摘要

异步编程广泛存在于涉及网络或硬盘 I/O 的高性能软件中，且常具有状态管理复杂，程序流程冗长的特征。因此，如何简洁优雅地表达异步逻辑，同时兼顾系统性能是高性能软件开发的难点。目前流行的 C++ 异步解决方案往往采用回调函数和手动管理内存的做法，使得编程门槛较高。本项目设计并完成了一个新颖的 C++ 异步协程框架 co_context，采用全新的 Linux io_uring 协议栈，适配 C++20 的协程标准，接管堆内存的申请和释放，针对 Linux 系统调用建模，统一化管理网络、硬盘、事件、计时器等计算机资源。co_context 以协程的形式提供异步接口，迎合初学者的直觉，使用户只需以同步编程的思维习惯，就能写出高质量的异步 I/O 库和应用，减轻了用户的心智负担，降低了学习成本和开发成本，具备易用性。co_context 保证了在内部实现和用户业务中完全不使用线程互斥锁，并针对 CPU 缓存做了足量优化，以取得更好的性能。在测试中，基于本异步框架重写的经典应用，相比原版应用，在性能上有可观的提升，在代码上显著简化了异步流程。

关键词： 异步框架；协程；io_uring；C++

目录

第一部分　毕业设计

第二部分 毕业设计开题报告

1 绪论

在计算机软件中，任何有意义的程序都需要执行 I/O(Input/Output)。基于操作系统的通用软件的较常用的两种 I/O 分别是网络 I/O 和外存 (例如磁盘，固态硬盘等)I/O。相比于 CPU，网络和外存设备通常具有较高的响应延迟和较低的数据处理吞吐量，因此有必要使用异步的形式处理类似的低速 I/O，以节省 CPU 资源。然而，异步的编程形式长期以来难以调和性能、设计难度和可维护性的矛盾。无论是在验证构想时，还是在构建产品级程序时，为了取得足够高的性能，开发者都需要在异步编程上花费较多的时间和精力，从而挤兑对业务逻辑本身的关注度。因此开发者和企业需要一个良好的异步框架，在保证不损耗性能的前提下，简化异步编程的繁琐步骤，来抵消开发难度和用人成本，更快地完成设计验证或产品研发。

1.1 项目背景及意义

1.1.1 本项目的提出背景

在高性能服务器和 I/O 密集型程序领域，Linux 及其衍生操作系统具备较高的市场占有率。2019 年 Linux 内核 5.1 版本发布，带来新一代 I/O 接口 io_uring[1]，经过三年以来不断迭代，io_uring 得到内核态协议栈应用的青睐。然而，因为历史包袱沉重，已有的热门应用尚未采用这一新生接口；另一方面，虽然基于 io_uring 的实验性应用在通信、数据库[2]等领域给出良好的使用反馈，但由于用户态接口库的原始和开发生态的稚嫩，仍有大量用户难以付出迁移到 io_uring 的经济成本。

在高性能场景下，C++ 是具有较高重要性和占有率的编程语言。但从历史看来，C++ 对异步编程的支持起步较晚。从 C++11 制定的内存模型和多线程标准开始，到 C++20 协程的正式引入为止，C++ 终于具备了高效编写异步程序的基础设施。在此间多年，Java、Go、Node.js 等语言在服务器端的兴起，表明了市场会倾向于一种表达能力强、使用成本低的异步编程开发工具。

io_uring 的兴起和 C++20 协程标准的推出是一个交叉点，提示着为 C++ 编写和推广一套全新的编程友好的异步协程框架的时机已经成熟。

1.1.2 本项目的意义

本项目设计并实现一种新颖的 C++ 异步协程框架，向用户提供同步阻塞式风格的系统调用接口，使用户仅需以同步编程的思维习惯进行产品开发，可以较大简化异步编程的开发和维护流程，缩短创意设计的验证周期或工程的开发周期；同时，在内部维持异步 I/O 的高性能和内存安全性。本框架针对 Linux 系统调用建模，使得已有的基于 Linux 系统调用的上层库（例如 OpenSSL）和应用可以快速移植到本框架上，从而丰富本框架的生态建设。

1.2 相关工作

Boost.asio[3]（简称 asio）是 C++ 中知名度较高的异步事件框架。在其官方用例和大量的现实工程用例中，广泛采用了注册-回调的程序组织形式，大量使用了 shared_ptr 智能指针的内存管理方法。这使得基于 asio 的 C++ 源代码难以编写、阅读和维护。截至 2022 年 3 月，asio 最新版本加入了对协程的支持，但为了兼容原有的接口，其协程的使用方式较为繁琐，未能充分发挥协程的抽象能力和表达能力，用户的使用门槛依然居高。

libco[4]是腾讯开发的 C++ 异步协程库，在微信的开发中广泛使用。这是 C++ 协程具备实用性的现实例子。但 libco 是基于 C++11，其协程功能不受 C++ 编译器支持，大量冗余的编程工作需要由开发者承担，增加了开发者的心智负担，令代码损失了部分简洁之美。阅读 libco 所写的业务逻辑时，仍难免被定式的异步代码分散注意力。

Workflow[5]是搜狗开发的 C++ 服务端引擎，不仅包含了一个异步运行时，还提供了 HTTP、Redis、MySQL 和 Kafka 等热门应用的通信实现。从易用性角度看，Workflow 针对“任务流”建模，采用函数回调的形式，支持任务的“拼装和复用”，是一种代码组织形式的进步。但是，抽象层次的提高也加剧了接口数量

的膨胀。用户在使用 Workflow 时，需要仔细翻查和阅读其 API 文档，也需要理解框架接口的整体组织。从性能角度看，Workflow 的调度器严重依赖于操作系统的互斥锁，在超高并发场景下存在性能瓶颈；大量将 std::function 用作接口使得编译器无法进行内联优化。

Monoio[6]是字节跳动的试验性 Rust 异步运行时，底层采用 io_uring，接口使用协程形式，利用“核绑定线程”最小化跨核通讯开销，目前代表了 Rust 异步运行时的最高性能[7]。

libunifex[8]是 Facebook 的实验性异步框架，采用 Sender/Receiver 模型[9]（简称 S/R 模型），是另一种与协程并列的异步编程模型。善用 S/R 模型可以更充分地发挥编译器的内联能力，从而提高的运行性能。但 S/R 模型本质上未能抛弃回调函数的编程形式，也没有正面解决内存管理困难的问题。

C/C++ 历史相对较长，同类知名的框架和库亦众多，如 libuv[10]，Muduo[11]等等不能全数列举。在众多的选择中，兼具性能和易用性的框架是罕见的；要求不同的框架互相兼容更是难上加难（基于一种框架的上层应用往往难以迁移到另一种框架），因为框架的接口定义各不相同。然而，设计一种具有兼容能力的框架是有可能的，本项目也朝这个方向前进。

1.3 项目内容

1.3.1 co_context：异步协程框架

本项目设计并实现了一个 C++ 异步协程框架——co_context。通过对接内核态 io_uring 协议栈和重新实现同步原语，实现了统一化管理网络、硬盘、事件、计时器、互斥锁等异步资源；利用 C++20 的协程标准，提供了与 Linux 标准系统调用几乎一致的异步 API。内置了完全无锁、缓存友好的调度器，以支撑高并发应用。进一步细化，本项目包括了以下三个模块：io_uring 的用户态接口——liburingcxx；I/O 管理器和协程调度器——io_context；协程库——coroutine。

1.3.2 liburingcxx：io_uring 的用户态接口

io_uring 是 Linux 内核 5.1 版本后引入的内核态高性能 I/O 实现，支持网络 I/O，磁盘 I/O，计时器等功能。为了保证性能，io_uring 的内核态和用户态的数据交换主要通过两条由内核暴露的无锁队列。这导致了用户态提交 I/O 请求相比以往更加麻烦。本项目开发了一套对接内核和用户 I/O 请求的用户态接口库——liburingcxx，以简化后续开发流程。为了提高并发性能，liburingcxx 支持了 I/O 请求的乱序提交，同时利用模板元编程等技术，尽可能地避免动态多态。liburingcxx 是通用的，既能在本项目整体上应用，又能作为单独的 io_uring 库支撑其他项目。

1.3.3 io_context：I/O 管理器和协程调度器

为了向用户提供高效率的通用异步能力，本项目实现了一个管理 I/O 请求和调遣协程的引擎——io_context。基于 liburingcxx，io_context，全权负责向 Linux 内核提交、从内核收割、和暂存 I/O 请求，同时将就绪的协程激活并调遣到合适的线程中运行。在实现上，io_context 采用无锁和完全用户态设计，并针对 CPU 和内存结构进行了优化，平均调度延迟可低至 37 纳秒（在测试环境下）。

1.3.4 coroutine：协程库

为了向用户提供友好的、符合直觉的同步阻塞式风格的编程体验，本项目将框架接口封装为协程形式。基于 io_context，本项目开发了一套简单易用的协程库——coroutine。该协程库不仅给用户提供同步阻塞式风格的系统调用接口，同时提供了协程同步原语 (互斥锁，条件变量，信号量)，以支撑并发应用。由于接口是针对系统调用设计的，所以已有的基于 Linux 系统调用的库（例如 OpenSSL）和应用有机会简单地移植到 co_context 上；corountine 提供的接口具有懒惰求值的特性，用户可以自由组合所需的功能（例如超时取消，链式 I/O 等等），并提供了内存不泄漏的保证。

1.4 本文结构

在当前章节，介绍了异步编程的需求和现状，提出了本报告的工作目标。在第二章，将介绍 co_context 在设计上面临的定位和需求，依此敲定 co_context 的整体设计和所需技术。在第三章，将着重介绍 co_context 在易用性需求上的设计和实现。在第四章，将重点介绍 co_context 在性能需求上的设计和实现。在第五章，将用两套测试对比验证 co_context 的性能指标。在第六章，将总结本设计的成果，并展望 co_context 的未来和提出建议。

1.5 小结

异步编程是一个难题，其难度已经显著左右了目前市场的选择。已有不少的企业从不同的方向向难题发起冲击，而 co_context 针对性能和易用性问题，将在本报告中给出一种新的解答。

2 co_context 设计概述

作为一个异步框架，co_context 的设计和实现有“四个出发点”，本章将解释每种出发点对应的具体需求，以及对应的实现方法：

1. 高性能：发挥 Linux 内核态协议栈的性能；降低框架本身损耗。

2. 易用：提供同步风格的 API；API 可组合；其行为迎合用户的直觉。

3. 内存不泄漏：用户不需要智能指针，仍能保证堆内存正确回收。

4. 上层可移植：基于本框架的库和应用能够迁移到其他同类框架，反之亦然。

2.1 整体需求分析

一个异步框架的核心职责是：当一个操作需要等待软硬件响应时，利用等待的时间，将其他的操作调度到前台。常见的需要等待的操作有网络 I/O、硬盘 I/O、互斥锁上锁、计时器倒计时等等。所以 co_context 面临的需求可总结为：实现这些需要等待的操作（并提供 API），处理调度，并在全过程保证“四个出发点”。

2.1.1 需要等待的操作：阻塞式系统调用

通用软件的等待的一大来源是阻塞式系统调用（另一来源是线程被抢占），例如 accept(2), read(2), pthread_mutex_lock(3) 等。当程序发起这些系统调用，CPU 可能陷入内核态，此时若调用结果没有准备好，操作系统将挂起调用线程，直到调用结果就绪，线程才有机会被唤醒。异步框架为了掌握调度权和保证性能，必然不能让用户线程陷入等待。因此，异步框架必须重新实现（或者等价地完成）阻塞式系统调用的所有功能。

co_context 基于全新的 Linux io_uring，向用户提供了这些阻塞式系统调用接口。co_context 在大多数调用的全过程可分为四个工作阶段：

1. 用户协程将调用请求存放至缓冲区，当前协程暂停，调度另一协程至当前线程。

2. 管理线程将缓冲区内的调用请求转发到 io_uring 的请求队列。

3. 管理线程将 io_uring 的收割队列上的调用结果转发到缓冲区。

4. 用户线程在缓冲区上发现调用结果，恢复对应的协程。

需要指明的是，上述过程的第 2 步与第 3 步中间的处理者是 io_uring。而 co_context 负责了用户态和 io_uring 之间的高效沟通。特别地，针对互斥锁、信号量、条件变量相关调用，co_context 提供了完全用户态的重新实现，从而绕过了 io_uring。

2.1.2 高性能：无锁，无等待，缓存友好

常见多线程框架（例如 Workflow、muduo）的调度器通常是基于任务队列和互斥锁实现的：各个工作线程争抢一个全局的互斥锁，持有锁的工作线程从全局的任务队列中弹出一个任务，并运行之。更进一步的做法是任务窃取（Work-Stealing）：当一个工作线程空闲时，可以争抢其他线程的任务。但这两种方法的共同缺点是，某些线程有可能饿死（Starvation），且在极高并发的场景下，互斥锁会成为性能瓶颈。

为了满足极高并发场景，co_context 的调度器 io_context 需要采用完全无锁和无等待的设计。io_context 具有一个管理者线程，负责将任务主动地推送到某一个工作线程的“线程本地”的“收割缓冲区”，而工作线程不断地扫描属于自己的收割缓冲区，从而发现任务和运行任务。每一个工作线程独享一个缓冲区，对于每个缓冲区而言，管理者线程是唯一的“生产者”（或“写者”），对应的工作线程是唯一的“消费者”（或“读者”）。可以证明，假设空间足够大，单生产者单消费者构成的通信队列是无等待的（Wait-free），从而是无锁的（Lock-free）。在默认配置下，除了原子操作和忙等待（Busy-wait）以外，io_context 不使用其他线程间同步工具，从而取得低响应延迟和高吞吐。但是，在低负载场景下，忙等待会浪费 CPU 资源。所以 io_context 也提供了基于 futex（一种互斥锁）的同步方式，可供用户根据场景选择。

在 co_context 中，管理者线程和工作线程的唯一通信渠道是“提交缓冲区”和“收割缓冲区”，两个缓冲区的原理是类似的（但是数据流动的方向相反）。限制通信渠道的目的是，保证数据跨线程传输的总体速度。通过数据压缩和缓冲（cacheline）对齐，co_context 保证了每个用户请求最多被两个线程读写，分别是用户线程和 io_uring 所在的线程，且 CPU 最少跨核传输次数最多为 6 次。io_context 的管理者线程本身不读写用户请求，只计算请求的内存地址，从而减少了缓冲未命中（cache miss）发生的次数。

无锁，无等待和缓存友好是 co_context 高性能的主要来源。针对计算机体系结构的优化是区分 io_context 和其他常见调度器的显著特征。

2.1.3 易用：同步形式，贴近标准，自由组合

传统的异步编程往往采用函数回调的组织形式，这种形式更接近程序的运行逻辑，对“编译器友好”，但不便于人类的阅读、理解、编写和维护。例如，代码 2.1是 Boost.asio 的官方计时器用例，代码表达的意思是，异步地等待 5 秒钟，随后调用 print() 函数。从中可以发现，函数回调分割了业务逻辑：在 print() 函数中看不见前面的业务过程，在 main() 函数中看不见计时器超时后的工作代码。而且，在函数回调之间传递数据是一件困难的事情，容易导致 shared_ptr 的滥用。随着业务逻辑愈加复杂，需要等待的操作愈多，函数回调的弊病将会愈明显。

代码 2.1: asio 的官方用例

```
void print(const boost::system::error_code& /*e*/) {
  std::cout << "Hello, world!" << std::endl;
}
int main() {
  boost::asio::io_context io;
  boost::asio::steady_timer t(io, boost::asio::chrono::
   seconds(5));
  t.async_wait(&print);
  io.run();
  return 0;
}
```

在现代异步编程中，协程可以避开函数回调的缺点，将异步逻辑以同步的形式表达出来，使人的心智负担大大降低。在 co_context 中，利用计时器制作一个“每秒触发器”的例子如代码 2.2所示。可以看出，“等待一秒钟”和“等待一秒钟前后的任务”是紧密相连，不被分割的。另一个优势是协程不需要在函数回调之间传递参数。在例子代码 2.2中，局部变量 cnt 实际上是在堆空间中分配的，但用户不需要关心这件事，co_context 会负责这些堆内存的释放。co_context 为用户提供了某种意义上的垃圾回收（Garbage Collection，GC），而这一切对用户而言都是无感知的。

代码 2.2: co_context 的每秒触发器

```
task<> my_clock() {
    for (int cnt = 0;;) {
        printf("Time = %d\n", cnt++);
        co_await timeout(1s);
    }
}
```

为了进一步降低用户的学习成本，co_context 提供的 API 是根据 Linux 开发者手册（Linux Programmer’s Manual）和 C++ 标准设计的。如代码 2.3所示，co_context 的 API 与对应标准接口几乎完全一致。因为全世界的开发者都共享同一套 Linux 编程接口，且都遵循同一个 C++ 标准，所以他们看到 co_context 设计的 API 时会感到亲切和熟悉。不只是 API，co_context 的表现行为也基本符合标准定义。凭借有这两点保证，用户在使用 co_context 时甚至可能不需要文档，全凭直觉和常识进行编程。

代码 2.3: co_context 的同步形式 API

```
// ----> read系统调用
// Linux标准同步接口
int nr = read(fd, buf, count);
// co_context接口
int nr = co_await read(fd, buf, count);
```

```
// ----> 互斥锁
// C++标准接口
std::mutex mtx;
mtx.lock();
// co_context接口
co_context::mutex mtx;
co_await mtx.lock();
```

用户可能并不满足于一般的系统调用，例如，用户想为某个 I/O 操作限时。为此，co_context 需要提供高阶的 API，并且这些 API 可以自由组合。如代码 2.4所示，由基础的 API 可以逐步拼接出复杂的操作，全过程易于理解，表达能力强。

代码 2.4: co_context 的自由组合 API

```
// 计时器等待3秒钟
co_await timeout(3s);
// TCP recv 调用
int nr = co_await sock.recv(buf);
// ------> TCP recv 限时3秒钟
int nr = co_await timeout(sock.recv(buf), 3s);
// ------------> TCP recv 限时3秒钟，然后关闭 socket
co_await (timeout(sock.recv(buf), 3s) && sock.close());
```

2.1.4 内存不泄漏：接管堆内存

在网络应用中，网络连接和服务的数量是不确定的，因此对应需要的内存是不确定的，所以异步编程通常伴随着动态堆内存分配。在传统的函数回调程序中，通常每个独立的连接和服务独享着若干堆内存，而这些堆内存的地址在多个回调函数之间传递。此时，为了避免堆内存泄漏，回调函数的接口通常会传递 shared_ptr。而 shared_ptr 本身有性能低下、污染接口、学习成本高的缺点。在 co_context 中，这个问题可以避免。

虽然网络连接和服务的数量是不确定的，但一个连接和服务所需的内存通常是确定的。在 co_context 中，用户欲申请堆空间，只需要定义局部变量即可。如代码 2.5所示。当局部变量进退作用域时，变量的行为会遵循 RAII（Resource

Acquisition Is Initialization），即正确调用构造函数和析构函数；当协程结束时，所申请的堆内存会被释放。

相比函数回调形式，co_context 避开了在多个函数之间传递堆内存，全局抛弃了笨重低效的 shared_ptr，自动管理堆内存的分配和释放。从用户角度看，co_context 为用户提供了符合 RAII 原则的高性能垃圾回收机制。从 Java、Go、Node.js 等语言的流行来看，内存不泄漏是一个非常大的优势。

代码 2.5: co_context 申请堆内存

```
task<> example(socket sock) {
    char buf_0[256];
    int nr = co_await sock.recv(buf_0);
    char buf_1[1024];
    // ...
    int nw = co_await sock.send(buf_1, length);
    char buf_2[8000];
    // ...
}
```

2.1.5 上层可移植：易用性的推广

在2.1.3中介绍了依据标准的 API 设计带来的易用性，而遵循标准的另一个优势是基于 co_context 的上层库和上层应用能够容易地迁移到另一个遵循标准的框架中。反之，严格按照 Linux Programmer's Manual 和 C++ 标准设计的程序，也容易移植到本框架上。一定的可移植性意味着不需要为 co_context 从头设计一套新的应用层，缩短了一个新生框架的生态荒漠时期。固然，可移植性的价值不在当下，而体现在对未来工作的方便，所以本报告的重点不在于此。

2.2 co_context 的定位和整体结构

2.2.1 co_context 的定位

在一个完整的应用中，co_context 的定位是异步资源和协程调度的管理者，为应用和库屏蔽了线程概念，处于相对底层的位置。如图 2.1所示，用户使用 co_context 提供的基础协程 I/O 构建自己的协程业务库，然后利用协程业务库和 co_context 提供的调度器组合成最终应用。

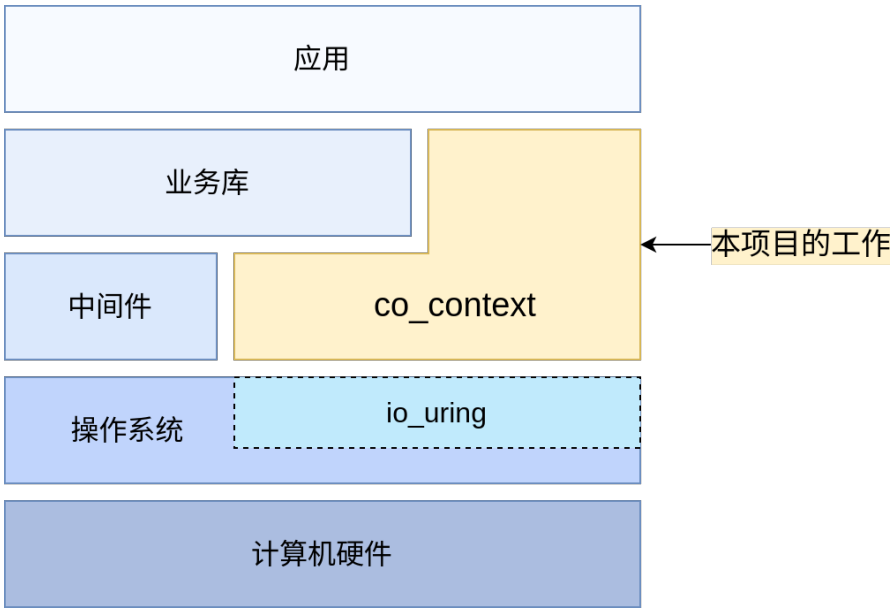

图 2.1 co_context 在实际应用中的定位

2.2.2 co_context 的整体结构

在1.3中简单介绍后，如图 2.2所示，co_context 的整体结构清晰：针对并发场景的 io_uring 用户态接口库——liburingcxx；提供高性能调度能力的调度器——io_context；提供易用灵活 API 的协程库——coroutine。其中 liburingcxx 可以单独成库，供给其他异步框架使用。出于性能考虑，三者互为白盒，具有一定的耦合性。考虑到一个异步框架应有的体量不大，以及用户不需要了解 co_context 的内部实现的事实，co_context 在提高性能和内部解耦之间选择了前者。

在设计业务库时，用户不需要一个具体的调度器，甚至可以忽略调度器的存在，只需利用 co_context 提供的协程库，将业务逻辑组合拼凑出来即可。在顶层应用中，用户必须定义一个 io_context 调度器，并设定线程数量等参数，随后即可直接调用业务库函数。co_context 利用“线程本地”变量优化了接口设计，使得业务库不需要传入具体的 io_context 调度器作为参数。

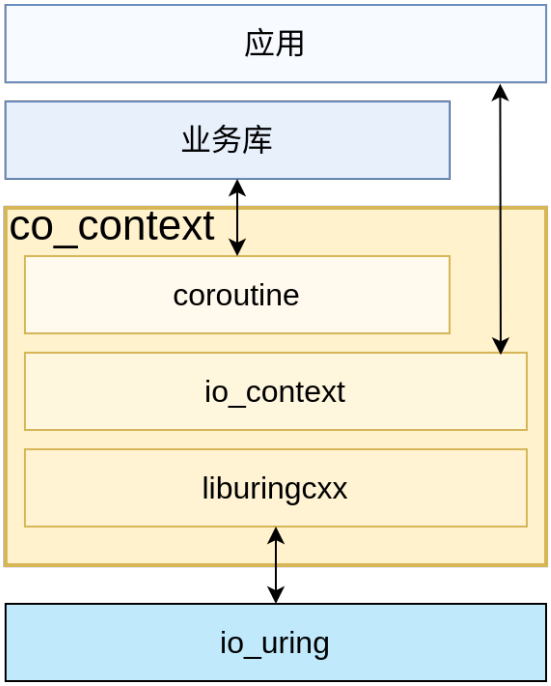

图 2.2 co_context 的整体结构和调用关系

2.3 关键技术分析

2.3.1 io_uring 浅析

io_uring 是 Linux 内核 5.1 版本后引入的内核态模块，可用于进行网络 I/O、硬盘 I/O 等耗时操作。在接口设计上，io_uring 采用完全异步的环形队列通信，在中高负载下，用户态和内核态的信息交换完全不经过系统调用，从而避免 CPU 上下文切换的开销。使得 io_uring 的综合性能在理论上高于传统的 Linux Epoll。

io_uring 的另一特色是遵循 Proactor（前摄器）模式。其含义是内核会负责将网卡等硬件上的数据拷贝到用户的内存中，待拷贝结束后才会通知用户。与

Proactor 模式相对立的是 Reactor（反应器）模式。其特点是在数据到达内核时立即通知用户，并将拷贝数据的责任抛给用户。两种模式孰优孰劣，视乎具体的场景。由于 io_uring 追求极致少的 CPU 上下文切换，Proactor 的每次 I/O 最多只需一次系统调用，而 Reactor 的每次 I/O 都需要两次系统调用，所以很自然地选择了 Proactor 模式。

因为 io_uring 在原理上具有更好的设计和性能，所以 co_context 选择了它作为底层。当然，另外还有两个不成文的原因：一是 co_context 默认采用了内核态 I/O，这也是大多数异步框架的选择；二是 io_uring 的用户态接口库屈指可数，相比 io_uring 原作者的 C 语言接口库 liburing，co_context 所设计的 liburingcxx 又具有更好的并发灵活性和运行时性能，从而更有可能得到社区的青睐。

2.3.2 协程浅析

协程是可暂停、可再入的函数。协程概念出现的时间比线程还要早，所以，协程的运行本质上与线程无关。在 C 语言中，可以用一个函数加一个结构体来实现任意的协程：只需在结构体内储存状态和局部变量，并在函数内对状态做“switch/case”，跳转至状态对应的代码，即可令这个函数具有“可暂停”“可再入”的外在表现，于是实现了协程。上述实现过程是机械重复性的，应该交给编译器或解释器来完成，于是出现了 C++20 协程标准。

C++20 协程标准是 2018 年通过 C++ 标准委员会审议，2020 年被主流编译器基本实现完毕的新标准。C++20 协程带来的新功能非常简陋，无法被上层用户直接使用，必须由开发者设计一套完整的协程库（这也是本项目的工作），才有可能投入生产使用。对于协程库的作者（例如本文作者）而言，新标准的最大好处是编译器会自动收集、打包协程中定义的局部变量成一个匿名结构体（存放于堆内存上），并且自动生成一个协程状态机，使得协程库不再需要从汇编的角度上进行调度设计。语言级别的支持也意味着有可能随着编译器的更新，协程可以获得免费的性能提升。

上述的优点，恰恰是在 C++20 以前的协程库的缺点。像腾讯的 libco 这样早

期的 C++ 协程库，大量使用了与平台有关的汇编指令作为协程调度（导致隐藏了多年后才被发现的 BUG），并且在使用协程局部变量时必须自定义结构体函数，用户的心智负担非常大。如今有编译器的加持，C++ 的协程功能终于有可能变得优雅。因为 co_context 同时追求高性能和易用性，C++ 和协程几乎是唯一的选择，所以 C++20 协程标准的采用是顺理成章。

2.4 小结

co_context 是接近底层的异步框架，为上层提供易用、高性能的协程服务。在整体设计上，通过无锁、缓存优化、协程式标准化 API 等顶层设计，co_context 具有良好性能、易用性、内存不泄漏等强大特性，借助 Linux io_uring 和 C++20 协程的东风，co_context 具备先声夺人的基础和获得社区的青睐的可能性。

3 co_context 的易用性实现

近几年在互联网上 Java、Go、Python 等有 GC 语言十分火爆，而 C++ 在招聘市场上的冷清与之形成鲜明对比。C++ 相对复杂的语法和规则会导致开发周期太长、维护成本太高的问题，在日新月异的业务场景下常常是不可接受的，迫使大量的开发者和企业选择其他开发工具。为了能参与市场竞争，co_context 的设计和实现必须克服易用性难题。

3.1 易用性需求分析

3.1.1 代码噪声小

试想当一个用户在寻找一款新的异步框架，那么这名用户大概是不喜欢已有的框架，因为过于难用而不得不放弃。当这名用户翻到 co_context 的宣传文章，头十秒钟阅读的可能是一个 co_context 用例，而这个用例决定了用户有没有兴趣继续阅读文档，所以理想的情况是，一个没有异步编程经验的人可以在 10 秒内读懂用 co_context 写的代码。例如，请尝试用 10 秒阅读并理解代码 3.1。

代码 3.1: co_context-echo

```
task<> echo(socket sock) {
    char buf[8192];
    int n = co_await sock.recv(buf);
    while (n > 0) {
        co_await sock.send({buf, n});
        n = co_await sock.recv(buf);
    }
}
```

在代码 3.1中，若读者知道 socket 的基本概念，并忽略 co_await 关键字，那么在 10 秒内理解代码的意思应该并不困难：程序从 socket 中读取一段字节流，并沿原路发送回去，周而复始。这就是取得易用性的需求关键：代码噪声小，删除“co_await”之后，剩余全部都是业务逻辑。

表 3.1 co_context 中符合 Linux 定义的 API

read	recvmsg	fsync	unlinkat	tee
readv	send	sync_file_range	renameat	
write	sendmsg	fallocate	mkdirat	
writev	connect	openat	symlinkat	
accept	close	openat2	linkat	
recv	shutdown	statx	splice	

3.1.2 符合标准

吸引一个群体的一种办法是满足这个群体的标准。作为一款在 Linux 上运行的 C++ 框架，要满足的就是符合 Linux Programmer's Manual 和 C++ 标准。在代码 2.3中，已经给出了两个例子。co_context 面临的需求是，除了“co_await”以外，接口的使用方法要与标准定义的一致。表 3.1展示了在 co_context 中实现的，且在 Linux Programmer's Manual 中有定义的接口。读者可使用 man 命令或者在 man7.org 网站上查询到这些接口的定义。

co_context 在实现同步原语时，要遵循的是 C++ 标准。代码 3.2给出了对比用例。模仿 C++ 标准不仅方便了用户迁移到 co_context 上，还能避免设计上的失误。

代码 3.2: co_context 同步原语，对比 C++ 标准

```
// C++标准信号量
std::counting_semaphore sem{number};
sem.acquire(n);
sem.release(m);
// co_context信号量
co_context::counting_semaphore co_sem{number};
co_await sem.acquire(n);
sem.release(m);
```

```
// C++标准互斥锁
std::mutex mtx;
mtx.lock();
mtx.unlock();
bool succ = mtx.try_lock();
// co_context互斥锁
co_context::mutex mtx;
co_await mtx.lock();
mtx.unlock();
bool succ = mtx.try_lock();

// C++标准条件变量
cv.wait(lock, [&] { return ready; });
cv.notify_one();
cv.notify_all();
// co_context条件变量
co_await cv.wait(mtx, [&] { return ready; });
cv.notify_one();
cv.notify_all();
```

3.1.3 难以误用

这一需求是为没有仔细阅读文档的用户准备的。对比原始真正的同步编程，co_context 多出了两个不同点。一是要在每个异步操作前面添加“co_await”；二是需要将函数返回值“xx”改为协程返回值“task<xx>”。如果粗心的用户忘记了，co_context 在编译期应当具备纠错能力。利用 C++20 的属性说明（Attribute specifier）功能，这种需求的实现并不困难。

3.1.4 自由组合

Java 中的流操作（Stream）备受好评，受此启发，co_context 提供的 API 也具有自由组合的能力。用户可以将计时器和 I/O 操作组合起来，变成一个限时的 I/O 操作；还可以令一个 I/O 链接上另一个 I/O，重复若干次，形成一连串 I/O 操

作。代码 2.4已经给出了一些用例，这些形象直观的用法令 co_context 的高阶功能更加容易理解和上手。

3.2 协程的实现：易用性的核心

在介绍 co_context 时总是三句话离不开协程，正因如此，“co_context”的前缀取自“coroutine”。本节以 read 系统调用为例，从用户的视角看待 co_context 是如何一步步实现 I/O 和调度的。在这之前，有必要先简单介绍 C++20 协程的运作方式。

3.2.1 C++20 协程的运作方式简介

C++20 协程是一个具有特殊返回值的函数，在 co_context 中，返回值是 task<xx> 的函数就是协程（其中 xx 可以是任意类型，表示协程的返回值类型）。当创建一个协程时，程序（通常）会调用 operator new 来构造一个协程帧，协程帧内部存放着必要的局部变量、协程参数和状态标记，以便支持协程的暂停、恢复、返回值、抛出异常等功能。协程创建后，会有一个协程句柄（coroutine_handle），可以理解为指向协程帧的指针。任何调度器都是利用协程句柄来调度协程。

创建一个协程后，协程通常不会立即运行（是否立即运行取决于开发者），这种模式被称为“懒惰求值”。“创建一个协程”本身的返回值是一个由开发者定义的 return_object，里面通常包含着对应的协程句柄。拿到协程句柄后，就可以将句柄放进调度器里调度了。

在 C++20 协程标准中，最重要的关键字是“co_await”。它是一组复杂操作的语法糖。co_await 是一元运算符，只能在协程内使用。“co_await expr”的通常含义是：等待 expr 的求值，若 expr 求值完毕，则返回其内部定义的返回值；若 expr 尚未就绪，则当前协程暂停，并且交出协程句柄（通常这个暂停的协程的句柄会在未来某个时刻被再次调度），此时开发者可以指定是否恢复任意协程，或者直接做函数返回（相当于 ret 指令），co_context 在此时获得控制权并运行调度函数，选出就绪的协程并恢复运行之。由此看来，每当调用 co_await 进行异步

I/O，co_context 都能获得控制权，从而能完成与 io_uring 的交互，并且调度协程。

co_await 的完整流程比较复杂，本报告制作了图 3.1以便初步理解，若读者对此感兴趣，建议参考 C++20 标准或者网络上的资料，并配合图 3.1理解。

3.2.2 co_context 的运行机制：以 read 调用为例

本节以 read 调用为例，介绍 co_context 的易用性协程 API 是如何工作的。read 调用用于读取一个文件的内容，需要指定读取的长度和开始读取的位置。代码 3.3是接下来要分析的用例，这段代码尝试读取文件的前 32 个字节，并打印到终端上。

代码 3.3: 使用 read 调用的协程例子

```
task<> print_head(int fd) {
    char buf[33];
    // 尝试读取文件的前32个字节
    int nr = co_await read(fd, {buf, 32}, 0);
    // 添加结束符并打印到终端
    buf[nr] = '\0';
    printf("%s\n", buf);
}
```

第一步：生成请求结构体 Q。代码 3.3中 read 是一个普通的函数（而非协程）。read 返回一个 lazy_read 结构体，称之为 Q。Q 在构造函数中，保存了 read 调用的实参。此时 read 函数结束，程序执行至 co_await Q。

第二步：调用 Q.await_ready()。co_await 关键字首先判断请求 Q 是否已经就绪。由于此时 read 调用还没实际发生，所以 Q.await_ready() 总是返回 false，表示 Q 未就绪，这会导致接下来的步骤。

第三步：协程已暂停，调用 Q.await_suspend(handle)。由于 Q.await_ready() 返回 false，协程会陷入暂停，然后调用 Q.await_suspend(handle)，其中 handle 表示当前协程（即 print_head）的协程句柄。在 await_suspend 函数中，程序将 read 的请求参数和协程句柄打包写入“提交缓冲区”（缓冲区的概念已在2.1.2中介绍过），await_suspend 函数结束后不恢复任何协程，而是令程序退回函数调用栈的上一

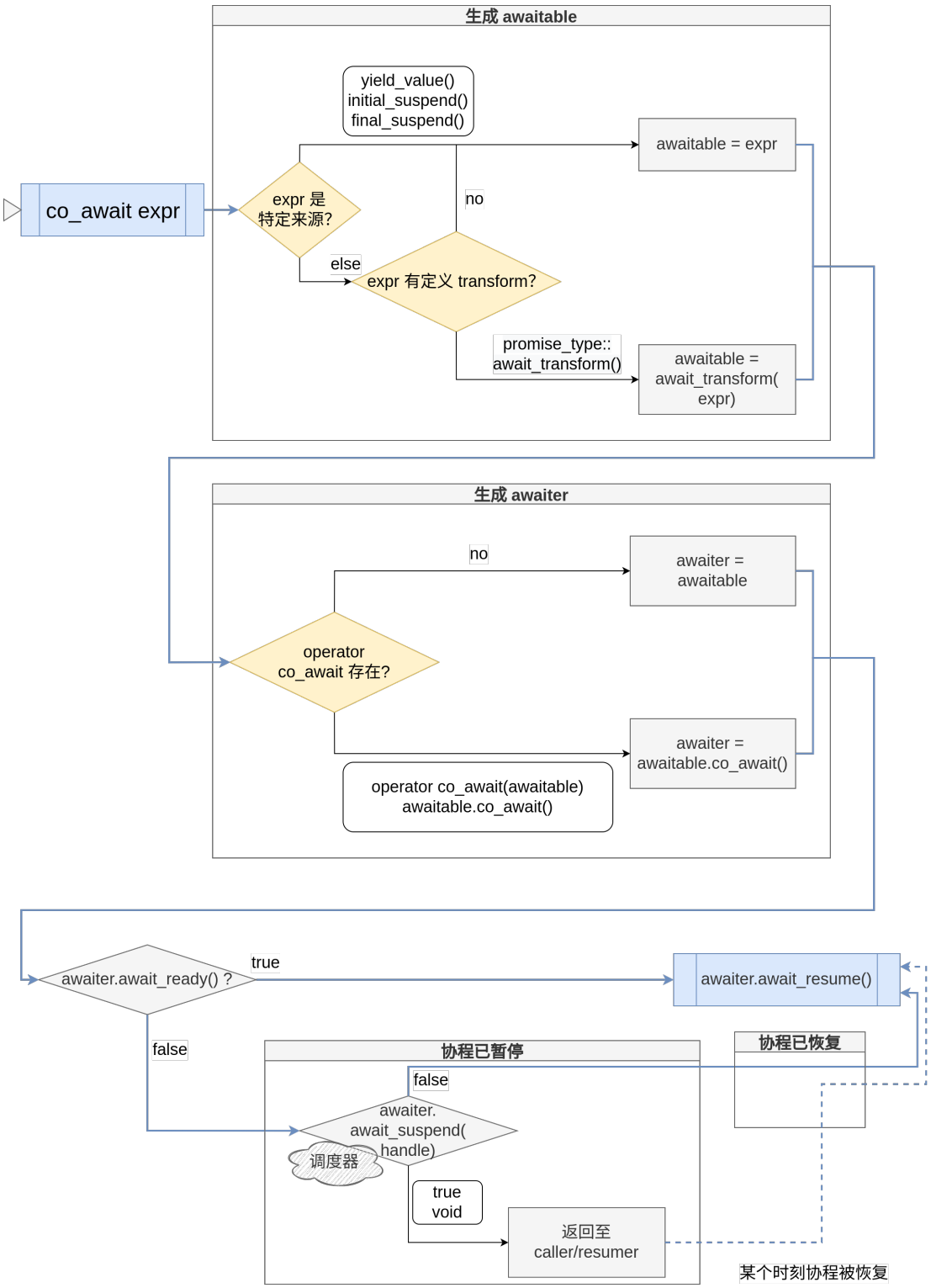

图 3.1 co_await 的完整流程

层，即 co_context 的工作者调度器，（如代码 3.4所示，此时对应“coro.resume()”调用返回）。

代码 3.4: 伪代码：工作者调度器

```
void worker::worker_loop() {
    while (true) {
        coroutine_handle coro = this->schedule();
        coro.resume(); // 协程恢复或开始
        // 此处协程暂停或结束
    }
}
```

第四步：工作者调度器轮询收割缓冲区，恢复另一协程。工作者调度器的大致设计如代码 3.4所示，此时“coro.resume()”调用结束，工作者调度器运行“this->schedule()”再次选出一个就绪协程，并且恢复运行之。由 schedule 函数负责轮询“收割缓冲区”，它会将已完成的 I/O 操作的结果写入协程帧，于是对应的协程进入就绪状态，所以令其恢复执行。总结前面的步骤，原来的 print_head 协程已经被暂停并向“提交缓冲区”提交 I/O 请求，工作者调度器找到另一个就绪的协程，并且恢复执行之。

第五步：管理者轮询提交缓冲区，向 io_uring 提交 I/O 请求。某时刻管理者线程轮询提交缓冲区，发现刚刚提交的 read 调用，于是将其转发到 io_uring 中。由于管理者线程只有一个，不需要考虑数据竞争的问题。管理者线程继续其他工作。

第六步：管理者轮询 io_uring 的 I/O 结果，转发至收割缓冲区。某时刻管理者线程轮询 io_uring 的结果，发现 read 调用已经完成，于是将其结果转发到某个工作者的收割缓冲区。前六步可以概括为图 3.2。

第七步：在第四步中，print_head 协程被恢复。由于工作者调度器轮询收割缓冲区，必然会发现 read 调用的结果，此时将 I/O 结果写入 Q，恢复对应的 print_head 协程。co_await 关键字最后调用 Q.await_resume() 的结果作为返回值。最终，变量 nr 获取到正确的结果，全过程结束。

暂停需要等待的任务，恢复就绪的任务，周而复始，是异步框架的核心职

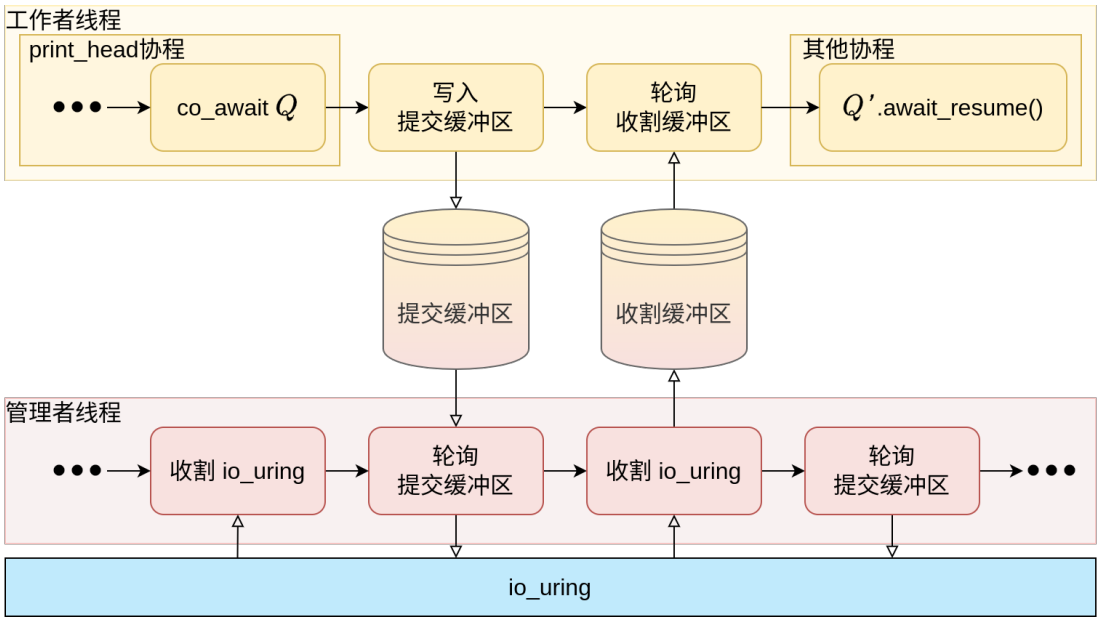

图 3.2 co_context 在 co_await 时的流程

责。co_context 使用协程的形式完成这些工作，将大量的啰唆代码隐藏在 C++ co_await 关键字之下，呈现在用户眼中的只余下纯粹的业务逻辑，实现了代码噪音小的目标。

3.3 其他易用性需求的实现

3.3.1 API 自由组合的实现

co_context 提供的默认 API 具有惰性求值的特性（也另提供了立即求值版本的 API），这些惰性 API 都会返回一个请求结构体 Q。co_context 针对各类 Q 的基类重载了“&&”运算符，并提供了 timeout 接口，以实现自由组合。如图 3.3所示，利用 C++ 的语法规则（&& 运算符的特殊性），co_context 在编译期将一条表达式转化为树型结构，并在运行期将树型调用转化为链表结构。co_context 在处理该链表时，会转化为符合 io_uring 规定的“LINK_IO”形式后再全部提交至 io_uring 的提交队列中。io_uring 能够正确处理这些链式调用，若中途任意调用发生错误，则链上后继的调用也认为是失败。co_context 在处理“LINK_IO”的结果时，默认只在最后一个 I/O 操作完成时才会恢复对应的协程。从用户的角度

看，当协程恢复时，所有的操作都保证已经完成；若中途任意 I/O 发生错误，也会导致协程恢复，并返回错误码。

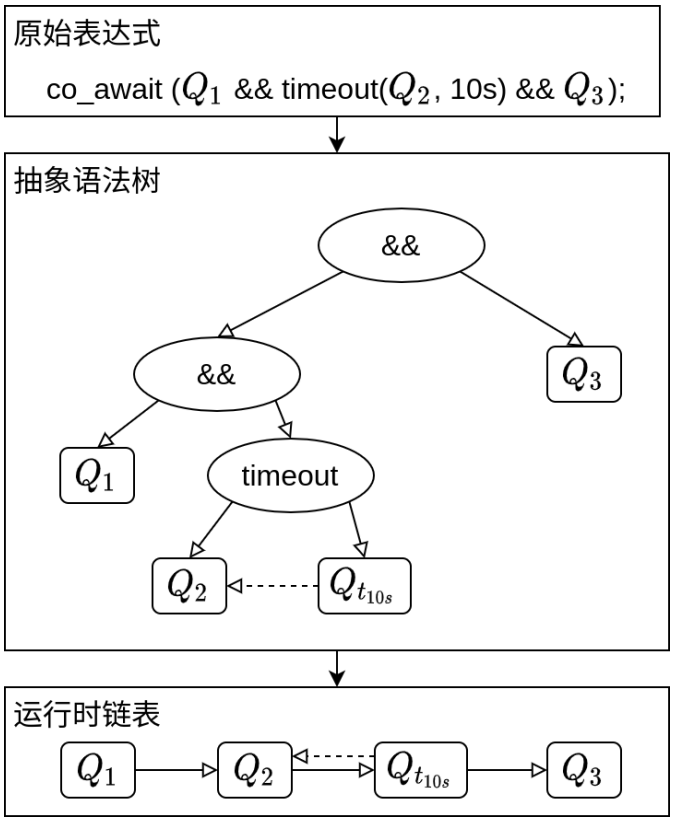

图 3.3 API 自由组合的基本原理

3.3.2 编译期提示的实现

不熟悉协程的新用户可能会忘记使用 co_await 关键字。可以分为两种情况讨论，一是用户用错误的类型来接收 API 的返回值；二是用户抛弃了 API 的返回值。如代码 3.5所示。针对第一种情况，co_context 将 API 的返回值类型隐藏到 detail 命名空间之下，防止用户不小心访问，因而编译器必然会抛出“类型不匹配”的错误报告，从而提醒用户修正。针对第二种情况，co_context 在所有需要被 co_await 的类型上都添加了 C++ nodiscard 标签，并配有提示文字，如代码 3.6所示。所以，用户会收到编译器的提示：“Did you forget to co_await?”，直观清楚地定位错误，从而修正代码。

代码 3.5: 误用 API 的例子

```
int nr = read(fd, {buf, 32}, 0); // 返回值类型不匹配
read(fd, {buf, 32}, 0);          // 返回值被抛弃
```

代码 3.6: 伪代码：使用 C++ nodiscard 标签

```
struct lazy_read {
    [[nodiscard("Did you forget to co_await?")]]
    lazy_read(/*...*/) {
        // ...
    }
};
```

3.4 小结

本章介绍了 co_context 是如何设计并实现易用性目标。co_context 基于 C++20 协程提供了简洁易懂的 API，特别压缩了代码噪音；不仅如此，co_context 还充分挖掘了 API 惰性求值的潜力，允许用户像搭积木一样拼凑出业务所需的高级功能。同时，co_context 还具备一定的防呆设计，方便没有仔细阅读文档的用户也能正确 co_context。

4 co_context 的性能实现

运行时性能，是衡量一个异步框架好坏的重要指标。通常，开发者和企业更关注时间性能，即每秒数据量、每秒请求数、平均延迟、中位数延迟、长尾延迟等指标。为了取得更好的时间性能，co_context 对比了多种常见的调度器实现方式，选择了比较激进的做法。本章将会介绍本框架在性能上的工作及其创新。

4.1 彻底无锁化

互斥锁，一般是指由操作系统提供的同步工具，用于防止多个线程的并发操作产生竞争条件（Race condition）。互斥锁最多被一个线程持有；若一个线程欲占有一个已经被其他线程持有的互斥锁，则它必须等待，直到互斥锁被释放。线程的等待功能是由操作系统实现的，意味着线程在陷入阻塞和被唤醒时必然发生用户态和内核态之间的转换，产生上下文切换，导致时间性能的损失。一些知名的异步框架如 Workflow，muduo 等在多线程调度过程中均使用了互斥锁。然而，在极高并发场景下，互斥锁竞争非常激烈，工作线程会频繁地陷入阻塞和唤醒，亦即产生高频率的上下文切换，浪费了 CPU 资源。即使忽略上下文切换的开销，从整体角度看，框架的最高并发性能也将受到一个互斥锁的钳制。基于此，co_context 决定彻底抛弃互斥锁和其他阻塞同步工具，彻底避免主动线程阻塞。

4.1.1 定长单生产者单消费者队列

多线程协作模式总是可以归结为“生产者-消费者”模型。不同线程之间的通讯必然需要某种中间数据结构作为数据交换区域。在各种生产者-消费者数据结构中，理论并发性能最好的是定长单生产者单消费者队列（Fix-sized Single-Producer-Single-Consumer Queue），简称定长 SPSC 队列。实现一个定长 SPSC 队列不需要操作系统参与，只需要使用由硬件提供的原子操作，更美好的是，完全不涉及“读-改-写”类的原子操作。（通常，“读-改-写”原子操作需要锁定计算机总线，从而拖慢 CPU 整体速度）

在定长 SPSC 队列中，有两个线程：一个生产者，一个消费者；有一条环形队列（Ring buffer），和对应的头指针（head）和尾指针（tail）。两个指针都是原子变量，但只需要支持“读”和“写”，不需要“读-改-写”。当生产者要生产数据时，首先判断队列是否已满。生产者需要原子地读取头指针的值，比较它和尾指针即可知道队列是否已满。若队列未满，则生产者将数据写入尾指针所指向的环形队列空间，最后将尾指针的值增加一（仅使用原子写），完成一次生产。当消费者要消费数据时，操作是十分对称的。消费者首先判断队列是否为空，此时要原子地读取尾指针的值，比较它和头指针即可知道队列是否为空。若队列不是空的，则读取头指针指向的数据，最后将头指针的值增加一（仅使用原子写），完成一次消费。在 x86 架构上，定长 SPSC 队列不需要任何同步指令，从而具有非常高的性能。

co_context 和 io_uring 大量使用了定长 SPSC 队列。需要指出的是，使用 SPSC 队列时必须仔细指定内存顺序，以防止错误的编译器指令乱序优化和 CPU 指令乱序优化。受篇幅限制，本报告不再详细介绍内存顺序。

4.1.2 调度器无锁化

常见的多线程调度器并没有单独的管理线程。一般是多个工作线程共享同一个任务队列，每个工作线程都争抢一个互斥锁，只有持有互斥锁的线程有权修改任务队列。这种做法保证了任务队列的线程安全性，每个任务既不会被重复执行，又不会被遗漏。而 co_context 的思想是，设置一个管理者线程，它有非常高的特权：

1. 有权与 io_uring 通信。

2. 有权决定每个任务将分配到的工作线程。

3. 有权读取每个工作者线程的请求。

因为管理者线程是唯一的，所以所有管理者-工作者都构成单生产者-单消费者（Single-Producer-Single-Consumer，简称 SPSC）关系；管理者与 io_uring 也构

成 SPSC 关系，所以可以用定长 SPSC 队列来支持所有的任务调度，如图 4.1所示。至此，可以发现所有线程不再具有竞争条件，也就不需要互斥锁等阻塞同步工具，成功实现了调度器的无锁化。

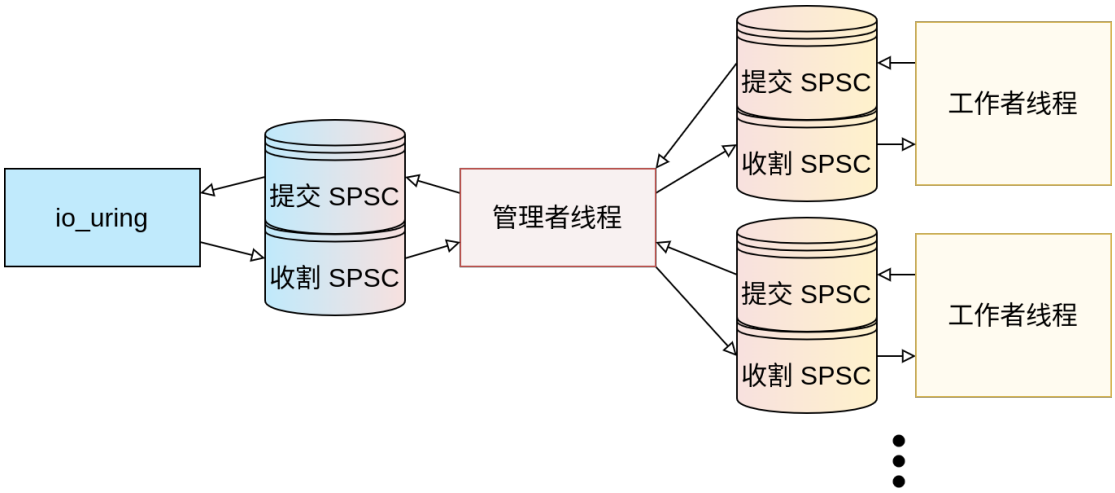

图 4.1 co_context 的跨线程通讯

4.1.3 协程无锁化

尽管调度器层面是彻底无锁的，但用户代码依然可能使用互斥锁，来避免应用层的竞争条件。所以，co_context 必须为用户提供协程间同步工具。co_context 遵循 C++ 标准，为用户提供了三种协程间同步工具：互斥锁、信号量、条件变量。三者的实现只有细微的差别，从性能上看，互斥锁的性能最优，于是本小节介绍互斥锁的实现关键。

当互斥锁最多只被一个协程申请时，任何协程都不需要阻塞，此时，互斥锁的 lock 函数和 unlock 函数只涉及一个原子计数器的增加和减少，此时的实现比较简单。当短时间内超过一个协程申请互斥锁时，只有一个协程取得了互斥锁，剩余的协程需要暂停，并且加入这个互斥锁的“等待队列”。一个难题是，等待队列的长度是无法事先知道的，所以必然涉及动态内存分配。但是动态内存分配在极高并发场景下容易拖慢整体性能，co_context 必须尽量减少动态内存分配的次数。co_context 的做法是，利用每个协程都有的一个协程帧（在3.2.1中有介

绍），在协程帧上做侵入式链表，从而构成等待队列，如图 4.2所示。于是所有的互斥锁操作都不再需要额外的动态内存分配。

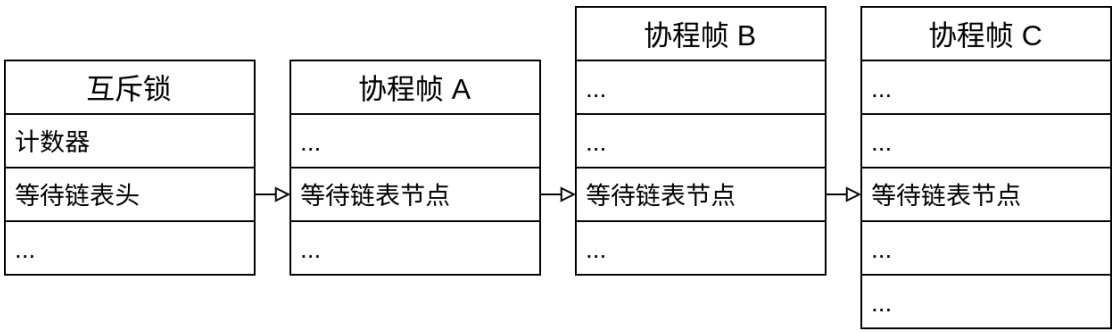

图 4.2 co_context 互斥锁的侵入式链表

然而，上述链表并不满足先进先出规则，严格地说，这是一种先进后出的栈，而非队列。在互斥锁中使用栈是不合适的——位于栈底的协程有可能饿死，因此 co_context 还有一步反转链表的操作。如图 4.3所示，当互斥锁被调用 unlock 函数时，互斥锁会检查等待队列（注意，不是等待链表）是否为空，若为空，则将等待链表反转，形成等待队列。由于每个协程帧对多参与一次链表反转，所以 unlock 函数的均摊时间复杂度是 $O(1)$。此时，按照等待队列的顺序就是协程上锁的顺序，满足了先进先出的性质。

读者可能会产生疑问：为什么不直接使用链表构成队列，而是先用链表构成栈，而后将栈反转为队列呢？答案是为了并发性能。用链表构成的队列至少需要头指针和尾指针，在 64 位机器上，两个指针共占用 16 个字节。在使用无锁原子操作时，硬件通常只提供 8 字节的无锁原子性保证（而且还要求内存对齐）。若直接实现链表队列，是没有办法仅采用无锁原子操作的。为了避免上锁和多余的操作，co_context 选择了反转链表算法。

当互斥锁的 unlock 函数被调用时，位于等待队列头部的协程就会进入就绪状态，进入“提交缓冲区”，等待接受管理者线程的委派。至此，一个完整的 lock-unlock 周期全部被实现。信号量、条件变量的关键实现方法与互斥锁大致相同，限于篇幅，本报告不再深入介绍。

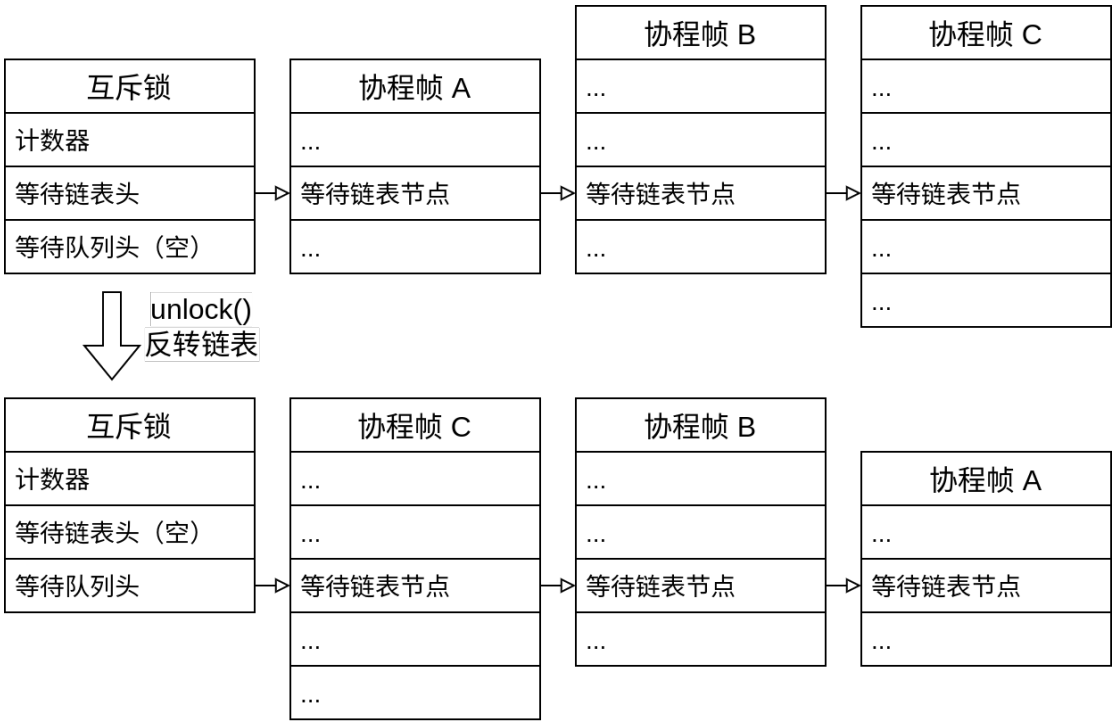

图 4.3 互斥锁 unlock 时反转链表

4.1.4 无锁化小结

co_context 在内部利用定长 SPSC 队列实现了无锁化，又向用户提供了完全无锁化的协程间同步工具（互斥锁、信号量、条件变量），从而实现了彻底的无锁化。无锁技术的应用从避免阻塞的角度为 co_context 提供了非常高的性能贡献。

4.2 缓存行优化

现代 CPU 通常具有 L1，L2 和 L3 级缓存。每一级缓存都可以看作是以 w 个字节为单位的高速储存空间，在主流 CPU 中，w 通常为 64。简单起见，本节只讨论 L1 缓存（以下称为“缓存”），每个地址对齐的独立的 64 字节缓存称为“缓存行”。

4.2.1 伪共享和乒乓缓存

当两个地址不同的变量位于同一个缓存行中，就称这两个变量是伪共享的。当两个CPU物理核心在高频率同时修改同一缓存行时，为保证缓存一致性，CPU会主动同步两个物理核心的缓存行，从而导致高频率的数据的跨核传输，这称为乒乓缓存问题。在每次数据传输完成前，CPU必须等待，从而引发性能的损失。容易发现，伪共享导致的乒乓缓存问题通常是可以避免的。在co_context的实现中，尤其注意了伪共享和乒乓缓存问题。

4.2.2 缓存行隔离

在调度器io_context中，存在大量多线程之间共享的数据（主要是多个定长SPSC队列和对应的头尾指针，还有一些状态标记变量）。设局部变量 v 所在的缓存行为 n，v 的所有可能访问线程的集合为 T，co_context保证了 $T_i \neq T_j \Rightarrow n_i \neq n_j$，即被不同线程共享的变量不可能位于同一缓存行。更进一步地，即使 $T_i = T_j$，若 v_i 只读，而 v_j 可读写，也有保证 $n_i \neq n_j$，即将可写数据与只读数据隔离开来。制造缓存行隔离的办法并不难理解，即用字符数组制造“隔离带”，使得需要隔离的数据地址对齐到64字节。在C++中，可以使用alignas(64)声明来实现缓存行隔离。

4.2.3 线程间通信量压缩

在常见的异步框架中，线程间的通信通常是生产者先将需要传递的信息写入一个结构体中，然后把该结构体的指针传递给消费者，最后消费者解引用这个指针，获得信息。但是这种做法会造成至少两次的缓存行同步：第一次是指针的传递，第二次是指针解引用。co_context拒绝了这种简单但是低效的做法。

在co_context中，线程间主要的通信渠道是定长SPSC队列，从时间性能的角度看，这些队列的容量是非常珍贵的。对于用户的每次I/O请求，co_context保证了工作者线程最多向管理者线程提交16个字节的请求信息。反之亦然，即管理者线程每次写入收割缓冲区最多写入16字节。由于一个缓存行的容量是64

字节，所以，伴随每次缓存行刷新，最多有 4 个请求在 CPU 不同物理核之间传递。若每次请求需要传递多于 16 字节，则每次缓存行刷新只能传递最多 3 个请求，传输效率至少下降 25%。“每次信息传递只使用 16 字节”的实现方法并不是显然的。在 64 位机器上，co_context 的主要压缩办法是：

1. 使用最小化的数据类型（但不使用位域）。

2. 将任务分成 6 种类型，每种类型之间使用联合体（Union）来表示参数。

3. 仅使用 61 比特来表示协程句柄，剩余 3 比特用来区分 6 种任务类型。

最终，在 16 字节之内，co_context 既避免了多余的缓存行同步，又保证了每次缓存行同步的信息传输效率。更多的压缩是没有意义的：16 个字节中，因为有 61 比特表示协程句柄，有 3 比特表示任务类型，最后 8 字节表示 io_uring 的返回信息，任何更多的压缩都是有损的。

4.3 小结

co_context 在保证了内部和用户代码彻底无锁的基础上，针对 CPU 缓存特性做了充足的优化。在 co_context 的实际运行中，在初始化之后，几乎完全没有系统调用开销，所以几乎不会主动陷入内核态；且具有较好的内存空间局部性、时间局部性和较少的跨核通信量和缓存行同步次数。

5 co_context 的性能测试

co_context 是同时看重性能和易用性的异步框架，这两者并不矛盾。其中，易用性已在前面的章节中分析，并在接下来的源代码中有所体现。所以，本章节的重点是展示 co_context 的性能表现。

5.1 测试环境

为了避开网卡和路由器的性能瓶颈，本章的所有测试均在单台机器上运行。测试机器是一台运行着 Arch Linux 操作系统，启用了图形界面的个人台式电脑。详细的软硬件配置如表 5.1所示。

表 5.1 测试机器的软硬件配置

CPU	AMD 5800x 8-Core
超线程	启用
核心隔离	禁用
内存	2 x 16 GB DDR4 3200 Mhz
内核	Linux 5.17.7

值得一提的是，超线程的启用对性能测试有负面影响，因为同一物理核心上的两个线程会互相挤占 CPU 缓存；若两个 co_context 线程被调度到同一物理核心上，则其性能也不如被调度到不同物理核心上。实际上，在例如量化高频交易的低延迟场景中，是必须关闭 CPU 的超线程功能，同时启用核心隔离功能的。本报告选取的测试环境是为了更符合普通用户的使用场景。

5.2 Netcat 吞吐测试

本小节要测量的是 co_context 的基础数据吞吐量。通过实现一个简单的 Netcat，只使用一个 TCP 连接，测试字节流的传输速率和稳定性。

5.2.1 Netcat 简介

Netcat 是用于建立任意 TCP 和 UDP 连接的测试工具。以 TCP 为例，Netcat 可以作为服务端监听某个端口，或者作为客户端连接某个地址。一旦连接建立，Netcat 不再区分服务端和客户端。Netcat 会将接收到的全部消息打印到 stdout。

5.2.2 Netcat-co_context 实现

本报告使用 co_context 的 net 库构建了一个基本的 Netcat。在建立 TCP 连接后，不断的接收 TCP 连接上的消息，然后将其打印到 stdout，周而复始，直到 TCP 报告连接中断为止。涉及性能表现的核心代码如代码 5.1所示。

代码 5.1: 基于 co_context 的 Netcat 的核心代码

```
using namespace co_context;

task<> run(socket peer) {
    char buf[8192];
    int nr = co_await peer.recv(buf);

    // 不断接收字节流，并打印到stdout
    while (nr > 0) {
        nr = co_await (
            lazy::write(STDOUT_FILENO, {buf, nr}, 0)
            && peer.recv(buf)
        );
    }
}
```

5.2.3 负载生成器 chargen

实现了 Netcat 之后，测试还需要一个高性能负载生成器，不断地向 TCP 连接输出字节流，它应该具有高于任何 Netcat 实现的性能。所幸的是，这样的负载生成器很容易实现，因为负载生成不涉及用户内存的写入，性能天然非常高。

为严谨起见，本报告使用了第三方的负载生成器 chargen[12]，采用的是 Thread-per-connection 实现。由于不需要任何调度，它的理论性能会高于任何使用内核态 I/O 的异步框架的实现。

5.2.4 测试命令、结果与分析

本测试使用 chargen -l 1234 为服务端，使用 netcat localhost 1234 > /dev/null 为客户端，对比 Linux 常用工具 nc localhost 1234 > /dev/null 的吞吐量。每秒采样一秒钟内的字节流体积，持续运行 300 秒，计算每秒吞吐量的平均值及其标准差。结果如图 5.1和所示。

表 5.2 Netcat 性能测试数据

Netcat 实现	每秒吞吐量 (MiB/s)	标准差 (MiB/s)
co_context	9571.1	83.6
OpenBSD	6915.2	97.5
Ubuntu	7979.8	1136.6

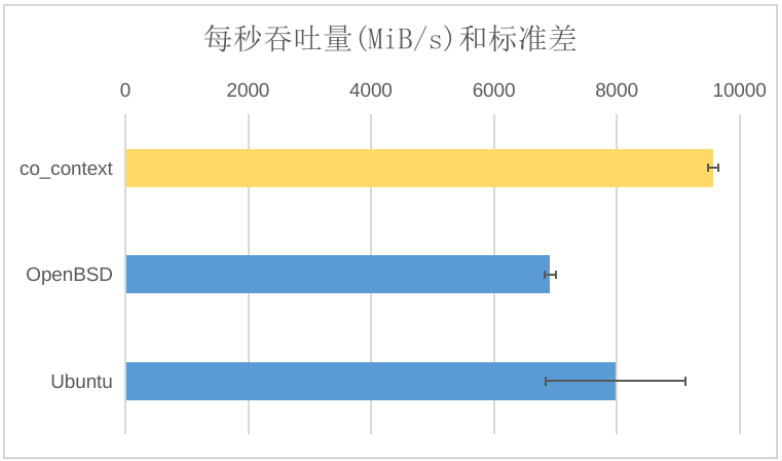

图 5.1 Netcat 性能对比

测试数据显示 co_context 实现的 Netcat 在平均每秒吞吐量上超过了其他实现 19% 以上，且在持续运行 300 秒时间内保持更稳定的吞吐速率。究其原因，在

单 TCP 连接场景下，假设内核态 I/O 的速率恒定，那么影响吞吐量的主要因素是每次 recv 调用和唤醒的延时。得益于 co_context 设计的定长 SPSC 队列、链式 I/O 和 Proactor 机制，节省了至少一半的系统调用和上下文切换，所以在减少通信延时上贡献了优势。另一方面，由于 OpenBSD 和 Ubuntu 实现的 Netcat 都使用 Epoll 为底层 I/O，与 co_context 的 io_uring 底层不同，可能产生细微的差别。但从互联网社区的反馈来看，在当前场景下两者带来的差别可以忽略。对于 Ubuntu 实现的明显抖动，一般是由于不同 Linux 发行版之间的 CPU 调度策略和应用程序内部的参数设置不同而造成。

有一件有趣的事值得一提。使用 Linux perf bench 工具可以测得本机的 memcpy 函数的性能是 23.2 GB/s，而 Netcat 在理论上是需要至少两次内存拷贝。所以，若以 memcpy 函数为理论最优值，则 co_context 基于 TCP 连接的数据吞吐能力已经达到了理想情况的 82.5%。这样的性能对于大多数应用而言，利用 TCP 做进程间通信工具已经足够。

5.3 Redis-Ping 并发测试

5.3.1 Redis-Benchmark 简介

Redis 是知名的内存数据库，以高性能、高并发著称，在大型互联网项目中有大量应用。Redis-Benchmark 是 Redis 官方的基准测试工具，用于测试 Redis 服务端的各类服务的多种性能指标（以每秒请求数和响应延迟为主要指标）。本报告构建一个简单的 Redis 服务端，并利用 Redis-Benchmark 测试其网络性能。

5.3.2 测试目标和方法

本节欲测试 co_context 面临不同并发量时的网络性能表现。在本机开启两个 Redis-Benchmark 线程，作为客户端负载，模拟 c 个客户端，向 Redis 服务端发送总共 10^7 个 PING-INLINE 请求。Redis 服务端通过 TCP 连接收到请求后，立即向客户端发送“+OK\r\n”（共 5 个字节），客户端得到回复后，一个 PING-INLINE 请求即完成。本次测试尝试不同的 c 的取值，以模拟不同的并发压力，对比 co_context

的实现和 Redis 的每秒请求数和多种延迟的区别。由于使用 co_context 的代码非常简洁，足够全部展示，所有代码如代码 5.2所示。

代码 5.2: 基于 co_context 的 Redis-Ping 服务器

```
#include "co_context/io_context.hpp"
#include "co_context/lazy_io.hpp"
#include "co_context/net/acceptor.hpp"

using namespace co_context;

constexpr uint16_t port = 6379;

task<> reply(co_context::socket sock) {
    char recv_buf[100];
    int n = co_await sock.recv(recv_buf);
    while (n > 0) {
        n = co_await (
            sock.send({"+OK\r\n", 5})
            && sock.recv(recv_buf)
        );
    }
}

task<> server() {
    acceptor ac{inet_address{port}};
    int sockfd;
    while ((sockfd = co_await ac.accept()) >= 0) {
        co_spawn(reply(co_context::socket{sockfd}));
    }
}

int main() {
    io_context ctx{32768};
    ctx.co_spawn(server());
    ctx.run();
    return 0;
}
```

5.3.3 测试结果与分析

如图 2.2所示，在多种并发压力下，co_context 的性能相对 Redis 而言均产生了一定的压制力。其中，在客户端数量超过 100 后，co_context 的调度能力尚未达到瓶颈，而 Redis 已经初显疲惫。在客户端数量为 1000 时，co_context 处于舒适的工作强度，而 Redis 已经因忙于调度而损失了 QPS。在随后的客户端不断倍增时，两者均受极端的并发量拖累，但 co_context 在 QPS 和平均延迟上保持了稳定的优势。

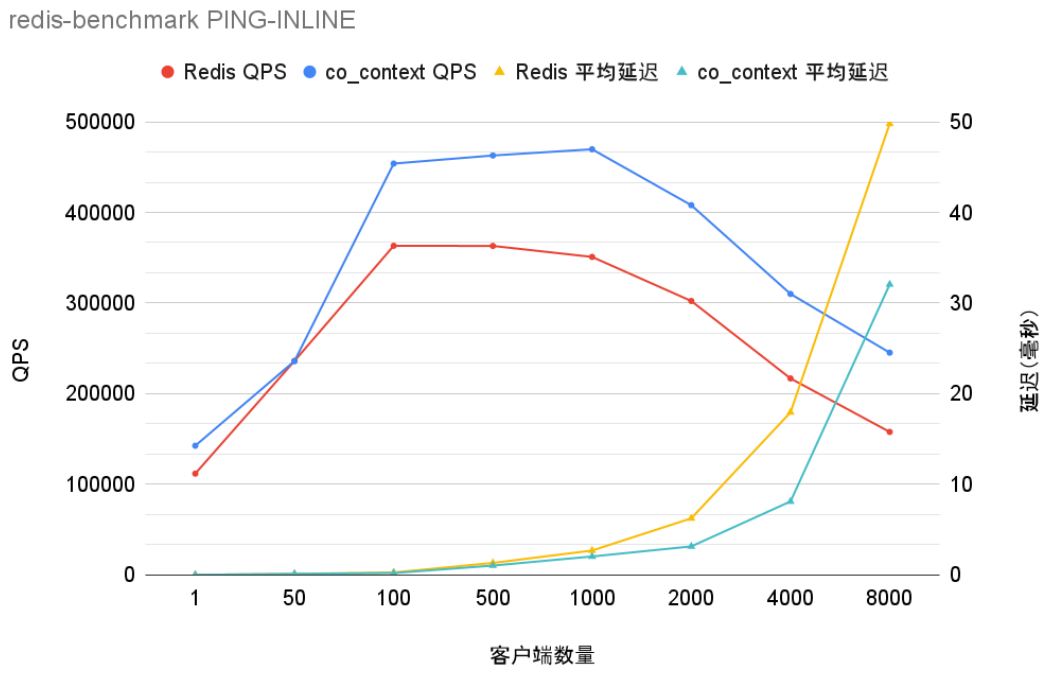

图 5.2 Redis-benchmark PING-INLINE 性能对比

随着并发量不断增加，对于单个请求而言，一旦陷入等待，则平均需要等待 $O(c)$ 个其他请求的完成才能被唤醒，其中 c 是客户端数量。所以，平均延迟随客户端数量的增加而线性增长是合理的。而 co_context 的平均延迟稳定在 Redis 的 65% 左右水平，则再次验证 co_context 的高频率无锁调度器和链式 I/O 节省上下文切换是十分有效的。

5.4 小结

本章经过针对网络 I/O 的测试，检验了 co_context 异步框架的数据吞吐量、稳定性、请求并发量和请求延迟均优于选定的对比对象，体现了框架本身具有比较低的调度损耗，验证了本框架在性能设计和实现上的有效性。同时，简洁的源代码再一次展现了 co_context 的易用性。

6 总结与展望

6.1 总结

本报告讲述了异步框架的需求和现状，阐述了一款新颖的 C++ 异步协程框架——co_context，重点分析了 co_context 的易用性和性能追求，以及它们是如何设计和实现的。在测试章节，验证了 co_context 的性能设计是有效的，测量出了比目前的热门产品更强的并发量和更低的延迟。在全过程中，本报告多次展示使用 co_context 写出的简洁而利于理解的业务代码，也展示了 API 自由组合的便利性和链式 I/O 的强性能。在过去，“简单”和“性能”可能是软件工程中两种互斥的属性，而 co_context 是一个特别的例子，通过充分利用 C++20 协程的简洁表达和足量挖掘体系结构的优化潜力，展示了“简单”和“性能”的水火相容之奇景。

co_context 是孤独的作品。在 Github 上，兼具“C++20 协程”和“io_uring”关键字的项目屈指可数，更不用说 co_context 额外包含一整套 io_uring 的接口库 liburingcxx。从这个角度上讲，据本文作者所知，co_context 的同类作品并不存在。这样的现象一定不是一个活跃、热闹、开放的开发者群体所应该呈现的。co_context 希望以其易用性和高性能为凭证，为 io_uring 和 C++ 注入开源力量。

6.2 展望

co_context 是针对 Linux 系统调用和 C++ 标准提供 API 的，意味着已有的大量基于 Linux 系统调用的库和上层应用，存在着移植到 co_context 的可能性。因为 co_context 的设计已经定型，所以为 co_context 开发应用层库的时机已经成熟。在短期未来，可以为 co_context 移植一些知名的、具有影响力的上层库和应用，例如 OpenSSL 和 libcurl 和 Kafka，以快速丰富 co_context 的生态，扩大影响力。更激进地，可以研究基于词法分析和语法分析的办法，将库的移植全自动化，从而实现“一夜之间建成罗马”。

最后，C++20 无栈协程是一块瑰宝，它不能被称作“能用 C 语言模拟出来

的功能”，因为有了编译器的支持，有些工作的规模从不可能完成变得可能完成。例如，研究“软件超线程”（在 CPU cache-miss 时暂停协程并切换到另一协程，等待缓存加载完毕后，再恢复原来的协程）。由于现代 CPU 的硬件超线程仍未充分发挥 CPU 的潜力（当两个超线程都 cache-miss 时，CPU 被迫等待），使用协程式“软件超线程”有望大幅提高程序的并发性能，同时不引入更多的编程复杂度。在有无栈协程之前，通用的软件超线程几乎是不可能做到的。

参考文献

[1] AXBOE J. Efficient IO with io_uring[EB/OL]. 2019. https://kernel.dk/io_uring.pdf.
[2] NEELY T. Sled and rio: modern database engineering with io_uring[EB/OL]. 2020. https: //archive.fosdem.org/2020/schedule/event/rust_techniques_sled/.
[3] KOHLHOFF C. Boost.Asio: Revision History[EB/OL]. 2022. https://www.boost.org/doc/libs/1_78_0/doc/html/boost_asio/history.html.
[4] Tencent. Libco[EB/OL]. 2013. https://github.com/Tencent/libco.
[5] Sogou. Workflow[EB/OL]. 2020. https://github.com/sogou/workflow.
[6] Bytedance. Monoio[EB/OL]. 2021. https://github.com/bytedance/monoio.
[7] Bytedance. Monoio-benchmark[EB/OL]. 2021. https://github.com/bytedance/monoio/blob /master/docs/zh/benchmark.md.
[8] Facebook. Libunifex[EB/OL]. 2018. https://github.com/facebookexperimental/libunifex.
[9] MICHAL DOMINIAK L B. P2300R3: std::execution[EB/OL]. 2021. http://open-std.org/JTC1/SC22/WG21/docs/papers/2021/p2300r3.html#intro-prior-art-coroutines.
[10] Libuv. Libuv[EB/OL]. 2012. https://github.com/libuv/libuv.
[11] 陈硕. Muduo[EB/OL]. 2014. https://github.com/chenshuo/muduo.
[12] 陈硕. Chargen.cc[EB/OL]. 2021. https://github.com/chenshuo/recipes/blob/master/tpc/bin /chargen.cc.
[13] ********** 2022. https://github.com/Codesire-Deng/co_context.

附录

A co_context 的源代码清单

co_context 的完整项目[13]已在 Github 上开源。源代码清单如下：

表 A.1 co_context 的源代码清单-1

./include/uring/uring.hpp
./include/uring/detail/CQ.hpp
./include/uring/detail/SQ.hpp
./include/uring/SQEntry.hpp
./include/uring/syscall.hpp
./include/uring/CQEntry.hpp
./include/co_context/detail/sem_task_meta.hpp
./include/co_context/detail/task_info.hpp
./include/co_context/detail/submit_info.hpp
./include/co_context/detail/eager_io_state.hpp
./include/co_context/detail/worker_meta.hpp
./include/co_context/detail/swap_zone.hpp
./include/co_context/detail/deprecated_sqe_task_meta.hpp
./include/co_context/detail/thread_meta.hpp
./include/co_context/detail/cv_task_meta.hpp
./include/co_context/detail/reap_info.hpp
./include/co_context/main_task.hpp
./include/co_context/lazy_io.hpp
./include/co_context/io_context.hpp

表 A.2 co_context 的源代码清单-2

./include/co_context/log/log.hpp
./include/co_context/utility/timing.hpp
./include/co_context/utility/as_atomic.hpp
./include/co_context/utility/polymorphism.hpp
./include/co_context/utility/compile_time.hpp
./include/co_context/utility/set_cpu_affinity.hpp
./include/co_context/utility/defer.hpp
./include/co_context/utility/buffer.hpp
./include/co_context/utility/bit.hpp
./include/co_context/net/socket.hpp
./include/co_context/net/inet_address.hpp
./include/co_context/net/acceptor.hpp
./include/co_context/lockfree/spsc_cursor.hpp
./include/co_context/lockfree/spsc_cursor.1.hpp
./include/co_context/lockfree/queue.hpp
./include/co_context/config.hpp
./include/co_context/eager_io.hpp
./include/co_context/task.hpp
./include/co_context/co/semaphore.hpp
./include/co_context/co/mutex.hpp
./include/co_context/co/condition_variable.hpp

表 A.3 co_context 的源代码清单-3

./include/uring/barrier.h
./include/uring/compat.h
./include/uring/io_uring.h
./lib/co_context/net/socket.cpp
./lib/co_context/net/inet_address.cpp
./lib/co_context/io_context.cpp
./lib/co_context/co/semaphore.cpp
./lib/co_context/co/condition_variable.cpp
./lib/co_context/co/mutex.cpp

B co_context 的更多用例

代码 B.1: 用于性能测试的 netcat 实现

```
#include "co_context/io_context.hpp"
#include <thread>
#include <string.h>
#include <unistd.h>
#include <string_view>
#include "co_context/net/inet_address.hpp"
#include "co_context/net/acceptor.hpp"
#include "co_context/net/socket.hpp"
#include "co_context/task.hpp"

co_context::task<> run(co_context::socket peer) {
    printf("run: Running\n");
    using namespace co_context;
    char buf[8192];
    int nr = co_await peer.recv(buf);

    // 不断接收字节流
    while (nr > 0) {
```

```
            nr = co_await (
                lazy::write(STDOUT_FILENO, {buf, (size_t)
nr}, 0) && peer.recv(buf)
            );
        }
    }

    co_context::task<> server(uint16_t port) {
        using namespace co_context;
        acceptor ac{inet_address{port}};
        // 不断接受 client, 每个连接生成一个 worker 协程
        for (int sockfd; (sockfd = co_await ac.accept())
>= 0;) {
            co_spawn(run(co_context::socket{sockfd}));
        }
    }

    co_context::task<> client(std::string_view hostname,
uint16_t port) {
        using namespace co_context;
        inet_address addr;
        if (inet_address::resolve(hostname, port, addr))
{
            co_context::socket sock{co_context::socket::
create_tcp(addr.family())};
            // 连接一个 server
            int res = co_await sock.connect(addr);
            if (res < 0) {
                printf("res=%d: %s\n", res, strerror(-res
));
                ::exit(0);
            }
            // 生成一个 worker 协程
            co_spawn(run(std::move(sock)));
        } else {
            printf("Unable to resolve %s\n", hostname.
data());
```

```
            ::exit(0);
        }
    }

    int main(int argc, const char *argv[]) {
        if (argc < 3) {
            printf("Usage:\n  %s hostname port\n  %s -l
port\n", argv[0], argv[0]);
            return 0;
        }

        using namespace co_context;
        io_context context{16};

        int port = atoi(argv[2]);
        if (strcmp(argv[1], "-l") == 0) {
            context.co_spawn(server(port));
        } else {
            context.co_spawn(client(argv[1], port));
        }

        context.run();

        return 0;
    }
```

代码 B.2: 用于测试的 mutex 例程

```
    #include "co_context/io_context.hpp"
    #include "co_context/co/mutex.hpp"
    #include <iostream>

    using namespace co_context;
    co_context::mutex mtx;
    int cnt = 0;

    task<> add() {
```

```
    auto lock = co_await mtx.lock_guard();
    for (int i = 0; i < 1000000; ++i) ++cnt;
    std::cout << cnt << std::endl;
}

int main() {
    io_context ctx{32768};
    for (int i = 0; i < 1000; ++i) ctx.co_spawn(add()
);
    ctx.run();
}
```

代码 B.3: 用于测试协程切换延迟的例程

```
#include "co_context/io_context.hpp"
#include "co_context/utility/timing.hpp"
#include "co_context/lazy_io.hpp"

using namespace co_context;

constexpr uint32_t total_switch = 1e8;
uint32_t count = 0;

task<> run() {
    auto start = std::chrono::steady_clock::now();

    while (++count < total_switch) [[likely]]
co_await lazy::yield();

    if (count == total_switch) [[unlikely]] {
        auto end = std::chrono::steady_clock::now();
        std::chrono::duration<double, std::micro>
duration = end - start;
        printf("Host time = %7.3f us.\n", duration.
count());
        printf(
            "Avg. switch time = %3.3f ns.\n",
```

```
                duration.count() / total_switch * 1000
            );
            co_context_stop();
        }
    }

    int main() {
        io_context ctx{128};
        for (int i = 0; i < config::swap_capacity / 2; ++
    i)
            ctx.co_spawn(run());
        ctx.run();
        return 0;
    }
```