RUCBASE数据库系统初步实现及学习

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | | **学生团队：**刘禹佟、于鸿达、韦贺文 | | **课 程 名 称 ：**  数据库系统 | | **项 目 名 称 ：**  RUCBASE数据库系统 | | **所 在 专 业 ：** 计算机科学与技术 | |

2025年4月13日

**目录**

一．环境配置指南

二．远程协作流程

三．存储管理实验

四．总结

1. **环境配置指南**

本组采用在Windows系统下使用Docker+vscode 对Rucbase进行开发。

Rucbase链接： <https://github.com/ruc-deke/rucbase-lab>

1. **Docker安装及其相关配置**

推荐参考bilibili视频进行配置

<https://www.bilibili.com/video/BV1xHA3euEcn?vd_source=ea73a00318195a9133a2c368a9d10839>

Docker官网下载链接https://www.docker.com

在安装Docker后，需要进行对WSL2的安装，因为Docker需要Linux内核来运行容器，WSL2提供了完整的兼容性。

**WSL2安装流程：**

a.启用WSL2功能

（1）打开开始菜单，在开始菜单中点击应用与程序；

（2）在应用与程序选项里面，鼠标滚到最底下，点击程序与功能；

（3）点击选项启用或关闭 Windows 功能；

（4）在弹出的窗口中勾选 虚拟机平台 和 适用于 Linux 的 Windows 子系统。

如下图所示



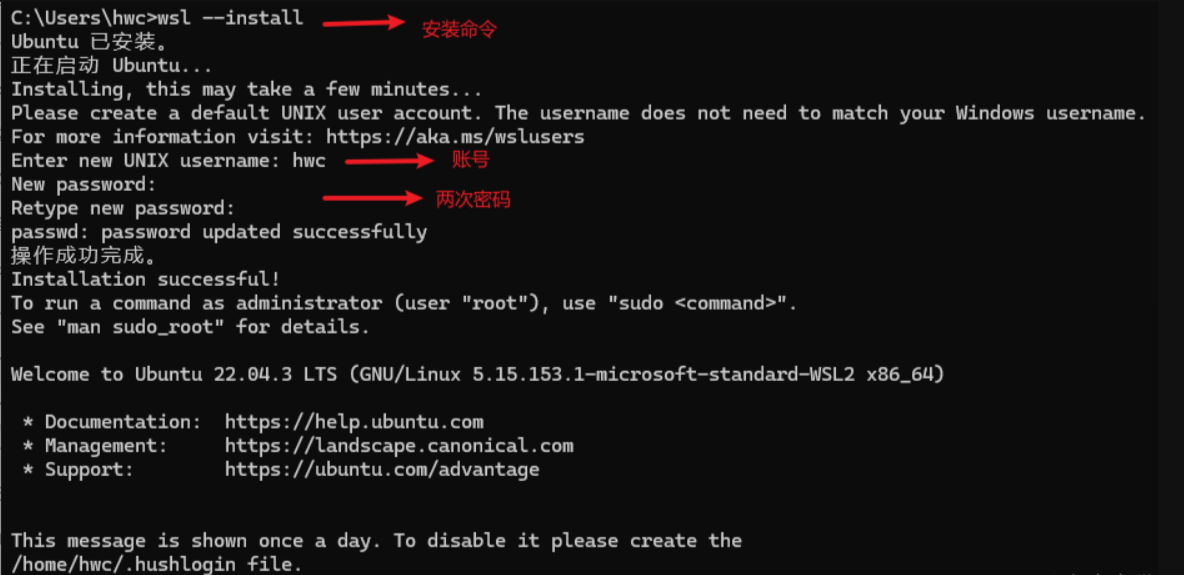
完成上述操作后重启电脑

b.进行对WSL2的安装

（1）win+r 打开命令行输入以下指令 wsl --install

（2）等待几分钟，会跳出输入账号密码的界面，首次启动需要设置用户名，输入用户名后进行密码设置，设置好后可以看到Ubuntu启动成功。

如下图所示



完成对Linux系统的安装

完成以上操作后重启计算机根据视频进行镜像源的配置。

1. **使用docker进行对rucbase的拉取**

参考b站视频

<https://www.bilibili.com/video/BV1Nc41147Pd?vd_source=ea73a00318195a9133a2c368a9d10839>

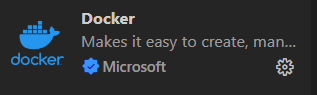
在搜索框中输入rucbase对下图所示进行拉取（Pull），注意将标签djb2024改成djb-ssh



也可以使用cmd输入以下指令进行拉取 docker pull djb2023/rucbase\_lab:djb-ssh

1. **进行对vscode插件的安装**

开发主要使用c++语言 除了下载与c++相关的插件 还需下载以下插件实现与docker的连接



1. **创建docker容器并完成对vscode调试的实现**

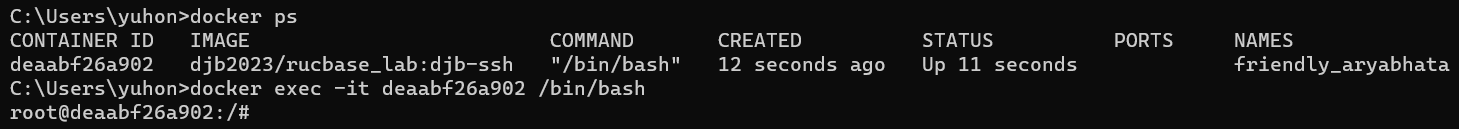
使用如下命令创建容器 docker run -dit djb2023/rucbase\_lab:djb-ssh



查看生成的容器并运行容器进入终端

docker ps

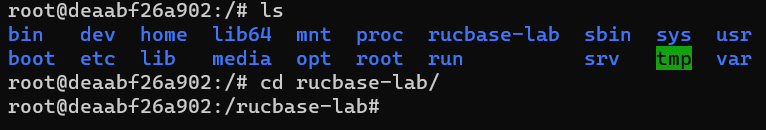
docker exec -it 容器id (deaabf26a902) /bin/bash



查看目录并进入文件

Ls

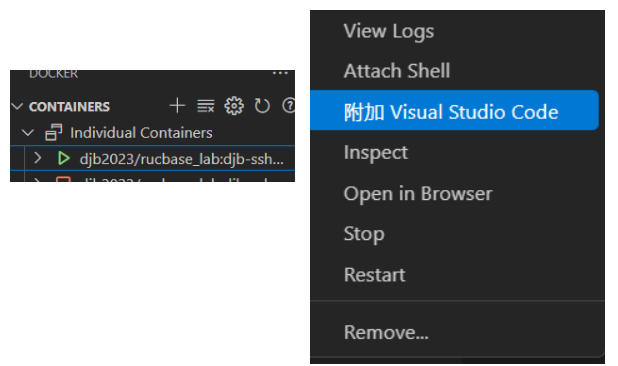
cd rucbase-lab/



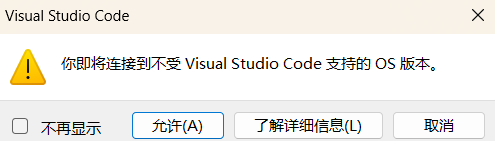
点击左侧任务栏如下图标



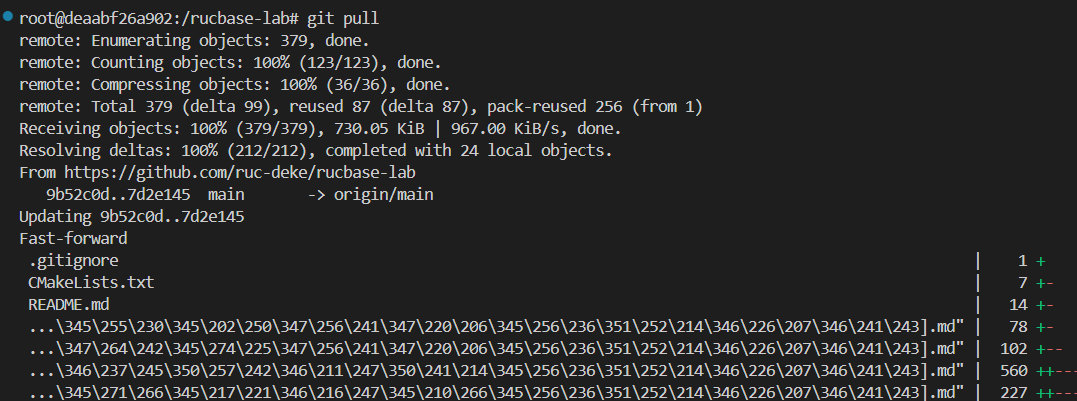
右键刚刚运行的文件 点击如下选项



安装服务器完成后点击允许



在vscode终端中进行操作 git pull对github中最新代码进行拉取



剩余一些测试用例具体查看如下视频时间段11:50--16:14

[https://www.bilibili.com/video/BV1Nc41147Pd?vd\_source=ea73a00318195a9133a2c368a9d108](https://www.bilibili.com/video/BV1Nc41147Pd?vd_source=ea73a00318195a9133a2c368a9d10839)

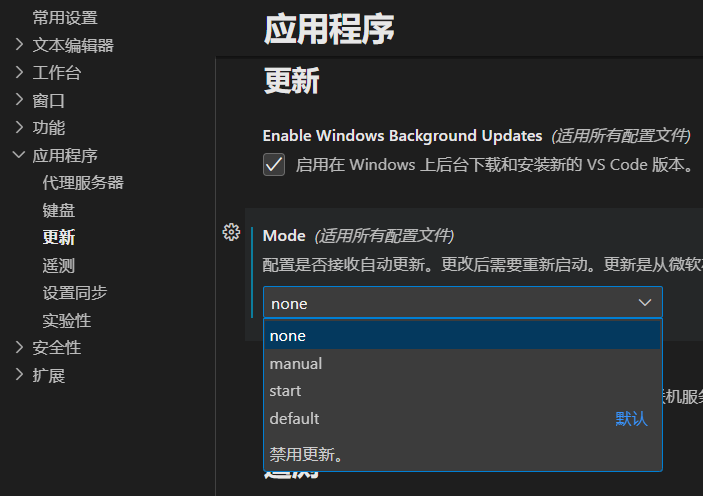
[39](https://www.bilibili.com/video/BV1Nc41147Pd?vd_source=ea73a00318195a9133a2c368a9d10839)

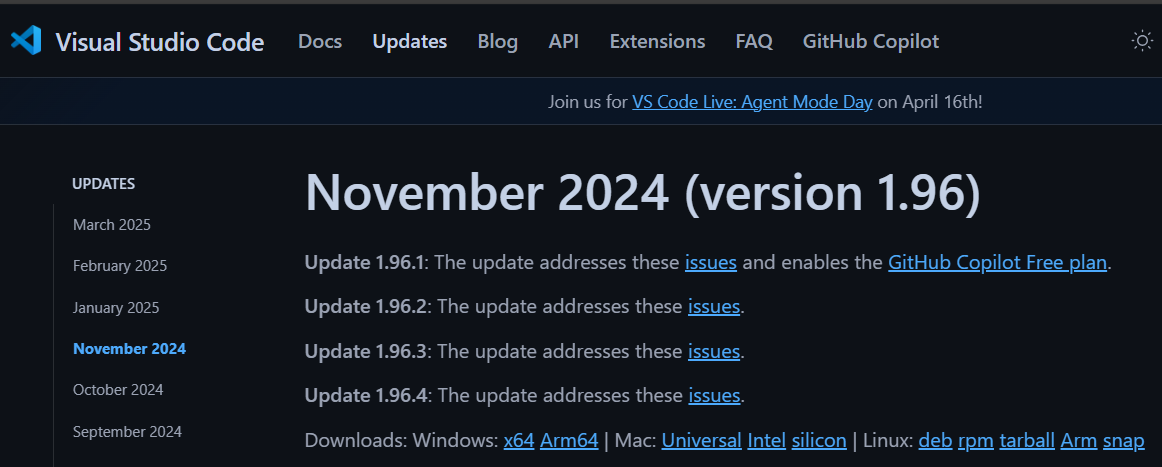
在完成以上操作后更改vscode中的launch.json文件如下后可以成功对vscode进行调试



注意事项：如果在后续与docker的链接中出现版本不匹配的问题

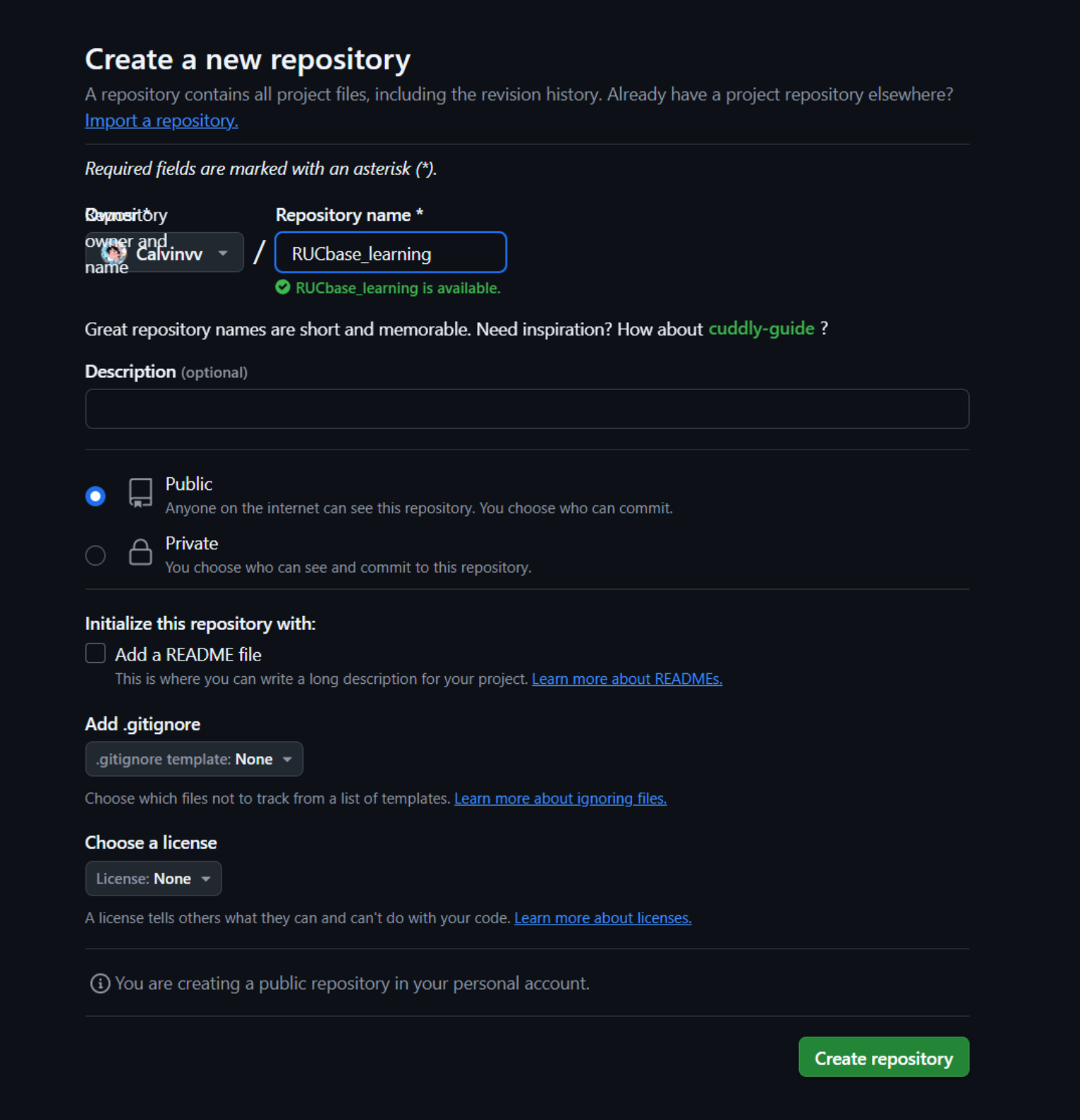
可以将设置中的更新改成如下后并下载vscode的合适版本后进行重启。



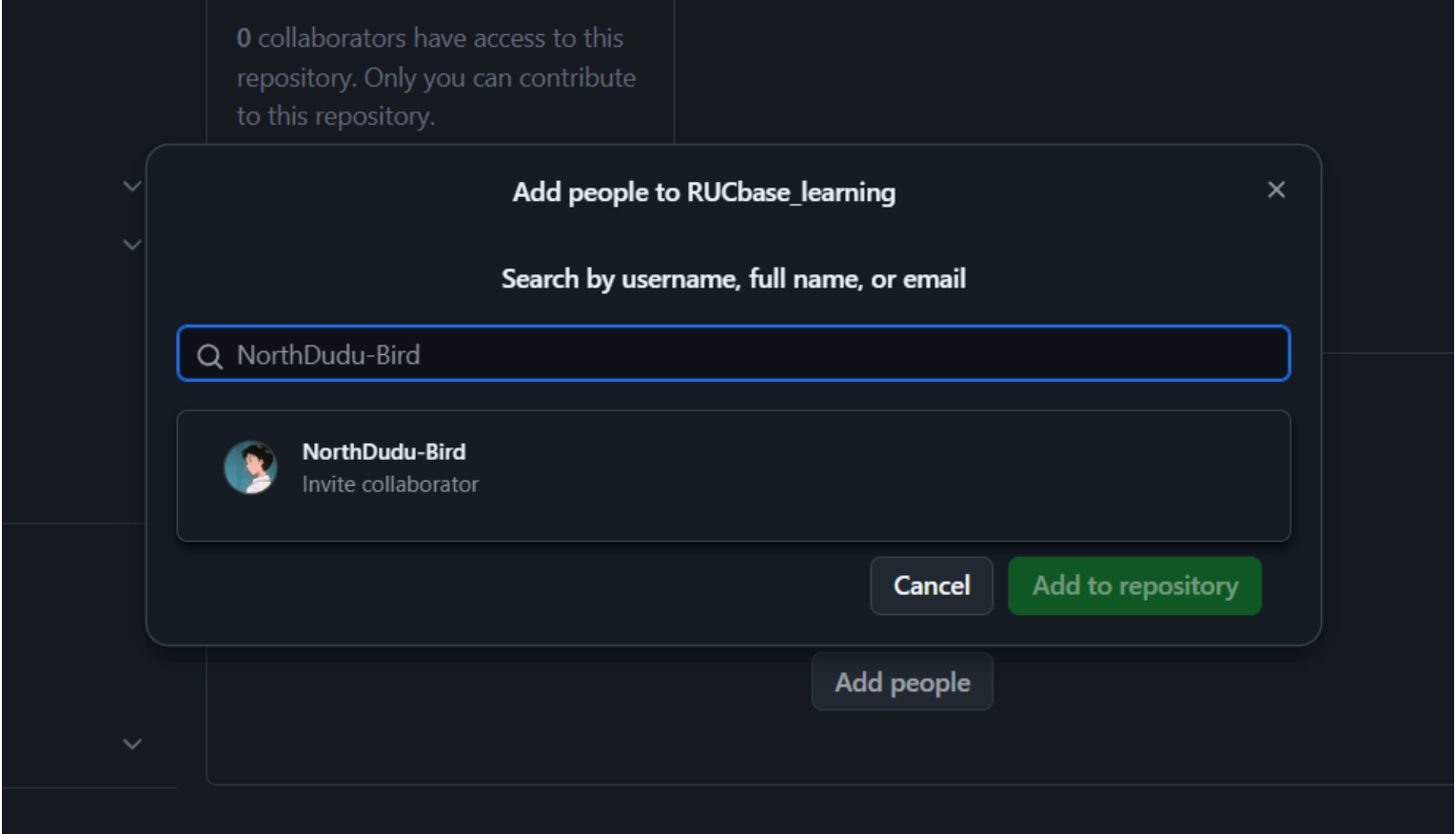
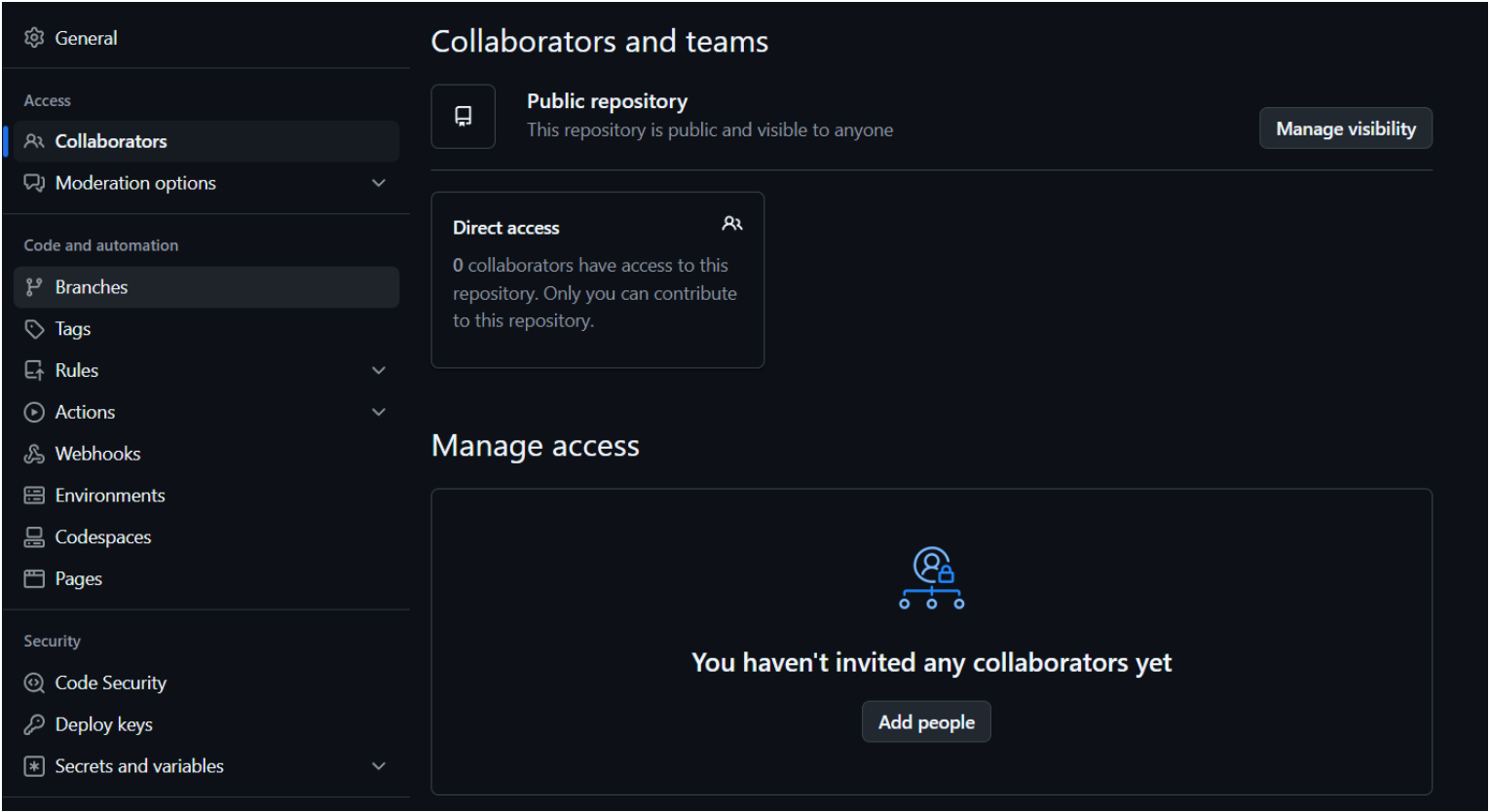


**二．远程协作开发**

要进行远程协作，我们首先需要创建一个团队仓库



创建完毕后，进入仓库的setting，在这里可以邀请协作者

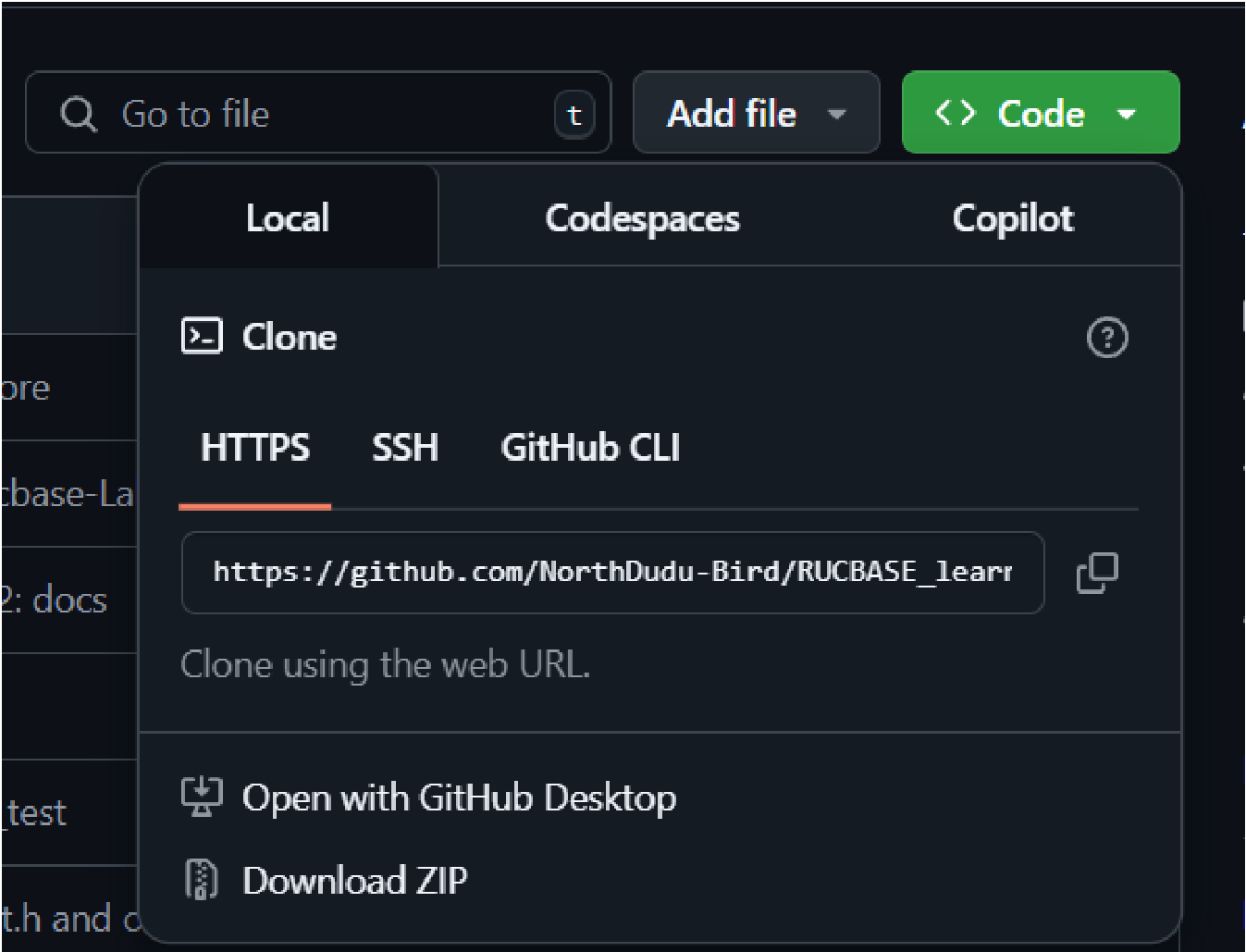
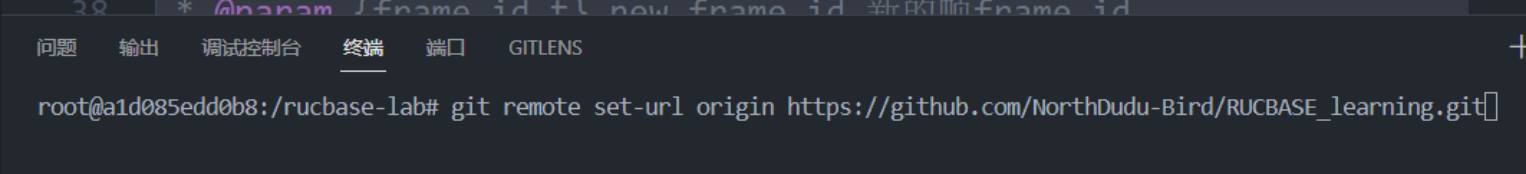


此时队友就获得了访问公共仓库的权利

下面将我们的本地仓库的代码推送到公共仓库中。

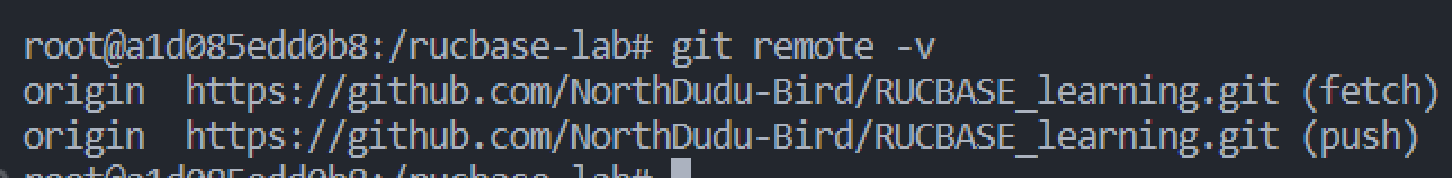
当我们从官方github仓库拉取源码以后，默认绑定的远程仓库是官方仓

库，我们需要使用git remote set-url 命令更改远程仓库URL

下面是从公共仓库中获取的URL

更改后可以使用以下命令查看当前远程仓库的信息：

**git remote -v**

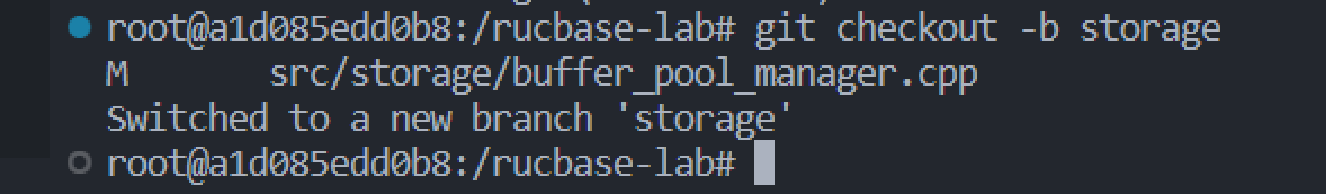


这里已经更换成功。

在实际协作中，我们往往使用分支进行管理。每个新功能创建一个新分支，开发评审测试通过后合并入主分支。

建立存储分支

**git checkout -b storage**

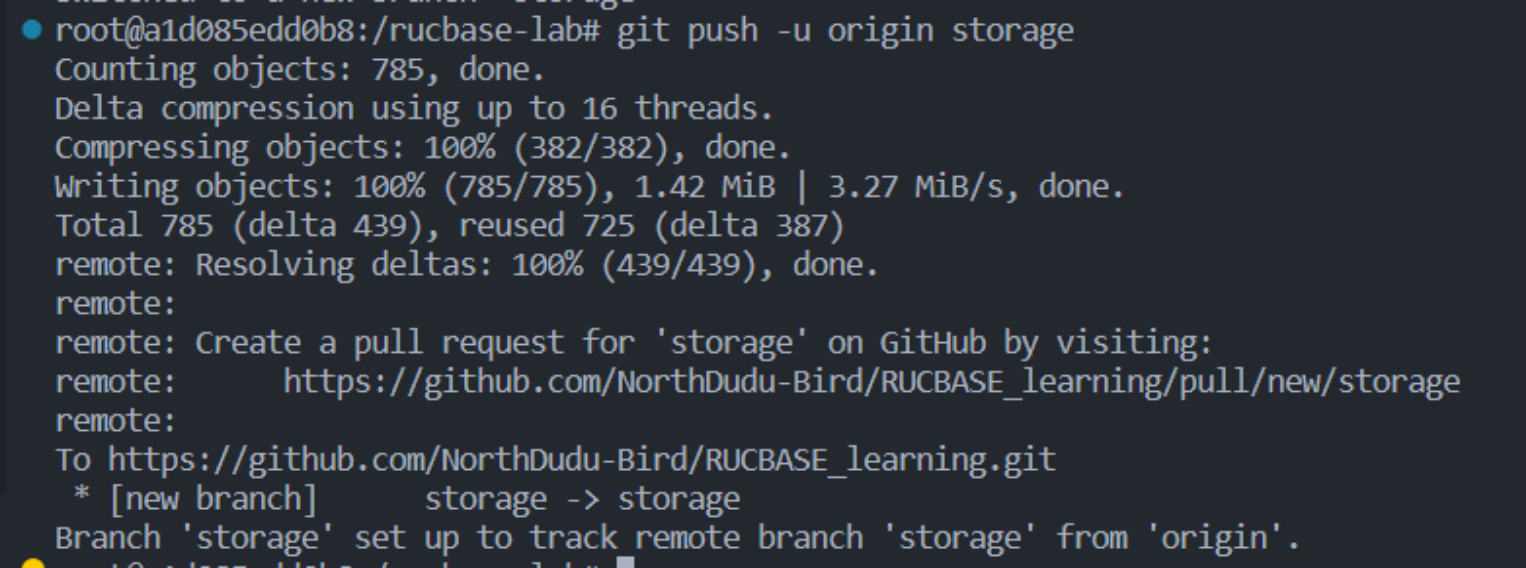


将本地的 storage 分支推送到远程仓库进行保存

**git add .**

**git commit -m "完成了storage部分的开发"**

**git push -u origin storage**



本地切换回主分支

**git checkout main** //获取最新的主分支状态

**git pull origin main**

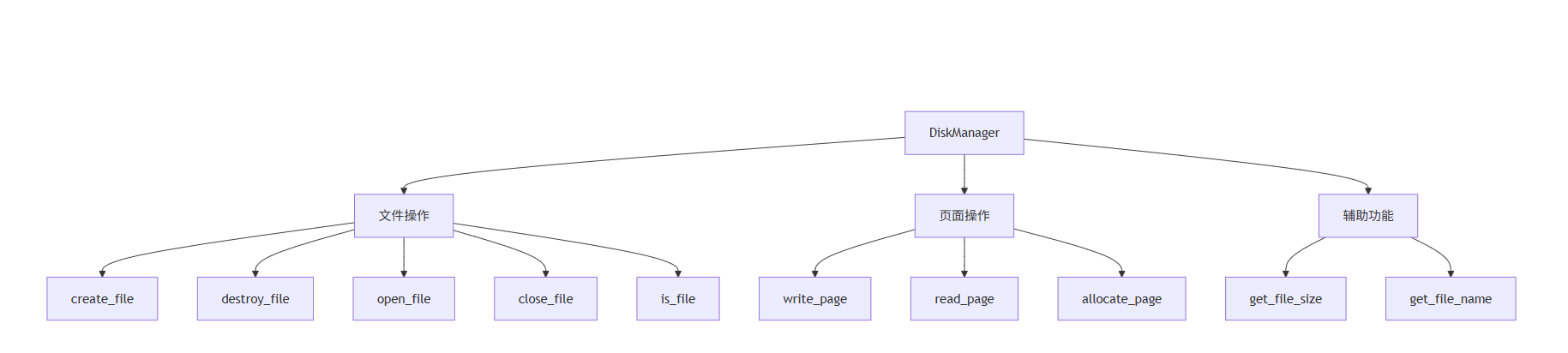
此时就完成了一次分支的推送

在这个过程中，可以使用GitLens等vscode插件进行可视化管理，可以更方便的完成协作流程

**三．存储管理的初步实现**

1. **磁盘管理器(DiskManager)**

**1.1 模块架构**

****

* 1. **完整函数实现**

**1.2.1 文件操作函数**

**// 创建文件**

**void DiskManager::create\_file(const std::string &path) {**

**if (is\_file(path)) throw FileExistsError(path);**

**int fd = open(path.c\_str(), O\_CREAT, 0600);**

**if (fd == -1) throw UnixError();**

**close(fd);**

**}**

**// 打开文件**

**int DiskManager::open\_file(const std::string &path) {**

**if (!is\_file(path)) throw FileNotFoundError(path);**

**if (path2fd\_.count(path)) return path2fd\_[path];**

**int fd = open(path.c\_str(), O\_RDWR);**

**if (fd == -1) throw UnixError();**

**fd2path\_[fd] = path;**

**path2fd\_[path] = fd;**

**return fd;**

**}**

**// 关闭文件**

**void DiskManager::close\_file(int fd) {**

**if (!fd2path\_.count(fd)) throw FileNotOpenError(fd);**

**std::string path = fd2path\_[fd];**

**fd2path\_.erase(fd);**

**path2fd\_.erase(path);**

**close(fd);**

**}**

**1.2.2 页面操作函数**

**// 读取页面**

**void DiskManager::read\_page(int fd, page\_id\_t page\_no, char \*offset, int num\_bytes) {**

**off\_t offset\_in\_file = page\_no \* PAGE\_SIZE;**

**if (lseek(fd, offset\_in\_file, SEEK\_SET) == -1) throw UnixError();**

**ssize\_t bytes\_read = read(fd, offset, num\_bytes);**

**if (bytes\_read != num\_bytes) {**

**throw InternalError("DiskManager::read\_page Error");**

**}**

**}**

**// 分配页面**

**page\_id\_t DiskManager::allocate\_page(int fd) {**

**assert(fd >= 0 && fd < MAX\_FD);**

**return fd2pageno\_[fd]++; // 原子操作保证线程安全**

**}**

**1.2.3 辅助函数**

**// 获取文件大小**

**int DiskManager::get\_file\_size(const std::string &file\_name) {**

**struct stat stat\_buf;**

**int rc = stat(file\_name.c\_str(), &stat\_buf);**

**return rc == 0 ? stat\_buf.st\_size : -1;**

**}**

**// 获取文件名**

**std::string DiskManager::get\_file\_name(int fd) {**

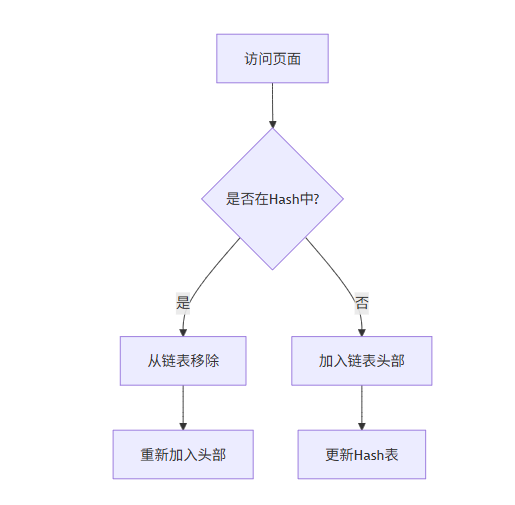
**if (!fd2path\_.count(fd)) throw FileNotOpenError(fd);**

**return fd2path\_[fd];**

**}**

1. **LRU替换策略(LRUReplacer)**

**2.1 算法实现**

****

* 1. **完整函数实现**

**2.2.1 核心函数**

**// 选择淘汰页面**

**bool LRUReplacer::victim(frame\_id\_t\* frame\_id) {**

**std::scoped\_lock lock{latch\_}; // 自动加锁解锁**

**if(LRUlist\_.empty()) return false; // 无页面可淘汰**

**\*frame\_id = LRUlist\_.back(); // 获取最近最少使用的页面**

**LRUlist\_.pop\_back(); // 从链表移除**

**LRUhash\_.erase(\*frame\_id); // 从哈希表移除**

**return true;**

**}**

**// 固定页面(不可被淘汰)**

**void LRUReplacer::pin(frame\_id\_t frame\_id) {**

**std::scoped\_lock lock{latch\_};**

**auto it = LRUhash\_.find(frame\_id);**

**if (it != LRUhash\_.end()) { // 如果存在于替换器**

**LRUlist\_.erase(it->second); // 从链表移除**

**LRUhash\_.erase(it); // 从哈希表移除**

**}**

**}**

**// 取消固定页面(可被淘汰)**

**void LRUReplacer::unpin(frame\_id\_t frame\_id) {**

**std::scoped\_lock lock{latch\_};**

**if (LRUhash\_.find(frame\_id) == LRUhash\_.end()) { // 如果不在替换器**

**LRUlist\_.push\_front(frame\_id); // 加入链表头部**

**LRUhash\_[frame\_id] = LRUlist\_.begin(); // 更新哈希表**

**}**

**}**

* 1. **数据结构说明**

**private:**

**std::mutex latch\_; // 互斥锁**

**std::list<frame\_id\_t> LRUlist\_; // 双向链表(LRU顺序)**

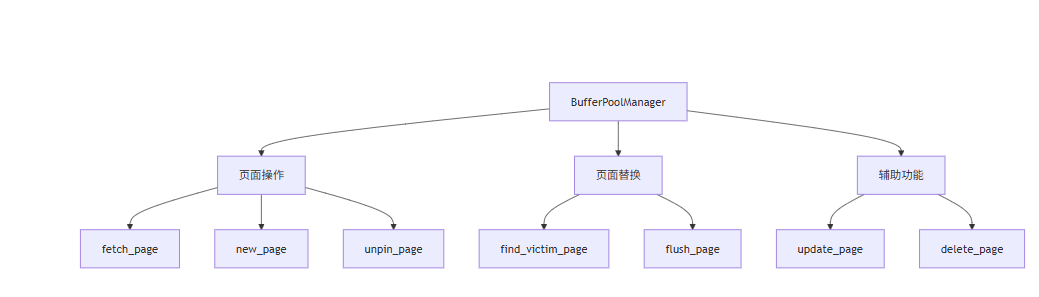
**std::unordered\_map<frame\_id\_t,**

**std::list<frame\_id\_t>::iterator> LRUhash\_; // 哈希表加速查找**

**size\_t max\_size\_; // 最大容量**

1. **缓冲池管理器(BufferPoolManager)**

**3.1 模块架构**

****

* 1. **完整函数实现**

**3.2.1 核心操作函数**

**// 获取页面**

**Page\* BufferPoolManager::fetch\_page(PageId page\_id) {**

**std::scoped\_lock lock{latch\_};**

**// 1. 检查页表**

**if(page\_table\_.count(page\_id)) {**

**frame\_id\_t frame\_id = page\_table\_[page\_id];**

**replacer\_->pin(frame\_id); // 固定页面**

**pages\_[frame\_id].pin\_count\_++;**

**return &pages\_[frame\_id];**

**}**

**// 2. 寻找可替换帧**

**frame\_id\_t frame\_id;**

**if(!find\_victim\_page(&frame\_id)) return nullptr;**

**// 3. 处理脏页**

**Page\* page = &pages\_[frame\_id];**

**if(page->is\_dirty()) {**

**disk\_manager\_->write\_page(page->get\_page\_id().fd,**

**page->get\_page\_id().page\_no,**

**page->get\_data(),**

**PAGE\_SIZE);**

**}**

**// 4. 读取新页面**

**disk\_manager\_->read\_page(page\_id.fd, page\_id.page\_no,**

**page->get\_data(), PAGE\_SIZE);**

**// 5. 更新元数据**

**update\_page(page, page\_id, frame\_id);**

**replacer\_->pin(frame\_id);**

**page->pin\_count\_ = 1;**

**return page;**

**}**

**// 新建页面**

**Page\* BufferPoolManager::new\_page(PageId\* page\_id) {**

**std::scoped\_lock lock{latch\_};**

**// 1. 寻找可用帧**

**frame\_id\_t frame\_id;**

**if(!find\_victim\_page(&frame\_id)) return nullptr;**

**// 2. 分配页面编号**

**page\_id->page\_no = disk\_manager\_->allocate\_page(page\_id->fd);**

**// 3. 更新页面数据**

**Page\* page = &pages\_[frame\_id];**

**update\_page(page, \*page\_id, frame\_id);**

**// 4. 固定页面**

**replacer\_->pin(frame\_id);**

**page->pin\_count\_ = 1;**

**return page;**

**}**

**3.2.2 辅助函数**

**// 寻找可替换帧**

**bool BufferPoolManager::find\_victim\_page(frame\_id\_t\* frame\_id) {**

**if(!free\_list\_.empty()) { // 优先使用空闲帧**

**\*frame\_id = free\_list\_.front();**

**free\_list\_.pop\_front();**

**return true;**

**}**

**return replacer\_->victim(frame\_id); // 使用LRU策略淘汰**

**}**

**// 更新页面数据**

**void BufferPoolManager::update\_page(Page\* page, PageId new\_page\_id, frame\_id\_t new\_frame\_id) {**

**// 写回脏页**

**if(page->is\_dirty()) {**

**disk\_manager\_->write\_page(page->get\_page\_id().fd,**

**page->get\_page\_id().page\_no,**

**page->get\_data(),**

**PAGE\_SIZE);**

**}**

**// 更新页表**

**page\_table\_.erase(page->id\_);**

**if(new\_page\_id.page\_no != INVALID\_PAGE\_ID) {**

**page\_table\_[new\_page\_id] = new\_frame\_id;**

**}**

**// 重置页面**

**page->reset\_memory();**

**page->id\_ = new\_page\_id;**

**page->is\_dirty\_ = false;**

**page->pin\_count\_ = 0;**

**}**

**3.3 关键数据结构**

**private:**

**size\_t pool\_size\_; // 缓冲池大小**

**DiskManager\* disk\_manager\_; // 磁盘管理器**

**std::deque<frame\_id\_t> free\_list\_; // 空闲帧列表**

**std::unordered\_map<PageId, frame\_id\_t> page\_table\_; // 页表**

**Page\* pages\_; // 页面数组**

**Replacer\* replacer\_; // 页面替换器**

**std::mutex latch\_; // 互斥锁**

**四．总结**

1. **基于Docker的标准化开发环境**
2. **Git分支协作与版本控制流程**
3. **包含LRU策略的缓冲池管理系统**
4. **支持并发访问的磁盘**