电子科技大学

实验报告

学 生 姓 名：王冉恒

学　　号：202422900232

课 程 名 称：Linux 网络服务并发设计技术

指 导 教 师：聂晓文

日　　期：　2025 年　4 月　18 日

实验项目名称：基于 iouring 的文件读写

报 告 评 分：_____　教师签字：_____

一、实验目的

1. 在 muduo 框架下扩展 EventLoop 类，集成 io_uring 和 eventfd，实现异步文件读写功能。
2. 设计 File 类封装文件操作，通过 io_uring 提交读写请求，利用事件驱动机制处理完成事件。
3. 验证异步文件读写的正确性和性能，通过 Google Test 框架编写测试用例。

二、实验环境

操作系统	Fedora
开发语言	C++17
依赖库	Liburing Moduo Google Test
构建工具	CMake

三、设计思路

1.扩展 EventLoop 类

（1） 核心组件:

io_uring: 通过共享的提交队列（SQ）和完成队列（CQ）实现异步 I/O。

eventfd: 用于内核通知用户态 I/O 完成事件。

Channel: 监听 eventfd 的可读事件，触发完成事件处理。

（2） 实现流程:

初始化 io_uring 和 eventfd，并将 eventfd 注册到 io_uring。

通过 Channel 监听 eventfd，当有完成事件时调用 handleCompletions 处理 CQE。

2.File 类设计

（1） 功能封装:

提供 open、close 方法管理文件描述符。

提供 asyncRead、asyncWrite 提交异步读写请求。

新增 asyncReadAppend 方法，从文件末尾读取指定长度的数据。

（2） 依赖注入:

通过 IoUringEventLoop 与 EventLoop 交互，提交 I/O 请求。

四、实现细节

1. 关键代码分析

IoUringEventLoop

(1) 提交请求

```
void IoUringEventLoop::submitRead(int fd, void* buf, size_t count, off_t offset, std::function<void(ssize_t)> cb)
    struct io_uring_sqe* sqe = io_uring_get_sqe(&ring_);
    assert(sqe);
    io_uring_prep_read(sqe, fd, buf, count, offset);
    // 将回调存放在 user_data
    auto cbPtr = new std::function<void(ssize_t)>(std::move(cb));
    io_uring_sqe_set_data(sqe, cbPtr);
    io_uring_submit(&ring_);
}

void IoUringEventLoop::submitWrite(int fd, const void* buf, size_t count, off_t offset, std::function<void(ssize_t)>
    struct io_uring_sqe* sqe = io_uring_get_sqe(&ring_);
    assert(sqe);
    io_uring_prep_write(sqe, fd, buf, count, offset);
    auto cbPtr = new std::function<void(ssize_t)>(std::move(cb));
    io_uring_sqe_set_data(sqe, cbPtr);
    io_uring_submit(&ring_);
}
```

(2) 处理完成事件

```
void IoUringEventLoop::handleCompletions() {
    uint64_t ev_count;
    ::read(efd_, &ev_count, sizeof(ev_count));  // 清除 eventfd 的计数
    struct io_uring_cqe* cqe;
    unsigned head;
    int processed = 0;
    io_uring_for_each_cqe(&ring_, head, cqe) {
        auto cbPtr = static_cast<std::function<void(ssize_t)>*>(io_uring_cqe_get_data(cqe));
        (*cbPtr)(cqe->res);
        delete cbPtr;
        ++processed;
    }
    io_uring_cq_advance(&ring_, processed);
}
```

(3) 初始化函数

```
IoUringEventLoop::IoUringEventLoop()
  : muduo::net::EventLoop()
{
    // 初始化io_uring
    int ret = io_uring_queue_init(1024, &ring_, 0);
    assert(ret == 0);
    // 创建eventfd
    efd_ = eventfd(0, EFD_NONBLOCK | EFD_CLOEXEC);
    assert(efd_ >= 0);

    ret = io_uring_register_eventfd(&ring_, efd_);
    assert(ret == 0);  // 注册 EventFD，以便内核在产生 CQE 时写入该 fd

    // Channel用于监听efd_可读
    efdChannel_.reset(new muduo::net::Channel(this, efd_));
    efdChannel_->setReadCallback(
        [this](muduo::Timestamp) { this->handleCompletions(); }
    );
    efdChannel_->enableReading();
}
```

File 类

(1) 异步追加写

```
void File::asyncWrite(const void* buf, size_t len, off_t offset, std::function<void(ssize_t)> cb) {
    assert(fd_ >= 0);
    loop_->submitWrite(fd_, buf, len, offset, std::move(cb));
}
```

(2) 异步读

```
void File::asyncReadAppend(void* buf, size_t len, std::function<void(ssize_t)> cb) {
    assert(fd_ >= 0);
    off_t size = fileSize();
    if (size < (off_t)len) {
        cb(-EINVAL);
        return;
    }
    off_t offset = size - len;  // 明确读取最后 len 字节
    loop_->submitRead(fd_, buf, len, offset, std::move(cb));  // 显式偏移
}
```

2. 内存管理

每个 I/O 请求的回调函数通过 `new` 分配内存，并在 `handleCompletions` 中 `delete`，确保无内存泄漏。

五、验证与测试

1. 测试用例设计

```
TEST(FileTest, ReadWrite) {
    IoUringEventLoop loop;
    File file(&loop, "test.txt");
    ASSERT_TRUE(file.open());
    const char* msg = "我是王冉恒，学号202422900232\n";
    size_t len = strlen(msg);
    std::vector<char> buf(len, 0);
    file.asyncWrite(msg, len, /*offset*/ 0, [&](ssize_t n) {
    EXPECT_EQ(n, len);
    std::cout << "Wrote " << n << " bytes\n" << std::endl;  // 写入成功提示
    file.asyncReadAppend(buf.data(), len, [&](ssize_t m) {
        if (m > 0) {\
            EXPECT_EQ(m, len);
            EXPECT_EQ(msg, buf.data());
            std::cout.write(buf.data(), m);
            std::cout << "\n" << std::endl;
        } else {
            std::cerr << "Read error: " << m << "\n";
        }
        loop.quit();
        });
    });

    loop.loop();
    file.close();
}
```

如图，使用 google test 作为单元测试框架，主要测试了四个方面，分别是：能否正确打开文件、写入的数据与要求写入的数据长度长度是否相等、读到的数据与之前要求写入的数据长度是否相等、写入的数据与读到的数据是否相等。

2. 测试结果

终端输出：

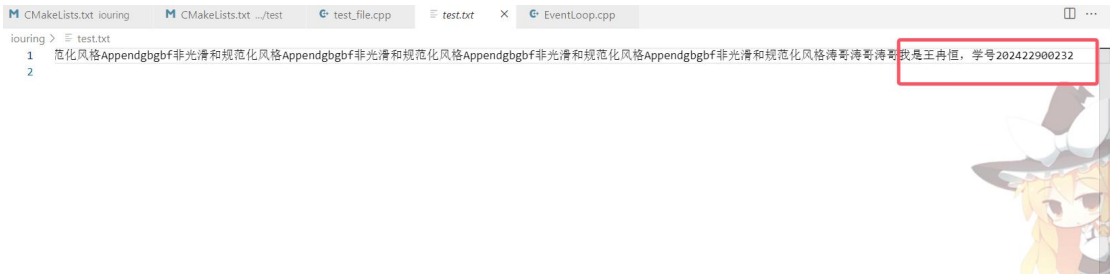
```
wangranheng@wangranheng-virtual-machine:~/project/iouring$ ./test/runTests
Running main() from /build/googletest-j5yxiC/googletest-1.10.0/googletest/src/gtest_main.cc
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from FileTest
[ RUN      ] FileTest.ReadWrite
Wrote 37 bytes

我是王冉恒，学号202422900232


[       OK ] FileTest.ReadWrite (21 ms)
[----------] 1 test from FileTest (21 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (21 ms total)
[  PASSED  ] 1 test.
```

Test.txt 文档：



可以看到之前要求写入的数据，成功写入了文档，且能够正确的读出来

3. 结论

数据写入后能正确读取，验证了异步 I/O 流程的正确性。

六、问题与改进

1. 存在的问题

错误处理：asyncReadAppend 在文件大小不足时返回-EINVAL，未处理部分读取场景。

性能优化：未实现批量提交请求（如 SQE 链），可能影响高并发性能。

2. 改进方向

支持动态调整读取长度，适应文件大小不足的场景。

增加 io_uring 的高级特性（如缓冲池、轮询模式）以提升性能。

七、实验总结

本实验通过扩展 muduo 的 EventLoop 类，结合 io_uring 和 eventfd 实现了高效的异步文件读写功能，并通过 File 类封装了相关操作。测试验证了

基础功能的正确性，后续可进一步优化错误处理和性能。