将f应用到main分支。先切换到main分支，再Cherry pick操作。

$ git checkout main

$ git cherry-pick f

上面的操作完成以后，代码库就变成了下面的样子。

a - b - c - d - f main

\

e - f - g feature

从上面可以看到，main分支的末尾增加了一个commit（提交）f。

git cherry-pick命令的参数，不一定是提交（commit）的哈希值，分支名也是可以的，表示转移该分支的最新commit。例如：git cherry-pick feature，上面代码表示将feature分支的最近一次提交，转移到当前分支。

**git cherry-pick 转移多个提交**

Cherry pick 支持一次转移多个提交。git cherry-pick <HashA> <HashB>命令表示将A和B两个commit应用到当前分支，这会在当前分支生成两个对应的新commit。

如果想要转移一系列的连续提交，可以使用命令的缩写。git cherry-pick A..B命令可以转移从A到B的所有commit。它们必须按照正确的顺序放置：commitA 必须早于commitB，否则命令将失败，但不会报错。

注意，使用上面的命令，commitA将不会包含在Cherry pick中。如果要包含commitA，可以使用git cherry-pick A^..B命令。

**git cherry-pick 配置项**

git cherry-pick命令的常用配置项如下

-e，--edit: 打开外部编辑器，编辑提交信息。

-n，--no-commit: 只更新工作区和暂存区，不产生新的提交。

-x: 在提交信息的末尾追加一行cherry picked from commit ...，方便以后查到这个commit是如何产生的。

-s，--signoff: 在提交信息的末尾追加一行操作者的签名，表示是谁进行了这个操作。

-m parent-number，--mainline parent-number: 如果原始提交是一个合并节点，来自于两个分支的合并，那么cherry pick 默认将失败，因为它不知道应该采用哪个版本的分支。

-m配置项告诉Git，应该采用哪个版本的分支。它的参数parent-number是一个从1开始的整数，代表原始提交的父分支编号。git cherry-pick -m 1 <commitHash>表示，Cherry pick 采用提交commitHash来自编号1的父分支的变动。一般来说，1号父分支是接受变动的分支，2号父分支是作为变动来源的分支。

**git cherry-pick 代码冲突**

如果操作过程中发生代码冲突，cherry pick会停下来，让用户决定如何继续操作。

（0）不加参数，冲突后直接修改冲突文件，然后add/commit即可。[本质上和方式1相同]

（1）--continue，用户解决代码冲突后，将修改的文件重新加入暂存区，然后使用git cherry-pick --continue <Hash>命令，让cherry pick过程继续执行。

（2）--abort发生代码冲突后，放弃合并，回到操作前的样子。

（3）--quit发生代码冲突后，退出cherry pick，但是不回到操作前的样子。

**git cherry-pick 转移另一个远程代码库**

cherry pick 也支持转移另一个代码库的提交。方法是先用git remote add target git://gitUrl链接到一个名为target的远程仓库。然后，用git fetch target将远程代码仓库抓取到本地。接着，用git log target/main查看远程仓库对应commit的哈希值。最后，使用git cherry-pick <commitHash>命令转移commit。

**Git 多人协作——查看远程分支**

当你从远程仓库克隆时，实际上Git 自动把本地的main分支和远程的main分支对应起来了，并且远程仓库的默认名称是origin。

要查看远程库的信息，用git remote或用git remote -v显示更详细的信息：

$ git remote -v

origin git@github.com:michaelliao/learngit.git (fetch)

origin git@github.com:michaelliao/learngit.git (push)

上面显示了可以抓取和推送的origin的地址。如果没有推送权限，就看不到push的地址。

**Git 多人协作——推送分支**

**推送分支，就是把该分支上的所有本地提交推送到远程库**。推送时要指定本地分支，这样Git 就会把该分支推送到远程库对应的远程分支上：git push origin main表示将本地main分支的所有commit推送到远程仓库origin对应的main分支上，git push origin dev表示将本地dev分支的所有commit推送到远程仓库origin对应的dev分支上。

但是，并不是一定要把本地分支往远程推送。例如：main分支是主分支，因此要时刻与远程同步；dev分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；bug分支只用于在本地修复bug，就没必要推到远程了；feature分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

总之，就是在 Git 中，分支完全可以在本地，你可以根据分支的需求来决定是否需要推送到远程！

**Git 多人协作——抓取分支**

多人协作时，大家都会往main和dev分支上推送各自的修改。首先开发者A(Linux)分别上传不同的dev和main分支。

$ git push origin dev

Counting objects: 23, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (16/16), done.

Writing objects: 100% (23/23), 1.88 KiB | 0 bytes/s, done.

Total 23 (delta 9), reused 0 (delta 0)

remote: Resolving deltas: 100% (9/9), done.

remote:

remote: Create a pull request for 'dev' on GitHub by visiting:

remote: https://github.com/litianling/Ls1b/pull/new/dev

remote:

To https://ghp\_C0xdPi74MUARQ7pXgcI6NGPCHiTvJL3uzl7r@github.com/litianling/Ls1b.git

\* [new branch] dev -> dev

$ git checkout main

Switched to branch 'main'

$ git push origin main

Total 0 (delta 0), reused 0 (delta 0)

To https://ghp\_C0xdPi74MUARQ7pXgcI6NGPCHiTvJL3uzl7r@github.com/litianling/Ls1b.git

1b5a233..806715a main -> main

现在，可以在另一台电脑（注意要把SSH Key添加到 GIT CODE）或者同一台电脑的另一个目录下克隆来模拟开发者B(Windos)：

$ git clone https://github.com/litianling/Ls1b

Cloning into 'Ls1b'...

remote: Enumerating objects: 411, done.

remote: Counting objects: 100% (411/411), done.

remote: Compressing objects: 100% (363/363), done.

remote: Total 411 (delta 33), reused 403 (delta 28), pack-reused 0

Receiving objects: 100% (411/411), 913.38 KiB | 1.18 MiB/s, done.

Resolving deltas: 100% (33/33), done.

当开发者B(Windos)从远程库clone时，默认情况下只能看到本地的main分支。

$ git branch

\* main

现在，当开发者B(Windos)要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是他用这个命令创建本地dev分支：

$ git checkout -b dev origin/dev

Switched to a new branch 'dev'

branch 'dev' set up to track 'origin/dev'.

现在，开发者B(Windos)就可以在dev上继续修改，时不时地把dev分支push到远程：

$ git add my\_name.txt

$ git commit -m "add windos name"

[dev ca37568] add windos name

1 file changed, 1 insertion(+)

create mode 100644 my\_name.txt

$ git push origin dev

Enumerating objects: 4, done.

Counting objects: 100% (4/4), done.

Delta compression using up to 4 threads

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 282 bytes | 282.00 KiB/s, done.

Total 3 (delta 1), reused 0 (delta 0), pack-reused 0

remote: Resolving deltas: 100% (1/1), completed with 1 local object.

To https://github.com/litianling/Ls1b

4115d3b..ca37568 dev -> dev

可以看到开发者B(Windos)已经向origin/dev分支推送了他的提交，而碰巧开发者A(Linux)也对同样的文件作了修改，并试图推送：

$ gedit my\_name.txt

$ git add my\_name.txt

$ git commit -m "add Linux name"

[dev d5cb2dd] add Linux name

1 file changed, 1 insertion(+)

create mode 100644 my\_name.txt

$ git push origin dev

To https://ghp\_C0xdPi74MUARQ7pXgcI6NGPCHiTvJL3uzl7r@github.com/litianling/Ls1b.git

! [rejected] dev -> dev (fetch first)

error: failed to push some refs to 'https://ghp\_C0xdPi74MUARQ7pXgcI6NGPCHiTvJL3uzl7r@github.com/litianling /Ls1b.git'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

上述信息显示推送失败，因为开发者B(Windos)的最新提交和开发者A(Linux)试图推送的提交有冲突， Git提示我们先用git pull把最新的提交从origin/dev抓下来，然后再本地合并解决冲突，最后再推送：

$ git pull

remote: Enumerating objects: 4, done.

remote: Counting objects: 100% (4/4), done.

remote: Compressing objects: 100% (1/1), done.

remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0

Unpacking objects: 100% (3/3), done.

From https://github.com/litianling/Ls1b

4115d3b..ca37568 dev -> origin/dev

There is no tracking information for the current branch.

Please specify which branch you want to merge with.

See git-pull(1) for details.

git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

git branch --set-upstream-to=origin/<branch> dev

git pull也失败了，原因是没有指定本地dev分支与远程origin/dev分支的链接，根据提示，设置dev和origin/dev的链接再pull：

$ git branch --set-upstream-to=origin/dev dev

Branch dev set up to track remote branch dev from origin.

$ git pull

Auto-merging my\_name.txt

CONFLICT (add/add): Merge conflict in my\_name.txt

Automatic merge failed; fix conflicts and then commit the result.

这回git pull成功，但是合并有冲突，需要手动解决，解决的方法和分支管理中的解决冲突完全一样。解决后提交再push：

$ gedit my\_name.txt

$ git add my\_name.txt

$ git commit -m "fix new Linux name"

[dev 3587a56] fix new Linux name

$ git push origin dev

Counting objects: 6, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (4/4), done.

Writing objects: 100% (6/6), 570 bytes | 0 bytes/s, done.

Total 6 (delta 2), reused 0 (delta 0)

remote: Resolving deltas: 100% (2/2), completed with 1 local object.

To https://ghp\_C0xdPi74MUARQ7pXgcI6NGPCHiTvJL3uzl7r@github.com/litianling/Ls1b.git

ca37568..3587a56 dev -> dev

**多人协作小结**

**因此，多人协作的工作模式通常是这样：首先，试图用git push origin <branch-name>推送自己的修改；如果推送失败，则因为远程分支比你的本地版本更新，需要先用git pull试图合并；如果合并有冲突则先解决冲突，并在本地提交；没有冲突或者解决掉冲突后，再用git push origin <branch-name>推送就能成功！【注】如果git pull提示no tracking information，则说明本地分支和远程分支的链接关系没有创建，用命令git branch --set-upstream-to <branch-name> origin/<branch-name>。**

**git rebase**

在上一节我们看到了，多人在同一个分支上协作时，很容易出现冲突。即使没有冲突，后push的开发者不得不先pull，在本地合并，然后才能push成功。每次合并再push后，分支就变得看上去很乱，如何将Git的提交历史变成一条干净的直线？

Git有一种称为rebase的操作，有人把它翻译成“变基”。我们来看看rebase是怎么把分叉的提交变成直线。

在和远程分支同步后，我们对hello.py这个文件做了两次提交。用git log命令看看：

$ git log --graph --pretty=oneline --abbrev-commit

\* f4e7ea8 (HEAD -> master) Merge branch 'dev' into master

|\

| \* ad728db (origin/dev, dev) fix env conflict

| |\

| | \* d7e6866 Update env.txt

| \* | 770ce3f new env

| |/

| \* 34bde7f add env

| \* 0944c8c fix bug 101

\* | b003293 (origin/master) merged bug fix 101

|\ \

| \* | 8842ff5 (issue-101) fix bug 101

|/ /

\* | fc76cf7 merge with no-ff

|\|

| \* c7b409a add merge

|/

\* fb8b190 conflict fixed

注意到Git用(HEAD -> main)和(origin/main)标识出当前分支的HEAD和远程origin的位置分别是582d922 add author和d1be385 init hello，本地分支比远程分支快两个提交。

现在我们尝试推送本地分支：

$ git push origin master

To gitcode.net:xiongjiamu/learning-git.git

! [rejected] master -> master (fetch first)

error: failed to push some refs to 'git@gitcode.net:xiongjiamu/learning-git.git'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

很不幸，失败了，这说明有人先于我们推送了远程分支。按照经验，先pull一下：

$ git pull

remote: Counting objects: 3, done.

remote: Compressing objects: 100% (1/1), done.

remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0

Unpacking objects: 100% (3/3), done.

From gitcode.net:xiongjiamu/learning-git.git

3010226..9dbfd41 master -> origin/master

Auto-merging hello.py

Merge made by the 'recursive' strategy.

hello.py | 1 +

1 file changed, 1 insertion(+)

再用git status看看状态：

$ git status

On branch master

Your branch is ahead of 'origin/master' by 3 commits.

(use "git push" to publish your local commits)

nothing to commit, working tree clean

加上刚才合并的提交，现在我们本地分支比远程分支超前3个提交。

用git log看看：

$ git log --graph --pretty=oneline --abbrev-commit

\* 00f5a14 (HEAD -> master) Merge branch 'master' of gitcode.net:xiongjiamu/learning-git

|\

| \* 9dbfd41 (origin/master) set exit=1

\* | 8789044 add author

\* | 01ebb65 add comment

|/

\* 3010226 init hello

...

对强迫症童鞋来说，现在事情有点不对头，提交历史分叉了。如果现在把本地分支push到远程，有没有问题？

这个时候，rebase就派上了用场。我们输入命令git rebase试试：

$ git rebase

First, rewinding head to replay your work on top of it...

Applying: add comment

Using index info to reconstruct a base tree...

M hello.py

Falling back to patching base and 3-way merge...

Auto-merging hello.py

Applying: add author

Using index info to reconstruct a base tree...

M hello.py

Falling back to patching base and 3-way merge...

Auto-merging hello.py

输出了一大堆操作，到底是啥效果？再用git log看看：

$ git log --graph --pretty=oneline --abbrev-commit

\* 22a1da7 - (HEAD -> master) add author

\* 00bae30 - add comment

\* 9dbfd41 - (origin/master) set exit=1

\* 3010226 - init hello

...

原本分叉的提交现在变成一条直线了！这种神奇的操作是怎么实现的？其实原理非常简单。我们注意观察，发现Git把我们本地的提交“挪动”了位置，放到了f005ed4 (origin/master) set exit=1之后，这样整个提交历史就成了一条直线。rebase操作前后，最终的提交内容是一致的，但是，我们本地的commit修改内容已经变化了，它们的修改不再基于d1be385 init hello，而是基于f005ed4 (origin/master) set exit=1，但最后的提交7e61ed4内容是一致的。

这就是rebase操作的特点：把分叉的提交历史“整理”成一条直线，看上去更直观。缺点是本地的分叉提交已经被修改过了。最后，通过push操作把本地分支推送到远程：

$ git push origin master

Enumerating objects: 8, done.

Counting objects: 100% (8/8), done.

Delta compression using up to 8 threads

Compressing objects: 100% (6/6), done.

Writing objects: 100% (6/6), 565 bytes | 565.00 KiB/s, done.

Total 6 (delta 3), reused 0 (delta 0), pack-reused 0

To gitcode.net:xiongjiamu/learning-git.git

9dbfd41..22a1da7 master -> master

再用git log看看效果：

$ git log --graph --pretty=oneline --abbrev-commit

\* 22a1da7 (HEAD -> master, origin/master) add author

\* 00bae30 add comment

\* 9dbfd41 set exit=1

\* 3010226 init hello

...

远程分支的提交历史也是一条直线。以上就是 rebase 的详细介绍了，现在让我们来回顾一下：**rebase操作可以****把本地未push的分叉提交历史整理成直线，rebase的目的是使得我们在查看历史提交的变化时更容易，因为分叉的提交需要三方对比**。

## Git标签

发布一个版本时，我们通常先在版本库中打一个标签（tag），这样就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像，但是分支可以移动，标签不能移动）。所以，创建和删除标签都是瞬间完成的。

Git有commit，为什么还要引入tag？“请把上周一的那个版本打包发布，commit号是6a5819e…”“一串乱七八糟的数字不好找！”如果换一个办法：“请把上周一的那个版本打包发布，版本号是v1.2”“好的，按照tag v1.2查找commit就行！”所以，tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。

**创建标签**

在 Git 中创建标签非常简单，首先，切换到需要打标签的分支上：

$ git branch

\* dev

Main

$ git checkout main

Switched to branch 'main'

然后，敲命令git tag <name>就可以打一个新标签，例如**git tag v1.0**，可以用命令git tag查看所有标签：

$ git tag

v1.0

默认标签是打在最新提交的commit上的。有时候，如果忘了打标签，比如现在已经是周五了，但应该在周一打的标签没有打，怎么办？找到历史提交的commit id，然后打上就可以了：

$ git lg

\* 13e6037 - (HEAD -> main, tag: v1.0, dev) merge remove RT code (2 hours ago) <litianling>

|\

| \* d179eea - remove RT code (2 hours ago) <litianling>

|/

\* 3587a56 - (origin/main, origin/dev) fix new Linux name (3 hours ago) <litianling>

比方说要对fix new Linux name这次commit打标签，它对应的commit id是3587a56，敲入命令：

$ git tag v0.9 3587a56

$ git tag

v0.9

v1.0

注：标签不是按时间顺序列出而是按字母排序的。可以用git show 查看标签详细信息：

$ git show v0.9

commit 3587a56745af3b86b29aa4e643f6334803b571b1

Merge: d5cb2dd ca37568

Author: litianling <2467280868@qq.com>

Date: Thu Jul 21 04:59:13 2022 -0700

fix new Linux name

diff --cc my\_name.txt

index 586a8ec,181c5f4..b2b9f63

mode 100644,100644..100755

--- a/my\_name.txt

+++ b/my\_name.txt

@@@ -1,1 -1,1 +1,2 @@@

-I am windos.

++First people is windos.

+I am Linux.

还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字，用命令git show <tagname>可以看到说明文字。

$ git tag -a v0.1 -m "This is the fiest version" 3d565c0

$ git show v0.1

tag v0.1

Tagger: litianling <2467280868@qq.com>

Date: Thu Jul 21 08:29:37 2022 -0700

This is the fiest version

commit 3d565c05547acb94f4aca05c79b419004d805a12

Author: litianling <104258709+litianling@users.noreply.github.com>

Date: Tue Jul 5 20:52:24 2022 +0800

Initial commit

diff --git a/README.md b/README.md

new file mode 100644

index 0000000..c60b547

--- /dev/null

+++ b/README.md

@@ -0,0 +1,2 @@

+# Ls1b

+Graphical operating system RT-Thread

**注意：标签总是和某个commit挂钩。如果这个commit既出现在main分支又出现在dev分支，那么在这两个分支上都可以看到这个标签。**

标签操作

因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除：

$ git tag -d v0.1

Deleted tag 'v0.1' (was 9fbb3b9)

如果要推送某个标签到远程，使用命令git push origin <tagname>：

$ git push origin v1.0

git push origin v1.0

Counting objects: 3, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 309 bytes | 0 bytes/s, done.

Total 3 (delta 2), reused 0 (delta 0)

remote: Resolving deltas: 100% (2/2), completed with 1 local object.

To https://ghp\_C0xdPi74MUARQ7pXgcI6NGPCHiTvJL3uzl7r@github.com/litianling/Ls1b.git

\* [new tag] v1.0 -> v1.0

一次性将所有未被推送的本地标签全部推送到远程：

$ git push origin --tags

Total 0 (delta 0), reused 0 (delta 0)

To https://ghp\_C0xdPi74MUARQ7pXgcI6NGPCHiTvJL3uzl7r@github.com/litianling/Ls1b.git

\* [new tag] v0.9 -> v0.9

如果标签已经推送到远程，从远程删除。删除命令也是push，但是格式如下：

$ git push origin :refs/tags/v1.0

To https://ghp\_C0xdPi74MUARQ7pXgcI6NGPCHiTvJL3uzl7r@github.com/litianling/Ls1b.git

- [deleted] v1.0

要看看是否真的从远程库删除了标签，可以登录github上查看。



**小结**

在对应的branch给最后一次commit创建标签 git tag <tagname>

给指定的某次commit创建标签 git tag <tagname> <commit id>

创建带有说明性文字的标签 git tag -a <tagname> -m <"massgges"> <commit id>

可以创建一个带gpg签名的标签git tag -s <tagname> -m <"messages"> <commit id>

查看所有存在的标签(按照字母排序) git tag

查看指定标签的具体内容 git show <tagname>

删除指定标签 git tag -d <tagname>

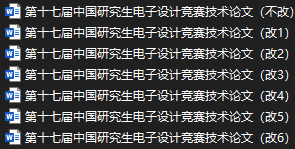
将指定标签推送到远程 git push origin <tagname>

一次性将所有未被推送的本地标签全部推送到远程 git push origin --tags

可以删除一个远程标签 git push origin :refs/tags/<tagname>

## Mac+VS Code+Git+Github

Github是全球最大的开源网站，即便是你不写代码，也需要用 Git和Github。在写论文尤其是毕业论文的时候，需要不断反复修改，直到最后改到头晕脑涨，也不知道改到了哪一版，就像图中那样。



然后呢，打印论文的时候还很可能打印的不是最终版，直接导致哭晕在打印店。到了这个时候，我就要使出绝技 Github和Git了。

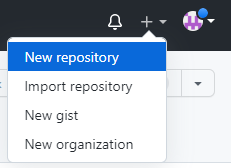
**Git和Github的异同**

**Git官方解释：**Git 是由 Linux 之父 Linus Tovalds 为了更好地管理linux内核开发而创立的分布式版本控制／软件配置管理软件。Git的特别之处就是他记录的不是文件本身，而是文件的改变。就是因为记录的是改变，所以针对上述版本杂乱的论文，如果我们用了Git，一个文件就能解决你所有烦恼。

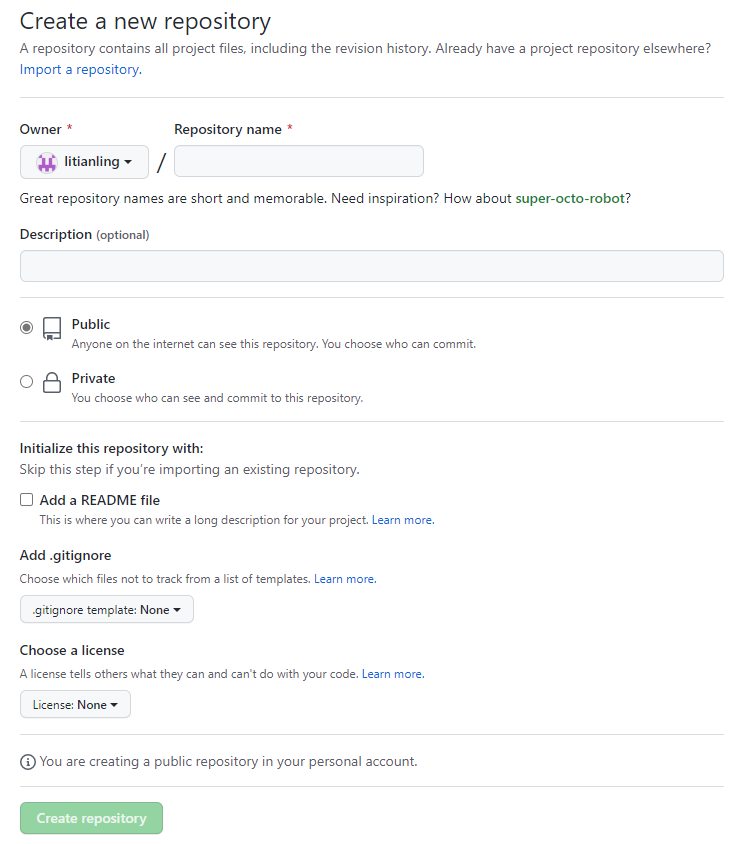
**Github** ：首先，Github是基于Git的一个全球最大的开源网站。方便程序员将自己的设计保存在网站上进行开源，也可以学习别人的开源代码，还可以多人共同开发维护一个项目。

**Git和Github怎么用呢**

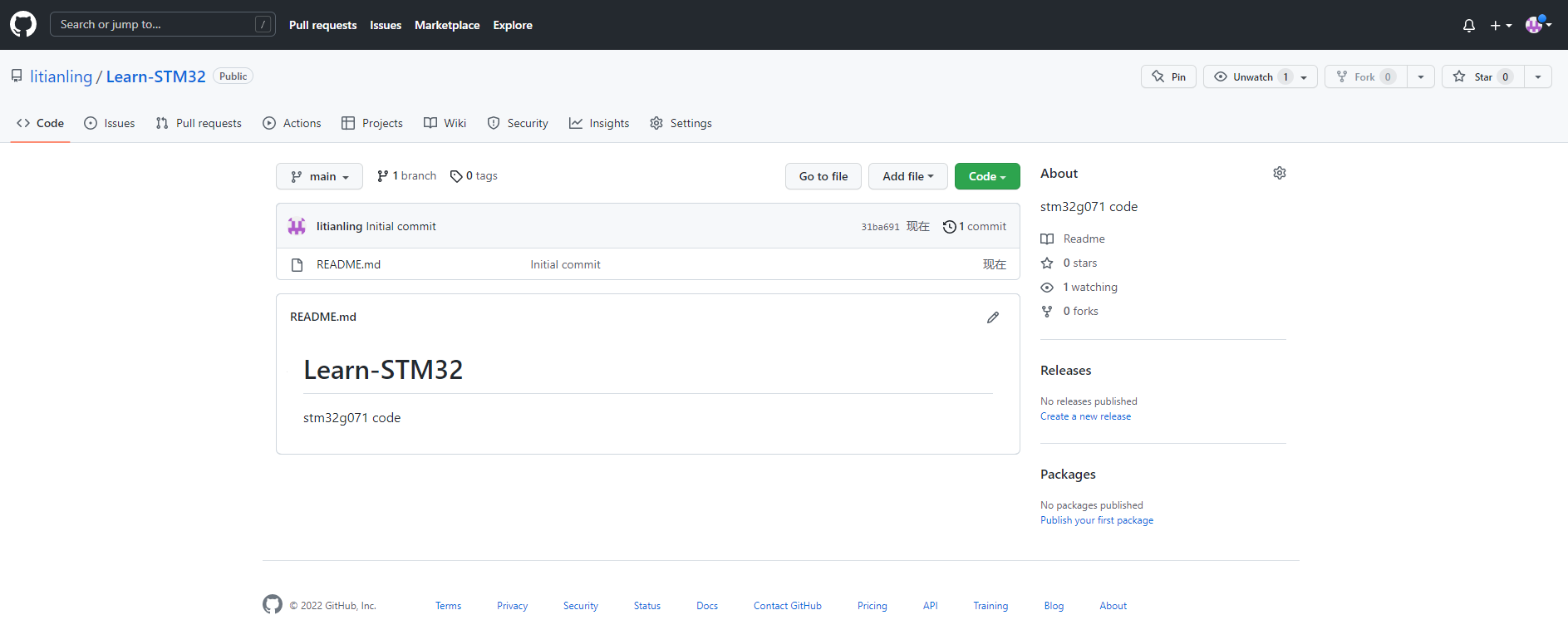
Github的官方网址如下所示<https://github.com/>其官方文档如下所示https://edu.csdn.net/skill/gml/gml-23b8a7b7033540e388c911bb0298aa7b?category= 635&typeId=21553。官方文档可以帮助大家很好地了解怎么使用Github。创建代码仓库方法如下：



然后按照提示配置该代码仓库设置：



图中画红框的两个部分： Add.gitignore 添加项用来告诉Github从本地上传至Github时需要忽略哪些文件。Add license添加项是开源协议，代表这个仓库不同等级的开源程度。仓库创建完成之后，如下图所示。



接下来呢，就是要和我们的本地仓库进行关联和同步。

**在VS Code用Git**

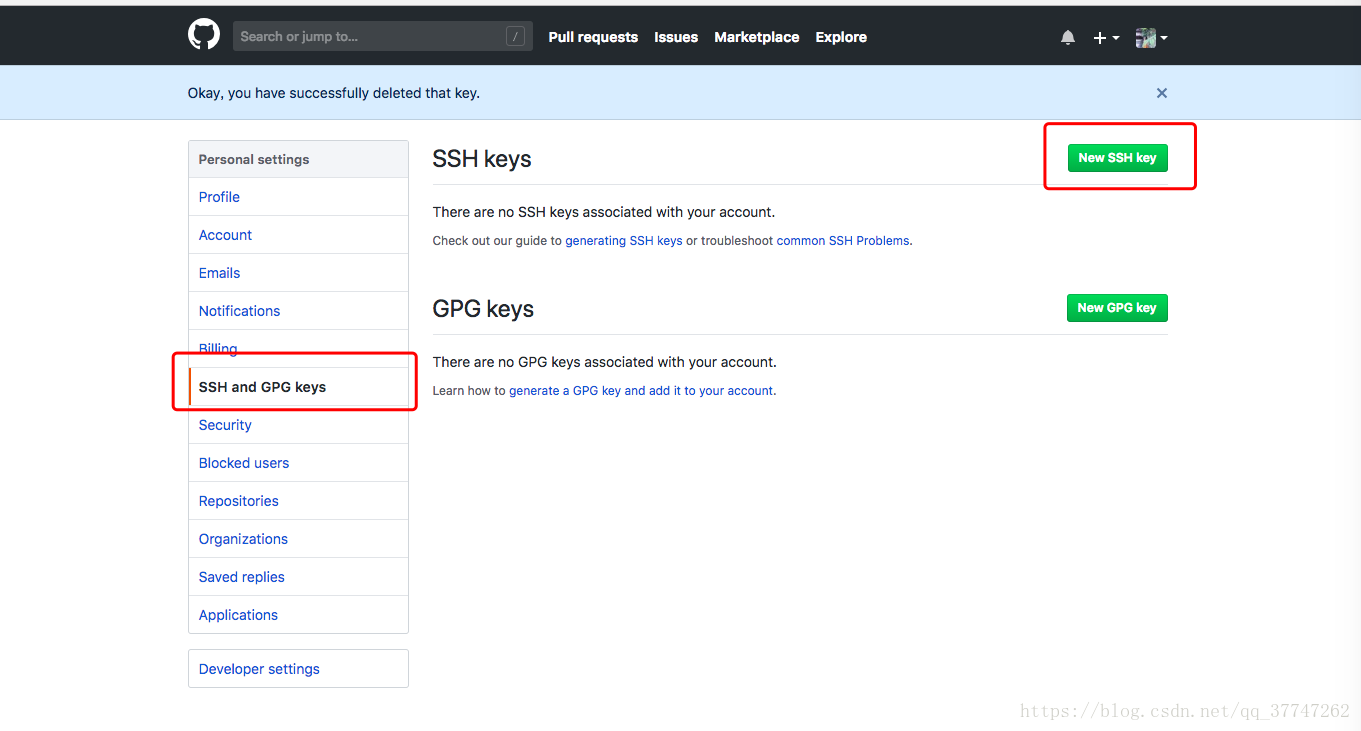
VS Code里是直接集成了Git，如果不用VS Code的同学，需要自行下载Git，具体的安装步骤网见上述《Git安装》，也可以参考https://blog.csdn.net/fen\_fen/article/details/105428400?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522165703052616781818794894%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request\_id=165703052616781818794894&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~sobaiduend~default -2-105428400-null-null.142^v31^pc\_search\_v2,185^v2^control&utm\_term=%E5%B0%86% E4%BB%A3%E7%A0%81%E4%B8%8A%E4%BC%A0%E5%88%B0github&spm=1018.2226.3001.4187。

接下来的操作都是在Git bash的终端进行的，当然也可以直接用VS Code终端或者VS Code自带的界面直接操作。

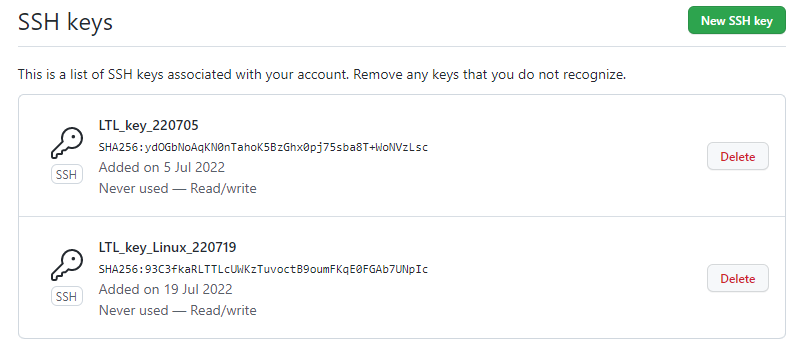
**配置专属SSH key**

SSH key是个什么东西？有了这个专属key，才可以实现身份的认证，然后才能正常地push到Github。首先在本地终端输入ssh-keygen -t rsa -C "2467280868@qq.com"。【注】-C后面是你自己注册 Github时申请的邮箱。然后一路enter，你的专属SSH key就生成了。去之前生成的SSH key路径下打开id\_rsa.pub文件，该文件中的内容就是SSH key公钥。

将id\_rsa.pub复制所得的内容粘贴到Github的SSH key中，具体的操作如下两张图所示。其中title可以随便取，key填的就是刚刚复制的公钥的内容。



添加成功之后是，如下图所示：



最后，在本地测试一下是否连接成功。

$ ssh -T ssh -T git@github.com

The authenticity of host 'github.com (20.205.243.166)' can't be established.

ECDSA key fingerprint is SHA256:p2QAMXNIC1TJYWeIOttrVc98/R1BUFWu3/LiyKgUfQM.

Are you sure you want to continue connecting (yes/no)?

连接测试成功之后如上图所示。至此，我们已经成功和Github通信啦，接下来就需要在本地进行后续的配置。

在本地配置Github

首先用git config --global user.name <user\_name>配置用户名。用git config --global user.email < user\_email >配置邮箱（注册 Github时申请的邮箱）。最后用git clone <https:// >克隆Github远程仓，用https克隆远程仓库不需要设置SSH key，但是用SSH克隆远程仓库需要SSH key，其具体命令例如：git clone [git@github.com:litianling/Ls1b.git](mailto:git@github.com:litianling/Ls1b.git)。若采用此方式，则需要SSH KEY的相关配置，大家可以各取所需。

这个时候，你已经成功的将远程仓库克隆到本地了，接下来看看怎么上传文件。用gedit <file\_name>创建、修改文件，用git add <file\_name>将文件添加在**本地版本库的暂存区**，用git commit -m <“information”>将暂存区全部文件一次性提交到**本地版本库的对应分支**生成一个commit。最后用命令git push <remote\_name> <branch\_name>将本地名为branch\_name的分支推动到远程仓库remote\_name对应的分支branch\_name。

其中，git commit可以简单填写这次提交的信息，便于后续版本查看的时候进行回忆。

git add之后的内容要换成你自己创建、修改或删除的文件的名字，remote\_name默认为origin，常用的branch\_name有默认的main（主要发布分支）和自己创建的dev（开发分支）。

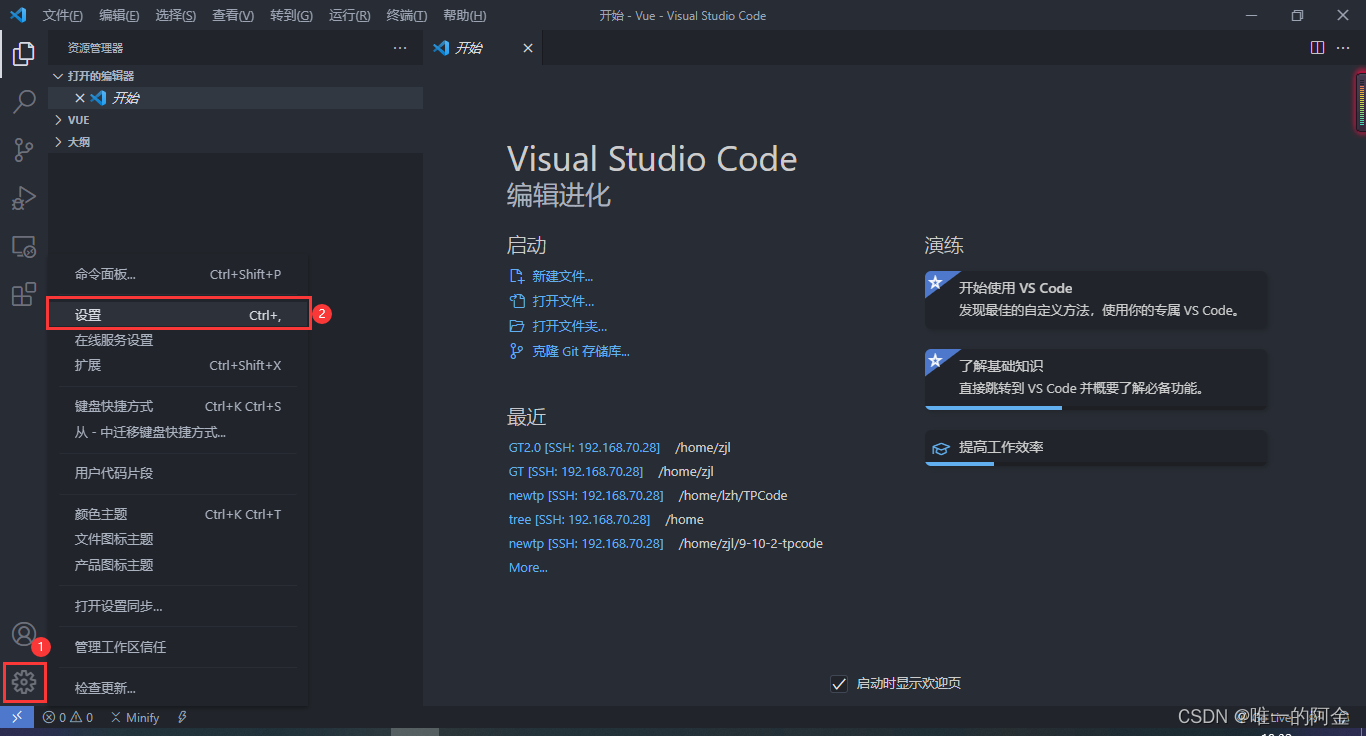
## Windos+VS Code+Git+Github

当我们在VScode中编写代码后，需要提交到git仓库时，但是我们又不想切换到git的命令行窗口，我们可以在VScode中配置git，然后就可以很方便快捷的把代码提交到仓库中。

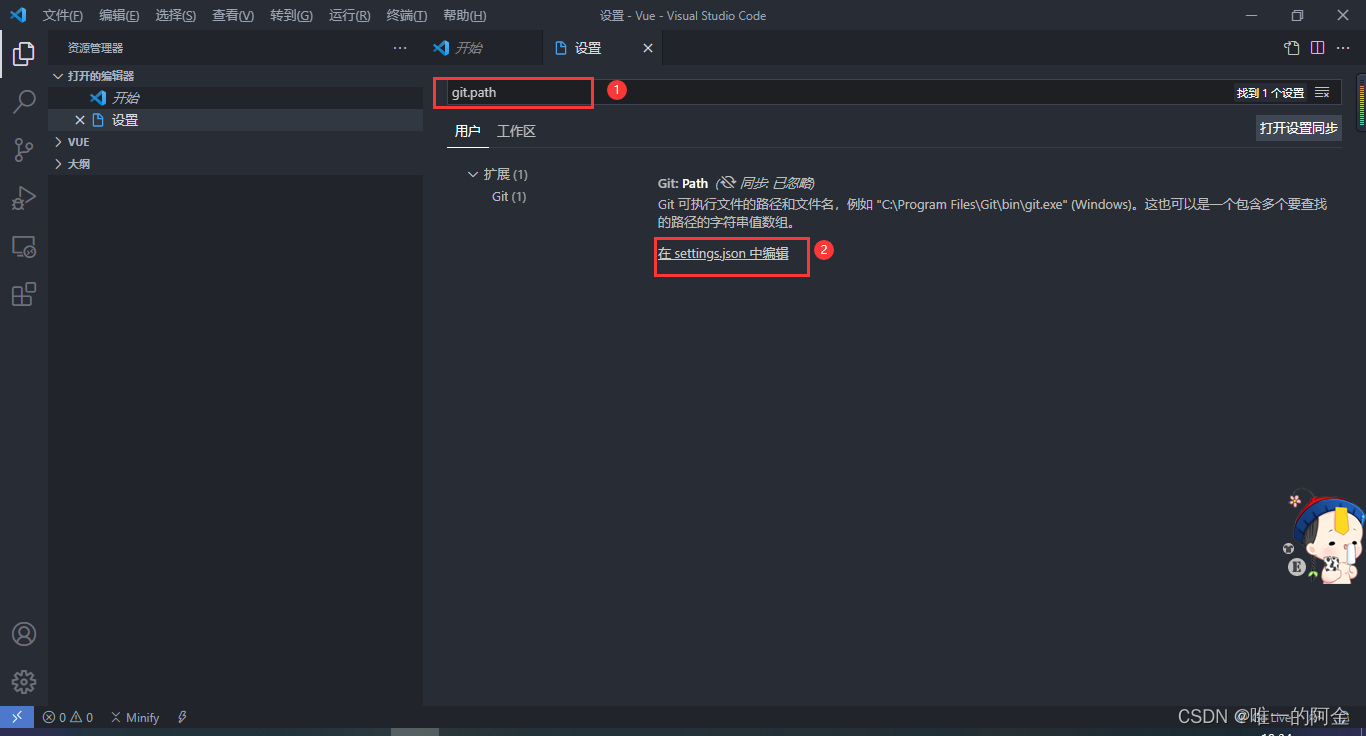
**第一步：安装Git命令行工具。**进入官网https://git-scm.com， 根据自己的电脑，下载相应的版本。注意：一定要记得自己安装的路径！！！安装完成后，在桌面上点击右键，出现Git gui和Git Bash的图标，则表示安装成功。

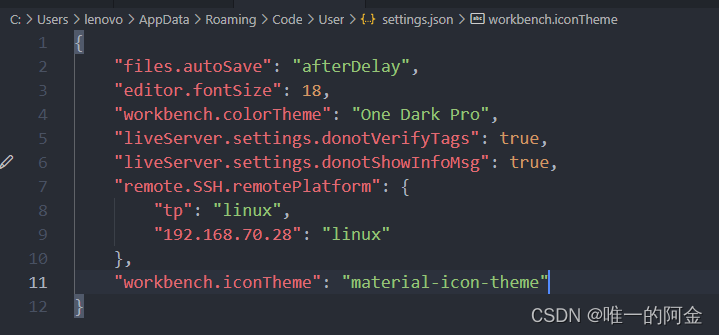
**第二步：配置VScode中的git。**

1. 打开VScode，依次点击下图中的地方



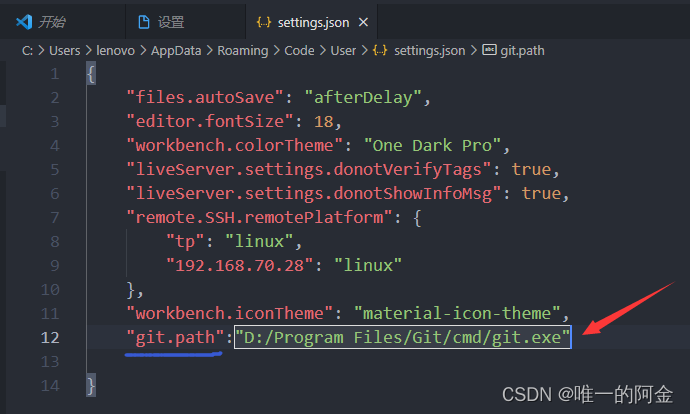
2. 在输入框中输入 git.path ，再点击第二个地方，打开 settings.json 文件





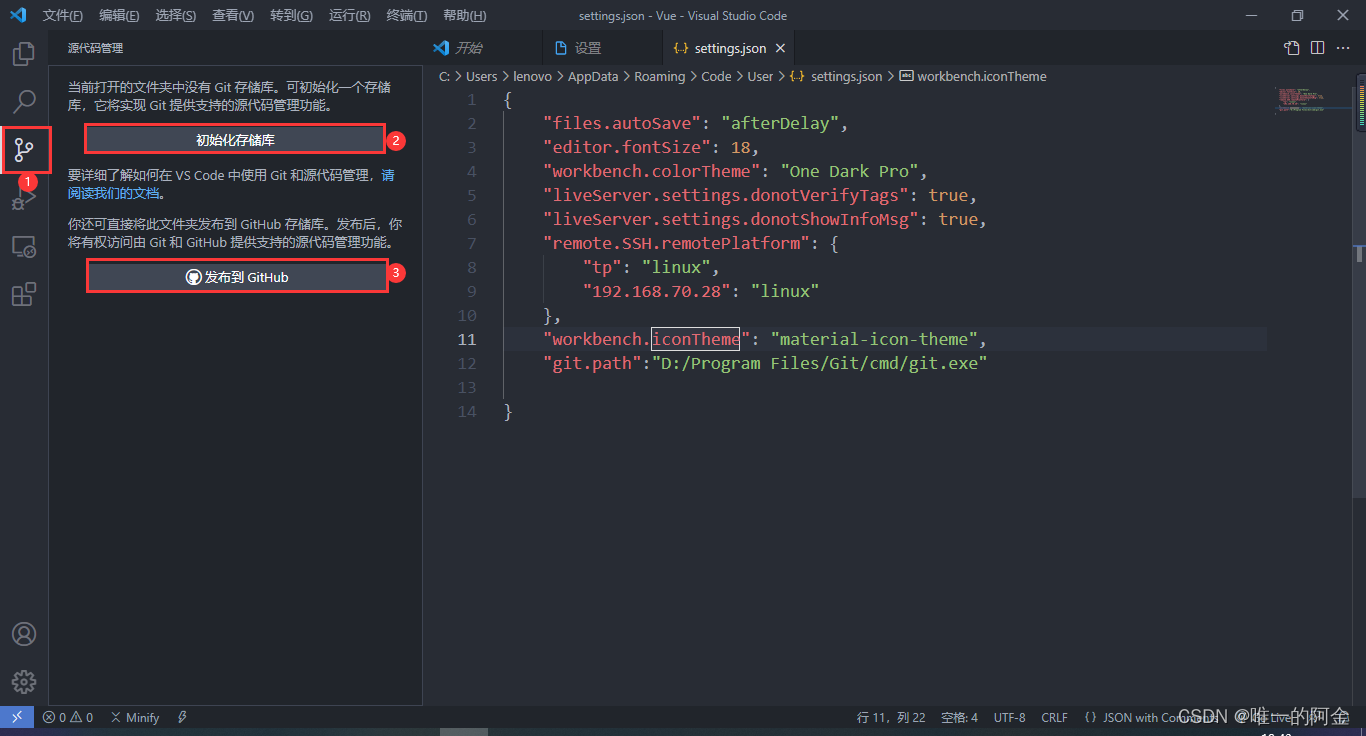
3. 在 settings.json 文件中，添加 git 的安装路径。





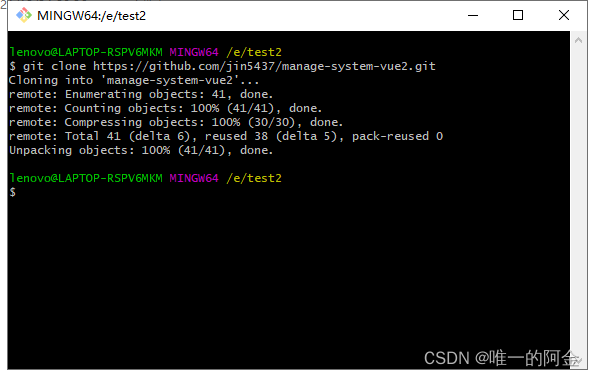
注意：冒号前面是 git.path ，路径是 cmd 文件夹中的 git.exe。

然后关闭 VScode ，再次打开，过几秒钟后，点击左侧的第3个图标，也就是下图的第一个地方，发现第2和第3个地方变成了可点击状态不是暗色的，表面git环境配置成功。

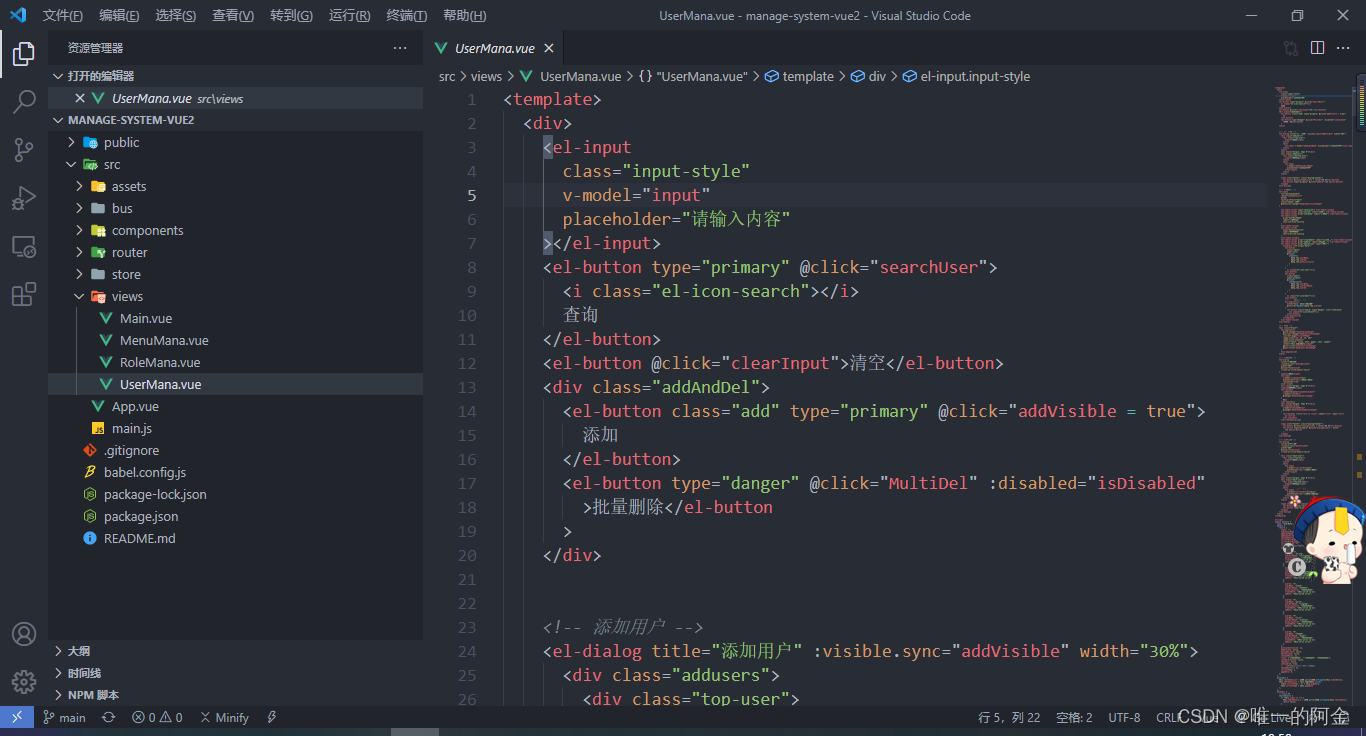


**第三步：使用 VScode + git，提交到仓库。**

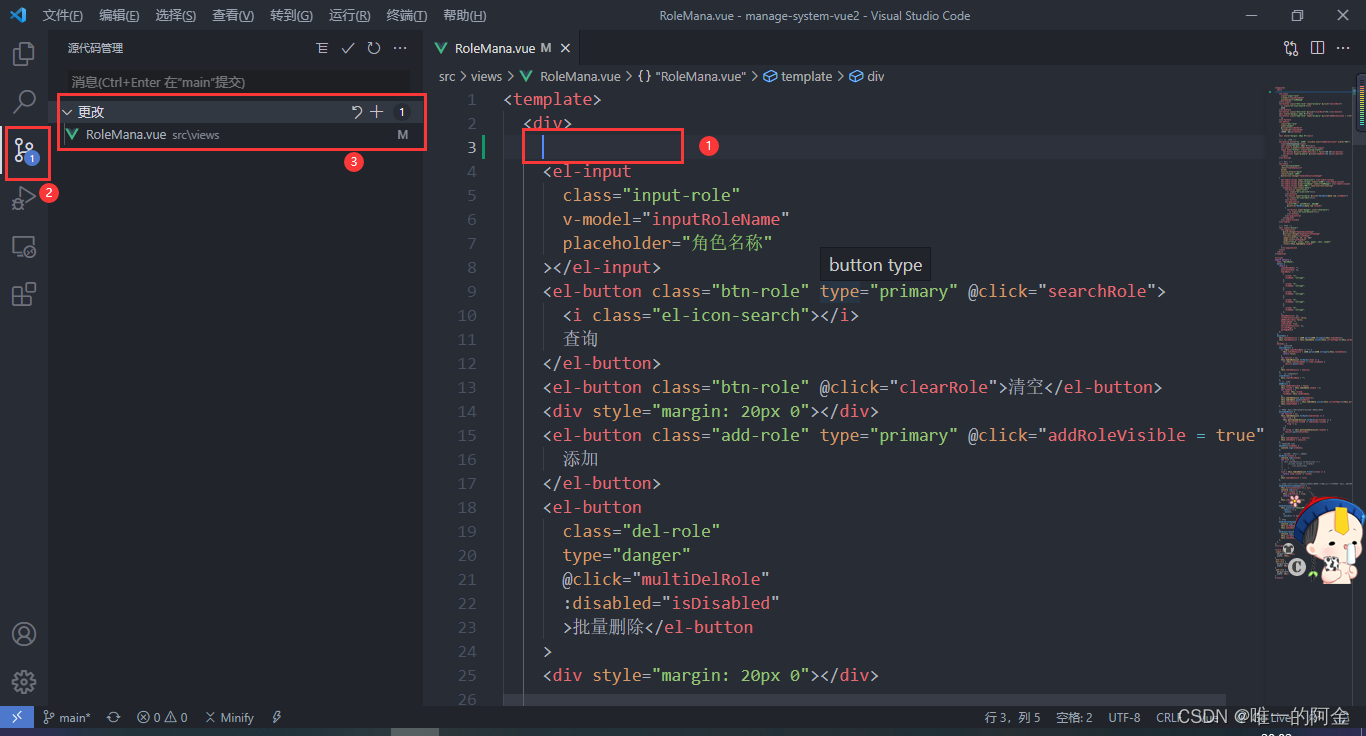
先从 GitHub 上面克隆下来一个仓库。



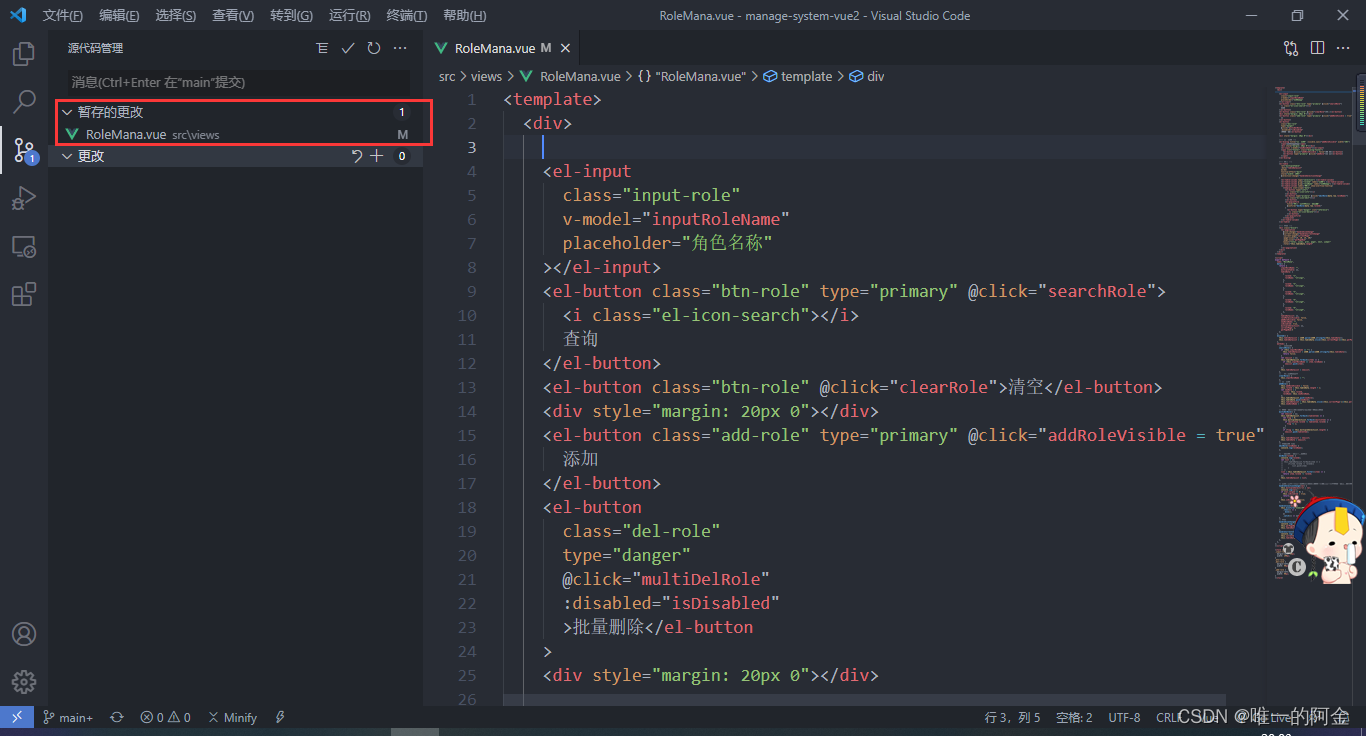
使用 VScode ，打开 clone 下来的文件。



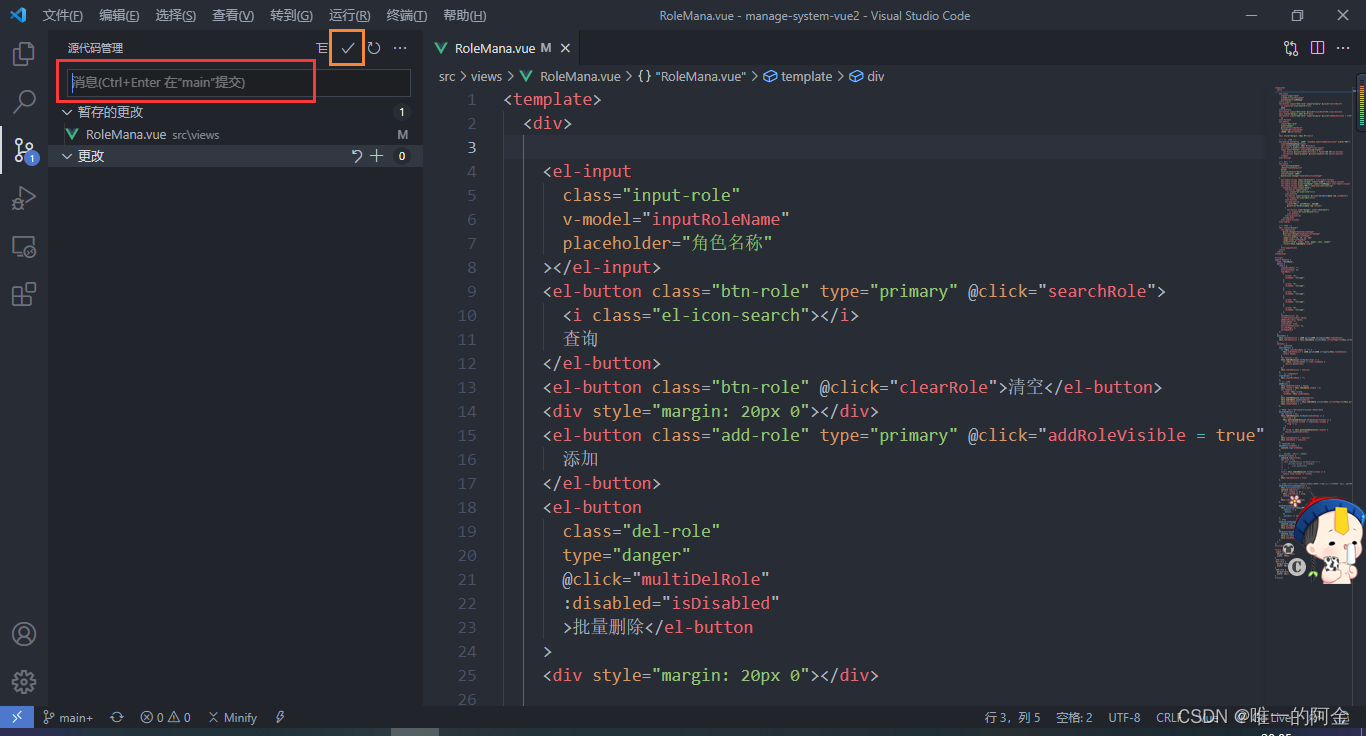
当我们修改代码时，左侧会提示我们哪个文件发生了改动。



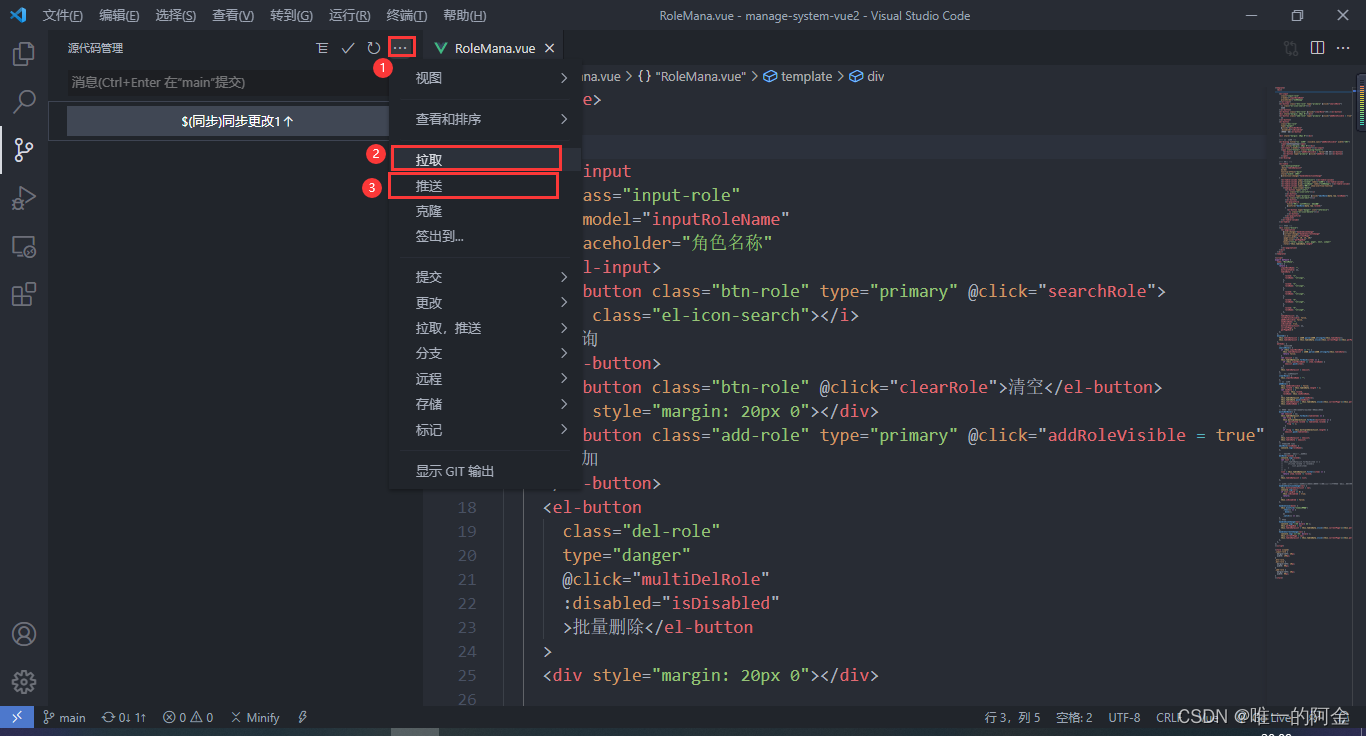
点击上图的第3处表示add命令的+号，如下图所示文件被放在暂存区。



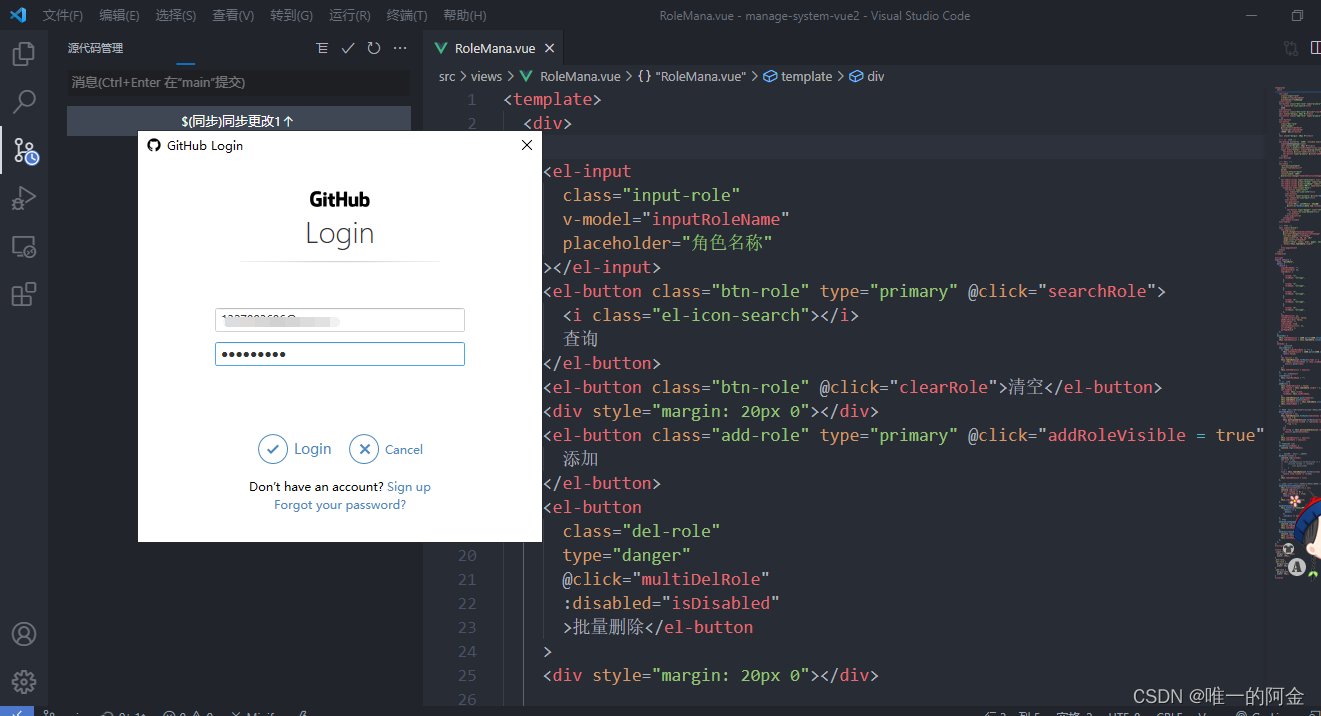
在下图的输入框中，输入要提交的备注信息，如 again test ，然后，可以按 Ctrl + enter ，或者点击橙色框中的对号，就是完成了commit命令。

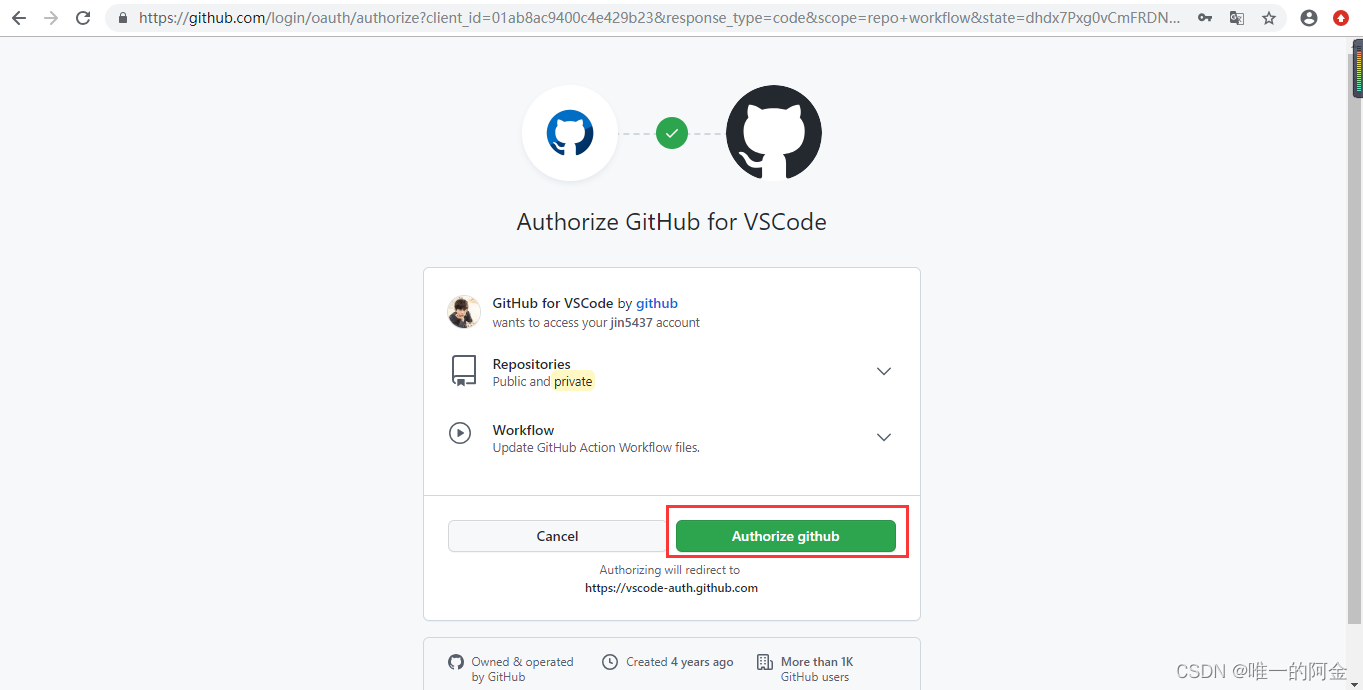


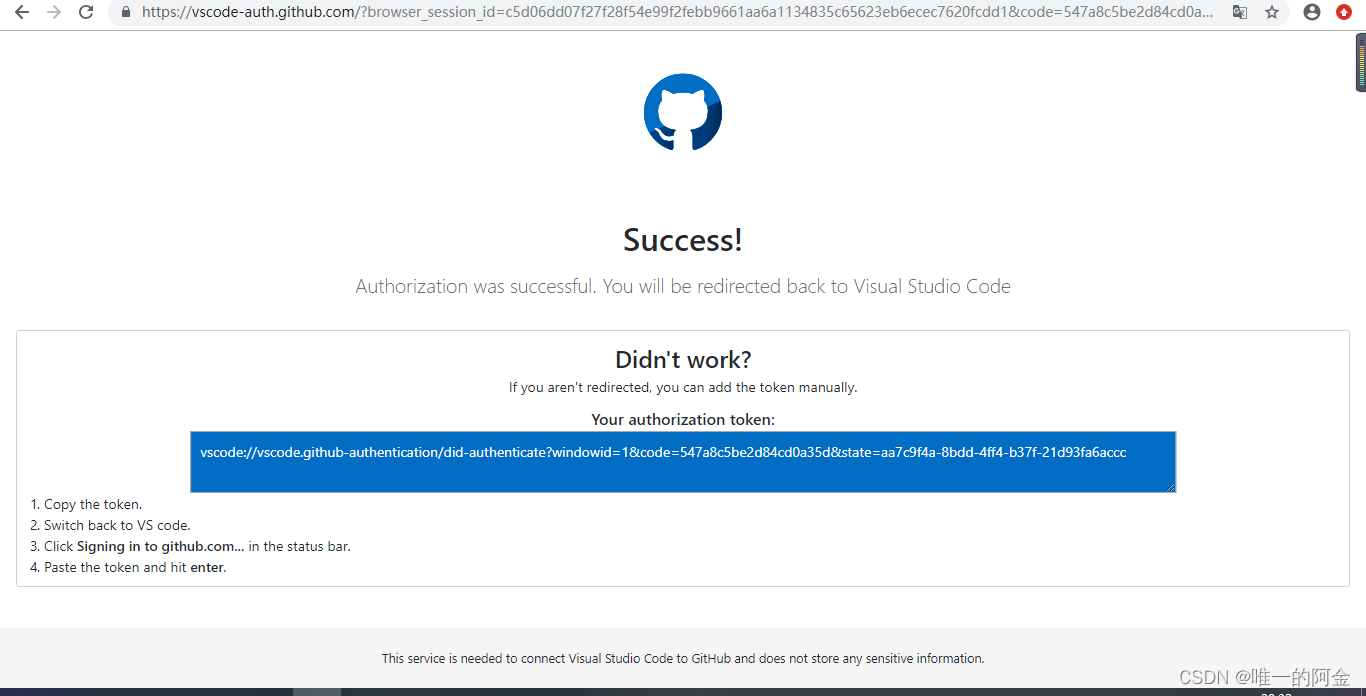
接着，点击下图的第一个地方，依次点击拉取、推送，即 pull 命令和 push 命令。



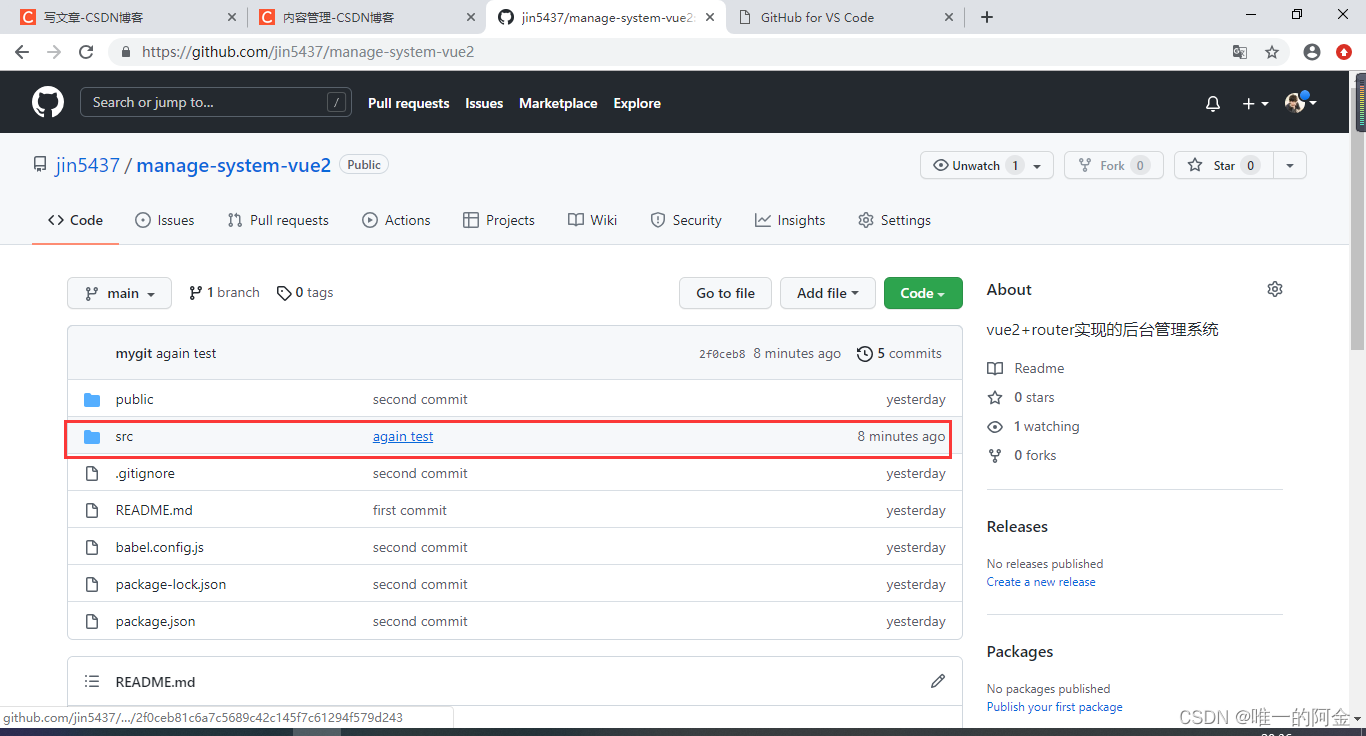
命令完成后，VScode 对GitHub 账号进行了认证，一步一步进行操作即可。







去 GitHub 官网，查看仓库，就能发现成功推送上去了！



至此，大功告成！