一、Git简介

Git是目前世界上最先进的分布式版本控制系统

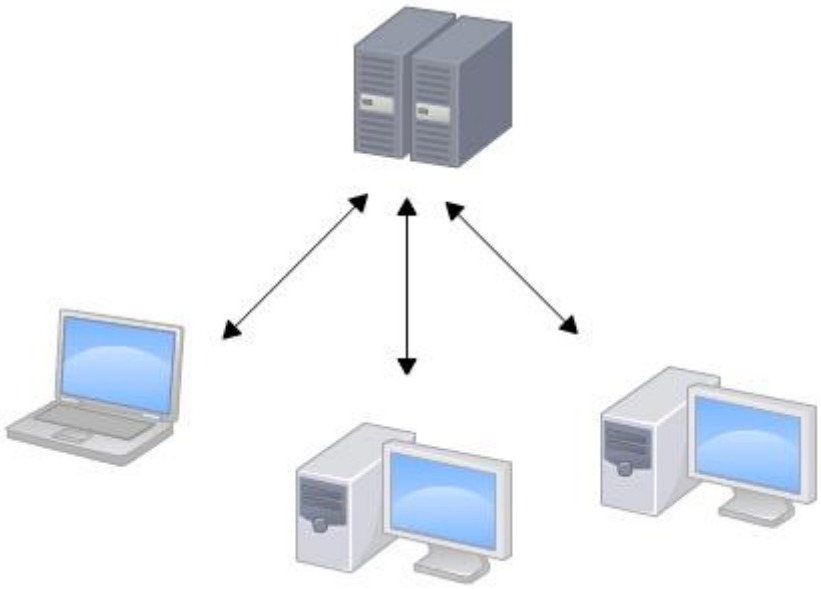
1.1 Git的诞生

- 源自Linux开源代码的管理
- Linus手工合并全球志愿者的Linux代码
- Linus用C语言制作分布式版本控制系统，即Git
- 2008年，GitHub上线，成为开源项目的天堂

1.2 集中式vs分布式

1.2.1 集中式版本控制系统

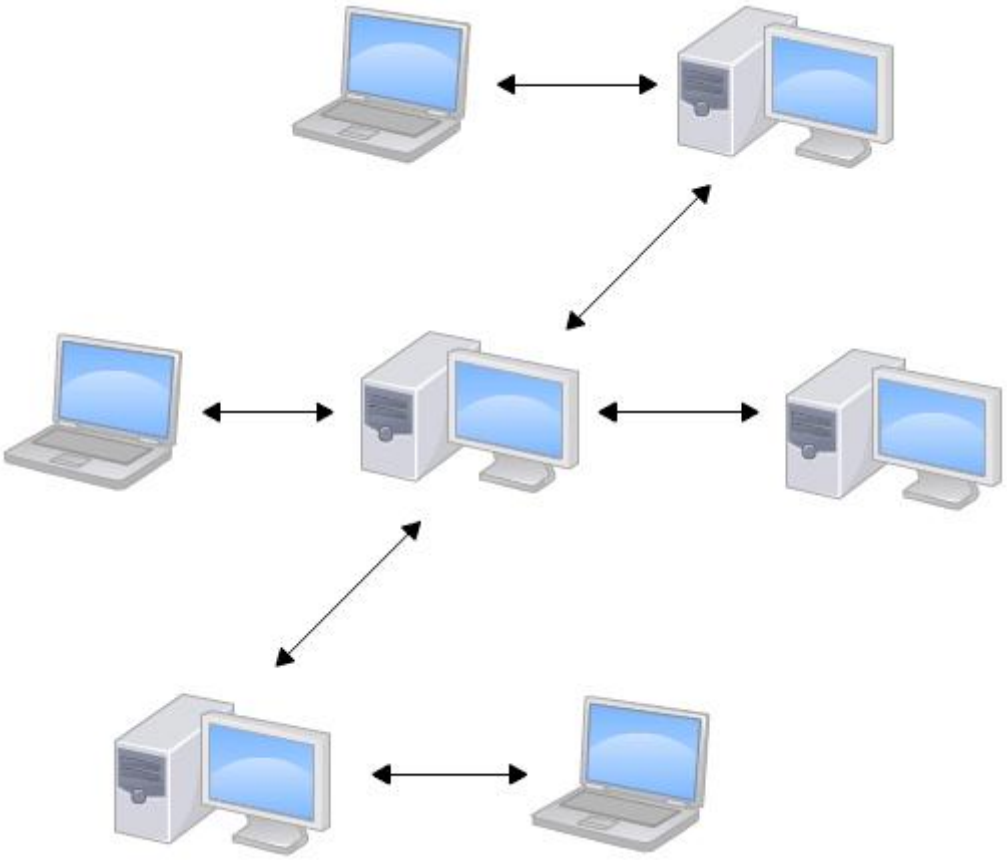
- 版本库存放在中央服务器，代表：CVS （Concurrent Versions System）



- 缺点：必须联网才能工作，不适合多人开发的项目
- 优点：管理方便，逻辑明确，符合一般人思维习惯。 易于管理，集中式服务器更能保证安全性。代码一致性非常高

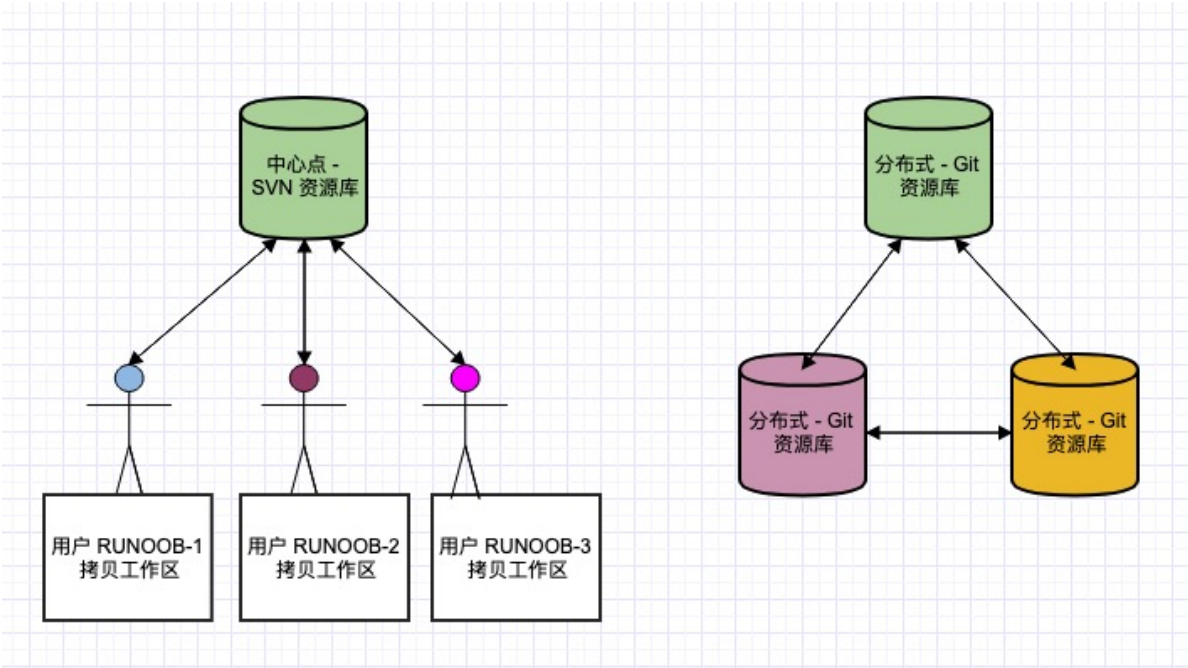
1.2.2 分布式版本控制系统

- 代表：Git



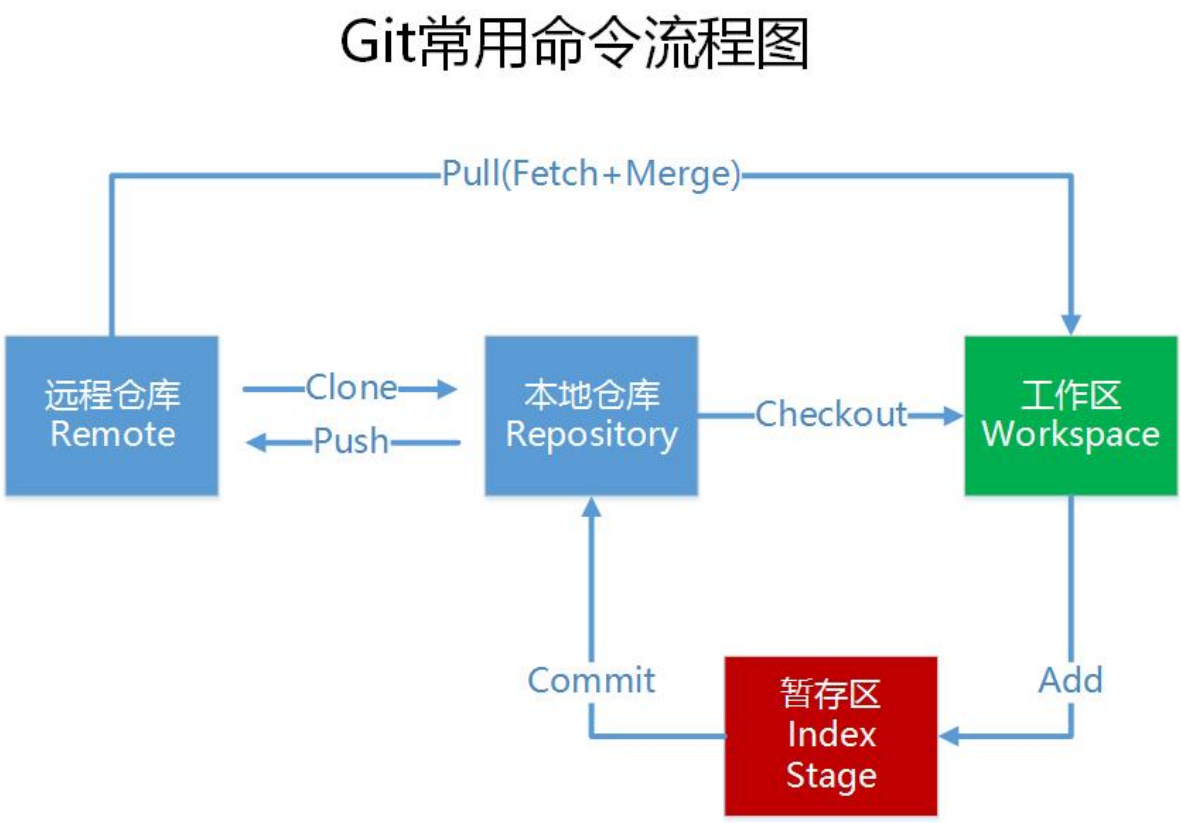
- 每台电脑都是一个完整的版本库，工作时无需联网
- 公共服务器压力和数据量都不会太大
- 速度快、灵活
- 任意两个开发者之间可以很容易的解决冲突
- 代码保密性差，一旦开发者把整个库克隆下来就可以完全公开所有代码和版本信息

1.2.3 架构对比



二、Git的基本概念

2.1 Git的组成



2.1.1 远程仓库Remote

托管代码的服务器，开发人员通过这个服务器将仓库克隆到自己的电脑上进行开发，同时也可以将自己的代码上传到远程仓库中，以便和他人共享

目前主流的远程仓库有：GitHub（需要梯子），Gitee（码云，国内使用）

2.1.2 本地仓库Repository

开发者个人电脑上的版本库，与远程仓库直接对接。版本库中的HEAD指针指向最新放入仓库的版本

2.1.3 工作区Workspace

开发者存放项目代码的地方

2.1.4 暂存区Index/Stage

用于临时存放改动，本质上是一个文件，保存即将提交的文件列表信息

2.2 Git文件的4大状态

2.2.1 untrack

未追踪的，比如仓库里面新建的文件，还未被add到暂存区的文件

可以理解成未被合法化的文件，还不属于仓库的一员

可用git status命令查看

2.2.2 modified

被修改过的，即仓库中被登记在案的文件又发生了改动，此前的记录已经不是最新的了

处于modified状态的文件可以用git add命令更新，也可用git restore命令回到untrack状态

可用git status命令查看

2.2.3 staged

暂存态，当前文件的所有改动都被记录下来，且系统中的记录是最新的

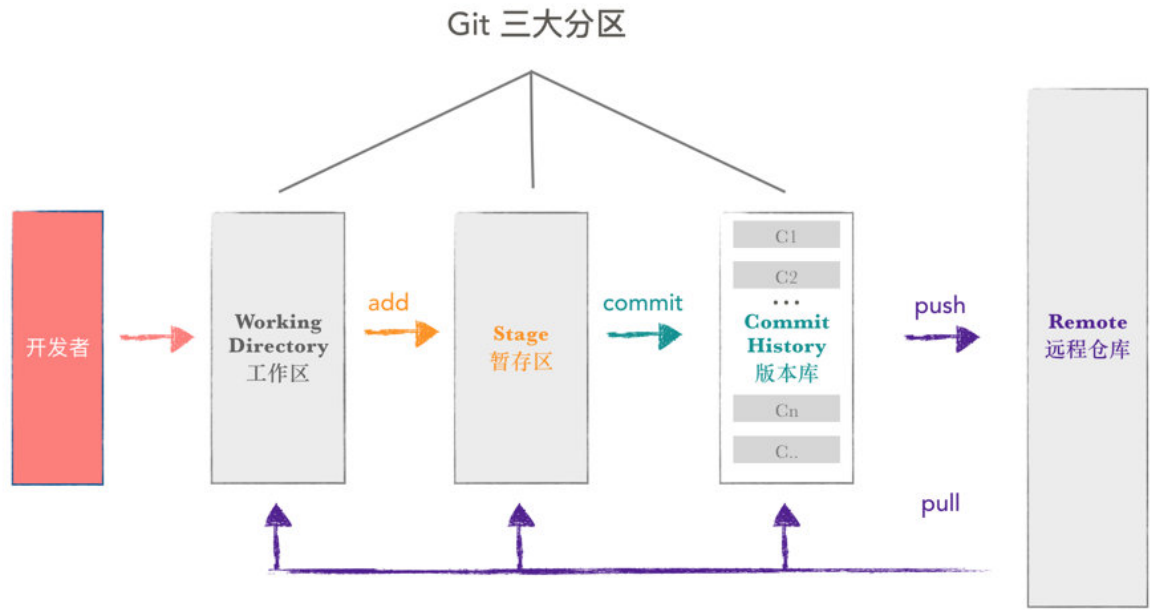
当对文件执行了git add命令后，文件状态就会变成staged

可用git status命令查看

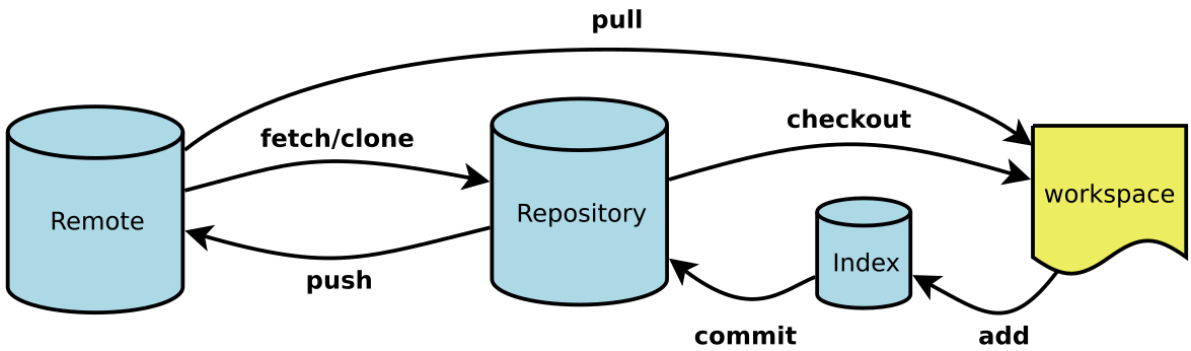
2.2.4 committed

已提交，staged态的文件从暂存区提交到仓库中，被提交的文件状态会变为committed

可用git status命令查看



2.3 Git的常见命令



2.3.1 创建仓库

命令	说明
`git init`	初始化仓库
`git clone`	拷贝一份远程仓库，也就是下载一个项目。

2.3.2 提交与修改

命令	说明
`git add`	添加文件到仓库
`git status`	查看仓库当前的状态，显示有变更的文件。
`git diff`	比较文件的不同，即暂存区和工作区的差异。
`git commit`	提交暂存区到本地仓库。
`git reset`	回退版本。
`git rm`	删除工作区文件。
`git mv`	移动或重命名工作区文件。

2.3.3 提交日志

命令	说明
`git log`	查看历史提交记录
`git blame`	以列表形式查看指定文件的历史修改记录

2.3.4 远程操作

命令	说明
`git remote`	远程仓库操作
`git fetch`	从远程获取代码库
`git pull`	下载远程代码并合并
`git push`	上传远程代码并合并

三、Git的使用

3.1 创建仓库

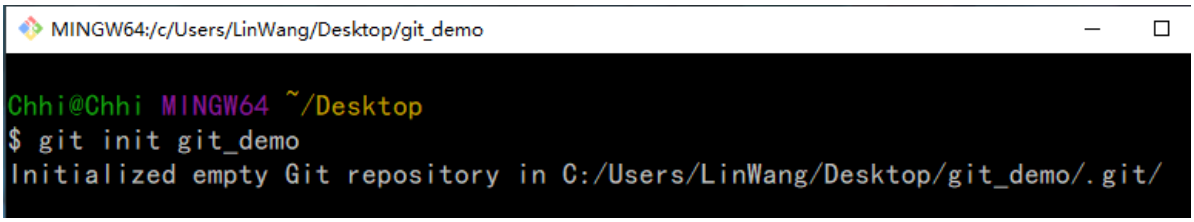
Git 使用 **git init** 命令来初始化一个 Git 仓库，Git 的很多命令都需要在 Git 的仓库中运行，所以 **git init** 是使用 Git 的第一个命令

在执行完成 **git init** 命令后，Git 仓库会生成一个 .git 目录，该目录包含了资源的所有元数据，其他的项目目录保持不变

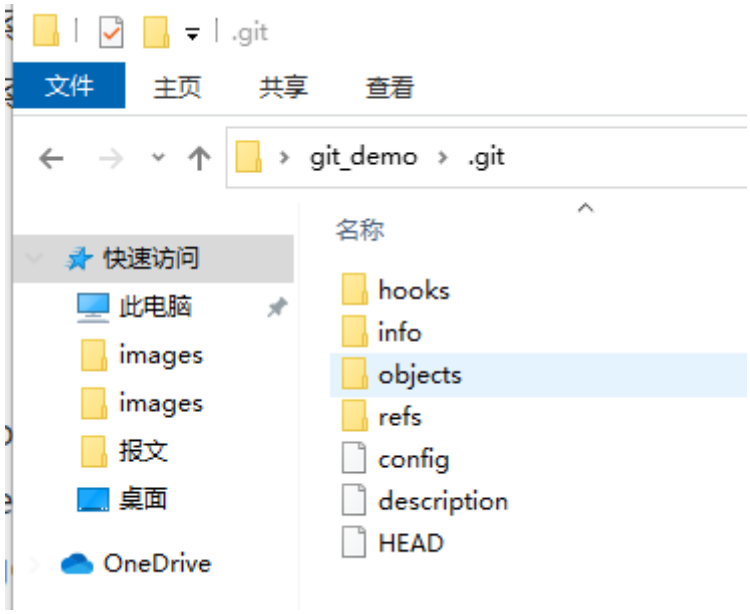
命令格式：

```
git init [仓库名]
```

在桌面打开Git Bash，输入：git init git_demo，创建一个名为git_demo的仓库



仓库根目录下有一个.git文件，里面存放了该仓库的元数据



3.2 克隆仓库

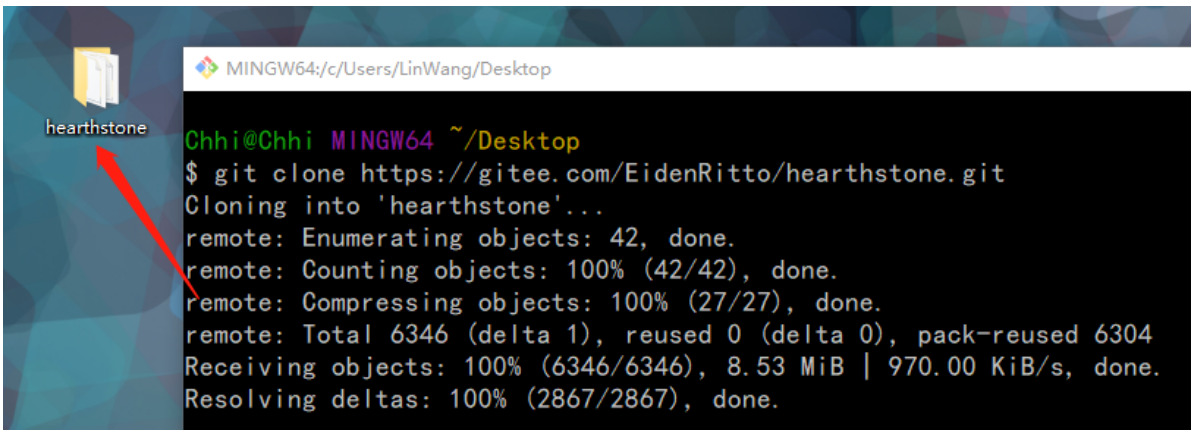
命令格式：

```
git clone <仓库地址> [指定目录]
```

打开Git Bash，从远程克隆一份仓库到本地

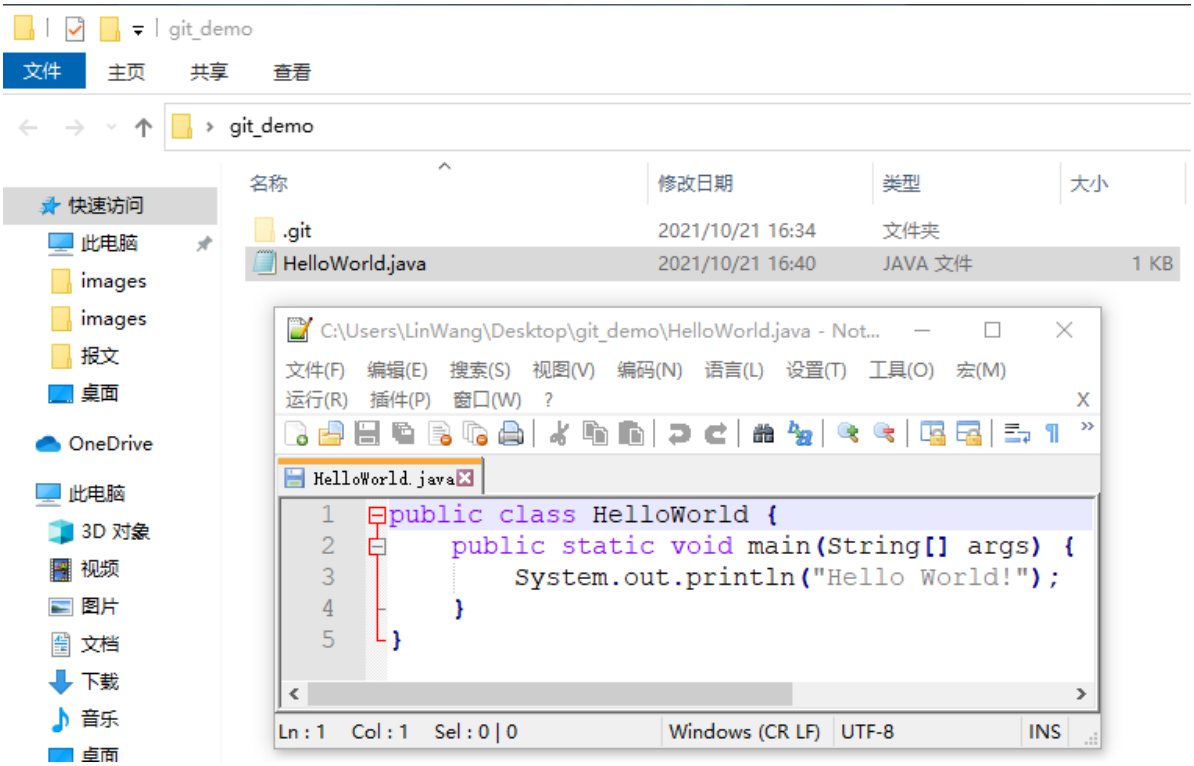
这里选择Gitee中一款开源的炉石传说模拟器

```
git clone https://gitee.com/EidenRitto/hearthstone.git
```



3.3 添加文件

在git_demo中添加文件HelloWorld.java



将文件纳入版本控制，命令格式：

```
git add <文件名> （告诉git对这些文件进行跟踪）
git commit -m "<备注信息>" （提交上述文件到暂存区）
```

3.4 分支管理（Git的核心）

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

Git通过保存一系列不同时刻的快照来保存文件的变化或差异

分支管理可以方便协同开发，从而使开发者间互不影响

3.4.1 Git仓库的树形结构

将Git仓库中所有commit节点之间的关系看作一棵树

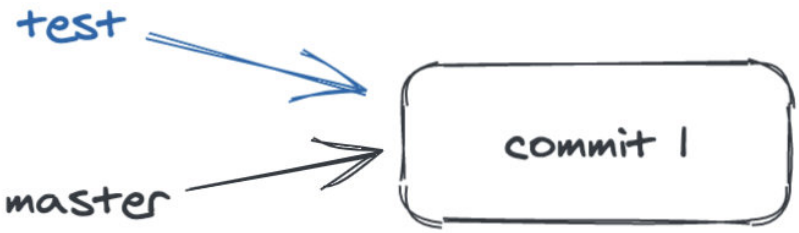
将分支视作指向commit节点的指针

初始化一个仓库，默认分支为master，它指向当前的第一个提交commit1

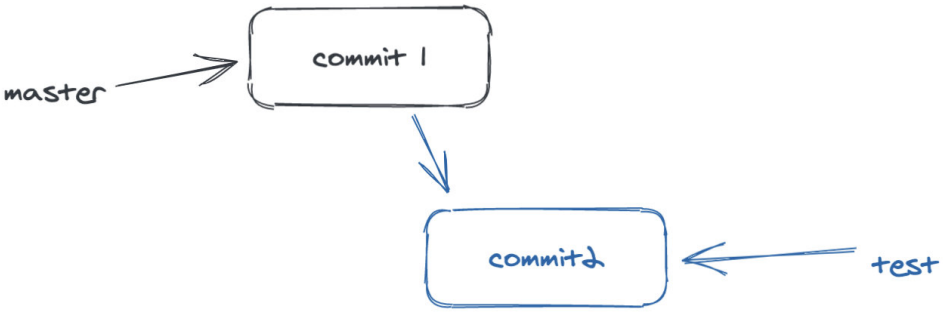
使用命令

```
git branch test
```

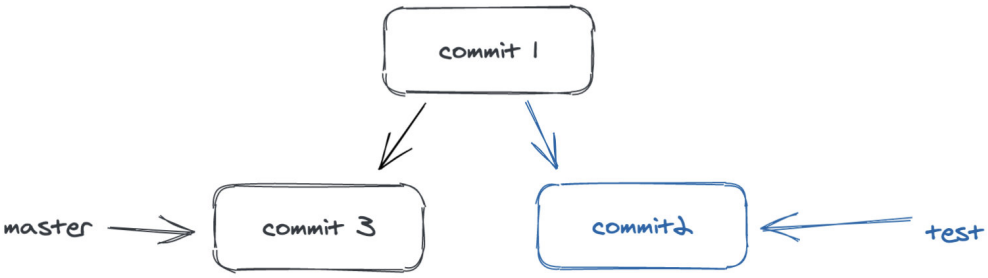
新建一个测试分支，test指针也指向commit1节点



此时在test分支上做改动，并产生一个新的提交commit2，test指针会指向commit2，master指针继续指向commit1上，此时commit2会成为commit1的子节点



此时如果再次回到master分支改动，并创建一个新的提交commit3，那么master指针会指向commit3，commit3与commit2互为兄弟节点



3.4.2 分支与HEAD指针

创建分支命令

```
git branch <新分支名>
```

切换分支命令

```
git checkout <已存在分支名>
```

创建并切换到新的分支

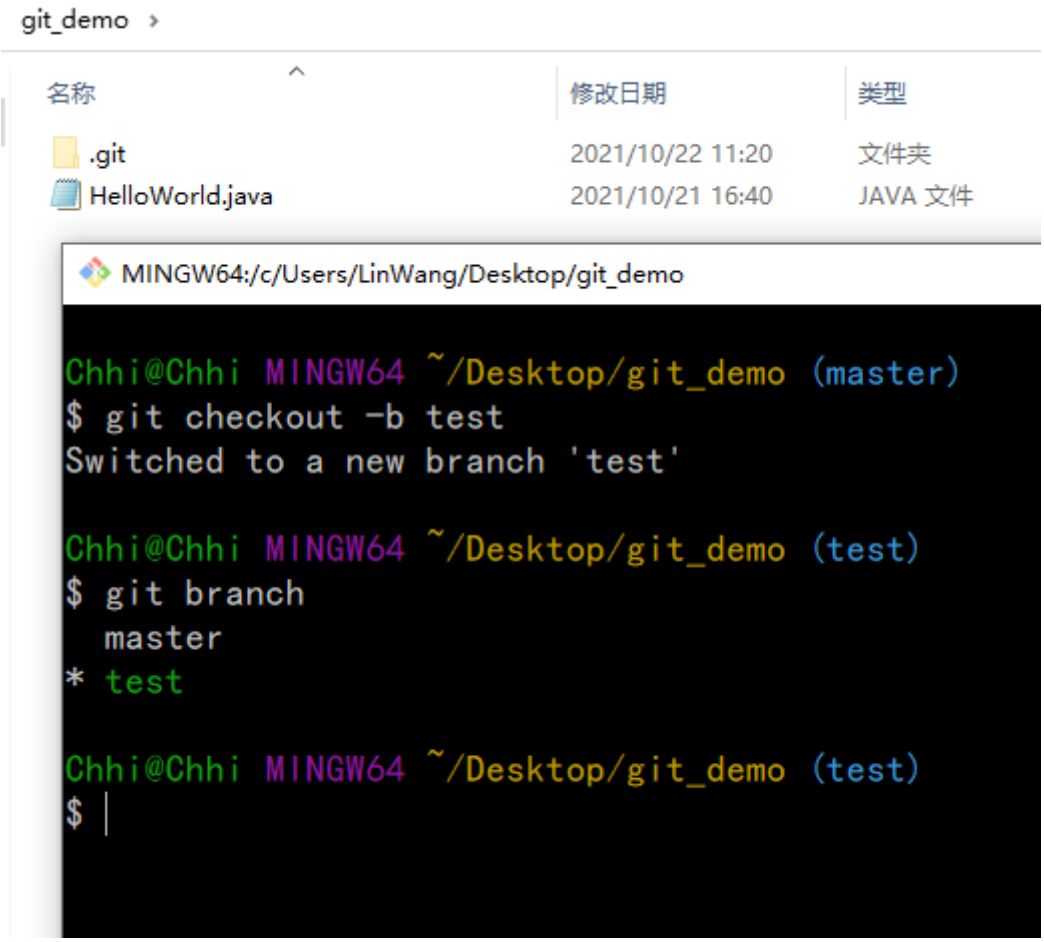
```
git checkout -b <新分支名>
```

Git内部有一个特殊的HEAD指针，它指向当前代码仓库的位置

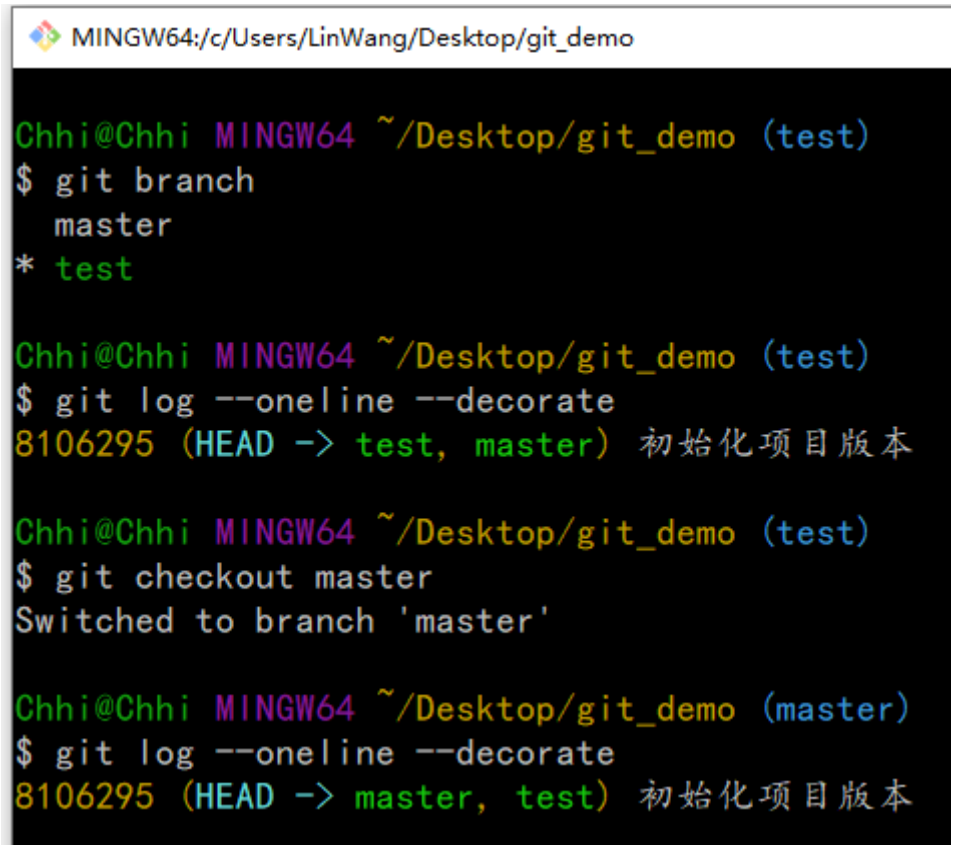
其实HEAD指针不仅可以往前移动，还可以移动到任意节点上，哪怕不再当前的分支上也可以。移动HEAD指针需要用到git checkout命令，它可以指定HEAD指针移动到其他位置。既可以是某一个分支，也可以是根据commit id来确定的节点

提交代码的时候，不止只有分支的这些指针会往前移动，HEAD指针也会随着移动。

创建test分支，并查看分支



查看HEAD指针的指向



上述操作表明使用git checkout命令可以改变HEAD指针指向的位置

3.4.3 远程分支

即远程代码仓库中的分支

在使用git clone将远程仓库克隆到本地时，git会自动的将仓库命名为origin

拉取它所有的数据之后，创建一个指向它master的指针，命名为origin/master

之后会在本地创建一个指向同样位置的指针，命名为master，和远程的master作为区分

3.4.4 代码拉取

从远程仓库拉取代码

```
git pull
或
git fetch
```

git fetch的作用是将远程的改动同步到本地

执行git fetch origin时，git会把远程所有的改动和分支都拉取到本地，并命名为origin/xxx

当使用git checkout切换分支时，git会自动生成一个本地的分支指针，即

```
git checkout -b test
相当于
git checkout -b test origin/test
```

git pull与git fetch的区别

git fetch只能拉取远程仓库版本最新的代码，但不会merge本地仓库的代码

git pull是将所有远程索引合并到本地分支中

总结：git pull = git fetch + git merge

3.4.5 代码推送

首先需要明白，本地分支是不会自动和远程分支同步的

这就需要人为进行git push操作

git push的命令格式如下

```
git push <远程主机名> <本地分支名>:<远程分支名>
```

如果本地分支名与远程分支名相同，则可以省略冒号

```
git push <远程主机名> <本地分支名>
```

例如，远程origin已有一个test分支，本地也有一个test分支

那么将本地test分支推送到origin只需执行命令

```
git push origin test
等价于
git push origin test:test
```

3.5 分支合并

在使用git进行协同开发的过程当中，虽然大家都在各自的分支。但是最后代码还是要合并到一起的，这样才可以投入使用。git当中代码的合并是通过分支合并来体现的

3.5.1 新的分支

一般来讲，仓库中会有一个主分支，用于项目的发布。因此发布者无需理解每一个分支做了什么。大多数的分支只是暂时的，用来暂时完成一项功能的，等功能完成之后，一般都会再合并回master分支，将所有的改动合并进去

对此前的HelloWorld.java进行修改，在其中添加一段代码：

```
public class HelloWorld {
    public static void main(String[] args) {
        print();
    }

    private static void print() {
        System.out.println("Hello World!");
    }
}
```

将此代码commit到test分支

合并的方式非常简单，只需先checkout到想要合并的目标分支。比如当前所在分支为test，要想合并到master，使用命令如下：

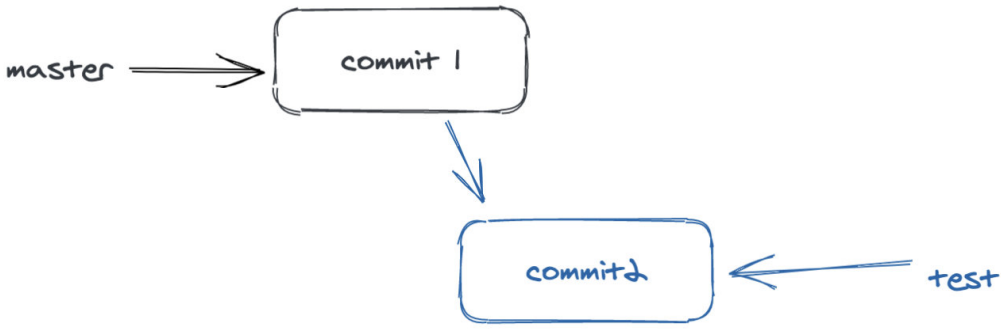
```
git checkout master
git merge test
```

```
MINGW64:/c/Users/LinWang/Desktop/git_demo

Chhi@Chhi MINGW64 ~/Desktop/git_demo (test)
$ git checkout master
Switched to branch 'master'

Chhi@Chhi MINGW64 ~/Desktop/git_demo (master)
$ git merge test
Updating 8106295..075632b
Fast-forward
 HelloWorld.java | 4 ++++
 1 file changed, 4 insertions(+)
```

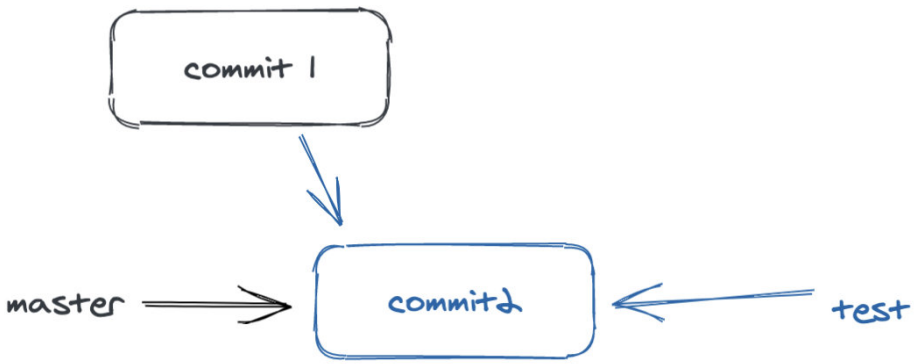
如果没有报错就算合并成功，这里会展示合并进来的代码改动

Fast-forward表示快速合并，因为test分支是从master分支拉出去的，后来master分支也没有进行过改动，所以这里合并的时候直接将指向master的指针指向了test分支

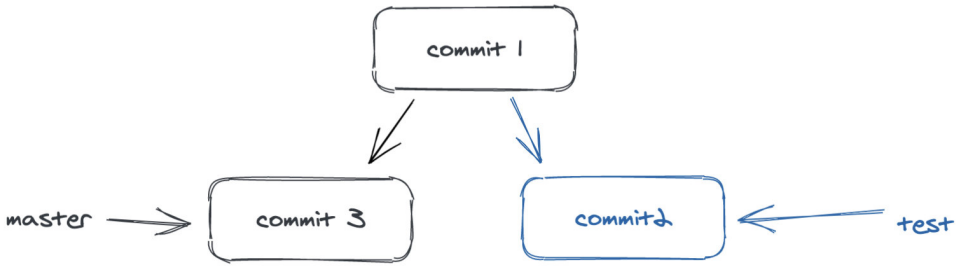
合并前



合并后

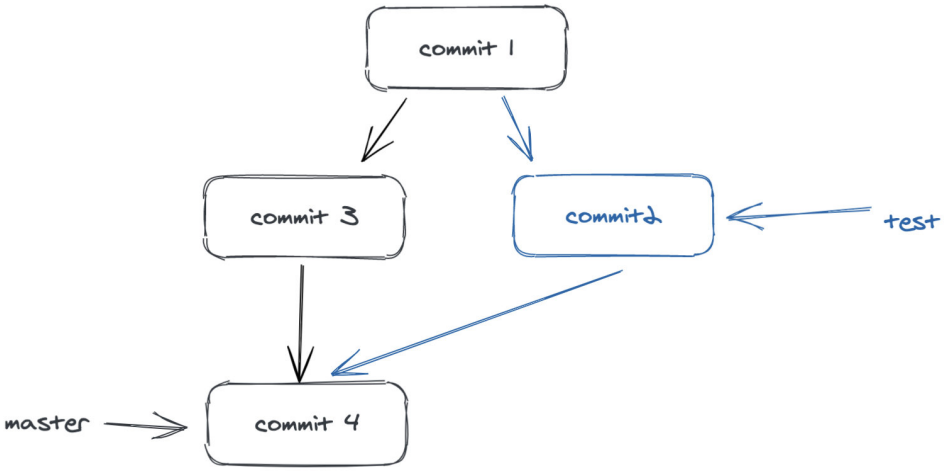


如果master和test分支都有改动，如图所示



3.5.2 解决冲突

此时合并master和test就会产生一个新的提交commit4

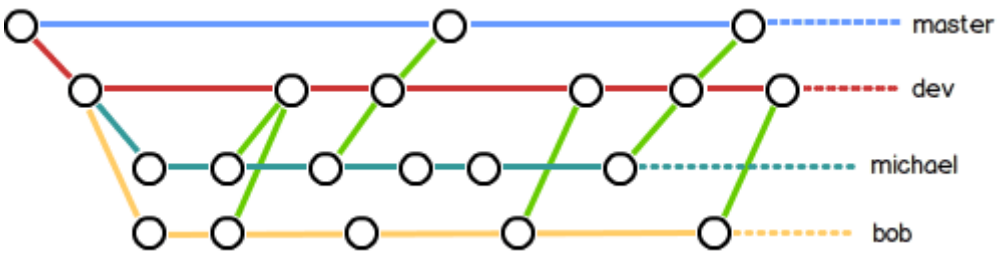


当合并分支出现代码冲突时，由开发人员人工解决冲突

3.5.3 分支策略

在实际开发中，一般按照以下基本原则进行分支管理：

- master分支应该是稳定的，仅仅用来发布新版本
- dev分支作为开发分支，当功能需求通过测试后可以合并到master分支上发布
- 开发人员根据不同的需求从dev分支上拉取分支，时不时往dev分支上合并



3.5.4 多人协作

通常多人协作的工作模式如下：

1. 首先试图借助命令

   ```
   git push origin <branch-name>
   ```

   推送自己的修改到远程仓库
2. 如果推送失败，说明远程分支比当前本地的更新，需要先用pull命令合并
3. 如果合并有冲突，则解决冲突，并在本地提交
4. 重复1，推送成功

注意：如果git pull提示no tracking information，则说明本地分支和远程分支没有链接关系

使用 git branch --set-upstream-to origin/ 即可建立链接关系

四、Git在实际开发中的应用

开发背景：以一个教育管理平台（education-platform，简称ep）的开发为例

4.1 开发流程

1. 开发人员领取一个需求req或一个bug
2. 创建该需求对应的开发分支dev
3. 设计并编写代码，本地测试通过后push代码到远程仓库
4. 创建dev到主分支master的合并请求
5. 由其他开发人员对代码进行评审
   - 评审不通过，回到步骤3
   - 评审通过，合并dev到master分支
6. 由专门的测试人员进行测试

4.2 开发流程实际举例

假设现在有req-0001和bug-0002

开发人员David 为代码实现人员，开发人员Jason为评审人员, 以及测试人员 Billy

目前稳定分支为 main 主分支，目前打算发布 1.0 版本

项目管理人员首先从主分支main创建ep-1.0分支

```
git checkout -b ep-1.0
```

ep-1.0分支将成为最终发布的分支

开发人员David觉得实现需求req-0001，具体步骤为：

1. 先切换到分支ep-1.0

   ```
   git checkout ep-1.0
   ```

2. 创建新的任务分支

```
git checkout -b ep-req-0001
```

3. 编写代码，本地测试并push代码

```
git add ...
git commit ...
git push ...
```

4. David 创建 合并请求, 并要求Jason进行代码评审

   David进行github界面操作，请求把 ep-req-000001 分支合并到ep-1.0中， 并要求Jason进行代码评审

   Jason 开始对代码进行阅读， 如果Jason对代码没有任何异议，Jason 进行github界面操作 通过合并请求

   如果Jason有异议，需要与David进行沟通讨论，如果David认为Jason正确，需要回到 步骤3, 并重复此过程直到双方都满意

5. 测试人员Billy开始测试 ep-req-000001

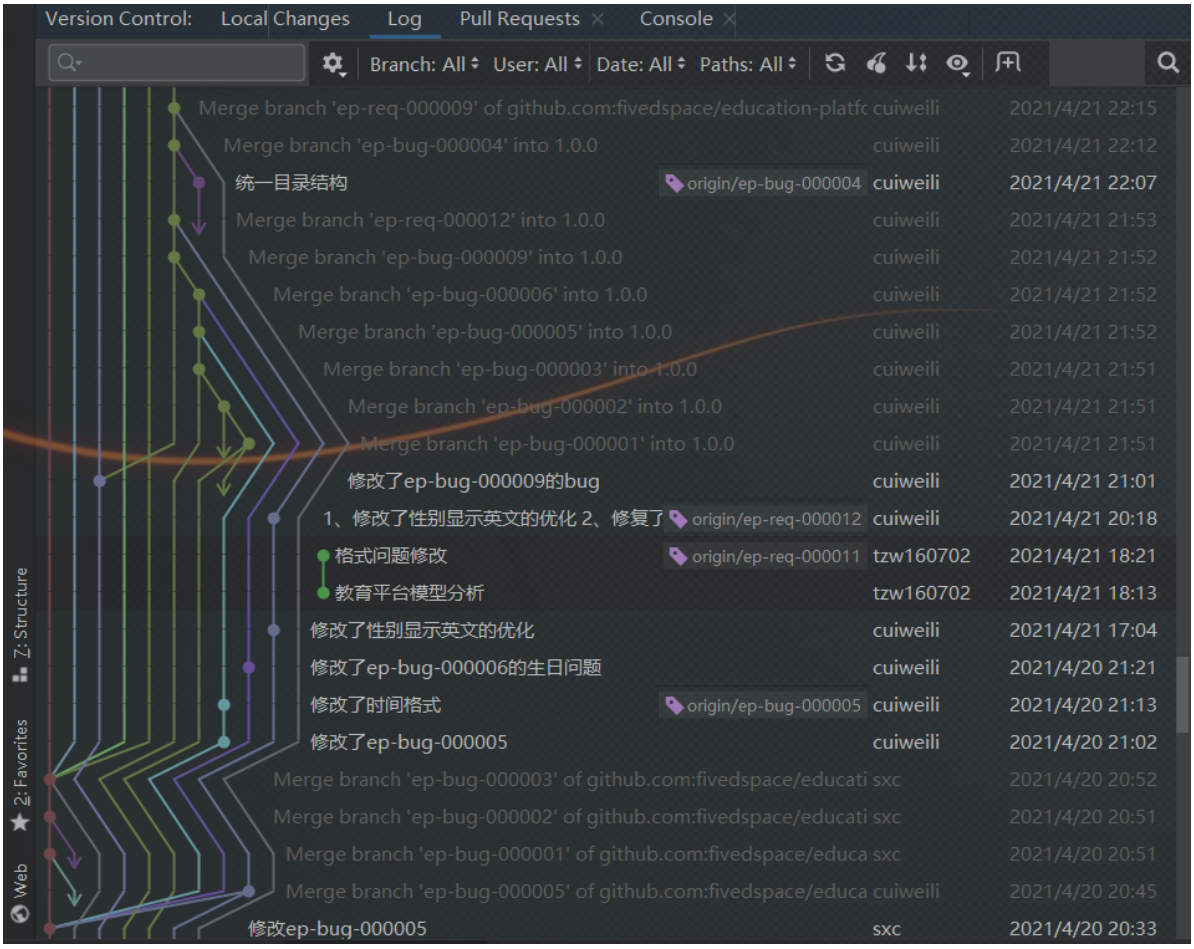
   测试失败：通知David需要重新开发，David回到步骤3

   测试成功: 该需求完成

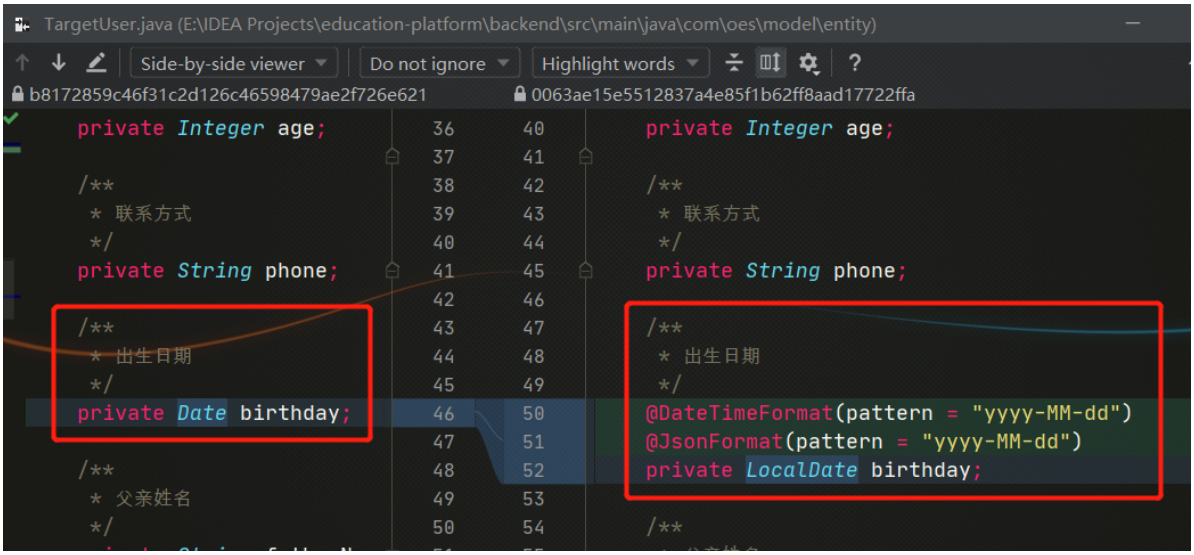
6. ep-1.0 的所有需求都完成并通过测试，发布ep-1.0
7. 将 ep-1.0 合并到主分支

4.3 分支视图

当项目处于大量需求开发时期的分支视图

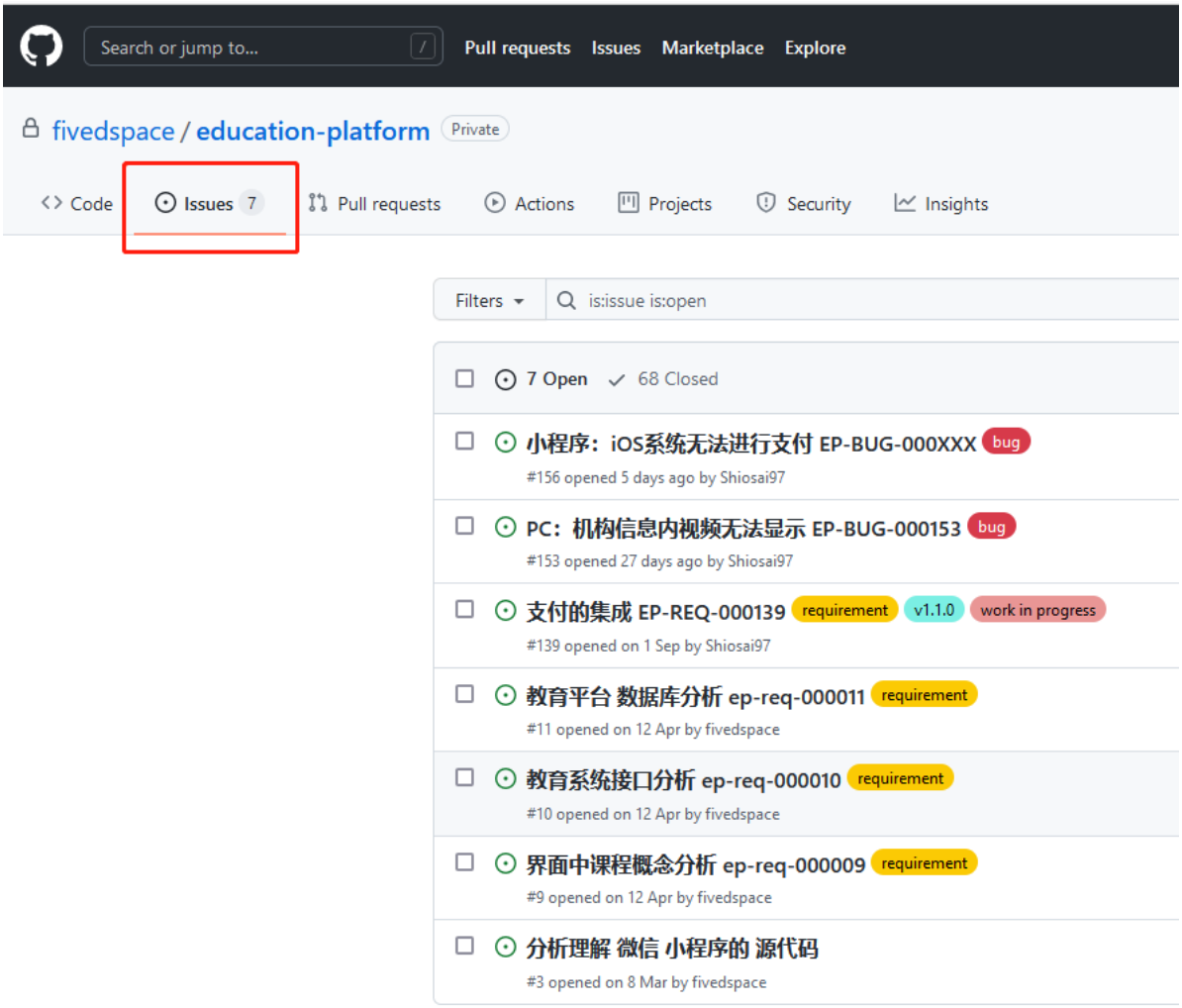


以上图中“修改了ep-bug-000006的生日问题”为例，通过对比提交前后代码，可以获悉该bug是如何解决的



五、Issue机制

5.1 GitHub中的Issue



5.2 项目管理中的Issue

对于开发而言，完成新的需求和修复bug在每日的工作中占用了大部分比重

而这部分的工作量目前没有很好的进行一个量化和统计

同时也没有一个很好的方式去展现目前已经发现的bug和已经修复的bug

使用Issue可以一定程度上解决上面所述的问题

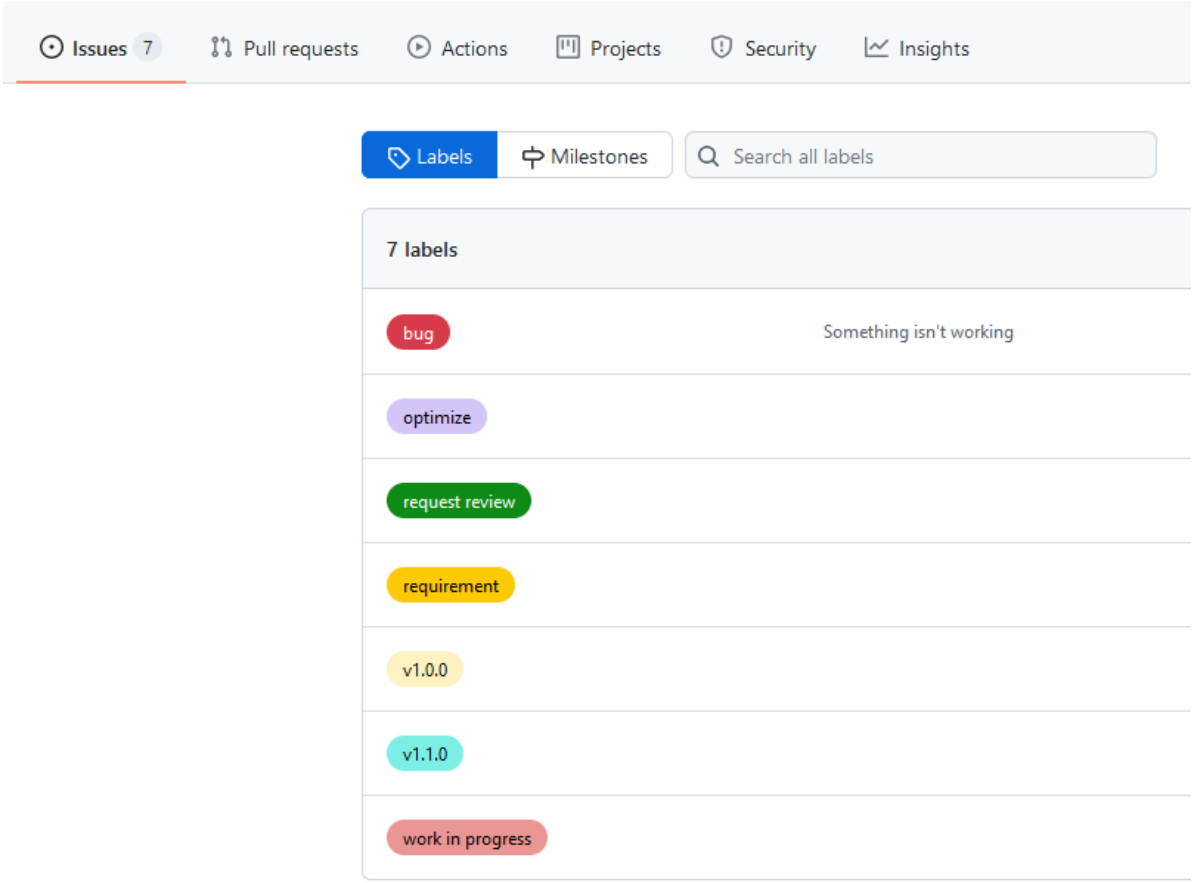
对于新的需求

项目管理人员可以在Issue中发布标签为req的issue，开发人员拉取对应的分支进行开发

对于bug

由测试人员发布标签为bug的issue，开发人员同样拉取对应的分支修复bug

自定义标签



5.3 开源仓库中的Issue

在开源仓库中，任何人都可以针对项目中的任何问题提出Issue，包括但不限于bug、优化或建议

Issue解决后，会被管理员Close，表示这个Issue已经被解决