

## 第十章 版本有点乱？用 Git

某某项目(final-version).zip、某某项目(final-final-version).zip、某某项目(final-final-打死不改-version).zip、某某项目(final-final-打死不改-final-version-2021-3-21).zip，哭笑不得。我想很多人都干过这事，别问我怎么知道的。使用标识或者日期来对研发项目进行版本管理，是一种形同虚设的方案。在这种管理方式下，最终的结果就是产生一大堆的带有各种标识的文件备份。凭借这些标识根本无从进行版本追溯。但值得肯定的是，大家都有基本的版本管理意识和需求，只是缺少一个实用有效的方法或工具。

万物皆有迭代，有迭代就有版本，有版本就有 Git。早日使用 Git，早日脱离苦海。

### 1、关于 Git

在 2015 年之前可以说我的版本管理是一团糟。我被动的使用过 svn、soucesafe 这些版本控制工具，但是都不得其法。根本原因是我并不知道理想的版本管理应该是怎样的，直到我上手 Git。

Git，我们可以单独写出一部书来，因为它足够博大精深，甚至已经成为了版本管理的实际标准。

#### 1.1 Git 的前世今生

Git 已经成为现在最优秀的分布式版本管理工具，没有之一。它的管理理念到现在仍然是很先进的。说起 Git 的起源，其实还是有些无奈的，可以说它是另一个大规模软件项目的副产品。

Linux 大家都知道，它的作者是世界上最伟大的程序员--Linus Torvalds（林纳斯·托瓦兹）。1969 年，Linus Torvalds 生于芬兰一个知识分子家庭。1988 年，Linus 进入赫尔辛基大学计算机科学系就读。芬兰人性格内敛，这与 Linus 的行事方式不谋而合，他对开源的信念是近乎执着的。在兴趣的驱使下，Linus 创造并发布了自制的开源操作系统，取名为 Linux。有人问过他，为什么叫 Linux。他回答：我是个任性的杂种，我把所有我做的项目以我自己命名。看来程序员是偏执自恋而可爱的，连通神的 Linus Torvalds 也不例外。

Linux 是一个非常宏大的软件项目，单靠 Linus 一个人是不可能完成的。开源软件的核心要意就是集思广益，团队协作，你在享用别人的代码的同时，也要为它创造贡献。在 2002 年以前，Linux 的维护研发是由世界各地的程序员共同参与的，他们写出来的代码全部都交给 Linus 去合并（这个工作量可想而知）。2002 年以后，经过十多年的发展参与的人越来越多，而一个人合并难以避免的就是效率低，这也直接引起了维护者们的不满。难道没有工具可以实现代码的自动合并吗？当然是有的。当时已经存在一些版本控制工具了，像 cvs、svn 等，但是这些工具都是要收费的，而且使用的还是集中式版本管理方式。这就受到了 Linus 的唾弃（他坚定的认为，软件应该是免费开源的）。

后来 Linus 选择了 BitKeeper 分布式版本控制工具（BK）来作为 Linux 的版本管理工具，这个工具的研发公司 BitMover 也是出于人道博爱的精神给他们免费使用了。但是 Linux 社区的很多贡献者对 BK 非常不满，原因是它不开源。既有怨气，必有勇士。一位叫 Andrew Tridgell 的程序员违反 BK 的使用原则，对其进行了逆向工程，写了一个可以连接 BK 仓库的外挂。BitMover 认为他反编译了 BK。Linus 花了很多时间精力从中协调磋商，但是最终还是失败了。2005 年，BitMover 同 Linux 内核开源社区的合作关系结束。

Linus 一怒之下，决定自己造车轮。他基于使用 BK 时的经验教训，仅花了 2 周就开发出了自己的版本管理系统，也就是后来的 Git。Linus 怒而不乱，其实他早有此意并对市面上多个版本管理方案进行过评估。他提出了极具前瞻性的三个诉求：可靠性、高效、分布式。后来，这三个特性被视为 git 的核心灵魂所在，深远的影响了 git 及其他同类软件的后续发

展。

## 1.2 Git 的爆发

伟大的软件一定是很好地解决了行业内长期饱受诟病的一些重大问题和痛点。在 Git 问世之前很多的版本管理软件都采用服务器集中式管理方式。如图 10.1 所示。

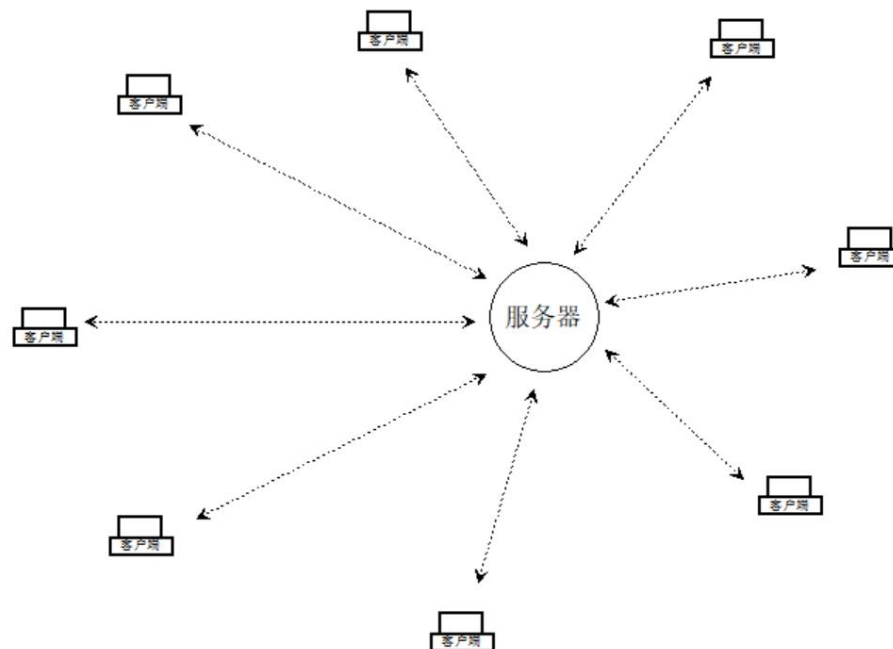


图 10.1 服务器集中式的版本管理方式

在这种管理方式下，程序员每次进行开发前，都要先从服务器拉取版本，在开发完成之后，再将它推回到服务器。这带来两个问题 1、开发用的电脑必须联网；2、因为代码都存在远端服务器上，一旦服务器出现问题都是灾难性的，程序员的工作可能付诸东流。

Git 反其道而行之，它采用分布式的版本管理方式（Linux 起初选择 BK 也是因为它是分布式的）。分布式的主要思想是去中心化和本地化。程序员可以从服务器上拉取项目的完整仓库到本地，然后以离线的方式进行本地化的开发和提交。用 Linux 的话说：你可以在本地作很多事情，而完全不依赖于服务器和网络。而且本地化的管理，使得类似于 commit、版本回滚等操作都变得非常快速（集中式的版本管理所有操作都是直接与服务器进行远程访问的，所以总是要等待服务器的回应，这造成它行动缓慢，效率不高）。

Git 成功地替代了 BK，成为 Linux 的版本控制的原生方案，但它仍然只不过是服务于局域网人群的一个工具而已。它如星星之火，要燎原还差一场风暴。要得到行业内普遍认同和接受是任重道远的。这个时候就不得不引出一个伟大的网站，是它最终成就了 Git，即 GitHub。这背后是三个年轻人创业的故事。

2007 年旧金山三个年轻人觉得 Git 是个好东西，就搞了一个公司名字叫 GitHub，第二年上线了使用 Ruby 编写的同名网站 GitHub，这是一个基于 Git 的自由代码托管网站（有付费服务）。十年间，该网站迅速蹿红，击败了实力雄厚的 Google Code，成为全世界最受欢迎的代码托管网站。2018 年 6 月，GitHub 被财大气粗的 Microsoft 收购。2019 年 1 月 GitHub 宣布用户可以免费创建私有仓库。根据 2018 年 10 月的 GitHub 年度报告显示，目前有 3100 万开发者创建了 9600 万个项目仓库，有 210 万企业入驻。

相比 Git，github 提供了更多的功能，比如 Web 管理界面、评论、组织、点赞、关注、图表，俨然已经是一个社交网站了，大家围绕着开源项目进行使用、讨论和贡献等。

关于 GitHub 的历史和里程碑大家可以去百度一下，这里就不赘述了。

GitHub 是世界上最大的开源代码仓库，这是程序员的天堂。在这里，你可以站在无数高手的肩膀上，高效而高质量的完成自己的开发。在你打开 [www.github.com](https://github.com)（如图 10.2）的一瞬间，你已经是开源主义军团中的一名战士了。

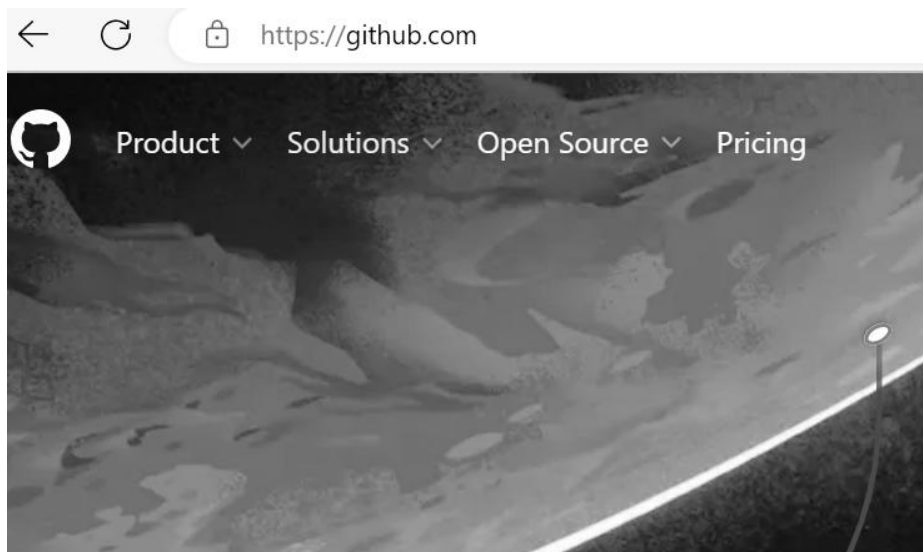


图 10.2 GitHub 网站一瞥

## 2、Git 的本地化使用

当你在开发一个完全独立的，不需要公开或多人协作的项目时，就可以使用 Git 的本地化仓库。下面振南举例进行说明。

安装好 Git 之后，在要管理的代码工程目录下右键，选择 Git Bash，如图 10.3。

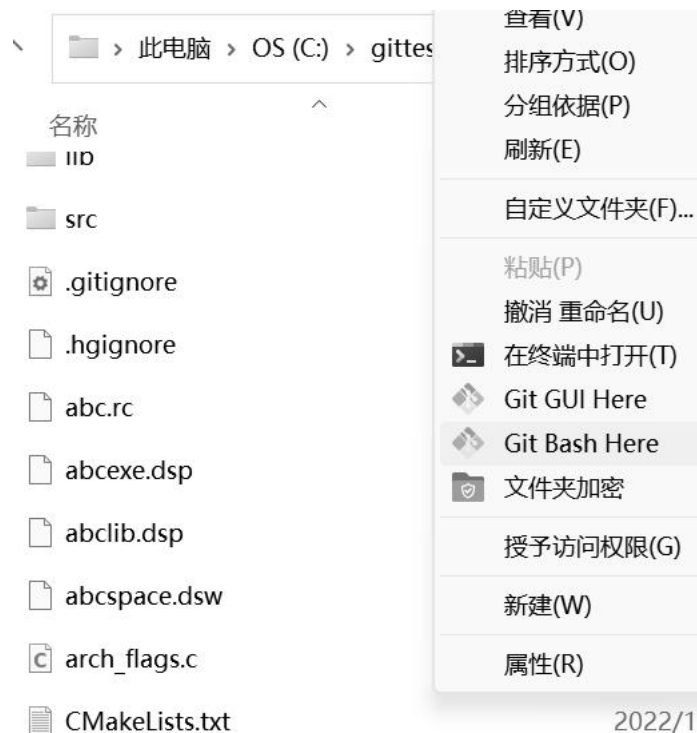


图 10.3 代码工程目录下右键选择 Git Bash

然后 `git init`，这样就创建了一个本地的仓库。对，就是这么简单。创建成功后，可以看到一个名为 `.git` 的目录，如图 10.4。

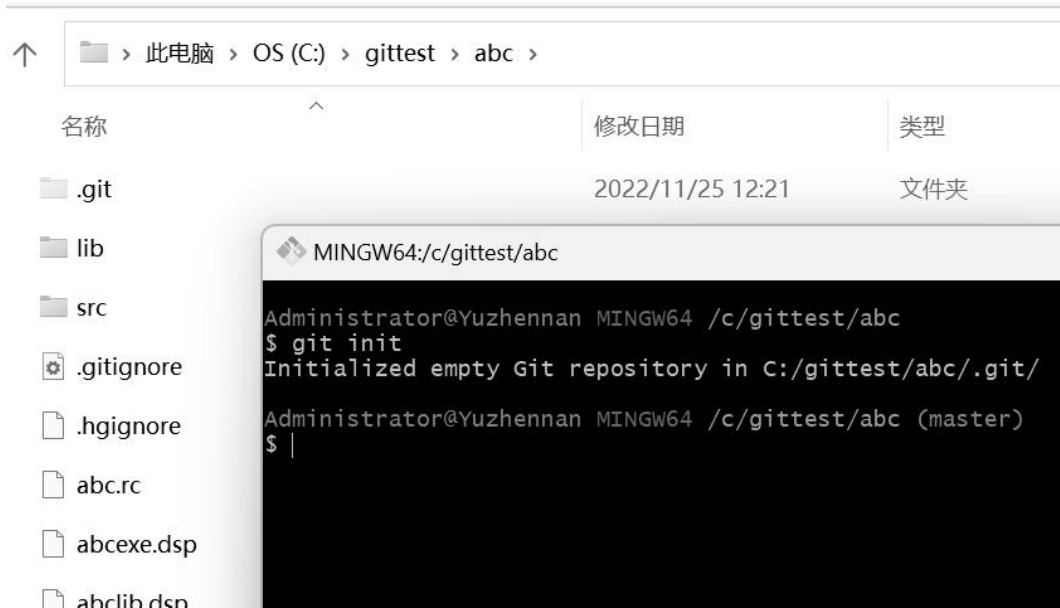


图 10.4 创建本地的 `git` 仓库

从图中我们可以看到（`master`），这是当前仓库所处的分支。分支（`Branch`）是 `Git` 的一个重要概念。一个仓库可能会有很多分支，其中有一个分支为默认主分支，一般主分支名称为 `master` 或 `main`。分支可以被创建、拷贝或删除，而各分支之间可以合并。这些是 `Git` 最基本的一些操作，请大家深入去理解。

OK，我们现在创建了一个空仓库，而且它有一个主分支 `master`。如图 10.5 所示。

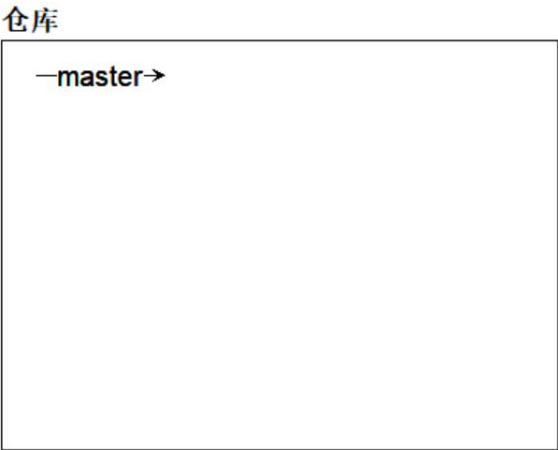


图 10.5 一个只有一个主分支 `master` 的空仓库

对，它就是这么空空如野。接下来我们把要进行版本管理的文件添加到仓库中，使用 `git add` 命令，如图 10.6。

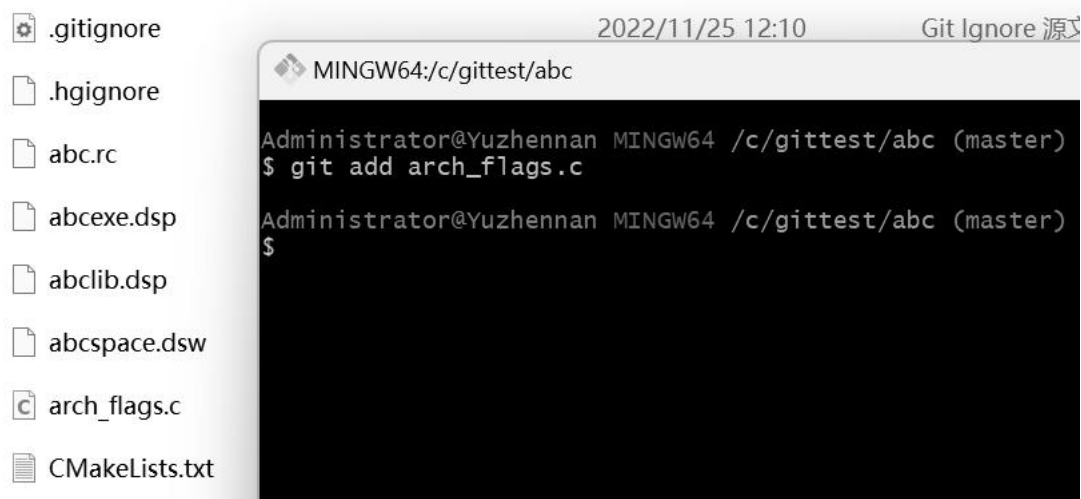


图 10.6 向仓库中添加文件以对其进行版本管理

只有被添加到仓库里来的文件，才能进入到 git 的版本管理体系中来。一个项目的文件可能会非常多，难道要一个个去 add 吗？如果真是这样，那 Git 就不会有今天的辉煌了。直接使用 `git add .` 即可。但是这样又出现一个问题，我可能并不想把所有文件都添加到仓库中，比如一些编译的中间文件 `.obj`、`.o` 等，因为对这些文件进行版本管理毫无意义，而且还会使仓库越来越臃肿。为什么会越来越臃肿？往后看。

为了解决这个问题，git 提供了 `.gitignore` 这个文件，我们可以把不想加入到仓库中的文件写到此文件中，比如 `*.obj`。这样我们执行 `git add .` 的时候，git 就会自动为我们忽略这些文件。

OK，现在我们将这个目录下的所有文件都加入到仓库中，如图 10.7。

```
Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git add .
```

图 10.7 向仓库中添加所有文件

如果文件比较多，这个操作可能会比较花时间。

接下来，我们来尝试进行第一次提交 `commit`，如图 10.8 所示。

```
Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git commit -m"first commit"
Author identity unknown

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'Administrator@Y
```

图 10.8 尝试进行第一次提交

git 提示 “Please tell me who you are.” 好吧，那我们利用提示中的 `git config` 命令来设置账

户邮箱和用户名，如图 10.9 所示。

```
Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git config --global user.email "987582714@qq.com"

Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git config --global user.name "yuzhennan"
```

图 10.9 设置帐户邮箱和用户名

再次尝试进行 commit，如图 10.10。

```
Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git commit -m"first commit"
[master (root-commit) a637962] first commit
1800 files changed, 1004764 insertions(+)
create mode 100644 .gitignore
create mode 100644 .hgignore
create mode 100644 CMakeLists.txt
create mode 100644 Makefile
create mode 100644 README.md
create mode 100644 abc.rc
create mode 100644 abcxex.dsp
create mode 100644 abclib.dsp
create mode 100644 abcspace.dsw
create mode 100644 arch_flags.c
create mode 100644 copyright.txt
create mode 100644 depends.sh
create mode 100644 i10.aig
create mode 100644 lib/pthread.h
create mode 100644 lib/sched.h
```

图 10.10 对代码进行提交

可以看到 git 罗列出了我们前面 add 的所有文件，这说明这些文件确实已经进入到 git 的管理体系中了。在提交的时候，可以通过 -m 来添加一些注释，来对此次提交进行一些必要的描述。

我们可以使用 git log 来查看当前分支曾经的提交历史，如图 10.11 所示。

```
Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git log
commit a637962d420b4bbdb5af51a79576110bea6e3ad (HEAD -> master)
Author: yuzhennan <987582714@qq.com>
Date: Fri Nov 25 14:00:38 2022 +0800

    first commit
```

图 10.11 通过 git log 查看当前分支的提交历史

好吧，我们只提交过一次。

接下来，我们对文件作一些修改（把 arch\_flags.c 文件内容清空），如图 10.12。

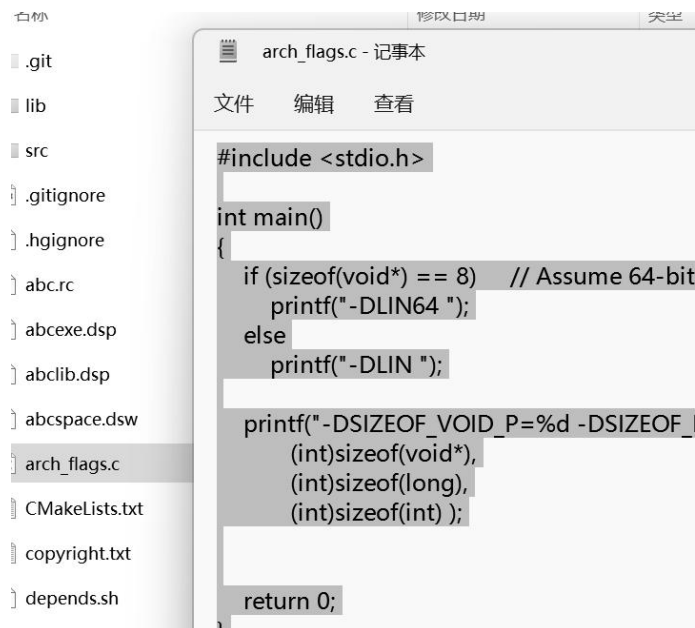


图 10.12 对文件作一些修改

然后再提交一次，如图 10.13。

```
Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git add .

Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git commit -m"arch_flags.c clear"
[master b6d1b42] arch_flags.c clear
1 file changed, 17 deletions(-)

Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git log
commit b6d1b42a1534bcc0a08e8b87039c71c4364bc3a3 (HEAD -> master)
Author: yuzhennan <987582714@qq.com>
Date:   Fri Nov 25 15:29:26 2022 +0800

    arch_flags.c clear

commit a637962d420b4bbdb5af51a79576110bea6e3ad
Author: yuzhennan <987582714@qq.com>
Date:   Fri Nov 25 14:00:38 2022 +0800

    first commit

Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$
```

图 10.13 对代码再一次提交

此时，如果我们想看一下上一个版本的代码，该如何操作？仔细观察每一次 commit，git 都会生产一个 commit-id（40 个字符），通过它我们可以进入任何一次提交，去查看当时的代码。如图 10.14。

```

Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git checkout a637962d420b4bbdb5af51a79576110bea6e3ad
Note: switching to 'a637962d420b4bbdb5af51a79576110bea6e3ad'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

    git switch -c <new-branch-name>

Or undo this operation with:

    git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at a637962 first commit
Administrator@Yuzhennan MINGW64 /c/gittest/abc ((a637962...))
$ |

```

图 10.14 使用 git checkout 进入到某一次提交

此时，我们再去看一下刚才修改的 arch\_flags.c 这个文件。如图 10.15 所示。

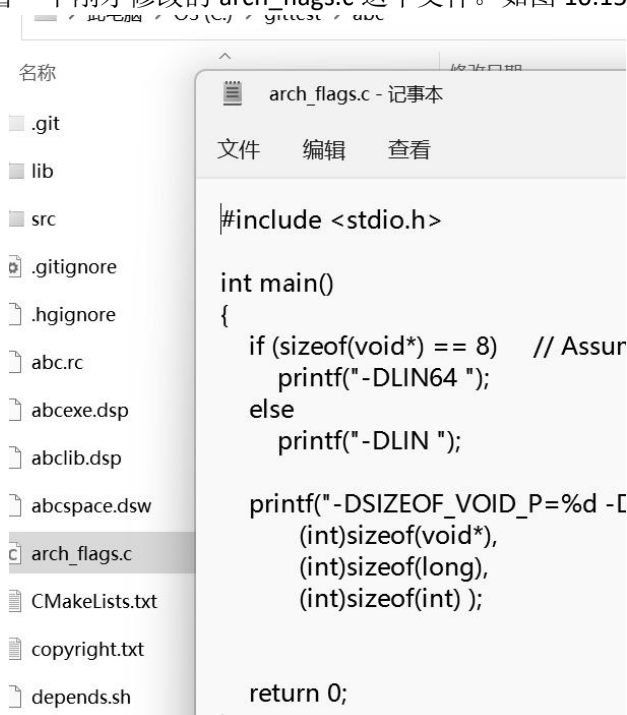


图 10.15 切换 commit 之后 arch\_flags.c 文件恢复了原来的内容

可以看到，arch\_flag.c 文件又恢复了原来的内容，是不是很神奇？这就是 git 为我们带来的版本管理强大功能的冰山一角。

进入某个 commit 后，可以查看代码，但是并不能修改它，因为每一次 commit 都是一个固定的版本。那如何基于某一个中间 commit 进行后续开发呢？那我先要问问你为什么会有这种自废武功的操作？你理直气壮的说：“因为我后悔了，我对这个 commit 之后的代码开发不满意，我希望回去重新来！” OK，git 给你后悔药。

我们让代码回滚，如图 10.16。



```

Administrator@Yuzhennan MINGW64 /c/gittest/abc ((a637962...))
$ git checkout master
Previous HEAD position was a637962 first commit
Switched to branch 'master'

Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git log
commit b6d1b42a1534bcc0a08e8b87039c71c4364bc3a3 (HEAD -> master)
Author: yuzhennan <987582714@qq.com>
Date:   Fri Nov 25 15:29:26 2022 +0800

    arch_flags.c clear

commit a637962d420b4bbdb5af51a79576110bea6e3ad
Author: yuzhennan <987582714@qq.com>
Date:   Fri Nov 25 14:00:38 2022 +0800

    first commit

Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git reset --hard a637962d420b4bbdb5af51a79576110bea6e3ad
HEAD is now at a637962 first commit

Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git log
commit a637962d420b4bbdb5af51a79576110bea6e3ad (HEAD -> master)
Author: yuzhennan <987582714@qq.com>
Date:   Fri Nov 25 14:00:38 2022 +0800

    first commit

Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$

```

图 10.16 将代码回滚到某一个 commit

上图中，先从 commit 的临时分支切回到 master 分支，然后使用 git reset 命令将版本回滚到某个 commit 上。最后，再次 git log 就会发现，第二次提交已经消失了。我们这个时候就可以开始在这颗“后悔药”上继续开发了。

但是你又怎么保证你不会后悔吃了后悔药？有点作，无作无 Die，想好再干。OK，git 满足你。如图 10.17、10.18 所示。

```

Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git log
commit dbdf5d5cc2926781bc0532f0b21371e3c46c1fd5 (HEAD -> master)
Author: yuzhennan <987582714@qq.com>
Date:   Fri Nov 25 16:12:35 2022 +0800

    arch_flags.c clear

commit a637962d420b4bbdb5af51a79576110bea6e3ad
Author: yuzhennan <987582714@qq.com>
Date:   Fri Nov 25 14:00:38 2022 +0800

    first commit

Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git checkout a637962d420b4bbdb5af51a79576110bea6e3ad
Note: switching to 'a637962d420b4bbdb5af51a79576110bea6e3ad'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

    git switch -c <new-branch-name>

Or undo this operation with:

    git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at a637962 first commit

Administrator@Yuzhennan MINGW64 /c/gittest/abc ((a637962...))
$ git checkout -b test
Switched to a new branch 'test'

Administrator@Yuzhennan MINGW64 /c/gittest/abc (test)
$ |

```

图 10.17 从 master 分支的某个 commit 开出一个新的分支 test

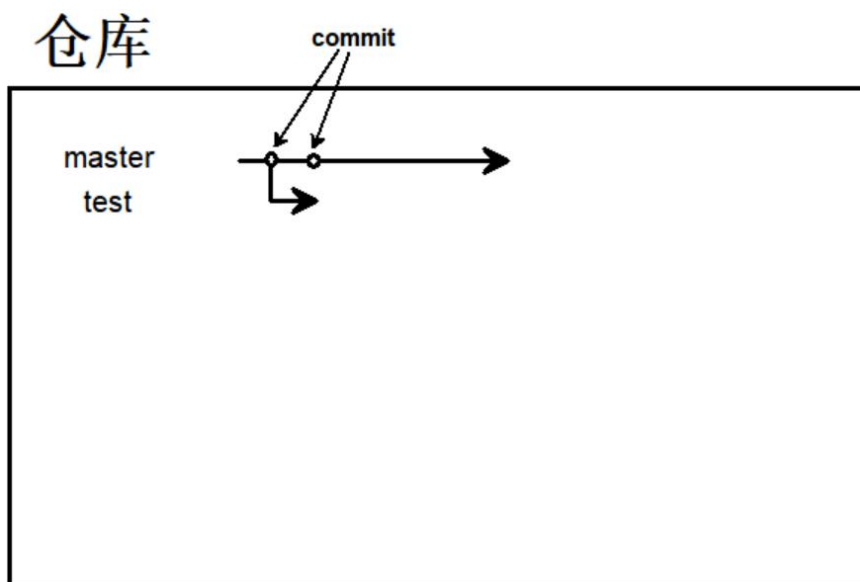


图 10.18 从 master 分支的某个 commit 开出一个新的分支 test (示意图)

我们可以从 master 分支的某个 commit 开出一个新的分支, 然后在这个新的分支上继续

开发。最后再合并到 master 上。如图 10.19、10.20、10.21 所示。

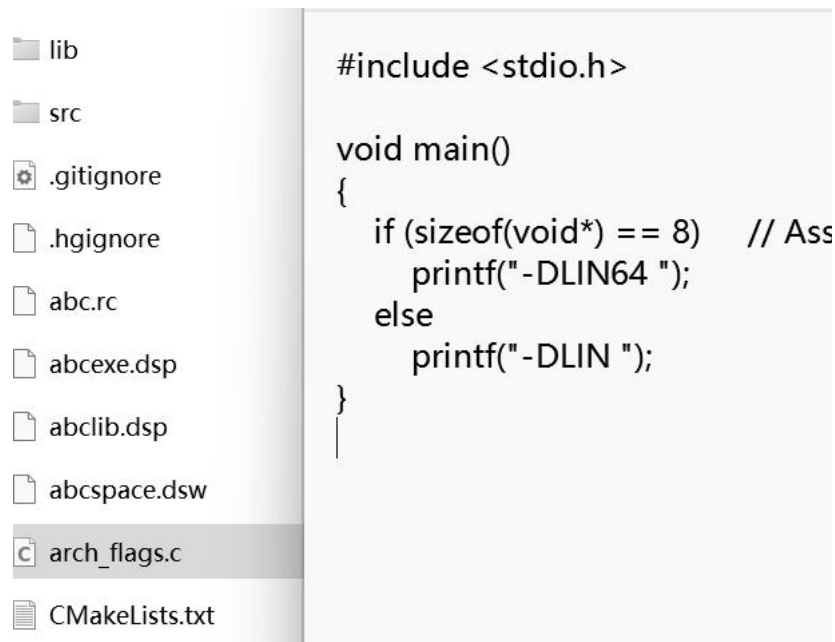


图 10.19 在 test 分支上对 arch\_flags.c 文件作一些修改

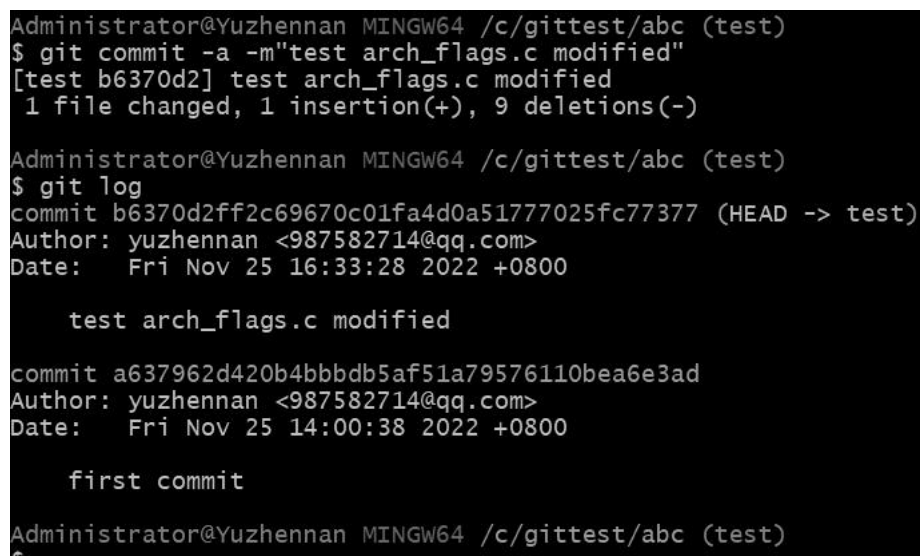


图 10.20 在 test 分支上对代码进行提交

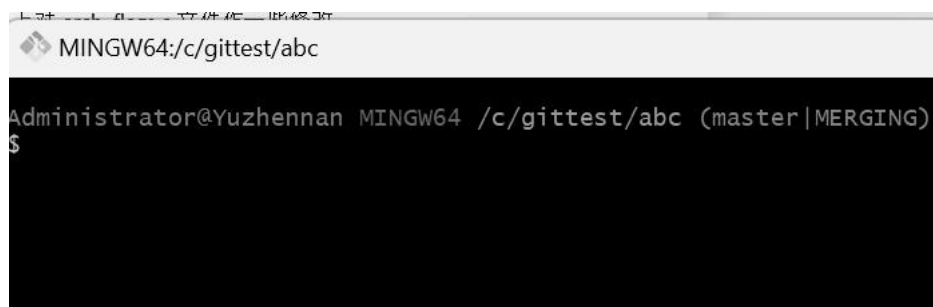


图 10.21 将 test 分支合并到 master 分支上

图 10.21 因为清屏的问题，没有截到图。过程是先 checkout master，然后 git merge test。

这个时候会提示 `arch_flags.c` 有冲突，并且自动处理冲突失败（需要手动处理），同时标识变成了（`master|MERGING`），说明当前分支正在进行合并。

有人会问：“冲突是怎样产生的？冲突是什么样的？”原则上来说，在两个分支的同一个文件中，同一行的内容不同，那么就会产生冲突，而且 `git` 并不能自动处理，因为它根本不知道该舍谁留谁。

冲突的解决通常需要借助于一些工具，比如 `kdiff3` 等。我一直在使用 `VScode`，我也建议大家使用它，因为它的功能实在是太强大了。如图 10.22 所示。

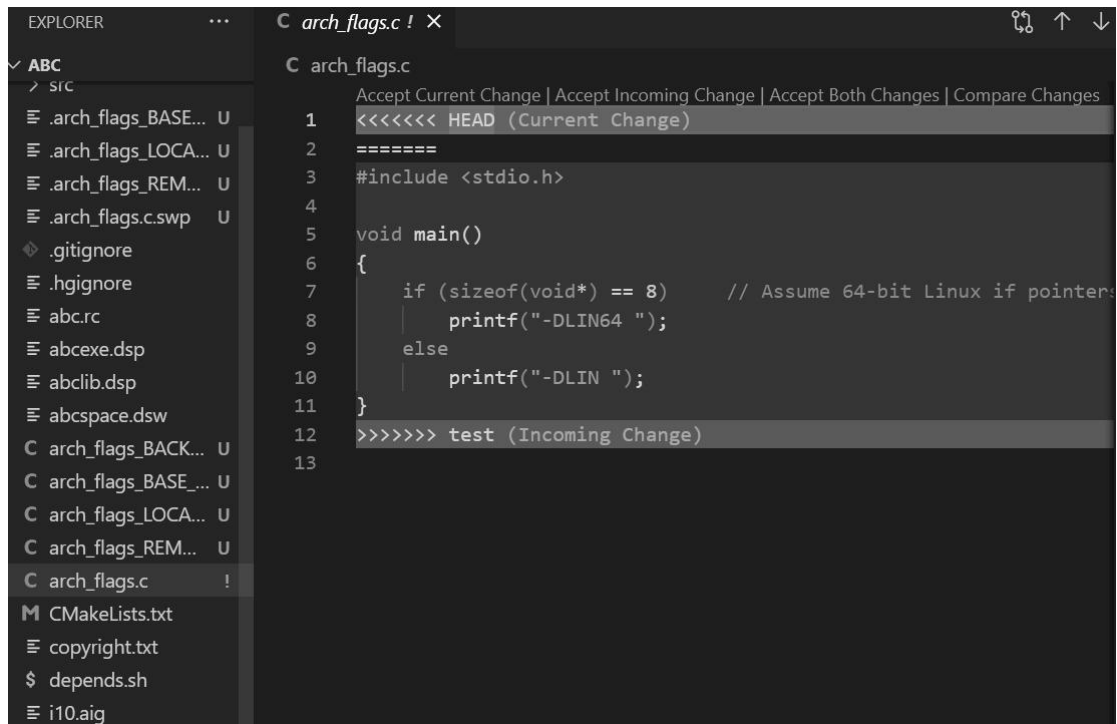


图 10.22 使用 VScode 对冲突进行解决

上面说，冲突的本质是一个去留的问题。仔细观察上图，会发现代码中有一个分割线 `=====`，它上面是 `master` 分支当前这一行的内容，下面则是合并的源分支，即 `test` 分支此行的内容。一个称为 `Current Change`，另一个称为 `Incoming Change`。你需要在这两者这章作出选择。在冲突的顶端有几个选项，`Accept Current Change` 和 `Accept Incoming Change`，我们选择后者。在解决了冲突之后，我们对 `master` 分支进行一次 `commit`，标识中的 `MERGING` 就消失了。如图 10.23 所示。

```

Administrator@Yuzhennan MINGW64 /c/gittest/abc (master|MERGING)
$ git add .

Administrator@Yuzhennan MINGW64 /c/gittest/abc (master|MERGING)
$ git commit -m"merged from test"
[master 609160b] merged from test

Administrator@Yuzhennan MINGW64 /c/gittest/abc (master)
$ git log
commit 609160b54feef0c5304e5e86fadbbc3b1fa709d5 (HEAD -> master)
Merge: dbdf5d5 b6370d2
Author: yuzhennan <987582714@qq.com>
Date:   Fri Nov 25 17:18:57 2022 +0800

    merged from test

commit b6370d2ff2c69670c01fa4d0a51777025fc77377 (test)
Author: yuzhennan <987582714@qq.com>
Date:   Fri Nov 25 16:33:28 2022 +0800

    test arch_flags.c modified

```

图 10.23 对完成 merge 的 master 分支进行 commit

此时，似乎 `test` 分支已经没有存在的必要了，我们可以将它删除，使用命令 `git branch --delete test`。

当然，如果我们发现在 `test` 分支上干不下去了，又想回到 `master` 分支，那么你可以直接干掉 `test` 分支，回归 `master`。

以上一整套的操作其实影射出几个问题：

- 1、不要輕易的删除分支或回滚，以及其它可能造成数据丢失的行为，三思而行（虽然在 `git` 的体系下，并不会真正的造成丢失）；
- 2、你应该有一个主分支，比如 `master` 或 `main`，并且秉持严肃的态度，不轻易地直接对其进行改动，并保证主分支上的代码是相对成熟的；
- 3、每开一个子分支一定要知道为什么开它，以及它的使命是什么？原则上来说，一切子分支都应该为主分支服务。

以上振南只是对 `Git` 的本地化操作的一些皮毛进行了介绍，我想已经足够大家应付一般的情况了。

### 3、Git 的远端使用

`Git` 的真正威力其实是其社区化的多人协作的模式。要协作就一定要公开你的代码，这就需要有一个代码托管服务器，并能够方便高效地在本地与远端服务器之间进行各种操作。每个公司可能都有自己的 `Git` 服务器，并且通常是不对外开放的，只有使用公司内网或 `VPN` 才能登录。而且有专人进行维护，还会有多重的数据备份。这对于阿里、百度这样典型的互联网公司来说，在管理上就更加谨慎和严格了。

振南公司的私有服务器是不可能拿来演示的，那就用 `GitHub` 吧。

我们先来尝试把前文那个本地仓库上传（`push`）到服务器。基本的流程是：在 `GitHub` 上创建一个仓库；将本地仓库与 `GitHub` 上的远端仓库建立关联；通过 `git` 命令操作远端仓库。

首先你要在 `GitHub` 上注册一个帐号，然后点击新建仓库。如图 10.24 与 10.25。

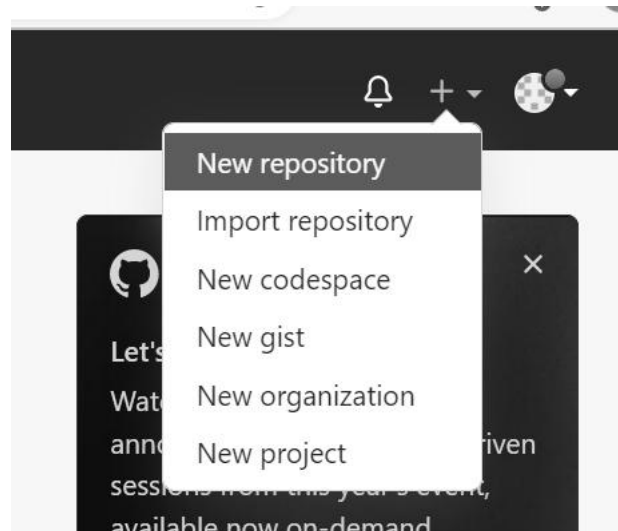



图 10.24 在 GitHub 上点击新建仓库

## Create a new repository

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

---

**Owner \*** **Repository name \***


 ZNelec / abc ✓


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

**Description (optional)**

abc

---

☒  **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:**  
Skip this step if you're importing an existing repository.

☐ **Add a README file**  
This is where you can write a long description for your project. [Learn more.](#)

**Add .gitignore**  
Choose which files not to track from a list of templates. [Learn more.](#)

.gitignore template: None ▼

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

License: None ▼

图 10.25 在 GitHub 上新建仓库

这样，我们就成功创建了一个远端的仓库，如图 10.26。

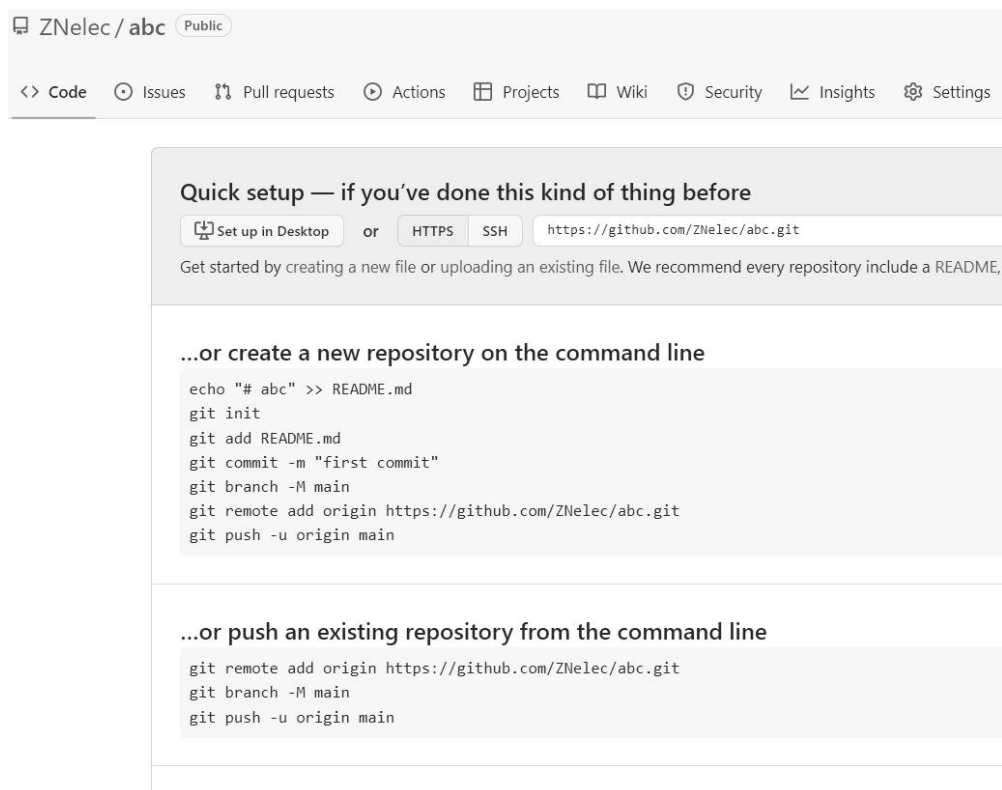


图 10.26 在 GitHub 上新创建的仓库

可以看到 GitHub 已经给出来操作方法。要注意的是，GitHub 默认使用 SSH 协议与本地仓库建立连接。但是设置 SSH 是比较麻烦的（涉及到 SSH-key 的生成和添加，后面会有介绍），所以我们使用 HTTPS。具体操作如图 10.27。



图 10.27 通过设备码进行认证从而将本地仓库推到 GitHub

OK，在填入了设备码之后，我的电脑认证通过，git 开始将本地仓库推向 GitHub。我们可以来看看 GitHub 上新建的仓库。如图 10.28。

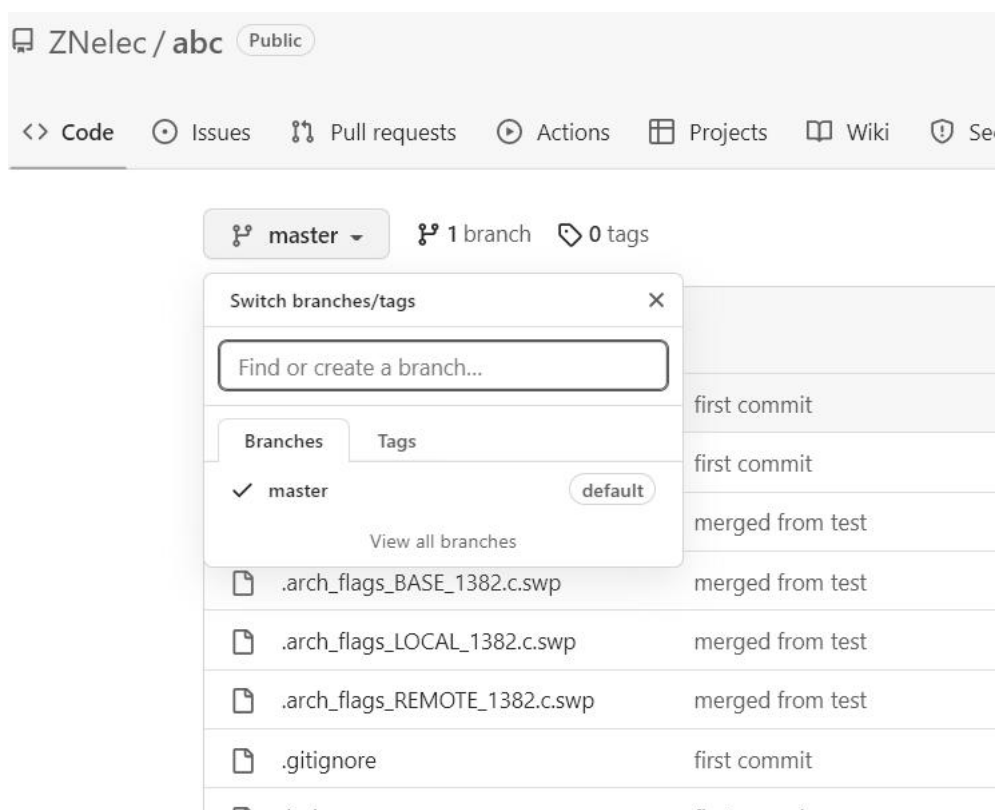


图 10.28 成功将本地仓库推到了远端

这里的讲解似乎与网上关于 Git 的使用有些不一样。我知道网上的很多教程都是以 SSH 为例来展开的，但其实 HTTPS 更加简单。

如果我们想去将别人在 GitHub 上的仓库同步到本地，又该如何操作呢？很简单，用 `git clone` 即可，而且这应该是使用频率最高的命令了。我们以 GitHub 上随便一个仓库为例，把它 Clone 下来。如图 10.29。

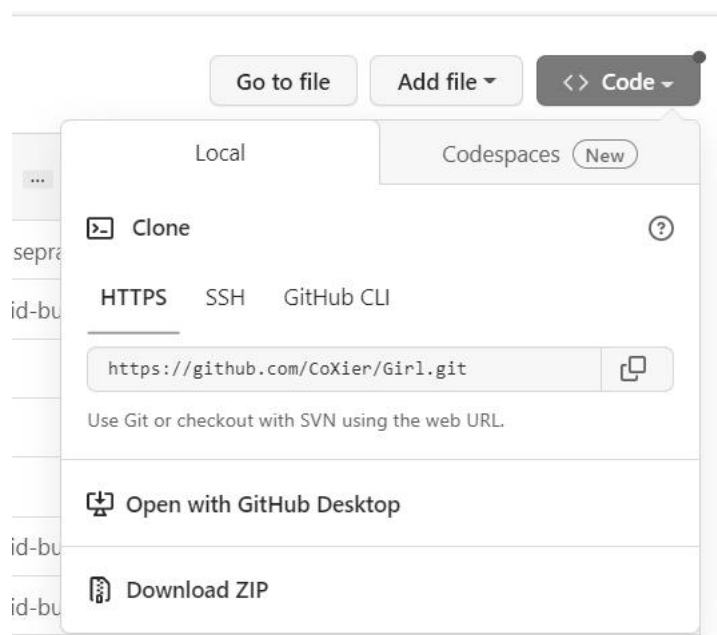


图 10.29 获取 GitHub 仓库的 HTTPS 链接



首先我们要获取 GitHub 仓库的 HTTPS 链接（如果你 Clone 的目的只是为了看一看，而不打算加入到这个仓库的维护中去，那直接 Download ZIP 就可以了），然后就可以开始 clone 了。如图 10.30。

```
Administrator@Yuzhennan MINGW64 /c/gittest/aaa
$ git clone https://github.com/Coxier/Girl.git
Cloning into 'Girl'...
remote: Enumerating objects: 1935, done.
remote: Total 1935 (delta 0), reused 0 (delta 0), pack-reused 1935
Receiving objects: 100% (1935/1935), 4.57 MiB | 2.23 MiB/s, done.
Resolving deltas: 100% (933/933), done.

Administrator@Yuzhennan MINGW64 /c/gittest/aaa
$ |
```

图 10.30 对 GitHub 上的仓库成功进行 Clone

OK，本章关于 Git 的介绍就到这里，希望大家早日上手。其实本文所讲的内容都是皮毛中的皮毛，Git 工具远比我们想像的要强大，GitHub 也远比我们想像的要浩瀚。关于更深层的应用，还要等着大家深入去研究发掘。

GitHub 是全世界程序员智慧的结晶，是一笔巨大的财富，是无数开源主义先驱努力奋斗的成果。在闭源软件圈地为王，Windows 等商业软件大行其道甚至垄断的历史背景下，开源者力排众议，自立更生，集结所有有生力量，共同构建了开源软件的庞大生态，这是伟大的，前无古人的！

向开源者致敬！！