packt

1ST EDITION

Asynchronous Programming with C++

Build blazing-fast software with multithreading and asynchronous programming for ultimate efficiency

JAVIER REGUERA-SALGADO
JUAN ANTONIO RUFES

Asynchronous Programming with C++

通过多线程和异步编程打造高速运行的软件

作者： Javier Reguera-Salgado / Juan Antonio Rufes

译者：陈晓伟

致谢

感谢我的妻子 Raquel，我们共同的人生旅程中，她一直是我深爱的伴侣，也是爱、力量和欢乐的源泉。感谢我的女儿 Julia，她在让我心中充满了希望和惊喜。

感谢我的父母，玛丽娜（Marina）和埃斯塔尼斯劳（Estanislao），感谢他们的牺牲、爱、支持和启发。

Javier Reguera-Salgado

献给我的妹妹，伊娃玛丽亚（Eva María），并深情怀念我的父母。

Juan Antonio Rufes

关于作者

Javier Reguera-Salgado 具有西班牙维哥大学高性能计算博士学位，是一位资深软件工程师，并拥有 19 年多的工作经验，专门从事高性能计算、实时数据处理和通信协议。他精通 C++、Python 以及其他编程语言和技术，工作涉及低延迟分布式系统、移动应用、Web 解决方案和企业产品。他为西班牙和英国的研究中心、初创公司、蓝筹公司和量化投资公司工作过。

首先，感谢我的妻子 Raquel 和女儿 Julia，感谢她们对我的爱和鼓励，让我每天都深受鼓舞。感谢我的父母 Marina 和 Estanislao，教会了我勤奋和坚持不懈。我还要感谢亲戚朋友对我的爱和支持。此外，还要感谢 Juan 与我共同撰写了这本书，以及 Packt Publishing 团队和审稿人。

Juan Antonio Rufes 具有拥有西班牙瓦伦西亚理工大学电气工程硕士学位，是一位拥有 30 年经验的软件工程师，专门从事底层和系统编程，主要使用 C、C++、0x86 汇编和 Python。其专长包括 Windows 和 Linux 优化、用于预防病毒和加密的 Windows 内核驱动程序、TCP/IP 协议分析，以及低延迟金融系统（例如：智能订单路由和基于 FPGA 的交易系统）。他在软件公司、投资银行和对冲基金中工作过。

衷心感谢父母对我的教导和支持。同时，感谢 Javier 与我共同创作了本书。非常感谢 Packt Publishing 团队和技术审阅人员的建议和对细节的关注。

关于评审

Eduard Drusa 从 8 岁开始编程，之后获得了计算机科学硕士学位。在职业生涯中，对知识的热情使他涉足了从汽车到商业桌面软件开发等各种行业。他利用这个机会接受了各种挑战，并学习了解决常见问题的不同方法。他将这些知识运用到他原来的行业之外，为旧问题带来了新的解决方案。最近，他的兴趣转向了嵌入式系统安全，已经开始着手一个创建创新实时操作系统的项目。他也组织了一系列的编程课程，为了更大程度地利用已获取到的知识。

目录

前言

异步编程是构建高效、响应迅速且高性能软件的必备，尤其是在多核处理器和实时数据处理领域。本书探讨了掌握 C++ 中异步编程的原理和技巧，提供处理从线程管理到性能优化等事务所需的知识。

开发异步软件有几个关键点：

- 线程管理和同步
- 异步编程概念、模型和库
- 调试、测试和优化多线程和异步软件

虽然许多资料都侧重于并行编程或通用软件开发的基础知识，但本书旨在全面探索这些要点。涵盖了管理并发、调试复杂系统和优化软件性能的基本技术，同时将这些概念应用于实际。

随着多核处理器和并行计算架构成为现代应用程序不可或缺的一部分，对异步编程的需求也正在迅速增长。掌握本书中的技术，不仅可以帮助应对当今复杂的软件开发挑战，还可以为性能关键型软件的未来发展做好准备。

无论是正在使用低延迟金融系统、开发高吞吐量应用程序，还是想提高编程技能，本书都会为提供相应的工具和知识。

适读人群

本书面向会使用最新 C++ 版本加深对异步编程的理解，并优化软件性能的软件工程师、开发人员和技术主管。主要目标受众包括：

- 软件工程师：希望提高 C++ 技能，并获得多线程和异步编程、调试和性能优化方面的实用见解。
- 技术领导：旨在实施高效异步系统的领导者，将发现管理复杂软件开发和提高团队生产力的策略和最佳实践。
- 学生和爱好者：渴望了解高性能计算和异步编程的个人将受益于详尽的解释和示例，助其在技术职业生涯中取得进步。

本书将帮助读者应对实际挑战，并在技术面试中脱颖而出，并提供在当今软件领域蓬勃发展的知识。

关于本书

第 1 章，*并行编程范式*，探讨了构建并行系统的不同架构和模型，以及各种并行编程范式及其性能指标。

第 2 章，*进程、线程和服务*，深入研究操作系统中的进程，了解其生命周期、进程间通信以及线程的作用，包括守护进程和多线程。

第 3 章，*如何在 C++ 中创建和管理线程*，了解如何创建和管理线程、传递参数、检索结果，以及处理异常以确保在多线程环境中高效执行。

第 4 章，*使用锁进行线程同步*，解释了 C++ 标准库同步原语（包括互斥锁和条件变量）的使用，并解决了竞争条件、死锁和活锁等问题。

第 5 章，*原子操作*，介绍了 C++ 原子类型、内存模型，以及如何实现基本的 SPSC 无锁队列，为未来的性能增强做好准备。

第 6 章，*Promise 和 Future*，介绍了异步编程概念，包括 promise、future 和打包任务，并展示了如何使用这些工具解决实际问题。

第 7 章，*异步函数*，介绍了 std::async 用于执行异步任务、定义启动策略、处理异常和优化性能的功能。

第 8 章，*使用协程*，介绍了 C++ 协程、其基本要求以及如何实现生成器和解析器，并且要如何处理协程内的异常。

第 9 章，*使用 Boost.Asio 进行异步编程*，介绍了如何使用 Boost.Asio 管理与外部资源相关的异步任务，重点关注 I/O 对象、执行上下文和事件处理。

第 10 章，*使用 Boost.Cobalt 实现协程*，探索使用 Boost.Cobalt 库轻松实现协程，避免低级复杂性并专注于函数式编程需求。

第 11 章，*记录和调试异步软件*，介绍了如何有效地使用日志和调试工具来识别和解决异步应用程序中的问题，包括死锁和竞争条件。

第 12 章，*消杀和测试异步软件*，介绍了如何使用清理器对多线程代码进行清理，并探讨了使用 GoogleTest 库为异步软件定制的测试技术。

第 13 章，*提高异步软件性能*，研究性能测量工具和技术，包括高分辨率计时器、缓存优化，以及避免虚和实共享的策略。

编译环境

需要具有使用 C++ 进行编程的经验，以及如何使用调试器查找错误。由于使用 C++20 功能，并且在某些示例中使用 C++23，因此需要安装 GCC 14 和 Clang 18。所有源代码示例均已在 Ubuntu 和 macOS 中进行了测试，由于它们与平台无关，因此应该可以在任何平台上编译和运行。

书中涉及的软件/硬件	**操作系统**
C++20 和 C++23	Linux（测试过 Ubuntu 24.04）
GCC 14.2	macOS（测试过 macOS Sonoma 14.x）
Clang 18	Windows 11
Boost 1.86	
GDB 15.1	

每章都包含一个*技术要求*部分，重点介绍如何安装编译本章示例所需的工具和库的相关信息。

如果正在使用这本书的数字版本，我们建议自己输入代码或通过 GitHub 库访问代码（链接在下一节中提供）。这样做将帮助您避免与复制和粘贴代码相关的任何潜在错误。

源码下载

本书的代码托管在 GitHub 上，地址为 https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP。此外，还可以在 https://github.com/PacktPublishing/ 浏览图书和视频目录中的其他代码包。欢迎查看！

第一部分 并行编程和流程管理

第一部分中，将深入探讨构成并行编程和进程管理基础的基本概念和范例。将介绍用于构建并行系统的架构，并探索可用于开发高效并行、多线程和异步软件的各种编程范例。此外，还将介绍与进程、线程和服务相关的关键概念，强调其在操作系统中进程生命周期、性能和资源管理方面的重要性。

本部分包含以下章节：

- 第 1 章，并行编程范式
- 第 2 章，进程、线程和服务

第 1 章 并行编程范式

深入研究使用 C++ 进行并行编程之前，我们将重点了解构建并行软件的不同方法，以及软件如何与机器硬件交互的一些基础知识。

本章中，将介绍并行编程，以及在开发高效、响应迅速、可扩展并发和异步软件时，可以使用的不同范式和模型。

对开发并行软件的不同方法进行分类时，有很多种方法可以对概念和方法进行分组。由于本书重点介绍使用 C++ 构建的软件，因此可以将不同的并行编程范例划分为以下几种：并发、异步编程、并行编程、反应式编程、数据流、多线程编程和事件驱动编程。

特定范式可能比其他范式更适合解决给定场景，了解不同的范式将有助于分析问题并缩小搜索最佳解决方案的范围。

本章中，将讨论以下主题：

- 什么是并行编程？为什么重要？
- 有哪些不同的并行编程范式？为什么需要了解它们？
- 将在本书中学到什么？

1.1. 技术要求

整本书中，将使用 C++20 开发不同的解决方案，在某些示例中，还会使用 C++23，因此需要安装 GCC 14 和 Clang 8。

本书中展示的所有代码块都可以在以下 GitHub 库中找到：https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP。

1.2. 分类、技术和模型

当任务或计算同时完成时，就会发生并行计算，其中任务是软件应用程序中的执行或工作单元。由于实现并行性的方法有很多，因此了解不同的方法将有助于编写高效的并行算法。这些方法可以通过范例和模型进行描述。

但首先，先对不同的并行计算系统进行分类。

1.2.1. 系统分类和技术

并行计算系统最早的分类之一是由 Michael J. Flynn 于 1966 年提出的。弗林（Flynn）根据并行计算架构可以处理的数据流和指令数量进行了以下分类：

- 单指令单数据（SISD）系统：定义顺序执行程序
- 单指令多数据（SIMD）系统：对大型数据集进行操作，例如：GPU 计算的信号处理数据
- 多指令单数据（MISD）系统：很少使用
- 多指令多数据（MIMD）系统：基于多核和多处理器计算机的（最常见）并行架构

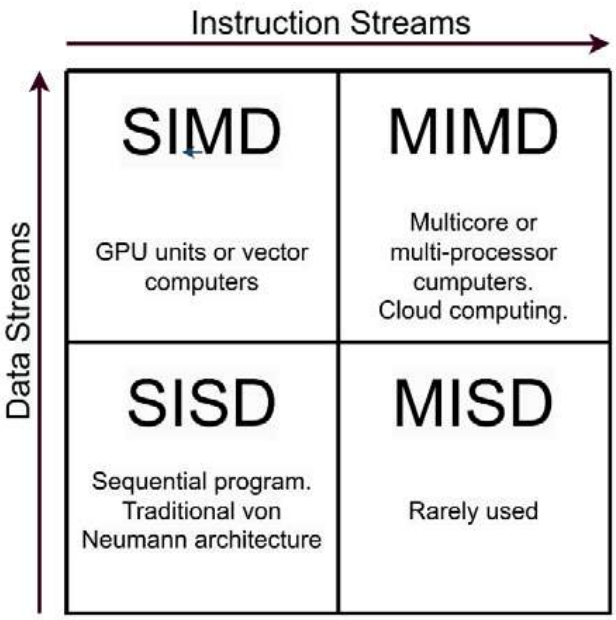

图 1.1：弗林分类法

本书不仅介绍了如何使用 C++ 构建软件，还介绍了其如何与底层硬件交互。软件层面上，我们可以进行更有趣的划分或分类，并定义技术。这些，将在后续章节中进行介绍。

数据并行

许多不同的数据单元由在不同处理单元（例如：CPU 或 GPU）中，运行的同一程序或指令序列并行处理。

数据并行性通过相同操作，同时处理多少个不相交的数据集来实现。利用并行性，可以将大型数据集划分为更小且独立的数据块。

因为更多的处理单元可以处理更多的数据，所以该技术还具有高度的可扩展性。

在这个子集中，可以包含 SIMD 指令集，例如 SSE、AVX、VMX 或 NEON，这些指令集可通过 C++ 中的内部函数访问。此外，还有用于 NVIDIA GPU 的 OpenMP 和 CUDA 等库。在机器学习训练和图像处理中可以找到它的一些使用示例，该技术与弗林定义的 SIMD 分类有关。

这种分类方式也存在一些缺点——数据必须能够轻松划分为独立的块。这种数据划分和后验合并也会带来一些开销，从而降低并行化的优势。

任务并行

每个 CPU 核心使用进程或线程运行不同的任务，当这些任务同时接收数据、处理数据并通过消息传递发回它们生成的结果时，就可以实现任务并行。

任务并行的优势在于能够设计异构、细粒度的任务，从而更好地利用处理资源，在设计具有潜在更高加速的解决方案时更加灵活。

根据数据创建的任务之间可能存在依赖关系，并且每个任务的性质不同，因此调度和协调比数据并行更复杂，所以任务创建会增加一些开销。

这里可以引入弗林的 MISD 和 MIMD 分类法，可以在 Web 服务器请求处理系统或用户界面事件处理程序中找到一些示例。

流并行

可将计算分为处理数据子集的各个阶段，来同时处理连续的数据元素序列（也称为数据流）。

阶段可以同时运行。一些阶段生成其他阶段的输入，根据阶段依赖关系构建管道。处理阶段可以将结果发送到下一个阶段，而无需等待整个流数据。

流并行技术在处理连续数据时非常有效。还具有高度可扩展性，可以通过添加更多处理单元来处理更多的输入数据。由于流数据在到达时进行处理，所以无需等待整个数据流发送，内存使用量也减少了。

然而，这些系统也存在一些缺点。由于逻辑处理、错误处理和恢复，这些系统的实现更加复杂。由于还需要实时处理数据流，因此硬件也可能是瓶颈之一。

这些系统的一些示例包括监控系统、传感器数据处理，以及音频和视频流。

隐式并行

编译器、运行时或硬件会并行执行指令。这使得编写并行程序变得更容易，但限制了开发者对所用策略的控制，甚至使分析性能或调试变得更加困难。

现在，我们对不同的并行系统和技术有了更好的了解，是时候了解在设计并行程序时可以使用的模型了。

1.2.2. 并行编程模型

并行编程模型是一种并行计算机架构，用于表达算法和构建程序。模型越通用，其价值就越大，可以用于更广泛的场景。从这个意义上讲，C++ 通过标准模板库（STL）中的库实现了并行模型，可用于实现顺序应用程序中程序的并行执行。

这些模型描述了程序生命周期内，不同任务如何交互以从输入数据中获取结果。其主要区别在于，任务如何交互以及如何处理传入数据。

阶段并行

阶段并行（也称为议程或自由同步范式）中，多个作业或任务并行执行独立计算。在某个时刻，程序需要使用栅栏执行同步交互操作来同步不同的进程。栅栏是一种同步机制，可确保一组任务在其执行过程中到达特定点执行完后，才能继续进行下一个步骤。接下来的步骤将执行其他异步操作，依此类推。

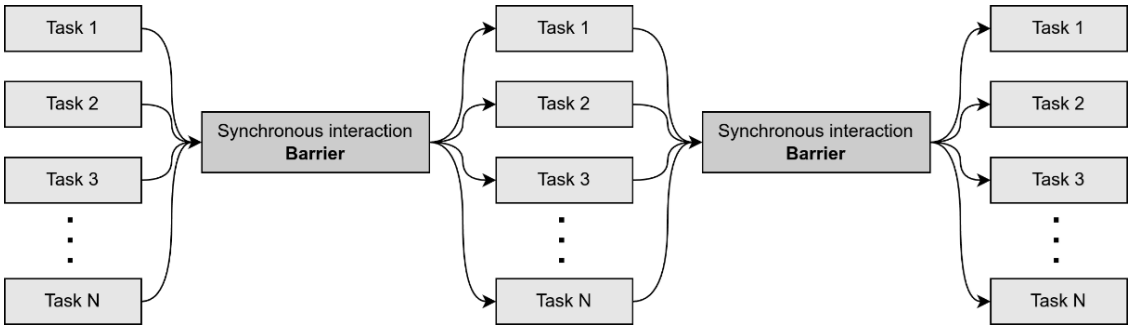

图 1.2: 阶段并行

这种模型的优点是任务间的交互不会与计算重叠，但各个处理单元之间的工作量和吞吐量很难达到均衡。

分而治之

使用此模型的应用程序使用主任务或作业，将工作量分配给其子任务。子任务并行计算结果将其返回给父任务，父任务将结果合并为最终结果。子任务还可以将分配的任务细分为更小的任务，并创建自己的子任务。

该模型具有与相并联模型相同的缺点，很难实现良好的负载平衡。

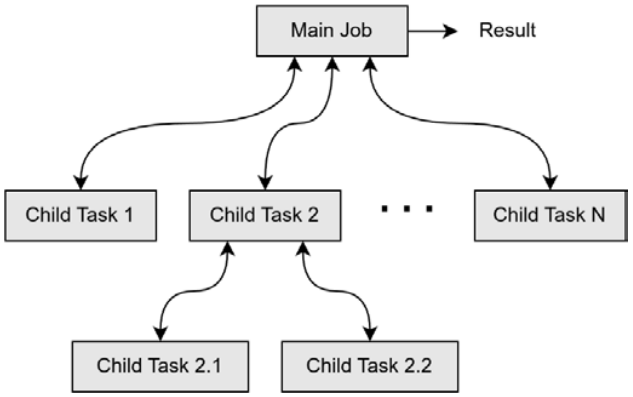

图 1.3：分而治之模型

图 1.3 中，可以看到主作业如何将工作划分给几个子任务，以及子任务 2 如何将其分配的工作细分为两个任务。

管道模型

多个任务相互连接，构建虚拟管道。此管道中，各个阶段可以同时运行，并在输入数据时重叠执行。

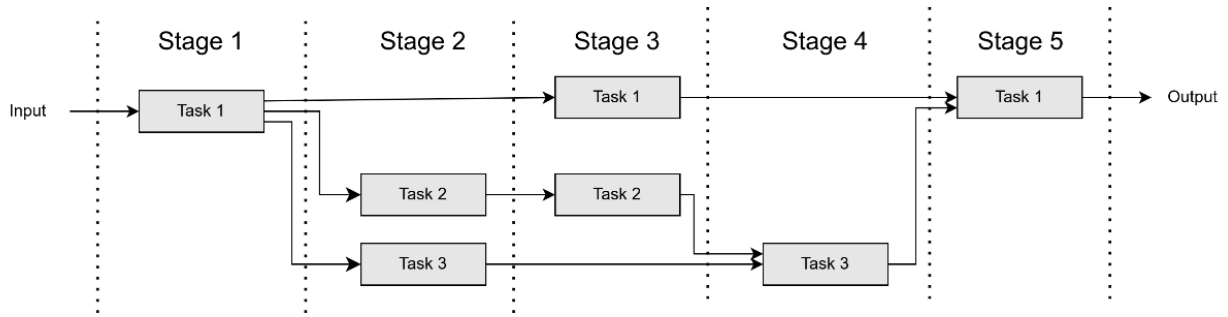

图 1.4：管道模型

上图中三个任务在由五个阶段组成的流水线中交互，每个阶段都有一些任务在运行，并产生输出结果，供下一个阶段的任务使用。

主从模型

使用主从模型（也称为进程农场），主执行者执行算法的顺序部分，并生成和协调在工作负载中执行并行操作的从属任务。当从属任务完成计算时，将结果通知主执行者，然后主执行者可能会将更多数据发送给从属任务进行处理。

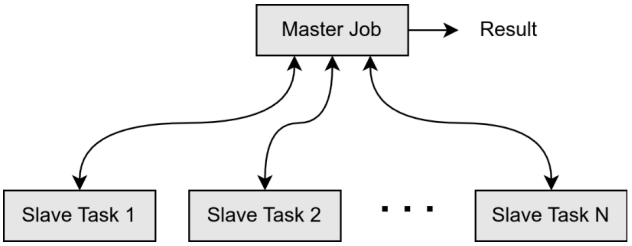

图 1.5：主从模型

缺点是，如果主服务器需要处理太多从服务器或任务太小，主服务器可能会成为瓶颈。在选择由主服务器执行的工作量时，需要权衡每个任务，这也称为任务粒度。当任务较小时，称为细粒度，当任务较大时，称为粗粒度。

工作池

工作池模型中，全局结构保存着要执行的工作项池。主程序会创建作业，从池中获取工作项并执行。

这些作业可以生成更多工作，并将其插入到工作池中。当所有工作都完成后，清空工作池，并行程序便会结束执行。

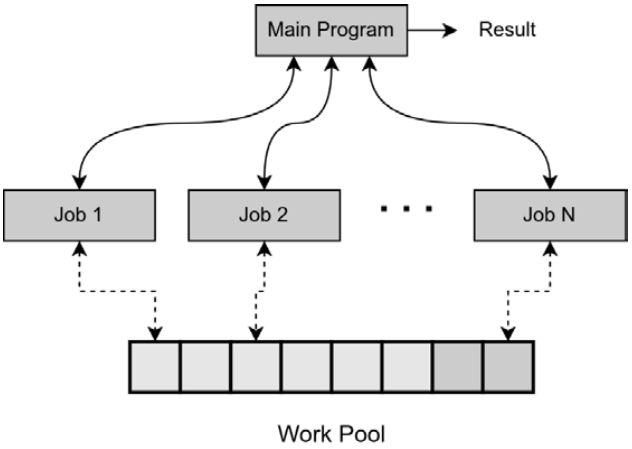

图 1.6：工作池模型

该机制有利于实现空闲处理单元之间的负载平衡。

在 C++ 中，这个池通常使用无序集合、队列或优先级队列来实现。我们将在本书中后续内容中进行实现。

了解了可用于构建并行系统的各种模型，可以继续探索用于开发高效并行任务的并行编程范例了。

1.3. 并行编程范式

现在，是时候转向更抽象的分类，通过探索并行编程语言范式，来了解编写并行程序的风格和原则。

1.3.1. 同步编程

同步编程语言用于构建按顺序执行代码的程序。执行一条指令时，程序将保持阻塞状态，直到该指令完成，没有多任务处理。这使得代码更易于理解和调试。

这种行为使得程序在运行指令时阻塞，无法响应外部事件，并且难以扩展。这是大多数编程语言（例如 C、Python 或 Java）使用的传统范式。

这种模式对于需要实时、有序地响应输入事件的反应式或嵌入式系统尤其有用。处理速度必须与环境所施加的速度相匹配，并有严格的时间限制。

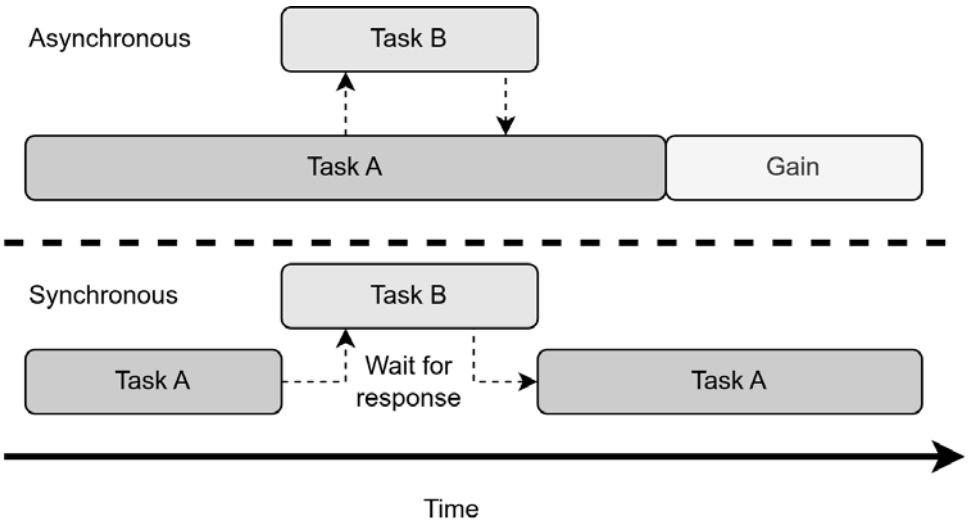

图 1.7：异步与同步执行时间

图 1.7 展示了系统中正在运行的两个任务。同步系统中，任务 A 被任务 B 中断，只有在任务 B 完成其工作后才会恢复执行。异步系统中，任务 A 和 B 可以同时运行，从而在更短的时间内完成工作。

1.3.2. 并发编程

通过并发编程，可以同时运行多个任务。任务可以独立运行，无需等待其他任务完成，还可以共享资源并相互通信。其指令可以异步运行，可以按任意顺序执行，而不会影响结果，增加了并行处理的潜力。不过，这使得程序难以理解和调试。

因为在一定时间间隔内完成的任务数量会随着并发性而增加（古斯塔夫森定律公式），所以并发性提高了程序的吞吐量。

此外，因为可以在等待期间执行其他任务，程序有更好的输入和输出响应能力。

并发软件的问题是实现正确的并发控制。协调对共享资源的访问并确保不同计算执行之间发生正确的交互顺序时，必须格外小心。错误的决策可能导致竞争条件、死锁或资源短缺。这些问题大多可以通过一致性或内存模型来解决，该模型定义了在访问共享内存时按何种顺序执行操作的规则。

设计高效并发算法是通过寻找协调任务执行、数据交换、内存分配和调度的技术，来最小化响应时间并最大化吞吐量。

第一篇介绍并发的学术论文《并发编程控制问题的解决方案》由 Dijkstra 于 1965 年发表，论文中也发现并解决了互斥问题。

并发的抢占会发生在操作系统级，即调度程序无需与任务交互即可切换上下文（从一个任务切换到另一个任务）。也可以以非抢占或协作方式发生，即任务将控制权交给调度程序，调度程序选择另一个任务继续。

调度程序通过保存程序的状态（内存和寄存器内容）来中断正在运行的程序，然后加载已保存的状态并将控制权移交给该程序，这称为上下文切换。根据任务的优先级，调度程序会允许高优先级任务比低优先级任务使用更多的 CPU 时间。

此外，一些特殊的操作软件（例如内存保护）会使用特殊硬件来保证监控软件不受用户模式程序错误的损坏。

该机制不仅用于单核计算机，也用于多核计算机，允许执行比可用核心数量多的任务。抢占式多任务处理还允许提前安排重要任务，以便快速处理重要的外部事件。当操作系统向这些任务发送触发中断的信号时，这些任务就会唤醒并处理重要工作。

旧版 Mac 和 Windows 操作系统使用非抢占式多任务处理。如今，RISC 操作系统仍在使用这种处理方式。Unix 系统于 1969 年开始使用抢占式多任务处理，这是所有类 Unix 系统、Windows NT 3.1 和 Windows 95 以后 Windows 系统的核心功能。

早期的 CPU 每次只能运行一条指令路径。并行通过在指令流之间切换来实现，通过看似重叠的执行，给人一种并行的假象。

2005 年，英特尔推出了多核处理器，允许在硬件层面同时执行多个指令流。这给编写软件带来了一些挑战，需要解决和利用硬件层面的并发性。

C++11 通过 std::thread 库支持并发编程。C++ 早期版本不包含该功能，因此开发者依赖于 Unix 系统中基于 POSIX 线程模型的平台特定库或 Windows 系统中的专有 Microsoft 库。

为了更好地理解了并发是什么，需要区分并发和并行。当许多执行路径可以在重叠的时间段内交错执行时，就会发生**并发**；而这些任务由不同的 CPU 单元同时执行，并利用可用的多核资源时，就会发生**并行**。

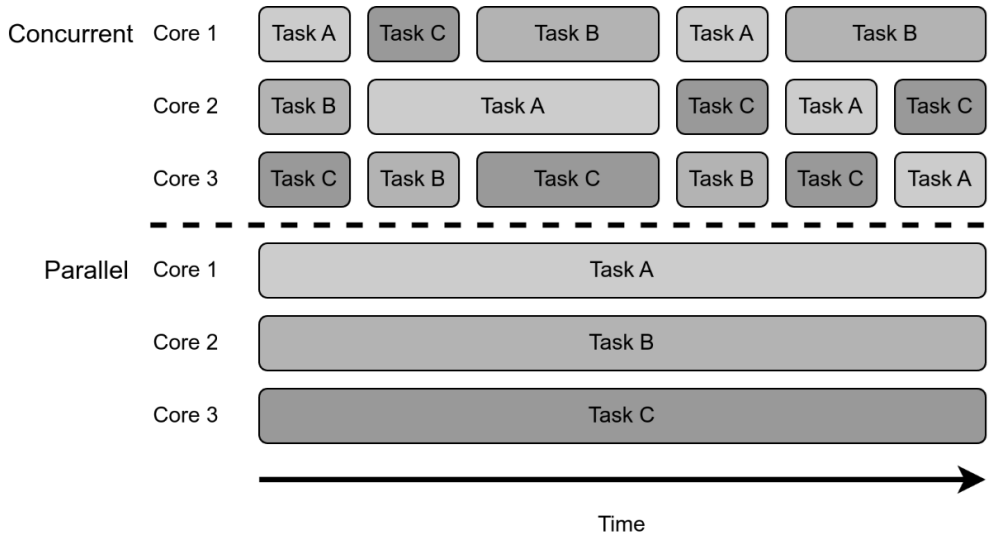

图 1.8：并发与并行

并发编程会比并行编程更通用，后者具有预定的通信模式，而前者可以涉及任务之间的通信和交互模式。

并行可以存在于没有并发性（没有交错的时间段）的情况，也可以存在于没有并行性（通过单核 CPU 上的分时多任务处理）的情况。

1.3.3. 异步编程

异步编程可以安排一些任务在后台运行，同时继续执行当前作业，而无需等待计划的任务完成。当这些任务完成后，它们会将其结果报告给主作业或调度程序。

同步应用程序的一个关键问题是，长时间操作可能会导致程序无法响应接下来的输入或处理。异步程序解决了这个问题，在执行某些操作时可以接收新的输入，同时创建非阻塞任务，并且系统可以一次执行多个任务，提高资源的利用率。

由于异步执行任务，并且在完成后会报告结果，所以该范例特别适合事件驱动程序。此外，它还是一种通常用于用户界面、Web 服务器、网络通信或长时间运行的后台处理的范例。随着硬件发展到单个处理器芯片上的多个处理核心，可以使用异步编程来通过在不同核心上并行运行任务，充分利用所有可用的计算能力。

异步编程也有其挑战，例如：增加了复杂性，以及竞争条件。此外，错误处理和测试对于确保程序稳定性和避免出现问题也至关重要。

现代 C++ 还提供了异步机制，例如协程（可以暂停并稍后恢复的程序），或者 future 和 promise（作为异步程序中未知结果的代理，用于同步程序执行）。

1.3.4. 并行编程

通过并行编程，多个计算任务可以在多个处理单元上同时完成，这些处理单元可以全部位于同一台计算机（多核）上，也可以位于多台计算机（集群）。

主要有两种方法：

- 共享内存并行：任务可以通过共享内存（所有处理器都可以访问的内存空间）进行通信
- 消息传递并行：每个任务都有自己的内存空间，并使用消息传递技术与其他任务进行通信

与之前的范例一样，为了充分发挥潜力并避免错误或问题，并行计算需要同步机制以避免任务相互干扰。还需要平衡工作负载以充分发挥其潜力，以及减少创建和管理任务时的开销。这些需求增加了设计、实施和调试的复杂性。

1.3.5. 多线程编程

多线程编程是并行编程的一个子集，其中程序可分成多个线程，在同一进程内执行独立单元。进程、内存空间和资源在线程之间共享。

前面我们已经提到，共享内存需要同步机制。另一方面，由于不需要进程间通信，资源共享变得简单。例如，多线程编程通常用于实现具有流畅动画的图形用户界面（GUI）响应能力、在 Web 服务器中处理多个客户端的请求或数据处理。

1.3.6. 事件驱动式编程

事件驱动编程中，控制流由外部事件驱动。应用程序实时检测事件，并通过调用适当的事件处理方法或回调来响应这些事件。

事件表示需要采取的操作。事件循环会监听此事件，并不断监听传入的事件，并将其分派给相应的回调，从而执行所需的操作。由于代码仅在发生操作时执行，因此这种模式提高了资源使用效率和可扩展性。

事件驱动编程对于处理用户界面、实时应用程序和网络连接监听器中发生的动作很有用。与许多其他范式一样，增加的复杂性、同步和调试使得该范式的实现和应用变得复杂。由于 C++ 是一种低级语言，需要使用回调或函子等技术来编写事件处理程序。

1.3.7. 响应式编程

反应式编程处理数据流，即随时间连续的数据或值流。程序通常使用声明式或函数式编程构建，定义应用于流的运算符和转换的管道。这些操作使用调度程序和背压处理机制异步进行。

当数据量超出消费者承受能力，消费者无法处理所有数据时，就会发生背压。为了避免系统崩溃，反应式系统需要使用背压策略来防止系统故障。

其中一些策略包括：

- 通过请求发布者降低发布事件的速率来控制输入吞吐量。这可以通过遵循拉取策略来实现，即发布者仅在消费者请求时发送事件，或者通过限制发送的事件数量来实现，从而创建有限且受控的推送策略。
- 缓冲多余的数据，这在短时间内出现数据突发或高带宽传输时尤其有用。
- 删除一些事件或延迟其发布，直到消费者从背压状态中恢复为止。

反应式程序可以基于拉取，也可以基于推送。基于拉取的程序实现了从数据源主动拉取事件的经典情况，基于推送的程序通过信号网络推送事件以到达订阅者。订阅者对变化做出反应而不会阻塞程序，这使得这些系统非常适合响应性敏感的用户界面环境。

反应式编程就像一个事件驱动模型，其中来自各种来源的事件流可以转换、过滤、处理等。两者都增加了代码的模块化，适用于实时应用程序。但二者也存在一些差异：

- 反应式编程会对事件流做出反应，而事件驱动编程处理离散事件。
- 事件驱动编程中，事件会触发回调或事件处理程序。通过反应式编程，可以创建具有不同转换运算符的管道，让数据流流动并修改事件。

使用反应式编程的系统和软件的示例包括 X Windows 系统和 Qt、WxWidgets 和 Gtk+ 等库。反应式编程还用于实时传感器数据处理和仪表板，还适用于处理网络或文件 I/O 流量和数据处理。

要充分发挥反应式编程的潜力，在使用反应式编程时需要解决一些挑战。例如，调试分布式数据流和异步进程或通过微调调度程序来优化性能。此外，使用声明式或函数式编程，会让使用反应式编程技术开发软件变得更难理解和学习。

1.3.8. 数据流编程

使用数据流编程，程序设计为有向图，其中节点表示计算单元，边表示数据流，节点仅在有可用数据时执行。这种范式是由麻省理工学院的杰克·丹尼斯于 20 世纪 60 年代发明的。

数据流编程使代码和设计更具可读性和清晰度，提供了不同计算单元及其交互方式的可视化表示。此外，独立节点可以与数据流编程并行运行，从而提高并行性和吞吐量。

类似于反应式编程，但为建模系统提供了基于图形的方法和视觉辅助。

要实现数据流程序，可以使用哈希表。键标识一组输入，值描述要运行的任务。当给定键的所有输入都可用时，将执行与该键相关联的任务，从而生成触发哈希表中其他键的任务的其他输入值。在这些系统中，调度程序可以通过对图形数据结构进行拓扑排序来寻找并行机会，按相互依赖性对不同的任务进行排序。

这种范式通常用于机器学习的大规模数据处理管道、传感器或金融市场数据的实时分析，以及音频、视频和图像处理系统。使用数据流范式的软件库的示例有 Apache Spark 和 TensorFlow。在硬件方面，可以找到数字信号处理、网络路由、GPU 架构、遥测和人工智能等示例。

数据流编程的一种变体是增量计算，即只重新计算依赖于更改的输入数据的输出。这就像当单元格值发生变化时重新计算 Excel 电子表格中受影响的单元格。

了解了不同的并行编程系统、模型和范例，现在是时候介绍一些有助于衡量并行系统性能的指标了。

1.4. 并行性指标

指标是一种测量方法，可以帮助我们了解系统的运行情况并比较不同的改进方法。以下是一些常用于评估系统并行性的指标和公式。

1.4.1. 并行度

并行度（DOP）是衡量计算机同时执行的操作数量的指标，可用于描述并行程序和多处理器系统的性能。

计算 DOP 时，可以使用同时执行的最大操作数，测量没有瓶颈或依赖性的理想情况。或者，可以使用给定时间点的平均操作数或同时执行的操作数，反映系统实现的实际 DOP。可以使用分析器和性能分析工具来测量特定时间段内的线程数，从而进行近似计算。

DOP 不是一个常数，它是一个在应用程序执行过程中发生变化的动态指标。

例如，考虑一个处理多个文件的脚本工具。这些文件可以按顺序或同时处理，从而提高效率。如果有一台有 N 个核的机器，并且想要处理 N 个文件，可以为每个核分配一个文件。

按顺序处理所有文件的时间如下：

$$t_{total} = t_{file1} + t_{file2} + t_{file3} + ... + t_{fileN} \widetilde{=} N \cdot avg(t_{file})$$

并且，并行处理的时间为：

$$t_{total} = max(t_{file1}, t_{file2}, t_{file3}, ..., t_{fileN})$$

DOP 为 N，即主动处理单独文件的核数。

并行化所能实现的加速比有一个理论上限，由阿姆达尔定律给出。

1.4.2. 阿姆达尔定律

并行系统中，可以认为将 CPU 核心数量增加一倍可以使程序运行速度提高一倍，从而将运行时间减半，但并行化带来的加速并不是线性的。在一定数量的核心之后，由于上下文切换、内存分页等不同情况，运行时间不再减少。

阿姆达尔定律公式计算了任务并行化后理论上的最大加速比：

$$S_{max}(s) = \frac{s}{s+p(1-s)} = \frac{1}{1-p+\frac{p}{s}}$$

s 是改进部分的加速因子，p 是可并行化部分占整个流程的比例。因此，1-p 表示不可并行化任务（瓶颈或顺序部分）的比例，而 p/s 表示可并行化部分实现的加速，最大加速受任务的顺序部分限制。可并行化任务的比例越大（p 接近 1），最大加速就越高，直至达到加速因子（s）。另一方面，当顺序部分变大（p 接近 0）时，S_{max} 趋向于 1，则不可能有任何改进。

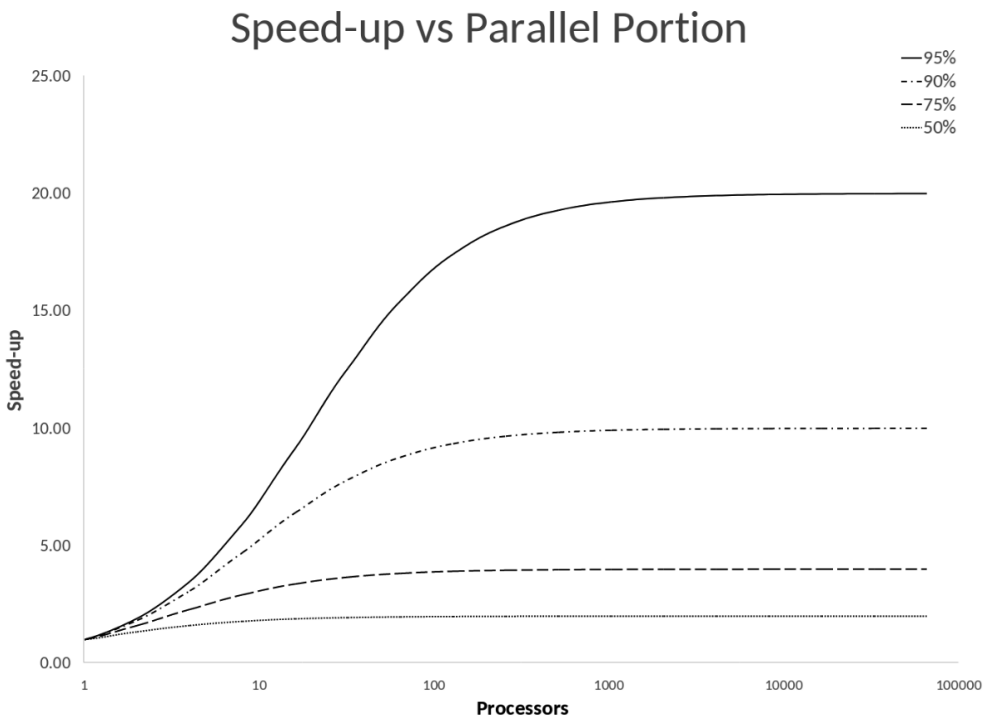

图 1.9：处理器数量和可并行部件百分比的加速限制

并行系统中的关键路径由最长的依赖计算链定义。由于关键路径几乎不可并行，因此它定义了顺序部分，从而决定了程序可以实现的更快运行时间。例如，如果一个进程的顺序部分占运行时间的 10%，那么可并行化部分的比例为 p=0.9。在这种情况下，无论有多少个处理器可用，潜在的加速都不会超过 10 倍。

1.4.3. 古斯塔夫森定律

阿姆达尔定律公式仅适用于固定规模的问题和不断增加的资源。当使用较大的数据集时，可并行化部分所花费的时间增长速度比顺序部分要快得多。而古斯塔夫森定律公式没那么悲观，也更准确，它考虑了固定的执行时间和使用资源不断增加的问题规模。

古斯塔夫森定律公式计算使用 `p` 个处理器所获得的加速比如下：

$$S_p = p + (1 - f) \cdot p$$

`p` 是处理器的数量，`f` 是保持连续的任务比例。因此，`(1-f)*p` 表示将 `(1-f)` 任务分布在 `p` 个处理器上进行并行化所实现的加速，`p` 表示增加资源时所做的额外工作。古斯塔夫森定律公式表明，降低 `f` 时，加速比受并行化影响，而增加 `p` 时，加速比受可扩展性影响。

与阿姆达尔定律一样，古斯塔夫森定律公式也是一种近似值，在衡量并行系统的改进时提供了有价值的视角。其他因素也会降低效率，例如：处理器之间的开销通信或内存和存储限制。

1.5. 总结

在本章中，了解了可用于构建并行系统的不同架构和模型。然后，探讨了可用于开发并行软件的各种并行编程范例的细节，并了解了其行为和细微差别。最后，定义了一些有用的指标来衡量并行程序的性能。

下一章中，将探讨硬件和软件之间的关系，以及软件如何映射和与底层硬件交互。还将了解什么是线程、进程和服务，线程如何调度，以及它们如何相互通信等内容。

1.6. 扩展阅读

- 拓扑排序：https://en.wikipedia.org/wiki/Topological_sorting
- C++ 编译器的支持：https://en.cppreference.com/w/cpp/compiler_support
- C++20 编译器的支持：https://en.cppreference.com/w/cpp/compiler_support/20
- C++23 编译器的支持：https://en.cppreference.com/w/cpp/compiler_support/23

第 2 章 进程、线程和服务

异步编程涉及启动操作而不等待任务完成再继续执行下一个任务，这种非阻塞行为允许开发高响应性和高效率的应用程序，能够同时处理大量操作，而不会出现延迟或浪费计算资源的等待。

异步编程非常重要，尤其是在网络应用程序、用户界面和系统编程的开发中。开发人员能够创建能够管理大量请求、执行输入/输出（I/O）操作或高效执行并发任务的应用程序，从而显著增强用户体验和应用程序性能。

Linux 操作系统（本书中，将重点介绍在代码无法独立于平台的情况下在 Linux 操作系统上进行开发）具有强大的进程管理、对线程的本机支持和高级 I/O 功能，是开发高性能异步应用程序的理想环境。这些系统提供了一组丰富的功能，例如：用于进程和线程管理的强大 API、非阻塞 I/O 和进程间通信（IPC）机制。

本章介绍了 Linux 环境中异步编程所必需的基本概念和组件。

我们将探讨以下主题：

- Linux 中的进程
- 服务和守护进程
- 线程和并发

本章结束时，将对 Linux 中的异步编程环境有一个基本的了解，为后续章节的更深入探索和实际应用奠定基础。

2.1. Linux 中的进程

进程可以定义为正在运行的程序的一个实例，包括程序代码、属于此进程的所有线程（由程序计数器表示）、堆栈（包含临时数据（如函数参数、返回地址和局部变量）的内存区域）、堆（用于动态分配的内存）及其包含全局变量和初始化变量的数据部分。每个进程都在自己的虚拟地址空间内运行，并与其他进程隔离，确保其操作不会直接干扰其他进程的操作。

2.1.1. 进程生命周期——创建、执行和终止

进程的生命周期可以分为三个阶段：

- **创建**：使用 fork() 系统调用创建新进程，该系统调用通过复制现有进程来创建新进程。调用 fork() 的进程为父进程，新创建的进程为子进程。此机制对于在系统内执行新程序至关重要，并且是同时执行不同任务的前提。
- **执行**：创建后，子进程可能会执行与父进程相同的代码，或者使用 exec() 系列系统调用来加载和运行不同的程序。如果父进程有多个执行线程，则只有调用 fork() 的线程会在子进程中重复，所以子进程只包含一个线程：执行 fork() 系统调用的线程。

  由于只有调用 fork() 的线程才会复制到子线程，在 fork 时其他线程持有的任何互斥（mutexes）、条件变量或其他同步原语在父线程中仍保持其当时的状态，但不会转移到子线程中。这可能会导致复杂的同步问题，因为其他线程锁定的互斥（子线程中不存在）可能会保持锁定状态，如果子线程尝试解锁或等待这些原语，则可能导致死锁。

在此阶段，进程执行其指定的操作，例如：读取或写入文件，以及与其他进程通信。

- **终止**：进程要么主动终止（通过调用 exit() 系统调用），要么非主动终止（由于收到另一个进程发出的终止信号）。终止时，进程会向其父进程返回退出状态，并将其资源释放回系统。

因为支持多个任务的并发执行，所以进程生命周期对于异步操作来说是不可或缺的一部分。

每个进程都由一个进程 ID（PID）唯一标识，这是一个整数，内核使用它来管理进程。PID 用于控制和监视进程。父进程还使用 PID 与子进程通信或控制子进程的执行，例如：等待子进程终止或发送信号。

Linux 提供了进程控制和信号机制，允许异步管理和通信进程。信号是 IPC 的主要方式之一，使进程能够中断或接收事件通知。例如，kill 命令可以发送信号来停止进程，或提示其重新加载配置文件。

进程调度是 Linux 内核为进程分配 CPU 时间的方式。调度程序根据优化响应能力和效率等因素的调度算法和策略，来确定在给定时间运行哪个进程。进程可以处于各种状态，例如：正在运行、等待或停止。调度程序会在这些状态之间转换，以有效地管理执行进程。

2.1.2. 探索 IPC

Linux 操作系统中，进程独立运行，无法直接访问其他进程的内存空间。当多个进程需要通信和同步其操作时，进程的独立性会带来挑战。为了应对这些挑战，Linux 内核提供了一套多功能的 IPC 机制。每种 IPC 机制都经过量身定制，以适应不同的场景和需求，使开发人员能够构建复杂、高性能的应用程序，并有效利用异步处理。

对于旨在创建可扩展且高效的应用程序的开发人员来说，了解这些 IPC 技术至关重要。IPC 允许进程交换数据、共享资源并协调其活动，从而促进软件系统不同组件之间顺畅可靠的通信。通过利用适当的 IPC 机制，开发人员可以在应用程序中实现更高的吞吐量、更低的延迟和更高的并发性，从而实现更好的性能和用户体验。

多任务环境中，多个进程同时运行，IPC 在实现任务的高效和协调执行方面起着至关重要的作用。例如，一个处理来自客户端的多个并发请求的 Web 服务器应用程序。Web 服务器进程可能使用 IPC 与负责处理每个请求的子进程进行通信。这种方法允许 Web 服务器同时处理多个请求，从而提高应用程序的整体性能和可扩展性。

IPC 必不可少的另一个常见场景是分布式系统或微服务架构，多个独立进程或服务需要进行通信和协作以实现共同目标。消息队列和套接字或远程过程调用（RPC）等 IPC 机制使这些进程能够交换消息、调用远程对象上的方法并同步其操作。

通过利用 Linux 内核提供的 IPC 机制，开发人员可以设计多个进程可以和谐协作的系统。这样就可以创建复杂、高性能的应用程序，这些应用程序可以高效利用系统资源、有效处理并发任务，并轻松扩展以满足日益增长的需求。

Linux 中的 IPC 机制

Linux 支持多种 IPC 机制，每种机制都有其特点和用例。

Linux 操作系统支持的基本 IPC 机制包括共享内存（通常用于单个服务器上的进程通信）和套接字（方便服务器间通信）。还有其他机制（本文将简要介绍），但最常用的是共享内存和套接字：

- **管道**：管道是 IPC 的最简单形式之一，允许进程之间进行单向通信。命名管道或先进先出（FIFO）扩展了此概念，通过提供可通过文件系统中的名称访问的管道，允许不相关的进程进行通信。
- **信号**：信号是一种软件中断，可以发送给进程以通知其事件。虽然信号不是传输数据的方法，但对于控制进程行为和触发进程内的操作非常有用。
- **消息队列**：消息队列允许进程以先进先出的方式交换消息。与管道不同，消息队列支持异步通信，即消息存储在队列中，接收进程可以在方便时检索消息。
- **信号量**：信号量用于同步，帮助进程管理对共享资源的访问。通过确保只有指定数量的进程，可以在给定时间访问资源来防止竞争条件。
- **共享内存**：共享内存是 IPC 中的一个基本概念，可使多个进程能够访问和操作同一段物理内存。提供了一种在不同进程之间交换数据的超快方法，减少了耗时的数据复制操作，这种技术在处理大型数据集或需要高速通信时特别有利。共享内存的机制涉及创建共享内存段，这是多个进程可访问的专用物理内存部分。此共享内存段可视为公共工作区，允许进程读取、写入和协作修改数据。为了确保数据完整性并避免冲突，共享内存需要同步机制，例如：信号量或互斥锁。这些机制规范对共享内存段的访问，防止多个进程同时修改同一数据。这种协调对于保持数据一致性和避免覆盖或损坏至关重要。

  在性能至关重要的单服务器环境中，共享内存通常是首选的 IPC 机制，其主要优势在于速度。由于数据直接在物理内存中共享，无需中间复制或上下文切换，因此显著降低了通信开销，并最大限度地减少了延迟。

  然而，共享内存也有一些注意事项，需要仔细管理以防止条件竞争和内存泄漏。访问共享内存的进程必须遵守明确定义的协议，以确保数据完整性并避免死锁。此外，共享内存通常作为系统级功能实现，需要特定的操作系统支持，并可能引入特定于平台的依赖关系。

  尽管存在这些考虑，共享内存仍然是一种强大且广泛使用的 IPC 技术，特别是在速度和性能是关键因素的应用程序中。
- **套接字**：套接字是操作系统中 IPC 的基本机制，为进程提供了一种相互通信的方式，无论是在同一台机器内还是跨网络。套接字用于建立和维护进程之间的连接，支持面向连接和无连接通信。

  面向连接通信是一种在传输数据之前，在两个进程之间建立可靠连接的通信类型。因为需要确保所有数据都以可靠的方式按正确的顺序传输非常重要，这种类型的通信通常用于文件传输和远程登录等应用程序。

  无连接通信是一种在传输数据之前，在两个进程间不建立可靠连接的通信类型。因为需要低延迟比保证所有数据的可靠传输更重要，所以这种类型的通信通常用于流媒体和实时游戏等应用程序。

  套接字是网络应用程序的支柱。会在各种各样的应用程序中使用，包括 Web 浏览器、电子邮件客户端和文件共享应用程序。套接字还用于许多操作系统服务使用，例如网络文件系统（NFS）和域名系统（DNS）。

  以下是使用套接字的一些主要优点：

  - **可靠性**：套接字提供了一种可靠的进程间通信方式，即使这些进程位于不同的机器上。
  - **可扩展性**：套接字可用于支持大量并发连接，使其成为需要处理大量流量的应用程序的

理想选择。
- **灵活性**：套接字可用于实现多种通信协议，适用于各种应用程序。
- **使用 IPC**：套接字是 IPC 的强大工具。可在各种各样的应用程序中使用，对于构建可扩展、可靠且灵活的网络应用程序至关重要。

基于微服务的应用程序是异步编程的一个示例，使用不同的进程以异步方式在它们之间进行通信，一个简单的例子是日志处理器。不同的进程生成日志条目并将其发送到另一个进程进行进一步处理，例如：特殊格式、重复数据删除和统计。生产者只需发送日志行，而无需等待接收日志进程的回复。

本节中，了解了 Linux 中的进程、生命周期以及操作系统如何实现 IPC。下一节中，将介绍一种特殊的 Linux 进程，称为守护进程。

2.2. Linux 中的服务和守护进程

Linux 操作系统领域，守护进程是一个基本组件，在后台运行，默默地执行基本任务。这些进程传统上用以字母 d 结尾的名称来标识，例如 ssh**d** 表示安全 Shell（SSH）守护进程，http**d** 表示 Web 服务器守护进程。处理对操作系统上运行的系统级任务着至关重要。

守护进程可用于多种用途，包括文件服务、Web 服务和网络通信，以及日志记录和监控服务。守护进程具有自主性和弹性，从系统启动时开始运行持续运行，直到系统关闭。与用户启动和控制的普通进程不同，守护进程具有以下特点：

- 后台操作：
  - 守护进程在后台运行
  - 无用于直接用户交互的控制终端
  - 不需要用户界面或手动干预执行
- 用户独立性：
  - 守护进程独立于用户运行
  - 自主运行，无需用户参与
  - 待系统事件或特定请求来触发
- 以任务为导向：
  - 每个守护进程都经过定制，以执行特定任务或一组任务
  - 旨在处理特定功能或监听特定事件或请求
  - 可确保高效执行任务

创建守护进程不仅涉及在后台运行进程。为了确保守护进程有效运行，开发人员必须考虑几个关键：

1. 终端分离：使用 fork() 系统调用将守护进程从终端分离。父进程在 fork 之后退出，子进程则在后台运行。
2. 会话创建：setsid() 系统调用创建新会话，并将调用进程指定为会话和进程组的领导者。此步骤对于完全从终端分离至关重要。
3. 更改目录：防止阻塞文件系统的卸载，守护进程通常将其工作目录更改为根目录。

4. 处理文件描述符：守护进程关闭继承的文件描述符，stdin、stdout 和 stderr 通常会重定向到 /dev/null。
5. 信号处理：正确处理信号（例如：用于重新加载配置的 SIGHUP 或用于正常关闭的 SIGTERM）对于有效的守护进程管理至关重要。

守护进程通过各种 IPC 机制与其他进程或守护进程进行通信，是许多异步系统架构不可或缺的一部分，提供基本服务无需直接与用户交互。

守护进程的一些经典用例：

- Web 服务器：httpd 和 nginx 等守护进程响应客户端请求提供网页，同时处理多个请求并确保无缝的网页浏览。
- 数据库服务器：mysqld 和 postgresql 等守护进程管理数据库服务，允许各种应用程序异步访问和操作数据库。
- 文件服务器：smbd、nfsd 等守护进程提供网络文件服务，实现不同系统之间的异步文件共享和访问。
- 日志记录和监控：syslogd 和 snmpd 等守护进程收集和记录系统事件，提供系统健康和性能的异步监控。

总而言之，守护进程是 Linux 系统必不可少的组件，其自主性和弹性维护着系统稳定性，并为用户和应用程序提供基本服务。

我们已经了解了进程和守护进程（一种特殊的进程）。一个进程可以有一个或多个执行线程。在下一节中，我们将介绍线程。

2.3. 线程

进程和线程代表并发执行的两种基本方式，但二者在操作和资源管理方面存在很大差异。进程是正在运行的程序的实例，拥有一系列私有资源，包括内存、文件描述符和执行上下文。进程彼此隔离，单个进程的故障通常不会影响其他进程，从而可为整个系统提供了强大的稳定性。

线程是计算机科学中的一个基本概念，代表在单个进程内执行多个任务的一种轻量且高效的方式。与具有私有内存空间和资源的独立进程不同，线程与其所属的进程交织在一起。这种密切的关系允许线程共享相同的内存空间和资源，包括文件描述符、堆内存，以及进程分配的其他全局数据。

线程的主要优势是能够高效地通信和共享数据。由于进程内的所有线程共享相同的内存空间，因此可以直接访问和修改公共变量，而无需通过 IPC 机制进行消息的传递。这种共享环境可实现快速数据交换，并有助于实现并发算法和数据结构。

然而，共享同一内存空间也对管理共享资源的访问带来了不小的挑战。为了避免数据损坏并确保共享数据的完整性，线程必须采用同步机制，例如：锁、信号量或互斥锁。这些机制遵循了执行访问共享资源的规则和协议，确保在限定时间内只有一个线程可以访问特定资源。

同步在多线程编程中至关重要，以避免竞争条件、死锁和其他与并发相关的问题。为了应对这些挑战，计算机科学家们开发了各种同步原语和技术。其中包括互斥锁（对共享资源的独占访问）、信号量（对有限数量的资源进行受控访问）和条件变量（使线程能够等待特定条件得到满足后再继续执行）。

通过管理同步并采用适当的并发模式，开发人员可以利用线程的强大功能，在应用程序中实现高性能和可扩展性。线程特别适合可并行化的任务，例如：图像处理、科学模拟和 Web 服务器，可以同时执行多个独立计算。

如前所述，线程是系统线程，所以它们由内核创建和管理。但有些场景中需要大量线程，系统可能没有足够的资源来创建大量系统线程，解决这个问题的方法是使用用户线程。实现用户线程的一种方法是通过协程，协程自 C++20 起已被 C++ 标准支持。

协程是 C++ 中一个较新的功能。协程定义为可在特定点暂停和恢复的函数，可在单个线程内进行协作式多任务处理。与不间断运行的标准函数不同，协程可以暂停执行并将控制权交还给调用者，后者稍后可以从暂停点恢复协程的执行。

协程比系统线程轻量得多，所以可以更快地创建和销毁，并且所需的开销更少。并且协程有协作性，必须明确地将控制权移交给调用者才能切换执行上下文，能使用户程序能够更好地控制协程的执行。

协程可用于创建各种不同的并发模式。例如，协程可用于实现任务，这些任务是可以调度并同时运行的轻量级工作单元。协程还可用于实现通道，属于协程间传递数据的通信通道。

协程可以分为有栈和无栈两类，C++20 协程是无栈的。我们将在第 8 章深入了解这些概念。

总体而言，协程是 C++ 中创建并发程序的强大工具。它们轻量级、协作性强，可用于实现各种不同的并发模式。协程不能完全用于实现并行性，因为其仍然需要 CPU 执行上下文，而这只能由线程提供。

2.3.1. 线程的生命周期

系统线程（通常称为轻量级进程）的生命周期包括从创建到终止的各个阶段，每个阶段在并发编程环境中管理和利用线程方面都发挥着至关重要的作用：

1. 创建：此阶段始于系统中创建新线程时。创建过程涉及使用函数，该函数需要几个参数。一个关键参数是线程的属性，例如：调度策略、堆栈大小和优先级。另一个重要参数是线程将执行的函数，成功创建后，线程将分配堆栈和其他资源。
2. 执行：创建后，线程开始执行其指定的启动函数。在执行期间，线程可以独立执行各种任务，或在必要时与其他线程进行交互。线程还可以创建和管理自己的局部变量和数据结构，使其自成一体并能够同时执行特定任务。
3. 同步：为了确保有序访问共享资源避免数据损坏，线程采用同步机制。常见的同步原语包括锁、信号量和栅栏。适当的同步允许线程协调其活动状态，避免并发编程中可能出现的竞争条件、死锁和其他问题。
4. 终止：线程可以通过多种方式终止，可以显式调用函数来终止自身，也可以通过从其启动函数返回来终止运行。某些情况下，线程可以被另一个使用该函数的线程取消。终止后，系统将回收分配给该线程的资源，并释放该线程持有的待处理操作或锁。

了解系统线程的生命周期，对于设计和实现并发程序至关重要。通过仔细管理线程的创建、执行、同步和终止，开发人员可以创建高效且可扩展的应用程序，以充分利用并发的优势。

2.3.2. 线程调度

系统线程由操作系统内核的调度程序管理，具有抢占式调度。调度程序根据线程优先级、分配时间或互斥阻塞等因素，决定何时在线程之间切换执行。这种由内核控制的上下文切换可能会产生大量开销，上下文切换的成本非常高，加上每个线程的资源使用量（例如：其私有的堆栈），使得协程成为更高效替代方案，可以在单个线程中运行多个协程。

协程具有多种优势，减少了与上下文切换相关的开销。由于协程的 `Yield` 或 `Await` 对上下文的切换在用户空间代码进行（非内核处理），该过程更加轻量和高效。这可以显著提高性能，尤其是在上下文切换频繁的情况下。

协程还提供了对线程调度的控制，开发人员可以根据其应用程序的需要定义调度策略。这种灵活性可以微调线程管理、优化资源利用率，并实现所需的性能特征。

协程的另一个重要特性是与系统线程相比，协程通常更轻量。协程不维护自己的堆栈，这是一个很大的资源消耗优势，使其适配资源受限的环境。

总体而言，协程提供了一种更高效、更灵活的线程管理方法，尤其是在需要频繁切换上下文或对线程调度进行细粒度控制的情况下。线程可以访问内存进程，并且该内存由所有线程共享，因此需要小心并控制内存访问。这种控制可通过“同步原语”机制实现。

2.4. 同步原语

同步原语是管理多线程编程并发访问共享资源的重要工具。有几种同步原语，每种都有各自的特定用途和特征：

- 互斥锁：用于强制对代码的关键部分进行独占访问。线程可以锁定互斥锁，从而阻止其他线程进入受保护的部分，直到互斥锁解锁。互斥锁保证在给定时间内只有一个线程可以执行关键部分，从而确保数据完整性并避免竞争条件。
- 信号量：比互斥量更通用，可用于更广泛的同步任务，包括线程之间的信号发送。信号量维护一个整数计数器，线程可以将其递增（信号发送）或递减（等待）。信号量允许更复杂的协调模式，例如：计数信号量（用于资源分配）和二进制信号量（类似于互斥量）。
- 条件变量：条件变量用于根据特定条件进行线程同步。线程可以阻塞（等待）条件变量，直到特定条件为真。其他线程可以向条件变量发送信号，从而唤醒等待的线程并继续执行。条件变量通常与互斥锁一起使用，以实现更细粒度的同步，并避免忙等待。
- 附加同步原语：除了前面讨论的核心同步原语之外，还有其他几种同步机制：
  - 栅栏：允许一组线程同步执行，确保所有线程在继续执行之前都达到某个点
  - 读写锁：读写锁提供了一种控制共享数据并发访问的方法，允许多个读取器但一次只能有一个写入器
  - 自旋锁：自旋锁是一种互斥锁，涉及忙等待，持续检查内存位置，直到可用

在第 4 章和第 5 章中，将深入了解 C++ 标准模板库（STL）中实现的同步原语，以及使用它们的示例。

2.4.1. 选择正确的同步原语

选择适当的同步原语取决于应用程序的具体要求，以及所访问的共享资源的性质。以下是一些准则：

- 互斥锁：需要对关键部分进行独占访问以确保数据完整性并防止竞争条件时，请使用互斥锁
- 信号量：需要更复杂的协调模式时使用信号量，例如：资源分配或线程之间的信号传递
- 条件变量：线程需要等待特定条件变为真才能继续执行时，请使用条件变量

有效使用同步原语，对于开发安全高效的多线程程序至关重要。通过了解不同同步机制的用途和特性，开发人员可以根据其特定需求选择最合适的原语，实现可靠且和可预测的并发程序。

2.5. 使用多线程时的常见问题

线程化带来了一些挑战，必须加以管理才能确保应用程序的正确性和性能。这些挑战源于多线程编程固有的并发性和不确定性。

- 当多个线程同时访问和修改共享数据时，就会发生竞争条件。结果取决于线程操作的非确定性顺序，这可能导致不可预测和不一致的结果。例如，有两个线程正在更新共享计数器，如果线程同时增加计数器，则由于竞争条件，最终值可能不正确。
- 当两个或多个线程无限期地等待彼此持有的资源时，就会发生死锁。这会形成无法解决的依赖关系循环，导致线程永久阻塞。例如，有两个线程正在等待彼此释放对共享资源的锁定，如果两个线程都没有释放其持有的锁，就会发生死锁。
- 当线程不断被拒绝访问其需要的资源时，就会发生饥饿。当其他线程不断获取并持有资源，导致饥饿线程无法执行时，就会发生这种情况。
- 活锁类似于死锁，但不会永久阻塞，线程仍然保持活动状态并反复尝试获取资源。

有多种技术可用于解决线程难题，其中包括：

- 同步机制：可以使用锁和互斥锁等同步原语来控制对共享数据的访问，并确保一次只有一个线程可以访问数据。
- 死锁预防和检测：死锁预防算法可用于避免死锁，而死锁检测算法可用于在发生死锁时识别和解决死锁。
- 线程调度：线程调度算法可用于确定哪个线程应在给定时间运行，以及哪个线程可避免资源匮乏并提高应用程序性能。

我们将更详细地了解多线程问题的不同解决方案。

2.6. 有效线程管理的策略

有多种处理线程的方法，可以避免多线程问题。以下是一些最常见的处理方法：

- 最小化共享状态：将线程设计为尽可能多地操作私有数据可显著减少同步需求。通过使用线程本地存储为线程特定数据分配内存，消除了对全局变量的需求，进一步降低了数据争用的可能性。通过同步原语仔细管理共享数据访问对于确保数据完整性至关重要。这种方法通过

最小化同步需求，并确保以受控且一致的方式访问共享数据，提高了多线程应用程序的效率和正确性。

- 锁的层次结构：建立明确定义锁的层次结构，对于防止多线程编程中的死锁至关重要。锁的层次结构规定了获取和释放锁的顺序，从而确保跨线程的锁定模式一致。通过以从最粗到最细的粒度分层方式获取锁，可以显著降低死锁的概率。

  粗粒度的锁用于控制对大部分共享资源的访问，而细粒度的锁用于对资源的特定细粒度部分进行访问。首先获取粗粒度锁，线程可以获得对大部分资源的独占访问，从而降低与试图访问同一资源的其他线程发生冲突的可能性。当获取了粗粒度锁，就可以获取更细粒度的锁来控制对特定资源部分的访问，从而提供更精细的控制并减少其他线程的等待时间。

  某些情况下，可以使用无锁数据结构来完全消除对锁的需求。无锁数据结构旨在提供对共享资源的并发访问，而无需显式锁定。相反，它们依靠原子操作和非阻塞算法来确保数据完整性和一致性。通过无锁数据结构，可以消除与锁获取和释放相关的开销，从而提高多线程应用程序的性能和可扩展性：

- 超时：避免线程在尝试获取锁时无限期等待，设置获取锁的超时非常重要。确保线程无法在指定的超时期限内获取锁时，将自动放弃并稍后重试。这有助于防止死锁并确保没有线程无限期等待。

- 线程池：管理可重用线程池是优化多线程应用程序性能的关键技术。通过动态创建和销毁线程，可以显著减少线程创建和终止的开销。线程池的大小应根据应用程序的工作负载和资源限制进行调整，太小的线程池可能会导致任务等待可用线程，而太大的线程池可能会浪费资源。工作队列用于管理任务并将其分配给池中的可用线程。任务也会添加到队列中，并由线程按照 FIFO 顺序进行处理。这确保了公平性并避免了任务匮乏。使用工作队列还可以实现负载平衡，任务可以均匀分布在可用线程中。

- 同步原语：了解不同类型的同步原语，例如：互斥锁、信号量和条件变量。根据特定场景的同步要求选择合适的原语。正确使用同步原语，避免竞争条件和死锁。

- 测试和调试：全面测试多线程应用程序以识别和修复线程问题。使用线程清理器和分析器等工具来检测数据竞争和性能瓶颈。采用逐步执行和线程转储等调试技术，来分析和解决线程问题。

- 可扩展性和性能考量：设计线程安全的数据结构和算法以确保可扩展性和性能。平衡线程数量和可用资源以避免超额分配，监控 CPU 利用率和线程争用等系统指标以识别潜在的性能瓶颈。

- 沟通与协作：促进多线程代码开发人员之间的协作，以确保一致性和正确性。建立线程管理的编码指南和最佳实践，以保持代码质量和可读性。随着应用程序的发展，定期审查和更新线程策略

线程是一种强大的工具，可用于提高应用程序的性能和可扩展性。但是，了解线程的挑战并使用适当的技术来应对这些挑战非常重要。这样，开发人员就可以创建正确、高效且可靠的多线程应用程序。

2.7. 总结

本章中，探讨了操作系统中的进程概念。进程是执行程序和管理计算机上资源的基本实体。深入研究了进程生命周期，研究了进程从创建到终止所经历的各个阶段。此外，还讨论了 IPC，其对于进程之间的交互和信息交换至关重要。

此外，在 Linux 操作系统中介绍了守护进程。守护进程是一种特殊类型的进程，作为服务在后台运行，并执行特定任务，例如：管理系统资源、处理网络连接或为系统提供其他基本服务。还探讨了系统线程和用户线程的概念，是与父进程共享相同地址空间的轻量级进程。并讨论了多线程应用程序的优势，包括改进的性能和响应能力，以及在单个进程内管理和同步多个线程所面临的挑战。

了解多线程产生的不同问题是了解如何解决这些问题的基础。在下一章中，将介绍如何创建线程，然后在第 4 章和第 5 章中，将深入研究标准 C++ 提供的不同同步原语及其不同的应用。

2.8. 扩展阅读

- [Butenhof, 1997] David R. Butenhof, Programming with POSIX Threads, Addison Wesley, 1997.
- [Kerrisk, 2010] Michael Kerrisk, The Linux Programming Interface, No Starch Press, 2010.
- [Stallings, 2018] William Stallings, Operating Systems Internals and Design Principles, Ninth Edition, Pearson Education 2018.
- [Williams, 2019] Anthony Williams, C++ Concurrency in Action, Second Edition, Manning 2019.

第二部分 高级线程管理和同步技术

第二部分中，将以并行编程的基础知识为基础，深入研究管理线程和同步并发操作的高级技术。探索线程创建和管理、跨线程异常处理和高效线程协调等基本概念，深入了解关键同步原语，包括互斥锁、信号量、条件变量和原子操作。所有这些知识将提供实现基于锁和无锁的多线程解决方案所需的工具，一窥高性能并发系统，并提供管理多线程系统时避免常见陷阱（如竞争条件、死锁和活锁）所需的技能。

本部分包含以下章节：

- 第 3 章，如何在 C++ 中创建和管理线程
- 第 4 章，使用锁进行线程同步
- 第 5 章，原子操作

第 3 章 如何在 C++ 中创建和管理线程

正如前两章中了解到的，线程是程序中最小、最轻量的执行单元。每个线程负责处理由操作系统调度程序在分配的 CPU 资源上运行的一系列指令定义的独特任务。管理程序中的并发性以最大限度地提高 CPU 资源的整体利用率时，线程起着至关重要的作用。

程序启动过程中，内核将执行权交给进程后，C++ 运行时会创建主线程并执行 main() 函数。此后，可以创建其他线程，将程序拆分为可并发运行并共享资源的不同任务。这样，程序就可以处理多个任务，从而提高效率和响应能力。

本章中，将介绍如何使用现代 C++ 功能创建和管理线程的基础知识。后续章节中，将介绍 C++ 锁同步原语（互斥锁、信号量、栅栏和自旋锁）、无锁同步原语（原子变量）、协调同步原语（条件变量），以及使用 C++ 解决或避免使用并发或多线程时的潜在问题（竞争条件或数据竞争、死锁、活锁、饥饿、超额分配、负载平衡和线程耗尽）的方法。

本章中，将讨论以下主要主题：

- 如何在 C++ 中创建、管理和取消线程
- 如何向线程传递参数并从线程获取结果
- 如何让线程休眠或让其他线程执行
- jthread 对象是什么

3.1. 技术要求

本章中，将使用 C++11 和 C++20 开发不同的解决方案。因此，需要安装 GNU 编译器集合（GCC），特别是 GCC 13，以及 Clang 8（有关 C++ 编译器支持的更多详细信息，请参阅https://en.cppreference.com/w/cpp/compiler_support）。

可以在 https://gcc.gnu.org 上找到有关 GCC 的更多信息，可以在此处找到有关如何安装 GCC 的信息：https://gcc.gnu.org/install/index.html。

有关 Clang（支持多种语言（包括 C++）的编译器前端）的更多信息，请访问 https://clang.llvm.org。Clang 是 LLVM 编译器基础架构项目（https://llvm.org）的一部分。Clang 中的 C++ 支持记录在此处：https://clang.llvm.org/cxx_status.html。

本书中的一些代码未显示所包含的库，一些函数（甚至是主要函数）可能会简化，仅显示相关指令。可以在以下 GitHub 库中找到所有完整代码：https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP。

GitHub 库根目录中的 scripts 文件夹下，可以找到一个名为 install_compilers.sh 的脚本，该脚本可能有助于在基于 Debian 的 Linux 系统中安装所需的编译器。该脚本已在 Ubuntu 22.04 和 24.04 中进行了测试。

本章的示例位于 Chapter_03 文件夹下。所有源代码文件都可以使用 C++20 和 CMake 进行编译：

```
cmake . && cmake —build .
```

可执行文件将在 bin 目录下生成。

3.2. 线程库–简介

C++ 中创建和管理线程的主要库是线程库，让我们回顾一下线程，然后再深入了解线程库提供的功能。

3.2.1. 什么是线程？回顾一下

线程的目的是在一个进程中同时执行多个任务。

线程有自己的堆栈、本地数据和 CPU 寄存器，例如指令指针（IP）和堆栈指针（SP），但共享其父进程的地址空间和虚拟内存。

用户空间中，可以区分本机线程和轻量级或虚拟线程。本机线程是操作系统在使用某些内核 API 时创建的线程。C++ 线程对象创建和管理这些类型的线程。另一方面，轻量级线程类似于本机线程，只由运行时或库模拟。在 C++ 中，协程属于这一组。轻量级线程比本机线程具有更快的上下文切换速度。此外，多个轻量级线程可以在同一个本机线程中运行，并且比本机线程小得多。

本章将介绍原生线程。第 8 章将介绍以协程形式出现的轻量级线程。

3.2.2. C++ 线程库

在 C++ 中，线程允许多个函数同时运行。线程类定义了一个类型安全的本机线程接口。在标准模板库（STL）中的 <thread> 头文件中的 std::thread 库中定义，自 C++11 起可用。

C++ STL 中包含线程库之前，开发人员使用特定于平台的库，例如：Unix 或 Linux 操作系统中的 POSIX 线程（pthread）库、Windows NT 和 CE 系统的 C 运行时（CRT）和 Win32 库或第三方库（例如：Boost.Threads）。本书中，将仅使用现代 C++ 功能。由于 <thread> 可用并在特定于平台的机制之上提供了可移植的抽象，因此不会使用或解释这些库中的任何一个。第 9 章中，将介绍 Boost.Asio。在第 10 章中，将介绍 Boost.Cobalt。这两个库都提供了处理异步 I/O 操作和协程的高级框架。

现在是时候了解不同的线程操作了。

3.3. 线程操作

本节中，将介绍如何创建线程、在构造期间传递参数、从线程返回值、取消线程执行、捕获异常等。

3.3.1. 创建线程

线程创建后会立即执行，只会被操作系统调度所延迟。如果没有足够的资源来并行运行父线程和子线程，则其运行顺序不确定。

构造函数参数定义线程要执行的函数或函数对象。此可调用对象不返回任何内容，将忽略其返回值。如果由于某种原因线程执行以异常结束，则除非捕获到异常，否则调用 std::terminate。

下面的例子中，使用不同的可调用对象创建了六个线程。

t1 使用函数指针：

```
void func() {
    std::cout << "Using function pointer\n";
}
std::thread t1(func);
```

t2 使用 lambda 函数：

```
auto lambda_func = []() {
    std::cout << "Using lambda function\n";
};
std::thread t2(lambda_func);
```

t3 使用嵌入的 lambda 函数：

```
std::thread t3([]() {
    std::cout << "Using embedded lambda function\n";
});
```

t4 使用 operator() 重载的函数对象：

```
class FuncObjectClass {
public:
    void operator()() {
        std::cout << "Using function object class\n";
    }
};
std::thread t4{FuncObjectClass()};
```

t5 使用非静态成员函数，通过传递成员函数的地址和对象的地址来调用成员函数：

```
class Obj {
public:
    void func() {
        std::cout << "Using a non-static member function"
                  << std::endl;
    }
};
Obj obj;
std::thread t5(&Obj::func, &obj);
```

t6 使用静态成员函数，由于方法是静态的，只需要成员函数的地址：

```
class Obj {
public:
    static void static_func() {
```

```
        std::cout << "Using a static member function\n";
    }
};
std::thread t6(&Obj::static_func);
```

线程创建会产生一些开销，可以使用线程池来减少这些开销，这些将在第 4 章中进行探讨。

检查硬件的并发性

有效线程管理的策略之一是平衡线程数和可用资源，以避免过度分配，这与可扩展性和性能有关。

要检索操作系统支持的并发线程数，可以使用 `std::thread::hardware_concurrency()`：

```
const auto processor_count = std::thread::hardware_concurrency();
```

必须认为此函数返回的值，仅提供有关将同时运行的线程数的提示。有时定义不明确，因此返回值为 0。

3.3.2. 同步流写入

当两个或多个线程使用 `std::cout` 将消息打印到控制台时，输出结果可能会很混乱。这是由于输出流中发生了竞争。

如上一章所述，条件竞争是并发和多线程程序中发生的软件错误，其行为取决于共享资源上发生的事件序列，其中至少有一个操作不是原子的。

以下代码显示了两个线程打印数字序列。`t1` 线程应打印包含“1 2 3 4”，`t2` 线程应打印“5 6 7 8”。每个线程打印其序列 100 次。在主线程退出之前，使用 `join()` 等待 `t1` 和 `t2` 完成。

```
#include <iostream>
#include <thread>

int main() {
    std::thread t1([]() {
        for (int i = 0; i < 100; ++i) {
            std::cout << "1 " << "2 " << "3 " << "4 "
            << std::endl;
        }
    });

    std::thread t2([]() {
        for (int i = 0; i < 100; ++i) {
            std::cout << "5 " << "6 " << "7 " << "8 "
            << std::endl;
        }
    });
```

```
18
19      t1.join();
20      t2.join();
21
22      return 0;
23  }
```

运行后，会显示以下内容：

```
6 1 2 3 4
1 5 2 6 3 4 7 8
1 2 3 5 6 7 8
```

为了避免这些问题，可以简单地从特定线程写入或使用对 `std::cout` 对象进行原子调用的 `std::ostringstream` 对象：

```
1  std::ostringstream oss;
2  oss << "1 " << "2 " << "3 " << "4 " << "\n";
3  std::cout << oss.str();
```

从 C++20 开始，还可以使用 `std::osyncstream` 对象。其行为类似于 `std::cout`，在访问同流的线程之间具有写入同步。由于只有从其内部缓冲区到输出流的传输步骤是同步的，因此每个线程都需要自己的 `std::osyncstream` 实例。

当流销毁时，即明确调用 `emit()` 时，内部缓冲区会转移。

以下是每条打印线同步的简单解决方案：

```
1  #include <iostream>
2  #include <syncstream>
3  #include <thread>
4
5  #define sync_cout std::osyncstream(std::cout)
6
7  int main() {
8      std::thread t1([]() {
9          for (int i = 0; i < 100; ++i) {
10             sync_cout << "1 " << "2 " << "3 " << "4 "
11             << std::endl;
12         }
13     });
14
15     std::thread t2([]() {
16         for (int i = 0; i < 100; ++i) {
17             sync_cout << "5 " << "6 " << "7 " << "8 "
18             << std::endl;
19         }
20     });
```

```
21
22      t1.join();
23      t2.join();
24
25      return 0;
26  }
```

输出为：

```
1 2 3 4
1 2 3 4
5 6 7 8
```

由于这种方法现在是输出内容时避免竞争条件的官方 C++20 方法，我们将在本书的其余部分使用 std::osyncstream 作为默认方法。

3.3.3. 休眠当前线程

std::this_thread 是一个命名空间，允许从当前线程访问函数，以将执行交给另一个线程或阻止当前任务的执行并等待一段时间。

std::this_thread::sleep_for 和 std::this_thread::sleep_until 函数会在给定的时间内阻止线程的执行。

std::this_thread::sleep_for 至少休眠一段给定的时间。阻塞时间可能会更长，具体取决于操作系统调度程序如何决定运行任务，或者由于某些资源争用延迟。

资源争用

当对某种共享资源的需求超过供应时，就会发生资源争用，从而导致性能下降。

std::this_thread::sleep_until 的工作方式与 std::this_thread::sleep_for 类似。但不是休眠一段时间，而是休眠到特定时间点。计算时间点的时钟必须满足时钟要求（可以在此处找到有关此要求的更多信息：https://en.cppreference.com/w/cpp/named_req/Clock）。标准建议使用稳定时钟，而非系统时钟来设置持续时间。

3.3.4. 识别线程

在调试多线程解决方案时，了解哪个线程正在执行给定函数很有用。每个线程都可以通过标识符来标识，可以记录其值以进行跟踪和调试。

std::thread::id 是一个轻量级类，定义了线程对象（std::thread 和 std::jthread）的唯一标识符。此标识符可通过 get_id() 函数检索。

线程标识符对象可以通过输出流进行比较、序列化和打印。还可以用作映射容器中的键，其支持 std::hash 函数。

以下示例打印 t 线程的标识符。本章后面，将介绍如何创建线程并休眠一段时间：

```cpp
#include <chrono>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

void func() {
    std::this_thread::sleep_for(1s);
}

int main() {
    std::thread t(func);
    std::cout << "Thread ID: " << t.get_id() << std::endl;
    t.join();
    return 0;
}
```

当一个线程完成时，其标识符可以在未来重用。

3.3.5. 传递参数

参数可以通过值、引用或指针传递给线程。

如何通过值传递参数：

```cpp
void funcByValue(const std::string& str, int val) {
    sync_cout << "str: " << str << ", val: " << val
    << std::endl;
}
std::string str{"Passing by value"};
std::thread t(funcByValue, str, 1);
```

按值传递可以避免数据争用。但由于数据需要复制，因此成本更高。

下一个示例显示如何通过引用传递值：

```cpp
void modifyValues(std::string& str, int& val) {
    str += " (Thread)";
    val++;
}
std::string str{"Passing by reference"};
int val = 1;
std::thread t(modifyValues, std::ref(str), std::ref(val));
```

或者作为 `const` 引用：

```cpp
void printVector(const std::vector<int>& v) {
    sync_cout << "Vector: ";
    for (int num : v) {
        sync_cout << num << " ";
    }
    sync_cout << std::endl;
}
std::vector<int> v{1, 2, 3, 4, 5};
std::thread t(printVector, std::cref(v));
```

通过使用 ref()（非常量引用）或 cref()（常量引用）来实现引用传递。两者都在 <functional> 头文件中定义，允许可变参数模板定义线程构造函数以将参数视为引用。

这些辅助函数用于生成 std::reference_wrapper 对象，该对象将引用包装在可复制和可赋值的对象中。传递参数时缺少这些函数会导致参数按值传递。

还可以按如下方式将对象移动到线程中：

```cpp
std::thread t(printVector, std::move(v));
```

但请注意，在将 v 移入 t 线程后，尝试在主线程中访问它会导致未定义行为。

最后，还可以通过 lambda 捕获来允许线程访问变量：

```cpp
std::string str{"Hello"};
std::thread t([&]() {
    sync_cout << "str: " << str << std::endl;
});
```

此示例中，str 变量会被 t 线程作为嵌入的 lambda 函数捕获的引用来访问。

3.3.6. 返回值

要返回在线程中计算的值，可以使用具有同步机制的共享变量，例如：互斥锁、锁或原子变量。

下面的代码段中，可以看到如何使用非常量引用传递的参数（使用 ref()）返回由线程计算的值。结果变量在 func 函数中的 t 线程内计算。结果值可以从主线程中看到。join() 函数只是等待 t 线程完成，然后让主线程继续运行，然后检查结果变量：

```cpp
#include <chrono>
#include <iostream>
#include <random>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;
```

```

namespace {
    int result = 0;
};

void func(int& result) {
    std::this_thread::sleep_for(1s);
    result = 1 + (rand () % 10);
}

int main() {
    std::thread t(func, std::ref(result));
    t.join();
    sync_cout << "Result: " << result << std::endl;
}
```

引用参数可以是对输入对象本身的引用，也可以对存储结果的另一个变量的引用。

还可以使用 lambda 捕获返回值：

```
std::thread t([&]() { func(result); });
t.join();
sync_cout << "Result: " << result << std::endl;
```

还可以通过写入受互斥锁保护的共享变量来实现，在执行写入操作之前锁定互斥锁（例如，使用 std::lock_guard）：

```
#include <chrono>
#include <iostream>
#include <mutex>
#include <random>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

namespace {
    int result = 0;
    std::mutex mtx;
};

void funcWithMutex() {
    std::this_thread::sleep_for(1s);
    int localVar = 1 + (rand() % 10);
    std::lock_guard<std::mutex> lock(mtx);
    result = localVar;
```

```
}

int main() {
    std::thread t(funcWithMutex);
    t.join();
    sync_cout << "Result: " << result << std::endl;
}
```

还有一种更优雅的方法可以从线程返回值。这涉及使用 Future 和 Promise，将在第 6 章中介绍它们。

3.3.7. 移动线程

线程可以移动但不能复制，为了避免有两个不同的线程对象代表同一个硬件线程。

以下示例中，使用 std::move 将 t1 移动到 t2。因此，t2 继承了与移动前的 t1 相同的标识符，并且 t1 不可汇入：

```
#include <chrono>
#include <thread>

using namespace std::chrono_literals;

void func() {
    for (auto i=0; i<10; ++i) {
        std::this_thread::sleep_for(500ms);
    }
}

int main() {
    std::thread t1(func);
    std::thread t2 = std::move(t1);
    t2.join();
    return 0;
}
```

当将 std::thread 对象移动到另一个 std::thread 对象时，移动自线程对象将达到不再代表真实线程的状态。这种情况也会发生在默认构造函数分离或汇入后产生的线程对象上。

3.3.8. 等待线程完成

某些情况下，一个线程需要等待另一个线程完成，以便可以使用后者计算的结果。其他用例包括在后台运行一个线程将其分离，然后继续执行主线程。

汇入一个线程

join() 函数阻塞当前线程，等待调用 join() 函数的线程对象所标识的汇入线程完成。这

确保汇入线程在 join() 返回后终止（有关更多详细信息，请参阅第 2 章中的线程生命周期部分）。

很容易忘记使用 join() 函数，汇入线程（jthread）解决了这个问题。它从 C++20 开始可用，我们将在下一节中介绍它。

检查线程是否可以汇入

如果线程中尚未调用 join() 函数，则该线程视为可汇入且因此处于活动状态。即使线程已完成代码执行但仍未汇入，情况也是如此。另一方面，默认构造的线程或已汇入的线程不可汇入。

要检查一个线程是否可连接，只需使用 std::thread::joinable() 函数。

以下示例中看一下 std::thread::join() 和 std::thread::joinable() 的用法：

```
#include <chrono>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

void func() {
    std::this_thread::sleep_for(100ms);
}

int main() {
    std::thread t1;
    std::cout << "Is t1 joinable? " << t1.joinable()
              << std::endl;

    std::thread t2(func);
    t1.swap(t2);
    std::cout << "Is t1 joinable? " << t1.joinable()
              << std::endl;
    std::cout << "Is t2 joinable? " << t2.joinable()
              << std::endl;

    t1.join();
    std::cout << "Is t1 joinable? " << t1.joinable()
              << std::endl;
}
```

使用默认构造函数（未指定可调用对象）构造 t1 后，线程将不可汇入。由于 t2 是指定函数构造的，因此构造后 t2 可汇入。但当 t1 和 t2 交换时，t1 再次变为可汇入，而 t2 不再可汇入。然后主线程等待 t1 汇入，因此它不再可汇入。尝试汇入不可汇入的线程 t2 会导致未定义行为。最后，不汇入可汇入的线程将导致资源泄漏，或由于意外使用共享资源而导致程序崩溃。

分离守护线程

如果希望线程继续作为守护线程在后台运行，但完成当前线程的执行，可以使用 std::thread::detach()。守护线程是在后台执行一些不需要运行完成的任务的线程。如果主

程序退出，所有守护线程都将终止。如前所述，线程必须在主线程终止之前汇入或分离，否则程序将中止其执行。

调用 detach 之后，分离的线程无法使用 std::thread 对象进行控制或汇入（因为它正在等待完成），因为其不再代表分离的线程。

以下示例显示了一个名为 t 的守护线程,在构造后立即分离,并在后台运行 daemonThread( ) 函数。此函数执行三秒钟，然后退出，完成线程执行。同时，主线程在退出之前比线程执行时间多休眠一秒钟：

```
#include <chrono>
#include <iostream>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

namespace {
    int timeout = 3;
}

void daemonThread() {
    sync_cout << "Daemon thread starting...\n";
    while (timeout-- > 0) {
        sync_cout << "Daemon thread is running...\n";
        std::this_thread::sleep_for(1s);
    }
    sync_cout << "Daemon thread exiting...\n";
}

int main() {
    std::thread t(daemonThread);
    t.detach();

    std::this_thread::sleep_for(
        std::chrono::seconds(timeout + 1));

    sync_cout << "Main thread exiting...\n";
    Return 0;
}
```

3.3.9. 汇入线程——jthread 类

从 C++20 开始，有一个新的类：std::jthread。此类与 std:thread 类似，但具有线程在销毁时重新汇入的功能，遵循资源获取即初始化（RAII）技术。在某些情况下，可以取消或停止。

从下面的例子中可以看到，jthread 线程具有与 std::thread 相同的接口。唯一的区别是不需要调用 join() 函数来确保主线程等待 t 线程汇入：

```cpp
#include <chrono>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

void func() {
    std::this_thread::sleep_for(1s);
}

int main() {
    std::jthread t(func);
    sync_cout << "Thread ID: " << t.get_id() << std::endl;
    return 0;
}
```

当两个 std::jthread 汇入销毁时，其析构函数将以与构造函数相反的顺序调用。为了演示此行为，让实现一个线程包装器类，该类在创建和销毁包装线程时打印一些消息：

```cpp
#include <chrono>
#include <functional>
#include <iostream>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

class JthreadWrapper {
public:
    JthreadWrapper(
    const std::function<void(const std::string&)>& func,
    const std::string& str)
    : t(func, str), name(str) {
        sync_cout << "Thread " << name
                  << " being created" << std::endl;
    }
    ~JthreadWrapper() {
        sync_cout << "Thread " << name
                  << " being destroyed" << std::endl;
    }
private:
    std::jthread t;
```

```
    std::string name;
};
```

使用这个 JthreadWrapper 包装器类，启动三个执行 func 函数的线程。每个线程将等待一秒钟后退出：

```
void func(const std::string& name) {
    sync_cout << "Thread " << name << " starting...\n";
    std::this_thread::sleep_for(1s);
    sync_cout << "Thread " << name << " finishing...\n";
}

int main() {
    JthreadWrapper t1(func, "t1");
    JthreadWrapper t2(func, "t2");
    JthreadWrapper t3(func, "t3");
    std::this_thread::sleep_for(2s);
    sync_cout << "Main thread exiting..." << std::endl;
    return 0;
}
```

该程序将显示以下输出：

```
Thread t1 being created
Thread t1 starting...
Thread t2 being created
Thread t2 starting...
Thread t3 being created
Thread t3 starting...
Thread t1 finishing...
Thread t2 finishing...
Thread t3 finishing...
Main thread exiting...
Thread t3 being destroyed
Thread t2 being destroyed
Thread t1 being destroyed
```

可以看到，首先创建 t1，然后创建 t2，最后创建 t3。析构函数遵循相反的顺序，首先销毁 t3，然后销毁 t2，最后销毁 t1。

由于 jthread 可以避免忘记在线程中使用 join 时出现的问题，因此我们更倾向使用 std:: jthread，而非 std::thread。某些情况下，需要使用对 join( ) 的显式调用，来确保在转到另一项任务之前线程已汇入，并且资源已正确释放。

3.3.10. 丢弃执行线程

线程还可以决定暂停执行，让实现重新安排线程的执行，并给予其他线程运行的机会。

`std::this_thread::yield` 方法向操作系统提供重新安排另一个线程的提示。此行为与实现有关，取决于操作系统调度程序和系统的当前状态。

一些 Linux 实现会暂停当前线程并将其移回线程队列，以调度所有具有相同优先级的线程。如果此队列为空，则让出无效。

下面的例子展示了两个线程 `t1` 和 `t2` 执行相同的工作函数。它们随机选择执行某些工作（锁定互斥锁，将在下一章中学习，并等待三秒钟）或将执行权交给另一个线程：

```cpp
#include <iostream>
#include <random>
#include <string>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono;

namespace {
    int val = 0;
    std::mutex mtx;
}

int main() {
    auto work = [&](const std::string& name) {
        while (true) {
            bool work_to_do = rand() % 2;
            if (work_to_do) {
                sync_cout << name << ": working\n";
                std::lock_guard<std::mutex> lock(mtx);
                for (auto start = steady_clock::now(),
                          now = start;
                          now < start + 3s;
                          now = steady_clock::now()) {
                }
            } else {
                sync_cout << name << ": yielding\n";
                std::this_thread::yield();
            }
        }
    };

    std::jthread t1(work, "t1");
    std::jthread t2(work, "t2");

    return 0;
}
```

运行此示例时，当执行到达 yield 命令时，可以看到当前正在运行的线程如何停止，并使另一个线程重新开始执行。

3.3.11. 线程取消

如果不再对线程正在计算的结果感兴趣，就会取消该线程并避免更多的计算成本。

终止线程可能是一种解决方案，但这会导致属于该线程处理的资源（例如：从该线程启动的其他线程、锁、汇入等）无法处理。这可能意味着以未定义的行为、在互斥锁下锁定的关键部分或其他意外问题结束程序。

为了避免这些问题，需要一个无数据争用机制，让线程知道停止执行的意图（请求停止），以便线程可以采取取消其工作并正常终止所需的所有具体步骤。

实现此目的的一种可能方法是使用由线程定期检查的原子变量。现在，将原子变量定义为一个变量，由于其原子事务操作和内存模型，许多线程可以写入或读取该变量，而无需任何锁定机制或数据争用。

举个例子，创建一个 Counter 类，每秒调用一次回调。这将无限执行，直到调用者使用 stop() 函数时，运行的原子变量设置为 false:

```
#include <chrono>
#include <functional>
#include <iostream>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

class Counter {
    using Callback = std::function<void(void)>;
public:
    Counter(const Callback &callback) {
        t = std::jthread([&]() {
            while (running.load() == true) {
                callback ();
                std::this_thread::sleep_for(1s);
            }
        });
    }
    void stop() { running.store(false); }
private:
    std::jthread t;
    std::atomic_bool running{true};
};
```

调用函数中，将按如下方式实例化 Counter。然后，在需要时（这里是三秒后）调用 stop()

函数，让 Counter 退出循环并终止线程执行：

```
int main() {
    Counter counter([&]() {
        sync_cout << "Callback: Running...\n";
    });
    std::this_thread::sleep_for(3s);
    counter.stop();
}
```

自 C++20 以来，出现了一种称为线程协作中断的新机制，可通过 std::stop_token 获得。线程通过检查调用 std::stop_token::stop_requested() 函数的结果知晓请求停止。

为了生成 stop_token，将通过 std::stop_source::get_token() 函数使用 stop_source 对象。

此线程取消机制在 std::jthead 对象中通过 std::stop_source 类型的内部成员实现，其中存储了共享的停止状态。jthread 构造函数接受 std::stop_token 作为其第一个参数。当在执行期间请求停止时，可使用此方法。

因此，std::jthread 相对于 std::thread 对象来说，暴露了一些函数来管理停止标记，这些函数分别是 get_stop_source()、get_stop_token() 和 request_stop()。

当调用 request_stop() 时，会向内部停止状态发出停止请求，该请求会原子地更新以避免竞争条件。

让在下面的例子中检查所有这些功能是如何工作的。

首先，定义一个模板函数来展示停止项对象（stop_token 或 stop_source）的属性：

```
#include <chrono>
#include <iostream>
#include <string_view>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

template <typename T>
void show_stop_props(std::string_view name,
                     const T& stop_item) {
    sync_cout << std::boolalpha
              << name
              << ": stop_possible = "
              << stop_item.stop_possible()
              << ", stop_requested = "
              << stop_item.stop_requested()
              << '\n';
};
```

main() 函数中，将启动一个工作线程，获取其停止令牌对象，并显示其属性：

```
auto worker1 = std::jthread(func_with_stop_token);
std::stop_token stop_token = worker1.get_stop_token();
show_stop_props("stop_token", stop_token);
```

Worker1 正在运行在随后的代码块中定义的 func_with_stop_token() 函数。此函数中，使用stop_requested() 函数检查停止令牌。如果此函数返回 true，则表示已请求停止，因此该函数将直接返回，终止线程执行。否则，将运行下一个循环迭代，使当前线程再休眠 300 毫秒，直到下一次停止请求检查：

```
void func_with_stop_token(std::stop_token stop_token) {
    for (int i = 0; i < 10; ++i) {
        std::this_thread::sleep_for(300ms);
        if (stop_token.stop_requested()) {
            sync_cout << "stop_worker: "
            << "Stopping as requested\n";
            return;
        }
        sync_cout << "stop_worker: Going back to sleep\n";
    }
}
```

可以使用线程对象返回的停止令牌，并向主线程请求停止：

```
worker1.request_stop();
worker1.join();
show_stop_props("stop_token after request", stop_token);
```

另外，可以从不同的线程请求停止。为此，需要传递一个 stop_source 对象。下面的代码中，可以看到如何使用从 worker2 工作线程获取的 stop_source 对象作为参数创建线程停止器：

```
auto worker2 = std::jthread(func_with_stop_token);
std::stop_source stop_source = worker2.get_stop_source();
show_stop_props("stop_source", stop_source);

auto stopper = std::thread( [](std::stop_source source) {
        std::this_thread::sleep_for(500ms);
        sync_cout << "Request stop for worker2 "
                  << "via source\n";
        source.request_stop();
    }, stop_source);

stopper.join();
std::this_thread::sleep_for(200ms);
show_stop_props("stop_source after request", stop_source);
```

stopper 线程等待 0.5 秒，然后向 stop_source 对象请求停止。然后，worker2 会获知该请求并终止其执行，如前所述。

还可以注册一个回调函数，当通过停止令牌或停止源请求停止时，该函数将调用一个函数。这可以通过使用 std::stop_callback 对象来完成：

```
std::stop_callback callback(worker1.get_stop_token(), []{
    sync_cout << "stop_callback for worker1 "
              << "executed by thread "
              << std::this_thread::get_id() << '\n';
});

sync_cout << "main_thread: "
          << std::this_thread::get_id() << '\n';
std::stop_callback callback_after_stop(
    worker2.get_stop_token(),[] {
        sync_cout << "stop_callback for worker2 "
                  << "executed by thread "
                  << std::this_thread::get_id() << '\n';
});
```

如果销毁 std::stop_callback 对象，则将阻止其执行。例如，回调对象在超出范围时被销毁，所以这个作用域停止回调将不会执行：

```
{
    std::stop_callback scoped_callback(
    worker2.get_stop_token(), []{
        sync_cout << "Scoped stop callback "
                  << "will not execute\n";
    }
    );
}
```

请求停止后，将立即执行新的停止回调对象。以下示例中，如果已请求停止 worker2，callback_after_stop 将在构造后立即执行 lambda 函数：

```
sync_cout << "main_thread: "
          << std::this_thread::get_id() << '\n';
std::stop_callback callback_after_stop(
    worker2.get_stop_token(), []{
        sync_cout << "stop_callback for worker2 "
                  << "executed by thread "
                  << std::this_thread::get_id() << '\n';
    }
);
```

3.3.12. 捕获异常

线程中抛出的未处理异常都需要在该线程中捕获。否则，C++ 运行时会调用 `std::terminate`，导致程序突然终止。这会导致意外行为、数据丢失，甚至程序崩溃。

一种解决方案是在线程中使用 `try-catch` 块来捕获异常。但只会捕获该线程内引发的异常，异常不会传播到其他线程。

要将异常传播到另一个线程，一个线程可以捕获它，并将其存储到 `std::exception_ptr` 对象中，然后使用共享内存技术将其传递给另一个线程，在那里将检查 `std::exception_ptr` 对象并在需要时重新抛出异常。

以下示例展示了这种方法：

```cpp
#include <atomic>
#include <chrono>
#include <exception>
#include <iostream>
#include <mutex>
#include <thread>

using namespace std::chrono_literals;

std::exception_ptr captured_exception;
std::mutex mtx;

void func() {
    try {
        std::this_thread::sleep_for(1s);
        throw std::runtime_error(
        "Error in func used within thread");
    } catch (...) {
        std::lock_guard<std::mutex> lock(mtx);
        captured_exception = std::current_exception();
    }
}

int main() {
    std::thread t(func);
    while (!captured_exception) {
        std::this_thread::sleep_for(250ms);
        std::cout << "In main thread\n";
    }
    try {
        std::rethrow_exception(captured_exception);
    } catch (const std::exception& e) {
        std::cerr << "Exception caught in main thread: "
        << e.what() << std::endl;
    }
```

```
    t.join();
}
```

可以看到，在 t 线程执行 func 函数时，抛出了 std::runtime_error 异常。该异常捕获并存储在 caught_exception 中，这是一个受互斥锁保护的 std::exception_ptr 共享对象。通过调用 std::current_exception() 函数可以确定抛出的异常的类型和值。

主线程中，while 循环运行直到捕获到异常。通过调用 std::rethrow_exception(captured_exception)，在主线程中重新抛出异常。主线程再次捕获该异常，并在执行 catch 块时通过 std::cerr 错误流将消息打印到控制台。

3.4. 线程本地存储

线程本地存储（TLS）是一种内存管理技术，允许每个线程拥有私有变量实例。因为此技术消除了访问这些变量的同步机制开销，允许线程存储其他线程无法访问的线程特定数据，从而避免竞争条件并提高性能。

TLS 由操作系统实现，可使用自 C++11 起可用的 thread_local 关键字访问。thread_local 提供了一种统一的方法来使用许多操作系统的 TLS 功能，并避免使用特定于编译器的语言扩展来访问 TLS 功能（此类扩展的一些示例是 TLS Windows API、__declspec(thread) MSVC 编译器语言扩展或 __thread GCC 编译器语言扩展）。

要将 TLS 与不支持 C++11 或更新版本的编译器一起使用，请使用 Boost.Library。它提供了boost::thread_specific_ptr 容器，该容器实现了可移植的 TLS。

线程局部变量可以声明如下：

- 全局
- 在命名空间中
- 作为类静态成员变量
- 在函数内部；与使用 static 关键字分配的变量具有相同的效果，变量在程序的整个生命周期内分配，并且其值会在下一次函数调用中保留

以下示例显示三个线程使用不同的参数调用 multiplyByTwo 函数。此函数将 val 线程局部变量的值设置为参数值，将其乘以 2，然后输出到控制台：

```
#include <iostream>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

thread_local int val = 0;

void setValue(int newval) { val = newval; }

void printValue() { sync_cout << val << ' '; }
```

```
void multiplyByTwo(int arg) {
    setValue(arg);
    val *= 2;
    printValue();
}

int main() {
    val = 1; // Value in main thread
    std::thread t1(multiplyByTwo, 1);
    std::thread t2(multiplyByTwo, 2);
    std::thread t3(multiplyByTwo, 3);
    t1.join();
    t2.join();
    t3.join();
    std::cout << val << std::endl;
}
```

运行此代码片将显示以下输出：

```
2 4 6 1
```

这里，可以看到每个线程都对其输入参数进行操作，导致 t1 打印 2、t2 输出 4 并且 t3 打印 6。运行主函数的主线程还可以访问其线程局部变量 val，该变量的值在程序启动时设置为 1，但仅在退出程序之前在主函数末尾输出到控制台时使用。

TLS 也存在一些缺点。TLS 会增加内存使用量，每个线程都会创建一个变量，在资源受限的环境中可能会出现问题。与常规变量相比，访问 TLS 变量可能会产生一些开销。这对于性能至关重要的软件来说可能有风险。

接下来，利用迄今为止学到的诸多技术，来构建一个计时器。

3.5. 实现计时器

实现一个接受间隔和回调函数的计时器。计时器将在每个间隔执行回调函数，用户可以通过调用其 stop() 函数来停止计时器。

以下代码片段展示了计时器的实现

```
#include <chrono>
#include <functional>
#include <iostream>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;
using namespace std::chrono;
```

```
11
12  template<typename Duration>
13  class Timer {
14      public:
15      typedef std::function<void(void)> Callback;
16      Timer(const Duration interval,
17            const Callback& callback) {
18          auto value = duration_cast<milliseconds>(interval);
19          sync_cout << "Timer: Starting with interval of "
20                    << value << std::endl;
21          t = std::jthread([&](std::stop_token stop_token) {
22              while (!stop_token.stop_requested()) {
23                  sync_cout << "Timer: Running callback "
24                            << val.load() << std::endl;
25                  val++;
26                  callback();
27                  sync_cout << "Timer: Sleeping...\n";
28                  std::this_thread::sleep_for(interval);
29              }
30              sync_cout << "Timer: Exit\n";
31          });
32      }
33      void stop() {
34          t.request_stop();
35      }
36      private:
37      std::jthread t;
38      std::atomic_int32_t val{0};
39  };
```

Timer 构造函数接受一个回调函数（一个 std::function<void(void)> 对象）和一个 std::chrono: :duration 对象，定义执行回调的时间段或间隔。

使用 lambda 表达式创建一个 std::jthread 对象，循环以时间间隔调用回调。此循环检查是否已通过 stop_token 请求停止，该请求可通过 stop( ) Timer API 函数启用。如果是，则循环退出，线程终止。

使用方法如下：

```
1  int main(void) {
2      sync_cout << "Main: Create timer\n";
3      Timer timer(1s, [&]() {
4          sync_cout << "Callback: Running...\n";
5      });
6
7      std::this_thread::sleep_for(3s);
8      sync_cout << "Main thread: Stop timer\n";
9      timer.stop();
```

```
10
11      std::this_thread::sleep_for(500ms);
12      sync_cout << "Main thread: Exit\n";
13      return 0;
14  }
```

此示例中，启动了计时器，该计时器每秒将打印一次 Callback: Running 消息。三秒后，主线程将调用 timer.stop() 函数，终止计时器线程，主线程等待 500 毫秒后退出。

这是输出：

```
Main: Create timer
Timer: Starting with interval of 1000ms
Timer: Running callback 0
Callback: Running...
Timer: Sleeping...
Timer: Running callback 1
Callback: Running...
Timer: Sleeping...
Timer: Running callback 2
Callback: Running...
Timer: Sleeping...
Main thread: Stop timer
Timer: Exit
Main thread: Exit
```

作为练习，可以稍微修改此示例以实现超时类，如果在给定的超时间隔内没有输入事件，则该类会调用回调函数。这是处理网络通信时的一种常见模式，如果一段时间内没有收到数据包，则会发送数据包重放请求。

3.6. 总结

本章中，介绍了如何创建和管理线程、如何传递参数或检索结果、TLS 的工作原理以及如何等待线程完成。还介绍了如何让线程将控制权让给其他线程或取消其执行。如果出现问题并引发异常，了解如何在线程之间传递异常并避免程序意外终止。最后，实现了一个定期运行回调函数的计时器类。

下一章中，将介绍线程安全、互斥和原子操作。其中包括互斥、锁定和无锁算法以及内存同步排序等主题。这些知识将有助于开发线程安全的数据结构和算法。

3.7. 扩展阅读

- 编译器支持情况： https://en.cppreference.com/w/cpp/compiler_support
- GCC 发布版本： https://gcc.gnu.org/releases.html
- Clang： https://clang.llvm.org

- Clang 8 文档：https://releases.llvm.org/8.0.0/tools/clang/docs/index.html
- LLVM 项目：https://llvm.org
- Boost.Threads：https://www.boost.org/doc/libs/1_78_0/doc/html/thread.html
- P0024 –并行性技术规范：https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0024r0.html
- TLS 提案：https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2659.htm
- C++0X 的线程启动：https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2184.html
- IBM 的 TLS：https://docs.oracle.com/cd/E19683-01/817-3677/chapter8-1/index.html
- 线程私有的数据：https://www.ibm.com/docs/en/i/7.5?topic=techniques-data-that-is-private-thread
- 资源获取即初始化（RAII）：https://en.cppreference.com/w/cpp/language/raii
- Bjarne Stroustrup，《C++ 之旅》，第三版，18.2 和 18.7

第 4 章 使用锁进行线程同步

第 2 章中，介绍了线程可以读取和写入它们所属进程共享的内存。操作系统实现了进程内存访问保护，但对于访问同一进程中共享内存的线程，并没有这样的保护。多个线程对同一内存地址的并发内存写入操作，需要同步机制来避免数据争用并确保数据完整性。

本章中，将详细介绍多个线程并发访问共享内存，所产生的问题以及如何解决这些问题。我们将详细介绍以下主题：

- 条件竞争–是什么以及如何发生
- 互斥作为一种同步机制，如何通过 `std::mutex` 在 C++ 中实现
- 通用的锁管理
- 什么是条件变量，以及如何将它们与互斥锁一起使用
- 使用 `std::mutex` 和 `std::condition_variable`实现完全同步队列
- C++20 引入的新同步原语 - 信号量、栅栏和门闩

这些都是基于锁的同步机制。无锁技术是下一章的主题。

4.1. 技术要求

本章的技术要求与上一章中解释的概念相同，要编译和运行示例，需要支持 C++20 的 C++ 编译器（用于信号量、门闩和栅栏示例），大多数示例只需要 C++11。示例已在 Linux Ubuntu LTS 24.04 上进行了测试。

本章的代码可以在 GitHub 上找到：https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP

4.2. 了解条件竞争

当程序运行的结果取决于其指令的执行顺序时，就会发生条件竞争。举一个简单的例子，展示条件竞争是如何发生的，然后了解如何解决这个问题。

下面的代码中，计数器全局变量由两个同时运行的线程增加：

```
#include <iostream>
#include <thread>

int counter = 0;

int main() {
    auto func = [] {
        for (int i = 0; i < 1000000; ++i) {
            counter++;
        }
    };

```

```
    std::thread t1(func);
    std::thread t2(func);

    t1.join();
    t2.join();

    std::cout << counter << std::endl;
    return 0;
}
```

运行上述代码三次后，我们得到以下计数器值：

```
1056205
1217311
1167474
```

这里看到两个问题：首先，计数器的值不正确；其次，程序每次执行都会以不同的计数器值结束。结果不确定，而且大多数情况下都不正确。

这个场景涉及两个线程 `t1` 和 `t2`，同时运行并修改同一个变量，该变量本质上是某个内存区域。这应该可以正常工作，因为只有一行代码会增加计数器值，从而修改内存内容（顺便说一句，使用像 `counter++` 中的后增量运算符或像 `++counter` 中的前增量运算符都没有关系；结果同样是错误的）。

仔细查看上面的代码，仔细研究以下行：

```
counter++;
```

分三步增加计数器：

- 计数器变量存储的内存地址的内容被加载到 CPU 寄存器中。本例中，`int` 数据类型从内存加载到 CPU 寄存器中。
- 寄存器中的值增加一。
- 寄存器中的值保存在计数器变量的内存地址中。

现在，考虑两个线程尝试同时增加计数器的可能情况，来看看表 4.1：

线程 1	线程 2
[1] 将计数器值装入寄存器	[3] 将计数器值装入寄存器
[2] 增加寄存器的值	[5] 增加寄存器的值
[4] 寄存器计数器	[6] 寄存器计数器

表 4.1：两个线程同时增加计数器

线程 1 执行 [1] 并将计数器的当前值（假设为 1）加载到 CPU 寄存器中。然后，它将寄存器中的值加 1 [2]（现在，寄存器值为 2）。

线程 2 被安排执行，并且 [3] 将计数器的当前值（记住 - 它还没有被修改，所以它仍然为 1 ）加载到 CPU 寄存器中。

现在，线程 1 再次执行，并且 [4] 将更新的值存储到内存中。计数器的值现在等于 2 。最后，线程 2 再次执行 [5] 和 [6]。寄存器值增加 1，然后将 2 存储在内存中。计数器变量只增加了一次，而它应该增加两次，其值应该是三。

发生上述问题是，计数器上的递增操作不是原子的。如果每个线程都可以在不中断的情况下，执行递增计数器变量所需的三条指令，则计数器将按预期递增两次。但根据操作执行的顺序，结果可能会有所不同，这称为条件竞争。

为了避免条件竞争，需要确保以受控的方式访问和修改共享资源。实现此目的的一种方法是使用锁。锁是一种同步原语，一次只允许一个线程访问共享资源。当线程想要访问共享资源时，必须首先获取锁。线程获取了锁后，就可以访问共享资源而不会受到其他线程的干扰。当线程访问完共享资源后，必须释放锁，以便其他线程可以访问相应的共享资源。

避免条件竞争的另一种方法是使用原子操作，可保证在单个不可分割的步骤中执行的操作。在执行原子操作时，没有其他线程可以干扰该操作。原子操作通常使用设计为不可分割的硬件指令来实现。

本节中，介绍了多线程代码产生的最常见和最重要的问题：条件竞争。根据执行操作的顺序，结果可能会有所不同。考虑到这个问题，将在下一节中研究如何解决它。

4.3. 为什么需要互斥?

互斥是并发编程中的一个基本概念，可确保多个线程或进程不会同时访问共享资源（例如：共享变量、代码的关键部分或文件或网络连接）。互斥对于避免条件竞争至关重要。

想象一下，一家小咖啡店只有一台浓缩咖啡机，这台机器一次只能制作一杯浓缩咖啡，所以这台机器是所有咖啡师必须共享的关键资源。

咖啡店有三位咖啡师：爱丽丝、鲍勃和卡罗尔。他们同时使用咖啡机，但不能同时使用：鲍勃将适量的现磨咖啡放入机器中，开始制作浓缩咖啡。然后，爱丽丝做了同样的事情，但先从机器中取出咖啡，以为鲍勃只是忘了做这件事。然后鲍勃从机器中取出浓缩咖啡，之后，爱丽丝发现没有浓缩咖啡了！这是一场灾难——如同之前的计数器程序一样。

为了解决咖啡店的问题，他们任命卡罗尔为机器管理员。在使用机器之前，爱丽丝和鲍勃都会询问她是否可以开始制作新的浓缩咖啡。这样就可以解决问题。

回到计数器程序，如果可以一次只允许一个线程访问计数器（卡罗尔在咖啡店里所做的），软件问题也将得到解决。互斥是一种可用于控制并发线程访问内存的机制。C++ 标准库提供了 `std::mutex` 类，这是一个同步原语，用于保护共享数据不被两个或多个线程同时访问。

上一节中，看到的这个新版本的代码实现了两种并发增加计数器的方式：如上一节所述的自由访问，以及使用互斥的同步访问：

```
#include <iostream>
#include <mutex>
#include <thread>

```

```cpp
std::mutex mtx;
int counter = 0;

int main() {
    auto funcWithoutLocks = [] {
        for (int i = 0; i < 1000000; ++i) {
            ++counter;
        };
    };

    auto funcWithLocks = [] {
        for (int i = 0; i < 1000000; ++i) {
            mtx.lock();
            ++counter;
            mtx.unlock();
        };
    };

    {
        counter = 0;
        std::thread t1(funcWithoutLocks);
        std::thread t2(funcWithoutLocks);

        t1.join();
        t2.join();

        std::cout << "Counter without using locks: " << counter <<
        std::endl;
    } {
        counter = 0;
        std::thread t1(funcWithLocks);
        std::thread t2(funcWithLocks);

        t1.join();
        t2.join();

        std::cout << "Counter using locks: " << counter << std::endl;
    }
    return 0;
}
```

当线程运行 funcWithLocks 时，会在增加计数器之前使用 mtx.lock() 获取锁。当计数器增加，线程就会释放锁（mtx.unlock()）。

该锁只能由一个线程拥有。例如，t1 获取了锁，然后 t2 也尝试获取该锁，则 t2 将阻塞等待，直到锁可用。由于任何时候都只有一个线程可以拥有锁，因此此同步原语称为互斥（来自互斥）。运行此程序几次，将始终得到正确的结果：2000000。

本节中，介绍了互斥的概念，并了解到 C++ 标准库提供了 std::mutex 类作为线程同步的

原语。在下一节中，将详细介绍 std::mutex。

4.3.1. C++ 标准库互斥实现

上一节中，介绍了互斥和互斥锁的概念，以及为什么需要它们来同步并发内存访问。本节中，将了解 C++ 标准库提供的用于实现互斥的类。还将了解 C++ 标准库提供的一些辅助类，它们使互斥锁的使用更加容易。

下表总结了 C++ 标准库提供的互斥锁类及其主要功能：

互斥类型	可访问	可递归	可超时
std::mutex	独占	NO	NO
std::recursive_mutex	独占	YES	NO
std::shared_mutex	1 - 独占 N - 共享	NO	NO
std::timed_mutex	独占	NO	YES
std::recursive_timed_mutex	独占	YES	YES
std::shared_timed_mutex	1 - 独占 N - 共享	NO	YES

表 4.2: C++ 标准库中的互斥类

std::mutex

std::mutex 类是在 C++11 中引入的，是 C++ 标准库提供的最重要和最常用的同步原语之一。

std::mutex 是一个同步原语，可用于保护共享数据不被多个线程同时访问。std::mutex 类提供独占、非递归所有权语义。主要特点如下：

- 调用线程从成功调用 lock() 或 try_lock() 开始直到调用 unlock() 为止都拥有该互斥锁。
- 调用线程在调用 lock() 或 try_lock() 之前不得拥有互斥锁，这是 std::mutex 的非递归所有权语义属性。
- 当一个线程拥有一个互斥锁时，所有其他线程将阻塞（调用 lock() 时）或收到 false 返回值（调用 try_lock() 时）。这是 std::mutex 的独占所有权语义。

如果拥有互斥锁的线程尝试再次获取，则结果行为未定义。通常，发生这种情况时会引发异常，但这由实现定义。如果线程释放互斥锁后，尝试再次释放它，这也是未定义行为（如前一种情况）。当线程锁定互斥锁时，互斥锁销毁，或者线程在未释放锁的情况下终止，也会导致未定义行为。

std::mutex 类有三种方法：

- lock()：获取互斥锁。如果互斥锁已锁定，则调用线程将阻止，直到互斥锁解锁。从应用程序的角度来看，这就像调用线程等待互斥锁可用一样。

- try_lock()：调用此函数时，返回 true（表示互斥锁已成功锁定）或 false（表示互斥锁已锁定）。try_lock 是非阻塞的，调用线程可以获取互斥锁，也可以不获取，但不会像调用 lock() 时那样被阻塞。try_lock() 方法通常用于我们不希望线程等待互斥锁可用时。当我们希望线程继续进行某些处理并稍后尝试获取互斥锁时，将调用 try_lock()。
- unlock()：调用 unlock() 释放互斥锁。

std::recursive_mutex

std::mutex 类提供独占、非递归所有权语义。虽然独占所有权语义至少对于线程来说总是必需的（毕竟它是一种互斥机制），但在某些情况下，可能需要以递归方式获取互斥锁。例如，递归函数可能需要获取互斥锁。可能还需要在从另一个获取相同互斥锁的函数 f() 调用的函数 g() 中获取互斥锁。

std::recursive_mutex 类提供独占、递归语义。其主要特点如下：

- 调用线程可以多次获取同一个互斥锁。将拥有该互斥锁，直到释放该互斥锁的次数与获取该互斥锁的次数相同。例如，如果一个线程递归获取互斥锁三次，将拥有该互斥锁，直到它第三次释放该互斥锁。
- 递归互斥锁可以递归获取的最大次数未指定，因此由实现定义。一旦互斥锁的获取次数达到最大次数，对 lock() 的调用将抛出 std::system_error，而对 try_lock() 的调用将返回 false。
- 所有权与 std::mutex 相同：如果一个线程拥有一个 std::recursive_mutex 类，则其他线程在尝试通过调用 lock() 来获取它时都将阻塞，或者在调用 try_lock() 时将返回 false。

std::recursive_mutex 接口与 std::mutex 完全相同。

std::shared_mutex

std::mutex 和 std::shared_mutex 都具有独占所有权语义，并且只能有一个线程成为互斥锁所有者。但某些情况下，可能需要让多个线程同时访问受保护的数据，并只给一个线程独占访问权限。

计数器示例要求每个线程都拥有对单个变量的独占访问权限，如果有只需要读取计数器中当前值的线程，并且只有一个线程来增加其值，那么让读取线程同时访问计数器并给予写入线程独占访问权限会更好。

此功能使用“读者-写者锁”实现。C++ 标准库实现了 std::shared_mutex 类，具有类似（但不完全相同）的功能。

std::shared_mutex 与其他互斥类型的主要区别在于它有两个访问级别：

- 共享：多个线程可以共享同一互斥锁的所有权。通过 lock_shared()、try_lock_shared()/unlock_shared() 来获取/释放共享所有权。当至少一个线程已获取对锁的共享访问权限时，其他线程都无法获得对锁的独占访问权限，但可以获得共享访问权限。
- 独占：只有一个线程可以拥有互斥锁。通过调用 lock()、try_lock()/unlock() 来获取/释放独占所有权。当一个线程获得对锁的独占访问权限时，其他线程都无法获得对锁的共享或独占访问权限。

看一个使用 std::shared_mutex 的简单例子:

```cpp
#include <algorithm>
#include <chrono>
#include <iostream>
#include <shared_mutex>
#include <thread>

int counter = 0;

int main() {
    using namespace std::chrono_literals;

    std::shared_mutex mutex;

    auto reader = [&] {
        for (int i = 0; i < 10; ++i) {
            mutex.lock_shared();
            // Read the counter and do something
            mutex.unlock_shared();
        }
    };

    auto writer = [&] {
        for (int i = 0; i < 10; ++i) {
            mutex.lock();
            ++counter;
            std::cout << "Counter: " << counter << std::endl;
            mutex.unlock();
            std::this_thread::sleep_for(10ms);
        }
    };

    std::thread t1(reader);
    std::thread t2(reader);
    std::thread t3(writer);
    std::thread t4(reader);
    std::thread t5(reader);
    std::thread t6(writer);

    t1.join();
    t2.join();
    t3.join();
    t4.join();
    t5.join();
    t6.join();

    return 0;
```

```
}
```

该示例使用 std::shared_mutex 同步六个线程：两个线程是写入器，增加计数器的值并需要独占访问权限。其余四个线程只读取计数器，仅需要共享访问权限。请注意，为了使用 std::shared_mutex，需要包含 <shared_mutex> 头文件。

定时互斥类型

到目前为止，所看到的互斥类型在我们想要获取独占使用的锁时的行为方式相同：

- std::lock()：调用线程阻塞，直到锁可用
- std::try_lock()：如果锁不可用，则返回 false

对于 std::lock() 来说，调用线程可能会等待很长时间，如果线程无法获取锁，可能只需要等待一段时间，然后让线程继续进行一些处理。

为了实现这个目标，可以使用 C++ 标准库提供的定时互斥锁：std::timed_mutex、std::recursive_timed_mutex 和 std::shared_time_mutex。

它们与非定时对应物类似，并实现以下功能允许等待锁在特定时间段内可用：

- try_lock_for()：尝试锁定互斥锁并阻止线程，直到指定的持续时间结束（超时）。如果互斥锁在指定的持续时间之前被锁定，则返回 true；否则，返回 false。
  如果指定的持续时间小于或等于零（timeout_duration.zero()），则该函数的行为与 try_lock() 完全相同。
  由于调度或争用延迟，此函数的阻塞时间可能会超过指定的时间段。
- try_lock_until()：尝试锁定互斥锁，直到指定的超时时间或互斥锁锁定（以先到者为准）。这种情况下，可指定未来的一个时间点作为等待的限制。

下面的示例显示如何使用 std::try_lock_for()：

```cpp
#include <algorithm>
#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

constexpr int NUM_THREADS = 8;
int counter = 0;
int failed = 0;

int main() {
    using namespace std::chrono_literals;

    std::timed_mutex tm;
    std::mutex m;

    auto worker = [&] {
```

```
        for (int i = 0; i < 10; ++i) {
            if (tm.try_lock_for(10ms)) {
                ++counter;
                std::cout << "Counter: " << counter << std::endl;
                std::this_thread::sleep_for(10ms);
                m.unlock();
            }
            else {
                m.lock();
                ++failed;
                std::cout << "Thread " << std::this_thread::get_id()
                << " failed to lock" << std::endl;
                m.unlock();
            }
            std::this_thread::sleep_for(12ms);
        }
    };

    std::vector<std::thread> threads;
    for (int i = 0; i < NUM_THREADS; ++i) {
        threads.emplace_back(worker);
    }

    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Counter: " << counter << std::endl;
    std::cout << "Failed: " << failed << std::endl;

    return 0;
}
```

上述代码使用了两个锁：tm（定时互斥锁），用于在获取 tm 成功时同步对 counter 的访问和写入屏幕；m（非定时互斥锁），用于在获取 tm 不成功时同步对 failed 的访问和屏幕输出。

4.3.2. 使用锁时会遇到的问题

已经看到了仅使用互斥锁（锁）的示例。如果只需要一个互斥锁，并且正确地获取和释放，那么编写正确的多线程代码通常并不困难。当需要多个锁，代码复杂性就会增加。使用多个锁时的两个常见问题是死锁和活锁。

死锁

为了执行某项任务，一个线程需要访问两个资源，并且不能被两个或多个线程同时访问（需要互斥来正确同步对所需资源的访问）。每个资源都与不同的 std::mutex 类同步。这种情况下，线程必须获取第一个资源互斥锁，然后获取第二个资源互斥锁，最后处理资源并释放两个互斥锁。

当两个线程尝试执行上述处理时，可能会发生如下情况：线程 1 和线程 2 需要获取两个互斥锁才能执行所需的处理。线程 1 获取第一个互斥锁，线程 2 获取第二个互斥锁。然后，线程 1 将永远阻塞，等待第二个互斥锁可用，而线程 2 将永远阻塞，等待第一个互斥锁可用，这被称为死锁，也是多线程代码中最常见的问题。

活锁

解决死锁的一个可能方法如下：当一个线程尝试获取锁时，只会阻塞一段有限的时间，如果仍然不成功，将释放它可能已获取的锁。

例如，线程 1 获取了第一个锁，线程 2 获取了第二个锁。经过一段时间后，线程 1 仍未获取第二个锁，因此释放了第一个锁。线程 2 也可能完成等待并释放它获取的锁（在此示例中为第二个锁）。

这种解决方案有时可能有效，但并不正确。想象一下：线程 1 已获取第一个锁，并获取了第二个锁。一段时间后，两个线程都释放了已经获取的锁，然后再次获取相同的锁。然后，线程释放锁，然后再次获取锁，依此类推。

线程无法做任何事情，只能获取锁、等待、释放锁，然后重复上述操作。这种情况称为活锁，因为线程不会永远等待（如死锁情况），而是处于某种活动状态，并不断获取和释放锁。

解决死锁和活锁问题的最常见方法是以一致的顺序获取锁。例如，一个线程需要获取两个锁，将始终先获取第一个锁，然后再获取第二个锁。锁将以相反的顺序释放（先释放第二个锁，然后释放第一个锁）。如果第二个线程尝试获取第一个锁，将不得不等到第一个线程释放两个锁，这样就永远不会发生死锁。

本节中，了解了 C++ 标准库提供的互斥锁类。研究了其主要功能以及使用多个锁时可能遇到的问题。

下一节中，将介绍 C++ 标准库提供的机制，以使获取和释放互斥锁更加容易。

4.4. 管理通用锁

上一节中，C++ 标准库提供的不同类型的互斥锁。本节中，将介绍提供的类，以便更轻松地使用互斥锁。这是通过使用不同的包装器类来实现的。下表总结了锁管理类及其主要功能：

互斥管理器类	支持的互斥锁类型	互斥对象管理
std::lock_guard	所有类型	1
std::scoped_lock	所有类型	零或更多
std::unique_lock	所有类型	1
std::shared_lock	std::shared_mutex std::shared_timed_mutex	1

表 4.3: 锁管理类别及其特性

4.4.1. std::lock_guard

std::lock_guard 类是一个资源获取即初始化（RAII）类，使使用互斥锁更加容易，并保证在调用 lock_guard 析构函数时释放互斥锁。这在处理异常等情况下非常有用。

以下代码展示了 std::lock_guard 的用法，以及当已获取锁时如何更容易地处理异常：

```cpp
#include <format>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex mtx;
uint32_t counter{};

void function_throws() { throw std::runtime_error("Error"); }

int main() {
    auto worker = [] {
        for (int i = 0; i < 1000000; ++i) {
            mtx.lock();
            counter++;
            mtx.unlock();
        }
    };

    auto worker_exceptions = [] {
        for (int i = 0; i < 1000000; ++i) {
            try {
                std::lock_guard<std::mutex> lock(mtx);
                counter++;
                function_throws();
            } catch (std::system_error& e) {
                std::cout << e.what() << std::endl;
                return;
            } catch (...) {
                return;
            }
        }
    };

    std::thread t1(worker_exceptions);
    std::thread t2(worker);

    t1.join();
    t2.join();

    std::cout << "Final counter value: " << counter << std::endl;
}
```

function_throws() 函数只是一个会引发异常的实用函数。

代码示例中，worker_exceptions() 函数由 t1 执行。这种情况下，处理异常以打印有意义的消息。未显式获取/释放锁。这委托给 lock，一个 std::lock_guard 对象。构造锁时，会包装互斥锁并调用 mtx.lock()，获取锁。当锁销毁时，互斥锁会自动释放。如果发生异常，互斥锁也将被释放，因为已退出定义锁的范围。

为 std::lock_guard 实现了另一个构造函数，接收 std::adopt_lock_t 类型的参数。基本上，这个构造函数可以包装已经获取的非共享互斥锁，将在 std::lock_guard 析构函数中自动释放。

4.4.2. std::unique_lock

std::lock_guard 类只是一个简单的 std::mutex 包装器，自动在其构造函数中获取互斥锁（线程将阻塞，等待另一个线程释放互斥锁）并在其析构函数中释放互斥锁。这非常有用，但有时需要更多控制。例如，std::lock_guard 将在互斥锁上调用 lock() 或假定互斥锁已被获取。可能更倾向于或确实需要调用 try_lock，可能还希望 std::mutex 包装器不要在其构造函数中获取锁；所以希望将锁定推迟到稍后。所有这些功能均能由 std::unique_lock 实现。

std::unique_lock 构造函数接受一个标签作为其第二个参数，以指示要对底层互斥锁执行的操作。这里有三个选项

- std::defer_lock：不获取互斥锁的所有权。互斥锁在构造函数中未锁定，如果从未获取过，则不会在析构函数中解锁。
- std::adopt_lock：假设调用线程已获取互斥锁。将在析构函数中释放。此选项也适用于 std::lock_guard。
- std::try_to_lock：尝试在不阻塞的情况下获取互斥锁。

如果仅将互斥锁作为唯一参数传递给 std::unique_lock 构造函数，其行为与 std::lock_guard 中的行为相同：其将阻塞直到互斥锁可用，然后获取锁，并将在析构函数中释放锁。

std::unique_lock 类与 std::lock_guard 不同，它允许调用 lock() 和 unlock() 来分别获取和释放互斥锁。

4.4.3. std::scoped_lock

std::scoped_lock 类与 std::unique_lock 一样，是实现 RAII 机制的 std::mutex 包装器（如果获取了互斥锁，则会在析构函数中释放）。主要区别在于，std::unique_lock 只包装一个互斥锁，而 std::scoped_lock 包装零个或多个互斥锁。此外，互斥锁的获取顺序与传递给 std::scoped_lock 构造函数的顺序相同，可以避免死锁。

来看看下面的代码：

```
std::mutex mtx1;
std::mutex mtx2;
```

```
3
4  // Acquire both mutexes avoiding deadlock
5  std::scoped_lock lock(mtx1, mtx2);
6
7  // Same as doing this
8  // std::lock(mtx1, mtx2);
9  // std::lock_guard<std::mutex> lock1(mtx1, std::adopt_lock);
10 // std::lock_guard<std::mutex> lock2(mtx2, std::adopt_lock);
```

前面的代码片段展示了如何使用两个互斥锁。

4.4.4. std::shared_lock

std::shared_lock 类是另一个通用互斥锁所有权包装器。与 std::unique_lock 和 std::scoped_lock 一样，允许延迟锁定和转移锁所有权。std::unique_lock 和 std::shared_lock 之间的主要区别在于，后者用于在共享模式下获取/释放包装的互斥锁，而前者用于在独占模式下执行相同操作。

本节中，介绍了互斥包装类及其主要功能。接下来，将介绍另一种同步机制：条件变量。

4.5. 条件变量

条件变量是 C++ 标准库提供的另一个同步原语，允许多个线程相互通信，还允许多个线程等待来自另一个线程的通知。条件变量始终与互斥锁相关联。

以下示例中，线程必须等待计数器等于某个值：

```
1  #include <chrono>
2  #include <condition_variable>
3  #include <iostream>
4  #include <mutex>
5  #include <thread>
6  #include <vector>
7
8  int counter = 0;
9
10 int main() {
11     using namespace std::chrono_literals;
12
13     std::mutex mtx;
14     std::mutex cout_mtx;
15     std::condition_variable cv;
16
17     auto increment_counter = [&] {
18         for (int i = 0; i < 20; ++i) {
19             std::this_thread::sleep_for(100ms);
```

```
            mtx.lock();
            ++counter;
            mtx.unlock();
            cv.notify_one();
        }
    };

    auto wait_for_counter_non_zero_mtx = [&] {
        mtx.lock();
        while (counter == 0) {
            mtx.unlock();
            std::this_thread::sleep_for(10ms);
            mtx.lock();
        }
        mtx.unlock();
        std::lock_guard<std::mutex> cout_lck(cout_mtx);
        std::cout << "Counter is non-zero" << std::endl;
    };

    auto wait_for_counter_10_cv = [&] {
        std::unique_lock<std::mutex> lck(mtx);
        cv.wait(lck, [] { return counter == 10; });

        std::lock_guard<std::mutex> cout_lck(cout_mtx);
        std::cout << "Counter is: " << counter << std::endl;
    };

    std::thread t1(wait_for_counter_non_zero_mtx);
    std::thread t2(wait_for_counter_10_cv);
    std::thread t3(increment_counter);

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

等待某个条件有两种方式：一种是循环等待，并使用互斥锁作为同步机制，这在 wait_for_counter_non_zero_mtx 中实现。该函数获取锁，读取 counter 中的值，然后释放锁。然后，休眠 10 毫秒，然后再次获取锁。这在 while 循环中完成，直到 counter 非零。

条件变量简化了前面的代码。wait_for_counter_10_cv 函数会等待直到计数器等于 10。线程将等待 cv 条件变量，直到它被 t1 通知，线程会循环增加计数器。

wait_for_counter_10_cv 函数的工作方式如下：条件变量 cv 等待互斥锁 mtx。调用 wait() 后，条件变量锁定互斥锁并等待，直到条件为真（条件在作为第二个参数传递给 wait 函

数的 l ambda 中实现)。如果条件不为真，条件变量将保持等待状态，直到收到信号并释放互斥锁。一旦满足条件，条件变量将结束其等待状态，并再次锁定互斥锁以同步其对计数器的访问。

一个重要问题是，条件变量可能由不相关的线程发出信号，这称为伪唤醒。为了避免因伪唤醒而导致错误，在等待时会检查条件。当条件变量发出信号时，会再次检查条件。

如果发生伪唤醒且计数器为零（条件检查返回 false），则等待将恢复。

另一个线程通过运行 increment_counter 来增加计数器。当计数器具有所需的值（在本例中，该值为 10），就会向等待线程条件变量发出信号。

有两个函数用于向条件变量发送信号：

- cv.notify_one()：仅向其中一个等待线程发送信号
- cv.notify_all()：向所有等待线程发出信号

本节中，介绍了条件变量，并看到了使用条件变量进行同步的简单示例，以及在某些情况下如何简化同步/等待代码。现在，将注意力转向使用互斥锁和两个条件变量实现同步队列。

4.6. 实现多线程安全队列

本节中，将介绍如何实现一个简单的多线程安全队列。该队列将由多个线程访问，其中一些线程向队列添加元素（生产者线程），一些线程从队列中删除元素（消费者线程）。首先，假设只有两个线程：一个生产者线程和一个消费者线程。

队列或先进先出（FIFO）是线程间通信的标准方式。例如，需要尽快从网络连接接收包含数据的数据包，那么可能没有足够的时间在一个线程中接收所有数据包并处理它们。这种情况下，使用第二个线程来处理第一个线程读取的数据包。只使用一个消费者线程更容易同步，并且可以保证数据包将按照它们到达队列的顺序进行处理。确实，无论有多少个作为消费者的线程，数据包的读取顺序实际上都会与复制到队列的顺序相同，但消费者线程可能会由操作系统调度进出，并且处理后的数据包的完整序列可能会按不同的顺序进行。

一般来说，最简单的问题是单生产者单消费者（SPSC）队列。如果处理每个项目对于一个线程来说成本太高，不同的问题可能需要多个消费者，并且可能有不同的数据源需要处理，需要多个生产者线程。本节中描述的队列适用于每种情况。

设计队列的第一步是决定使用什么数据结构来存储队列中的项目。希望队列包含任何类型 T 的元素，因此将其实现为模板类。此外，限制队列的容量，以便固定队列中存储的最大元素数，这可以在类构造函数中设置。例如，使用列表并使队列不受限制，甚至可以使用标准模板库（STL）队列 std::queue，并让队列增长到任意大小。本章中，将实现一个固定大小的队列。对于当前的实现，将使用 STL std::vector<T> 来存储队列中的项目，将为队列类构造函数中的所有元素分配内存，之后不会有内存分配即。当队列销毁时，vector 将销毁自身并释放分配的内存。这很方便，并简化了实现。

使用 vector 作为环形缓冲区，当将一个元素存储在 vector 的末尾，下一个元素将存储在开头，可以绕行两个位置来从 vector 中写入和读取元素。

这是队列类的第一个版本：

```
template <typename T>
class synchronized_queue {
public:
    explicit synchronized_queue(size_t size) :
        capacity_{ size }, buffer_(capacity_)
        {}

private:
    std::size_t head_{ 0 };
    std::size_t tail_{ 0 };
    std::size_t capacity_;
    std::vector<T> buffer_;
};
```

`head` 和 `tail` 变量分别用于指示在何处读取或写入下一个元素，还需要知道队列何时为空或满。如果队列为空，则消费者线程将无法从队列中获取元素。如果队列已满，则生产者线程将无法将元素放入队列中。

有多种方式来指示队列何时为空以及何时为满。在此示例中，遵循以下约定：

- 如果 `tail_ == head_`，则队列为空
- 如果 `(tail_ + 1) % capacity_ == head_`，则队列已满

另一种实现方法是仅检查 `tail_ == head_` 是否为空，并使用标志来指示队列是否已满（或使用计数器来了解队列中有多少项）。此示例中避免使用额外的标志或计数器，因为标志将由消费者和生产者线程读取和写入，并且目标是尽可能减少线程之间的数据共享。

这里有一个小问题。由于检查队列是否已满的方式，会在缓冲区中丢失了一个插槽，因此实际容量是 `capacity_ - 1`。当只有一个空插槽时，会将队列视为已满。

这里还有另一个细节需要考虑：`head_ + 1` 必须考虑到将索引环绕到缓冲区（它是一个环形缓冲区）。因此，必须执行 `(head_ + 1) % capacity_`，模数运算符计算索引值除以队列容量后的余数。

以下代码展示了在同步队列中作为辅助函数的实现：

```
template <typename T>
class synchronized_queue {
public:
    explicit synchronized_queue(size_t size) :
        capacity_{ size }, buffer_(capacity_) {
    }

private:
    std::size_t next(std::size_t index) {
        return (index + 1)% capacity_;
    }
    bool is_full() const {
        return next(tail_) == head_;
    }
```

```
    bool is_empty() const {
        return tail_ == head_;
    }

    std::size_t head_{ 0 };
    std::size_t tail_{ 0 };
    std::size_t capacity_;
    std::vector<T> buffer_;
};
```

实现了一些有用的函数来更新环形缓冲区的头部和尾部，并检查缓冲区是满的还是空的。现在，可以开始实现队列功能了。

完整队列实现的代码位于本书的 GitHub 存储库中。这里，仅展示重要部分以简化流程，并仅关注队列实现的同步方面。

队列的接口有以下两个功能：

```
void push(const T& item);
void pop(T& item);
```

push 函数将一个元素插入队列，而 pop 函数从队列中获取一个元素。

在队列中插入一个元素。如果队列已满，推送将等待，直到队列至少有一个空槽（消费者从队列中删除了一个元素）。这样，生产者线程将被阻塞，直到队列至少有一个空槽（满足未满条件）。有一种称为条件变量的同步机制可以实现。push 函数将检查条件是否满足，如果满足，将在队列中插入一个元素。如果条件不满足，则将释放与条件变量关联的锁，并且线程将等待条件变量，直到条件得到满足。

条件变量可以一直等待直到释放锁。仍然需要检查队列是否已满，条件变量可能会因伪唤醒而结束等待。当条件变量收到其他线程未明确发送的通知时，就会发生这种情况。

再向队列类中添加如下三个成员变量

```
std::mutex mtx_;
std::condition_variable not_full_;
Std::condition_variable not_empty_;
```

需要两个条件变量——一个用于通知消费者队列未满（not_full_），另一个用于通知生产者队列不为空（not_empty_）。

push 的实现代码：

```
void push(const T& item) {
    std::unique_lock<std::mutex> lock(mtx_); // [1]
    not_full_.wait(lock, [this]{ return !is_full(); }); // [2]

    buffer_[tail_] = T; // [3]
    tail_ = increment(tail_);

```

```
    lock.unlock(); // [4]
    not_empty_.notify_one(); // [5]
}
```

当前值考量只有一个生产者和一个消费者的情况。

稍后将看到 pop 函数，但作为一项进步，还与互斥/条件变量同步。两个线程都尝试同时访问队列——生产者在插入元素时，消费者在移除元素时。

假设消费者首先获取锁，这发生在 [1] 中。条件变量需要使用 std::unique_lock 才能使用互斥锁。在 [2] 中，等待条件变量，直到满足等待函数谓词中的条件。如果不满足，则释放锁，以便消费者线程能够访问队列。

一旦满足条件，就会再次获取锁，并在 [3] 中更新队列。更新队列后，[4] 释放锁，然后 [5] 通知可能正在等待 not_empty 的消费者线程，队列现在实际上不为空。

std::unique_lock 类可以在其析构函数中释放互斥锁，因为不想在通知条件变量后释放锁，所以需要在 [4] 释放锁。

pop( ) 函数遵循类似的逻辑：

```
void pop(T& item)
{
    std::unique_lock<std::mutex> lock(mtx_); // [1]
    not_empty_.wait(lock, [this]{return !is_empty()}); // [2]

    item = buffer_[head_]; // [3]
    head_ = increment(head_);

    lock.unlock(); // [4]
    not_full_.notify_one(); // [5]
}
```

该代码与 push 函数中的代码非常相似。[1] 创建使用 not_empty_ 条件变量所需的 std::unique_lock 类。[2] 等待 not_empty_，直到收到队列不为空的通知。[3] 从队列中读取项目，将其分配给 item 变量，然后在 [4] 中释放锁。最后，在 [5] 通知 not_full_ 条件变量以向消费者指示队列未满。

推送和弹出函数都是阻塞的，并且分别等待队列不满或空。可能需要线程在无法向队列插入消息或从队列中获取消息时继续运行（例如，让它进行一些独立处理），然后再次尝试访问队列。

try_push 函数正是这样做的。如果可以自由获取互斥锁并且队列未满，则功能与 push 函数相同，但在这种情况下，try_push 不需要使用条件变量进行同步（但必须通知消费者）。以下是 try_push 的实现代码：

```
bool try_push(const T& item) {
    std::unique_lock<std::mutex> lock(mtx_, std::try_to_lock); // [1]
    if (!lock || is_full()) { // [2]
        return false; // [3]
    }
```

```
    buffer_[tail_] = item; // [4]
    tail_ = next(tail_);

    lock.unlock(); // [5]

    not_empty_.notify_one(); // [6]

    return true;
}
```

代码的工作原理如下：[1] 尝试获取锁并返回，而不会阻塞调用线程。如果已获取锁，则计算结果为 false。在 [2] 中，如果尚未获取锁或队列已满，try_push 将返回 false，以向调用者指示队列中未插入任何项目，并将等待/阻塞委托给调用者。[3] 返回 false，函数终止。如果已获取锁，则在函数退出并调用 std::unique_lock 析构函数时将释放锁。

获取锁并检查队列未满后，[4] 将项目插入队列，并更新 tail_。在 [5] 中，释放锁，在 [6] 中，通知消费者队列不再为空。此通知是必需的，因为消费者可能会调用 pop ，而不是 try_pop。最后，函数返回 true 来向调用者表明该项目已成功插入队列。

对应的 try_pop 函数的代码如下所示。作为练习，请尝试理解它的工作原理：

```
bool try_pop(T& item) {
    std::unique_lock<std::mutex> lock(mtx_, std::try_to_lock);
    if (!lock || is_empty()) {
        return false;
    }

    item = buffer_[head_];
    head_ = next(head_);

    lock.unlock();

    not_empty_.notify_one();

    return true;
}
```

这是本节中实现的队列的完整代码：

```
#pragma once

#include <condition_variable>
#include <mutex>
#include <vector>

namespace async_prog {
template <typename T>
```

```
class queue {
public:
    queue(std::size_t capacity) : capacity_{capacity}, buffer_
    (capacity) {}
    void push(const T& item) {
        std::unique_lock<std::mutex> lock(mtx_);
        not_full_.wait(lock, [this] { return !is_full(); });

        buffer_[tail_] = item;
        tail_ = next(tail_);

        lock.unlock();
        not_empty_.notify_one();
    }

    bool try_push(const T& item) {
        std::unique_lock<std::mutex> lock(mtx_, std::try_to_lock);
        if (!lock || is_full()) {
            return false;
        }

        buffer_[tail_] = item;
        tail_ = next(tail_);

        lock.unlock();

        not_empty_.notify_one();

        return true;
    }

    void pop(T& item) {
        std::unique_lock<std::mutex> lock(mtx_);
        not_empty_.wait(lock, [this] { return !is_empty(); });

        item = buffer_[head_];
        head_ = next(head_);

        lock.unlock();

        not_full_.notify_one();
    }

    bool try_pop(T& item) {
        std::unique_lock<std::mutex> lock(mtx_, std::try_to_lock);
        if (!lock || is_empty()) {
            return false;
        }
```

```
57
58          item = buffer_[head_];
59          head_ = next(head_);
60
61          lock.unlock();
62
63          not_empty_.notify_one();
64
65          return true;
66      }
67
68  private:
69      [[nodiscard]] std::size_t next(std::size_t idx) const noexcept {
70          return ((idx + 1) % capacity_);
71      }
72
73      [[nodiscard]] bool is_empty() const noexcept { return (head_ ==
74      tail_); }
75
76      [[nodiscard]] bool is_full() const noexcept { return (next(tail_)
77      == head_); }
78
79  private:
80      std::mutex mtx_;
81      std::condition_variable not_empty_;
82      std::condition_variable not_full_;
83
84      std::size_t head_{0};
85      std::size_t tail_{0};
86      std::size_t capacity_;
87      std::vector<T> buffer_;
88  };
89  }
```

本节中，引入了条件变量并实现了一个与互斥锁和两个条件变量同步的基本队列，这是 C++11 以来 C++ 标准库提供的两个基本同步原语。

队列示例展示了如何使用这些同步原语实现同步，并且可以用作更复杂的实用程序（例如线程池）的基本构建块。

4.7. 信号量

C++20 引入了新的同步原语来编写多线程应用程序。本节中，将介绍信号量。

信号量是一种计数器，用于管理可用于访问共享资源的许可证数量。

信号量可分为两种主要类型：

- 二进制信号量类似于互斥锁，只有两种状态：0 和 1。尽管二进制信号量在概念上类似于互

斥锁，但二进制信号量与互斥锁之间存在一些差异。

- 计数信号量可以具有大于 1 的值，用于控制对具有有限数量实例的资源的访问。

C++20 实现了二进制和计数信号量。

4.7.1. 二进制信号量

二进制信号量是一种同步原语，可用于控制对共享资源的访问。它有两种状态：0 和 1。值为 0 的信号量表示资源不可用，值为 1 的信号量表示资源可用。

二进制信号量可用于实现互斥，通过使用二进制信号量来控制对资源的访问来实现。当一个线程想要访问资源时，首先检查信号量。如果信号量为 1，则该线程可以访问该资源。如果信号量为 0，则线程必须等到信号量为 1 后才能访问该资源。

互斥锁和信号量之间最显著的区别在于，互斥锁具有独占所有权，而二进制信号量则没有。线程都可以向信号量发出信号，只有获取互斥锁的线程才能释放。互斥锁是临界区的锁定机制，而信号量更像是一种信号机制。这方面，信号量比互斥锁更接近条件变量。因此，信号量通常用于发出信号，而不是用于互斥。

C++20 中，std::binary_semaphore 是 std::counting_semaphore 特化的别名，其 LeastMaxValue 为 1。

二进制信号量必须用 1 或 0 初始化：

```
std::binary_semaphore sm1{ 0 };
std::binary_semaphore sm2{ 1 };
```

如果初始值为 0，则获取信号量将阻塞尝试获取线程，并且在获取之前，必须由另一个线程释放它。获取信号量会减少计数器，而释放它会增加计数器。如果计数器为 0 并且线程尝试获取锁（信号量），则将阻塞该线程，直到信号量计数器大于 0。

4.7.2. 计数信号量

计数信号量允许多个线程访问共享资源。计数器可以初始化为任意数字，并且每次线程获取信号量时，计数器都会减少。作为如何使用计数信号量的一个例子，修改上一节中实现的多线程安全队列，并使用信号量而不是条件变量来同步对队列的访问。

新类的成员变量如下：

```
template <typename T>
class queue {
// public methods and private helper methods

private:
    std::counting_semaphore<> sem_empty_;
    std::counting_semaphore<> sem_full_;
    std::size_t head_{ 0 };
```

```
    std::size_t tail_{ 0 };
    std::size_t capacity_;
    std::vector<T> buffer_;
};
```

仍然需要 head_ 和 tail_ 来知道在哪里读取和写入元素，需要 capacity_ 来表示索引的环绕，还需要 buffer_（一个 std::vector<T>）。但目前，不使用互斥锁，将使用计数信号量而不是条件变量。将使用其中两个：sem_empty_ 来计算缓冲区中的空槽（初始设置为 capacity_），sem_full_ 来计算缓冲区中的非空槽，初始设置为 0。

现在，来看看如何实现 push，即用于在队列中插入项目的函数。

在 [1] 中，获取 sem_empty_，减少信号量计数器。如果队列已满，则线程将阻塞，直到另一个线程释放（发出信号）sem_empty_。如果队列未满，则将项目复制到缓冲区，并在 [2] 和 [3] 更新 tail_。最后，[4] 释放 sem_full_，向另一个线程发出信号，表明队列不为空，并且缓冲区中至少有一个元素：

```
void push(const T& item) {
    sem_empty_.acquire(); // [1]

    buffer_[tail_] = item; // [2]
    tail_ = next(tail_); // [3]

    sem_full_.release(); // [4]
}
```

pop 函数用于从队列中获取元素：

```
void pop(T& item) {
    sem_full_.acquire(); // [1]

    item = buffer_[head_]; // [2]
    head_ = next(head_); // [3]

    sem_empty_.release(); // [4]
}
```

这里，在 [1] 中，如果队列不为空，我们成功获取了 sem_full_。然后，在 [2] 和 [3] 中读取项目并更新 head_。最后，向消费者线程发出信号，告知队列未满，释放 sem_empty。

第一个版本的 push 存在几个问题，最重要的问题是 sem_empty_ 允许多个线程访问队列中的临界区（[2] 和 [3]）。需要同步这个临界区并使用互斥锁。

这是使用互斥锁进行同步新版本的推送。

[2] 获取了锁（使用 std::unique_lock）。[4] 释放了锁。使用锁将同步临界区，防止多个线程同时访问它并在没有任何同步的情况下并发更新队列：

```
void push(const T& item)
{
    sem_empty_.acquire();

    std::unique_lock<std::mutex> lock(mtx_);
    buffer_[tail_] = item;
    tail_ = next(tail_);
    lock.unlock();

    sem_full_.release();
}
```

第二个问题是获取信号量是阻塞的，有时调用者线程可以做一些处理，而不仅仅是等待。`try_push` 函数（及其对应的 `try_pop` 函数）实现了此功能。看一下 `try_push` 的代码，`try_push` 可能仍会在互斥锁上阻塞：

```
bool try_push(const T& item) {
    if (!sem_empty_.try acquire()) {
        return false;
    }

    std::unique_lock<std::mutex> lock(mtx_);

    buffer_[tail_] = item;
    tail_ = next(tail_);

    lock.unlock();

    sem_full_.release();

    return true;
}
```

获取信号量时不再阻塞，而是尝试获取，如果失败，则返回 `false` 。即使可以获取信号量（计数不为零），`try_acquire` 函数也可能虚假失败并返回 `false`。

以下是使用信号量同步的队列的完整代码：

```
#pragma once

#include <mutex>
#include <semaphore>
#include <vector>

namespace async_prog {
template <typename T>
class semaphore_queue {
public:
```

```
    semaphore_queue(std::size_t capacity)
    : sem_empty_(capacity), sem_full_(0), capacity_{capacity},
    buffer_(capacity)
    {}

    void push(const T& item) {
        sem_empty_.acquire();

        std::unique_lock<std::mutex> lock(mtx_);

        buffer_[tail_] = item;
        tail_ = next(tail_);

        lock.unlock();

        sem_full_.release();
    }

    bool try_push(const T& item) {
        if (!sem_empty_.try_acquire()) {
            return false;
        }

        std::unique_lock<std::mutex> lock(mtx_);

        buffer_[tail_] = item;
        tail_ = next(tail_);

        lock.unlock();
        sem_full_.release();
        return true;
    }

    void pop(T& item) {
        sem_full_.acquire();

        std::unique_lock<std::mutex> lock(mtx_);

        item = buffer_[head_];
        head_ = next(head_);

        lock.unlock();
        sem_empty_.release();
    }

    bool try_pop(T& item) {
        if (!sem_full_.try_acquire()) {
            return false;
```

```
        }

        std::unique_lock<std::mutex> lock(mtx_);

        item = buffer_[head_];
        head_ = next(head_);

        lock.unlock();
        sem_empty_.release();

        return true;
    }
private:
    [[nodiscard]] std::size_t next(std::size_t idx) const noexcept {
        return ((idx + 1) % capacity_);
    }
private:
    std::mutex mtx_;
    std::counting_semaphore<> sem_empty_;
    std::counting_semaphore<> sem_full_;

    std::size_t head_{0};
    std::size_t tail_{0};
    std::size_t capacity_;
    std::vector<T> buffer_;
};
```

本节中，介绍了信号量，这是自 C++20 以来包含在 C++ 标准库中的一种新同步原语。介绍了如何使用它们来实现之前实现的相同队列，但使用信号量作为同步原语。

下一节中，将介绍栅栏和门闩，这是自 C++20 以来 C++ 标准库中包含的两种新同步机制。

4.8. 栅栏和门闩

本节中，将介绍栅栏门闩，这是 C++20 中引入的两个新同步原语。这些机制允许线程相互等待，从而协调并发任务的执行。

4.8.1. std::latch

std::latch 门闩是一种同步原语，允许一个或多个线程阻塞，直到指定数量的操作完成。它是一个一次性对象，当计数达到零，就无法重置。

以下示例简单说明了门闩在多线程应用程序中的使用。要编写一个函数，将向量数组的每个元素乘以二，然后将向量数组的所有元素相加。这里，使用三个线程将向量数组中的元素乘以二，然后使用一个线程将向量数组中的所有元素相加并得出结果。

这里需要两个门闩。第一个门闩将由三个线程中的每一个线程乘以两个向量元素而减少。添加

线程将等待此门闩为零。然后，主线程将等待第二个门闩以同步打印添加所有向量数组中元素的结果。也可以等待执行添加的线程在其上调用 join，这也可以使用门闩来完成。

现在，分析功能块中的代码：

```
std::latch map_latch{ 3 };
auto map_thread = [&](std::vector<int>& numbers, int start, int end) {
    for (int i = start; i < end; ++i) {
        numbers[i] *= 2;
    }
    map_latch.count_down();
};
```

每个乘法线程都会运行此 lambda 函数，将向量数组中某个范围（从开始到结束）的两个元素相乘。线程完成后，会将 map_latch 计数器减一。所有线程完成任务后，门闩计数器将为零，而阻塞等待 map_latch 的线程将能够继续将向量的所有元素相加。注意，线程访问向量数组的不同元素，因此不需要同步对向量数组本身的访问，但在完成所有乘法之前，不能立即进行累加。

累加线程的代码如下：

```
std::latch reduce_latch{ 1 };
auto reduce_thread = [&](const std::vector<int>& numbers, int& sum) {
    map_latch.wait();

    sum = std::accumulate(numbers.begin(), numbers.end(), 0);

    reduce_latch.count_down();
};
```

该线程等待 map_latch 计数器降至零，然后添加向量的所有元素，最后减少 reduce_latch 计数器（将降至零）以便主线程能够输出最终结果：

```
reduce_latch.wait();
std::cout << "All threads finished. The sum is: " << sum << '\n';
```

了解了门闩的基本应用之后，接下来让介绍一下栅栏。

4.8.2. std::barrier

std::barrier 栅栏是另一个用于同步一组线程的同步原语。std::barrier 栅栏是可重复使用的。每个线程到达栅栏并等待，直到所有参与线程都到达相同的栅栏点（与使用门闩的情况一样）。

std::barrier 和 std::latch 之间的主要区别在于重置功能。std::latch 门闩是一次性栅栏，具有无法重置的倒计时机制。一旦达到零，就会保持在零。相比之下，std::barrier 是可重复使用的。会在所有线程到达栅栏后重置，从而允许同一组线程多次在同一个栅栏处同步。

何时使用门闩，何时使用栅栏？当有线程的一次性聚集点时，请使用 `std::latch`，例如：等待多个初始化完成后再继续。当需要在任务的多个阶段或迭代计算中重复同步线程时，请使用 `std::barrier`。

现在将重写前面的示例，这次使用栅栏，而非门闩。每个线程将将其对应的向量数组元素范围乘以两倍，然后将其相加。此示例中，主线程将使用 `join()` 等待处理完成，然后将每个线程获得的结果相加。

工作线程的代码：

```
std::barrier map_barrier{ 3 };
auto worker_thread = [&](std::vector<int>& numbers, int start, int
end, int id) {
    std::cout << std::format("Thread {0} is starting...\n", id);

    for (int i = start; i < end; ++i) {
        numbers[i] *= 2;
    }

    map_barrier.arrive_and_wait();

    for (int i = start; i < end; ++i) {
        sum[id] += numbers[i];
    }

    map_barrier.arrive();
};
```

代码与栅栏同步。当工作线程完成乘法运算时，会减少 `map_barrier` 计数器并等待栅栏计数器为零。一旦降为零，线程就会结束等待并开始执行加法。重置栅栏计数器，其值再次等于三。当完成加法运算时，栅栏计数器就会再次减少，但这一次，线程不会等待，因为其任务已经完成。

当然——每个线程都可以先完成加法，然后乘以 2。它们不需要互相等待，线程完成的工作都独立于其他线程的工作，但这是一个用简单示例解释栅栏如何工作的好方法。

主线程只需使用 `join` 等待工作线程完成，然后输出结果：

```
for (auto& t : workers) {
    t.join();
}
std::cout << std::format("The total sum is {0}\n",
                         std::accumulate(sum.begin(), sum. End(), 0));
```

以下是门闩和栅栏示例的完整代码：

```
#include <algorithm>
#include <barrier>
#include <format>
#include <iostream>
```

```cpp
#include <latch>
#include <numeric>
#include <thread>
#include <vector>

void multiply_add_latch() {
    const int NUM_THREADS{3};

    std::latch map_latch{NUM_THREADS};
    std::latch reduce_latch{1};

    std::vector<int> numbers(3000);
    int sum{};
    std::iota(numbers.begin(), numbers.end(), 0);

    auto map_thread = [&](std::vector<int>& numbers, int start, int
    end) {
        for (int i = start; i < end; ++i) {
            numbers[i] *= 2;
        }
        map_latch.count_down();
    };

    auto reduce_thread = [&](const std::vector<int>& numbers, int&
    sum) {
        map_latch.wait();

        sum = std::accumulate(numbers.begin(), numbers.end(), 0);

        reduce_latch.count_down();
    };

    for (int i = 0; i < NUM_THREADS; ++i) {
        std::jthread t(map_thread, std::ref(numbers), 1000 * i, 1000 *
        (i + 1));
    }

    std::jthread t(reduce_thread, numbers, std::ref(sum));

    reduce_latch.wait();

    std::cout << "All threads finished. The total sum is: " << sum << '\n';
}

void multiply_add_barrier() {
    const int NUM_THREADS{3};

    std::vector<int> sum(3, 0);
```

```
    std::vector<int> numbers(3000);
    std::iota(numbers.begin(), numbers.end(), 0);

    std::barrier map_barrier{NUM_THREADS};

    auto worker_thread = [&](std::vector<int>& numbers, int start, int
    end, int id) {
        std::cout << std::format("Thread {0} is starting...\n", id);

        for (int i = start; i < end; ++i) {
            numbers[i] *= 2;
        }

        map_barrier.arrive_and_wait();

        for (int i = start; i < end; ++i) {
            sum[id] += numbers[i];
        }

        map_barrier.arrive();
    };

    std::vector<std::jthread> workers;
    for (int i = 0; i < NUM_THREADS; ++i) {
        workers.emplace_back(worker_thread, std::ref(numbers), 1000 *
        i, 1000 * (i + 1), i);
    }

    for (auto& t : workers) {
        t.join();
    }

    std::cout << std::format("All threads finished. The total sum is: {0}\n",
    std::accumulate(sum.begin(), sum.end(), 0));
}

int main() {
    std::cout << "Multiplying and reducing vector using barriers..." << std::endl;
    multiply_add_barrier();

    std::cout << "Multiplying and reducing vector using latches..." << std::endl;
    multiply_add_latch();
    return 0;
}
```

本节中，介绍了栅栏和门闩。虽然不像互斥锁、条件变量和信号量那样常用，但了解一下总是好的。这里介绍的简单示例说明了栅栏和门闩的常见用途：同步在不同阶段执行处理的线程。

最后，将看到一种只执行一次代码的机制，该代码会在不同的线程中多次调用。

4.9. 仅执行一次任务

有时，只需要执行一次某项任务。例如，多线程应用程序中，多个线程可能会运行相同的函数来初始化变量。正在运行的线程都可以执行此操作，但希望初始化只执行一次。

C++ 标准库提供了 std::once_flag 和 std::call_once 来实现该功能。

下面的例子将有助于理解如何使用 std::once_flag 和 std::call_once，来实现当多个线程尝试执行某项任务时只执行一次的目标：

```
#include <exception>
#include <iostream>
#include <mutex>
#include <thread>

int main() {
    std::once_flag run_once_flag;
    std::once_flag run_once_exceptions_flag;

    auto thread_function = [&] {
        std::call_once(run_once_flag, []{
            std::cout << "This must run just once\n";
        });
    };

    std::jthread t1(thread_function);
    std::jthread t2(thread_function);
    std::jthread t3(thread_function);

    auto function_throws = [&](bool throw_exception) {
        if (throw_exception) {
            std::cout << "Throwing exception\n";
            throw std::runtime_error("runtime error");
        }
        std::cout << "No exception was thrown\n";
    };

    auto thread_function_1 = [&](bool throw_exception) {
        try {
            std::call_once(run_once_exceptions_flag,
            function_throws,
            throw_exception);
        }
        catch (...) {
        }
    };

    std::jthread t4(thread_function_1, true);
    std::jthread t5(thread_function_1, true);
```

```
    std::jthread t6(thread_function_1, false);

    return 0;
}
```

示例的第一部分中，三个线程 t1、t2 和 t3 运行 thread_function 函数。此函数从 std::call_once 调用 lambda。如果运行示例，将看到消息”This must run just once”仅输出一次，正如预期的那样。

示例的第二部分中，三个线程 t4、t5 和 t6 运行 thread_function_1 函数。此函数调用 function_throws，根据参数的不同，可能会抛出或不抛出异常。此代码表明，从 std::call_once 调用的函数未成功终止，则不算完成，应再次调用 std::call_once，只有成功的函数才算运行函数。

最后一节展示了一个简单的机制，可以使用它来确保一个函数只执行一次，即使在多个线程中调用多次。

4.10. 总结

本章中，介绍了如何使用 C++ 标准库提供的基于锁的同步原语。

首先解释了条件竞争和互斥的必要性，然后研究了 std::mutex 以及如何使用它来解决条件竞争。还介绍了使用锁进行同步时的主要问题：死锁和活锁。

之后，研究了条件变量，并使用互斥锁和条件变量实现了同步队列。最后，了解了 C++20 中引入的新同步原语：信号量、门闩和栅栏。

最后，研究了 C++ 标准库提供的仅运行一次函数的机制。

本章中，了解了线程同步的基本构成要素以及多线程异步编程的基础，基于锁的线程同步是同步线程最常用的方法。

下一章，将介绍无锁线程同步。首先回顾 C++20 标准库提供的原子性、原子操作和原子类型。展示无锁绑定单生产者单消费者队列的实现，还将介绍 C++ 内存模型。

4.11. 扩展阅读

- David R. Butenhof, Programming with POSIX Threads, Addison Wesley, 1997.
- Anthony Williams, C++ Concurrency in Action, Second Edition, Manning, 2019.

第 5 章 原子操作

第 4 章中，介绍了基于锁的线程同步，了解了互斥锁、条件变量和其他线程同步原语，它们都基于获取和释放锁。这些同步机制建立在本章主题原子类型和操作之上。

接下来，将介绍原子操作是什么，以及它们与基于锁的同步原语有何不同。读完本章后，将对原子操作及其一些应用有基本的了解。基于原子操作的无锁（不使用锁）同步是一个非常复杂的主题，但希望为读者们提供有关该主题的入门介绍。

本章中，将讨论以下主要主题：

- 什么是原子操作?
- C++ 内存模型简介
- C++ 标准库提供了哪些原子类型和操作?
- 原子操作的示例，从用于收集统计数据的简单计数器和基本互斥锁到完整的单生产者单消费者（SPSC）无锁有界队列

5.1. 技术要求

需要一个支持 C++20 的最新 C++ 编译器，一些简短的代码示例将作为非常有用的 godbolt 网站（https://godbolt.org）的链接提供。对于完整的代码示例，将使用本书的代码库，该存储库位于 https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP。

这些示例可以在本地编译和运行。我们在运行 Linux（Ubuntu 24.04 LTS）的 Intel CPU 计算机上测试了代码。对于原子操作，尤其是内存排序（本章后面将详细介绍），Intel CPU 与 Arm CPU 有所不同。

注意，代码性能和分析将是第 13 章的主题，所以在本章中对性能做一些简单的评论。

5.2. 原子操作简介

原子操作是不可分割的（“原子”这个词源自希腊语 ἄτομος，atomos，不可分割）。在本节中，将介绍原子操作、其是什么，以及使用（和不使用）原子操作的原因。

5.2.1. 原子操作与非原子操作——示例

如果还记得第 4 章中的简单计数器示例，其中需要使用同步机制（我们使用了互斥锁）来从不同线程修改计数器变量以避免条件竞争。条件竞争的原因是增加计数器需要三个操作：读取计数器值、增加计数器值，以及将修改后的计数器值写回内存。如果能一次性完成这些操作，就不会出现条件竞争。

这正是原子操作可以实现的：如果有某种原子增量操作，每个线程都可以在一条指令中读取、增量和写入计数器，从而避免条件竞争，那么计数器增量操作都会完全完成。所谓完全完成，是每个线程要么增加计数器，要么什么都不做，这样就不可能在计数器增量操作的中间中断。

以下两个示例仅用于说明目的，并非多线程。在此仅关注操作，无论是原子操作还是非原子操作。

有关以下示例中显示的 C++ 代码和生成的汇编语言，请参阅https://godbolt.org/z/f4dTacsKW：

```
int counter {0};
int main() {
    counter++;
    return 0;
}
```

代码增加了一个全局计数器。现在让我们看看编译器生成的汇编代码以及 CPU 执行了哪些指令（完整汇编代码可以在前面的链接中找到）：

```
Mov eax, DWORD PTR counter[rip] // [1]
Add eax, 1 // [2]
Move DWORD PTR counter[rip], eax // [3]
```

[1] 将存储在 counter 中的值复制到 eax 寄存器，[2] 将存储在 eax 中的值加 1，最后，[3] 将 eax 寄存器的内容复制回 counter 变量。因此，线程可以执行 [1]，然后调度出去，然后另一个线程执行所有三个指令。当第一个线程完成结果递增时，计数器将仅递增一次，因此结果不正确。

以下代码的作用相同：增加一个全局计数器。不过，这次使用了原子类型和操作。要获取以下示例中的代码和生成的程序集，请参阅 https://godbolt.org/z/9hrbo31vx：

```
#include <atomic>
std::atomic<int> counter {0};
int main() {
    counter++;
    return 0;
}
```

稍后会解释 std::atomic<int> 类型和原子增量操作。生成的汇编代码如下：

```
lock add DWORD PTR counter[rip], 1
```

只生成了一条指令，用于将存储在计数器变量中的值加 1。此处的 lock 前缀表示以下指令（本例中为 add）将以原子方式执行。第二个示例中，线程在增加计数器的过程中不能中断。顺便提一下，一些 Intel x64 指令以原子方式执行，不使用 lock 前缀。

原子操作允许线程不可分割地读取、修改（例如，增加一个值）和写入，也可以用作同步原语（类似于我们在第 4 章中看到的互斥锁）。目前为止，本书中看到的所有基于锁的同步原语都是使用原子操作实现的，原子操作必须由 CPU 提供（如 lock add 指令）。

本节中，介绍了原子操作，定义了原子操作，并通过查看编译器生成的汇编指令研究了一个非常简单的原子操作实现示例。在下一节中，将介绍原子操作的一些优点和缺点。

5.2.2. 何时使用（何时不使用）原子操作

使用原子操作是一个复杂的主题，需要大量的经验，我们参加了一些关于这个主题的课程，有人建议我们不要这样做！无论如何，总是可以学习基础知识并在学习过程中进行实验。希望这本书能帮助各位在学习之旅中取得进步。

原子操作可以在以下情况下使用：

- 如果多个线程共享一个可变状态：需要同步线程是最常见的情况。当然，可以使用互斥锁之类的锁，但有时原子操作会提供更好的性能，但使用原子操作并不能保证更好的性能。
- 如果对共享状态的同步访问是细粒度的：如果必须同步的数据是整数、指针或其他 C++ 内在类型的变量，使用原子操作可能比使用锁更好。
- 提高性能：如果想要实现最大性能，原子操作可以帮助减少线程上下文切换（参见第 2 章）并减少锁带来的开销，从而降低延迟。请记住，分析代码以确保性能得到改善（将在第 13 章中深入了解这一点）。

锁可以在下列情况下使用：

- 如果受保护的数据不是细粒度的：例如，正在同步对大于 8 个字节的数据结构或对象的访问（现代 CPU 中）。
- 如果性能不是问题：锁的使用和推理就简单得多（某些情况下，使用锁比使用原子操作具有更好的性能）。
- 避免获取底层知识：要从原子操作中获得最大性能，需要大量的底层知识。我们将在 C++ 内存模型部分介绍其中的一些内容。

刚刚介绍了何时使用和何时不使用原子操作，某些应用程序（例如低延迟/高频交易系统）需要最大性能并使用原子操作来实现尽可能低的延迟。大多数应用程序都可以很好地与锁同步。

下一节将研究阻塞和非阻塞数据结构的区别，以及一些相关的概念定义。

5.3. 非阻塞数据结构

第 4 章中，介绍了同步队列的实现，并使用互斥锁和条件变量作为同步原语。因为线程被（操作系统）阻塞，所以与锁同步的数据结构称为阻塞数据结构。

不使用锁的数据结构称为非阻塞数据结构，大多数（但不是全部）非阻塞数据结构都是无锁的。

如果每个同步操作在有限的步骤内完成，而不允许无限期地等待条件变为真或假，则数据结构或算法可认为是无锁的。

无锁数据结构的类型如下：

- 无阻塞：如果所有其他线程都暂停，则线程将在有限的步骤内完成其操作
- 无锁：当多个线程同时处理数据结构时，一个线程将在有限的步骤内完成其操作
- 无等待：当多个线程同时处理数据结构时，所有线程将在有限的步骤内完成其操作

实现无锁数据结构非常复杂，需要确保它是必要的。

使用无锁数据结构的原因如下：

- 实现最大并发性：如前所述，当数据访问同步涉及细粒度数据（如本机类型变量）时，原子操作是一个不错的选择。根据前面的定义，无锁数据结构允许至少一个访问数据结构的线程，在有限数量的步骤中取得一些进展。无等待结构将允许所有访问数据结构的线程取得一些进展。但当我们使用锁时，一个线程拥有锁，而其余线程只是等待锁可用，使用无锁数据结构可以实现的并发性会好得多。
- 无死锁：不涉及锁，所以代码中不可能出现死锁。
- 性能：某些应用程序必须实现尽可能低的延迟，因此等待锁定是不可接受的。当线程尝试获取锁定但无法获取时，操作系统会阻止该线程。当线程阻塞时，调度程序会进行上下文切换，以便能够调度另一个线程执行。这些上下文切换需要时间，而对于低延迟应用程序（例如：高性能网络数据包接收器/处理器），这段时间可能太多了。

已经介绍了什么是阻塞和非阻塞数据结构，以及什么是无锁代码。我们将在下一节介绍 C++ 内存模型。

5.4. C++ 内存模型

本节介绍 C++ 内存模型及其处理并发性的方式。C++ 内存模型随 C++11 一起提供，并定义了 C++ 中内存的两个主要特性：

- 对象在内存中的布局方式（即结构方面），本书不涉及此主题。
- 内存修改顺序（即并发方面），将看到内存模型中指定的不同内存修改顺序。

5.4.1. 内存访问顺序

在解释 C++ 内存模型及其支持的不同内存顺序之前，让我们先澄清一下内存顺序的含义。内存顺序是指访问内存（即程序中的变量）的顺序，内存访问可以是读取或写入（加载和存储）。但是访问程序变量的实际顺序是什么？对于以下代码，有三个角度：编写的代码顺序、编译器生成的指令顺序，以及最后 CPU 执行指令的顺序。这三个顺序可以相同或（更可能）不同。

第一个明显的顺序是代码中的顺序。以下代码就是一个例子：

```
void func_a(int& a, int& b) {
    a += 1;
    b += 10;
    a += 2;
}
```

func_a 函数首先将变量 a 加 1，然后将变量 b 加 10，最后将变量 a 加 2。这就是我们的意图，也是我们定义要执行的语句的顺序。

编译器会将上述代码转换为汇编指令。如果代码执行结果不变，编译器可以改变语句的顺序，使生成的代码更高效。例如，对于上述代码，编译器可以先对变量 a 执行两次加法，然后对变量 b 执行加法，也可以简单地将 3 加到 a 上，然后将 10 加到 b 上。如果结果相同，编译器可以对代码进行优化。

现在让看看以下代码：

```
void func_a(int& a, int& b) {
    a += 1;
    b += 10 + a;
    a += 2;
}
```

对 b 的操作依赖于之前对 a 的操作，所以编译器无法对语句进行重新排序，生成的代码会像我们写的代码一样（操作顺序相同）。

CPU（本书使用的是现代 Intel x64 CPU）将运行生成的代码，可以按不同的顺序执行编译器生成的指令，这称为无序执行。如果结果正确，CPU 也可以这样做。

请参阅此链接以获取上述示例中显示的生成代码：https://godbolt.org/z/Mhrcnsr9e

首先，为 func_1 生成的指令显示出了优化：编译器通过将 3 添加到变量 a 中，将两个加法合并为一个。其次，为 func_2 生成的指令与 C++ 语句的顺序相同。这种情况下，因为操作之间没有依赖关系，CPU 可以无序执行指令。

总而言之，可以说 CPU 运行的代码可能与编写的代码不同（同样，假设执行结果与我们在编写的程序中预期的结果相同）。

展示的所有示例都适用于在单线程中运行的代码。代码指令可能会根据编译器优化和 CPU 无序执行以不同的顺序执行，但结果仍然正确。

有关无序执行的示例请参阅以下代码：

```
mov eax, [var1] ; load variable var1 into reg eax ;[1]
inc eax ; eax += 1 ;[2]
mov [var1], eax ; store reg eax into var1 ;[3]
xor ecx, ecx ; ecx = 0 ;[4]
inc ecx ; ecx += 1 ;[5]
add eax, ecx ; eax = eax + ecx ;[6]
```

CPU 可以按照上述代码中显示的顺序执行指令，即加载 var1 [1]。然后，在读取变量时，可以发出一些后面的指令，例如 [4] 和 [5]，然后读取了 var1，就执行 [2]，然后是 [3]，最后是 [6]。指令的执行顺序不同，但结果仍然相同。这是一个典型的乱序执行示例：CPU 发出加载指令，而不是等待数据可用，而是尽可能执行一些其他指令，以避免空闲并最大限度地提高性能。这里提到的所有优化（编译器和 CPU 都一样），都是在不考虑线程间交互的情况下进行的。

编译器和 CPU 都不知道不同的线程。这些情况下，需要告诉编译器它能做什么和不能做什么。原子操作和锁就是实现这一点的方法。

例如，使用原子变量时，可能不仅要求操作是原子的，而且还要求操作遵循一定的顺序，以便代码在运行多个线程时正常工作。因为没有涉及多个线程的任何信息，所以这不能仅由编译器或 CPU 来完成。为了指定使用顺序，C++ 内存模型提供了不同的选项：

- 宽松排序：std::memory_order_relaxed
- 获取和释放顺序：std::memory_order_acquire、std::memory_order_release、std::memory_order_acq_rel 和 std::memory_order_consume

- 顺序一致性排序：std::memory_order_seq_cst

C++ 内存模型定义了一个抽象模型，以实现与任何特定 CPU 的独立性。但是，CPU 仍然存在，并且内存模型中可用的功能可能不适用于特定 CPU。例如，Intel x64 架构非常严格，并且强制执行相当严格的内存顺序。

Intel x64 架构使用按处理器排序的内存排序模型，该模型可定义为按存储缓冲区转发进行写入排序。在单处理器系统中，内存排序模型遵循以下原则：

- 读取不会与读取重新排序
- 写入不会与写入重新排序
- 写入不会与较早的读取重新排序
- 读取可能会与较旧的写入重新排序（如果要重新排序的读取和写入涉及不同的内存位置）
- 读取和写入不会使用锁定（原子）指令重新排序

Intel 手册中有更多详细信息（请参阅章节末尾的参考资料），但前面的原则是最相关的。

在多处理器系统中，适用以下原则：

- 每个单独的处理器都使用与单处理器系统相同的排序原则
- 所有处理器都会以相同的顺序观察单个处理器的写入
- 单个处理器的写入相对于其他处理器的写入没有排序
- 内存排序遵循因果关系
- 除了执行该存储的处理器之外，其他处理器都以一致的顺序看待两个存储地址
- 锁定（原子）指令具有完全串行

Intel 架构是严格有序的；每个处理器的存储操作（写入指令）都以执行顺序被其他处理器观察到，并且每个处理器都按照程序中出现的顺序执行存储，这称为“全存储排序”（TSO ）。

ARM 架构支持弱排序（WO）。这是主要的原则：

- 读取和写入可以无序执行。与 TSO 不同，除了写入不同地址后读取之外，没有本地重新排序，而 ARM 架构允许本地重新排序（除非使用特殊指令另行指定）。
- 与在 Intel 架构中不同，写入操作不能保证同时对所有线程可见。
- 这种相对不受限制的内存排序允许核心更自由地重新排序指令，有可能提高多核性能。

必须在此指出，内存顺序越宽松，就越难推断执行的代码，并且使用原子操作正确同步多个线程也就越有挑战性。此外，无论内存顺序如何，原子性始终能得到保证。

本节中，介绍了访问内存时的顺序是什么，以及在代码中指定的顺序可能与 CPU 执行代码的顺序不同。下一节中，将介绍如何使用原子类型和操作来强制执行某些顺序。

5.4.2. 强制分配

我们已经在第 4 章和本章前面看到，从不同线程执行的对同一内存地址的非原子操作可能会导致数据争用和未定义行为。为了强制线程之间操作的顺序，将使用原子类型及其操作。本节将探讨原子在多线程代码中的效果。

以下简单示例将帮助我们了解原子操作可以做什么：

```cpp
#include <atomic>
#include <chrono>
#include <iostream>
#include <string>
#include <thread>

std::string message;
std::atomic<bool> ready{false};

void reader() {
    using namespace std::chrono::literals;

    while (!ready.load()) { // [3]
        std::this_thread::sleep_for(1ms);
    }

    std::cout << "Message received = " << message << std::endl; // [4]
}

void writer() {
    message = "Hello, World!"; // [1]
    ready.store(true); // [2]
}

int main() {
    std::thread t1(reader);
    std::thread t2(writer);

    t1.join();
    t2.join();

    return 0;
}
```

在此示例中，reader() 等待 ready 变量为真，然后打印由 writer() 设置的消息。writer() 函数设置消息，然后将 store 变量设置为真。

原子操作为我们提供了两个功能，用于在多线程代码中强制执行特定的执行顺序：

- 发生在之前：上面的代码中，[1]（设置消息变量）发生在 [2]（将原子 ready 变量设置为 true）之前。此外，[3]（循环读取 ready 变量直到其为 true）发生在 [4]（打印消息）之前。这种情况下，使用顺序一致性内存顺序（默认内存顺序）。
- 同步：这只发生在原子操作之间。上面的例子中，当 [1] 为 ready 时，该值将对不同线程中的后续读取（或写入）可见（当然，在当前线程中可见），并且当 [3] 读取 ready 时，更改的值将可见。

现在已经了解了原子操作如何强制从不同线程进行内存访问顺序，以及 C++ 内存模型提供的每个内存顺序选项。

Intel x64 架构（Intel 和 AMD 台式机处理器）在内存顺序方面非常严格，不需要额外的获取/释放指令，并且顺序一致性在性能成本方面很低。

5.4.3. 顺序一致性

顺序一致性保证程序按照您编写的方式执行。1979 年，Leslie Lamport 将顺序一致性定义为“执行结果与读取和写入按某种顺序发生的结果相同，并且每个处理器的操作都按照其程序指定的顺序按此顺序出现。”

在 C++ 中，使用 std::memory_order_seq_cst 选项指定顺序一致性。这是最严格的内存顺序，也是默认顺序。如果未指定排序选项，则将使用顺序一致性。

C++ 内存模型默认确保在代码中不存在竞争条件的情况下顺序一致性。将其视为一个约定：如果正确同步程序以防止竞争条件，C++ 将保持程序按编写顺序执行的外观。

这个模型中，所有线程都必须看到相同的操作顺序。只要计算的可见结果与无序代码的结果相同，操作仍然可以重新排序。如果读取和写入的顺序与编译代码中的顺序相同，则可以重新排序指令和操作。如果依赖关系得到满足，CPU 可以自由地在读取和写入之间重新排序其他指令。由于它定义的一致顺序，顺序一致性是最直观的排序形式。

为了说明顺序一致性，看一下以下示例：

```
#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>

std::atomic<bool> x{ false };
std::atomic<bool> y{ false };
std::atomic<int> z{ 0 };

void write_x() {
    x.store(true, std::memory_order_seq_cst);
}

void write_y() {
    y.store(true, std::memory_order_seq_cst);
}

void read_x_then_y() {
    while (!x.load(std::memory_order_seq_cst)) {}
    if (y.load(std::memory_order_seq_cst)) {
        ++z;
    }
}

void read_y_then_x()
{
```

```
    while (!y.load(std::memory_order_seq_cst)) {}
    if (x.load(std::memory_order_seq_cst)) {
        ++z;
    }
}

int main() {
    std::thread t1(write_x);
    std::thread t2(write_y);
    std::thread t3(read_x_then_y);
    std::thread t4(read_y_then_x);

    t1.join();
    t2.join();
    t3.join();
    t4.join();

    if (z.load() == 0) {
        std::cout << "This will never happen\n";
    }
    {
        std::cout << "This will always happen and z = " << z << "\n";
    }

    return 0;
}
```

运行代码时可使用 `std::memory_order_seq_cst`，所以应该注意以下几点：

- 每个线程中的操作都按照给定的顺序执行（不重新排序原子操作）。
- `t1` 和 `t2` 按顺序更新 `x` 和 `y`，`t3` 和 `t4` 看到的也是同样的顺序。如果没有这个属性，`t3` 可以看到 `x` 和 `y` 按顺序变化，但 `t4` 看到的却是相反的顺序。
- 其他顺序都可能输出相同的情况永远不会发生，因为 `t3` 和 `t4` 可以以相反的顺序看到 `x` 和 `y` 的变化。我们将在下一节中看到一个例子。

本例中的顺序一致性可表明以下两件事：

- 所有线程都可以看到每个存储操作，每个存储操作都与每个变量的所有加载操作同步，并且所有线程都以相同的顺序看到这些更改
- 每个线程的操作都以相同的顺序发生（操作按照与代码相同的顺序运行）

注意，不同线程中操作之间的顺序无法保证，并且由于线程可能被调度，因此不同线程中的指令可能会按任何顺序执行。

5.4.4. 获取-释放顺序

获取-释放顺序的严格程度不如顺序一致性顺序，无法获得顺序一致性顺序中操作的完全顺序，但仍可以进行一些同步。一般来说，随着为内存排序增加更多自由度，可能会看到性能提升，但推断代码的执行顺序将变得更加困难。

在该排序模型中，原子加载操作是 `std::memory_order_acquire` 操作，原子存储操作是 `std::memory_order_release` 操作，原子读取-修改-写入操作可能是 `std::memory_order_acquire`、`std::memory_order_release` 或 `std::memory_order_acq_rel` 操作。

获取语义（与 `std::memory_order_acquire` 一起使用）确保在源代码中，出现在获取操作之后的一个线程中的所有读取或写入操作都发生在获取操作之后。这可以防止内存对获取操作之后的读取和写入进行重新排序。

释放语义（与 `std::memory_order_release` 一起使用）确保在源代码中，出现在释放操作之前的某个线程中的读取或写入操作在释放操作之前完成。这可避免在释放操作之后的读取和写入发生内存重新排序。

下面的示例显示了与上一节关于顺序一致性的代码相同的代码，这时对原子操作使用获取-释放内存顺序：

```
#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>

std::atomic<bool> x{ false };
std::atomic<bool> y{ false };
std::atomic<int> z{ 0 };

void write_x() {
    x.store(true, std::memory_order_release);
}

void write_y() {
    y.store(true, std::memory_order_release);
}

void read_x_then_y() {
    while (!x.load(std::memory_order_acquire)) {}
    if (y.load(std::memory_order_acquire)) {
        ++z;
    }
}

void read_y_then_x() {
    while (!y.load(std::memory_order_acquire)) {}
    if (x.load(std::memory_order_acquire)) {
```

```
28          ++z;
29      }
30  }
31
32  int main() {
33      std::thread t1(write_x);
34      std::thread t2(write_y);
35      std::thread t3(read_x_then_y);
36      std::thread t4(read_y_then_x);
37
38      t1.join();
39      t2.join();
40      t3.join();
41      t4.join();
42
43      if (z.load() == 0) {
44          std::cout << "This will never happen\n";
45      } {
46          std::cout << "This will always happen and z = " << z << "\n";
47      }
48
49      return 0;
50  }
```

z 的值可能为 0；在 t1 将 x 设置为 true 并且 t2 将 y 设置为 true 之后，不再具有顺序一致性，所以 t3 和 t4 对内存访问的执行方式可能有不同的看法。由于使用了获取-释放内存顺序，t3 可能认为 x 为 true 而 y 为 false（请记住，没有强制顺序），而 t4 可能认为 x 为 false 而 y 为 true。发生这种情况时，z 的值将为 0。

除了 std::memory_order_acquire、std::memory_order_release 和 std::memory_order_acq_rel 之外，获取-释放内存顺序还包括 std::memory_order_consume 选项，根据在线 C++ 参考，“释放-消耗顺序的规范正在修订，暂时不鼓励使用 std::memory_order_consume。”

5.4.5. 宽松的内存排序

为了以宽松的内存排序执行原子操作，指定 std::memory_order_relaxed 作为内存排序选项。

宽松内存排序是最弱的同步形式。提供两项保证：

- 操作的原子性。
- 单个线程中对同一原子变量的原子操作不会重新排序，这称为修改顺序一致性。但无法保证其他线程将以相同的顺序看到这些操作。

考虑以下场景：一个线程（th1）将值存储到原子变量中。经过一段随机的时间间隔后，该变量将新的随机值覆盖。为了便于说明，假设写入的序列为 2、12、23、4、6。

另一个线程 th2 定期读取同一变量。第一次读取变量时，th2 得到的值是 23。该变量是原子的，并且加载和存储操作都使用宽松的内存顺序完成。

如果 th2 再次读取该变量，可以获取相同的值，也可以获取在之前读取的值之后写入的值。它不能读取之前写入的任何值，会违反修改顺序的一致性。当前示例中，第二次读取可能获取 23、4 或 6，但不会获取 2 或 12。如果获取 4，th1 将继续写入 8、19 和 7。现在 th2 可能获取 4、6、8、19 或 7，但不会获取 4 之前的数字，依此类推。

两个或多个线程之间，不保证任何顺序，但是当读取了一个值，就无法读取先前写入的值。

宽松模型不能用于同步线程，无法保证可见性顺序。但它在不需要在线程之间紧密协调的场景中很有用，这可以提高性能。

当执行顺序不影响程序的正确性时通常是安全的，例如：用于统计的递增计数器或参考计数器，其中递增的确切顺序并不重要。

本节中，介绍了 C++ 内存模型，以及如何允许具有不同内存顺序约束的原子操作的排序和同步。下一节中，将介绍 C++ 标准库提供的原子类型和操作。

5.5. C++ 标准库原子类型和操作

现在介绍 C++ 标准库提供，用于支持原子类型和操作的数据类型和函数，原子操作是不可分割的操作。为了能够在 C++ 中执行原子操作，需要使用 C++ 标准库提供的原子类型。

5.5.1. C++ 标准库原子类型

C++ 标准库提供的原子类型在 <atomic> 头文件中定义。

可以在在线 C++ 参考中查看 <atomic> 标头中定义的所有原子类型的文档，网址为 https://en.cppreference.com/w/cpp/atomic/atomic。我们不会在这里包含此参考中的所有内容（这就是参考的目的!），但将介绍主要概念并使用示例进一步进行阐述。

C++ 标准库提供的原子类型如下：

- std::atomic_flag：原子布尔类型（与 std::atomic<bool> 不同），是唯一保证无锁的原子类型。不提供加载或存储操作，是所有原子类型中最基本的类型。我们将使用它来实现一个非常简单的类似互斥锁的锁。
- std::atomic<T>：这是定义原子类型的模板。所有内在类型都有使用此模板定义的相应原子类型。以下是这些类型的一些示例：
  - std::atomic<bool>（及其别名 atomic_bool）：我们将使用此原子类型从多个线程实现变量的惰性一次性初始化。
  - std::atomic<int>（及其别名 atomic_int）：已经在简单的计数器示例中看到了这种原子类型。我们将在一个示例中再次使用它来收集统计数据（与计数器示例非常相似）。
  - std::atomic<intptr_t>（及其别名 atomic_intptr_t）。
  - C++20 引入了原子智能指针：std::atomic<std::shared_ptr<U>> 和 std::atomic<std::weak_ptr<U>>。

- 自 C++20 发布以来，出现了一个新的原子类型，std::atomic_ref<T>。

本章中，将重点介绍 std::atomic_flag 和一些 std::atomic 类型。对于在此处提到的其他原子类型，可以使用 C++ 参考的在线网站进行查询。

进一步解释这些类型之前，需要澄清一个非常重要的问题：仅因为类型是原子的，并不能保证无锁。这里的原子是指不可分割的操作，无锁是指具有特殊的 CPU 原子指令支持。如果某些原子操作没有硬件支持，则 C++ 标准库将使用锁来实现。

要检查原子类型是否无锁，可以使用 std::atomic<T> 类型的成员函数：

- bool is_lock_free() const noexcept：如果此类型的所有原子操作都是无锁的，则返回 true ，否则返回 false（std::atomic_flag 除外，保证始终无锁）。其余原子类型可以使用锁（如互斥锁）来实现，以保证操作的原子性。此外，某些原子类型可能只是有时无锁。如果在某个 CPU 中只有对齐的内存访问才能无锁，则同一原子类型的未对齐对象将使用锁来实现。

还有一个常量用于指示原子类型是否始终无锁：

- static constexpr bool is_always_lock_free = /* 实现/定义 */：如果原子类型始终是无锁的（例如，即使对于未对齐的对象），则此常数的值将为真

需要注意的是：原子类型不能保证无锁。std::atomic<T> 模板并不是一个可以把所有原子类型变成无锁原子类型的机制。

5.5.2. C++ 标准库原子操作

原子操作主要有两种类型：

- 原子类型的成员函数：例如，std::atomic<int> 具有 load() 成员函数，可以原子地读取其值
- 自由函数：const std::atomic_load(const std::atomic<T>* obj) 函数与前一个函数完全相同

可以在 https://godbolt.org/z/Yhdr3Y1Y8 上访问以下代码（还可以访问生成的汇编代码）。此代码显示了成员函数和自由函数的使用：

```
#include <atomic>
#include <iostream>
std::atomic<int> counter {0};
int main() {
    // Using member functions
    int count = counter.load();
    std::cout << count << std::endl;
    count++;
    counter.store(count);

    // Using free functions
```

```
    count = std::atomic_load(&counter);
    std::cout << count << std::endl;

    count++;
    std::atomic_store(&counter, count);

    return 0;
}
```

大部分原子操作函数都有一个参数来表示内存顺序。我们在 C++ 内存模型一节中已经解释了什么是内存顺序，以及 C++ 提供了哪些内存顺序类型。

5.5.3. 示例–使用 C++ 实现的简单自旋锁 atomic_flag

std::atomic_flag 原子类型是最基本的标准原子类型。它只有两种状态：设置和未设置（也可以称之为 true 和 false）。与任何其他标准原子类型相比，它始终是无锁的。因为它非常简单，所以主要用作构建块。

这是原子标志示例的代码：

```
#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>

class spin_lock {
    public:
    spin_lock() = default;

    spin_lock(const spin_lock &) = delete;

    spin_lock &operator=(const spin_lock &) = delete;

    void lock() {
        while (flag.test_and_set(std::memory_order_acquire)) {
        }
    }

    void unlock() {
        flag.clear(std::memory_order_release);
    }

private:
    std::atomic_flag flag = ATOMIC_FLAG_INIT;
};
```

使用 std::atomic_flag 之前，需要对其进行初始化。以下代码显示了如何执行此操作：

```
std::atomic_flag flag = ATOMIC_FLAG_INIT;
```

将 std::atomic_flag 初始化为确定值的唯一方法，ATOMIC_FLAG_INIT 的值由实现定义。

标志初始化后，就可以对其执行两个原子操作：

- clear：这会自动将标志设置为 false
- test_and_set：原子地将标志设置为 true 并获取其先前的值

clear 函数只能以放宽、释放或顺序一致性内存顺序调用。test_and_set 函数只能以放宽、获取或顺序一致性调用，使用其他内存顺序都会导致未定义行为。

现在来看看如何使用 std::atomic_flag 实现一个简单的自旋锁。首先，知道操作是原子的，因此线程要么清除标志，要么不清除，如果线程清除标志，则标志会完全清除。线程不可能半清除标志（请记住，对于某些非原子标志，这是可能的）。test_and_set 函数也是原子的，因此标志会设置为 true，只需一次即可获得之前的状态。

为了实现基本的自旋锁，需要一个原子标志来原子地处理锁状态和两个函数：lock () 来获取锁（就像对互斥锁的操作一样）和 unlock () 来释放锁。

简单自旋锁 unlock() 函数

我们将从最简单的函数 unlock() 开始，只会重置标志（通过将其设为 false）：

```
void unlock()
{
    flag.clear(std::memory_order_release);
}
```

代码很简单。如果省略 std::memory_order_seq_cst 参数，则会应用最严格的内存顺序选项，即顺序一致性。

简单自旋锁 lock() 函数

lock 函数有更多步骤。首先，解释一下它的作用：lock() 必须查看原子标志是否打开。如果关闭，则将其打开并完成。如果标志打开，则继续查看，直到另一个线程将其关闭。将使用 test_and_set() 来使此函数工作：

```
void lock()
{
    while (flag.test_and_set(std::memory_order_acquire)) {}
}
```

上述代码的工作方式如下：在 while 循环中，test_and_set 将标志设置为 true 并返回先前的值。如果标志已设置，再次设置它不会改变内容，并且函数返回 true，循环继续设置标志。当 test_and_set 最终返回 false 时，所以标志已清除，可以退出循环。

简单的自旋锁问题

本章包含了简单的自旋锁实现，以介绍原子类型（std::atomic_flag，最简单的标准原子类型）和操作（clear 和 test_and_set）的使用，但其存在一些严重的问题：

- 第一个缺点是性能不佳。库中的代码可供试验，自旋锁的性能会比互斥锁的性能差很多。
- 线程一直在忙等，等待标志清除。这种忙碌等待需要避免，尤其是在存在线程争用的情况下。

可以尝试上述代码作为本示例。运行该代码时，可得到表 5.1 所示的结果。该代码在每个线程中将计数器加 12 亿次。

	std::mutex	自旋锁	原子计数器
单线程	1.03 s	1.33 s	0.82 s
双线程	10.15 s	39.14 s	4.52 s
四线程	24.61 s	128.84 s	9.13 s

表 5.1：同步原语分析结果

从上表可以看出，简单的自旋锁效果很差，随着线程的增加，效果会越来越差。这个简单的例子仅用于学习，简单的 std::mutex 自旋锁和原子计数器都可以改进，以便原子类型的性能更好。

本节中，介绍了 std::atomic_flag，这是 C++ 标准库提供的最基本的原子类型。有关此类型以及 C++20 中添加的新功能的更多信息，请参阅在线 C++ 参考，网址为 https://en.cppreference.com/w/cpp/atomic/atomic_flag。

下一节中，将介绍如何创建一种简单的方法让线程告诉主线程已处理了多少个项目。

5.5.4. 示例–线程进度报告

有时，想检查线程的进度或在线程完成时收到通知。这可以通过不同的方式实现，例如：使用互斥锁和条件变量，或使用由互斥锁同步的共享变量。我们还在本章中看到了如何使用原子操作来同步计数器，将在以下示例中使用类似的计数器：

```
#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>

constexpr int NUM_ITEMS{100000};

int main() {
    std::atomic<int> progress{0};

    std::thread worker([&progress] {
        for (int i = 1; i <= NUM_ITEMS; ++i) {
            progress.store(i, std::memory_order_relaxed);
            std::this_thread::sleep_for(std::chrono::milliseconds(1));
        }
```

```
    });

    while (true) {
        int processed_items = progress.load(std::memory_order_
        relaxed);
        std::cout << "Progress: "
                  << processed_items << " / " << NUM_ITEMS
                  << std::endl;
        if (processed_items == NUM_ITEMS) {
            break;
        }
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }

    worker.join();

    return 0;
}
```

上述代码实现了一个线程（工作者），用于处理一定数量的项目（此处的处理仅通过使线程休眠来模拟）。每次线程处理一个项目时，它都会增加变量进度。主线程执行一个 `while` 循环，在每次迭代中，都会访问进度变量并写入进度报告（处理的项目数）。处理完所有项目后，循环结束。

此示例中，使用 `std::atomic<int>` 原子类型（原子整数）和两个原子操作：

- `load()`：原子地检索进度变量的值
- `store()`：原子地修改进度变量的值

工作线程处理原子读写，因此当两个线程访问读取变量时不会发生竞争条件。

`load()` 和 `store()` 原子操作有一个额外的参数来指示内存顺序。这个例子中，使用了 `std::memory_order_relaxed`。这是使用宽松内存顺序的典型示例：一个线程增加计数器，另一个线程读取。唯一需要的顺序是读取增加的值，所以宽松的内存顺序就足够了。

介绍了 `load()` 和 `store()` 原子操作来原子地读取和写入变量之后，来看看另一个简单的统计收集应用程序的示例。

5.5.5. 示例–简单统计数据

此示例基于与上一个示例相同的理念：一个线程可以使用原子操作将进度（例如，处理的项目数）传达给另一个线程。在这个新示例中，一个线程将生成一些数据，另一个线程将读取这些数据。需要同步内存访问，因为有两个线程共享同一内存，并且其中至少一个线程正在更改内存。与上一个示例一样，将使用原子操作来实现。

以下代码声明了将用于收集统计数据的原子变量 - 用于处理的项目数，另外两个（分别用于总处理时间和每个项目的平均处理时间）：

```
std::atomic<int> processed_items{0};
std::atomic<float> total_time{0.0f};
std::atomic<double> average_time{0.0};
```

使用原子浮点数和双精度数来计算总时间和平均时间。完整的示例代码中，需要确保这两种类型都无锁，所以可以使用来自 CPU 的原子指令（所有现代 CPU 都应该具有此功能）。

现在来看看工作线程如何使用变量：

```
processed_items.fetch_add(1, std::memory_order_relaxed);
total_time.fetch_add(elapsed_s, std::memory_order_relaxed);
average_time.store(total_time.load() / processed_items.load(),
std::memory_order_relaxed);
```

第一行以原子方式将处理的项目增加 1，fetch_add 函数将变量值加 1 并返回旧值（在本例中不使用）。

第二行将 elapsed_s（处理一个项目所花费的时间，以秒为单位）添加到 total_time 变量中，使用该变量来跟踪处理所有项目所花费的时间。

然后，第三行通过原子读取 total_time 和 treated_items 并原子地将结果写入 average_time 来计算每个项目的平均时间。或者，可以使用来自 fetch_add() 调用的值来计算平均时间，但不包括最后处理的项目。也可以在主线程中计算 average_time，但这里在工作线程中执行此操作，仅作为示例并练习使用原子操作。请记住，这里的目标（至少在本章中）不是速度，而是学习如何使用原子操作。

以下是统计示例的完整代码：

```
#include <atomic>
#include <chrono>
#include <iostream>
#include <random>
#include <thread>

constexpr int NUM_ITEMS{10000};

void process() {
    std::random_device rd;
    std::mt19937 gen(rd());

    std::uniform_int_distribution<> dis(1, 20);

    int sleep_duration = dis(gen);
        std::this_thread::sleep_for(std::chrono::milliseconds(sleep_
    duration));
}

int main() {
    std::atomic<int> processed_items{0};
```

```
    std::atomic<float> total_time{0.0f};
    std::atomic<double> average_time{0.0};

    std::thread worker([&] {
        for (int i = 1; i <= NUM_ITEMS; ++i) {
            auto now = std::chrono::high_resolution_clock::now();
            process();
            auto elapsed =
                std::chrono::high_resolution_clock::now() - now;
            float elapsed_s =
                std::chrono::duration<float>(elapsed).count();

            processed_items.fetch_add(1, std::memory_order_relaxed);
            total_time.fetch_add(elapsed_s, std::memory_order_relaxed);
            average_time.store(total_time.load() / processed_items.
            load(), std::memory_order_relaxed);
        }
    });

    while (true) {
        int items = processed_items.load(std::memory_order_relaxed);
        std::cout << "Progress: " << items << " / " << NUM_ITEMS <<
        std::endl;

        float time = total_time.load(std::memory_order_relaxed);
        std::cout << "Total time: " << time << " sec" << std::endl;
        double average = average_time.load(std::memory_order_relaxed);
        std::cout << "Average time: " << average * 1000 << " ms" << std::endl;

        if (items == NUM_ITEMS) {
            break;
        }
        std::this_thread::sleep_for(std::chrono::seconds(5));
    }
    worker.join();

    return 0;
}
```

总结一下目前为止我们看到的情况：

- C++ 标准原子类型：使用 `std::atomic_flag` 实现了一个简单的自旋锁，并且使用了一些 `std::atomic<T>` 类型来实现线程之间的简单数据通信，并且目前所见过的所有原子类型都无锁。
- `load()` 原子操作以原子方式读取原子变量的值。
- `store()` 原子操作以原子方式将新值写入原子变量。
- `clear()` 和 `test_and_set()`，`std::atomic_` 标志提供的特殊原子操作。

- fetch_add()，用于原子地将某个值添加到原子变量并获取其先前的值。整数和浮点类型还实现了 fetch_sub()，用于从原子变量中减去某个值，并返回其先前的值。一些用于执行按位逻辑运算的函数已专门针对整数类型实现：fetch_and()、fetch_or() 和 fetch_xor()。

下表总结了原子类型和操作。有关详尽描述，请参阅在线 C++ 参考：https://en.cppreference.com/w/cpp/atomic/atomic 表格中显示了三个新操作：exchange、compare_exchange_weak 和 compare_exchange_strong 。稍后将使用示例对其进行解释，大多数操作（即函数，而不是运算符）都有另一个用于内存顺序的参数。

操作	atomic_flag	atomic <bool>	atomic <integral>	atomic <floating-point>	atomic <other>
test_and_set	YES				
Clear	YES				
Load		YES	YES	YES	YES
Store		YES	YES	YES	YES
fetch_add, +=			YES	YES	
fetch_sub, -=			YES	YES	
fetch_and, &=			YES		
fetch_or, |=			YES		
fetch_xor, ^=			YES		
++, --			YES		
Exchange		YES	YES	YES	YES
compare_exchange_weak, compare_exchange_strong		YES	YES	YES	YES

表 5.2：原子类型和操作

回顾一下 is_lock_free() 函数和 is_always_lock_free 常量，如果 is_lock_free() 为真，则原子类型具有使用特殊 CPU 指令的无锁操作。原子类型有时可以无锁，因此 is_always_lock_free 常量告诉我们该类型是否始终是无锁的。到目前为止，看到的所有类型都无锁。来看看当原子类型是非无锁时，会发生什么。

以下显示了非无锁原子类型的代码：

```
#include <atomic>
#include <iostream>

struct no_lock_free {
    int a[128];

    no_lock_free() {
```

```
        for (int i = 0; i < 128; ++i) {
            a[i] = i;
        }
    }
};

int main() {
    std::atomic<no_lock_free> s;

    std::cout << "Size of no_lock_free: " << sizeof(no_lock_free) << "bytes\n";
    std::cout << "Size of std::atomic<no_lock_free>: " << sizeof(s) <<
    " bytes\n";

    std::cout << "Is std::atomic<no_lock_free> always lock-free: " <<
    std::boolalpha << std::atomic<no_lock_free>::is_always_lock_free <<
    std::endl;

    std::cout << "Is std::atomic<no_lock_free> lock-free: " <<
    std::boolalpha << s.is_lock_free() << std::endl;

    no_lock_free s1;
    s.store(s1);

    return 0;
}
```

执行代码时，会注意到 std::atomic<no_lock_free> 类型不是无锁的，其大小为 512 字节，这是造成这种情况的原因。当为原子变量赋值时，该值是原子写入的，但此操作不使用 CPU 原子指令，也就是说，不是无锁的。此操作的实现取决于编译器，但通常其使用互斥锁或特殊自旋锁（例如：Microsoft Visual C++）。

这里的教训是，所有原子类型都有原子操作，但并非都是无锁类型。如果原子类型不是无锁的，那么使用锁来实现它总是更好的选择，因此有些原子类型不是无锁的。

现在来看另一个示例，该示例展示了尚未涉及的原子操作：exchange 和 compare_exchange 操作。

5.5.6. 示例–惰性一次性初始化

有时初始化对象的成本可能很高。例如，给定对象可能需要连接到数据库或服务器，而建立此连接可能需要很长时间。在这些情况下，应该在使用对象之前初始化，而不是在程序中定义它时初始化，这称为延迟初始化。现在，让假设多个线程需要第一次使用该对象。如果多个线程初始化该对象，将创建不同的连接，这将是错误的，该对象只打开和关闭一个连接。因此，必须避免多次初始化。为了确保对象只初始化一次，可以使用一种称为“延迟一次性初始化”的方法。

下面显示了延迟一次性初始化的代码：

```cpp
#include <atomic>
#include <iostream>
#include <random>
#include <thread>
#include <vector>

constexpr int NUM_THREADS{8};

void process() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, 1000000);

    int sleep_duration = dis(gen);

    std::this_thread::sleep_for(std::chrono::microseconds(sleep_
    duration));
}

int main() {
    std::atomic<int> init_thread{0};

    auto worker = [&init_thread](int i) {
        process();

        int init_value = init_thread.load(std::memory_order::seq_cst); // [1]
        if (init_value == 0) {
            int expected = 0;
            if (init_thread.compare_exchange_strong(expected, i,
            std::memory_order::seq_cst)) { // [2]
                std::cout << "Previous value of init_thread: " <<
                expected << "\n";
                std::cout << "Thread " << i << " initialized\n";
            } else {
                // init_thread was already initialized
            }
        } else {
            // init_thread was already initialized
        }
    };

    std::vector<std::thread> threads;
    for (int i = 1; i <= NUM_THREADS; ++i) {
        threads.emplace_back(worker, i);
    }

    for (auto &t: threads) {
        t.join();
```

```
49      }
50
51      std::cout << "Thread: " << init_thread.load() << " initialized\n";
52      return 0;
53  }
```

本章前面看到的原子类型操作表中，有一些操作还没有介绍过。现在将使用一个示例来解释 compare_exchange_strong。示例中，有一个以 0 值开头的变量。多个线程正在运行，每个线程都有一个唯一的整数 ID（1、2、3 等）。我们希望将变量的值设置为首先设置它的线程的 ID，并且只初始化一次变量。第 4 章中，介绍了 std::once_flag 和 std::call_once，可以用它们来实现这种一次性初始化，但本章是关于原子类型和操作的，所以将使用它们来实现。

为了确保 init_thread 变量的初始化仅进行一次，并避免由于来自多个线程的写入访问而导致的竞争条件，这里使用原子 int。行 [1] 原子地读取 init_thread 的内容。如果值不为 0，则已经初始化，线程不执行其他操作。

init_thread 的当前值存储在 expected 变量中，该变量表示尝试初始化 init_thread 时期望其具有的值。现在行 [2] 执行以下步骤：

1. 将 init_thread 当前值与预期值（同样，该值等于 0）进行比较。
2. 如果比较不成功，则将 init_thread 当前值复制到 expected 中并返回 false。
3. 如果比较成功，则将 init_thread 当前值复制到 expected 中，然后将 init_thread 当前值设置为 i 并返回 true。

仅当 compare_exchange_strong 返回 true 时，当前线程才会初始化 init_thread。另外，请注意，我们需要再次进行比较（即使行 [1] 返回 0 作为 init_thread 的当前值），因为另一个线程可能已经初始化了该变量。

如果 compare_exchange_strong 返回 false，则比较失败，如果返回 true，则比较成功。另一方面，即使比较成功，compare_exchange_weak 也可能失败（即返回 false）。使用的原因是，在某些平台上，在循环内调用它可以提供更好的性能。

关于这两个函数的更多信息，可以参阅在线 C++ 参考：https://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange 在有关 C++ 标准库原子类型和操作的本节中，可以看到了以下内容：

- 最常用的标准原子类型，例如：std::atomic_flag 和 std::atomic<int>
- 最常用的原子操作：load()、store() 和 exchange_compare_strong() / exchange_compare_weak()
- 包含这些原子类型和操作的基本示例，包括惰性一次性初始化和线程进度通信

大多数原子操作（函数）都允许选择想要使用的内存顺序。下一节中，将实现一个无锁编程示例：SPSC 无锁队列。

5.6. SPSC 无锁队列

已经了解了 C++ 标准库的原子特性，例如：原子类型和操作以及内存模型和排序。现在，将看到一个使用原子实现 SPSC 无锁队列的完整示例。

这个队列的主要特点如下：

- SPSC：此队列设计为使用两个线程工作，一个线程将元素推送到队列，另一个线程从队列中获取元素。
- 有界：此队列具有固定大小。需要一种方法来检查队列何时达到其容量，以及何时没有元素。
- 无锁：此队列使用在现代 Intel x64 CPU 上始终无锁的原子类型。

开始实现队列之前，请记住无锁与无等待不同（还请记住无等待并不能完全消除等待；只是确保每个队列推送/弹出所需的步骤数有限制）。本章中，将构建一个正确且性能良好的 SPSC 无锁队列——将在后面展示如何提高其性能。

第 4 章中使用互斥锁和条件变量创建了一个 SPSC 队列，消费者线程和生产者线程可以安全访问该队列。本章将使用原子操作来实现相同的目标。我们将使用相同的数据结构 std::vector<T> 来存储队列中的项目，其大小固定，即 2 的幂。

这样，可以提高性能并快速找到下一个头和尾索引，而无需使用需要除法指令的模数运算符。当使用无锁原子类型来获得更好的性能时，需要注意影响性能的一切。

5.6.1. 为什么使用 2 的幂的缓冲区大小?

将使用一个向量数组来保存队列项。该向量数组将具有固定大小，例如 N。使向量数组的行为类似于环形缓冲区，所以访问向量数组中元素的索引将在结束后循环回到开头。第一个元素将跟在最后一个元素后面。正如在第 4 章中所了解到的，可以使用模运算符来做到这一点：

```
size_t next_index = (curr_index + 1) % N;
```

例如，大小为 4 个元素，则下一个元素的索引将按照上述代码计算。对于最后一个索引，有以下代码：

```
next_index = (3 + 1) % 4 = 4 % 4 = 0;
```

正如我们所说，向量数组将是一个环形缓冲区，因为在最后一个元素之后，将返回到第一个元素，然后是第二个元素，依此类推。

可以使用此方法获取任何缓冲区大小 N 的下一个索引。但为什么只使用 2 的幂的大小？答案很简单：性能。模数（%）运算符需要除法指令，这很昂贵。当大小 N 是 2 的幂时，可以执行以下操作：

```
size_t next_index = curr_index & (N - 1);
```

这比使用模运算符要快得多。

5.6.2. 缓冲区访问同步

要访问队列缓冲区，我们需要两个索引：

- head：当前要读取的元素的索引
- tail：下一个要写入的元素的索引

消费者线程将使用头部索引进行读写，生产者线程将使用尾部索引进行读写。需要同步对这些变量的访问，因为：

- 只有一个线程（消费者）写入 head，因为它始终看到自己的更改，所以可以以宽松的内存顺序读取。读取 tail 由读取器线程完成，它需要与生产者对 tail 的写入同步，因此需要获取内存顺序。可以使用顺序一致性，但由想要最好的性能。当消费者线程写入 head 时，需要与生产者对它的读取同步，因此需要释放内存顺序。
- 对于 tail，只有生产者线程会写入它，可以使用宽松的内存顺序来读取，但需要释放内存顺序来写入，并将其与消费者线程的读取同步。为了与消费者线程的写入同步，需要获取内存顺序来读取 head。

队列类的成员变量如下：

```
const std::size_t capacity_; // power of two buffer size
std::vector<T> buffer_; // buffer to store queue items handled like a
ring buffer
std::atomic<std::size_t> head_{ 0 };
std::atomic<std::size_t> tail_{ 0 };
```

本节中，介绍了如何同步对队列缓冲区的访问。

5.6.3. 将元素推送到队列

决定了队列的数据表示以及如何同步对其元素的访问，现在来实现将元素推送到队列的函数：

```
bool push(const T& item) {
    std::size_t tail =
        tail_.load(std::memory_order_relaxed); // [1]

    std::size_t next_tail =
        (tail + 1) & (capacity_ - 1); // [2]

    if (next_tail != head_.load(std::memory_order_acquire)) { // [3]
        buffer_[tail] = item; // [4]
        tail_.store(next_tail, std::memory_order_release); // [5]
        return true;
    }

    return false;
}
```

当前尾部索引，即数据项（如果可能）被推送到队列的缓冲区槽，在行 [1] 中原子读取。如前所述，此读取可以使用 std::memory_order_relaxed，因为只有生产者线程会更改此变量，并且它是唯一调用推送的线程。

行 [2] 计算下一个索引模容量（记住缓冲区是一个环），需要这样做来检查队列是否已满。

行 [3] 中执行检查，首先使用 std::memory_order_acquire 原子地读取 head 的当前值，希望生产者线程观察消费者线程对此变量所做的修改。然后，将其值与下一个 head 索引进行比较。

如果下一个尾部值等于当前头值，那么（按照我们的惯例）队列已满，返回 false。如果队列未满，行 [4] 将数据项复制到队列缓冲区。这里值得一提的是，数据复制不是原子的。行 [5] 原子地将新的尾部索引值写入 tail_。然后，使用 std::memory_order_release 使更改对使用 std::memory_order_acquire 原子地读取此变量的消费者线程可见。

5.6.4. 从队列中弹出元素

现在，看看 pop 函数如何实现：

```
bool pop(T& item) {
    std::size_t head =
        head_.load(std::memory_order_relaxed); // [1]

    if (head == tail_.load(std::memory_order_acquire)) { // [2]
        return false;
    }

    item = buffer_[head]; // [3]

    head_.store((head + 1) & (capacity_ - 1), std::memory_order_release); // [4]

    return true;
}
```

行 [1] 原子地读取 head_ 的当前值（要读取的下一个项的索引），使用 std::memory_order_relaxed，因为 head_ 变量仅由消费者线程修改，因此不需要强制执行顺序，而消费者线程是唯一调用 pop 的线程。

行 [2] 检查队列是否为空。如果 head_ 的当前值与 tail_ 的当前值相同，则队列为空，函数返回 false。使用 std::memory_order_acquire 原子读取 tail_ 的值，以查看生产者线程对 tail_ 所做的最新更改。

行 [3] 将数据从队列复制到作为 pop 参数传递的项目引用，此复制并非原子操作。

最后，行 [4] 更新 head_ 的值，使用 std::memory_order_release 原子地写入值，以便消费者线程查看消费者线程对 head_ 所做的更改。

SPSC 无锁队列实现的代码如下

```cpp
#include <atomic>
#include <cassert>
#include <iostream>
#include <vector>
#include <thread>

template<typename T>
class spsc_lock_free_queue {
    public:
    // capacity must be power of two to avoid using modulo operator
    when calculating the index
    explicit spsc_lock_free_queue(size_t capacity) : capacity_(capacity), buffer_(capacity)
    ↪ {
        assert((capacity & (capacity - 1)) == 0 && "capacity must be a
        power of 2");
    }

    spsc_lock_free_queue(const spsc_lock_free_queue &) = delete;

    spsc_lock_free_queue &operator=(const spsc_lock_free_queue &) = delete;

    bool push(const T &item) {
        std::size_t tail = tail_.load(std::memory_order_relaxed);
        std::size_t next_tail = (tail + 1) & (capacity_ - 1);
        if (next_tail != head_.load(std::memory_order_acquire)) {
            buffer_[tail] = item;
            tail_.store(next_tail, std::memory_order_release);
            return true;
        }

        return false;
    }

    bool pop(T &item) {
        std::size_t head = head_.load(std::memory_order_relaxed);
        if (head == tail_.load(std::memory_order_acquire)) {
            return false;
        }

        item = buffer_[head];
        head_.store((head + 1) & (capacity_ - 1), std::memory_order_release);
        return true;
    }
private:
    const std::size_t capacity_;
    std::vector<T> buffer_;
    std::atomic<std::size_t> head_{0};
    std::atomic<std::size_t> tail_{0};
```

```
48 };
```

完整示例的代码可以在以下书籍存储库中找到：https://github.com/PacktPublishing/Asynchronous-Programming-in-CPP/blob/main/Chapter_05/5x09-SPSC_lock_free_queue.cpp

本节中，实现了 SPSC 无锁队列作为原子类型和操作的应用。第 13 章中，将重新讨论此实现并改进其性能。

5.7. 总结

本章介绍了原子类型和操作、C++ 内存模型以及 SPSC 无锁队列的基本实现。

以下是我们所研究内容的总结：

- C++ 标准库原子类型和操作、它们是什么，以及如何通过一些示例展示如何使用。
- C++ 内存模型，尤其是它定义的不同内存顺序。请记住，这是一个非常复杂的主题，本节只是对它的基本介绍。
- 如何实现基本的 SPSC 无锁队列。如前所述，我们将在第 13 章中演示如何提高其性能。性能改进操作的示例包括消除错误共享（当两个变量位于同一缓存行中并且每个变量仅由一个线程修改时会发生这种情况）和减少真共享。如果现在不了解其中何内容，请不要担心，我们将在稍后介绍它并演示如何运行性能测试。

这是对原子操作的基本介绍，用于同步来自不同线程的内存访问。某些情况下，原子操作的使用相当容易，类似于收集统计数据和简单的计数器。更复杂的应用程序（例如：SPSC 无锁队列的实现），需要对原子操作有更深入的了解。本章中看到的内容有助于理解基础知识，并为进一步研究这个复杂的主题奠定基础。

下一章中，将介绍 C++ 中异步编程的两个基本构建块，即 promise 和 future。

5.8. 扩展阅读

- [Butenhof, 1997] David R. Butenhof, Programming with POSIX Threads, Addison Wesley, 1997.
- [Williams, 2019] Anthony Williams, C++ Concurrency in Action, Second Edition, Manning, 2019.
- Memory Model: Get Your Shared Data Under Control, Jana Machutová, https://www.youtube.com/watch?v=L5RCGDAan2Y.
- C++ Atomics: From Basic To Advanced, Fedor Pikus, https://www.youtube.com/watch?v=ZQFzMfHIxng.
- Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1, Intel Corporation, https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-softwaredeveloper-vol-3a-part-1-manual.pdf.

第三部分 使用 Promise、Future 和协程

在这一部分中，我们将重点转移到本书的核心主题——异步编程，这是构建响应式高性能应用程序的关键方面。将介绍如何利用诸如 Promise、Future、打包任务、std::async 函数和协程等工具并发执行任务，而不阻塞主执行流程，协程是一项革命性的功能，无需创建线程即可实现异步编程。这里还将介绍共享未来的高级技术，并研究这些概念必不可少的真实场景。这些强大的机制使我们能够开发现代软件系统所需的高效、可扩展且可维护的异步软件。

本部分包含以下章节：

- 第 6 章 Promise 和 Future
- 第 7 章，异步函数
- 第 8 章，使用协程进行异步编程

第 6 章 Promise 和 Future

前面的章节中，介绍了使用 C++ 管理和同步线程执行的基础知识。还在第 3 章中提到，要从线程返回值，可以使用 Future 和 Promise。现在，是时候学习如何在 C++ 中使用这些功能来做到这些以及做更多事情了。

Future 和 Promise 是实现异步编程必不可少的块，其定义了一种管理将来完成的任务结果的方法，通常在单独的线程中完成。

本章中，将讨论以下主要主题：

- 什么是 Future 和 Promise?
- 什么是共享 Future? 与普通 Future 有何不同?
- 什么是打包任务以及何时使用?
- 如何检查未来的状态和错误?
- 使用 Future 和 Promise 有哪些优点和缺点?
- 现实场景和解决方案的示例

那么，让开始吧!

6.1. 技术要求

自 C++11 以来, Promise 和 Future 就已经可用, 但本章中实现的一些示例使用了 C++20 中的功能，例如 std::jthread，因此本章中显示的代码可以由支持 C++20 的编译器进行编译。

请查看第 3 章中的技术要求部分，以获取有关如何安装 GCC 13 和 Clang 8 编译器的指导。

可以在以下 GitHub 库中找到所有完整的代码理论：https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP

本章的示例位于 Chapter_06 文件夹下。所有源代码文件都可以使用 CMake 进行编译：

```
cmake . && cmake build .
```

可执行二进制文件将在 bin 目录下生成。

6.2. 探索 Promise 和 Future

Future 是一个对象，表示一些未确定的结果，这些结果将在未来某个时间完成。Promise 是该结果的提供者。

自版本 C++11 以来，Promise 和 Future 一直是 C++ 标准的一部分，可以通过包含 <future> 头文件、通过类 std::promise 获得 Promise，以及通过类 std::future 获得 Future 来。

std::promise 和 std::future 对实现了一次性生产者-消费者通道，其中 Promise 为生产者，Future 为消费者。消费者（std::future）可以阻塞，直到生产者（std::promise）

的结果可用为止。

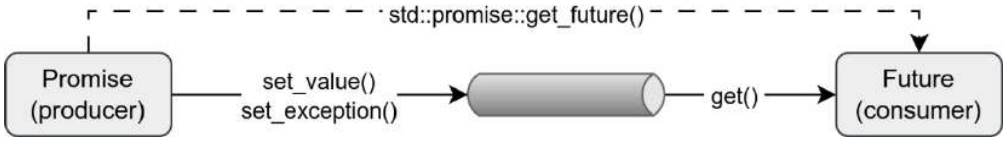

图 6.1 – Promise-Future 沟通渠道

许多现代编程语言都提供了类似的异步方法，例如 Python（带有 asyncio 库）、Scala（scala.concurrent 库中）、JavaScript（核心库）、Rust（标准库 (std) 或诸如 prospector_future 之类的包中）、Swift（Combine 框架中）和 Kotlin 等。

使用 Promise 和 Future 实现异步执行的基本原理是，想要运行以生成结果的函数在后台执行，使用新线程或当前线程，初始线程使用 Future 对象来检索函数计算的结果。函数完成时将存储此结果值，同时将使用 Future 对象作为占位符。异步函数将使用 Promise 对象将结果存储在 Future 中，而无需在初始线程和后台线程之间建立显式同步机制。当初始线程需要该值时，将从 Future 对象中检索该值。如果该值仍未准备好，则阻止执行初始线程，直到 Future 准备好为止。

有了这个思路，让一个函数异步运行就变得很简单了。只要知道这个函数可以在一个单独的线程上运行，需要避免数据竞争，但线程之间的结果通信和同步由 Promise-Future 对管理。

使用 Promise 和 Future 可以通过卸载计算来提高响应能力，并提供一种与线程和回调相比处理异步操作的结构化方法。

现在来了解一下这两个对象。

6.2.1. Promise

Promise 在 <future> 头文件中定义为 std::promise。

通过 Promise，达成了一项协议，即结果将在未来的某个时间可用。这样，就可以让后台任务完成其工作并计算结果。同时，主线程也将继续执行其任务，并在需要结果时请求它。那时，结果可能已经准备好了。

此外，Promise 可以传达是否引发异常而不是返回有效值，并且它们将确保其生命周期持续到线程完成并将结果写入其中。

因此，Promise 是一种存储结果（值或异常）的工具，稍后可通过 Future 异步获取结果。Promise 对象仅供使用一次，之后无法修改。

除了结果之外，每个 Promise 还拥有一个共享状态。共享状态是一个存储完成状态、同步机制和指向结果的指针的内存区域。允许 Promise 存储结果或异常、在完成时发出信号以及允许 Future 访问结果（如果 Promise 尚未准备好则阻塞）来确保 Promise 和 Future 之间的正确通信和同步。Promise 可以使用以下操作更新其共享状态：

- 准备就绪：Promise 将结果存储在共享状态中，并使 Promise 的状态变为就绪状态，解除等待与 Promise 关联的 Future 线程的阻塞。记住，结果可以是值（甚至是 void）或异常。

- 释放：Promise 释放对共享状态的引用，如果这是最后一个引用，则该引用将销毁。此内存释放机制类似于共享指针，及其控制块所使用的机制。除非共享状态由 std::async 创建且尚未处于就绪状态，否则此操作不会阻塞。
- 放弃：Promise 中存储了一个类型为 std::future_error 的异常，错误代码为 std::future_errc ::broken_promise，使共享状态处于准备状态，然后释放它。

可以使用其默认构造函数或自定义分配器来构造 std::promise 对象。在这两种情况下，都会创建一个具有空共享状态的新 Promise。还可以使用移动构造函数来构造 Promise，新 Promise 将具有另一个 Promise 拥有的共享状态。初始 Promise 没有共享状态。

移动 Promise 在与资源管理、通过避免副本进行优化，以及保持正确的所有权语义相关的场景中很有用；例如，当 Promise 需要在另一个线程中完成、存储在容器中、返回给 API 调用的调用者或发送到回调处理程序。

Promise 无法复制（其复制构造函数或复制赋值运算符被删除），避免两个 Promise 对象共享相同的共享状态，并在结果存储在共享状态时存在数据竞争的风险。

Promise 可以移动，同样也可以交换。标准模板库（STL）中的 std::swap 函数具有针对承诺的模板特化。

当一个 Promise 对象删除时，关联的 Future 仍将有权访问共享状态。如果在 Promise 设置值之后发生删除，则共享状态将处于释放模式，因此 Future 可以访问结果并使用。但如果在设置结果值之前删除了 Promise，则共享状态将移至放弃状态，并且 Future 在尝试获取结果时将获得 std::future_errc::broken_promise。

可以使用 std::promise 函数 set_value() 设置值，使用 set_exception() 函数设置异常。结果以原子方式存储在 Promise 的共享状态中，使其状态准备就绪。来看一个例子：

```
auto threadFunc = [](std::promise<int> prom) {
    try {
        int result = func();
        prom.set_value(result);
    } catch (...) {
        prom.set_exception(std::current_exception());
    }
};

std::promise<int> prom;
std::jthread t(threadFunc, std::move(prom));
```

创建 prom 并作为参数移入 threadFunc lambda 函数中。由于 Promise 不可复制，需要使用按值传递并将 Promise 移入参数以避免复制。

在 lambda 函数内部,调用 func() 函数,并使用 set_value() 将其结果存储在 promise 中。如果 func() 抛出异常，则会使用 set_exception() 捕获该异常并将其存储在 promise 中，可以使用 Future 在调用线程中提取此结果（值或异常）。

C++14 中，还可以使用广义 lambda 捕获将 Promise 传递到 lambda 捕获中：

```cpp
using namespace std::literals;
std::promise<std::string> prom;
auto t = std::jthread([prm = std::move(prom)] mutable {
    std::this_thread::sleep_for(100ms);
    prm.set_value("Value successfully set");
});
```

`prm = std::move(prom)` 将外部 `prom` 移至 lambda 的内部 `prm`。默认情况下，参数被捕获为常量，因此需要将 lambda 指定为可变的，以允许修改 `prm` 。

如果 `Promise` 没有共享状态（错误代码设置为 `no_state`）或者共享状态已经有存储的结果（错误代码设置为 `promise_already_satisfied`），则 `set_value()` 会抛出 `std::future_error` 异常。

`set_value()` 也可以在不指定值的情况下使用。这种情况下，只是使状态准备就绪，这可以用作栅栏。

图 6.2 显示了表示不同共享状态转换的图表。

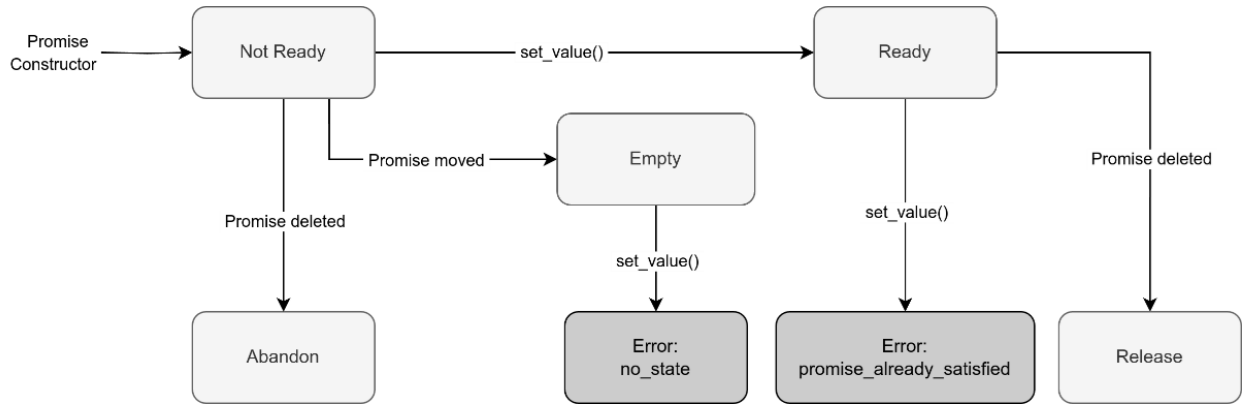

图 6.2 – Promise 共享状态转换图

还有两个函数可以设置 `Promise` 的值，即 `set_value_at_thread_exit` 和 `set_exception_at_thread_exit`。与以前一样，结果会立即存储，但使用这些新函数时，状态尚未准备就绪。当线程退出且所有线程局部变量均已销毁时，状态将变为就绪状态。当线程管理在退出前需要清理的资源（即使发生异常）时，或者当提供准确的日志活动或监控线程退出时，此功能非常有用。

抛出异常或避免数据竞争的同步机制方面，这两个函数的行为都与 `set_value()` 和 `set_exception()` 相同。

现在了解了如何将结果存储在 `Promise` 中，让我们来了解这个组合的另一个成员，即 `Future`。

6.2.2. Future

`<future>` 头文件中定义了 `std::future`。

正如我们之前看到的，`Future` 是通信渠道的消费者端，提供对 `Promise` 存储的结果的访问。

必须通过调用 get_future() 从 std::promise 对象创建 std::future 对象，或者通过 std::packaged_task 对象（本章后面有更多详细信息）或调用 std::async 函数（在第 7 章中）：

```
std::promise<int> prom;
std::future<int> fut = prom.get_future();
```

与 Promise 一样，出于同样的原因，Future 可以移动但不能复制。要从多个 Future 引用相同的共享状态，需要使用共享 Future。

get() 方法可用于检索结果。如果共享状态仍未就绪，此调用将通过内部调用 wait() 进行阻塞。当共享状态就绪时，将返回结果值。如果共享状态中存储了异常，则将重新抛出该异常：

```
try {
    int result = fut.get();
    std::cout << "Result from thread: " << result << '\n';
} catch (const std::exception& e) {
    std::cerr << "Exception: " << e.what() << '\n';
}
```

上面的例子中，使用 get() 函数从 fut future 中检索结果。如果结果是一个值，将以”Result from thread” 开头的一行输出出来。另一方面，如果抛出异常并将其存储在 Promise 中，将在调用者线程中重新抛出并捕获，并输出出以“Exception”开头的一行。

调用 get() 方法后，valid() 将返回 false。出于某种原因在 valid() 为 false 时调用 get()，则行为未定义，但 C++ 标准建议抛出 std::future_error 异常，错误代码为 std::future_errc::no_state。valid() 函数返回 false 的 Future 仍可移动。

当 Future 销毁时，会释放其共享状态引用。如果这是最后一个引用，则共享状态将销毁。除非在使用 std::async 的特定情况下，否则这些操作不会阻塞，我们将在第 7 章中进行介绍。

Future 错误和错误代码

正如前面的例子，一些处理异步执行和共享状态的函数可能会抛出 std::future_error 异常。

此异常类继承自 std::logic_error，而后者又继承自 std::exception，分别在 <stdexcept> 和 <exception> 头文件中定义。

与 STL 中定义的其他异常一样，可以使用其 code() 检查错误代码函数或通过使用其 what() 函数的解释字符串。

Future 报告的错误代码由 std::future_errorc（一个范围枚举（枚举类））定义。C++ 标准定义了以下错误代码，但实现可能会定义其他错误代码：

- broken_promise：设置结果之前删除 Promise 时报告，因此共享状态在生效之前释放。
- future_already_retrieved：当 std::promise::get_future() 调用多次时发生。
- promise_already_satisfied：如果共享状态已经有存储的结果，则由 std::promise:: set_value() 报告。
- no_state：当使用某些方法但没有共享状态时报告，因为 Promise 是使用默认构造

函数创建的或从中移动的。当调用某些打包任务（std::packaged_task）方法（例如 get_future()、make_ready_at_thread_exit() 或 reset()）时，当共享状态尚未创建时，或者当使用 std::future::get() 和尚未准备好的 Future 时（std::future::valid() 返回 false），就会发生这种情况。

等待结果

std::future 还提供了用于阻塞线程并等待结果可用的函数。这些函数是 wait()、wait_for() 和 wait_until()。wait() 函数将无限期阻塞，直到结果准备好为止，wait_for() 阻塞一段时间，wait_until() 阻塞直到达到特定时间为止。在等待期内，一旦结果可用，所有函数都将立即返回。

仅当 valid() 为真时才必须调用这些函数；否则，行为未定义。但 C++ 标准鼓励抛出带有 std::future_errc::no_state 错误代码的 std::future_error 异常。

如前所述，使用 std::promise::set_value() 而不指定值会将共享状态设置为就绪状态。它与 std::future::wait() 一起可用于实现屏障并阻止线程继续运行，直到收到信号为止。以下示例展示了此机制的作用。

首先添加所需的头文件：

```
#include <algorithm>
#include <cctype>
#include <chrono>
#include <future>
#include <iostream>
#include <iterator>
#include <sstream>
#include <thread>
#include <vector>
#include <set>
using namespace std::chrono_literals;
```

在 main() 函数中，程序将首先创建两个 Promise，numbers_promise 和 letters_promise，以及对应的 Future，numbers_ready 和 letters_ready：

```
std::promise<void> numbers_promise, letters_promise;
auto numbers_ready = numbers_promise.get_future();
auto letter_ready = letters_promise.get_future();
```

然后，input_data_thread 模拟两个按顺序运行的 I/O 线程操作，一个将数字复制到向量数组中，另一个将字母插入集合中：

```
std::istringstream iss_numbers{"10 5 2 6 4 1 3 9 7 8"};
std::istringstream iss_letters{"A b 53 C,d 83D 4B ca"};
std::vector<int> numbers;
std::set<char> letters;

```

```
std::jthread input_data_thread([&] {
    // Step 1: Emulating I/O operations.
    std::copy(std::istream_iterator<int>{iss_numbers},
              std::istream_iterator<int>{},
              std::back_inserter(numbers));

    // Notify completion of Step 1.
    numbers_promise.set_value();

    // Step 2: Emulating further I/O operations.
    std::copy_if(std::istreambuf_iterator<char>
                    {iss_letters},
                 std::istreambuf_iterator<char>{},
                 std::inserter(letters,
                               letters.end()),
                               ::isalpha);

    // Notify completion of Step 2.
    letters_promise.set_value();
});
// Wait for numbers vector to be filled.
numbers_ready.wait();
```

在此过程中，主线程使用 numbers_ready.wait() 停止执行，等待 numbers_promise 准备就绪。读取所有数字后，input_data_thread 将调用 numbers_promise.set_value()，唤醒主线程并继续执行。

如果尚未读取字母，则使用 letters_ready 的 wait_for() 函数对数字进行排序并打印，并检查它是否超时：

```
std::sort(numbers.begin(), numbers.end());
if (letter_ready.wait_for(1s) == std::future_status::timeout) {
    for (int num : numbers) std::cout << num << ' ';
    numbers.clear();
}
// Wait for letters vector to be filled.
letter_ready.wait();
```

这部分代码展示了主线程如何执行一些工作。与此同时，input_data_thread 继续处理传入的数据。然后，主线程将通过调用 letters_ready.wait() 再次等待。

最后，当所有字母都添加到集合中时，主线程将通过使用 letters_promise.set_value() 再次发出信号来唤醒，并且数字（如果尚未打印）和字母将按顺序打印：

```
for (int num : numbers) std::cout << num << ' ';
std::cout << std::endl;
for (char let : letters) std::cout << let << ' ';
std::cout << std::endl;
```

正如前面的例子中看到的，等待函数返回一个未来状态对象。接下来，了解一下这些对象是什么。

Future 状态

wait_for() 和 wait_until() 返回一个 std::future_status 对象。

Future 可以处于以下任一状态：

- 就绪：共享状态已就绪，表明可以检索结果。
- 延迟：共享状态包含延迟函数，只有在明确请求时才会计算结果。将在下一章介绍 std::async 时了解有关延迟函数的更多信息。
- 超时：共享状态准备就绪之前已过了指定的超时时间。

接下来，将介绍如何使用共享 Future 在多个 Future 之间共享一个 Promise 结果。

6.2.3. 共享 Future

std::future 只能移动，因此只有一个 Future 对象可以引用特定的异步结果。另一方面，std::shared_future 是可复制的，因此多个共享 Future 对象可以引用相同的共享状态。

因此，std::shared_future 允许从不同线程对同一共享状态进行线程安全访问。共享 Future 可用于在多个消费者或相关方之间共享计算密集型任务的结果，从而减少冗余计算。此外，还可用于通知事件或用作同步机制，其中多个线程必须等待单个任务的完成。本章后面，将介绍如何使用共享 Future 链接异步操作。

std::shared_object 的接口与 std::future 的接口相同，因此有关等待和 getter 函数的所有解释都适用于此。

可以通过使用 std::future::share() 创建共享对象：

```
std::shared_future<int> shared_fut = fut.share();
```

这会使原来的未来无效（其 valid () 函数将返回 false）。

下面的示例展示了如何同时将相同的结果发送给多个线程：

```
#define sync_cout std::osyncstream(std::cout)

int main() {
    std::promise<int> prom;
    std::future<int> fut = prom.get_future();
    std::shared_future<int> shared_fut = fut.share();
    std::vector<std::jthread> threads;
    for (int i = 1; i <= 5; ++i) {
        threads.emplace_back([shared_fut, i]() {
            sync_cout << "Thread " << i << ": Result = "
            << shared_fut.get() << std::endl;
        });
    }
    prom.set_value(5);
```

```
    return 0;
}
```

首先创建一个 Promise prom，从中获取 Future fut，最后通过调用 share() 获取共享 Future shared_fut。

然后，创建五个线程并将其添加到一个向量数组中，每个线程都有一个共享 Future 实例和一个索引。所有这些线程都将通过调用 shared_future.get() 等待 Promise prom 准备就绪。当在 Promise 共享状态中设置一个值时，所有线程都可以访问该值。运行上面的程序的输出如下：

```
Thread 5: Result = 5
Thread 3: Result = 5
Thread 4: Result = 5
Thread 2: Result = 5
Thread 1: Result = 5
```

因此，共享 Future 也可用于同时向多个线程发出信号。

6.2.4. 打包任务

打包任务或 std::packaged_task 也在 <future> 头文件中定义，它是一个类模板，用于包装要异步调用的可调用对象。其结果存储在共享状态中，可通过 std::future 对象访问。要创建 st d::packaged_task 对象，需要将表示将要调用的任务的函数签名定义为模板参数，并将所需函数作为其构造函数参数传递。以下是一些示例：

```
// Using a thread.
std::packaged_task<int(int, int)> task1(
                        std::pow<int, int>);
std::jthread t(std::move(task1), 2, 10);

// Using a lambda function.
std::packaged_task<int(int, int)> task2([](int a, int b)
{
    return std::pow(a, b);
});
task2(2, 10);

// Binding to a function.
std::packaged_task<int()> task3(std::bind(std::pow<int, int>, 2, 10));
task3();
```

上面的例子中，task1 是使用函数创建的，并使用线程执行的。另一方面，task2 是使用 lambda 函数创建的，并通过调用其方法 operator() 来执行。最后，task3 是使用 std::bind 的转发调用包装器创建的。

要获取与任务相关的 future，只需从其 packaged_task 对象中调用 get_future()：

```
std::future<int> result = task1.get_future();
```

与 Promise 和 Future 一样，可以使用默认构造函数、移动构造函数或分配器构建没有共享状态的打包任务。因此，打包任务只能移动且不可复制，赋值运算符和 swap 函数的行为与 Promise 和 Future 类似。

打包任务的析构函数的行为类似于 Promise 的析构函数；如果共享状态在有效之前被释放，则会抛出 std::future_error 异常，错误代码为 std::future_errc::broken_promise。与 Future 一样，打包任务定义了一个 valid() 函数，如果 std::packaged_task 对象具有共享状态，则该函数返回 true。

与 Promise 一样，get_future() 只能调用一次。如果多次调用此函数，则会抛出带有 future_already_retrieved 代码的 std::future_error 异常。如果打包任务是从默认构造函数创建的，因此没有共享状态，则错误代码将为 no_state。

如前面的例子所示，可以使用 operator() 来调用存储的可调用对象：

```
task1(2, 10);
```

有时，仅当运行打包任务的线程退出并且其所有线程本地对象销毁时，才使结果准备就绪。这可以通过使用 make_ready_at_thread_exit() 函数来实现。即使结果在线程退出之前尚未准备好，也会像往常一样立即计算结果，其计算也不会被推迟。

作为示例，定义以下函数：

```
void task_func(std::future<void>& output) {
    std::packaged_task<void(bool&)> task{[](bool& done){
            done = true;
    }};
    auto result = task.get_future();
    bool done = false;
    task.make_ready_at_thread_exit(done);

    std::cout << "task_func: done = "
              << std::boolalpha << done << std::endl;

    auto status = result.wait_for(0s);
    if (status == std::future_status::timeout)
        std::cout << "task_func: result not ready\n";

    output = std::move(result);
}
```

此函数创建一个名为 task 的打包任务，将其布尔参数设置为 true。还从此任务创建一个名为 result 的 Future。当通过调用 make_ready_at_thread_exit() 执行任务时，其 done 参数设置为 true，但 Future 结果仍未标记为就绪。当 task_func 函数退出时，结

果 Future 将移动到传递的引用。此时，线程退出，结果 Future 将设置为就绪。

因此，假设使用以下代码从主线程调用此任务：

```
std::future<void> result;

std::thread t{task_func, std::ref(result)};
t.join();

auto status = result.wait_for(0s);
if (status == std::future_status::ready)
    std::cout << "main: result ready\n";
```

该程序将显示以下输出：

```
task_func: done = true
task_func: result not ready
main: result ready
```

如果没有共享状态（no_state 错误代码）或者任务已经调用（promise_already_satisfied 错误代码），make_ready_at_thread_exit() 将抛出 std::future_error 异常。

打包的任务状态也可以通过调用 reset() 来重置，该函数将放弃当前状态并构造一个新的共享状态。显然，在调用 reset() 时没有状态，则会抛出 no_state 错误代码的异常。重置后，必须通过调用 get_future() 获取新的 Future。

以下示例打印前 10 个 2 的幂数，每个数字都是通过调用相同的 packaged_task 对象计算得出的。每次循环迭代中，packaged_task 都会重置，并检索一个新的 Future 对象：

```
std::packaged_task<int(int, int)> task([](int a, int b){
    return std::pow(a, b);
});

for (int i=1; i<=10; ++i) {
    std::future<int> result = task.get_future();
    task(2, i);
    std::cout << "2^" << i << " = "
              << result.get() << std::endl;
    task.reset();
}
```

这是执行上述代码时的输出：

```
2^1 = 2
2^2 = 4
2^3 = 8
2^4 = 16
2^5 = 32
```

```
2^6 = 64
2^7 = 128
2^8 = 256
2^9 = 512
2^10 = 1024
```

下一章中，std::async 提供了一种更简单的方法来实现相同的结果。std::packaged_task 的唯一优势是能够准确指定任务将在哪个线程中运行。

现在，了解了如何使用 promise、future 和打包任务，现在是时候了解这种方法的优点，以及可能存在的缺点了。

6.3. Promise 和 Future 的优点和缺点

使用 Promise 和 Future 既有优点，也有缺点：

6.3.1. 优点

作为管理异步操作的高级抽象，使用 Promise 和 Future 来编写和推理并发代码变得更加简单且不容易出错。

Future 和 Promise 支持并发执行任务，让程序能够高效使用多个 CPU 核心。这可以提高计算密集型任务的性能并缩短执行时间。

此外，通过将操作的启动与完成分离来促进异步编程。正如稍后将看到的，这对于 I/O 密集型任务（例如：网络请求或文件操作）特别有用。在这些任务中，程序可以在等待异步操作完成的同时继续执行其他任务。因此，可以返回一个值，也可以返回一个异常，从而允许异常从异步任务传播到等待其完成的调用者代码部分，从而为错误处理和恢复提供了一种更清晰的方式。

它们还提供了一种同步任务完成和检索其结果的机制，这有助于协调并行任务并管理它们之间的依赖关系。

6.3.2. 缺点

不幸的是，并非所有都是好消息。

例如，使用 Future 和 Promise 进行异步编程可能会对处理任务之间的依赖关系，或管理异步操作的生命周期时增加复杂性。此外，如果存在循环依赖关系，则可能会发生死锁。

同样，使用 Future 和 Promise 可能会带来一些性能开销，因为在幕后发生的同步机制涉及协调异步任务和管理共享状态。

与其他并发或异步解决方案一样，使用 Future 和 Promise 的代码调试与同步代码相比更具挑战性，执行流程可能是非线性的并且涉及多个线程。

现在是时候通过一些示例来解决现实问题了。

6.4. 现实场景和解决方案的示例

现在已经了解了一些创建异步程序的新构建块，让我们为一些实际场景构建解决方案。本节中，介绍如何执行以下操作：

- 取消异步操作
- 返回合并结果
- 链接异步操作并创建管道
- 创建线程安全的单生产者单消费者（SPSC）任务队列

6.4.1. 取消异步操作

正如之前看到的，future 提供了在等待结果之前检查完成或超时的功能。这可以通过检查 std::future、wait_for() 或 wait_until() 函数返回的 std::future_status 对象来完成。

通过将 Future 与取消标志（通过 std::atomic_bool）或超时等机制相结合，可以在必要时优雅地终止长时间运行的任务。只需使用 wait_for() 和 wait_until() 函数即可实现超时取消。

使用取消标志或令牌取消任务可以通过传递对定义为 std::atomic_bool 的取消标志的引用来实现，其中调用者线程将其值设置为 true 以请求取消任务，而工作线程会定期检查此标志以及它是否已设置。如果已设置，将正常退出并执行要完成的清理工作。

首先定义一个长时间运行的任务函数：

```
const int CHECK_PERIOD_MS = 100;

bool long_running_task(int ms,
            const std::atomic_bool& cancellation_token) {
    while (ms > 0 && !cancellation_token) {
        ms -= CHECK_PERIOD_MS;
        std::this_thrsead::sleep_for(100ms);
    }
    return cancellation_token;
}
```

long_running_task 函数接受一个以毫秒（ms）为单位的任务运行周期和一个表示取消标记的原子布尔值（cancellation_token）的引用作为参数，函数将定期检查取消标记是否设置为 true。当运行周期过去或取消标记设置为 true 时，线程将退出。

主线程中可以使用两个封装好的 task 对象来执行该函数，task1 在线程 t1 中运行，时长 500ms，task2 在线程 t2 中运行，时长 1s，二者共享同一个取消标记：

```
std::atomic_bool cancellation_token{false};
std::cout << "Starting long running tasks...\n";

```

```
std::packaged_task<bool(int, const std::atomic_bool&)>
                task1(long_running_task);

std::future<bool> result1 = task1.get_future();
std::jthread t1(std::move(task1), 500,
                std::ref(cancellation_token));

std::packaged_task<bool(int, const std::atomic_bool&)>
                task2(long_running_task);
std::future<bool> result2 = task2.get_future();
std::jthread t2(std::move(task2), 1000,
                std::ref(cancellation_token));

std::cout << "Cancelling tasks after 600 ms...\n";
this_thread::sleep_for(600ms);
cancellation_token = true;

std::cout << "Task1, waiting for 500 ms. Cancelled = "
          << std::boolalpha << result1.get() << "\n";
std::cout << "Task2, waiting for 1 second. Cancelled = "
          << std::boolalpha << result2.get() << "\n";
```

两个任务启动后，主线程休眠 600 毫秒。唤醒时，将取消标记设置为 true。此时，任务 1 已经完成，但任务 2 仍在运行。因此，任务 2 取消。

这个解释与获得的输出一致：

```
Starting long running tasks...
Cancelling tasks after 600 ms...
Task1, waiting for 500 ms. Cancelled = false
Task2, waiting for 1 second. Cancelled = true
```

接下来看看如何将几个异步计算结果合并到一个 Future 中。

6.4.2. 返回合并结果

异步编程中的另一种常见方法是，使用多个 Promise 和 Future 将复杂任务分解为较小的独立子任务。每个子任务都可以在单独的线程中启动，其结果存储在相应的 Promise 中。然后，主线程可以使用 Future 等待所有子任务完成并合并以获得结果。

这种方法有助于实现多个独立任务的并行处理，从而可以有效利用多个核心来加快计算速度。

看一个模拟值计算和 I/O 操作的任务示例。我们希望该任务返回一个元组，其中包含结果、计算值（int 值）和信息从文件中读取为字符串对象。因此，定义 CombineFunc 函数，该函数接受一个 CombineProm Prom 作为参数，该 Promise 包含一个包含结果类型的元组。

此函数将创建两个线程，computeThread 和 fetchData，各自的 computeProm 和 fetchProm，以及 computeFut 和 fetchFut：

```cpp
void combineFunc(std::promise<std::tuple<int,
                 std::string>> combineProm) {
    try {
        // Thread to simulate computing a value.
        std::cout << "Starting computeThread...\n";
        auto computeVal = [](std::promise<int> prom)
                          mutable {
            std::this_thread::sleep_for(1s);
            prom.set_value(42);
        };
        std::promise<int> computeProm;
        auto computeFut = computeProm.get_future();
        std::jthread computeThread(computeVal,
                                   std::move(computeProm));

        // Thread to simulate downloading a file.
        std::cout << "Starting dataThread...\n";
        auto fetchData = [](
        std::promise<std::string> prom) mutable {
            std::this_thread::sleep_for(2s);
            prom.set_value("data.txt");
        };
        std::promise<std::string> fetchProm;
        auto fetchFut = fetchProm.get_future();
        std::jthread dataThread(fetchData,
                                   std::move(fetchProm));

        combineProm.set_value({
            computeFut.get(),
            fetchFut.get()
        });
    } catch (...) {
        combineProm.set_exception(
                    std::current_exception());
    }
}
```

可以看到，两个线程将异步且独立地执行，生成结果并将其存储在各自的 Promise 中。

组合 Promise 的设置方式是，在每个未来上调用 get() 函数，并将它们的结果组合成一个元组，该元组用于通过调用其 set_value() 函数来设置组合 Promise 的值：

```cpp
combineProm.set_value({computeFut.get(), fetchFut.get()});
```

通过使用线程并设置 CombineProm 及其 CombineFut，可以像往常一样调用 CombineFunc 任务。在此 Future 上调用 get() 函数将返回一个元组：

```
std::promise<std::tuple<int, std::string>> combineProm;
auto combineFuture = combineProm.get_future();
std::jthread combineThread(combineFunc,
                           std::move(combineProm));

auto [data, file] = combineFuture.get();
std::cout << "Value [ " << data
          << " ] File [ « << file << « ]\n»;
```

运行该示例将显示以下结果：

```
Creating combined promise...
Starting computeThread...
Starting dataThread...
Value [ 42 ] File [ data.txt ]
```

现在，让继续了解如何使用 Promise 和 Future 创建管道。

6.4.3. 链接异步操作

Promise 和 Future 可以串联在一起，按顺序执行多个异步操作。可以创建一个管道，其中一个 Future 的结果将成为下一个操作的 Promise 的输入。这允许组合复杂的异步工作流，其中一个任务的输出将输入到下一个任务中。

此外，可以在管道中进行分支，并保持某些任务处于关闭状态，直到需要时才执行。这可以通过使用具有延迟执行的 Future 来实现，这在计算成本很高但结果可能并不总是需要的情况下很有用。因此，可以使用 Future 异步启动计算，并仅在需要时检索结果。由于只能使用 std::async 创建具有延迟状态的 Future，将在下一章中讨论这个问题。

本节中，将重点创建以下任务图：

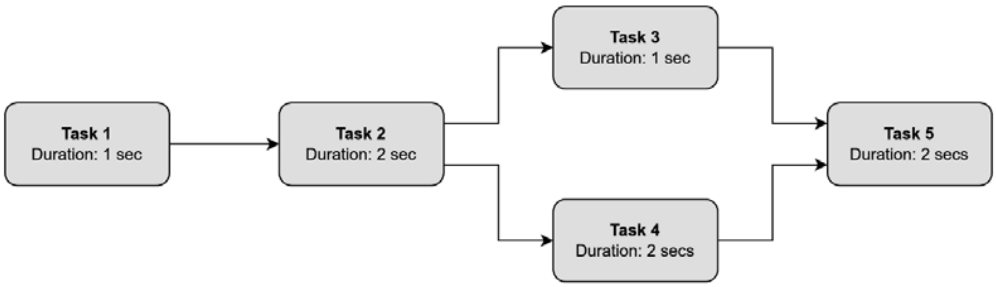

图 6.3 –管道示例

首先定义一个名为 Task 的模板类，该类接受可调用函数作为模板参数，定义要执行的函数。该类还创建与依赖任务共享 Future 的任务。这些任务将使用共享的 Future 等待前置任务通过在相关 Promise 中调用 set_value() 来发出完成信号，然后再运行自己的任务：

```
#define sync_cout std::osyncstream(std::cout)

```

```cpp
template <typename Func>
class Task {
    public:
    Task(int id, Func& func)
        : id_(id), func_(func), has_dependency_(false) {
        sync_cout << "Task " << id
        << " constructed without dependencies.\n";
        fut_ = prom_.get_future().share();
    }

    template <typename... Futures>
    Task(int id, Func& func, Futures&&... futures)
        : id_(id), func_(func), has_dependency_(true) {
        sync_cout << "Task " << id
        << " constructed with dependencies.\n";
        fut_ = prom_.get_future().share();
        add_dependencies(
        std::forward<Futures>(futures)...);
    }

    std::shared_future<void> get_dependency() {
        return fut_;
    }

    void operator()() {
        sync_cout << "Running task " << id_ << '\n';
        wait_completion();
        func_();
        sync_cout << "Signaling completion of task "
                  << id_ << '\n';
        prom_.set_value();
    }

private:
    template <typename... Futures>
    void add_dependencies(Futures&&... futures) {
        (deps_.push_back(futures), ...);
    }

    void wait_completion() {
        sync_cout << "Waiting completion for task "
        << id_ << '\n';
        if (!deps_.empty()) {
            for (auto& fut : deps_) {
                if (fut.valid()) {
                    sync_cout << "Fut valid so getting "
                              << "value in task "
                              << id_ << '\n';
```

```
                    fut.get();
                }
            }
        }
    }

private:
    int id_;
    Func& func_;
    std::promise<void> prom_;
    std::shared_future<void> fut_;
    std::vector<std::shared_future<void>> deps_;
    bool has_dependency_;
};
```

一步一步拆解这个 Task 类是如何实现的。

有两个构造函数：一个用于初始化与其他任务没有依赖关系的 Task 类型对象，另一个模板化构造函数用于初始化具有可变数量依赖任务的任务。两者都初始化标识符（id_）、执行任务时调用的函数（func_）、布尔值变量指示任务是否具有依赖项（has_dependency_），以及共享 Future fut_，与依赖于此任务的任务共享。fut_ 是从用于发出任务完成信号的 prom_ 中检索的，模板化构造函数还调用 add_dependencies 函数转发作为参数传递的 Future，这些 Future 将存储在 deps_ 向量数组中。

get_dependency() 函数仅返回依赖任务用于等待当前任务完成的共享 Future。

最后，operator() 通过调用 wait_completion() 等待先前任务的完成，这会检查存储在 deps_ 向量数组中的每个共享 Future 是否有效，并通过调用 get() 等待结果准备就绪。当所有共享 Future 都准备就绪，即所有先前任务都已完成，就会调用 func_ 函数来运行任务，然后通过调用 set_value() 将 prom_ promise 设置为就绪，从而触发依赖任务。

主线程中，定义的管道如下，创建如图 6.3 所示的图：

```
auto sleep1s = []() { std::this_thread::sleep_for(1s); };
auto sleep2s = []() { std::this_thread::sleep_for(2s); };

Task task1(1, sleep1s);
Task task2(2, sleep2s, task1.get_dependency());
Task task3(3, sleep1s, task2.get_dependency());
Task task4(4, sleep2s, task2.get_dependency());
Task task5(5, sleep2s, task3.get_dependency(),
                       task4.get_dependency());
```

然后，需要通过触发所有任务并调用它们的 operator() 来启动管道。由于 task1 没有依赖项，将立即开始运行。所有其他任务都将等待其上游任务完成其工作：

```
sync_cout << "Starting the pipeline..." << std::endl;
task1();
```

```
task2();
task3();
task4();
task5();
```

最后，需要等待管道完成所有任务的执行。可以通过简单地等待最后一个任务 `task 5` 返回的共享 `Future` 准备就绪来实现：

```
sync_cout << "Waiting for the pipeline to finish...\n";
auto finish_pipeline_fut = task5.get_dependency();
finish_pipeline_fut.get();
sync_cout << "All done!" << std::endl;
```

以下是运行此示例的输出：

```
Task 1 constructed without dependencies.
Getting future from task 1
Task 2 constructed with dependencies.
Getting future from task 2
Task 3 constructed with dependencies.
Getting future from task 2
Task 4 constructed with dependencies.
Getting future from task 4
Getting future from task 3
Task 5 constructed with dependencies.
Starting the pipeline...
Running task 1
Waiting completion for task 1
Signaling completion of task 1
Running task 2
Waiting completion for task 2
Fut valid so getting value in task 2
Signaling completion of task 2
Running task 3
Waiting completion for task 3
Fut valid so getting value in task 3
Signaling completion of task 3
Running task 4
Waiting completion for task 4
Fut valid so getting value in task 4
Signaling completion of task 4
Running task 5
Waiting completion for task 5
Fut valid so getting value in task 5
Fut valid so getting value in task 5
Signaling completion of task 5
Waiting for the pipeline to finish...
```

```
Getting future from task 5
All done!
```

必须注意这种方法可能出现的一些问题。首先，依赖关系图必须是有向无环图（DAG），因此依赖任务之间不能有循环或环路。否则，就会发生死锁，因为任务可能在等待未来发生的任务，但尚未启动。此外，需要足够的线程来同时运行所有任务或按顺序启动任务；否则，线程等待尚未启动的任务完成也可能导致死锁。

这种方法的一个常见用例可以在 MapReduce 算法中找到，其中大型数据集在多个节点上并行处理，并且可以使用 Future 和线程来同时执行 map 和 Reduce 任务，从而实现高效的分布式数据处理。

6.4.4. 线程安全的 SPSC 任务队列

最后一个例子中，将展示如何使用 Promise 和 Future 来创建 SPSC 队列。

生产者线程会为要添加到队列中的每个项目创建一个 Promise。消费者线程会等待从空队列槽中获取的 Future。生产者添加完项目后，它会在相应的 Promise 上设置值，通知等待的消费者。这允许在线程之间进行高效的数据交换，同时保持线程安全。

首先定义线程安全的队列类：

```
template <typename T>
class ThreadSafeQueue {
public:
    void push(T value) {
        std::lock_guard<std::mutex> lock(mutex_);
        queue_.push(std::move(value));
        cond_var_.notify_one();
    }

    T pop() {
        std::unique_lock<std::mutex> lock(mutex_);
        cond_var_.wait(lock, [&]{
            return !queue_.empty();
        });
        T value = std::move(queue_.front());
        queue_.pop();
        return value;
    }

private:
    std::queue<T> queue_;
    std::mutex mutex_;
    std::condition_variable cond_var_;
};
```

此示例中，仅使用互斥锁在推送或弹出元素时对所有队列数据结构进行互斥。我们希望保持此

示例简单，并专注于 Promise 和 Future 的交互。更好的方法可能是使用向量或循环数组，并使用互斥锁控制对队列中各个元素的访问。

队列还使用条件变量 cond_var_，在尝试弹出元素时等待队列是否为空，并在推送元素时通知一个等待线程。通过移动元素，可以将元素移入和移出队列。这是必要的，因为队列将存储 Future，而 Future 可移动，但不可复制。

线程安全队列将用于定义一个存储 Future 的任务队列：

```
using TaskQueue = ThreadSafeQueue<std::future<int>>;
```

然后，定义一个函数 producer，接受对队列的引用，以及将要生成的值 val。此函数仅创建一个 Promise，从 Promise 中检索 Future，并将该 Future 推送到队列中，通过让线程等待随机数毫秒来模拟运行并生成值 val 的任务。最后，该值存储在 Promise 中：

```
void producer(TaskQueue& queue, int val) {
    std::promise<int> prom;
    auto fut = prom.get_future();
    queue.push(std::move(fut));

    std::this_thread::sleep_for(
            std::chrono::milliseconds(rand() % MAX_WAIT));

    prom.set_value(val);
}
```

通信通道的另一端，消费者函数接受对同一队列的引用。同样，可通过等待随机的毫秒数来模拟在消费者端运行的任务。然后，从队列中弹出一个 Future，并检索其结果：

```
void consumer(TaskQueue& queue) {
    std::this_thread::sleep_for(
            std::chrono::milliseconds(rand() % MAX_WAIT));

    std::future<int> fut = queue.pop();
    try {
        int result = fut.get();
        std::cout << "Result: " << result << "\n";
    } catch (const std::exception& e) {
        std::cerr << "Exception: " << e.what() << '\n';
    }
}
```

对于此示例，将使用以下常量：

```
const unsigned VALUE_RANGE = 1000;
const unsigned RESULTS_TO_PRODUCE = 10; // Numbers of items to produce.
const unsigned MAX_WAIT = 500; // Maximum waiting time (ms) when producing items.
```

主线程中，启动了两个线程；第一个线程运行生产者函数 producerFunc，将一些 Future 推送到队列中，而第二个线程运行消费者函数 consumerFunc，从队列中使用元素：

```
TaskQueue queue;

auto producerFunc = [](TaskQueue& queue) {
    auto n = RESULTS_TO_PRODUCE;
    while (n-- > 0) {
        int val = rand() % VALUE_RANGE;
        std::cout << "Producer: Sending value " << val
                  << std::endl;
        producer(queue, val);
    }
};

auto consumerFunc = [](TaskQueue& queue) {
    auto n = RESULTS_TO_PRODUCE;
    while (n-- > 0) {
        std::cout << "Consumer: Receiving value"
                  << std::endl;
        consumer(queue);
    }
};

std::jthread producerThread(producerFunc, std::ref(queue));
std::jthread consumerThread(consumerFunc, std::ref(queue));
```

以下是执行此代码的示例输出：

```
Producer: Sending value 383
Consumer: Receiving value
Producer: Sending value 915
Result: 383
Consumer: Receiving value
Producer: Sending value 386
Result: 915
Consumer: Receiving value
Producer: Sending value 421
Result: 386
Consumer: Receiving value
Producer: Sending value 690
Result: 421
Consumer: Receiving value
Producer: Sending value 926
Producer: Sending value 426
Result: 690
Consumer: Receiving value
Producer: Sending value 211
```

```
Result: 926
Consumer: Receiving value
Result: 426
Consumer: Receiving value
Producer: Sending value 782
Producer: Sending value 862
Result: 211
Consumer: Receiving value
Result: 782
Consumer: Receiving value
Result: 862
```

使用像这样的生产者-消费者队列，消费者和生产者是分离的，并且它们的线程异步通信，从而允许生产者和消费者在另一方生成或处理值时的其他工作。

6.5. 总结

本章中，介绍了 Promise 和 Future，如何使用它们在单独的线程中执行异步代码，以及如何使用打包的任务运行可调用函数。这些对象和机制构成并实现了许多编程语言（包括 C++）使用的异步编程的关键概念。

还了解了为什么 Promise、Future 和打包任务不能复制，以及如何通过使用共享 Future 对象来共享 Future。

最后，展示了如何使用 Future、Promise 和打包任务来解决实际问题。

如果想更深入地探索 Promise 和 Future，值得一提的是一些第三方开源库，尤其是 Boost.Thread 和 Facebook Folly。这些库包含很多功能，包括回调、执行器和组合器。

下一章中，将学习一种使用 std::async 异步调用可调用函数的更简单的方法。

6.6. 扩展阅读

- Boost futures and promises: https://theboostcpplibraries.com/boost.thread-futures-and-promises
- Facebook Folly open source library: https://github.com/facebook/folly
- Futures for C++11 at Facebook: https://engineering.fb.com/2015/06/19/developer-tools/futuresfor-c-11-at-facebook
- 'Futures and Promises' ——Instagram 如何利用它来提高资源利用率：https://scaleyourapp.com/futures-and-promises-and-how-instagram-leverages-it/
- SeaStar：面向高性能服务器应用程序的开源 C++ 框架：https://seastar.io

第 7 章 异步函数

上一章中，介绍了 Promise、Future 和打包任务。在介绍打包任务时，提到 std::async 提供了一种更简单的方式来实现同样的结果，代码更少，更干净、更简洁。

异步函数（std::async）是一个异步运行可调用对象的函数模板，还可以通过传递一些定义启动策略的标志来选择执行方法。它是处理异步操作的强大工具，但其自动管理和对执行线程缺乏控制等，使其不适合某些需要细粒度控制或取消的任务。

在本章中，我们将讨论以下主要主题：

- 什么是异步函数以及如何使用？
- 有哪些不同的启动政策？
- 与以前的方法，特别是打包任务有什么不同？
- 使用 std::async 有哪些优点和缺点？
- 实际场景和示例

7.1. 技术要求

async 函数自 C++11 起可用，但一些示例使用了 C++14 中的功能，例如：chrono_literals 和 C++20 中的功能（例如 counting_semaphore），因此本章中展示的代码可以由支持 C++20 的编译器进行编译。

请查看第 3 章中的技术要求部分，以获取有关如何安装 GCC 13 和 Clang 8 编译器的指导。

可以在以下 GitHub 库中找到所有完整的代码：https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP

本章的示例位于 Chapter_07 文件夹下。所有源代码文件都可以使用 CMake 进行编译：

```
cmake . && cmake —build .
```

可执行二进制文件将在 bin 目录下生成。

7.2. 什么是 std::async？

std::async 是 C++ 中的一个函数模板，由 C++ 标准在 <future> 标头中引入，作为 C++11 线程支持库的一部分，用于异步运行函数，允许主线程（或其他线程）继续并发运行。

std::async 是 C++ 中用于异步编程的强大工具，可以更轻松地并行运行任务并有效地管理其结果。

7.2.1. 启动异步任务

要使用 std::async 异步执行函数，我们可以使用与第 3 章启动线程时相同的方法，但使用不同的可调用对象。

一种方法是使用函数指针：

```
void func() {
    std::cout << "Using function pointer\n";
}
auto fut1 = std::async(func);
```

另一种方法是使用 lambda 函数：

```
auto lambda_func = []() {
    std::cout << "Using lambda function\n";
};
auto fut2 = std::async(lambda_func);
```

还可以使用嵌入的 lambda 函数：

```
auto fut3 = std::async([]() {
    std::cout << "Using embedded lambda function\n";
});
```

可以使用 operator() 重载的函数对象：

```
class FuncObjectClass {
    public:
    void operator()() {
        std::cout << "Using function object class\n";
    }
};
auto fut4 = std::async(FuncObjectClass());
```

可以使用非静态成员函数，通过传递成员函数的地址和对象的地址来调用成员函数：

```
class Obj {
    public:
    void func() {
        std::cout << "Using a non-static member function"
        << std::endl;
    }
};
Obj obj;
auto fut5 = std::async(&Obj::func, &obj);
```

还可以使用静态成员函数，由于方法是静态的，只需要成员函数的地址：

```
class Obj {
    public:
    static void static_func() {
```

```
        std::cout << "Using a static member function"
        << std::endl;
    }
};
auto fut6 = std::async(&Obj::static_func);
```

当调用 std::async 时，会返回一个未来，其中会存储函数的结果。

7.2.2. 传递值

同样，与创建线程时传递参数类似，参数可以通过值、引用或指针传递给线程。
这里，可以看到如何通过值传递参数：

```
void funcByValue(const std::string& str, int val) {
    std::cout << "str: " << str << ", val: " << val
              << std::endl;
}
std::string str{"Passing by value"};
auto fut1 = async(funcByValue, str, 1);
```

按值传递表明可创建一个临时对象并将参数值复制到其中。这可以避免数据竞争，但成本更高。
下一个示例显示如何通过引用传递值：

```
void modifyValues(std::string& str) {
    str += " (Thread)";
}
std::string str{"Passing by reference"};
auto fut2 = std::async(modifyValues, std::ref(str));
```

还可以将值作为 const 引用传递：

```
void printVector(const std::vector<int>& v) {
    std::cout << "Vector: ";
    for (int num : v) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}
std::vector<int> v{1, 2, 3, 4, 5};
auto fut3 = std::async(printVector, std::cref(v));
```

引用传递通过使用 std::ref()（非常量引用）或 std::cref()（常量引用）来实现，这两个函数均定义在 <functional> 头文件中，让定义线程构造函数的可变参数模板（支持任意数量参数的类或函数模板）将参数视为引用。
还可以将对象移动到由 std::async 创建的线程中：

```
auto fut4 = std::async(printVector, std::move(v));
```

注意，向量数组 v 在移动后，处于有效的空状态。

最后，还可以通过 lambda 捕获传递值：

```
std::string str5{"Hello"};
auto fut5 = std::async([&]() {
    std::cout << "str: " << str5 << std::endl;
});
```

此示例中，std::async 执行的 lambda 函数作为引用访问了 str 变量。

7.2.3. 返回值

当调用 std:async 时，会立即返回一个 Future，将保存函数或可调用对象将计算的值。

前面的例子中，没有使用 std::async 返回的对象。这里重写第 6 章打包任务部分的最后一个示例，使用 std::packaged_task 对象来计算两个值的幂。本例中，将使用 std::async 生成几个异步任务来计算这些值，等待任务完成存储结果，最后在控制台中显示：

```
#include <chrono>
#include <cmath>
#include <future>
#include <iostream>
#include <thread>
#include <vector>
#include <syncstream>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

int compute(unsigned taskId, int x, int y) {
    std::this_thread::sleep_for(std::chrono::milliseconds(
                                rand() % 200));
    sync_cout << "Running task " << taskId << '\n';
    return std::pow(x, y);
}

int main() {
    std::vector<std::future<int>> futVec;
    for (int i = 0; i <= 10; i++)
        futVec.emplace_back(std::async(compute,
                            i+1, 2, i));

    sync_cout << "Waiting in main thread\n";
    std::this_thread::sleep_for(1s);
```

```
28
29      std::vector<int> results;
30      for (auto& fut : futVec)
31          results.push_back(fut.get());
32
33      for (auto& res : results)
34          std::cout << res << ' ';
35      std::cout << std::endl;
36      return 0;
37  }
```

compute() 函数仅获取两个数字 x 和 y，并计算 x^y 。需要获取一个代表任务标识符的数字，并等待最多两秒钟，然后在控制台中打印消息并计算结果。

在 main() 函数中，主线程启动几个任务来计算一系列 2 的幂值。调用 std::async 返回的 Future 存储在 futVec 向量中。主线程等待一秒钟后，模拟一些工作。最后，遍历 futVec 并在每个 Future 元素中调用 get() 函数，从而等待该特定任务完成并返回一个值，然后将返回的值存储在另一个名为 results 的向量数组中。然后，在退出程序之前输出 results 向量数组的内容。

这是运行该程序时的输出：

```
Waiting in main thread
Running task 11
Running task 9
Running task 2
Running task 8
Running task 4
Running task 6
Running task 10
Running task 3
Running task 1
Running task 7
Running task 5
1 2 4 8 16 32 64 128 256 512 1024
```

每个任务完成所需的时间不同，输出不是按任务标识符排序的。但当按顺序遍历 futVec 向量数组获取结果时，这些结果会按顺序显示。

现在，已经了解了如何启动异步任务并传递参数和返回值。接下来，介绍如何使用启动策略来控制执行方法。

7.3. 启动策略

除了在使用 std::async 函数时指定函数或可调用对象作为参数外，还可以指定启动策略。启动策略控制 std::async 如何安排异步任务的执行。这些在 <future> 头文件中定义。

调用 std::async 时，必须将启动策略指定为第一个参数。此参数的类型为 std::launch，

是一个位掩码值，其中的位控制允许的执行方法，可以是以下一个或多个枚举常量：

- `std::launch::async`：任务在单独的线程中执行。
- `std::launch::deferred`：通过在首次通过 `Future` 的 `get()` 或 `wait()` 方法请求结果时在调用线程中执行任务来启用惰性求值。对同一 `std::future` 的所有后续访问都将立即返回结果，所以只有在明确请求结果时才会执行任务，这可能会导致意外延迟。

如果未定义，则默认启动策略为 `std::launch::async | std::launch::deferred`。此外，实现可以提供其他启动策略。

默认情况下，C++ 标准规定 `std::async` 可以在异步或延迟模式下运行。

当指定多个标志时，行为实现定义，这取决于使用的编译器。如果指定了默认启动策略，标准建议使用可用的并发并推迟任务。

实现以下示例来测试不同的启动策略行为。首先，定义 `square()` 函数用作异步任务：

```
#include <chrono>
#include <future>
#include <iostream>
#include <string>
#include <syncstream>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

int square(const std::string& task_name, int x) {
    sync_cout << "Launching " << task_name
              << " task...\n";
    return x * x;
}
```

`main()` 函数中，程序通过启动三个不同的异步任务开始，一个使用 `std::launch::async` 启动策略，另一个使用 `std::launch::deferred` 启动策略，第三个任务使用默认启动策略：

```
sync_cout << "Starting main thread...\n";
auto fut_async = std::async(std::launch::async,
                        square, "async_policy", 2);
auto fut_deferred = std::async(std::launch::deferred,
                        square, "deferred_policy", 3);
auto fut_default = std::async(square,
                        "default_policy", 4);
```

如上一章所述，`wait_for()` 返回一个 `std::future_status` 对象，指示 `Future` 是否已准备就绪、已延迟或已超时。因此，可以使用该函数来检查返回的 `Future` 是否已延迟。通过 `lambda` 函数 `is_deferred()` 来做到这一点，该函数返回 `true`。预计至少一个 `Future` 对象 `fut_deferred` 将返回 `true`：

```cpp
auto is_deferred = [](std::future<int>& fut) {
    return (fut.wait_for(0s) ==
    std::future_status::deferred);
};

sync_cout << "Checking if deferred:\n";
sync_cout << " fut_async: " << std::boolalpha
          << is_deferred(fut_async) << '\n';
sync_cout << " fut_deferred: " << std::boolalpha
          << is_deferred(fut_deferred) << '\n';
sync_cout << " fut_default: " << std::boolalpha
          << is_deferred(fut_default) << '\n';
```

然后，主程序等待一秒钟，模拟一些处理，最后从异步任务中检索结果并输出其值：

```cpp
sync_cout << "Waiting in main thread...\n";
std::this_thread::sleep_for(1s);

sync_cout << "Wait in main thread finished.\n";
sync_cout << "Getting result from "
          << "async policy task...\n";
int val_async = fut_async.get();
sync_cout << "Result from async policy task: "
          << val_async << '\n';

sync_cout << "Getting result from "
          << "deferred policy task...\n";
int val_deferred = fut_deferred.get();
sync_cout << "Result from deferred policy task: "
          << val_deferred << '\n';
sync_cout << "Getting result from "
          << "default policy task...\n";
int val_default = fut_default.get();
sync_cout << "Result from default policy task: "
          << val_default << '\n';
```

这是运行上述代码的输出：

```
Starting main thread...
Launching async_policy task...
Launching default_policy task...
Checking if deferred:
  fut_async: false
  fut_deferred: true
  fut_default: false
Waiting in main thread...
Wait in main thread finished.
```

```
Getting result from async policy task...
Result from async policy task: 4
Getting result from deferred policy task...
Launching deferred_policy task...
Result from deferred policy task: 9
Getting result from default policy task...
Result from default policy task: 16
```

注意，使用默认和 std::launch::async 启动策略的任务是在主线程休眠时执行的，任务会在可以安排时立即启动；使用 std::launch::deferred 启动策略的延迟任务在请求值后开始执行。

接下来介绍如何处理异步任务中发生的异常。

7.4. 处理异常

使用 std::async 时不支持从异步任务到主线程的异常传播。为了启用异常传播，可能需要一个 Promise 对象来存储异常，稍后可以通过调用 std::async 返回的 Future 来访问该异常。但 std::async 无法访问或提供该 Promise 对象。

实现此目的的一种可行方法是使用 std::packaged_task 对象包装异步任务，应该直接使用上一章中描述的打包任务。

还可以使用自 C++11 起可用的嵌套异常，方法是使用 std::nested_exception，这是一个可以捕获和存储当前异常的多态混合类，允许任意类型的嵌套异常。从 std::nested_exception 对象中，可以使用 nested_ptr() 方法检索存储的异常，或通过调用 rethrow_nested() 重新抛出。

要创建嵌套异常，可以使用 std::throw_with_nested() 方法抛出异常。如果只想在异常嵌套时重新抛出异常，可以使用 std::rethrow_if_nested()。所有这些函数都在 <exception> 头文件中定义。

使用所有这些函数，可以实现以下示例，其中异步任务抛出 std::runtime_error 异常，该异常在异步任务的主体中捕获并作为嵌套异常重新抛出。然后，此嵌套异常对象在主函数中再次被捕获，并输出出异常序列：

```
#include <exception>
#include <future>
#include <iostream>
#include <stdexcept>
#include <string>

void print_exceptions(const std::exception& e,
                      int level = 1) {
    auto indent = std::string(2 * level, ' ');
    std::cerr << indent << e.what() << '\n';
    try {
        std::rethrow_if_nested(e);
```

```
    } catch (const std::exception& nestedException) {
        print_exceptions(nestedException, level + 1);
    } catch (...) { }
}

void func_throwing() {
    throw std::runtime_error(
    "Exception in func_throwing");
}

int main() {
    auto fut = std::async([]() {
        try {
            func_throwing();
        } catch (...) {
            std::throw_with_nested(
                std::runtime_error(
                    "Exception in async task."));
        }
    });

    try {
        fut.get();
    } catch (const std::exception& e) {
        std::cerr << "Caught exceptions:\n";
        print_exceptions(e);
    }
    return 0;
}
```

正如示例中看到的，创建了一个异步任务，该任务在 try-catch 块内执行 func_throwing() 函数。此函数仅抛出一个 std::runtime_error 异常，捕获该异常，然后通过 std::throw_with_nested() 函数作为 std::nested_exception 类的一部分重新抛出。稍后，在主线程中，尝试通过调用其 get() 方法从 fut future 对象中检索结果时，嵌套异常抛出并在主 try-catch 块中再次捕获，其中调用 print_exceptions()，并使用捕获的嵌套异常作为参数。

print_exceptions() 函数打印当前异常（e.what()）的原因，如果嵌套则重新抛出异常，再次捕获它并按嵌套级别缩进递归输出异常原因。

由于每个异步任务都有自己的 Future，程序可以分别处理来自多个任务的异常。

7.4.1. 调用 std::async 时发生异常

除了异步任务中发生的异常之外，还存在 std::async 可能抛出异常的情况：

- std::bad_alloc：如果没有足够的内存来存储 std::async 所需的内部数据结构。
- std:system_error：如果在使用 std::launch::async 作为启动策略时无法启动新线

程，错误条件将是 std::errc::resource_unavailable_try_again。根据实现，如果策略是默认策略，可能会回退到延迟调用或实现定义的策略。

大多数情况下，这些异常是由于资源耗尽而引发的。一种解决方案是，当当前正在运行的一些异步任务完成并释放其资源后，稍后重试。另一种更可靠的解决方案是限制在给定时间内运行的异步任务数量。

7.5. 异步 Future 和性能

std::async 返回的 Future 在调用其析构函数时的行为与从 Promise 获得的 Future 不同。当这些 Future 销毁时，会调用 ~future 析构函数，会执行 wait() 函数，使得创建时生成的线程汇入主线程。

如果 std::async 使用的线程尚未汇入，则会增加一些开销，从而影响程序性能，因此需要了解 Future 对象何时超出范围，从而调用其析构函数。

通过几个简短的例子来了解这些 Future 是如何表现的，以及如何使用的一些建议。

首先定义一个任务，func 只是将其输入值乘以 2，并等待一段时间，模拟一个昂贵的操作：

```
#include <chrono>
#include <functional>
#include <future>
#include <iostream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

unsigned func(unsigned x) {
    std::this_thread::sleep_for(10ms);
    return 2 * x;
}
```

为了测量代码块的性能，异步运行多个任务（在此示例中为 NUM_TASKS = 32），并使用 <chrono> 库中的稳定时钟测量运行时间。为此，只需使用以下命令记录表示任务启动当前时间点的时间点：

```
auto start = std::chrono::high_resolution_clock::now();
```

可以在 main() 函数中定义以下 lambda 函数，任务完成时调用该函数来获取持续时间（以毫秒为单位）：

```
auto duration_from = [](auto start) {
    auto dur = std::chrono::high_resolution_clock::now()
               - start;
```

```
    return std::chrono::duration_cast
                <std::chrono::milliseconds>(dur).count();
};
```

这个代码，就可以开始衡量未来的不同使用方法。

首先运行几个异步任务，但丢弃 std::async 返回的 Future：

```
constexpr unsigned NUM_TASKS = 32;

auto start = std::chrono::high_resolution_clock::now();

for (unsigned i = 0; i < NUM_TASKS; i++) {
    std::async(std::launch::async, func, i);
}

std::cout << "Discarding futures: "
          << duration_from(start) << '\n';
```

在我的测试 PC 上，此项测试的持续时间为 334 毫秒，我的测试 PC 是 Pentium i7 4790K，4 GHz，四核八线程。

对于下一个测试，存储返回的 Future，但不要等待结果准备好。显然，这不是通过产生异步任务来消耗资源，而非处理结果来使用计算机能力的正确方法。需要强调的是，这样做是为了测试和学习：

```
start = std::chrono::high_resolution_clock::now();

for (unsigned i = 0; i < NUM_TASKS; i++) {
    auto fut = std::async(std::launch::async, func, i);
}

std::cout << "In-place futures: "
          << duration_from(start) << '\n';
```

持续时间仍为 334 毫秒。这两种情况下，会创建一个 Future，当在每次循环迭代结束时超出范围时，必须等待 std::async 生成的线程完成并加入。

这里启动了 32 个任务，每个任务至少耗时 10 毫秒。总计 320 毫秒，相当于这些测试中获得的 334 毫秒。其余性能成本来自启动线程、检查 for 循环变量、存储使用稳定时钟时的时间点等。

为了避免每次调用 std::async 时都创建一个新的 Future 对象，并等待其析构函数被调用。重用 Future 对象，如下面的代码所示。同样，这不是正确的方法，放弃了对先前任务结果的访问：

```
std::future<unsigned> fut;
start = std::chrono::high_resolution_clock::now();
```

```
for (unsigned i = 0; i < NUM_TASKS; i++) {
    fut = std::async(std::launch::async, func, i);
}

std::cout << "Reusing future: "
          << duration_from(start) << '\n';
```

现在持续时间为 166 毫秒，时间的减少是因为不必等待每个 Future，所以这些 Future 也不会销毁。

但这并不理想，我们可能想知道异步任务的结果。因此，需要将结果存储在一个向量数组中。修改前面的示例，使用 res 向量数组来存储每个任务的结果：

```
std::vector<unsigned> res;

start = std::chrono::high_resolution_clock::now();

for (unsigned i = 0; i < NUM_TASKS; i++) {
    auto fut = std::async(std::launch::async, func, i);
    res.push_back(fut.get());
}

std::cout << "Reused future and storing results: "
          << duration_from(start) << '\n';
```

持续时间仍为 334 毫秒。这两种情况下，都会创建一个 Future，当在每次循环迭代结束时超出范围时，必须等待 std::async 生成的线程完成并加入。

这里启动了 32 个任务，每个任务至少耗时 10 毫秒。总计 320 毫秒，相当于这些测试中获得的 334 毫秒。其余性能成本来自启动线程、检查 for 循环变量、存储使用稳定时钟时的时间点等。

为了避免每次调用 std::async 时都创建一个新的 Future 对象，并等待调用其析构函数。这里重用 Future 对象，如下面的代码所示。同样，这不是正确的方法，放弃了对先前任务结果的访问：

```
std::vector<unsigned> res;
std::vector<std::future<unsigned>> futsVec;

start = std::chrono::high_resolution_clock::now();

for (unsigned i = 0; i < NUM_TASKS; i++) {
    futsVec.emplace_back(std::async(std::launch::async,
                         func, i));
}

for (unsigned i = 0; i < NUM_TASKS; i++) {
    res.push_back( futsVec[i].get() );
```

```
}

std::cout << "Futures vector and storing results: "
          << duration_from(start) << '\n';
```

现在持续时间只有 22 毫秒！但这是为什么呢?

现在，所有任务都真正异步运行。第一个循环启动所有任务并将 Future 存储在 futsVec 向量数组中。由于调用 Future 析构函数，因此不再会有等待期。

第二个循环遍历 futsVec，检索每个结果，并将其存储在结果向量 res 中。执行第二个循环的时间大约是遍历 res 向量所需的时间加上最慢任务的调度和执行时间。

如果测试运行的系统有足够的线程来同时运行所有异步任务，则运行时间可以减半。有些系统可以通过让调度程序决定运行哪些任务来自动管理后台的多个异步任务。其他系统中，当尝试同时启动多个线程时，可能会通过引发异常来发出抱怨。下一节中，将使用信号量实现线程限制器。

7.6. 限制线程数

如果没有足够的线程来运行多个 std::async 调用，则会引发 std::runtime_system 异常并指示资源耗尽。

可以通过使用计数信号量（std::counting_semaphore）创建线程限制器来实现一个简单的解决方案，这是第 4 章中解释的多线程同步机制。

这里使用一个 std::counting_semaphore 对象，将其初始值设置为系统允许的最大并发任务数，可以通过调用 std::thread::hardware_concurrency() 函数来检索，然后在任务函数中使用该信号量来阻止异步任务的总数超过最大并发任务数。

以下代码实现了这个想法:

```
#include <chrono>
#include <future>
#include <iostream>
#include <semaphore>
#include <syncstream>
#include <vector>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

void task(int id, std::counting_semaphore<>& sem) {
    sem.acquire();

    sync_cout << "Running task " << id << "...\n";
    std::this_thread::sleep_for(1s);

    sem.release();
```

```
}

int main() {
    const int total_tasks = 20;
    const int max_concurrent_tasks =
              std::thread::hardware_concurrency();

    std::counting_semaphore<> sem(max_concurrent_tasks);

    sync_cout << "Allowing only "
              << max_concurrent_tasks
              << " concurrent tasks to run "
              << total_tasks << " tasks.\n";

    std::vector<std::future<void>> futures;
    for (int i = 0; i < total_tasks; ++i) {
        futures.push_back(
            std::async(std::launch::async,
                task, i, std::ref(sem)));
    }

    for (auto& fut : futures) {
        fut.get();
    }
    std::cout << "All tasks completed." << std::endl;
    return 0;
}
```

该程序首先设置将要启动的任务总数。然后，创建一个计数信号量 `sem`，并将其初始值设置为硬件并发值。最后，启动所有任务并等待其 `Future` 准备就绪。

本例的关键点是，每个任务在执行其工作之前都会获取信号量，从而减少内部计数器或阻塞直到计数器可以减少。当工作完成后，释放信号量，这会增加内部计数器并解除阻塞，此时尝试获取信号量的其他任务。所以，只有当存在是用于该任务的空闲硬件线程。否则，线程将阻塞，直到另一个任务释放信号量。

探讨一些实际场景之前，先了解一下使用 `std::async` 的缺点。

7.7. 何时不应使用 std::async

`std::async` 不提供对所用线程数或对线程对象本身的访问的直接控制。现在，了解了如何通过使用计数信号量来限制异步任务的数量，但在某些应用程序中，这可能不是最佳解决方案，需要细粒度的控制。

此外，线程的自动管理可能会引入开销，从而降低性能，尤其是在启动许多小任务时，会导致过多的上下文切换和资源争用。

该实现对可使用的并发线程数施加了一些限制，这可能会降低性能甚至引发异常。由于 `std::async` 和可用的 `std::launch` 策略依赖于实现，因此不同编译器和平台之间的性能并不一致。

最后，没有提到如何取消由 `std::async` 启动的异步任务，因为在完成之前没有标准的方法。

7.8. 实例

现在是时候使用 `std::async` 实现一些示例来处理实际场景了。我们将介绍如何执行以下操作：

- 执行并行计算和聚合
- 跨不同容器或大型数据集异步搜索
- 异步乘以两个矩阵
- 链式异步操作
- 改进上一章的管道示例

7.8.1. 并行计算和聚合

数据聚合是从多个来源收集原始数据并组织、处理和提供数据摘要以方便使用的过程。此过程在许多领域都很有用，例如：业务报告、金融服务、医疗保健、社交媒体监控、研究和学术领域。

举一个简单的例子，计算 `1` 到 `n` 之间的所有数字的平方，并得出它们的平均值。使用以下公式可以计算平方值的总和会更快，并且需要更少的计算机能力。此外，任务可能更有意义，但本示例的目的是了解任务之间的关系，而不是任务本身。

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)\frac{n(n+1)(2n+1)}{6}(2n+1)}{6}$$

以下示例中的 `average_squares()` 函数针对 `1` 到 `n` 之间的每个值启动一个异步任务来计算平方值。生成的 `Future` 对象存储在 `futsVec` 向量中，稍后 `sum_results()` 函数会使用该向量计算平方值的总和。然后，将结果除以 `n` 以获得平均值：

```
#include <future>
#include <iomanip>
#include <iostream>
#include <vector>

int square(int x) {
    return x * x;
}

int sum_results(std::vector<std::future<int>>& futsVec) {
    int sum = 0;
    for (auto& fut : futsVec) {
        sum += fut.get();
    }
    return sum;
}

```

```
int average_squares(int n) {
    std::vector<std::future<int>> futsVec;
    for (int i = 1; i <= n; ++i) {
        futsVec.push_back(std::async(
        std::launch::async, square, i));
    }
    return double(sum_results(futures)) / n;
}

int main() {
    int N = 100;
    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Sum of squares for N = " << N
              << " is " << average_squares(N) << '\n';
    return 0;
}
```

例如，对于 n = 100 ，可以检查该值是否与函数除以 n 返回的值相同，即 3,383.50。

可以轻松修改此示例，以使用 MapReduce 编程模型实现解决方案，从而高效处理大型数据集。MapReduce 的工作原理是将数据处理分为两个阶段：Map 阶段，独立的数据块在多台计算机上并行过滤、排序和处理；Reduce 阶段，汇总 Map 阶段的结果，汇总数据。这就像我们刚刚实现的一样，在 Map 阶段使用 square() 函数，在 Reduce 阶段使用 average_squares() 和 sum_results() 函数。

7.8.2. 异步搜索

加快在大型容器中搜索目标值的一种方法是并行化搜索。接下来，我们将介绍两个示例。

第一个示例涉及跨不同容器进行搜索，每个容器使用一个任务；第二个示例涉及跨大型容器进行搜索，将其划分为较小的部分，每个部分使用一个任务。

跨不同容器搜索

这个例子中，需要在包含动物名称的不同类型的容器（vector、list、forward_list 和 set）中搜索目标值：

```
#include <algorithm>
#include <forward_list>
#include <future>
#include <iostream>
#include <list>
#include <set>
#include <string>
#include <vector>

int main() {
    std::vector<std::string> africanAnimals =
```

```
                {"elephant", "giraffe", "lion", "zebra"};
    std::list<std::string> americanAnimals =
                {"alligator", "bear", "eagle", "puma"};
    std::forward_list<std::string> asianAnimals =
                {"orangutan", "panda", "tapir", "tiger"};
    std::set<std::string> europeanAnimals =
                {"deer", "hedgehog", "linx", "wolf"};

    std::string target = "elephant";
    /* .... */
}
```

为了搜索目标值，使用 `search()` 模板函数为每个容器启动一个异步任务，该函数只是调用容器中的 `std::find` 函数，如果找到目标值则返回 `true`，否则返回 `false`：

```
template <typename C>
bool search(const C& container, const std::string& target) {
    return std::find(container.begin(), container.end(),
                     target) != container.end();
}
```

这些异步任务是使用带有 `std::launch::async` 启动策略的 `std::async` 函数启动的：

```
int main() {
    /* .... */
    auto fut1 = std::async(std::launch::async,
                search<std::vector<std::string>>,
                africanAnimals, target);
    auto fut2 = std::async(std::launch::async,
                search<std::list<std::string>>,
                americanAnimals, target);
    auto fut3 = std::async(std::launch::async,
                search<std::forward_list<std::string>>,
                asianAnimals, target);
    auto fut4 = std::async(std::launch::async,
                search<std::set<std::string>>,
                europeanAnimals, target);
    /* .... */
```

最后，只需从使用 `std::async` 创建的未来中检索所有返回值并对其进行按位或处理：

```
int main() {
    /* .... */
    bool found = fut1.get() || fut2.get() ||
    fut3.get() || fut4.get();
    if (found) {
        std::cout << target
```

```
                  << " found in one of the containers.\n";
    } else {
        std::cout << target
                  << " not found in any of "
                  << "the containers.\n";
    }

    return 0;
}
```

这个例子还展示了标准模板库（STL）的强大功能，提供了可应用于不同容器和数据类型的通用和可重用的算法。

在大型容器中搜索

下一个例子中，将实现一个解决方案，在包含 500 万个整数值的大向量中找到目标值。

为了生成向量，使用具有均匀整数分布的随机数生成器：

```
#include <cmath>
#include <iostream>
#include <vector>
#include <future>
#include <algorithm>
#include <random>

// Generate a large vector of random integers using a uniform
distribution
std::vector<int> generate_vector(size_t size) {
    std::vector<int> vec(size);
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dist(1, size);
    std::generate(vec.begin(), vec.end(), [&]() {
        return dist(gen);
    });
    return vec;
}
```

为了在向量的某个段中搜索目标值，可以使用 std::find 函数，其中 begin 和 end 迭代器指向段限制：

```
bool search_segment(const std::vector<int>& vec, int target, size_t
begin, size_t end) {
    auto begin_it = vec.begin() + begin;
    auto end_it = vec.begin() + end;
    return std::find(begin_it, end_it, target) != end_it;
}
```

main() 函数中，首先使用 generate_vector() 函数生成大向量，然后定义要查找的目标值，以及向量将拆分为并行搜索的段数（num_segments）：

```
const int target = 100;

std::vector<int> vec = generate_vector(5000000);
auto vec_size = vec.size();

size_t num_segments = 16;
size_t segment_size = vec.size() / num_segments;
```

然后，对于每个段，定义其开始和结束迭代器，并启动一个异步任务来搜索该段中的目标值。因此，使用 std::async 和 std::launch::async 启动策略在单独的线程中异步执行 search_segment。为了避免在将其作为 search_segment 的输入参数传递时复制大型向量，使用常量引用 std::cref。std::async 返回的 Future 存储在 futs 向量中：

```
std::vector<std::future<bool>> futs;
for (size_t i = 0; i < num_segments; ++i) {
    auto begin = std::min(i * segment_size, vec_size);
    auto end = std::min((i + 1) * segment_size, vec_size);
    futs.push_back( std::async(std::launch::async,
                               search_segment,
                               std::cref(vec),
                               target, begin, end) );
}
```

注意，向量大小并不总是段大小的倍数，因此最后一个段可能比其他段短。为了处理这种情况并避免在检查最后一个段时访问越界内存时出现问题，需要正确设置每个段的开始和结束索引。为此，使用 std::min 来获取向量大小与当前段中最后一个元素的假设索引之间的最小值。

最后，通过在每个未来上调用 get() 来检查所有结果，如果在段中找到目标值，则将消息输出到控制台：

```
bool found = false;
for (auto& fut : futs) {
    if (fut.get()) {
        found = true;
        break;
    }
}

if (found) {
    std::cout << "Target " << target
              << " found in the large vector.\n";
} else {
    std::cout << "Target " << target
              << " not found in the large vector.\n";
```

```
}
```

该解决方案可作为处理分布式系统中的海量数据集的更高级解决方案的基础，其中每个异步任务都尝试在特定机器或集群中找到目标值。

7.8.3. 异步矩阵乘法

矩阵乘法是计算机科学中最相关的运算之一，用于计算机图形学、计算机视觉、机器学习和科学计算等许多领域。

下面的示例中，将通过将计算分布在多个线程中来实现并行计算解决方案。

首先定义一个矩阵类型 `matrix_t`，作为保存整数值的向量的向量：

```
#include <cmath>
#include <exception>
#include <future>
#include <iostream>
#include <vector>

using matrix_t = std::vector<std::vector<int>>;
```

然后，实现 `matrix_multiply` 函数，该函数接受两个矩阵 `A` 和 `B`，将其作为常量引用传递，并返回其乘积。如果 `A` 是一个 `mxn` 矩阵（`m` 代表行，`n` 代表列），`B` 是一个 `pxq` 矩阵，如果 `n = p`，可以将 `A` 和 `B` 相乘，得到的矩阵尺寸为 `mxq`（`m` 行和 `q` 列）。

`matrix_multiply` 函数首先为结果矩阵 `res` 保留一些空间。然后，循环遍历矩阵，从 `B` 中提取第 `j` 列并将其乘以 `A` 中的第 `i` 行：

```
matrix_t matrix_multiply(const matrix_t& A,
                         const matrix_t& B) {
    if (A[0].size() != B.size()) {
        throw new std::runtime_error(
                  "Wrong matrices dimmensions.");
    }
    size_t rows = A.size();
    size_t cols = B[0].size();
    size_t inner_dim = B.size();
    matrix_t res(rows, std::vector<int>(cols, 0));
    std::vector<std::future<int>> futs;
    for (auto i = 0; i < rows; ++i) {
        for (auto j = 0; j < cols; ++j) {
            std::vector<int> column(inner_dim);
            for (size_t k = 0; k < inner_dim; ++k) {
                column[k] = B[k][j];
            }
            futs.push_back(std::async(std::launch::async,
                                      dot_product,
```

```
                                        A[i], column));
        }
    }
    for (auto i = 0; i < rows; ++i) {
        for (auto j = 0; j < cols; ++j) {
            res[i][j] = futs[i * cols + j].get();
        }
    }
    return res;
}
```

通过使用 std::async 和 std::launch::async 启动策略，运行 dot_product 函数，异步完成乘法。std::async 返回的每个 Future 都存储在 futs 向量中。dot_product 函数计算向量 a 和 b 的点积，表示来自 A 的一行和来自 B 的一列，方法是将元素逐个相乘并返回这些乘积的总和：

```
int dot_product(const std::vector<int>& a,
                const std::vector<int>& b) {
    int sum = 0;
    for (size_t i = 0; i < a.size(); ++i) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

由于 dot_product 函数需要两个向量，因此需要在启动每个异步任务之前从 B 中提取每一列。这还可以提高整体性能，因为向量可能存储在连续的内存块中，从而在计算过程中更有利于缓存。

在 main() 函数中，仅定义两个矩阵 A 和 B，并使用 matrix_multipy 函数计算其乘积。所有矩阵都使用 show_matrix lambda 函数输出：

```
int main() {
    auto show_matrix = [](const std::string& name,
                          matrix_t& mtx) {
        std::cout << name << '\n';
        for (const auto& row : mtx) {
            for (const auto& elem : row) {
                std::cout << elem << " ";
            }
            std::cout << '\n';
        }
        std::cout << std::endl;
    };
    matrix_t A = {{1, 2, 3},
                  {4, 5, 6}};

```

```
    matrix_t B = {{7, 8, 9},
                  {10, 11, 12},
                  {13, 14, 15}};

    auto res = matrix_multiply(A, B);

    show_matrix("A", A);
    show_matrix("B", B);
    show_matrix("Result", res);

    return 0;
}
```

这是运行此示例的输出：

```
A
1 2 3
4 5 6

B
7 8 9
10 11 12
13 14 15

Result
66 72 78
156 171 186
```

使用连续的内存块可以提高遍历向量时的性能，可以一次将许多元素读入缓存。使用 `std::vector` 时不能保证使用连续的内存分配，最好使用 `new` 或 `malloc`。

7.8.4. 链式异步操作

这个例子中，将实现一个由三个阶段组成的简单管道，每个阶段都获取前一个阶段的结果并计算一个值。

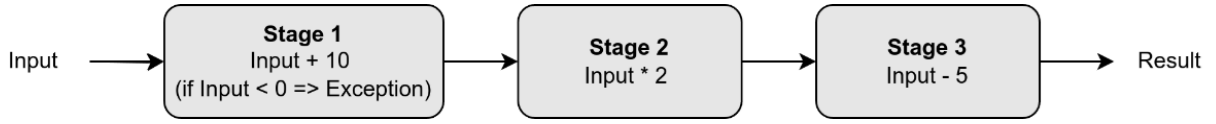

图 7.1 –简单管道示例

第一阶段仅接受正整数作为输入，否则会引发异常，并在返回结果之前将该值加 10。第二阶段将其输入乘以 2，第三阶段从其输入中减去 5：

```
#include <future>
#include <iostream>
```

```
#include <stdexcept>

int stage1(int x) {
    if (x < 0) throw std::runtime_error(
    "Negative input not allowed");
    return x + 10;
}

int stage2(int x) {
    return x * 2;
}

int stage3(int x) {
    return x - 5;
}
```

在 main() 函数中，对于中间阶段和最终阶段，通过使用前几个阶段生成的 Future 作为输入来定义管道。这些 Future 通过引用传递给运行异步代码的 lambda 表达式，其中使用 get() 函数来获取其结果。

要从管道中检索结果，只需从上一个阶段返回的 Future 中调用 get() 函数即可。如果发生异常，例如，当 input_value 为负数时，将被 try-catch 块捕获：

```
int main() {
    int input_value = 5;
    try {
        auto fut1 = std::async(std::launch::async,
                               stage1, input_value);

        auto fut2 = std::async(std::launch::async,
                               [&fut1]() {
                                   return stage2(fut1.get()); });

        auto fut3 = std::async(std::launch::async,
                               [&fut2]() {
                                   return stage3(fut2.get()); });

        int final_result = fut3.get();
        std::cout << "Final Result: "
                  << final_result << std::endl;

    } catch (const std::exception &ex) {
        std::cerr << "Exception caught: "
                  << ex.what() << std::endl;
    }
    return 0;
}
```

本例中定义的管道很简单，每个阶段都使用前一个阶段的 Future 来获取输入值并产生结果。下一个示例中，将使用 std:async 和延迟启动策略重写上一章中实现的管道，以便仅执行所需的阶段。

7.8.5. 异步管道

当实现管道时，提到可以使用具有延迟执行的 Future 将不同的任务保持关闭状态，直到需要时才执行，这在计算成本很高但结果可能并不总是需要的情况下很有用。由于只能使用 std::async 创建具有延迟状态的 Future ，现在是时候看看如何做到这一点了。

我们将实现上一章中描述的相同管道，其遵循下一个任务图：

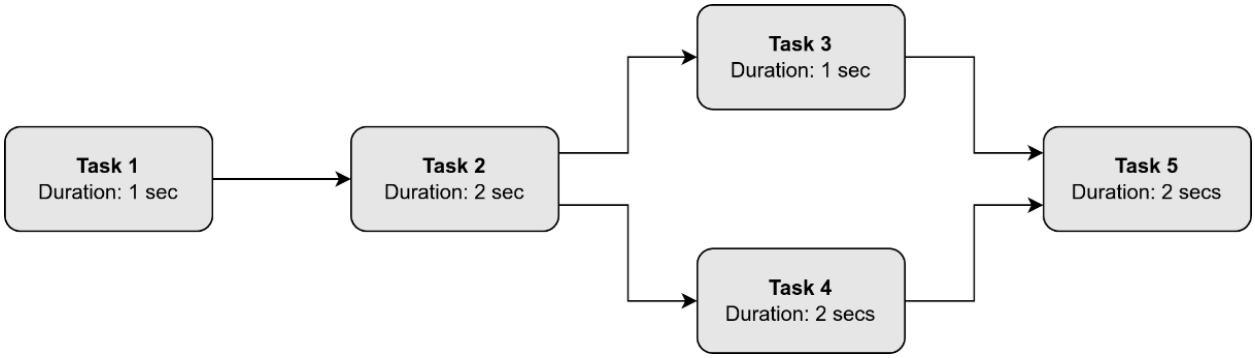

图 7.2 –管道示例

首先定义 Task 类，类似于上一章示例中实现的类，但使用 std::async 函数并存储返回的 Future，而不是之前使用的 Promise。这里，只会评论该示例中的相关代码更改，因此请查看该示例以了解 Task 类的完整说明，或在 GitHub 库中查到。

任务构造函数存储任务标识符 （id_）、要启动的函数 （func_） 以及任务是否具有依赖项 (has_d ependency_), 还使用 std::async 和 std::launch::deferred 启动策略以延迟启动模式启动异步任务，所以任务已创建但直到需要时才启动。返回的 Future 存储在 fut_ 变量中：

```
template <typename Func>
class Task {
    public:
    Task(int id, Func& func)
        : id_(id), func_(func), has_dependency_(false) {
        sync_cout << "Task " << id
                  << " constructed without dependencies.\n";
        fut_ = std::async(std::launch::deferred,
        [this](){ (*this)(); });
    }
    template <typename... Futures>
    Task(int id, Func& func, Futures&&... futures)
        : id_(id), func_(func), has_dependency_(true) {
        sync_cout << "Task " << id
                  << " constructed with dependencies.\n";
```

```
        fut_ = std::async(std::launch::deferred,
                          [this](){ (*this)(); });
        add_dependencies(std::forward<Futures>
                                    (futures)...);
    }

private:
    int id_;
    Func& func_;
    std::future<void> fut_;
    std::vector<std::shared_future<void>> deps_;
    bool has_dependency_;
};
```

由 std::async 启动的异步任务会调用自身实例（this 对象）的 operator()。发生这种情况时，将调用 wait_completion()，通过 valid() 函数检查共享 Future 向量 deps_ 中存储依赖任务的所有元素是否有效。如果都有效，则通过调用 get() 函数等待它们完成。当所有依赖任务都完成后，将调用 func_ 函数：

```
public:
void operator()() {
    sync_cout << "Starting task " << id_ << std::endl;

    wait_completion();

    sync_cout << "Running task " << id_ << std::endl;
    func_();
}
private:
void wait_completion() {
    sync_cout << "Waiting completion for task "
              << id_ << std::endl;
    if (!deps_.empty()) {
        for (auto& fut : deps_) {
            if (fut.valid()) {
                sync_cout << "Fut valid so getting "
                          << "value in task " << id_
                          << std::endl;
                fut.get();
            }
        }
    }
}
```

还有一个新的成员函数 start()，在调用 std::async 时等待在任务构造期间创建的 fut_future 。这将用于通过请求最后一个任务的结果来触发管道：

```
public:
void start() {
    fut_.get();
}
```

与上一章中的示例一样，还定义了一个名为 get_dependency() 的成员函数，返回由 fut_ 构造的共享 Future：

```
std::shared_future<void> get_dependency() {
    sync_cout << "Getting future from task "
              << id_ << std::endl;
    return fut_;
}
```

main() 函数中,通过链接任务对象并设置其依赖关系,以及要运行的 lambda 函数 sleep1s 或 sleep2s 来定义管道，可按照图 7.2 所示的图表进行操作：

```
int main() {
    auto sleep1s = [](){
        std::this_thread::sleep_for(1s);
    };
    auto sleep2s = [](){
        std::this_thread::sleep_for(2s);
    };
    Task task1(1, sleep1s);
    Task task2(2, sleep2s, task1.get_dependency());
    Task task3(3, sleep1s, task2.get_dependency());
    Task task4(4, sleep2s, task2.get_dependency());
    Task task5(5, sleep2s, task3.get_dependency(),
               task4.get_dependency());

    sync_cout << "Starting the pipeline..." << std::endl;
    task5.start();

    sync_cout << "All done!" << std::endl;
    return 0;
}
```

启动管道非常简单，只需从上一个任务的 Future 获取结果即可。可以通过调用 task5 的 start() 方法来做到这一点，这将使用依赖向量递归调用其依赖任务并启动延迟的异步任务。

这是执行上述代码的输出：

```
Task 1 constructed without dependencies.
Getting future from task 1
Task 2 constructed with dependencies.
Getting future from task 2
```

```
Task 3 constructed with dependencies.
Getting future from task 2
Task 4 constructed with dependencies.
Getting future from task 4
Getting future from task 3
Task 5 constructed with dependencies.
Starting the pipeline...
Starting task 5
Waiting completion for task 5
Fut valid so getting value in task 5
Starting task 3
Waiting completion for task 3
Fut valid so getting value in task 3
Starting task 2
Waiting completion for task 2
Fut valid so getting value in task 2
Starting task 1
Waiting completion for task 1
Running task 1
Running task 2
Running task 3
Fut valid so getting value in task 5
Starting task 4
Waiting completion for task 4
Running task 4
Running task 5
All done!
```

可以通过调用每个任务的构造函数，并从先前的依赖任务中获取 Future 来了解如何创建管道。

然后，当触发管道时，将启动 task5，并递归启动 task3、task2 和 task1。由于 task1 没有依赖项，因此不需要等何其他任务运行其工作。当其已完成，则进行 task2 完成，然后是 task3。

接下来，task5 继续检查其依赖任务，因此现在轮到 task4 运行了。由于 task4 的所有依赖任务都已完成，因此 task4 只需执行，然后 task5 即可运行，从而完成管道。

可以通过执行实际计算并在任务之间传输结果来改进此示例。此外，除了延迟任务，可以考虑具有多个并行步骤的阶段，这些步骤可以在单独的线程中计算。可以将这些改进作为练习来实现。

7.9. 总结

本章中，介绍了 std::async，如何使用该函数执行异步任务，如何使用启动策略定义其行为，以及如何处理异常。

现在还了解了异步函数返回的 Future 如何影响性能以及如何明智地使用它们，还了解了如何使用计数信号量通过系统中可用的线程数来限制异步任务的数量。

还提到了一些场景，其中 std::async 可能不是完成这项工作的最佳工具。

最后，实现了几个涵盖实际场景的示例，这对于并行化许多常见任务很有用。

通过本章获得的所有知识，现在了解了何时（以及何时不）使用 std::async 函数并行运行异步任务，从而提高应用程序的整体性能，实现更好的计算机资源利用率，并减少资源耗尽。

下一章中，将介绍如何使用自 C++20 以来就可用的协程来实现异步执行。

7.10. 扩展阅读

- Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14, Scott Meyers, O'Reilly Media, Inc., 1st Edition –Chapter 7, Item 35 and Item 36
- std::async: https://en.cppreference.com/w/cpp/thread/async
- std::launch: https://en.cppreference.com/w/cpp/thread/launch
- Strassen algorithm: https://en.wikipedia.org/wiki/Strassen_algorithm
- Karatsuba algorithm: https://en.wikipedia.org/wiki/Karatsuba_algorithm
- OpenBLAS: https://www.openblas.net
- BLIS library: https://github.com/flame/blis
- MapReduce: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

第 8 章 使用协程

前面的章节中，了解了在 C++ 中编写异步代码的不同方法。使用了线程（基本执行单元）和一些更高级的异步代码机制，例如 Future、Promise 和 std::async。将在下一章中介绍 Boost.Asio 库。所有这些方法通常使用由内核创建和管理的多个系统线程。

例如，程序的主线程可能需要访问数据库。这种访问可能很慢，因此会在另一个线程中读取数据，以便主线程可以继续执行其他任务。另一个示例是“生产者-消费者”模型，其中一个或多个线程生成要处理的数据项，并且一个或多个线程以完全异步的方式处理这些项。

上述两个示例都使用了线程（也称为系统（内核）线程），并且需要不同的执行单元，每个线程一个。

本章中，将研究一种编写异步代码的另一种方式——协程。协程是 20 世纪 50 年代末的一个古老概念，直到最近才添加到 C++(C++20) 中。协程不需要单独的线程（当然，可以让不同的线程运行协程），而是一种机制，允许在单个线程中执行多项任务。

本章中，将讨论以下主要主题：

- 什么是协同程序以及 C++ 如何实现和支持情况?
- 实现基本协程，了解 C++ 协程的要求
- 生成器协程和新的 C++23 std::generator
- 用于解析整数的字符串解析器
- 协程中的异常

本章介绍不使用任何第三方库实现的 C++ 协程。这种编写协程的方式相当底层，需要编写代码来支持。

8.1. 技术要求

对于本章，需要一个 C++20 编译器。对于生成器示例，需要一个 C++23 编译器。

已经使用 GCC 14.1 测试了这些示例。代码与平台无关，因此即使本书以 Linux 为重点，所有示例也应该可以在 macOS 和 Windows 上运行。请注意，Visual Studio 17.11 尚不支持 C++23 std::generator。

本章的代码可以在本书的 GitHub 库中找到：https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP。

8.2. 协程

开始 C++ 中实现协程之前，将从概念上介绍协程，并了解其在程序中有何用处。

从定义开始。协程是一种可以自行暂停的函数，在等待输入值时（暂停时不会执行）或在产生值（例如计算结果）后自行暂停。当输入值可用或调用者请求另一个值，协程就会恢复执行。我们很快会回到 C++ 中的协程，通过一个实际例子来了解协程的工作原理。

想象一下有人在做助理，一天的开始就是阅读电子邮件。其中一封电子邮件是一份报告请求。阅读完电子邮件后，开始撰写所要求的文档。写完介绍段落后，注意到需要同事的另一份报告来获

取上一季度的一些会计结果。停止撰写报告，给同事写一封电子邮件请求所需的信息，然后阅读下一封电子邮件，这是一份预订下午重要会议会议室的请求。打开公司开发的用于自动预订会议室的特殊应用程序，以优化会议室的使用并预订会议室。

过了一会儿，又从同事那里收到所需的会计数据并继续撰写报告。

助理总是忙于完成自己的任务。编写报告就是协程的一个很好的例子：开始编写报告，然后在等待所需信息时暂停写作，当信息到达，他们就会继续写作。当然，助理不想浪费时间，在等待期间，也会执行其他任务。如果同事等待请求然后发送适当的响应，则可以将其视为另一个协程。

现在回到软件上。假设需要编写一个函数，在处理一些输入信息后将数据存储在数据库中。

如果数据一次性全部到达，可以只实现一个函数。该函数将读取输入，对其进行所需的处理，最后将结果写入数据库。但如果数据要处理的数据以块的形式到达，并且处理每个块都需要前一个块处理的结果（为了这个例子，假设第一个块处理只需要一些默认值）？解决问题的一个可能方法是让函数等待每个数据块，处理它后将结果存储在数据库中，然后等待下一个数据块，依此类推。但如果这样做，可能会在等待每个数据块到达时浪费大量时间。

阅读完前面的章节后，可能会考虑不同的潜在解决方案：可以创建一个线程来读取数据，将块复制到队列，然后第二个线程（可能是主线程）将处理数据。这是一个可接受的解决方案，但使用多个线程可能有点过头了。

另一个解决方案可能是实现一个函数来仅处理一个块。调用者将等待输入传递给函数，并保留处理每个数据块所需的前一个块处理的结果。这个解决方案中，必须将数据处理函数所需的状态保存在另一个函数中。对于一个简单的示例来说，这可能是可以接受的，但是当处理变得更加复杂（例如，需要保留具有不同中间结果的多个步骤），代码可能很难理解和维护。

可以使用协程来解决这个问题。来看一些可能的协程伪代码，以块的形式处理数据并保留中间结果：

```
processing_result process_data(data_block data) {
    while (do_processing == true) {
        result_type result{ 0 };
        result = process_data_block(previous_result);
        update_database();
        yield result;
    }
}
```

上述协程从调用者处接收一个数据块，执行所有处理，更新数据库，并保存处理下一个块所需的结果。将结果交给调用者后（稍后将详细介绍如何交出结果），其会自行暂停。当调用者再次调用协程并请求处理新数据块时，协程将恢复执行。

这样的协同程序简化了状态管理，可以在调用之间保持状态。

对协程进行概念性介绍之后，将开始在 C++20 中实现它们。

8.3. C++ 协程

协程只是函数，但与我们习惯的函数不同。其具有特殊的属性，我们将在本章中研究这些属性。本节中，将重点介绍 C++ 中的协程。

函数在调用时开始执行，并且通常以返回语句或刚到达函数末尾时终止。

函数从头到尾运行，可能会调用另一个函数（如果是递归函数，则甚至会调用自身），也可能会抛出异常或有不同的返回点，但它总是从头到尾运行。

协程则不同。协程是一个可以暂停自身的函数。协程的流程可能类似于以下伪代码：

```
void coroutine() {
    do_something();
    co_yield;
    do_something_else();
    co_yield;
    do_more_work();
    co_return;
}
```

我们很快就会看到带有 co_ 前缀的术语的含义。

对于协程，需要一种机制来保持执行状态，以便能够暂停/恢复协程。这是由编译器完成的，但必须编写一些辅助代码，让编译器协助实现。

C++ 中的协程是无堆栈的,所以需要存储的状态才能暂停/恢复协程,存储在调用 new/delete 来分配/释放动态内存的堆中。这些调用由编译器创建。

8.3.1. 新关键字

因为协程本质上是一个函数（具有一些特殊属性，但仍然是一个函数），所以编译器需要某种方法来知道给定函数是否是协程。C++20 引入了三个新关键字：co_yield、co_await 和 co_return。如果一个函数至少使用这三个关键字中的一个，那么编译器就知道它是一个协程。

下表总结了新关键字的功能：

关键字	输入/输出	协程状态
co_yield	输出	暂停
co_await	输入	暂停
co_return	输出	终止

表 8.1：新的协程关键字

上表中，在 co_yield 和 co_await 之后，协程会自行挂起，而在 co_return 之后，协程会终止（co_return 相当于 C++ 函数中的 return 语句）。协程不能有 return 语句，必须始终使用 co_return。如果协程不返回任何值并且使用了其他两个协程关键字中的任何一个，则可以省略 co_return 语句。

8.3.2. 协程的限制

使用新的 coroutines 关键字声明的函数是协程，但是协程有以下限制：

- 使用可变参数的可变数量参数的函数不能是协同程序（可变参数函数模板可以是协同程序）
- 类构造函数或析构函数不能是协同程序
- constexpr 和 consteval 函数不能是协程
- 返回 auto 的函数不能是协程，但带有尾随返回类型的 auto 可以
- main() 函数不能是协程
- Lambda 可以是协程

研究了协程的限制（基本上就是什么样的 C++ 函数不可以成为协程），下一节就要开始实现协程了。

8.4. 实现协程

上一节中，介绍了协程的基础知识、其是什么以及一些用例。

本节中，将实现三个简单的协程来说明实现和使用的基础知识：

- 最简单的协程，仅返回
- 协程将值发送回调用者
- 协程从调用者处获取值

8.4.1. 最简单的协程

协程是一个可以自行暂停并可由调用者恢复的函数，如果函数至少使用一个 co_yield、co_await 或 co_return 表达式，编译器就会将该函数标识为协程。

编译器将转换协程源代码并创建一些数据结构和函数，使协程能够正常运行并能够暂停和恢复。这是保持协程状态并能够与协程进行通信所必需的。

编译器会处理所有这些细节，但 C++ 对协程的支持还处于相当低的水平。有一些库可以让我们在 C++ 中使用协程时更加轻松。其中一些是 Lewis Baker 的 cppcoro 和 Boost.Cobalt。Boost.Asio 库也支持协程。这些库是接下来两章的主题。

让我们从头开始，编写一些代码，并根据编译器错误和 C++ 参考来编写一个基本但功能齐全的协程。

下面的代码是协程的最简单实现：

```
1  void coro_func() {
2      co_return;
3  }
4
5  int main() {
6      coro_func();
7  }
```

很简单，不是吗？第一个协程将不返回任何内容，不会做任何其他事情。遗憾的是，上述代码对于功能性协程来说太简单了，无法编译。使用 GCC 14.1 进行编译时，还会有以下错误：

```
error: coroutines require a traits template; cannot find 'std::coroutine_traits'

note: perhaps '#include <coroutine>' is missing
```

编译器给了我们一个提示：可能忘记包含一个必需的文件。先包含 `<coroutine>` 头文件，稍后再处理有关特征模板的错误：

```cpp
#include <coroutine>

void coro_func() {
    co_return;
}

int main() {
    coro_func();
}
```

编译上述代码时，我们得到以下错误

```
error: unable to find the promise type for this coroutine
```

第一个版本的协程给出了一个编译器错误，指出无法找到类型 `std::coroutine_traits` 模板。现在得到了一个与 `Promise` 类型相关的错误。

查看 C++ 参考发现 `std::coroutine_traits` 模板确定了协程的返回类型和参数类型。该参考还指出，协程的返回类型必须定义一个名为 `promise_type` 的类型。按照参考建议，编写一个新版本的协程：

```cpp
#include <coroutine>

struct return_type {
    struct promise_type {
    };
};

template<>
struct std::coroutine_traits<return_type> {
    using promise_type = return_type::promise_type;
};

return_type coro_func() {
    co_return;
}

int main() {
    coro_func();
}
```

请注意，协程的返回类型可以有名称（这里将其称为 return_type，对于这个简单的例子来说很方便）。

再次编译上述代码会有一些错误（为了清晰起见，对其进行了编辑）。所有错误都是关于 promise_type 结构中缺少函数的：

```
error: no member named 'return_void' in
↪  'std::__n4861::coroutine_traits<return_type>::promise_type'
error: no member named 'initial_suspend' in
↪  'std::__n4861::coroutine_traits<return_type>::promise_type'
error: no member named 'unhandled_exception' in
↪  'std::__n4861::coroutine_traits<return_type>::promise_type'
error: no member named 'final_suspend' in
↪  'std::__n4861::coroutine_traits<return_type>::promise_type'
error: no member named 'get_return_object' in
↪  'std::__n4861::coroutine_traits<return_type>::promise_type'
```

目前为止，看到的所有编译器错误都与代码中缺少的功能有关。用 C++ 编写协程需要遵循一些规则并帮助编译器使其生成的代码具有功能性。

以下是最简单协程的最终版本：

```cpp
#include <coroutine>

struct return_type {
    struct promise_type {
        return_type get_return_object() noexcept {
            return return_type{ *this };
        }

        void return_void() noexcept {}

        std::suspend_always initial_suspend() noexcept {
            return {};
        }

        std::suspend_always final_suspend() noexcept {
            return {};
        }

        void unhandled_exception() noexcept {}
    };

    explicit return_type(promise_type&) {
    }

    ~return_type() noexcept {
    }
};
```

```
28
29 return_type coro_func() {
30     co_return;
31 }
32
33 int main() {
34     coro_func();
35 }
```

有些读者可能注意到了，这里删除了 std::coroutine_traits 模板。实现 return 和 promise 类型就足够了。

上述代码编译时没有任何错误，可以运行它。而它什么也没做！但这是第一个协程，我们已经了解到，需要提供编译器创建协程所需的一些代码。

promise 类型

编译器需要 promise 类型。需要始终定义此类型（它可以是类或结构），必须命名为 promise_type，并且必须实现 C++ 引用中指定的某些函数。如果我们不这样做，编译器会报错。

协程返回的类型中必须定义 promis 类型，否则代码将无法编译。返回的类型（有时也称为包装类型，因为包装了 promise_type）可以命名。

8.4.2. 协程让步

不做事的协程很适合用来说明一些基本概念。现在，将实现另一个可以将数据发送回调用者的协程。

在第二个示例中，将实现一个生成消息的协程。这将是协程的“hello world”。协程会说 hello，调用函数会打印从协程收到的消息。

为了实现该功能，需要建立从协程到调用者的通信通道。此通道是允许协程将值传递给调用者，并从调用者接收信息的机制。此通道是通过协程的 Promise 类型和句柄建立的，用于管理协程的状态。

通信通道的工作方式如下：

- 协程框架：调用协程时，会创建一个协程框架，其中包含暂停和恢复其执行所需的所有状态信息。这包括局部变量、promise 类型和内部状态。
- Promise 类型：每个协程都有一个关联的 Promise 类型，负责管理协程与调用者函数的交互。Promise 是存储协程返回值的地方，提供函数来控制协程的行为。Promise 是调用者与协程交互的接口。
- 协程句柄：协程句柄是一种允许访问协程框架（协程的内部状态），并允许调用者恢复或销毁协程的类型。调用者可以使用句柄在协程暂停后恢复协程（例如，在 co_await 或 co_yield 之后）。句柄还可用于检查协程是否已完成或清理其资源。
- 暂停和恢复机制：当协程产生一个值（co_yield）或等待异步操作（co_await）时，会暂停执行，并将其状态保存在协程框架中。然后，调用者可以在稍后恢复协程，通过协程句柄

检索产生的或等待的值并继续执行。

下面的例子中，这个通信通道需要大量的代码来帮助编译器生成协程运行所需的所有代码。

以下代码是调用函数和协程的新版本：

```
return_type coro_func() {
    co_yield "Hello from the coroutine\n"s;
    co_return;
}

int main() {
    auto rt = coro_func();
    std::cout << rt.get() << std::endl;
    return 0;
}
```

变化如下：

- 协程产生并发送一些数据（在本例中为 std::string 对象）给调用者
- 调用者读取该数据并打印出来

所需的通信机制在 promise 类型和返回类型（承诺类型包装器）中实现。当编译器读取 co_yield 表达式时，将生成对 promise 类型中定义的 yield_value 函数的调用。

以下代码是我们版本的该函数的实现，生成（或产生）一个 std::string 对象：

```
std::suspend_always yield_value(std::string msg) noexcept {
    output_data = std::move(msg);
    return {};
}
```

该函数获取一个 std::string 对象并将其移动到 promise 类型的 output_data 成员变量，但这只会将数据保留在 promise 类型内。我们需要一种机制来将该字符串从协程中取出。

句柄类型

我们需要一个与协程之间的通信通道，就需要一种方法来引用已暂停或正在执行的协程。C++ 标准库在所谓的协程句柄中实现了这种机制。其类型为 std::coroutine_handle ，是返回类型的成员变量。此结构还负责句柄的整个生命周期，包括创建和销毁它。

以下代码片段是添加到返回类型中以管理协程句柄的功能：

```
std::coroutine_handle<promise_type> handle{};

explicit return_type(promise_type& promise) : handle{ std::coroutine_
    handle<promise_type>::from_promise(promise)} {
}

~return_type() noexcept {
    if (handle) {
```

```
        handle.destroy();
    }
}
```

上述代码声明了一个 std::coroutine_handle<promise_type> 类型的协程句柄，并在返回类型构造函数中创建该句柄。该句柄在返回类型析构函数中销毁。

现在，回到 yielding 协程。唯一缺少的是 get() 函数，以便调用者函数能够访问协程生成的字符串：

```
std::string get() {
    if (!handle.done()) {
        handle.resume();
    }
    return std::move(handle.promise().output_data);
}
```

如果协程未终止，get() 函数将恢复协程，然后返回字符串对象。

以下是第二个协程的完整代码：

```
#include <coroutine>
#include <iostream>
#include <string>

using namespace std::string_literals;

struct return_type {
    struct promise_type {
        std::string output_data { };

        return_type get_return_object() noexcept {
            std::cout << "get_return_object\n";
            return return_type{ *this };
        }

        void return_void() noexcept {
            std::cout << "return_void\n";
        }

        std::suspend_always yield_value(
        std::string msg) noexcept {
            std::cout << "yield_value\n";
            output_data = std::move(msg);
            return {};
        }

        std::suspend_always initial_suspend() noexcept {
            std::cout << "initial_suspend\n";
```

```
            return {};
        }

        std::suspend_always final_suspend() noexcept {
            std::cout << "final_suspend\n";
            return {};
        }

        void unhandled_exception() noexcept {
            std::cout << "unhandled_exception\n";
        }
    };

    std::coroutine_handle<promise_type> handle{};

    explicit return_type(promise_type& promise)
        : handle{ std::coroutine_handle<
                  promise_type>::from_promise(promise)}{
        std::cout << "return_type()\n";
    }

    ~return_type() noexcept {
        if (handle) {
            handle.destroy();
        }
        std::cout << "~return_type()\n";
    }

    std::string get() {
        std::cout << "get()\n";
        if (!handle.done()) {
            handle.resume();
        }
        return std::move(handle.promise().output_data);
    }
};

return_type coro_func() {
    co_yield "Hello from the coroutine\n"s;
    co_return;
}

int main() {
    auto rt = coro_func();
    std::cout << rt.get() << std::endl;
    return 0;
}
```

运行上述代码将输出以下消息：

```
get_return_object
return_type()
initial_suspend
get()
yield_value
Hello from the coroutine
~return_type()
```

此输出展示了协程执行期间发生的情况：

1. 调用 get_return_object 后创建 return_type 对象
2. 协程最初处于挂起状态
3. 调用者希望从协程中获取消息，因此调用 get()
4. 调用 yield_value，恢复协程，并将消息复制到成员
5. 调用函数打印消息，协程返回

注意，Promise（和 promise 类型）与第 6 章中解释的 C++ 标准库 std::promise 类型无关。

8.4.3. 协程等待

上一个示例中，了解了如何实现一个协程，该协程可以通过向调用方发送 std::string 对象来与调用方进行通信。现在，将实现一个协程，该协程可以等待调用方发送的输入数据。示例中，协程将等待，直到获得 std::string 对象，然后输出。当让协程“等待”时，其会暂停（即不执行）直到收到数据。

从协程和调用函数的改变开始：

```
1  return_type coro_func() {
2      std::cout << co_await std::string{ };
3      co_return;
4  }
5
6  int main() {
7      auto rt = coro_func();
8      rt.put("Hello from main\n"s);
9      return 0;
10 }
```

上面的代码中，调用者函数调用 put() 函数（返回类型结构中的方法），协程调用 co_await 等待来自调用者的 std::string 对象。

返回类型的改变很简单，增加 put() 函数即可：

```
void put(std::string msg) {
    handle.promise().input_data = std::move(msg);
    if (!handle.done()) {
        handle.resume();
    }
}
```

需要将 input_data 变量添加到 promise 结构中。但仅因为对第一个示例（将其作为本章其余示例的起点，这是实现协程的最小代码）和上一个示例中的协程句柄进行了这些更改，代码就无法编译。编译器给出了以下错误：

```
error: no member named 'await_ready' in 'std::string' {aka 'std::__cxx11::basic_string<char>'}
```

回到 C++ 引用，看到当协程调用 co_await 时，编译器将生成代码来调用 promise 对象中名为 await_transform 的函数，该函数具有与协程正在等待的数据相同类型的参数。顾名思义，await_transform 是一个将对象（示例中为 std::string）转换为可等待对象的函数。std::string 不可等待，所以出现前面的编译错误。

await_transform 必须返回一个 awaiter 对象。这只是一个简单的结构,实现了 awaiter 所需的接口，以便编译器可以使用。

以下代码展示了我们对 await_transform 函数和 awaiter 结构的实现：

```
auto await_transform(std::string) noexcept {
    struct awaiter {
        promise_type& promise;

        bool await_ready() const noexcept {
            return true;
        }

        std::string await_resume() const noexcept {
            return std::move(promise.input_data);
        }

        void await_suspend(std::coroutine_handle<
                           promise_type>) const noexcept {
        }
    };
    return awaiter(*this);
}
```

promise_type 函数 await_transform 是编译器所必需的，不能为此函数使用不同的标识符。参数类型必须与协程正在等待的对象相同。awaiter 结构可以任意命名，在这里使用 awaiter。

awaiter 结构必须实现三个函数：

- await_ready：调用此函数检查协程是否已暂停。如果是，则返回 false。在示例中，始

终返回 `true`，表示协程未暂停。

- `await_resume`：这将恢复协程并生成 `co_await` 表达式的结果。
- `await_suspend`：简单等待程序中，返回 `void`，会将控制权传递给调用者，并且协程暂停。`await_suspend` 也可能返回布尔值。这种情况下返回 `true` ，返回 `false` 则意味着协程已恢复运行。

这是等待协程的完整示例代码：

```
#include <coroutine>
#include <iostream>
#include <string>

using namespace std::string_literals;

struct return_type {
    struct promise_type {
        std::string input_data { };

        return_type get_return_object() noexcept {
            return return_type{ *this };
        }

        void return_void() noexcept {
        }

        std::suspend_always initial_suspend() noexcept {
            return {};
        }

        std::suspend_always final_suspend() noexcept {
            return {};
        }

        void unhandled_exception() noexcept {
        }

        auto await_transform(std::string) noexcept {
            struct awaiter {
                promise_type& promise;

                bool await_ready() const noexcept {
                    return true;
                }

                std::string await_resume() const noexcept {
                    return std::move(promise.input_data);
                }

```

```
                void await_suspend(std::coroutine_handle<
                                   promise_type>) const noexcept {
                }
            };

            return awaiter(*this);
        }
    };

    std::coroutine_handle<promise_type> handle{};

    explicit return_type(promise_type& promise)
        : handle{ std::coroutine_handle<
                          promise_type>::from_promise(promise)} {
    }

    ~return_type() noexcept {
        if (handle) {
            handle.destroy();
        }
    }

    void put(std::string msg) {
        handle.promise().input_data = std::move(msg);
        if (!handle.done()) {
            handle.resume();
        }
    }
};

return_type coro_func() {
    std::cout << co_await std::string{ };
    co_return;
}

int main() {
    auto rt = coro_func();
    rt.put("Hello from main\n"s);
    return 0;
}
```

本节中，了解了协程的三个基本示例。实现了最简单的协程，然后实现了具有通信通道的协程，以便为调用者生成数据（co_yield）并等待来自调用者的数据（co_await ）。

下一节中，将实现一种称为生成器的协程并生成数字序列。

8.5. 协程生成器

生成器是一个协同程序，通过从暂停点反复恢复自身来生成一系列元素。

生成器可以看作是一个无限序列，可以生成任意数量的元素。调用函数可以根据需要从生成器中，获取任意数量的新元素。

当我们说无限时，指的是理论上。生成器协同程序将产生没有明确最后一个元素的元素（可以实现具有有限范围的生成器）。但在实践中，必须处理诸如数字序列溢出之类的问题。

让我们从头开始实现一个生成器，应用在本章前面章节中获得的知识。

8.5.1. 斐波那契数列生成器

假设正在实现一个应用程序，需要使用斐波那契数列。斐波那契数列是一个序列，其中每个数字都是前两个数字的总和。第一个元素是 `0`，第二个元素是 `1`，然后应用定义并生成一个又一个元素。

斐波那契数列：$F(n) = F(n-2) + F(n-1); F(0) = 0, F(1) = 1$

可以用 `for` 循环生成这些数字，但需要在程序的不同点生成相应的值，所以需要实现一种方法来存储序列的状态，还需要在程序的某个地方保存我们生成的最后一个元素。

协程是解决此问题的一个非常好的方法，将自行保持所需的状态并且将暂停直到请求序列中的下一个数字。

以下是使用生成器协程的代码：

```
int main() {
    sequence_generator<int64_t> fib = fibonacci();

    std::cout << "Generate ten Fibonacci numbers\n"s;

    for (int i = 0; i < 10; ++i) {
        fib.next();
        std::cout << fib.value() << " ";
    }
    std::cout << std::endl;

    std::cout << "Generate ten more\n"s;

    for (int i = 0; i < 10; ++i) {
        fib.next();
        std::cout << fib.value() << " ";
    }
    std::cout << std::endl;

    std::cout << "Let's do five more\n"s;

    for (int i = 0; i < 5; ++i) {
        fib.next();
```

```
        std::cout << fib.value() << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

正如上面的代码所示，生成了所需的数字，而不必担心最后一个元素是什么。序列由协程生成。请注意，尽管理论上该序列是无限的，但程序必须意识到非常大的斐波那契数的潜在溢出。为了实现生成器协程，遵循本章前面解释的原则。

首先实现协程函数：

```
sequence_generator<int64_t> fibonacci() {
    int64_t a{ 0 };
    int64_t b{ 1 };
    int64_t c{ 0 };

    while (true) {
        co_yield a;
        c = a + b;
        a = b;
        b = c;
    }
}
```

协程只是通过应用公式来生成斐波那契数列中的下一个元素。元素是在无限循环中生成的，但协程会在 `co_yield` 之后自行暂停。

返回类型是 `sequence_generator` 结构（我们使用模板来使用 32 位或 64 位整数），包含一个承诺类型，非常类似于我们在上一节中看到的协程让步中的 `promise` 类型。

在 `sequence_generator` 结构中，添加了两个在实现序列生成器时很有用的函数。

```
void next() {
    if (!handle.done()) {
        handle.resume();
    }
}
```

`next()` 函数为将要生成的序列中的新斐波那契数恢复协程。

```
int64_t value() {
    return handle.promise().output_data;
}
```

`value()` 函数返回最后生成的斐波那契数。

这样，将元素生成与其检索 `Qvalue` 分离。

请在本书附带的 `GitHub` 库中查找本示例的完整代码。

C++23 的 std::generator

C++ 中实现即使是最基本的协程也需要一定数量的代码。这种情况可能会在 C++26 中发生变化，C++ 标准库将对协程提供更多支持，这将使我们能够更轻松地编写协程。

C++23 引入了 std::generator 模板类，可以编写基于协程的生成器，而无需编写任何必需的代码，例如：promise 类型、返回类型及其所有函数。要运行此示例，需要一个 C++23 编译器，这里使用了 GCC 14.1。std::generator 在 Clang 中不可用。

来看看使用新的 C++23 标准库功能的斐波那契数列生成器：

```
#include <generator>
#include <iostream>

std::generator<int> fibonacci_generator() {
    int a{ };
    int b{ 1 };
    while (true) {
        co_yield a;
        int c = a + b;
        a = b;
        b = c;
    }
}
auto fib = fibonacci_generator();

int main() {
    int i = 0;
    for (auto f = fib.begin(); f != fib.end(); ++f) {
        if (i == 10) {
            break;
        }
        std::cout << *f << " ";
        ++i;
    }
    std::cout << std::endl;
}
```

第一步是包含 <generator> 头文件。然后，只需编写协程，所有代码都已为编写完成。前面的代码中，使用迭代器（由 C++ 标准库提供）访问生成的元素，能够使用 range-for 循环、算法和范围。

还可以编写斐波那契生成器的一个版本，来生成一定数量的元素而不是无限数列：

```
std::generator<int> fibonacci_generator(int limit) {
    int a{ };
    int b{ 1 };
    while (limit--) {
        co_yield a;
        int c = a + b;
```

```
        a = b;
        b = c;
    }
}
```

代码的变化非常简单：只需传递希望生成器生成的元素数量，并将其用作 while 循环中的终止条件。

本节中，实现了最常见的协程类型之一——生成器。我们从头开始以及使用 C++23 的 std::generator 类模板实现了生成器。

我们将在下一节中实现一个简单的字符串解析器协程。

8.6. 简单的协程字符串解析器

本节中,将实现最后一个例子:一个简单的字符串解析器。协程将等待输入(一个 std::string 对象)，并在解析输入字符串后产生输出（一个数字)。为了简化示例，假设数字的字符串表示没有错误，并且数字的结尾由井号 # 表示；还假设数字类型为 i nt64_t，并且字符串不包含该整数类型范围之外的值。

8.6.1. 解析算法

看看如何将表示整数的字符串转换为数字。例如，字符串“-12321#”表示数字 -1232 1。要将字符串转换为数字，可以编写如下函数：

```
int64_t parse_string(const std::string& str) {
    int64_t num{ 0 };
    int64_t sign { 1 };

    std::size_t c = 0;
    while (c < str.size()) {
        if (str[c] == '-') {
            sign = -1;
        }
        else if (std::isdigit(str[c])) {
            num = num * 10 + (str[c] - '0');
        }
        else if (str[c] == '#') {
            break;
        }
        ++c;
    }
    return num * sign;
}
```

由于假设字符串格式正确，因此代码非常简单。如果读取减号 -，则将其更改为 -1（默认情况下，假设为正数，如果有 + 符号，则将其忽略）。然后，逐个读取数字，并按如下方式计算数值。

num 的初始值是 0。读取第一位数字，并将其数值添加到当前 num 值乘以 10 之后。这就是读取数字的方式：最左边的数字将乘以 10，次数与其右边的数字数量相同。

使用字符来表示数字时，可根据 ASCII 表示具有一些值（我们假设不使用宽字符或任何其他字符类型）。字符 0 到 9 具有连续的 ASCII 码，因此只需减去 0 即可将它们转换为数字。

即使对于上述代码，最后一个字符检查不是必需的，但还是在此处包含了它。当解析器例程找到 # 字符时，它会终止解析循环并返回最终的数字值。

我们可以使用此函数来解析任何字符串并获取数字值，但需要完整的字符串才能将其转换为数字。

考虑一下这个场景：从网络连接接收到字符串，需要解析它并将其转换为数字。可以将字符保存到临时字符串中，然后调用前面的函数。

但还有另一个问题：如果字符到达速度很慢，比如每隔几秒才到达一次，因为这就是字符的传输方式，那该怎么办？希望让 CPU 保持忙碌，如果可能的话，在等待每个字符到达时执行一些其他任务（或任务）。

解决这个问题的方法有很多种。可以创建一个线程并同时处理字符串，但对于这样一个简单的任务来说，这会浪费大量的计算机时间，也可以使用 std::async。

8.6.2. 解析协程

本章将使用协程，因此将使用 C++ 协程实现字符串解析器。不需要多余的线程，而且由于协程的异步特性，在字符到达时执行其他处理都非常容易。

解析协程所需的样板代码，与前面的示例中已经看到的代码几乎相同。解析器本身则完全不同。请参阅以下代码：

```
async_parse<int64_t, char> parse_string() {
    while (true) {
        char c = co_await char{ };
        int64_t number { };
        int64_t sign { 1 };

        if (c != '-' && c != '+' && !std::isdigit(c)) {
            continue;
        }
        if (c == '-') {
            sign = -1;
        }
        else if (std::isdigit(c)) {
            number = number * 10 + c - '0';
        }

        while (true) {
            c = co_await char{};
```

```
            if (std::isdigit(c)) {
                number = number * 10 + c - '0';
            }
            else {
                break;
            }
        }
        co_yield number * sign;
    }
}
```

现在可以轻松识别返回类型（async_parse<int64_t、char>），并且解析器协程会暂停自身以等待输入字符。解析完成后，协程将在产生数字后暂停自身。

但也可以看到，前面的代码并不像第一次尝试将字符串解析为数字那么简单。

首先，解析器协程会逐个解析字符，不会获取完整的字符串进行解析，因此会无限循环 while (true)。不知道完整字符串中有多少个字符，因此需要继续接收和解析。

外循环意味着协程将随着字符的到来而一个接一个地解析数字——永远如此。但它会暂停自身以等待字符，这样就不会浪费 CPU 时间。

现在，一个字符到达。首先，检查它是否是我们数字的有效字符。如果该字符不是减号 -、加号 + 或数字，则解析器等待下一个字符。

如果下一个字符是有效字符，则适用以下情况：

- 如果是减号，将符号值改为 -1
- 如果是加号，忽略
- 如果是数字，将其解析为数字，并使用与解析器的第一个版本中看到的相同方法更新当前数字值

在第一个有效字符之后，进入一个新的循环来接收其余的字符，无论是数字还是分隔符 (#)。注意，当说有效字符时，指的是适合数字转换。仍然假设输入的字符形成一个正确终止的有效数字。

一旦数字转换，协程就会将其返回，然后再次执行外循环。这里需要终止字符，输入字符流理论上是无穷无尽的，并且可以包含许多数字。

协程其余部分的代码可以在 GitHub 中找到，遵循与其他协程相同的约定。首先，定义返回类型：

```
template <typename Out, typename In>
struct async_parse {
    // ···
};
```

使用模板来提高灵活性,其允许参数化输入和输出数据类型。本例中,这些类型分别是 int64_t 和 char。

输入和输出数据项如下：

```
std::optional<In> input_data { };
Out output_data { };
```

对于输入，可使用 std::optional<In>，因为需要一种方法来知道我们是否收到了一个字符。这里，使用 put() 函数将字符发送到解析器：

```
void put(char c) {
    handle.promise().input_data = c;
    if (!handle.done()) {
        handle.resume();
    }
}
```

此函数仅将值赋给 std::optional input_data 变量。为了管理字符的等待，实现了以下 awaiter 类型：

```
auto await_transform(char) noexcept {
    struct awaiter {
        promise_type& promise;

        [[nodiscard]] bool await_ready() const noexcept {
            return promise.input_data.has_value();
        }

        [[nodiscard]] char await_resume() const noexcept {
            assert (promise.input_data.has_value());
            return *std::exchange(
                            promise.input_data,
                            std::nullopt);
        }

        void await_suspend(std::coroutine_handle<
                           promise_type>) const noexcept {
        }
    };
    return awaiter(*this);
}
```

awaiter 结构实现了两个函数来处理输入数据：

- await_ready()：如果可选的 input_data 变量包含有效值，则返回 true。否则返回 false。
- await_resume()：返回存储在可选 input_data 变量中的值并将其清空，并将其分配给 std::nullopt。

本节中，了解了如何使用 C++ 协程实现一个简单的解析器。这是最后一个例子，说明了使用协程实现的一个非常基本的流处理功能。下一节中，将介绍协程中的异常。

8.7. 协程和异常

前面的部分中，实现了几个基本示例来学习主要的 C++ 协程概念。首先实现了一个非常基本的协程，以了解编译器对我们的要求：返回类型（有时称为包装器类型，包装了 promise 类型）和 promise 类型。

即使对于这样一个简单的协程，也必须实现我们在编写示例时解释过的一些函数。但有一个函数尚未解释：

```
1 void unhandled_exception() noexcept {}
```

当时假设协程不会抛出异常，但事实是它们会抛出异常。我们可以在 unhandled_exception() 函数体中添加处理异常的功能。

协程中的异常可能发生在创建返回类型或 promiase 类型对象时，以及执行协程时（与在正常函数中一样，协程可能会引发异常）。

不同之处在于，如果异常是在协程执行前引发的，则创建协程的代码必须处理该异常，而如果异常是在协程执行时引发的，则会调用 unhandled_exception()。

第一种情况只是普通的异常处理，没有调用任何特殊函数。可以将协程创建放在 try-catch 块中，并像在代码中通常做的那样处理可能的异常。

另一方面，如果调用 unhandled_exception()（在 promise 类型内部），必须在该函数内部实现异常处理功能。

有不同的策略来处理此类异常。其中包括：

- 重新抛出异常，以便可以在 promise 类型之外（即在我们的代码中）处理。
- 终止程序（例如，调用 std::terminate）。
- 函数为空。协程将崩溃，很可能会使程序崩溃。

因为实现的协程非常简单，所以将该函数置为空。

最后一节中，介绍了协程的异常处理机制。正确处理异常非常重要。例如，知道协程内部发生异常后，将无法恢复；那么，最好让协程崩溃并从程序的另一部分（通常是从调用者函数）处理异常。

8.8. 总结

本章中，了解了协程，这是 C++ 中最近引入的一项新功能，允许编写异步代码而无需创建新线程。我们实现了几个简单的协程来解释 C++ 协程的基本要求。此外，还介绍了如何实现生成器和字符串解析器。最后，了解了协程中的异常。

协程在异步编程中非常重要，允许程序在特定点暂停执行并稍后恢复，同时允许其他任务运行，所有任务都在同一个线程中运行。它们可以更好地利用资源，减少等待时间，并提高应用程序的可扩展性。

下一章中，将介绍 Boost.Asio——一个用于在 C++ 中编写异步代码的非常强大的库。

8.9. 扩展阅读

- C++ Coroutines for Beginners, Andreas Fertig, Meeting C++ Online, 2024
- Deciphering Coroutines, Andreas Weiss, CppCon 2022

第四部分 使用 Boost 库进行高级异步编程

在本部分中，将使用强大的 Boost 库来学习高级异步编程技术，能够高效地管理与外部资源和系统级服务交互的任务。这里将介绍 Boost.Asio 和 Boost.Cobalt 库，了解它们如何简化异步应用程序的开发，同时提供对复杂流程（如任务管理和协程执行）的细粒度控制。通过实际操作示例，将了解 Boost.Asio 如何在单线程和多线程环境中处理异步 I/O 操作，以及 Boost.Cobalt 如何抽象出 C++20 协程的复杂性，使开发者能够专注于功能，而非底层协程管理。

本部分包含以下章节：

- 第 9 章，使用 Boost.Asio 进行异步编程
- 第 10 章，使用 Boost.Cobalt 实现协程

第 9 章 使用 Boost.Asio 进行异步编程

Boost.Asio 是著名的 Boost 库系列中的一个 C++ 库，简化了处理由操作系统（OS）管理的异步输入/输出（I/O）任务的解决方案的开发，从而更容易开发处理内部和外部资源的异步软件，例如：网络通信服务或文件操作。

为此，Boost.Asio 定义了 OS 服务（属于 OS 并由 OS 管理的服务）、I/O 对象（提供 OS 服务的接口）和 I/O 执行上下文对象（充当服务注册表和代理的对象）。

在接下来的内容中，将介绍 Boost.Asio，介绍它的主要构建块，并解释一些使用该库开发异步软件的常见模式，这些模式在业界中广泛使用。

本章中，将讨论以下主要主题：

- Boost.Asio 是什么，以及它如何利用外部资源简化异步编程
- 什么是 I/O 对象和 I/O 执行上下文，如何与 OS 服务交互，以及相互之间如何交互
- Proactor 和 Reactor 设计模式是什么，以及与 Boost.Asio 有何关系
- 如何保证程序线程安全，以及如何使用 strand 序列化任务
- 如何使用缓冲区高效地将数据传递给异步任务
- 如何取消异步操作
- 计时器和网络应用程序的实践示例

9.1. 技术要求

对于本章，需要安装 Boost C++ 库。撰写本书时最新版本是 Boost 1.85.0。以下是发行说明：

https://www.boost.org/users/history/version_1_85_0.html

有关 Unix 变体系统（Linux、macOS）中的安装说明，请查看以下链接：

https://www.boost.org/doc/libs/1_85_0/more/getting_started/unix-variants.html

对于 Windows 系统，请查看此链接：

https://www.boost.org/doc/libs/1_85_0/more/getting_started/windows.html

此外，根据要开发的项目，可能需要配置 Boost.Asio 或安装依赖项：

https://www.boost.org/doc/libs/1_85_0/doc/html/boost_asio/using.html

本章中显示的所有代码均受 C++20 版本的支持。请查看第 3 章中的技术要求部分，其中提供了有关如何安装 GCC 13 和 Clang 8 编译器的一些指导。

可以在以下 GitHub 库中找到完整的代码：

https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP

本章的示例位于 Chapter_09 文件夹下。所有源代码文件都可以使用 CMake 进行编译：

```
cmake . && cmake —build .
```

可执行二进制文件将在 bin 目录下生成。

9.2. 什么是 Boost.Asio?

Boost.Asio 是一个跨平台的 C++ 库，由 Chris Kohlhoff 创建，提供可移植的网络和底层 I/O 编程，包括套接字、计时器、主机名解析、套接字 iostream、串行端口、文件描述符和 Windows HANDLE，提供一致的异步模型。还提供协程支持，它们现在在 C++20 中可用，因此在本章中仅简要进行介绍。

Boost.Asio 允许程序管理长时间运行的操作，而无需明确使用线程和锁。此外，由于它在操作系统服务之上实现了一层，因此可以使用最合适的底层操作系统机制来实现可移植性、效率、易用性和可扩展性，例如：分散-聚集 I/O 操作或移动数据，同时最大限度地减少昂贵的复制。

首先，先了解基本的 Boost.Asio 块、I/O 对象和 I/O 执行上下文对象。

9.2.1. I/O 对象

有时，应用程序需要访问 OS 服务，在其上运行异步任务并收集结果或错误。Boost.Asio 提供了一种由 I/O 对象和 I/O 执行上下文对象组成的机制来实现此功能。

I/O 对象是面向任务的对象，表示执行 I/O 操作的实际实体。如图 9.1 所示，Boost.Asio 提供了核心类来管理并发、流、缓冲区或库中的其他核心功能，还包括用于通过传输控制协议/Internet 协议（TCP/IP）、用户数据报协议（UDP）或 Internet 控制消息协议（ICMP）进行网络通信的可移植网络类，用于定义安全层、传输协议和串行端口等任务类，以及用于处理取决于底层操作系统的特定设置的平台特定类。

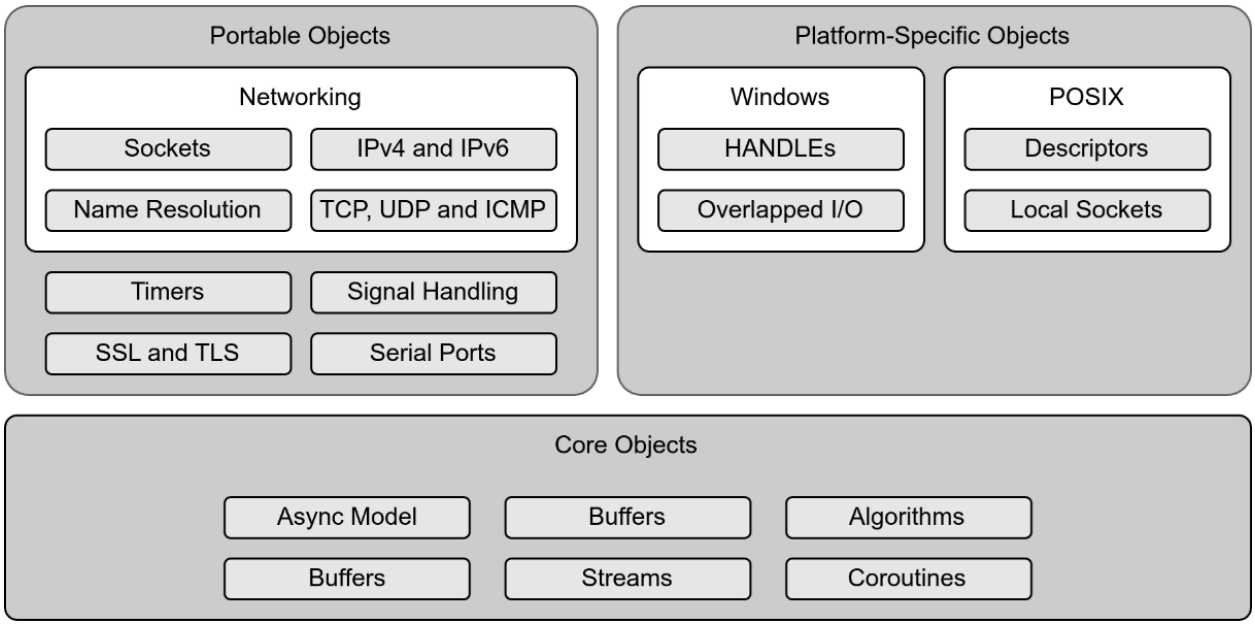

图 9.1 –I/O 对象

I/O 对象不直接在 OS 中执行其任务，需要通过 I/O 执行上下文对象与 OS 进行通信。上下文对象的实例作为 I/O 对象构造函数中的第一个参数传递。这里，定义一个 I/O 对象（一个到期时间为三秒的计时器）并通过其构造函数传递一个 I/O 执行上下文对象（io_context）：

```cpp
#include <boost/asio.hpp>
#include <chrono>

using namespace std::chrono_literals;

boost::asio::io_context io_context;
boost::asio::steady_timer timer(io_context, 3s);
```

大多数 I/O 对象都有名称以 async_ 开头的方法。这些方法触发异步操作，当操作完成时，将调用完成处理程序（即作为参数传递给方法的可调用对象）。这些方法立即返回，不会阻塞程序流。当前线程可以在任务未完成时继续执行其他任务。完成后，将调用并执行完成处理程序，处理异步任务的结果或错误。

I/O 对象还提供了阻塞对应方法，这些方法将阻塞直至完成。这些方法不需要接收处理程序作为参数。

如前所述，I/O 对象不直接与操作系统交互，需要一个 I/O 执行上下文对象。

让我们了解一下这类对象。

9.2.2. I/O 执行上下文对象

为了访问 I/O 服务，程序至少使用一个 I/O 执行上下文对象，该对象代表 OS I/O 服务的网关。使用 boost::asio::io_context 类实现，为 I/O 对象提供 OS 服务的核心 I/O 功能。Windows 中，boost::asio::io_context 基于 I/O 完成端口（IOCP）；在 Linux 上，基于 epoll；在 FreeBSD/macOS 上，基于 kqueue。

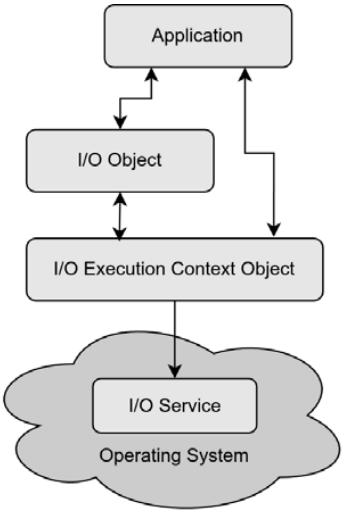

图 9.2 – Boost.Asio 架构

boost::asio::io_context 是 boost::asio::execution_context 的子类，是函数对象执行的基类，也被其他执行上下文对象继承，例如：boost::asio::thread_pool 或 boost::asio::system_context。

本章中，将使用 boost::asio::io_context 作为执行上下文对象。自 1.66.0 版本以来，boost::asio::io_context 类已替代了 boost::asio::io_service 类，并采用了

C++ 中更多现代特性和实践。boost::asio::io_service 仍可向后兼容。

如前所述，Boost.Asio 对象可以使用以 async_ 开头的方法调度异步操作。当所有异步任务都调度完毕后，程序需要调用 boost::asio::io_context::run() 函数来执行事件处理循环，让操作系统处理任务并将结果传递给程序并触发处理程序。

回到之前的示例，现在将设置完成处理程序 on_timeout()，这是一个可调用对象（在本例中是一个函数），在调用异步 async_wait() 函数时将其作为参数传递。以下是代码示例：

```
#include <boost/asio.hpp>
#include <iostream>

void on_timeout(const boost::system::error_code& ec) {
    if (!ec) {
        std::cout << "Timer expired.\n" << std::endl;
    } else {
        std::cerr << "Error: " << ec.message() << '\n';
    }
}

int main() {
    boost::asio::io_context io_context;
    boost::asio::steady_timer timer(io_context,
                                    std::chrono::seconds(3));
    timer.async_wait(&on_timeout);
    io_context.run();
    return 0;
}
```

运行此代码，三秒后应该在控制台中看到消息“Timer expired.”，或者如果异步调用由于任何原因失败，则会看到一条错误消息。

boost::io_context::run() 是一个阻塞调用。这旨在保持事件循环运行，允许异步操作运行，并防止程序退出。显然，可以在新线程中调用此函数，并使主线程保持畅通，以继续执行其他任务，正如在前面的章节中看到的那样。

当没有待处理的异步操作时，boost::io_context::run() 将返回。有一个模板类 boost::asio::executor_work_guard，可以保持 io_context 忙碌，并在需要时避免其退出。让我们通过一个例子看看它是如何工作的。

首先定义一个后台任务，该任务将等待两秒钟，然后使用 boost::asio::io_context::post() 函数通过 io_context 发布一些工作：

```
#include <boost/asio.hpp>
#include <chrono>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

```

```
void background_task(boost::asio::io_context& io_context) {
    std::this_thread::sleep_for(2s);
    std::cout << "Posting a background task.\n";
    io_context.post([]() {
        std::cout << "Background task completed!\n";
    });
}
```

main() 函数中，创建了 io_context 对象，并使用该 io_context 对象构造了一个 work_guard 对象。

然后，创建两个线程，io_thread（io_context 在其中运行）和 worker（background_task() 在其中运行），还将 io_context 作为对后台任务的引用传递以发布工作。

有了这个，主线程就会做一些工作（等待五秒钟），然后调用 reset() 函数删除工作保护，让 io_context 退出其 run() 函数，并在退出之前连接两个线程：

```
int main() {
    boost::asio::io_context io_context;
    auto work_guard = boost::asio::make_work_guard(
                      io_context);

    std::thread io_thread([&io_context]() {
        std::cout << "Running io_context.\n";
        io_context.run();
        std::cout << "io_context stopped.\n";
    });

    std::thread worker(background_task,
                       std::ref(io_context));

    // Main thread doing some work.
    std::this_thread::sleep_for(5s);
    std::cout << "Removing work_guard." << std::endl;
    work_guard.reset();
    worker.join();
    io_thread.join();
    return 0;
}
```

如果运行上面的代码，则会输出：

```
Running io_context.
Posting a background task.
Background task completed!
Removing work_guard.
io_context stopped.
```

可以看到后台线程如何正确地发布后台任务，并且这在工作保护被移除并且 I/O 上下文对象停止执行之前完成。

保持 io_context 对象处于活动状态，并处理请求的另一种方法是通过连续调用 async_ 函数，或从完成处理程序发布工作来提供异步任务。这是读取或写入套接字或流时的常见模式：

```
#include <boost/asio.hpp>
#include <chrono>
#include <functional>
#include <iostream>

using namespace std::chrono_literals;

int main() {
    boost::asio::io_context io_context;
    boost::asio::steady_timer timer(io_context, 3s);

    std::function<void(const boost::system::error_code&)>
                    timer_handler;

    timer_handler = [&timer, &timer_handler](
                    const boost::system::error_code& ec) {
        if (!ec) {
            std::cout << "Handler: Timer expired.\n";
            timer.expires_after(1s);
            timer.async_wait(timer_handler);
        } else {
            std::cerr << "Handler error: "
                      << ec.message() << std::endl;
            }
        };
    timer.async_wait(timer_handler);
    io_context.run();
    return 0;
}
```

timer_handler 是完成处理程序，定义为一个 lambda 函数，用于捕获计时器和其自身。每秒，当计时器到期时，它会输出”Handler: Timer expired.” 消息，并通过将新的异步任务（async_wait() 函数）通过计时器对象加入 io_context 对象来重新启动自身。

io_context 对象可以从线程运行。默认情况下，此对象线程安全，但在某些情况下，我们想要更好的性能，可能希望避免这种安全性。

并发提示

io_context 构造函数接受并发提示作为参数，向实现建议应用于运行完成处理程序的活动线程数。

默认情况下，此值为 BOOST_ASIO_CONCURRENCY_HINT_SAFE（值 1），表示 io_context 对象将从单个线程运行，因此可以实现多项优化。这并不意味着 io_context 只能从一个线

程使用；它仍然提供线程安全性，并且可以从多个线程使用 I/O 对象。

可以指定的其他值如下：

- BOOST_ASIO_CONCURRENCY_HINT_UNSAFE：禁用锁定，因此对 io_context 或 I/O 对象的所有操作都必须在同一个线程中发生。
- BOOST_ASIO_CONCURRENCY_HINT_UNSAFE_IO：禁用反应器中的锁定，但将其保留在调度程序中，因此 io_context 对象中的所有操作都可以使用除 run() 函数和与执行事件处理循环相关的其他方法之外的不同线程。在解释库背后的设计原理时，将了解调度程序和反应器。

现在，我们了解什么是事件处理循环，以及如何管理它。

9.2.3. 事件处理循环

使用 boost::asio::io_context::run() 方法，io_context 会阻塞并继续处理 I/O 异步任务，直到所有任务完成并通知完成处理程序。此 I/O 请求处理在内部事件处理循环中完成。

还有其他方法可以控制事件循环并避免阻塞，直到所有异步事件都处理完毕。这些方法如下：

- poll：运行事件处理循环来执行已就绪的处理程序
- poll_one：运行事件处理循环来执行一个就绪的处理程序
- run_for：运行事件处理循环指定的时间
- run_until：与上一个相同，但只到指定时间
- run_one：运行事件处理循环以执行最多一个处理程序
- run_one_for：与上一个相同，但只持续指定的时间
- run_one_until：与上一个相同，但只到指定时间

还可以通过调用 boost::asio::io_context::stop() 方法来停止事件循环，或者通过调用 boost:asio::io_context::stopped() 检查其状态是否已停止。当事件循环未运行时，已安排的任务将继续执行，其他任务将保持待处理状态。可以通过使用前面提到的方法之一，再次启动事件循环来恢复待处理的任务并收集待处理的结果。

前面的示例中，应用程序通过调用异步方法或使用 post() 函数将一些工作发送给 io_context。现在我们了解 dispatch() 与 post() 的区别。

给 io_context 分配一些工作除了通过不同 I/O 对象的异步方法或使用 executor_work_guard（如下所述）将工作发送到 io_context 之外，还可以使用 boost::asio::post() 和 boost::asio::dispatch() 模板方法。这两个函数都用于将一些工作安排到 io_context 对象中。

post() 函数保证任务一定会执行。它把完成处理程序放入执行队列，最终会执行该任务：

```
boost::asio::io_context io_context;
io_context.post([] {
    std::cout << "This will always run asynchronously.\n";
});
```

另一方面，如果 io_context 或 strand（本章后面会详细介绍 strands）与分派任务的线程相同，则 dispatch() 可以立即执行任务，否则将其放置在队列中以进行异步执行：

```
boost::asio::io_context io_context;
io_context.dispatch([] {
    std::cout << "This might run immediately or be queued.\n";
});
```

使用 dispatch()，可以通过减少上下文切换或排队延迟来优化性能。

即使有其他待处理事件排队，已调度事件也可以直接从当前工作线程执行。已发布的事件必须始终由 I/O 执行上下文管理，等待其他处理程序完成后才可执行。

现在，已经了解了一些基本概念，接下来介绍同步和异步操作的工作原理。

9.3. 与操作系统交互

Boost.Asio 可以使用同步和异步操作与 I/O 服务交互，来了解它们的行为方式以及主要区别是什么。

9.3.1. 同步操作

如果程序想要以同步方式使用 I/O 服务，通常它会创建一个 I/O 对象并使用其同步操作方法：

```
boost::asio::io_context io_context;
boost::asio::steady_timer timer(io_context, 3s);
timer.wait();
```

调用 timer.wait() 时，请求将发送到 I/O 执行上下文对象（io_context），该对象调用 OS 执行操作。OS 完成任务后，将结果返回给 io_context，然后 io_context 将结果（如果出现问题，则为错误）转换回 I/O 对象（定时器）。错误类型为 boost::system::error_code。如果发生错误，则会引发异常。

如果不想抛出异常，可以通过引用同步方法传递一个错误对象来捕获操作的状态并在之后进行检查：

```
boost::system::error_code ec;
Timer.wait(server_endpoint, ec);
```

9.3.2. 异步操作

对于异步操作，还需要将完成处理程序传递给异步方法。此完成处理程序是一个可调用对象，当异步操作完成时，将由 I/O 上下文对象调用，通知程序有关结果或操作错误。

其签名如下：

```
void completion_handler(const boost::system::error_code& ec);
```

继续计时器示例，现在需要调用异步操作：

```
socket.async_wait(completion_handler);
```

再次，I/O 对象（计时器）将请求转发到 I/O 执行上下文对象（io_context）。io_context 请求操作系统启动异步操作。

当操作完成后，操作系统将结果放入队列中，io_context 正在监听该队列。然后，io_context 将结果从队列中取出，将错误转换为错误代码对象，并触发完成处理程序以通知任务和结果的完成情况。

为了允许 io_context 执行这些步骤，程序必须执行 boost::asio::io_context::run()（或前面介绍的管理事件处理循环的类似函数），并在处理任何未完成的异步操作时阻塞当前线程。如前所述，如果没有待处理的异步操作，boost::asio::io_context::run() 将退出。

完成处理程序必须是可复制构造的，所以必须有一个可用的复制构造函数。如果需要临时资源（例如：内存、线程或文件描述符），则会在调用完成处理程序之前释放此资源。这能够调用相同的操作而不会重叠资源使用，从而避免增加系统中的峰值资源使用率。

错误处理

Boost.Asio 允许用户以两种不同的方式处理错误：使用错误代码或抛出异常。如果在调用 I/O 对象方法时传递对 boost::system::error_code 对象的引用，则实现将通过该变量传递错误；否则，将抛出异常。

已经按照第一种方法通过检查错误代码实现了几个示例，现在来看看如何捕获异常。

以下示例创建一个有效期为三秒的计时器，io_context 对象从后台线程 io_thread 运行。当计时器通过调用其 async_wait() 函数启动异步任务时，会传递 boost::asio::use_future 参数，因此该函数返回一个 Future 对象 fut，稍后在 try-catch 块中使用该对象来调用其 get() 函数并检索存储的结果或异常，正如第 6 章中学到的那样。启动后异步操作，主线程等待一秒钟，计时器通过调用其 cancel() 函数取消操作。由于这发生在到期时间（三秒）之前，因此会引发异常：

```
#include <boost/asio.hpp>
#include <chrono>
#include <future>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

int main() {
    boost::asio::io_context io_context;
    boost::asio::steady_timer timer(io_context, 1s);

```

```
    auto fut = timer.async_wait(
                        boost::asio::use_future);

    std::thread io_thread([&io_context]() {
                        io_context.run();
    });

    std::this_thread::sleep_for(3s);

    timer.cancel();

    try {
        fut.get();
        std::cout << "Timer expired successfully!\n";
    } catch (const boost::system::system_error& e) {
        std::cout << "Timer failed: "
                  << e.code().message() << '\n';
    }
    io_thread.join();
    return 0;
}
```

捕获类型为 boost::system::system_error 的异常，并输出其消息。如果计时器在异步操作完成后取消其操作（在本例中，通过使主线程休眠超过三秒），计时器将成功过期，并且不会抛出异常。

现在，了解了 Boost.Asio 的主要构建块以及如何相互作用，再来了解其实现背后的设计模式。

9.4. Reactor 和 Proactor 设计模式

使用事件处理应用程序时，可以遵循两种方法来设计并发解决方案：Reactor（反应堆）和 Proactor（前摄器）设计模式。这些模式描述了处理事件所遵循的机制，表明如何发起、接收、解复用和分派这些事件。当系统收集并排队来自不同资源的 I/O 事件时，解复用这些事件，将它们分离并分派给正确的处理程序。

Reactor 模式以同步、串行的方式对服务请求进行多路分解和分派。通常遵循非阻塞同步 I/O 策略，如果操作可以执行则返回结果，如果系统没有资源来完成操作则返回错误。

而 Proactor 模式则允许以高效的异步方式解复用和调度服务请求，立即将控制权返回给调用者，表明操作已启动。然后，当操作完成时，调用系统将通知调用者。因此，Proactor 模式将责任分配给两个任务：异步执行的长时间操作和处理结果并通常调用其他异步操作的完成处理程序。

Boost.Asio 通过使用以下元素实现了 Proactor 设计模式：

- 发起者：发起异步操作的 I/O 对象。
- 异步操作：由操作系统异步运行的任务。
- 异步操作处理器：执行异步操作并将结果排队到完成事件队列中。

- 完成事件队列：异步操作处理器推送事件，异步事件出队的事件队列。
- 异步事件多路分解器：这会阻止 I/O 上下文，等待事件，并将完成的事件返回给调用者。
- 完成处理程序：处理异步操作结果的可调用对象。
- Proactor：调用异步事件多路分解器来将事件从队列中取出，并将它们分派给完成处理程序。这就是 I/O 执行上下文所做的工作。

图 9.3 清楚地显示了所有这些元素之间的关系：

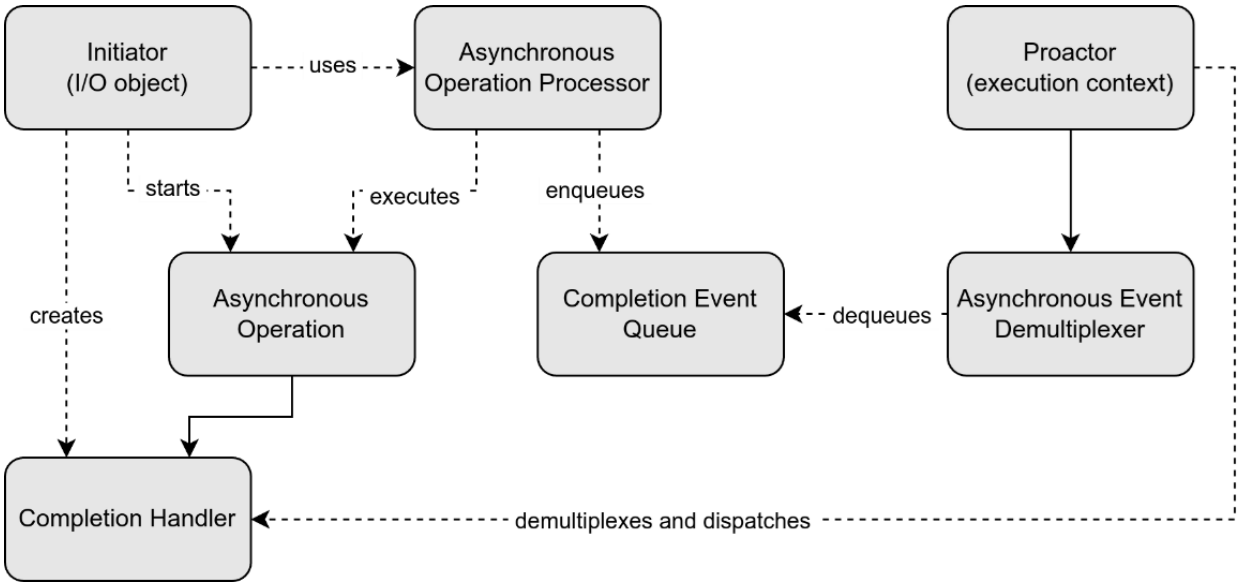

图 9.3 –前摄器设计模式

Proactor 模式在封装并发机制、简化应用程序同步、提高性能的同时，分离了关注点。

另一方面，无法控制异步操作的调度方式和时间，也无法控制操作系统执行这些操作的效率。此外，由于完成事件队列的存在，内存使用量也会增加，调试和测试的复杂性也会增加。

Boost.Asio 设计的另一个方面是执行上下文对象的线程安全性。现在，来深入研究 Boost.Asio 的线程工作原理。

9.5. 使用 Boost.Asio 进行线程处理

I/O 执行上下文对象是线程安全的，其方法可以从不同的线程安全地调用。所以可以使用单独的线程来运行阻塞的 io_context.run() 方法，并使主线程保持畅通，以继续执行其他不相关的任务。

现在，从如何使用线程的角度解释一下配置异步应用程序的不同方法。

9.5.1. 单线程方法

Boost.Asio 应用程序的起点和首选解决方案都应遵循单线程方法，其中 I/O 执行上下文对象在处理完成处理程序的同一线程中运行，这些处理程序必须简短且无阻塞。以下是与 I/ O 上下文在同一线程（即主线程）中运行的稳定计时器完成处理程序的示例：

```cpp
#include <boost/asio.hpp>
#include <chrono>
#include <iostream>

using namespace std::chrono_literals;

void handle_timer_expiry(
            const boost::system::error_code& ec) {
    if (!ec) {
        std::cout << "Timer expired!\n";
    } else {
        std::cerr << "Error in timer: "
                  << ec.message() << std::endl;
    }
}

int main() {
    boost::asio::io_context io_context;
    boost::asio::steady_timer timer(io_context,
                                    std::chrono::seconds(1));
    timer.async_wait(&handle_timer_expiry);
    io_context.run();
    return 0;
}
```

steady_timer 计时器调用异步 async_wait() 函数，设置 handle_timer_expiry() 完成处理程序，该函数在执行 io_context.run() 函数的同一线程中。当异步函数完成时，其完成处理程序将在同一线程中运行。

由于完成处理程序在主线程中运行，其执行应该很快，以避免冻结主线程和程序应执行的其他相关任务。下一节中，将介绍如何处理长时间运行的任务，或完成处理程序并保持主线程响应。

9.5.2. 线程化长时间运行的任务

对于长时间运行的任务，可以将逻辑保留在主线程中，但使用其他线程传递工作并将结果返回到主线程：

```cpp
#include <boost/asio.hpp>
#include <iostream>
#include <thread>

void long_running_task(boost::asio::io_context& io_context,
                       int task_duration) {
    std::cout << "Background task started: Duration = "
              << task_duration << " seconds.\n";
    std::this_thread::sleep_for(
                      std::chrono::seconds(task_duration));
```

```
    io_context.post([&io_context]() {
        std::cout << "Background task completed.\n";
        io_context.stop();
    });
}

int main() {
    boost::asio::io_context io_context;

    auto work_guard = boost::asio::make_work_guard
                                          (io_context);

    io_context.post([&io_context]() {
        std::thread t(long_running_task,
                      std::ref(io_context), 2);
        std::cout << "Detaching thread" << std::endl;
        t.detach();
    });

    std::cout << "Running io_context...\n";
    io_context.run();
    std::cout << "io_context exit.\n";
    return 0;
}
```

这个例子中，创建 io_context 之后，使用工作保护来避免 io_context.run( ) 函数在发布工作之前立即返回。

已发布的工作包括创建 t 线程以在后台运行 long_running_task( ) 函数。该 t 线程在 lambda 函数退出之前分离；否则，程序将终止。

后台任务函数中，当前线程休眠一段时间，然后将另一个任务发布到 io_context 对象中以输出消息并停止 io_context 本身。如果不调用 io_context.stop( )，事件处理循环将永远继续运行，程序将无法完成，io_context.run( ) 将继续因工作保护而阻塞。

9.5.3. 每线程一个 I/O 执行上下文对象

这种方法类似于单线程方法，其中每个线程都有自己的 io_context 对象，并处理简短且非阻塞的完成处理程序：

```
#include <boost/asio.hpp>
#include <chrono>
#include <iostream>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)
```

```cpp

using namespace std::chrono_literals;

void background_task(int i) {
    sync_cout << "Thread " << i << ": Starting...\n";
    boost::asio::io_context io_context;
    auto work_guard =
                boost::asio::make_work_guard(io_context);

    sync_cout << "Thread " << i << ": Setup timer...\n";
    boost::asio::steady_timer timer(io_context, 1s);
    timer.async_wait(
        [&](const boost::system::error_code& ec) {
        if (!ec) {
            sync_cout << "Timer expired successfully!"
                      << std::endl;
        } else {
            sync_cout << "Timer error: "
                      << ec.message() << '\n';
        }
        work_guard.reset();
    });

    sync_cout << "Thread " << i << ": Running io_context...\n";
    io_context.run();
}

int main() {
    const int num_threads = 4;
    std::vector<std::jthread> threads;

    for (auto i = 0; i < num_threads; ++i) {
        threads.emplace_back(background_task, i);
    }
    return 0;
}
```

此示例中，创建了四个线程，每个线程运行 `background_task()` 函数，其中创建一个 `io_context` 对象，并设置一个计时器，与其完成处理程序一起在一秒后超时。

9.5.4. 多线程使用一个 I/O 执行上下文对象

现在，只有一个 `io_context` 对象，正在从不同线程的不同 I/O 对象启动异步任务。这种情况下，可以从任何这些线程调用完成处理程序。以下是一个例子：

```cpp
#include <boost/asio.hpp>
#include <chrono>
#include <iostream>
#include <syncstream>
#include <thread>
#include <vector>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

void background_task(int task_id) {
    boost::asio::post([task_id]() {
        sync_cout << "Task " << task_id
                  << " is being handled in thread "
                  << std::this_thread::get_id()
                  << std::endl;
        std::this_thread::sleep_for(2s);
        sync_cout << "Task " << task_id
        << " complete.\n";
    });
}

int main() {
    boost::asio::io_context io_context;
    auto work_guard = boost::asio::make_work_guard(
                                    io_context);
    std::jthread io_context_thread([&io_context]() {
        io_context.run();
    });

    const int num_threads = 4;
    std::vector<std::jthread> threads;
    for (int i = 0; i < num_threads; ++i) {
        background_task(i);
    }

    std::this_thread::sleep_for(5s);
    work_guard.reset();

    return 0;
}
```

在此示例中，仅创建了一个 `io_context` 对象，并在单独的线程 `io_context_thread` 中运行。然后，创建另外四个后台线程，将工作发布到 `io_context` 对象中。最后，主线程等待五秒钟，让所有线程完成工作并重置工作保护。如果没有更多待处理工作，则让 `io_context.run()` 函数返回。当程序退出时，所有线程都会自动汇入（是 `std::jthread` 的实例）。

9.5.5. 单个 I/O 执行上下文并行完成工作

上例中，使用了一个唯一的 I/O 执行上下文对象，其 run() 函数从不同的线程调用。然后，每个线程在完成时发布完成处理程序在可用线程中执行的一些工作。

这是并行化一个 I/O 执行上下文所做工作的常用方法，即从多个线程调用其 run() 函数，将异步操作的处理分布在这些线程中。因为 io_context 对象提供了一个线程安全的事件调度系统，所以可能。

下面是另一个示例，其中创建了一个线程池，每个线程都运行 io_context.run()，使这些线程竞争从队列中拉取任务并执行。这种情况下，使用两秒后到期的计时器仅创建一个异步任务。其中一个线程将拾取任务并执行：

```
#include <boost/asio.hpp>
#include <iostream>
#include <thread>
#include <vector>

using namespace std::chrono_literals;

int main() {
    boost::asio::io_context io_context;

    boost::asio::steady_timer timer(io_context, 2s);
    timer.async_wait(
    [](const boost::system::error_code& /*ec*/) {
        std::cout << "Timer expired!\n";
    });

    const std::size_t num_threads =
                std::thread::hardware_concurrency();
    std::vector<std::thread> threads;
    for (std::size_t i = 0;
        i < std::thread::hardware_concurrency(); ++i) {
            threads.emplace_back([&io_context]() {
                io_context.run();
            });
    }
    for (auto& t : threads) {
        t.join();
    }
    return 0;
}
```

这种技术提高了可扩展性，应用程序可以更好地利用多个内核，并通过同时处理异步任务来减少延迟。此外，通过减少单线程代码处理许多同时 I/O 操作时产生的瓶颈，可以减少争用并提高吞吐量。

注意，完成处理程序在不同线程之间共享或修改共享资源，则也必须使用同步原语并且是线程

安全的。

此外，无法保证完成处理程序的执行顺序。由于许多线程可以同时运行，所以其中一个线程都可以先完成并调用其关联的完成处理程序。

当线程竞争从队列中提取任务时，如果线程池大小不是最佳的（理想情况下与硬件线程数匹配，如本例所示），则可能存在潜在的锁争用或上下文切换开销。

现在，是时候了解对象的生命周期如何影响使用 Boost.Asio 开发的异步程序的稳定性了。

9.6. 管理对象的生命周期

异步操作可能发生的主要灾难性问题之一是：当操作发生时，一些所需的对象已销毁。

因此，管理对象的生命周期至关重要。在 C++ 中，对象的生命周期从构造函数结束时开始，到析构函数开始时结束。保持对象活动的常见模式是让对象创建指向自身的共享指针实例，确保只要有指向它的共享指针，该对象就保持有效。

这种技术称为 shared-from-this，并使用自 C++11 起可用的 std::enable_shared_from_this 模板基类，提供了对象用来获取指向自身的共享指针的 shared_from_this() 方法。

9.6.1. 实现回显服务器——示例

通过创建一个回显服务器来了解它的工作原理。同时，将讨论这项技术，还将介绍如何使用 Boost.Asio 进行联网。

通过网络传输数据可能需要很长时间才能完成，并且可能会出现多个错误。这使得网络 I/O 服务成为 Boost.Asio 处理的一个特殊情况。网络 I/O 服务是第一个包含在库中的服务。

Boost.Asio 在业界的主要常见用途是开发网络应用程序，因为支持互联网协议 TCP、UDP 和 ICMP。该库还提供了基于 Berkeley Software Distribution（BSD）套接字 API 的套接字接口，允许使用底层接口开发高效且可扩展的应用程序。

然而，由于在本书对异步编程感兴趣，所以将重点介绍如何使用高级接口实现回显服务器。

回显服务器是一个监听特定地址和端口，并写回从该端口读取的所有内容的程序，将创建一个 TCP 服务器。

主程序将简单地创建一个 io_context 对象，通过传递 io_context 对象和要监听的端口号来设置 EchoServer 对象，然后调用 io_context.run() 来启动事件处理循环：

```
#include <boost/asio.hpp>
#include <memory>

constexpr int port = 1234;

int main() {
    try {
        boost::asio::io_context io_context;
        EchoServer server(io_context, port);
```

```
        io_context.run();
    } catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << "\n";
    }
    return 0;
}
```

当 EchoServer 初始化时，将开始监听传入的连接，通过使用 boost::asio::tcp::acceptor 对象来实现。此对象通过其构造函数接受 io_context 对象（与 I/O 对象一样）和 boost:: asio::tcp::endpoint 对象，该对象指示用于监听的连接协议和端口号。由于 boost::asio::tcp::v4 () 对象用于初始化端点对象，EchoServer 将使用的协议是 IPv4。未向端点构造函数指定 IP 地址，端点 IP 地址将是实际地址（对于 IPv4 为 INADDR_ANY，对于 IPv6 为 in6 addr_any)。接下来，实现 EchoServer 构造函数的代码如下：

```
using boost::asio::ip::tcp;
class EchoServer {
public:
    EchoServer(boost::asio::io_context& io_context,
               short port)
        : acceptor_(io_context,
                    tcp::endpoint(tcp::v4(),
                    port)) {
        do_accept();
    }

private:
    void do_accept() {
        acceptor_.async_accept([this](
                    boost::system::error_code ec,
                    tcp::socket socket) {
            if (!ec) {
                std::make_shared<Session>(
                std::move(socket))->start();
            }
            do_accept();
        });
    }

    tcp::acceptor acceptor_;
};
```

EchoServer 构造函数在设置接受器对象后调用 do_accept() 函数，其中调用 async_accept() 函数等待传入连接。当客户端连接到服务器时，操作系统通过 io_context 对象返回连接的套接字（boost::asio::tcp::socket）或错误。

如果没有错误并且建立了连接，则创建 Session 对象的共享指针，将套接字移动到 Session

对象中，Session 对象运行 start() 函数。

Session 对象封装了特定连接的状态，本例中为 socket_ 对象和 data_ 缓冲区。还使用 do_read() 和 do_write() 管理对该缓冲区的异步读取和写入，稍后将实现这两个函数。Session 继承自 std::enable_shared_from_this<Session>，所以 Session 对象创建指向自身的共享指针，确保只要至少有一个共享指针指向管理该连接的 Session 实例，会话对象就会在需要异步操作的整个生命周期内保持活动状态。此共享指针是在建立连接时，在 EchoServer 对象中的 do_accept() 函数中创建的。

以下是 Session 类的实现：

```
class Session
    : public std::enable_shared_from_this<Session>
{
public:
    Session(tcp::socket socket)
        : socket_(std::move(socket)) {}

    void start() { do_read(); }

private:
    static const size_t max_length = 1024;

    void do_read();
    void do_write(std::size_t length);

    tcp::socket socket_;
    char data_[max_length];
};
```

使用 Session 类，可以将管理连接的逻辑与管理服务器的逻辑分开。EchoServer 只需接受连接并为每个连接创建一个 Session 对象。这样，一个服务器就可以管理多个客户端，保持连接独立并异步管理。

Session 使用 do_read() 和 do_write() 函数来管理该连接的行为。当 Session 启动时，其 start() 函数会调用 do_read() 函数：

```
void Session::do_read() {
    auto self(shared_from_this());
    socket_.async_read_some(boost::asio::buffer(data_,
                                            max_length),
    [this, self](boost::system::error_code ec,
                 std::size_t length) {
        if (!ec) {
            do_write(length);
        }
    });
}
```

do_read() 函数创建指向当前会话对象 (self) 的共享指针，并使用套接字的 async_read_some() 异步函数将一些数据读入 data_buffer。如果成功，此操作将返回复制到 data_buffer 中的数据以及长度变量中读取的字节数。

然后，使用该长度变量调用 do_write()，使用 async_write() 函数将 data_ 缓冲区的内容异步写入套接字。当此异步操作成功时，通过再次调用 do_read() 函数重新启动循环：

```
void Session::do_write(std::size_t length) {
    auto self(shared_from_this());
    boost::asio::async_write(socket_,
                             boost::asio::buffer(data_,
                                                 length),
    [this, self](boost::system::error_code ec,
                 std::size_t length) {
        if (!ec) {
            do_read();
        }
    });
}
```

想知道为什么定义了 self 却没有使用？看起来 self 是多余的，但是当 lambda 函数按值捕获它时，会创建一个副本，从而增加指向 this 对象的共享指针的引用计数，确保在 lambda 处于活动状态时会话不会破坏。捕获 this 对象是为了在 lambda 函数中提供对其成员的访问。

作为练习，尝试实现一个 stop() 函数来中断 do_read() 和 do_write() 之间的循环。当所有异步操作完成并且 lambda 函数退出，self 对象将销毁，并且将不再有其他共享指针指向 Session 对象，因此会话将销毁。

这种模式确保在异步操作期间对对象的生命周期进行稳健和安全的管理，避免悬垂指针或早期破坏，从而导致不良行为或崩溃。

要测试此服务器，只需启动服务器，打开一个新终端，然后使用 telnet 命令连接到服务器并向其发送数据。作为参数，可以传递本地主机地址，表示我们正在连接到在同一台机器上运行的服务器（IP 地址为 127.0.0.1）和端口（在本例中为 1234）。

telnet 命令将启动并显示有关连接的一些信息，并表明我们需要按 Ctrl + } 键来关闭连接。

输入任何内容并按下 Enter 键将会把输入的行发送到回显服务器，该服务器将监听并发回相同的内容；在这个例子中，是“Hello world!”。

只需关闭连接并使用 quit 命令退出 telnet 即可退出返回终端：

```
$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello world!
Hello world!
telnet> quit
```

```
Connection closed.
```

此示例中，已经使用了缓冲区。让我们在下一节中进一步进行了解。

9.7. 使用缓冲区传输数据

缓冲区是 I/O 操作期间用于传输数据的连续内存区域。

Boost.Asio 定义了两种类型的缓冲区：可变缓冲区（boost::asio::mutable_buffer），其中可以写入数据；常量缓冲区（boost::asio::const_buffers），用于创建只读缓冲区。可变缓冲区可以转换为常量缓冲区，但不能转换为常量缓冲区。这两种类型的缓冲区都提供溢出保护。

还有 boost::buffer 函数来帮助从不同数据类型（指向原始内存和大小的指针、字符串（std::string）或普通旧数据（POD）结构数组或向量（意味着类型、结构或类没有用户定义的复制赋值运算符或析构函数，也没有私有或受保护的非静态数据成员）创建可变或常量缓冲区。例如，要从字符数组创建缓冲区。

可以使用以下方式创建：

```
char data[1024];
mutable_buffer buffer = buffer(data, sizeof(data));
```

另请注意，缓冲区的所有权和生存期是程序的责任，而不是 Boost.Asio 库的责任。

9.7.1. 分散-聚集操作

可以通过使用分散-集中操作来高效使用缓冲区，其中多个缓冲区一起用于接收数据（分散-读取）或发送数据（集中-写入）。

分散-读取：是将数据从唯一源读取到不同的不连续内存缓冲区的过程。

聚集-写入：是逆向的过程；数据从不同的不连续的内存缓冲区聚集并写入单个目的地。

这些技术可以减少系统调用或数据复制的次数，从而提高效率和性能。不仅用于 I/O 操作，还可用于其他用例，例如：数据处理、机器学习或并行算法（例如排序或矩阵乘法）。

为了允许分散-集中操作，可以将多个缓冲区一起传递到容器（std::vector、std::list、std::array 或 boost::array）内的异步操作。

下面是一个散射读取的例子，其中套接字将一些数据异步读入 buf1 和 buf2 缓冲区：

```
std::array<char, 128> buf1, buf2;
std::vector<boost::asio::mutable_buffer> buffers = {
    boost::asio::buffer(buf1),
    boost::asio::buffer(buf2)
};
socket.async_read_some(buffers, handler);
```

以下是如何实现聚集读取：

```
std::array<char, 128> buf1, buf2;
std::vector<boost::asio::const_buffer> buffers = {
    boost::asio::buffer(buf1),
    boost::asio::buffer(buf2)
};
socket.async_write_some(buffers, handler);
```

套接字执行相反的操作，将两个缓冲区中的一些数据写入套接字缓冲区以进行异步发送。

9.7.2. 流缓冲区

还可以使用流缓冲区来管理数据。流缓冲区由 boost::asio::basic_streambuf 类定义，该类基于 std::basic_streambuf 类，并在 <streambuf> 头文件中定义。允许动态缓冲区，其大小可以适应正在传输的数据量。

以下示例中看看流缓冲区如何与分散-集中操作协同工作。本例中，实现了一个 TCP 服务器，该服务器监听并接受来自给定端口的客户端连接，将客户端发送的消息读入两个流缓冲区，并将其内容打印到控制台。由于这里有兴趣了解流缓冲区和分散-集中操作，可通过使用同步操作来简化示例。

与上例一样，在 main() 函数中，使用 boost::asio::ip::tcp::acceptor 对象来设置 TCP 服务器用于接受连接的协议和端口。然后，在无限循环中，服务器使用该 acceptor 对象连接 TCP 套接字（boost::asio::ip::tcp::socket）并调用 handle_client() 函数：

```
#include <array>
#include <iostream>
#include <boost/asio.hpp>
#include <boost/asio/streambuf.hpp>

using boost::asio::ip::tcp;

constexpr int port = 1234;

int main() {
    try {
        boost::asio::io_context io_context;
        tcp::acceptor acceptor(io_context,
                               tcp::endpoint(tcp::v4(), port));
        std::cout << "Server is running on port "
                  << port << "...\n";

        while (true) {
            tcp::socket socket(io_context);
            acceptor.accept(socket);
            std::cout << "Client connected...\n";
```

```
22
23              handle_client(socket);
24              std::cout << "Client disconnected...\n";
25          }
26      } catch (std::exception& e) {
27          std::cerr << "Exception: " << e.what() << '\n';
28      }
29      return 0;
30  }
```

handle_client( ) 函数创建两个流缓冲区：buf1 和 buf2，并将它们添加到容器（在本例中为 std ::array）中，以用于分散-聚集操作。

然后，调用套接字的同步 read_some( ) 函数。此函数返回从套接字读取的字节数并将它们复制到缓冲区中。如果套接字连接出现问题，则会在错误代码对象中返回错误，服务器将输出错误消息并退出。

实现如下：

```
1   void handle_client(tcp::socket& socket) {
2       const size_t size_buffer = 5;
3       boost::asio::streambuf buf1, buf2;
4
5       std::array<boost::asio::mutable_buffer, 2> buffers = {
6           buf1.prepare(size_buffer),
7           buf2.prepare(size_buffer)
8       };
9
10      boost::system::error_code ec;
11      size_t bytes_recv = socket.read_some(buffers, ec);
12      if (ec) {
13          std::cerr << "Error on receive: "
14                    << ec.message() << '\n';
15
16          return;
17      }
18
19      std::cout << "Received " << bytes_recv << " bytes\n";
20
21      buf1.commit(5);
22      buf2.commit(5);
23
24      std::istream is1(&buf1);
25      std::istream is2(&buf2);
26      std::string data1, data2;
27      is1 >> data1;
28      is2 >> data2;
29
```

```
    std::cout << "Buffer 1: " << data1 << std::endl;
    std::cout << "Buffer 2: " << data2 << std::endl;
}
```

如果没有错误，则使用流缓冲区的 commit() 函数将五个字节传输到流缓冲区 buf1 和 buf2 中。使用 std::istream 对象提取这些缓冲区的内容并将其输出到控制台。

要执行此示例，需要打开两个终端。在一个终端中，执行服务器，在另一个终端中，执行 telnet 命令，如前所示。在 telnet 终端中，可以输入一条消息（例如，"Hello World"）。此消息将发送到服务器。然后服务器终端将显示以下内容：

```
Server is running on port 1234...
Client connected...
Received 10 bytes
Buffer 1: Hello
Buffer 2: Worl
Client disconnected...
```

可以看到，只有 10 个字节被处理并分配到两个缓冲区中。两个单词之间的空格字符被处理但在 iostream 对象解析输入时丢弃。

当传入数据的大小可变且事先未知时，流缓冲区非常有用。这些类型的缓冲区可以与固定大小的缓冲区一起使用。

9.8. 信号处理

信号处理能够捕获操作系统发送的信号，并在操作系统决定终止应用程序的进程之前正常关闭该应用程序。

Boost.Asio 为此目的提供了 boost::asio::signal_set 类，启动异步等待一个或多个信号的发生。

这是如何处理 SIGINT 和 SIGTERM 信号的示例：

```
#include <boost/asio.hpp>
#include <iostream>

int main() {
    try {
        boost::asio::io_context io_context;
        boost::asio::signal_set signals(io_context,
                                        SIGINT, SIGTERM);

        auto handle_signal = [&](
                const boost::system::error_code& ec,
                int signal) {
            if (!ec) {
                std::cout << "Signal received: "
```

```
                          << signal << std::endl;

                // Code to perform cleanup or shutdown.
                io_context.stop();
            }
        };

        signals.async_wait(handle_signal);
        std::cout << "Application is running. "
                  << "Press Ctrl+C to stop...\n";

        io_context.run();
        std::cout << "Application has exited cleanly.\n";
    } catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << '\n';
    }
    return 0;
}
```

信号对象是 signal_set，列出了程序等待的信号，SIGINT 和 SIGTERM。此对象有一个 async_wait() 方法，该方法异步等待这些信号的发生，并触发完成处理程序 handle_signal()。

与完成处理程序中通常的情况一样，handle_signal() 检查错误代码 ec，如果没有错误，则可能会执行一些清理代码以干净利落地退出程序。此示例中，只需调用 io_context.stop() 即可停止事件处理循环。

还可以使用 signals.wait() 方法同步等待信号。如果应用程序是多线程的，则信号事件处理程序必须与 io_context 对象在同一个线程中运行，通常是主线程。

下一节中，将介绍如何取消操作。

9.9. 取消操作

某些 I/O 对象（例如套接字或计时器）通过调用 close() 或 cancel() 方法在对象范围内取消未完成的异步操作。如果异步操作被取消，完成处理程序将收到带有 boost::asio::error::operation_aborted 代码的错误。

以下示例中,创建了一个计时器,并将其超时时间设置为 5 秒。但在主线程仅休眠两秒后,通过调用其 cancel() 方法取消了计时器,从而使用 boost::asio::error::operation_aborted 错误代码调用完成处理程序:

```
#include <boost/asio.hpp>
#include <chrono>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;
```

```
7
8  void handle_timeout(const boost::system::error_code& ec) {
9      if (ec == boost::asio::error::operation_aborted) {
10         std::cout << "Timer canceled.\n";
11     } else if (!ec) {
12         std::cout << "Timer expired.\n";
13     } else {
14         std::cout << "Error: " << ec.message()
15                   << std::endl;
16     }
17 }
18 int main() {
19     boost::asio::io_context io_context;
20     boost::asio::steady_timer timer(io_context, 5s);
21
22     timer.async_wait(handle_timeout);
23
24     std::this_thread::sleep_for(2s);
25     timer.cancel();
26
27     io_context.run();
28     return 0;
29 }
```

但若想要每个操作都取消，需要设置一个取消槽，当发出取消信号时，该槽将触发。这个取消信号/槽对组成了一个轻量级通道来传达取消操作。取消框架自 1.75 版起在 Boost.Asio 中可用。

这种方法实现了更灵活的取消机制，可以使用同一信号取消多个操作，并与 Boost.Asio 的异步操作无缝集成。同步操作只能通过使用前面描述的 cancel() 或 close() 方法取消；取消槽机制不支持。

修改前面的示例，并使用取消信号/槽来取消计时器。只需要修改在 main() 函数中取消计时器的方式，当执行异步 async_wait() 操作时，通过使用 boost::asio::bind_cancellation_slot() 函数将取消信号和完成处理程序中的槽绑定在一起，即可创建取消槽。

与之前一样，定时器的有效期为五秒。同样，主线程仅休眠两秒。这次，通过调用 cancel_signal.emit() 函数发出取消信号。该信号将触发对应的取消槽并使用 boost::asio::error::operation_aborted 错误代码执行完成处理程序，在控制台中输出"Timer expired." 的消息；参见以下内容：

```
1  int main() {
2      boost::asio::io_context io_context;
3      boost::asio::steady_timer timer(io_context, 5s);
4
5      boost::asio::cancellation_signal cancel_signal;
6
```

```
    timer.async_wait(boost::asio::bind_cancellation_slot(
        cancel_signal.slot(),
        handle_timeout
    ));

    std::this_thread::sleep_for(2s);
    cancel_signal.emit(
        boost::asio::cancellation_type::all);

    io_context.run();
    return 0;
}
```

发出信号时，必须指定取消类型，让目标操作知道应用程序需要什么，以及操作保证什么，从而控制取消的范围和行为。

各种取消类别如下：

- None：不执行取消。如果想测试是否应该发生取消，这会很有用。
- Terminal：该操作具有未指定的副作用，因此取消操作的唯一安全方法是关闭或销毁 I/O 对象，其结果是最终的，例如完成任务或交易。
- Partial：操作具有明确定义的副作用，完成处理程序可以采取所需的操作来解决问题，所以操作已部分完成并可以恢复或重试。
- Total 或 All：操作没有副作用。取消终端操作和部分操作，可通过停止所有正在进行的异步操作实现全面取消。

如果异步操作不支持取消类型，则取消请求将被丢弃。例如，计时器操作支持所有取消类别，但套接字仅支持 Total 和 All，所以如果尝试使用 Partial 取消来取消套接字异步操作，则将忽略此取消。如果 I/O 系统尝试处理不支持的取消请求，这可以防止出现未定义的行为。

此外，操作发起之后，但在操作开始之前或操作完成后，发出的取消请求均无效。

有时，需要按顺序运行一些工作。接下来我们将介绍如何使用 strand 来进行实现。

9.10. 使用 strands 序列化工作负载

strand 是严格顺序且非并发的完成处理程序调用。使用 strand，可以使用互斥锁或本书前面介绍的其他同步机制对异步操作进行排序，而无需显式锁定。strand 可以是隐式的，也可以是显式的。

如本章前面所示，如果仅从一个线程执行 boost::asio::io_context::run()，则所有事件处理程序将在隐式链中执行，将按顺序逐个排队并从 I/O 执行上下文触发。

另一种隐式链发生在存在链式异步操作时，其中一个异步操作会调度下一个异步操作，依此类推。本章中之前的一些示例已经使用了这种技术，但这里还有另一种。

如果没有错误，计时器将在 handle_timer_expiry() 事件处理程序中通过递归设置到期时间并调用 async_wait() 方法来不断重新启动自身：

```cpp
#include <boost/asio.hpp>
#include <chrono>
#include <iostream>

using namespace std::chrono_literals;

void handle_timer_expiry(boost::asio::steady_timer& timer,
                         int count) {
    std::cout << "Timer expired. Count: " << count
    << std::endl;
    timer.expires_after(1s);
    timer.async_wait([&timer, count](
                const boost::system::error_code& ec) {
        if (!ec) {
            handle_timer_expiry(timer, count + 1);
        } else {
            std::cerr << "Error: " << ec.message()
                      << std::endl;
        }
    });
}

int main() {
    boost::asio::io_context io_context;
    boost::asio::steady_timer timer(io_context, 1s);
    int count = 0;
    timer.async_wait([&](
                const boost::system::error_code& ec) {
        if (!ec) {
            handle_timer_expiry(timer, count);
        } else {
            std::cerr << "Error: " << ec.message()
                      << std::endl;
        }
    });
    io_context.run();
    return 0;
}
```

运行此示例将每秒打印”Timer expired. Count: <number>” 行，并且计数器在每一行上增加。

如果某些工作需要序列化但这些方法不合适，可以使用 boost::asio::strand 或其 I/O 上下文执行对象特化 boost::asio::io_context::strand 来使用显式 strand。使用这些 strand 对象发布的工作将按照进入 I/O 执行上下文队列的顺序序列化其处理程序执行。

以下示例中，将创建一个记录器，将多个线程的写入操作序列化为单个日志文件。

将从四个线程记录消息，每个线程写入五条消息。这里期望输出正确，但这次不使用互斥锁或同步机制。

首先，定义 Logger 类：

```cpp
#include <boost/asio.hpp>
#include <chrono>
#include <fstream>
#include <iostream>
#include <memory>
#include <string>
#include <thread>
#include <vector>

using namespace std::chrono_literals;

class Logger {
    public:
    Logger(boost::asio::io_context& io_context,
            const std::string& filename)
        : strand_(io_context), file_(filename
        , std::ios::out | std::ios::app)
    {
        if (!file_.is_open()) {
            throw std::runtime_error(
            "Failed to open log file");
        }
    }

    void log(const std::string message) {
        strand_.post([this, message](){
            do_log(message);
        });
    }

private:
    void do_log(const std::string message) {
        file_ << message << std::endl;
    }

    boost::asio::io_context::strand strand_;
    std::ofstream file_;
};
```

Logger 构造函数接受 I/O 上下文对象（用于创建 strand 对象 (boost::asio::io_context::strand)) 和 std::string（指定用于打开日志文件或创建日志文件（如果不存在）的日志文件名）。

日志文件已打开以添加新内容。如果在构造函数完成之前文件未打开（意味着访问或创建文件时出现问题），则构造函数将引发异常。

记录器还提供了接受 std::string 的公共 log() 函数，指定一条消息作为参数。此函数使

用 strand 将新工作发布到 io_context 对象中。它通过使用 lambda 函数来实现这一点，通过值捕获记录器实例（对象 this）和消息，并调用私有 do_log() 函数，其中 std::fstream 对象用于将消息写入输出文件。

程序中只会有一个 Logger 类的实例，由所有线程共享，线程将写入同一个文件。

再定义一个 worker() 函数，每个线程将运行该函数以将 num_messages_per_thread 消息写入输出文件：

```
void worker(std::shared_ptr<Logger> logger, int id) {
    for (unsigned i=0; i < num_messages_per_thread; ++i) {
        std::ostringstream oss;
        oss << "Thread " << id << " logging message " << i;
        logger->log(oss.str());
        std::this_thread::sleep_for(100ms);
    }
}
```

此函数接受指向 Logger 对象的共享指针和线程标识符，可使用前面解释过的 Logger 的公共 log() 函数输出所有消息。

为了交错线程执行并严格测试线程的工作方式，每个线程在写入每条消息后将休眠 100 毫秒。

最后，在 main() 函数中，启动 io_context 对象和一个工作保护，以避免提前退出 io_context。然后，创建一个指向 Logger 实例的共享指针，并传递前面解释的必要参数。

使用 worker() 函数并将共享指针传递给记录器以及每个线程的唯一标识符来创建线程池（std::jthread 对象的向量）。此外，还将运行 io_context.run() 函数的线程添加到线程池中。

下面的例子中，知道所有消息将在不到两秒的时间内输出，所以可以使用 io_context.run_for(2s) 让 io_context 仅在那段时间内运行。

当 run_for() 函数退出时，程序会向控制台输出“Done!”并完成任务：

```
const std::string log_filename = "log.txt";
const unsigned num_threads = 4;
const unsigned num_messages_per_thread = 5;

int main() {
    try {
        boost::asio::io_context io_context;
        auto work_guard = boost::asio::make_work_guard(
                                  io_context);
        std::shared_ptr<Logger> logger =
            std::make_shared<Logger>(
                io_context, log_filename);

        std::cout << "Logging "
                  << num_messages_per_thread
                  << " messages from " << num_threads
```

```
                  << " threads\n";

        std::vector<std::jthread> threads;
        for (unsigned i = 0; i < num_threads; ++i) {
            threads.emplace_back(worker, logger, i);
        }

        threads.emplace_back([&]() {
            io_context.run_for(2s);
        });
    } catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << '\n';
    }

    std::cout << "Done!" << std::endl;
    return 0;
}
```

运行此示例将显示以下输出：

```
Logging 5 messages from 4 threads
Done!
```

这是生成的 `log.txt` 日志文件的内容。由于每个线程的睡眠时间相同，因此所有线程和消息都按顺序排列：

```
Thread 0 logging message 0
Thread 1 logging message 0
Thread 2 logging message 0
Thread 3 logging message 0
Thread 0 logging message 1
Thread 1 logging message 1
Thread 2 logging message 1
Thread 3 logging message 1
Thread 0 logging message 2
Thread 1 logging message 2
Thread 2 logging message 2
Thread 3 logging message 2
Thread 0 logging message 3
Thread 1 logging message 3
Thread 2 logging message 3
Thread 3 logging message 3
Thread 0 logging message 4
Thread 1 logging message 4
Thread 2 logging message 4
Thread 3 logging message 4
```

如果删除工作保护，日志文件将只包含以下内容：

```
Thread 0 logging message 0
Thread 1 logging message 0
Thread 2 logging message 0
Thread 3 logging message 0
```

发生这种情况的原因是，第一批工作被迅速发布并从每个线程排队到 io_object 中，但 io_object 在完成调度工作保护，并在发布第二批消息之前通知完成处理程序之后退出。

如果还删除工作线程中的 sleep_for( ) 指令，则现在日志文件的内容如下：

```
Thread 0 logging message 0
Thread 0 logging message 1
Thread 0 logging message 2
Thread 0 logging message 3
Thread 0 logging message 4
Thread 1 logging message 0
Thread 1 logging message 1
Thread 1 logging message 2
Thread 1 logging message 3
Thread 1 logging message 4
Thread 2 logging message 0
Thread 2 logging message 1
Thread 2 logging message 2
Thread 2 logging message 3
Thread 2 logging message 4
Thread 3 logging message 0
Thread 3 logging message 1
Thread 3 logging message 2
Thread 3 logging message 3
Thread 3 logging message 4
```

以前，内容是按消息标识符排序的，现在则按线程标识符排序。因为现在，当一个线程启动并运行 worker( ) 函数时，会立即发布所有消息，没有延迟。因此，第一个线程（线程 0）在第二个线程有机会执行此操作之前将其所有工作排入队列，依此类推。

继续进一步的实验，当将内容发布到 strand 中时，可使用以下指令按值捕获了记录器实例和消息：

```
1  strand_.post([this, message]() { do_log(message); });
```

通过值捕获允许运行 do_log( ) 的 lambda 函数使用所需对象的副本，并使其保持活动状态。

假设，由于某种原因，决定使用以下指令通过引用捕获：

```
1  strand_.post([&]() { do_log(message); });
```

然后，生成的日志文件将包含不完整的日志消息，甚至不正确的字符，记录器正在从属于 do_log( ) 函数执行时，不再存在的消息对象的内存区域进行打印。

因此，始终假设异步变化；操作系统可能会执行一些无法控制的更改，所以始终要清楚我们的控制范围。

最后，除了使用 lambda 表达式并通过值捕获 this 和 message 对象外，还可以使用 std::bind：

```
strand_.post(std::bind(&Logger::do_log, this, message));
```

现在，了解了如何通过使用协程来简化之前实现的回显服务器，并通过添加从客户端退出连接的命令进行改进。

9.11. 协程

Boost.Asio 自 1.56.0 版本起还包含对协程的支持，并从 1.75.0 版本起支持本机协程。

使用协程可以简化程序的编写方式，因为无需添加完成处理程序并将程序流程拆分为不同的异步函数和回调。相反，使用协程时，程序遵循顺序结构，其中异步操作调用会暂停协程的执行。当异步操作完成时，协程将恢复，让程序从之前暂停的位置继续执行。

在较新的版本（1.75.0 以上）中，可以通过 co_await 使用本机 C++ 协程，等待协程内的异步操作，使用 boost::asio::co_spawn 启动协程，并使用 boost::asio::use_awaitable 让 Boost.Asio 知道异步操作将使用协程。在较早的版本（从 1.56.0 开始）中，可以使用 boost::asio::spawn() 和 yield 上下文来使用协程。由于更喜欢较新的方法，不仅因为它支持本机 C++20 协程，而且代码也更现代、更干净、更易读，将在本节中重点介绍这种方法。

再次实现回显服务器，但这次使用 Boost.Asio 的可等待接口和协程。还将添加一些改进，例如：支持在发送 QUIT 命令时从客户端关闭连接，显示如何在服务器端处理数据或命令，以及在抛出异常时停止处理连接并退出。

从实现 main() 函数开始，程序首先使用 boost::asio::co_spawn 创建一个基于协程的新线程。此函数接受执行上下文（io_context，也可以使用 strand）、具有 boost::asio::awaitable<R,E> 返回类型的函数（将用作协程的入口点）（将在下文实现和解释的 listener() 函数），以及将在线程完成时调用的完成令牌作为参数。如果想在不收到完成通知的情况下运行协程，可以传递 boost::asio::detached 令牌。

最后，通过调用 io_context.run() 开始处理异步事件。

如果发生异常，将被 try-catch 块捕获，并通过调用 io_context.stop() 停止事件处理循环：

```
#include <boost/asio.hpp>
#include <iostream>
#include <sstream>
#include <string>

using boost::asio::ip::tcp;

```

```
int main() {
    boost::asio::io_context io_context;
    try {
        boost::asio::co_spawn(io_context,
                    listener(io_context, 12345),
                    boost::asio::detached);
        io_context.run();
    } catch (std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        io_context.stop();
    }
    return 0;
}
```

listener() 函数接收一个 io_context 对象和侦听器将接受连接的端口号作为参数，使用前面解释过的接受器对象。它还必须具有 boost::asio::awaitable<R,E> 的返回类型，其中 R 是协程的返回类型，E 是可能抛出的异常类型。E 设置为默认值，因此未明确指定。

通过调用 async_accept 接受函数来接受连接。由于现在使用协程，需要为异步函数指定 boost::asio::use_awaitable，并使用 co_await 停止协程执行，直到异步任务完成后才恢复。

当侦听器协程任务恢复时，acceptor.async_accept() 返回一个套接字对象。协程继续生成一个新线程，使用 boost::asio::co_spawn 函数，执行 echo() 函数，并将套接字对象传递给它：

```
boost::asio::awaitable<void> listener(boost::asio::io_context& io_context, unsigned short
↪  port) {
    tcp::acceptor acceptor(io_context,
                           tcp::endpoint(tcp::v4(), port));
    while (true) {
        std::cout << "Accepting connections...\n";
        tcp::socket socket = co_await
            acceptor.async_accept(
                boost::asio::use_awaitable);

        std::cout << "Starting an Echo "
                  << "connection handler...\n";
        boost::asio::co_spawn(io_context,
                              echo(std::move(socket)),
                              boost::asio::detached);
    }
}
```

echo() 函数负责处理单个客户端连接。必须遵循与 listener() 函数类似的签名，需要返回 boost::asio::awaitable<R,E> 类型。如前所述，套接字对象从侦听器移入此函数。

该函数以异步方式从套接字读取内容并将其写回无限循环中，只有在收到 QUIT 命令或引发异常时才会完成。

异步读取是通过使用 socket.async_read_some() 函数完成的，该函数使用 boost::asio::buffer 将数据读入数据缓冲区并返回读取的字节数（bytes_read）。由于异步任务由协程管理，因此 bo ost::asio::use_awaitable 可传递给异步操作。然后，co_wait 只是指示协程引擎暂停执行，直到异步操作完成。

一旦接收到一些数据，协程就会恢复执行，检查是否真的有一些数据需要处理；否则，可通过退出循环来结束连接，从而退出 echo() 函数。

如果读取了数据，会将其转换为 std::string 以便于操作。会删除 \r\n 结尾（如果存在），并将字符串与 QUIT 进行比较。

如果存在 QUIT，将执行异步写入，发送"Good bye!" 消息，然后退出循环。否则，将接收到的数据发送回客户端。在这两种情况下，异步写入操作都是通过使用 boost::asio::async_write() 函数执行的，传递套接字、boost:asio::buffer 包装要发送的数据缓冲区，以及 boost::asio::use_awaitable，就像异步读取操作一样。

然后，再次使用 co_await 在执行操作时暂停协程的执行。完成后，协程将恢复并在新的循环迭代中重复以下步骤：

```
boost::asio::awaitable<void> echo(tcp::socket socket) {
    char data[1024];

    while (true) {
        std::cout << "Reading data from socket...\n";
        std::size_t bytes_read = co_await
                 socket.async_read_some(
                        boost::asio::buffer(data),
                        boost::asio::use_awaitable);

        if (bytes_read == 0) {
            std::cout << "No data. Exiting loop...\n";
            break;
        }

        std::string str(data, bytes_read);
        if (!str.empty() && str.back() == '\n') {
            str.pop_back();
        }
        if (!str.empty() && str.back() == '\r') {
            str.pop_back();
        }

        if (str == "QUIT") {
            std::string bye("Good bye!\n");
            co_await boost::asio::async_write(socket,
                        boost::asio::buffer(bye),
                        boost::asio::use_awaitable);
            break;
        }
```

```
31
32          std::cout << "Writing '" << str
33                    << "' back into the socket...\n";
34          co_await boost::asio::async_write(socket,
35                      boost::asio::buffer(data,
36                                          bytes_read),
37                      boost::asio::use_awaitable);
38      }
39  }
```

协程循环直到没有读取到数据，当客户端关闭连接、收到 QUIT 命令，或者发生异常时都会发生这种情况。

始终使用异步操作来确保服务器保持响应，即使同时处理多个客户端。

9.12. 总结

本章中，介绍了 Boost.Asio ，以及如何使用该库来管理处理操作系统管理的外部资源的异步任务。

为此，介绍了 I/O 对象和 I/O 执行上下文对象，并深入解释了它们如何工作和交互、如何访问和与 OS 服务通信、它们背后的设计原则，以及如何在单线程和多线程应用程序中正确使用它们。

还展示了 Boost.Asio 中可用的不同技术，使用 strands 序列化工作，管理异步操作所使用的对象的生命周期，如何启动、中断或取消任务，如何管理库使用的事件处理循环，以及如何处理操作系统发送的信号。

还介绍了与网络和协同程序相关的其他概念，并且还使用这个强大的库实现了一些有用的示例。

所有这些概念和示例更深入地了解如何管理 C++ 中的异步任务，以及广泛使用的库如何在后台工作以实现这一目标。

下一章中，将介绍另一个 Boost 库 Boost.Cobalt，它提供了丰富的高级接口来开发基于协程的异步软件。

9.13. 扩展阅读

- Boost.Asio official site: https://www.boost.org/doc/libs/1_85_0/doc/html/boost_asio.html
- Boost.Asio reference: https://www.boost.org/doc/libs/1_85_0/doc/html/boost_asio/reference.html
- Boost.Asio revision history: https://www.boost.org/doc/libs/1_85_0/doc/html/boost_asio/history.html
- Boost.Asio BSD socket API: https://www.boost.org/doc/libs/1_85_0/doc/html/boost_asio/overview/networking/bsd_sockets.html

- BSD socket API: https://web.mit.edu/macdev/Development/MITSupportLib/SocketsLib/Documentation/sockets.html
- The Boost C++ Libraries, Boris Schälig: https://theboostcpplibraries.com/boost.asio
- Thinking Asynchronously: Designing Applications with Boost.Asio, Christopher Kohlhoff: https://www.youtube.com/watch?v=D-lTwGJRxOo
- CppCon 2016: Asynchronous IO with Boost.Asio, Michael Caisse: https://www.youtube.com/watch?v=rwOv_tw2eA4
- Pattern-Oriented Software Architecture –Patterns for Concurrent and Networked Objects, Volume 2, D. Schmidt et al, Wiley, 2000
- Boost.Asio C++ Network Programming Cookbook, Dmytro Radchuk, Packt Publishing, 2016
- Proactor: An Object Behavioral Pattern for Demultiplexing and Dispatching handlers for Asynchronous events, Irfan Pyarali, Tim Harrison, Douglas C Schmidt, Thomas D Jordan. 1997
- Reactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Synchronous events, Douglas C Schmidt, 1995
- Input/Output Completion Port: https://en.wikipedia.org/wiki/Input/output_completion_port
- kqueue: https://en.wikipedia.org/wiki/Kqueue
- epoll: https://en.wikipedia.org/wiki/Epoll

第 10 章 使用 Boost.Cobalt 实现协程

前面的章节介绍了 C++20 协程和 Boost.Asio 库，这是使用 Boost 编写异步输入/输出（I/O）操作的基础。本章中，将探索 Boost.Cobalt，这是一个基于 Boost.Asio 的高级抽象，可简化使用协程的异步编程。

Boost.Cobalt 可编写清晰、可维护的异步代码，同时避免在 C++ 中手动实现协程的复杂性（如第 8 章所述）。Boost.Cobalt 与 Boost.Asio 完全兼容，可在项目中无缝结合这两个库。通过使用 Boost.Cobalt，可专注于构建应用程序，而不必担心协程的底层细节。

本章中，将介绍以下内容：

- Boost.Cobalt 库简介
- Boost.Cobalt 生成器
- Boost.Cobalt 任务和 promise
- Boost.Cobalt 通道
- Boost.Cobalt 同步函数

10.1. 技术要求

要构建和执行本章中的代码示例，需要支持 C++20 的编译器。我们使用了 Clang 18 和 GCC 14.2。

确保使用 Boost 1.84 或更新版本，并且 Boost 库是在 C++20 支持下编译的。撰写本书时，Cobalt 支持在 Boost 中还很新，并非所有预编译发行版都提供此组件。阅读本书时，情况通常会有所改善。如果由于某些原因，系统中的 Boost 库不满足这些要求，则必须从其源代码构建。使用早期版本（例如 C++17）进行编译将不会包含 Boost.Cobalt(严重依赖 C++20 协程)。

可以在以下 GitHub 存储库中找到完整的代码：https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP

本章的示例位于 Chapter_10 文件夹下。

10.2. Boost.Cobalt 库简介

第 8 章中介绍了 C++20 如何支持协程。显然，编写协程并不是一件容易的事，主要有两个原因：

- 在 C++ 中编写协程需要一定数量的代码才能使协程工作，但与想要实现的功能无关。例如，编写的用于生成斐波那契数列的协程非常简单，但必须实现包装器类型、promise，以及使其可用所需的所有函数。
- 开发普通的 C++20 协程需要很好地了解协程在 C++ 中实现的底层方面，编译器如何转换代码以实现保持协程状态所需的所有机制，以及如何调用和何时调用，函数必须详尽的实现。

即使没有那么多细节，异步编程也已经够难了。如果能专注于我们的程序，远离底层概念和代码，那就好了。我们看到了 C++23 如何引入 std::generator 来实现这一点，只编写生成器代码，让 C++ 标准库和编译器处理其余部分。预计这种协程支持将在下一个 C++ 版本中得到改进。

Boost.Cobalt 是 Boost C++ 库中包含的库之一，可做到这一点 - 避免实现协程细节。Boost.Cobalt 是在 Boost 1.84 中引入的，并且需要 C++20。基于 Boost.Asio，可以在程序中使用这两个库。

Boost.Cobalt 的目标是使用协程编写简单的单线程异步代码 - 可以在单个线程中同时执行多项操作的应用程序。当然，这里所说的同时是指并发，而不是并行，因为只有一个线程。通过使用 Boost.Asio 多线程功能，可以在不同的线程中执行协程，但在本章中，将重点介绍单线程应用程序。

10.2.1. 立即和惰性协程

介绍 Boost.Cobalt 实现的协程类型之前，需要定义两种协程：

- 立即协程：立即协程在调用后立即开始执行。协程逻辑立即开始运行，并按照其序列进行，直到到达暂停点（例如 co_await 或 co_yield）。协程的创建实际上启动了其处理，并且其主体中的任何副作用都将立即生效。
  当希望协程在创建后立即启动其工作（例如启动异步网络操作或准备数据）时，立即协程非常有用。
- 惰性协程：惰性协程会推迟执行，直到明确等待或使用。可以在不运行主体的情况下创建协程对象，直到调用者决定与其交互（通常通过使用 co_await 等待它）。
  当想要设置一个协程但延迟其执行直到满足某个条件或需要将其执行与其他任务协调时，惰性协程很有用。

定义了立即协程和惰性协程之后，我们将介绍在 Boost.Cobalt 中实现的不同类型的协程。

10.2.2. Boost.Cobalt 协程类型

Boost.Cobalt 实现了四种类型的协程。我们将在本节中介绍它们，然后在本章后面查看一些示例：

- Promise：这是 Boost.Cobalt 中的主要协程类型，用于实现返回单个值的异步操作（调用 co_return）。它是一个立即协程，支持 co_await，允许异步暂停和继续。例如，promise 可用于执行网络调用，完成后将返回其结果而不会阻止其他操作。
- 任务：任务是 Promise 的惰性版本，直到明确等待时才会开始执行。提供了更大的灵活性来控制协程的运行时间和方式。等待时，任务开始执行，允许延迟处理异步操作。
- 生成器：在 Boost.Cobalt 中，生成器是唯一可以产生值的协程类型。每个值都使用 co_yield 单独产生。其功能类似于 C++23 中的 std::generator，但允许使用 co_await 进行等待（std::generator 不允许）。

- 分离：这是一个可以使用 co_await 但不能使用 co_return 值的立即协程。它无法恢复，通常也不会等待。

目前为止，我们介绍了 Boost.Cobalt。定义了什么是急切协程和惰性协程，然后定义了库中的四种主要协程类型。

下一节中，将深入探讨与 Boost.Cobalt 相关的最重要的主题之一 - 生成器；还将实现一些简单的生成器示例。

10.3. Boost.Cobalt 生成器

如第 8 章所述，生成器协程是专门设计用于增量产生值的协程。产生每个值后，协程会自行挂起，直到调用者请求下一个值。在 Boost.Cobalt 中，生成器的工作方式相同，是唯一可以产生值的协程类型。当需要协程随时间产生多个值时，生成器就必不可少了。

Boost.Cobalt 生成器的一个关键特性是默认即时执行，所以在调用后立即开始执行。此外，这些生成器为异步，允许使用 co_await，这与 C++23 中引入的 std::generator 有一个重要区别，后者是惰性执行的，不支持 co_await。

10.3.1. 一个基本的例子

从最简单的 Boost.Cobalt 程序开始。此示例不是生成器，但将借助其解释一些重要细节：

```
#include <iostream>

#include <boost/cobalt.hpp>

boost::cobalt::main co_main(int argc, char* argv[]) {
    std::cout << "Hello Boost.Cobalt\n";
    co_return 0;
}
```

上面的代码中，可观察到以下情况：

- 要使用 Boost.Cobalt，必须包含 <boost/cobalt.hpp> 头文件。
- 还必须将 Boost.Cobalt 库链接到应用程序。这里提供了一个 CMakeLists.txt 文件来执行此操作，不仅针对 Boost.Cobalt，还针对所有必需的 Boost 库。要明确链接 Boost.Cobalt（即并非所有必需的 Boost 库），只需将以下行添加到您的 CMakeLists.txt 文件：

```
target_link_libraries(${EXEC_NAME} Boost::cobalt)
```

- 使用 co_main 函数。Boost.Cobalt 引入了一个名为 co_main 的基于协程的入口点，而不是通常的 main 函数。此函数可以使用协程特定的关键字，例如 co_return。Boost.Cobalt 在内部实现了所需的 main 函数。

使用 co_main 可以将程序的主函数（入口点）实现为协程，从而能够调用 co_await 和 co_return。请记住第 8 章中的内容，主函数不能是协程。

如果无法更改当前的主函数，则可以使用 Boost.Cobalt。只需从 main 调用一个函数，该函数将成为使用 Boost.Cobalt 的异步代码的顶层函数。

Cobalt 正在做的事情：实现了一个 main 函数，是程序的入口点，并且该（对您隐藏的）main 函数调用 co_main。

使用自己的主函数的最简单方法：

```
cobalt::task<int> async_task() {
    // your code here
    // ···
    return 0;
}

int main() {
    // main function code
    // ···
    return cobalt::run(async_code());
}
```

该示例只是输出了一条问候消息，然后调用 co_await 返回 0。在所有后续示例中，将遵循此模式：包括 <boost/cobalt.hpp> 头文件并使用 co_main 而不是 main。

10.3.2. Boost.Cobalt 简单生成器

利用前面的基本示例所掌握的知识，可实现一个非常简单的生成器协程：

```
#include <chrono>
#include <iostream>

#include <boost/cobalt.hpp>

using namespace std::chrono_literals;

using namespace boost;

cobalt::generator<int> basic_generator()
{
    std::this_thread::sleep_for(1s);
    co_yield 1;
    std::this_thread::sleep_for(1s);
    co_return 0;
}

cobalt::main co_main(int argc, char* argv[]) {
```

```
    auto g = basic_generator();
    std::cout << co_await g << std::endl;
    std::cout << co_await g << std::endl;
    co_return 0;
}
```

上面的代码展示了一个简单的生成器，产生一个整数值（使用 co_yield）并返回另一个整数值（使用 co_return）。

cobalt::generator 是一个结构模板：

```
template<typename Yield, typename Push = void>
struct generator
```

两个参数类型如下：

- Yield：生成的对象类型
- Push：输入参数类型（默认为 void）

co_main 函数在使用 co_await（调用者等待值可用）获取这两个数字后，会输出这两个数字。我们引入了一些延迟来模拟生成器生成数字必须进行的处理。

第二个生成器将产生一个整数的平方：

```
#include <chrono>
#include <iostream>

#include <boost/cobalt.hpp>

using namespace std::chrono_literals;

using namespace boost;

cobalt::generator<int, int> square_generator(int x){
    while (x != 0) {
        x = co_yield x * x;
    }

    co_return 0;
}

cobalt::main co_main(int argc, char* argv[]){
    auto g = square_generator(10);

    std::cout << co_await g(4) << std::endl;
    std::cout << co_await g(12) << std::endl;
    std::cout << co_await g(0) << std::endl;

    co_return 0;
```

```
}
```

`square_generator` 得出 `x` 参数的平方。这显示了如何将值推送到 `Boost.Cob alt` 生成器。在 `Boost.Cobalt` 中，将值推送到生成器则为传递参数（上例中，传递的参数是整数）。

本例中的生成器虽然正确，但可能会令人困惑。请看以下代码行：

```
auto g = square_generator(10);
```

这将创建以 `10` 作为初始值的生成器对象。然后，查看以下代码行：

```
std::cout << co_await g(4) << std::endl;
```

这将打印 `10` 的平方并将 `4` 推送到生成器，输出的值不是传递给生成器的值的平方。这是因为生成器用一个值（在此示例中为 `10`）初始化，并且当调用者调用 `co_await` 传递另一个值时，会生成平方值。生成器在收到新值 `4` 时将产生 `100`，然后在收到值 `12` 时将产生 `16`，依此类推。

`Boost.Cobalt` 生成器是立即的，但也可以让它在开始执行时立即等待（`co_await`）。以下示例显示了如何执行此操作：

```
#include <iostream>
#include <boost/cobalt.hpp>

boost::cobalt::generator<int, int> square_generator() {
    auto x = co_await boost::cobalt::this_coro::initial;
    while (x != 0) {
        x = co_yield x * x;
    }
    co_return 0;
}

boost::cobalt::main co_main(int, char*[]) {
    auto g = square_generator();

    std::cout << co_await g(4) << std::endl;
    std::cout << co_await g(10) << std::endl;
    std::cout << co_await g(12) << std::endl;
    std::cout << co_await g(0) << std::endl;

    co_return 0;
}
```

该代码与前面的示例非常相似，但也有一些区别：

- 不传递参数的情况下创建生成器：

  ```
  auto g = square_generator();
  ```

- 看一下生成器代码的第一行：

```
auto x = co_await boost::cobalt::this_coro::initial;
```

这使得生成器等待第一个压入的整数。这表现为一个惰性生成器（实际上，立即开始执行，生成器立即执行，但它做的第一件事是等待一个整数）。

- 生成的值期望从代码中得到：

```
std::cout << co_await g(10) << std::endl;
```

这将打印 100，而不是之前输入的整数的平方。

这里总结一下这个示例的作用：co_main 函数调用 square_generator 协程来生成整数值的平方。生成器协程在开始时暂停自身，等待第一个整数，并在产生每个平方后暂停自身。这个示例故意很简单，只是为了说明如何使用 Boost.Cobalt 编写生成器。

上述程序的一个重要特点是它在单线程中运行，所以 co_main 和生成器协程会相继运行。

10.3.3. 斐波那契数列生成器

本节中,将实现一个类似于第 8 章中实现的斐波那契数列生成器。体验到使用 Boost.Cobalt 编写生成器协程，比使用纯 C++20 容易得多。

我们编写了两个版本的生成器。第一个版本计算斐波那契数列的任意项。推送想要生成的项，然后就得到了它。此生成器使用 lambda 作为斐波那契计算器：

```
boost::cobalt::generator<int, int> fibonacci_term() {
    auto fibonacci = [](int n) {
        if (n < 2) {
            return n;
        }

        int f0 = 0;
        int f1 = 1;
        int f;

        for (int i = 2; i <= n; ++i) {
            f = f0 + f1;
            f0 = f1;
            f1 = f;
        }

        return f;
    };

    auto x = co_await boost::cobalt::this_coro::initial;
    while (x != -1) {
        x = co_yield fibonacci(x);
    }
```

```
24
25      co_return 0;
26  }
```

前面的代码中，了解到这个生成器与我们在上一节中实现的用于计算数字平方的生成器非常相似。在协程的开头，有以下内容：

```
1  auto x = co_await boost::cobalt::this_coro::initial;
```

这行代码暂停协程以等待第一个输入值。

然后有以下内容：

```
1  while (x != -1) {
2      x = co_yield fibonacci(x);
3  }
```

这将生成请求的斐波那契数列项，并暂停自身，直到请求下一个项。当请求的项不等于 `-1` 时，可以继续请求更多值，直到按下 `-1` 终止协程。

下一个版本的斐波那契生成器将根据要求生成无限数量的项，可以将此生成器视为始终准备生成另一个斐波那契数列：

```
1  boost::cobalt::generator<int> fibonacci_sequence() {
2      int f0 = 0;
3      int f1 = 1;
4      int f = 0;
5
6      while (true) {
7          co_yield f0;
8
9          f = f0 + f1;
10         f0 = f1;
11         f1 = f;
12     }
13 }
```

上述代码很容易理解：协程产生一个值并将自身挂起，直到请求另一个值，协程计算出新值并产生它并再次陷入无限循环中。

可以看到协程的优势：可以在需要时逐个生成斐波那契数列的项。不需要保留任何状态来生成下一个项，因为状态保存在协程中。

需要注意的是，即使函数执行了无限循环，由于它是协程，会一次又一次地暂停和恢复，避免阻塞当前线程。

10.4. Boost.Cobalt 任务和 promise

正如本章中已经看到的，Boost.Cobalt promise 是返回一个值的立即协程，而 Boost.Cobalt 任务是惰性版本的 promise。

我们可以将它们视为不会像生成器那样产生多个值的函数，可以反复调用 Promise 来获取多个值，但调用之间不会保留状态（就像生成器一样）。基本上，Promise 是一个可以使用 co_await 的协程（也可以使用 co_return）。

Promise 的不同用例包括套接字侦听器接收网络数据包、处理数据包、查询数据库，然后从数据中生成一些结果。一般来说，其功能需要异步等待某个结果，然后对该结果执行某些处理（或者可能只是将其返回给调用者）。

第一个例子是一个简单的 promise，可生成一个随机数（也可以用生成器来完成）：

```
#include <iostream>
#include <random>

#include <boost/cobalt.hpp>

boost::cobalt::promise<int> random_number(int min, int max) {
    std::random_device rd;
    std::mt19937 gen(rd());

    std::uniform_int_distribution<> dist(min, max);
    co_return dist(gen);
}

boost::cobalt::promise<int> random(int min, int max) {
    int res = co_await random_number(min, max);
    co_return res;
}

boost::cobalt::main co_main(int, char*[]) {
    for (int i = 0; i < 10; ++i) {
        auto r = random(1, 100);
        std::cout << "random number between 1 and 100: "
                  << co_await r << std::endl;
    }
    co_return 0;
}
```

上面的代码中，编写了三个协程：

- co_main：Boost.Cobalt 中，co_main 是一个协同程序，调用 co_return 来返回一个值。
- random()：此协程向调用者返回一个随机数。使用 co_await 调用 random() 来生成随机数，异步等待随机数生成。
- random_number()：该协程在两个值（最小值和最大值）之间生成一个均匀分布的随机数，

并将其返回给调用者。random_number() 也是一个 promise。

以下协程返回一个随机数的 std::vector<int>。循环调用 co_await random_number() 来生成一个包含 n 个随机数的向量：

```
boost::cobalt::promise<std::vector<int>> random_vector(int min, int
max, int n) {
    std::vector<int> rv(n);
    for (int i = 0; i < n; ++i) {
        rv[i] = co_await random_number(min, max);
    }
    co_return rv;
}
```

上述函数返回 std::vector<int> 的 promise。要访问该向量，需要调用 get()：

```
auto v = random_vector(1, 100, 20);
for (int n : v.get()) {
    std::cout << n << " ";
}
std::cout << std::endl;
```

上述代码打印了 v 向量的元素。要访问该向量，需要调用 v.get()。

实现第二个示例来说明 promise 和任务的执行有何不同：

```
#include <chrono>
#include <iostream>
#include <thread>

#include <boost/cobalt.hpp>

void sleep(){
    std::this_thread::sleep_for(std::chrono::seconds(2));
}

boost::cobalt::promise<int> eager_promise(){
    std::cout << "Eager promise started\n";
    sleep();
    std::cout << "Eager promise done\n";
    co_return 1;
}

boost::cobalt::task<int> lazy_task(){
    std::cout << "Lazy task started\n";
    sleep();
    std::cout << "Lazy task done\n";
    co_return 2;
}
```

```
24
25  boost::cobalt::main co_main(int, char*[]){
26      std::cout << "Calling eager_promise...\n";
27      auto promise_result = eager_promise();
28      std::cout << "Promise called, but not yet awaited.\n";
29
30      std::cout << "Calling lazy_task...\n";
31      auto task_result = lazy_task();
32      std::cout << "Task called, but not yet awaited.\n";
33
34      std::cout << "Awaiting both results...\n";
35      int promise_value = co_await promise_result;
36      std::cout << "Promise value: " << promise_value
37                << std::endl;
38
39      int task_value = co_await task_result;
40      std::cout << "Task value: " << task_value
41                << std::endl;
42
43      co_return 0;
44  }
```

这个例子中，实现了两个协程：一个 Promise 和一个 Task。Promise 是 Eager 类型的，它在调用时立即开始执行。而 Task 是 Lazy 类型的，在调用后就会暂停。

运行程序时，会输出所有消息，可确切地知道协程是如何执行的。

co_main() 前三行执行完成后，输出如下：

```
Calling eager_promise...
Eager promise started
Eager promise done
Promise called, but not yet awaited.
```

从这些消息中，知道 promise 已经执行了，直到调用 co_return。

执行 co_main() 的接下来三行之后，输出有以下新消息：

```
Calling lazy_task...
Task called, but not yet awaited.
```

这里，我们看到任务尚未执行。这是一个惰性协程，在调用后会立即挂起，并且此协程尚未输出任何消息。

再执行三行 co_main()，程序输出的新消息如下：

```
Awaiting both results...
Promise value: 1
```

对 promise 的 co_await 调用给出了其结果（在本例中设置为 1）并且其执行结束。

最后，在任务上调用 co_await，然后它执行并返回其值（在本例中设置为 2）。输出如下：

```
Lazy task started
Lazy task done
Task value: 2
```

此示例显示了任务如何惰性化并开始暂停，并且仅当调用者对其调用 co_await 时才恢复执行。

本节中，与生成器的情况一样，使用 Boost.Cobalt 编写 promise 和任务协程比仅使用纯 C++ 要容易得多，不需要编写 C++ 实现协程所需的所有支持代码。还了解了任务和 promise 的区别。

下一节中，将研究通道的示例，它是生产者/消费者模型中两个协程之间的通信机制。

10.5. Boost.Cobalt 通道

Boost.Cobalt 中，通道为协程提供了一种异步通信方式，允许以安全高效的方式在生产者和消费者协程之间传输数据。其受到 Golang 通道的启发，允许通过消息传递进行通信，从而促进通过通信共享内存的范式。

通道是一种机制，通过该机制，值从一个协程（生产者）异步传递到另一个协程（消费者）。这种通信是非阻塞的，所以协程可以在等待通道上的数据可用时，或将数据写入容量有限的通道时暂停执行。澄清一下：如果阻塞是指协程暂停，则读取和写入操作都可能阻塞，具体取决于缓冲区大小，但另一方面，从线程的角度来看，这些操作不会阻塞线程。

如果缓冲区大小为零，则读取和写入需要同时发生并充当会合点（同步通信）。如果通道大小大于零且缓冲区未满，则写入操作不会暂停协程。同样，如果缓冲区不为空，则读取操作也不会暂停。

与 Golang 通道类似，Boost.Cobalt 通道是强类型的。通道是为特定类型定义的，并且只能通过该类型发送数据。例如，int 类型的通道（boost::cobalt::channel<int>）只能传输整数。

来看一个通道的示例：

```
#include <iostream>
#include <boost/cobalt.hpp>
#include <boost/asio.hpp>
boost::cobalt::promise<void> producer(boost::cobalt::channel<int>& ch)
{
    for (int i = 1; i <= 10; ++i) {
        std::cout << "Producer waiting for request\n";
        co_await ch.write(i);
        std::cout << "Producing value " << i << std::endl;
    }
    std::cout << "Producer end\n";
    ch.close();
    co_return;
}
```

```
boost::cobalt::main co_main(int, char*[]) {
    boost::cobalt::channel<int> ch;
    auto p = producer(ch);
    while (ch.is_open()) {
        std::cout << "Consumer waiting for next number \n";
        std::this_thread::sleep_for(std::chrono::seconds(5));
        auto n = co_await ch.read();
        std::cout << "Consuming value " << n << std::endl;
        std::cout << n * n << std::endl;
    }
    co_await p;
    co_return 0;
}
```

这个例子中，创建一个大小为 0 的通道和两个协程：生产者 promise 和充当消费者的 co_main()。生产者将整数写入通道，消费者将其读回并输出它们的平方。

我们添加了 std::this_thread::sleep 来延迟程序执行，因此能够看到程序运行时发生的情况。来看一下示例输出的摘录，看看它是如何工作的：

```
Producer waiting for request
Consumer waiting for next number
Producing value 1
Producer waiting for request
Consuming value 1
1
Consumer waiting for next number
Producing value 2
Producer waiting for request
Consuming value 2
4
Consumer waiting for next number
Producing value 3
Producer waiting for request
Consuming value 3
9
Consumer waiting for next number
```

消费者和生产者都等待下一个操作发生，生产者将始终等待消费者请求下一个项目。这基本上就是生成器的工作方式，也是使用协程的异步代码中的常见模式。

消费者执行以下代码行：

```
auto n = co_await ch.read();
```

然后，生产者将下一个数字写入通道并等待下一个请求。这在以下代码行中完成：

```
co_await ch.write(i);
```

可以在前面的输出摘录的第四行中看到生产者如何返回等待下一个请求。Boost.Cobalt 通道使编写这种异步代码非常干净且易于理解。该示例显示两个协程通过通道进行通信。

下一节将介绍同步函数 - 等待多个协程的机制。

10.6. Boost.Cobalt 同步函数

以前，实现了协程，并且在每次调用 co_await 时，只对一个协程进行调用，所以只等待一个协程的结果。Boost.Cobalt 具有允许等待多个协程的机制，这些机制称为同步函数。

Boost.Cobalt 中实现了四个同步函数：

- race：等待一组协程中的一个完成，但其以伪随机方式执行。此机制有助于避免协程的匮乏，确保一个协程不会主导其他协程的执行流程。当有多个异步操作并且希望第一个完成以确定流程时，race 将允许准备好的协程以不确定的顺序继续执行。
  当有多个任务（一般意义上的任务，而不是 Boost.Cobalt 任务）并且有兴趣首先完成一个任务，而不优先考虑哪一个，但想防止一个协程在同时准备就绪的情况下总是获胜时将使用竞争。
- join：等待给定集合中的所有协程完成并将其结果作为值返回。如果协程抛出异常，join 会将异常传播给调用者。这是一种从多个异步操作中收集结果的方法，这些操作必须全部完成才能继续。
  当需要多个异步操作的结果在一起，并且希望在其中任何一个失败时抛出错误时，可使用 join。
- gather：与 join 类似，会等待一组协程完成，但其处理异常的方式不同。gather 不会在其中一个协程失败时立即抛出异常，而是会单独捕获每个协程的结果，所以可以单独检查每个协程的结果（成功或失败）。
  当需要所有异步操作完成，但想要单独捕获所有结果和异常以分别处理它们时，可使用 gather。
- left_race：与 race 类似，但具有确定性行为。它从左到右评估协程，并等待第一个协程准备就绪。当协程完成的顺序很重要，并且希望根据提供协程的顺序确保可预测的结果时，这会很有用。
  当有多个潜在结果并且需要按照提供的顺序支持第一个可用的协同程序时，会使行为比 race 更可预测，可使用 left_race。

本节中，将探讨 join 和 gather 函数的示例，这两个函数都会等待一组协程完成。它们之间的区别在于，如果协程抛出异常，join 就会抛出异常，而 gather 始终会返回所有等待的协程的结果。对于 gather 函数，每个协程的结果要么是错误（缺失值），要么是值。join 返回一个值元组或抛出异常；gather 返回一个可选值元组，如果发生异常（可选变量未初始化），则该元组没有值。

以下示例的完整代码位于 GitHub 库中，书中将重点介绍主要部分。

我们定义了一个简单的函数来模拟数据处理，这只是一个延迟。如果传递大于 5,000 毫秒的延迟，该函数将引发异常：

```
boost::cobalt::promise<std::chrono::milliseconds::rep>
process(std::chrono::milliseconds ms) {
    if (ms > std::chrono::milliseconds(5000)) {
        throw std::runtime_error("delay throw");
    }

    boost::asio::steady_timer tmr{ co_await boost::cobalt::this_coro::executor, ms };
    co_await tmr.async_wait(boost::cobalt::use_op);
    co_return ms.count();
}
```

该功能是 Boost.Cobalt 的 promise。

现在，在代码的下一部分中，将等待这个 promise 的三个实例运行：

```
auto result = co_await boost::cobalt::join(process(100ms),
                                           process(200ms),
                                           process(300ms));

std::cout << "First coroutine finished in: "
          << std::get<0>(result) << "ms\n";
std::cout << "Second coroutine took finished in: "
          << std::get<1>(result) << "ms\n";
std::cout << "Third coroutine took finished in: "
          << std::get<2>(result) << "ms\n";
```

上述代码调用 join 等待三个协程执行完毕，然后打印出执行时间。结果是一个元组，为了代码尽可能简洁，只对每个元素调用 std::get<i>(result)。这样，所有执行时间都在有效范围内，也没有抛出异常，因此可以得到所有执行协程的结果。

如果抛出异常，那么将不会得到任何值：

```
try {
    auto result throw = co_await
    boost::cobalt::join(process(100ms),
                        process(20000ms),
                        process(300ms));
}
catch (...) {
    std::cout << "An exception was thrown\n";
}
```

上述代码将抛出异常，因为第二个协程收到的处理时间超出了有效范围，将输出一条错误消息。当调用 join 函数时，希望所有协程都被视为处理的一部分，并且如果发生异常，则整个处理失败。

如果需要获取每个协程的所有结果，将使用 gather 函数：

```
try{
    auto result throw =
    boost::cobalt::co_await lt::gather(process(100ms),
                                       process(20000ms),
                                       process(300ms));

    if (std::get<0>(result throw).has value()) {
        std::cout << "First coroutine took: "
                  << *std::get<0>(result throw)
                  << "msec\n";
    }
    else {
        std::cout << "First coroutine threw an exception\n";
    }
    if (std::get<1>(result throw).has value()) {
        std::cout << "Second coroutine took: "
                  << *std::get<1>(result throw)
                  << "msec\n";
    }
    else {
        std::cout << "Second coroutine threw an exception\n";
    }
    if (std::get<2>(result throw).has value()) {
        std::cout << "Third coroutine took: "
                  << *std::get<2>(result throw)
                  << "msec\n";
    }
    else {
        std::cout << "Third coroutine threw an exception\n";
    }
}
catch (...) {
    // this is never reached because gather doesn't throw exceptions
    std::cout << "An exception was thrown\n";
}
```

我们把代码放在了 try-catch 块中，但没有抛出任何异常。gather 函数返回一个可选值的元组，需要检查每个协程是否返回了一个值（可选值是否有值）。

当希望协程在成功执行时返回一个值时，则使用 gather。

这些 join 和 gather 函数的示例结束了对 Boost.Cobalt 同步函数的介绍。

10.7. 总结

本章中，了解了如何使用 Boost.Cobalt 库实现协程，是最近才添加到 Boost 中的，关于它的信息并不多。它简化了使用协程开发异步代码的过程，避免了编写 C++20 协程所需的底层代码。

我们研究了主要的图书馆概念并开发了一些简单的示例来理解它们。

使用 Boost.Cobalt，使用协程编写异步代码变得简单。在 C++ 中编写协程的所有低级细节都由库实现，可以只关注我们想要在程序中实现的功能。

下一章中，将介绍如何调试异步代码。

10.8. 扩展阅读

- Boost.Cobalt reference: Boost.Cobalt reference guide (https://www.boost.org/doc/libs/1_86_0/libs/cobalt/doc/html/index.html#overview)
- A YouTube video on Boost.Cobalt: Using coroutines with Boost.Cobalt (https://www.youtube.com/watch?v=yElSdUqEvME)

第五部分 异步编程的调试、测试和性能优化

最后一部分中，将重点介绍调试、测试和优化多线程和异步程序性能的基本实践。

首先使用日志记录和高级调试工具和技术（包括反向调试和代码清理器）来识别和解决异步应用程序中的细微错误,例如：崩溃、死锁、竞争条件、内存泄漏和线程安全问题,然后使用 GoogleTest 框架针对异步代码量身定制测试策略。最后，将深入研究性能优化，理解缓存共享、错误共享等关键概念以及如何缓解性能瓶颈。掌握这些技术将提供一个全面的工具包，用于识别、诊断和提高异步应用程序的质量和性能。

本部分包含以下章节：

- 第 11 章，记录和调试异步软件
- 第 12 章，清理和测试异步软件
- 第 13 章，提高异步软件性能

第 11 章 记录和调试异步软件

无法确保软件产品没有错误，因此时不时就会出现错误。这时日志记录和调试就必不可少。

日志记录和调试对于识别和诊断软件系统中的问题至关重要。它们提供了对代码运行时行为的可见性，帮助开发人员跟踪错误、监控性能并了解执行流程。通过有效地使用日志记录和调试，开发人员可以检测错误、解决意外行为并提高整体系统稳定性和可维护性。

撰写本章时，假设读者们已经熟悉使用调试器调试 C++ 程序，并且了解一些基本的调试器命令和术语，例如断点、观察器、框架或堆栈跟踪。要复习这些知识，可以参考本章末尾的“扩展阅读”部分提供的参考资料。

本章中，将讨论以下主要主题：

- 如何使用日志记录来发现错误
- 如何调试异步软件

11.1. 技术要求

本章中，需要安装第三方库来编译示例。

需要安装 spdlog 和 {fmt} 库才能编译日志记录部分中的示例。请查看它们的文档(spdlog 的文档位于 https://github.com/gabime/spdlog，{fmt} 的文档位于 https://github.com/fmtlib/fmt)，并按照适合您平台的安装步骤进行操作。

一些示例需要支持 C++20 的编译器，请查看第 3 章中的技术要求部分，其中提供了有关如何安装 GCC 13 和 Clang 8 编译器的一些指导。

可以在以下 GitHub 库中找到所有完整的代码：

https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP

本章的示例位于 Chapter_11 文件夹下。所有源代码文件都可以使用 CMake 进行编译：

```
$ cmake . && cmake --build .
```

可执行二进制文件将在 bin 目录下生成。

11.2. 如何使用日志记录来发现错误

让我们从一个简单但有用的方法开始，以了解软件程序在执行时的作用——记录日志。

日志记录是保存程序中发生的事件日志的过程，通过使用消息记录程序的执行方式来存储信息，跟踪其流程，并帮助识别问题和错误。

大多数基于 Unix 的日志系统都使用标准协议 syslog，该协议由 Eric Altman 于 1980 年作为 Sendmail 项目的一部分创建。此标准协议定义了生成日志消息的软件、存储日志消息的系统，以及报告和分析这些日志事件的软件之间的界限。

每条日志消息都包含一个设施代码和一个严重性级别。设施代码标识了产生特定日志消息的系统类型（用户级、内核、系统、网络等），严重性级别描述了系统的状况，表明了处理特定问题的紧急程度，严重性级别包括紧急、警报、严重、错误、警告、通知、信息和调试。

大多数日志系统或记录器为日志消息提供了各种目的地或接收器：控制台、稍后可以打开和分析的文件、远程系统日志服务器或中继，以及其他目的地。

日志记录在没有调试器的地方很有用，正如我们稍后会看到的，特别是在分布式、多线程、实时、科学或以事件为中心的应用程序中，使用调试器检查数据或跟踪程序流程可能会成为一项繁琐的任务。

日志库通常还提供线程安全的单例类，允许多线程和异步写入日志文件，有助于日志轮换，通过动态创建新的日志文件来避免日志文件过大而不会丢失日志事件以及时间戳，以便更好地跟踪日志事件发生的时间。

与实现我们自己的多线程日志系统相比，更好的方法是使用一些经过充分测试和记录的可用于生产的库。

11.2.1. 如何选择第三方库

选择日志库（或任何其他库）时，需要在将其集成到我们的软件之前调查以下几点，以避免将来出现问题：

- 支持情况：库是否定期更新和升级？库是否有社区或活跃的生态系统可以帮助解决可能出现的任何问题？社区是否乐意使用库？
- 质量：是否有公开的错误报告系统？错误报告是否得到及时处理，提供解决方案并修复库中的错误？它是否支持最新的编译器版本并支持最新的 C++ 功能？
- 安全性：该库或其任何依赖库是否有任何报告的漏洞？
- 许可证：库许可证是否符合我们的开发和产品需求？费用是否可以承受？

对于复杂的系统，可能值得考虑使用集中式系统来收集和生成日志报告或仪表板，例如 Sentry（https://sentry.io）或 Logstash（https://www.elastic.co/logstash），它们可以收集、解析和转换日志，并且可以与其他工具集成，例如：Graylog（https://graylog.org）、Grafana（https://grafana.com）或 Kibana（https://www.elastic.co/kibana）。

下一节介绍一些有趣的日志库。

11.2.2. 一些相关的日志库

市场上有许多日志库，每个库都涵盖一些特定的软件要求。根据程序约束和需求，以下某个库可能比其他库更合适。

第 9 章中，探索了 Boost.Asio。Boost 还提供了另一个库 Boost.Log（https://github.com/boostorg/log），这是一个功能强大且可配置的日志库。

Google 还提供了许多开源库，其中包括 Google 日志库 glog（https://github.com/google/glog ），它是一个提供 C++ 风格的流 API 和帮助宏的 C++14 库。

如果开发人员熟悉 Java，那么一个很好的选择就是 Apache Log4cxx（https://logging.apache.org/log4cxx），基于 Log4j（https://logging.apache.org/log4j），这是一个多功能、工业级的 Java 日志框架。

其他值得考虑的日志库如下：

- spdlog（https://github.com/gabime/spdlog）是一个有趣的日志库，可以将其与 {fmt} 库一起使用。此外，程序可以从启动时开始记录消息并将其排队，甚至在指定日志输出文件名之前。
- uill（https://github.com/odygrd/quill）是一个异步低延迟 C++ 日志库。
- NanoLog（https://github.com/PlatformLab/NanoLog）是一个具有类似 printf API 的纳秒级日志系统。
- lwlog（https://github.com/ChristianPanov/lwlog）是一个非常快的异步 C++17 日志库。
- XTR（https://github.com/choll/xtr）是一个快速便捷的 C++ 日志库，适用于低延迟和实时环境。
- Reckless（https://github.com/mattiasflodin/reckless）是一个低延迟、高吞吐量的日志库。
- uberlog（https://github.com/IMQS/uberlog）是一个跨平台、多进程的 C++ 日志系统。
- Easylogging++（https://github.com/abumq/easyloggingpp）是一个单头 C++ 日志库，具有编写我们自己的接收器和跟踪性能的能力。
- tracetool（https://github.com/froglogic/tracetool）是一个日志记录和跟踪共享库。

作为指导原则，根据要开发的系统，我们可能会选择以下库之一：

- 对于低延迟或实时系统：Quill、XTR 或 Reckless
- 实现纳秒级高性能记录：NanoLog
- 对于异步日志记录：Quill 或 lwlog
- 对于跨平台、多进程应用程序：uberlog
- 对于简单灵活的日志记录：Easylogging++ 或 glog
- 熟悉 Java 日志记录：Log4cxx

所有库都有优点，但也有缺点，选择要包含在系统中的库之前需要进行权衡。下表总结了这些要点：

库	优点	缺点
spdlog	易于集成、关注性能、可定制	缺乏高级功能来满足低延迟需求
Quill	低延迟系统中的高性能	与简单的记录器相比，设置更复杂
NanoLog	速度一流，性能优化	特征有限的；适合于专门的用例
lwlog	轻量级，有利于快速集成	不成熟和功能丰富的替代品
XTR	非常高效，用户友好的界面	更适合于特定的实时应用程序
Reckless	高度优化的吞吐量和低延迟	与更通用的记录器相比，灵活性有限
uberlog	非常适合多进程和分布式系统	不如专门的低延迟记录器快
Easylogging++	易于使用，可定制的输出汇	性能优化不如其他一些库
tracetool	在一个库中结合日志记录和跟踪	不关注低延迟或高吞吐量
Boost.Log	多功能，与 Boost 库很好地集成	高复杂性，对于简单的日志需求不合适

glog	使用简单，适合需要简单 api 的项目	功能不够丰富，无法进行高级定制
Log4cxx	可靠的，具有工业强度的记录	更复杂的设置，适合较小的项目

表 11.1：各种库的优缺点

请访问日志库的网站以更好地了解它们提供的功能并比较它们之间的性能。

由于 spdlog 是 GitHub 上 fork 次数最多、收藏最多的 C++ 日志库存储库，因此在下一节中，将实现一个使用该库捕获竞争条件的示例。

11.2.3. 记录死锁——示例

实现这个示例之前，需要安装 spdlog 和 {fmt} 库。{fmt} (https://github.com/fmtlib/fmt) 是一个开源格式化库，为 C++ IOStreams 提供了一种快速、安全的替代方案。

请检查其文档并根据您的平台进行安装步骤。

让我们实现一个发生死锁的示例，当两个或多个线程需要获取多个互斥锁才能执行其工作时，就会发生死锁。如果互斥锁的获取顺序不同，则一个线程可以获取一个互斥锁并永远等待另一个线程获取另一个互斥锁。

此示例中，两个线程需要获取两个互斥锁 mtx1 和 mtx2，以增加 counter1 和 counter2 计数器的值并交换值。由于线程获取互斥锁的顺序不同，因此可能会发生死锁。

首先包含所需的头文件：

```
#include <fmt/core.h>
#include <spdlog/sinks/basic_file_sink.h>
#include <spdlog/sinks/stdout_color_sinks.h>
#include <spdlog/spdlog.h>

#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>

using namespace std::chrono_literals;
```

main() 函数中，定义计数器和互斥锁：

```
uint32_t counter1{};
std::mutex mtx1;

uint32_t counter2{};
std::mutex mtx2;
```

生成线程之前，设置一个多接收器记录器，该记录器可以同时将日志消息写入控制台和日志文件。还将其日志级别设置为调试，使记录器发布所有严重性级别大于调试的日志消息，以及由时间戳、线程标识符、日志级别和日志消息组成的每行日志的格式：

```
auto console_sink = std::make_shared<
            spdlog::sinks::stdout_color_sink_mt>();
console_sink->set_level(spdlog::level::debug);

auto file_sink = std::make_shared<
            spdlog::sinks::basic_file_sink_mt>("logging.log",
                                               true);
file_sink->set_level(spdlog::level::info);

spdlog::logger logger("multi_sink",
        {console_sink, file_sink});

logger.set_pattern(
            "%Y-%m-%d %H:%M:%S.%f - Thread %t [%l] : %v");
logger.set_level(spdlog::level::debug);
```

还声明了一个 increase_and_swap lambda 函数，可增加两个计数器的值并交换它们：

```
auto increase_and_swap = [&]() {
    logger.info("Incrementing both counters...");
    counter1++;
    counter2++;

    logger.info("Swapping counters...");
    std::swap(counter1, counter2);
};
```

两个 worker lambda 函数，worker1 和 worker2，在退出前获取两个互斥锁并调用 increase_and_swap()。由于使用了锁保护 (std::lock_guard) 对象，在析构期间离开 worker lambda 函数时会释放互斥锁：

```
auto worker1 = [&]() {
    logger.debug("Entering worker1");

    logger.info("Locking mtx1...");
    std::lock_guard<std::mutex> lock1(mtx1);
    logger.info("Mutex mtx1 locked");

    std::this_thread::sleep_for(100ms);

    logger.info("Locking mtx2...");
    std::lock_guard<std::mutex> lock2(mtx2);
    logger.info("Mutex mtx2 locked");

    increase_and_swap();

    logger.debug("Leaving worker1");
```

```
};

auto worker2 = [&]() {
    logger.debug("Entering worker2");

    logger.info("Locking mtx2...");
    std::lock_guard<std::mutex> lock2(mtx2);
    logger.info("Mutex mtx2 locked");

    std::this_thread::sleep_for(100ms);

    logger.info("Locking mtx1...");

    std::lock_guard<std::mutex> lock1(mtx1);
    logger.info("Mutex mtx1 locked");

    increase_and_swap();

    logger.debug("Leaving worker2");
};

logger.debug("Starting main function...");

std::thread t1(worker1);
std::thread t2(worker2);

t1.join();
t2.join();
```

这两个 worker lambda 函数类似，但略有不同：worker1 先获取 mutex1，然后获取 mutex2，而 worker2 则遵循相反的顺序，先获取 mutex2，然后获取 mutex1。在两个互斥锁获取之间有一个休眠期，以让另一个线程获取其互斥锁。因此，由于 worker1 将获取 mutex1，而 wor ker2 将获取 mutex2，因此会引发死锁。

睡眠之后，worker1 将尝试获取互斥锁 2，而 worker2 也将尝试获取互斥锁 1，但都不会成功，并会永远陷入死锁状态。

运行此代码时的输出如下：

```
2024-09-04 23:39:54.484005 - Thread 38984 [debug] : Starting main
function...
2024-09-04 23:39:54.484106 - Thread 38985 [debug] : Entering worker1
2024-09-04 23:39:54.484116 - Thread 38985 [info] : Locking mtx1...
2024-09-04 23:39:54.484136 - Thread 38986 [debug] : Entering worker2
2024-09-04 23:39:54.484151 - Thread 38986 [info] : Locking mtx2...
2024-09-04 23:39:54.484160 - Thread 38986 [info] : Mutex mtx2 locked
2024-09-04 23:39:54.484146 - Thread 38985 [info] : Mutex mtx1 locked
2024-09-04 23:39:54.584250 - Thread 38986 [info] : Locking mtx1...
```

```
2024-09-04 23:39:54.584255 - Thread 38985 [info] : Locking mtx2...
```

检查日志时要注意的第一个症状是程序从未完成，因此可能陷入死锁。

从记录器输出中，我们可以看到 t1（线程 38985）正在运行 worker1，而 t2（线程 38986）正在运行 worker2。t1 一进入 worker1，它就获取 mtx1。但是，一旦 worker2 启动，t2 就会获取 mtx2 互斥锁。然后，两个线程都等待 100 毫秒并尝试获取另一个互斥锁，但均未成功，程序仍处于阻塞状态。

日志记录在生产系统中必不可少，但如果滥用，则会造成一定的性能损失，而且大多数情况下需要人工干预来调查问题。作为日志详细程度和性能损失之间的折衷，人们可以选择实现不同的日志记录级别，并仅在正常运行期间记录主要事件，同时仍保留在需要时提供极其详细的日志（如果选择）的能力。在开发周期的早期检测代码错误的更自动化的方法是使用测试和代码消杀器，我们将在下一章中进行介绍。

并非所有错误都能检测到，因此通常使用调试器来追踪和修复软件中的错误。接下来让来介绍如何调试多线程和异步代码。

11.3. 如何调试异步软件

调试是查找和修复计算机程序中的错误的过程。

本节中，将探讨几种调试多线程和异步软件的技术。读者们必须具备一些关于如何使用调试器（例如：GDB（GNU 项目调试器）或 LLDB（LLVM 底层调试器））以及调试过程术语（例如：断点、观察器、回溯、框架和崩溃报告）的知识。

GDB 和 LLDB 都是出色的调试器，它们的大多数命令相同，只有少数命令不同。如果在 macOS 上调试程序或针对大型代码库，则可能首选 LLDB。另一方面，GDB 拥有悠久的历史，为许多开发人员所熟悉，并且支持更广泛的架构和平台。本节中，将使用 GDB 15.1，因为它是 GNU 框架的一部分，并且旨在与 g++ 编译器一起使用，但在调试使用 clang++ 编译的程序时，后面显示的大多数命令也可以与 LLDB 一起使用。

由于一些处理多线程和异步代码的调试器功能仍在开发中，请始终将调试器更新到最新版本以包含最新的功能和修复。

11.3.1. 一些有用的 GDB 命令

从一些在调试任何类型的程序时很有用的 GDB 命令开始，并为下一部分奠定基础。

调试程序时，可以启动调试器并将程序作为参数传递。可以使用 --args 选项传递程序可能需要的其他参数：

```
$ gdb <program> --args <args>
```

或者，可以使用进程标识符（PID）将调试器附加到正在运行的程序：

```
$ gdb -p <PID>
```

进入调试器后，可以运行程序（使用 run 命令）或启动程序（使用 start 命令）。运行意味着程序一直执行直到到达断点或完成。start 只是在 main() 函数的开头放置一个临时断点并运行程序，在程序开头停止执行。

例如，想调试一个已经崩溃的程序，可以使用崩溃生成的核心转储文件，该文件可能存储在系统中的特定位置（在 Linux 系统上通常是 /var/lib/apport/coredump/，但请访问官方文档以查看系统中的确切位置）。另请注意，通常情况下，核心转储默认是禁用的，需要在程序崩溃之前在同一个 shell 中运行 ulimit -c unlimited 命令。如果处理的程序非常大或系统磁盘空间不足，则可以将 unlimited 参数更改为某个文件大小限制。

生成 coredump 文件后，只需将其复制到程序二进制文件所在的位置，并使用以下命令：

```
$ gdb <program> <coredump>
```

注意，所有二进制文件都必须有调试符号，因此要使用-g 选项进行编译。在生产系统中，发布二进制文件通常会剥离符号并将其存储在单独的文件中。有 GDB 命令可以包含这些符号，还有命令行工具可以检查它们，但这个主题超出了本书的范围。

调试器启动后，可以使用 GDB 命令浏览代码或检查变量。一些有用的命令如下：

- info args：显示有关调用当前函数所用参数的信息。
- info locals：显示当前范围内的局部变量。
- whatis：显示给定变量或表达式的类型。
- return：从当前函数返回，不执行其余指令。可以指定返回值。
- backtrace：列出了当前调用堆栈中的所有堆栈帧。
- frame：这可以改变到特定的堆栈。
- up 和 down：在调用堆栈中移动，朝向当前函数的调用者（向上）或被调用者（向下）。
- print：计算并显示表达式的值，该表达式可以是变量名、类成员、指向内存区域的指针或直接是内存地址。还可以定义漂亮的输出来显示我们自己的类。

用最基本但也最常用的程序调试技术之一来结束本节，这种技术称为 printf。每个开发人员都使用过 printf 或其他命令来打印代码路径上的变量内容，以在战略代码位置显示其内容。在 GDB 中，dprintf 命令有助于设置 printf 样式的断点，当遇到这些断点时，这些断点会打印信息，但不停止程序执行。这样，可以在调试程序时使用 print 语句，而无需修改代码、重新编译和重新启动程序。

其语法如下：

```
$ dprintf <location>, <format>, <args>
```

例如，如果想在第 25 行设置一个 printf 语句来打印 x 变量的内容，但仅当其值大于 5 时，则命令如下：

```
$ dprintf 25, "x = %d\n", x if x > 5
```

现在已经有了一些基础知识，让我们开始调试多线程程序。

11.3.2. 调试多线程程序

这里显示的示例永远不会完成，因为两个互斥锁被不同的线程以不同的顺序锁定，从而导致死锁，正如本章前面介绍日志记录时所解释的那样：

```
#include <chrono>
#include <mutex>
#include <thread>

using namespace std::chrono_literals;

int main() {
    std::mutex mtx1, mtx2;

    std::thread t1([&]() {
        std::lock_guard lock1(mtx1);
        std::this_thread::sleep_for(100ms);
        std::lock_guard lock2(mtx2);
    });

    std::thread t2([&]() {
        std::lock_guard lock2(mtx2);
        std::this_thread::sleep_for(100ms);
        std::lock_guard lock1(mtx1);
    });

    t1.join();
    t2.join();

    return 0;
}
```

首先，让使用 g++ 编译此示例，并添加调试符号（-g 选项）并禁止代码优化（-O0 选项），以防止编译器重组二进制代码，并使调试器更难通过使用 --fno-omit-frame-pointer 选项来查找和显示相关信息。

以下命令编译 test.cpp 源文件并生成测试二进制文件。我们也可以使用相同选项的 clang++ ：

```
$ g++ -o test -g -O0 --fno-omit-frame-pointer test.cpp
```

如果运行生成的程序，将永远不会完成：

```
$ ./test
```

要调试正在运行的程序，首先使用 Unix 命令 ps 检索其 PID：

```
$ ps aux | grep test
```

然后，通过提供 pid 附加调试器并开始调试程序：

```
$ gdb -p <pid>
```

假设调试器以以下消息启动：

```
ptrace: Operation not permitted.
```

然后，只需运行以下命令：

```
$ sudo sysctl -w kernel.yama.ptrace_scope=0
```

GDB 正确启动，将能够在其提示符中输入命令。
可以执行的第一个命令是检查哪些线程正在运行的下一个命令：

```
(gdb) info threads
Id Target Id Frame
* 1 Thread 0x79d1f3883740 (LWP 14428) "test" 0x000079d1f3298d61 in
__futex_abstimed_wait_common64 (private=128, cancel=true, abstime=0x0,
op=265, expected=14429, futex_word=0x79d1f3000990)
at ./nptl/futex-internal.c:57
2 Thread 0x79d1f26006c0 (LWP 14430) "test" futex_wait (private=0,
expected=2, futex_word=0x7fff5e406b00) at ../sysdeps/nptl/futexinternal.h:146
3 Thread 0x79d1f30006c0 (LWP 14429) "test" futex_wait (private=0,
expected=2, futex_word=0x7fff5e406b30) at ../sysdeps/nptl/futexinternal.h:146
```

输出显示，GDB 标识符为 1 的 0x79d1f3883740 线程是当前线程。如果有很多线程，而我们只对特定子集感兴趣，比如线程 1 和 3，可以使用以下命令仅显示这些线程的信息：

```
(gdb) info thread 1 3
```

运行 GDB 命令将影响当前线程。例如，运行 bt 命令将显示线程 1 的回溯（输出已简化）：

```
(gdb) bt
#0 0x000079d1f3298d61 in __futex_abstimed_wait_common64
(private=128, cancel=true, abstime=0x0, op=265, expected=14429, futex_word=0x79d1f3000990) at
↪  ./nptl/futex-internal.c:57
#5 0x000061cbaf1174fd in main () at 11x18-debug_deadlock.cpp:22
```

要切换到另一个线程，例如线程 2，可以使用 thread 命令：

```
(gdb) thread 2
[Switching to thread 2 (Thread 0x79d1f26006c0 (LWP 14430))]
```

现在，bt 命令将显示线程 2 的回溯（输出简化）：

```
(gdb) bt
#0 futex_wait (private=0, expected=2, futex_word=0x7fff5e406b00) at
../sysdeps/nptl/futex-internal.h:146
#2 0x000079d1f32a00f1 in lll_mutex_lock_optimized
(mutex=0x7fff5e406b00) at ./nptl/pthread_mutex_lock.c:48
#7 0x000061cbaf1173fa in operator() (__closure=0x61cbafd64418) at 11x18-debug_deadlock.cpp:19
```

要在不同的线程中执行命令，只需使用线程应用命令。在本例中，在线程 1 和 3 上执行 bt 命令：

```
(gdb) thread apply 1 3 bt
```

要在所有线程中执行命令，只需使用 thread apply all <command>。

当在多线程程序中到达断点时，所有执行线程都会停止运行，从而允许检查程序的整体状态。当通过 continue、step 或 next 等命令重新启动执行时，所有线程都会恢复。当前线程将向前移动一个语句，但其他线程则不能保证，可能会向前移动几个语句，甚至在语句中间停止。

当执行停止时，调试器将跳转并显示当前线程中的执行上下文。为了避免调试器通过锁定调度程序在线程之间跳转，可以使用以下命令：

```
(gdb) set scheduler-locking <on/off>
```

还可以使用以下命令来检查调度程序锁定状态：

```
(gdb) show scheduler-locking
```

现在已经了解了一些用于多线程调试的新命令，来检查一下附加到调试器的应用程序都发生了什么。

如果检索线程 2 和 3 中的回溯，可以看到以下内容（输出简化，仅显示相关部分）：

```
(gdb) thread apply all bt
Thread 3 (Thread 0x79d1f30006c0 (LWP 14429) "test"):
#0 futex_wait (private=0, expected=2, futex_word=0x7fff5e406b30) at
../sysdeps/nptl/futex-internal.h:146
#5 0x000061cbaf117e20 in std::mutex::lock (this=0x7fff5e406b30) at /
usr/include/c++/14/bits/std_mutex.h:113
#7 0x000061cbaf117334 in operator() (__closure=0x61cbafd642b8) at
11x18-debug_deadlock.cpp:13
Thread 2 (Thread 0x79d1f26006c0 (LWP 14430) "test"):
#0 futex_wait (private=0, expected=2, futex_word=0x7fff5e406b00) at
../sysdeps/nptl/futex-internal.h:146
#5 0x000061cbaf117e20 in std::mutex::lock (this=0x7fff5e406b00) at /
usr/include/c++/14/bits/std_mutex.h:113
#7 0x000061cbaf1173fa in operator() (__closure=0x61cbafd64418) at
11x18-debug_deadlock.cpp:19
```

运行 std::mutex::lock() 之后，两个线程都在第 13 行等待线程 3，在第 19 行等

待线程 2，这分别与 std::thread t1 中的 std::lock_guard lock2 和 std::thread t2 中的 std::lock_guard l ock1 匹配。

因此，检测到这些代码位置的线程中发生了死锁。

现在，通过捕获条件竞争来进一步了解如何调试多线程软件。

11.3.3. 调试条件竞争

条件竞争是最难检测和调试的错误之一，通常偶尔发生且每次的影响都不同，并且有时在程序达到失败点之前会发生一些昂贵的计算。

这种不稳定的行为不仅仅是由条件竞争引起的。与不正确的内存分配相关的其他问题也会导致类似的症状，除非进行调查并得出根本原因诊断，否则不可能将错误归类为条件竞争。

调试条件竞争的一种方法是使用观察点手动检查变量是否在当前线程中执行的任何语句都没有修改它的情况下更改了它的值，或者将断点放置在由特定线程触发的战略位置：

```
(gdb) break <linespec> thread <id> if <condition>
```

例如，请参阅以下内容：

```
(gdb) break test.cpp:11 thread 2
```

或者，甚至使用断言并检查不同线程访问的任何变量的当前值是否具有预期值。下一个示例遵循此方法：

```
#include <cassert>
#include <chrono>
#include <cmath>
#include <iostream>
#include <mutex>
#include <thread>

using namespace std::chrono_literals;

static int g_value = 0;
static std::mutex g_mutex;

void func1() {
    const std::lock_guard<std::mutex> lock(g_mutex);
    for (int i = 0; i < 10; ++i) {
        int old_value = g_value;
        int incr = (rand() % 10);
        g_value += incr;
        assert(g_value == old_value + incr);
        std::this_thread::sleep_for(10ms);
    }
}
```

```
23
24  void func2() {
25      for (int i = 0; i < 10; ++i) {
26          int old_value = g_value;
27          int incr = (rand() % 10);
28          g_value += (rand() % 10);
29          assert(g_value == old_value + incr);
30          std::this_thread::sleep_for(10ms);
31      }
32  }
33
34  int main() {
35      std::thread t1(func1);
36      std::thread t2(func2);
37
38      t1.join();
39      t2.join();
40
41      return 0;
42  }
```

两个线程 t1 和 t2 正在运行函数，这些函数将 g_value 全局变量增加一个随机值。每次增加时，都会将 g_value 与预期值进行比较，如果不相等，则 assert 指令将停止程序。

编译该程序并运行调试器：

```
$ g++ -o test -g -O0 test
$ gdb ./test
```

调试器启动后，使用 run 命令运行程序。程序将运行，并在某个时刻通过接收 SIGABRT 信号中止，表明断言未得到满足。

```
test: test.cpp:29: void func2(): Assertion `g_value == old_value + incr' failed.
Thread 3 "test" received signal SIGABRT, Aborted.
```

当程序停止后，可以使用 backtrace 命令检查该点的回溯，并将该故障点的源代码更改为特定的框架或列表。

这个例子非常简单，所以通过检查断言输出可以清楚地发现 g_value 变量出了问题，而且这很可能是条件竞争。

对于更复杂的程序来说，手动调试问题的过程非常艰巨，因此关注另一种可以帮助解决这个问题的技术，称为反向调试。

11.3.4. 反向调试

反向调试，也称为“时间旅行调试”，允许调试器在程序失败后停止程序并返回程序执行的历史记录以调查失败的原因。此功能通过记录每条机器指令的执行，来实现调试程序的运行情况以及

内存和寄存器值的每次变化，然后使用这些记录随意重放和后退程序。

Linux 上，可以使用 GDB（自 7.0 版起）、rr（最初由 Mozilla 开发，https://rr-project.org）或 Undo 的时间旅行调试器（UDB）（https://docs.undo.io）。在

Windows 上，可以使用时间旅行调试（https://learn.microsoft.com/en-us/windows-hardware/drivers/debuggercmds/time-travel-debugging-overview）。

反向调试仅受有限数量的 GDB 目标支持，例如：远程目标 Simics、系统集成和设计（SID）模拟器或本机 Linux 的进程记录和重放目标（仅适用于 i386、amd64、moxie-elf 和 arm）。撰写本书时，Clang 的反向调试功能仍在开发中。

因此，由于这些限制，我们决定使用 rr 做一个小展示。请按照项目网站上的说明构建和安装 rr 调试工具：https://github.com/rr-debugger/rr/wiki/Building-And-Installing。

安装后，要录制和回放程序，请使用以下命令：

```
$ rr record <program> --args <args>
$ rr replay
```

例如，有一个名为 test 的程序，命令序列将如下所示：

```
$ rr record test
rr: Saving execution to trace directory `/home/user/.local/share/rr/test-1'.
```

假设显示的是以下致命错误：

```
[FATAL src/PerfCounters.cc:349:start_counter()] rr needs /proc/sys/kernel/perf_event_paranoid
↪  <= 3, but it is 4. Change it to <= 3.
Consider putting 'kernel.perf_event_paranoid = 3' in /etc/sysctl.d/10-rr.conf.
```

然后，使用以下命令调整内核变量 kernel.perf_event_paranoid：

```
$ sudo sysctl kernel.perf_event_paranoid=1
```

当有记录可用，请使用重放命令开始调试程序：

```
$ rr replay
```

或者，如果程序崩溃而您只想在录制结束时开始调试，请使用-e 选项：

```
$ rr replay -e
```

此时，rr 将使用 GDB 调试器启动程序并加载其调试符号。然后，可以使用以下命令进行逆向调试：

- reverse-continue：开始反向执行程序。当到达断点或由于同步异常时，执行将停止。
- reverse-next：向后运行到当前堆栈帧中执行的上一行的开头。
- reverse-nexti：这将反向执行单个指令，跳过那些移动到内部堆栈帧的指令。
- reverse-step：向后运行程序，直到控制到达新源行的开头。

- reverse-stepi：反向执行一条机器指令。
- reverse-finish：执行到当前函数调用，也就是当前函数的开始。

还可以反转调试的方向，并在相反方向上使用常规的正向调试命令（例如：next，step，continue 等）：

```
(rr) set exec-direction reverse
```

要将执行方向重新设置为正向：

```
(rr) set exec-direction forward
```

作为练习，可安装 rr 调试器，并尝试使用反向调试来调试前面的示例。

现在，继续讨论如何调试协程，由于其异步特性，这是一项具有挑战性的任务。

11.3.5. 调试协程

异步代码可以像同步代码一样进行调试，方法是在具有特定条件的战略位置使用断点，使用观察点检查变量，并进入或跳过代码。此外，使用前面描述的技术选择特定线程，并锁定调度程序有助于避免调试时不必要的干扰。

异步代码很复杂，例如：执行异步代码时将使用哪个线程，这使其调试起来更加困难。对于 C++ 协程，由于其暂停/恢复特性，调试甚至更难掌握。

Clang 使用协程编译程序分为两步：Clang 进行语义分析，LLVM 中端构建并优化协程框架。由于调试信息是在 Clang 前端生成的，所以在编译过程的后期生成协程框架时，调试信息会不足，GCC 遵循类似的方法。

此外，如果执行在协程内部中断，当前帧将只有一个变量 frame_ptr。在协程中，没有指针或函数参数。协程在暂停之前将其状态存储在堆中，并且在执行期间仅使用堆栈。frame_ptr 用于访问协程正常运行所需的所有必要信息。

我们来调试一下第 9 章中实现的 Boost.Asio 协程示例，这里只展示相关指令，完整源代码请访问第 9 章协程部分：

```
1  boost::asio::awaitable<void> echo(tcp::socket socket) {
2      char data[1024];
3
4      while (true) {
5          std::cout << "Reading data from socket...\n";//L12
6          std::size_t bytes_read = co_await
7              socket.async_read_some(
8                  boost::asio::buffer(data),
9                              boost::asio::use_awaitable);
10         /* .... */
11         co_await boost::asio::async_write(socket,
12                 boost::asio::buffer(data, bytes_read),
13                 boost::asio::use_awaitable);
```

```
    }
}

boost::asio::awaitable<void>
listener(boost::asio::io_context& io_context,
         unsigned short port) {
    tcp::acceptor acceptor(io_context,
                           tcp::endpoint(tcp::v4(), port));
    while (true) {
        std::cout << "Accepting connections...\n"; // L45
        tcp::socket socket = co_await
            acceptor.async_accept(
                boost::asio::use_awaitable);

        boost::asio::co_spawn(io_context,
            echo(std::move(socket)),
            boost::asio::detached);
    }
}
/* main function */
```

由于使用 Boost，因此可包含 Boost.System 库，以在编译源代码时添加更多符号以供调试：

```
$ g++ --std=c++20 -ggdb -O0 --fno-omit-frame-pointer -lboost_system test.cpp -o test
```

然后，使用生成的程序启动调试器，并在第 12 行和第 45 行设置断点，它们是每个协程的 while 循环内第一条指令的位置：

```
$ gdb -q ./test
(gdb) b 12
(gdb) b 45
```

我们还启用 GDB 内置优雅输出来显示标准模板库容器的可读输出：

```
(gdb) set print pretty on
```

如果现在我们运行该程序（run 命令），将在接受连接之前到达协程侦听器内部第 42 行的断点。使用 info locals 命令，可以检查局部变量。

协程会创建一个具有多个内部字段的状态机，例如：promise 对象、其他线程、调用者对象的地址、待处理的异常等。此外，还存储恢复和销毁回调。这些结构依赖于编译器，与编译器的实现相关联，如果使用 Clang，则可以通过 frame_ptr 访问。

如果继续运行程序（使用 continue 命令），服务器将等待客户端连接。要退出等待状态，使用 telnet（如第 9 章所示）将客户端连接到服务器。此时，执行将停止，因为到达了 echo() 协程内第 12 行的断点，并且 info locals 显示每个 echo 连接使用的变量。

使用 backtrace 命令将显示一个调用堆栈，该堆栈可能由于协程的暂停特性而具有一些复杂性。

第 8 章描述的纯 C++ 例程中，有两个表达式设置断点可能会很有趣：

- co_await：执行暂停，直到等待的操作完成。可以通过检查底层 await_suspend、await_resume 或自定义 awaitable 代码在协程恢复的位置设置断点。
- co_yield：暂停执行并产生一个值。调试期间，进入 co_yield 以观察协程及其调用函数之间的控制流。

由于协程在 C++ 世界中相当新，并且编译器也在不断发展，我们希望协程的调试很快就能变得更加简单。

当发现并调试了一些错误，并且可以重现导致这些特定错误的场景，设计一些涵盖这些情况的测试将很方便，以避免更改代码而导致类似的问题或事件。

在下一章中学习如何测试多线程和异步代码。

11.4. 总结

本章中，介绍如何使用日志记录和调试异步程序。

首先使用日志来发现正在运行的软件中的问题，并展示了使用 spdlog 日志库检测死锁的实用性。还讨论了许多其他库，并描述了可能适合特定场景的相关功能。

但是，并非所有错误都可以通过使用日志来发现，有些错误可能只能在软件开发生命周期的后期才被发现，当生产中出现一些问题时，甚至在处理程序崩溃和事故时也是如此。调试器是检查正在运行或崩溃的程序、了解其代码路径和查找错误的有用工具。引入了几个示例和调试器命令来处理通用代码，但也特别适用于多线程和异步软件、条件竞争和协程。此外，还引入了 rr 调试器，展示了将反向调试纳入开发人员工具箱的潜力。

下一章中，将介绍使用消杀器和测试技术，来提高异步程序的运行时间和资源使用率的性能和优化技术。

11.5. 扩展阅读

- Logging: https://en.wikipedia.org/wiki/Logging_(computing)
- Syslog: https://en.wikipedia.org/wiki/Syslog
- Google Logging Library: https://github.com/google/glog
- Apache Log4cxx: https://logging.apache.org/log4cxx
- spdlog: https://github.com/gabime/spdlog
- Quill: https://github.com/odygrd/quill
- xtr: https://github.com/choll/xtr
- lwlog: https://github.com/ChristianPanov/lwlog
- uberlog: https://github.com/IMQS/uberlog
- Easylogging++: https://github.com/abumq/easyloggingpp
- NanoLog: https://github.com/PlatformLab/NanoLogReckless

- Logging Library: https://github.com/mattiasflodin/reckless
- tracetool: https://github.com/froglogic/tracetool
- Logback Project: https://logback.qos.ch
- Sentry: https://sentry.io
- Graylog: https://graylog.org
- Logstash: https://www.elastic.co/logstash
- Debugging with GDB: https://sourceware.org/gdb/current/onlinedocs/gdb.html
- LLDB tutorial: https://lldb.llvm.org/use/tutorial.html
- Clang Compiler User's Manual: https://clang.llvm.org/docs/UsersManual.html
- GDB: Running programs backward: https://www.zeuthen.desy.de/dv/documentation/unixguide/infohtml/gdb/Reverse-Execution.html#Reverse-Execution
- Reverse Debugging with GDB: https://sourceware.org/gdb/wiki/ReverseDebug
- Debugging C++ Coroutines: https://clang.llvm.org/docs/DebuggingCoroutines.html
- SID Simulator User's Guide: https://sourceware.org/sid/sid-guide/book1.html
- Intel Simics Simulator for Intel FPGAs: User Guide: https://www.intel.com/content/www/us/en/docs/programmable/784383/24-1/about-this-document.html
- IBM Support: How do I enable core dumps: https://www.ibm.com/support/pages/how-do-i-enable-core-dumps
- Core Dumps –How to enable them?: https://medium.com/@sourabhedake/coredumps-how-to-enable-them-73856a437711

第 12 章 消杀和测试异步软件

测试是评估和验证软件解决方案是否按预期完成任务的过程，验证其质量并确保满足用户要求。通过适当的测试，可以避免出现错误并提高性能。

本章中，将探讨几种测试异步软件的技术，主要使用 GoogleTest 库以及 GNU 编译器集合（GCC）和 Clang 编译器提供的清理器。需要一些单元测试方面的知识。本章末尾的“扩展阅读”部分中，可以找到一些参考资料，它们可能有助于更新和扩展在这些领域的知识。

本章中，将讨论以下主要主题：

- 消杀代码以分析软件并发现潜在问题
- 测试异步代码

12.1. 技术要求

对于本章，需要安装 GoogleTest（https://google.github.io/googletest）来编译一些示例。

一些示例需要支持 C++20 的编译器，请查看第 3 章中的技术要求部分，包含有关如何安装 GCC 13 和 Clang 8 编译器的一些指导。

可以在以下 GitHub 库中找到所有完整的代码：https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP

本章的示例位于 Chapter_12 文件夹下。所有源代码文件都可以使用 CMake 进行编译：

```
$ cmake . && cmake --build .
```

可执行二进制文件将在 bin 目录下生成。

12.2. 消杀代码以分析软件并查找潜在问题

Sanitizer 是 Google 最初开发的用于检测和预防代码中各类问题或安全漏洞的工具，帮助开发人员在开发过程早期发现错误，减少问题后期修复的成本，提高软件的稳定性和安全性。

Sanitizer 通常集成到开发环境中，并且通常在手动测试期间或运行单元测试、持续集成（CI）管道或代码审查时启用。

C++ 编译器（例如：GCC 和 Clang）具有编译器选项，可在构建程序时生成代码，以跟踪运行时的执行情况并报告错误和漏洞。在 Clang 3.1 版和 GCC 4.8 版中实现。

由于额外的指令会注入到程序的二进制代码中，因此根据清理器的类型，性能会降低大约 1.5 倍到 4 倍。此外，总内存开销为 2 倍到 4 倍，堆栈大小最多增加 3 倍。与使用其他检测框架或动态分析工具（例如 Valgrind（https://valgrind.org））时遇到的减速相比，减速要低得多，后者造成的减速要比生产二进制文件慢 50 倍。另一方面，使用 Valgrind 的好处是不需要重新编译。这两种方法都只在程序运行时检测问题，并且只在执行遍历的代码路径上检测问题。因此，需要确保足够的覆盖范围。还有静态分析工具和 linter，可用于检测编译过程中的问题并检

查程序中包含的所有代码。例如，GCC 和 Clang 等编译器可以通过启用 -Werror、-Wall 和 -pedantic 选项来执行检查并提供有用的信息。

还有一些开源替代方案，例如：Cppcheck 或 Flawfinder，或者对开源项目免费的商业解决方案，例如 PVS-Studio 或 Coverity Scan。其他解决方案，例如：SonarQube、CodeSonar 或 OCLint，可用于持续集成/持续交付（CI/CD）管道，以进行持续质量跟踪。

本节中，将重点介绍消杀器，可以通过向编译器传递一些特殊选项来启用。

12.2.1. 编译器选项

为了启用清理器，需要在编译程序时传递一些编译器选项。

主要选项是 --fsanitize=sanitizer_name，其中 sanitizer_name 是以下选项之一：

- address：这是 AddressSanitizer（ASan），用于检测内存错误，例如缓冲区溢出和释放后使用错误
- 线程：针对 ThreadSanitizer（TSan）的，通过监视线程交互来识别多线程程序中的数据争用和其他线程同步问题
- 泄漏：这是 LeakSanitizer（LSan）的功能，通过跟踪内存分配并确保正确释放所有分配的内存来发现内存泄漏
- 内存：这是 MemorySanitizer（MSan），用于发现未初始化内存的使用情况
- undefined：这是 UndefinedBehaviorSanitizer（UBSan），用于检测未定义的行为，例如：整数溢出、无效类型转换和其他错误操作

Clang 还包括数据流、cfi（控制流完整性）、safe_stack 和 realtime。

GCC 添加了 kernel-address、hwaddress、kernel-hwaddress、pointer-compare、pointer-subtract 和 shadow-call-stack。

由于此列表和标志行为会随着时间而改变，建议检查编译器的官方文档。

可能需要其他编译选项：

- -fno-omit-frame-pointer：帧指针是编译器用来跟踪当前堆栈帧的寄存器，其中包含当前函数的基址等信息。省略帧指针可能会提高程序的性能，但代价是调试难度大大增加；使定位局部变量和重建堆栈跟踪变得更加困难。
- -g：在警告消息中包含调试信息并显示文件名和行号。如果使用调试器 GDB，则可能需要使用 -ggdb 选项，因为编译器可以生成更具表现力的符号以供调试时使用。此外，可以使用-g[level] 指定级别，其中 [level] 是 0 到 3 之间的值，每增加一个级别就会添加更多调试信息。默认级别为 2。
- -fsanitize-recover：这些选项使清理程序尝试继续运行程序，就好像没有检测到错误一样。
- -fno-sanitize-recover：清理程序将仅检测第一个错误，并且程序将以非零退出代码退出。

为了保持合理的性能，可能需要通过指定 -O[num] 选项来调整优化级别。不同的消杀程序在达到一定优化级别之前，效果最佳。最好从 -O0 开始，如果速度明显下降，则尝试增加到 -O1、

-O2 等。此外，由于不同的消杀程序和编译器会推荐特定的优化级别，请查看其文档。

使用 Clang 时，为了使堆栈跟踪易于理解并让清理器将地址转换为源代码位置，除了使用前面提到的标志外，还可以将特定的环境变量 [X]SAN_SYMBOLIZER_PATH 设置为 llvm-symbolizer 的位置（其中 [X] 为 AddressSatinizer 的 A、LSan 的 L、MSan 的 M，依此类推）。还可以将此位置包含在 PATH 环境变量中，以下是使用 AddressSatinizer 时设置 PATH 变量的示例：

```
export ASAN_SYMBOLIZER_PATH=`which llvm-symbolizer`
export PATH=$ASAN_SYMBOLIZER_PATH:$PATH
```

注意，使用某些清理程序启用 -Werror 可能会导致误报。此外，可能需要其他编译器标志，但执行期间的警告消息将显示问题正在发生，并且显然需要标志。查看消杀程序和编译器的文档，以了解在这些情况下应使用哪个标志。

避免消杀部分代码

有时，可能希望消除某些清理程序警告并跳过清理某些函数：这是一个众所周知的问题、该函数是正确的、这是一个误报、该函数需要加速，或者这是第三方库中的问题。这些情况下，可以使用抑制文件或使用一些宏指令排除代码区域。还有一个黑名单机制，但由于已弃用，取而代之的是抑制文件；我们不会在这里进行评论。

使用抑制文件，只需创建一个文本文件，列出不希望消杀程序运行的代码区域。每一行都由遵循特定格式的模式组成，具体取决于消杀程序，通常结构如下所示：

```
type:location_pattern
```

此处，type 表示抑制的类型，例如：泄漏和竞争值，location_pattern 是与要抑制的函数或库名称匹配的正则表达式。以下是 ASan 抑制文件的示例，下一节将对此进行解释：

```
# Suppress known memory leaks in third-party function Func1 in library
Lib1
leak:Lib1::Func1
# Ignore false-positive from function Func2 in library Lib2
race:Lib2::Func2
# Suppress issue from libc
leak:/usr/lib/libc.so.*
```

将此文件命名为 myasan.supp，编译此抑制文件并通过 [X]SAN_OPTIONS 将其传递给清理程序：

```
$ clang++ -O0 -g -fsanitize=address -fno-omit-frame-pointer test.cpp -o test
$ ASAN_OPTIONS=suppressions=myasan.supp ./test
```

还可以在源代码中使用宏来排除要清理的特定函数，方法是使用 __attribute__((no_sanitize("<sanitizer_name>"))):

```
#if defined(__clang__) || defined (__GNUC__)
# define ATTRIBUTE_NO_SANITIZE_ADDRESS __attribute__((no_sanitize_
address))
#else
# define ATTRIBUTE_NO_SANITIZE_ADDRESS
#endif
...
ATTRIBUTE_NO_SANITIZE_ADDRESS
void ThisFunctionWillNotBeInstrumented() {...}
```

该技术提供了对清理器应检测的内容的细粒度编译时控制。

现在来探索最常见的代码消杀器类型，从最相关的检查地址误用类型开始。

12.2.2. 地址消杀器

ASan 的目的是检测由于数组越界访问期间的缓冲区溢出（堆、堆栈和全局）、使用通过释放或删除操作释放的内存块，以及其他内存泄漏而发生的内存相关错误。

除了设置 `-fsanitize=address` 和之前推荐的其他标志之外，还可以使用`-fsanitize-address-use-after-scope` 来检测超出范围后使用的内存，或者设置`ASAN_OPTIONS=option detect_stack_use_after_return=1` 环境变量来检测返回后的使用情况。

ASAN_OPTIONS 还可用于指示 ASan 打印堆栈跟踪或设置日志文件：

```
ASAN_OPTIONS=detect_stack_use_after_return=1,print_stacktrace=1,log_path=asan.log
```

Linux 上的 Clang 完全支持 ASan，其次是 Linux 上的 GCC。默认情况下，ASan 禁用，会增加额外的运行时开销。

此外，ASan 还会处理对 glibc 的所有调用 -glibc 是 GNU C 库，为 GNU 系统提供核心库。但其他库并非如此，建议使用`-fsanitize=address` 选项重新编译此类库，而使用 Valgrind 不需要重新编译。

ASan 可以与 UBSan 结合使用，但它会使性能降低约 50%。

如果想要更积极的诊断消杀，可以使用以下标志组合：

```
ASAN_OPTIONS=strict_string_checks=1:detect_stack_use_after_return=1:
↪  check_initialization_order=1:strict_init_order=1
```

来看两个使用 ASan 检测常见软件问题的例子，分别是内存释放后使用和检测缓冲区溢出。

释放后的内存使用情况

软件中一个常见的问题是释放内存后仍使用内存。此示例中，堆中分配的内存在删除后仍在使用：

```
#include <iostream>
#include <memory>
```

```
3
4  int main() {
5      auto arr = new int[100];
6      delete[] arr;
7      std::cout << "arr[0] = " << arr[0] << '\n';
8      return 0;
9  }
```

假设上述源代码位于名为 `test.cpp` 的文件中。要启用 ASan，只需使用以下命令编译该文件：

```
$ clang++ -fsanitize=address -fno-omit-frame-pointer -g -O0 -o test test.cpp
```

然后，执行结果输出测试程序，得到以下输出（注意，输出是简化的，只显示相关内容，并且可能因不同的编译器版本和执行上下文而有所不同）：

```
ERROR: AddressSanitizer: heap-use-after-free on address 0x514000000040 at pc 0x63acc82a0bec bp
↪  0x7fff2d096c60 sp 0x7fff2d096c58 READ of size 4 at 0x514000000040 thread T0
    #0 0x63acc82a0beb in main test.cpp:7:31
0x514000000040 is located 0 bytes inside of 400-byte region [0x514000000040,0x5140000001d0)
freed by thread T0 here:
    #0 0x63acc829f161 in operator delete[](void*)
    ↪  (/mnt/StorePCIE/Projects/Books/Packt/Book/Code/build/bin/Chapter_11/
    ↪  11x02-ASAN_heap_use_after_free+0x106161)
    ↪  (BuildId:7bf8fe6b1f86a8b587fbee39ae3a5ced3e866931)
previously allocated by thread T0 here:
    #0 0x63acc829e901 in operator new[](unsigned long)
    ↪  (/mnt/StorePCIE/Projects/Books/Packt/Book/Code/build/bin/Chapter_11/
    ↪  11x02-ASAN_heap_use_after_free+0x105901)
    ↪  (BuildId:7bf8fe6b1f86a8b587fbee39ae3a5ced3e866931)
SUMMARY: AddressSanitizer: heap-use-after-free test.cpp:7:31 in main
```

输出显示已应用 ASan 并检测到堆释放后使用错误。此错误发生在 T0 线程（主线程）中。输出还指向分配并随后释放该内存区域的代码及其大小（400 字节区域）。

这些错误不仅发生在堆内存中，还发生在堆栈或全局区域中分配的内存区域。ASan 可用于检测这些类型的问题，例如：内存溢出。

内存溢出

内存溢出（也称为缓冲区溢出或超限）发生在，当某些数据写入超出缓冲区分配内存的内存地址时。

以下示例显示了堆内存溢出：

```
1  #include <iostream>
2
3  int main() {
4      auto arr = new int[100];
```

```
    arr[0] = 0;
    int res = arr[100];
    std::cout << "res = " << res << '\n';
    delete[] arr;
    return 0;
}
```

编译并运行生成的程序后，输出如下：

```
ERROR: AddressSanitizer: heap-buffer-overflow on address
0x5140000001d0 at pc 0x582953d2ac07 bp 0x7ffde9d58910 sp
0x7ffde9d58908
READ of size 4 at 0x5140000001d0 thread T0
    #0 0x582953d2ac06 in main test.cpp:6:13
0x5140000001d0 is located 0 bytes after 400-byte region
[0x514000000040,0x5140000001d0)
allocated by thread T0 here:
    #0 0x582953d28901 in operator new[](unsigned long) (test+0x105901)
(BuildId: 82a16fc86e01bc81f6392d4cbcad0fe8f78422c0)
    #1 0x582953d2ab78 in main test.cpp:4:14
(test+0x2c374) (BuildId: 82a16fc86e01bc81f6392d4cbcad0fe8f78422c0)
SUMMARY: AddressSanitizer: heap-buffer-overflow test.cpp:6:13 in main
```

从输出中可以看到，现在 ASan 在访问超出 400 字节区域（arr 变量）的内存地址时，主线程（T0）中会报告堆缓冲区溢出错误。

集成到 ASan 中的清理器是 LSan。

接下来，了解如何使用此消杀器检测内存泄漏。

12.2.3. LeakSanitizer

LSan 用于检测在分配内存但未正确释放时发生的内存泄漏。

LSan 集成到 ASan 中，在 Linux 系统上默认启用。在 macOS 上可以使用 ASAN_OPTIONS=detect_leaks=1 启用它。要禁用它，只需设置 detect_leaks=0。

如果使用 -fsanitize=leak 选项，程序将链接到支持 LSan 的 ASan 子集，从而禁用编译时检测并减少 ASan 减速。请注意，此模式的测试不如默认模式那么充分。

来看一个内存泄漏的例子：

```
#include <string.h>
#include <iostream>
#include <memory>

int main() {
    auto arr = new char[100];
    strcpy(arr, "Hello world!");
    std::cout << "String = " << arr << '\n';
```

```
    return 0;
}
```

此示例中，分配了 100 个字节（arr 变量）但从未释放。

要启用 LSan，只需使用以下命令编译文件：

```
$ clang++ -fsanitize=leak -fno-omit-frame-pointer -g -O2 -o test test.cpp
```

运行生成的测试二进制文件，得到以下结果：

```
ERROR: LeakSanitizer: detected memory leaks
Direct leak of 100 byte(s) in 1 object(s) allocated from:
    #0 0x5560ba9a017c in operator new[](unsigned long) (test+0x3417c)
(BuildId: 2cc47a28bb898b4305d90c048c66fdeec440b621)
    #1 0x5560ba9a2564 in main test.cpp:6:16
SUMMARY: LeakSanitizer: 100 byte(s) leaked in 1 allocation(s).
```

LSan 正确报告，使用运算符 new 分配了 100 字节的内存区域，但从未删除。

由于本书探讨了多线程和异步编程，现在来介绍一种用于检测数据竞争和其他线程问题的消杀器：TSan。

12.2.4. ThreadSanitizer

TSan 用于检测线程问题，尤其是数据争用和同步问题。它不能与 ASan 或 LSan 结合使用。TSan 是最符合本书内容的消杀工具。

通过指定 -fsanitize=thread 编译器选项可以启用此清理程序，并且可以使用 TSAN_OPTIONS 环境变量修改其行为。例如，想在第一个错误后停止，只需使用以下命令：

```
TSAN_OPTIONS=halt_on_error=1
```

此外，为了获得合理的性能，请使用编译器的 -O2 选项。

TSan 仅报告运行时发生的竞争条件，因此不会对未在运行时执行的代码路径中存在的竞争条件发出警报，需要设计提供良好覆盖率并使用实际工作负载的测试。

来看一些 TSan 检测数据竞争的示例。下一个示例中，将使用全局变量（不使用互斥锁保护其访问）来执行此操作：

```
#include <thread>

int globalVar{0};

void increase() {
    globalVar++;
}

```

```
void decrease() {
    globalVar--;
}

int main() {
    std::thread t1(increase);
    std::thread t2(decrease);

    t1.join();
    t2.join();

    return 0;
}
```

编译程序后，使用以下命令启用 TSan：

```
$ clang++ -fsanitize=thread -fno-omit-frame-pointer -g -O2 -o test test.cpp
```

运行结果程序将生成以下输出：

```
WARNING: ThreadSanitizer: data race (pid=31692)
    Write of size 4 at 0x5932b0585ae8 by thread T2:
        #0 decrease() test.cpp:10:12 (test+0xe0b32) (BuildId:
895b75ef540c7b44daa517a874d99d06bd27c8f7)
    Previous write of size 4 at 0x5932b0585ae8 by thread T1:
        #0 increase() test.cpp:6:12 (test+0xe0af2) (BuildId:
895b75ef540c7b44daa517a874d99d06bd27c8f7)
    Thread T2 (tid=31695, running) created by main thread at:
        #0 pthread_create <null> (test+0x6062f) (BuildId:
895b75ef540c7b44daa517a874d99d06bd27c8f7)
    Thread T1 (tid=31694, finished) created by main thread at:
        #0 pthread_create <null> (test+0x6062f) (BuildId:
895b75ef540c7b44daa517a874d99d06bd27c8f7)
SUMMARY: ThreadSanitizer: data race test.cpp:10:12 in decrease()
ThreadSanitizer: reported 1 warnings
```

从输出中可以清楚地看出，在 increase() 中访问 globalVar 时存在数据竞争和 duce() 函数。

如果决定使用 GCC 而不是 Clang，则运行生成的程序时可能会报告以下错误：

```
FATAL: ThreadSanitizer: unexpected memory mapping 0x603709d10000 - 0x603709d11000
```

此内存映射问题是由称为地址空间布局随机化（ASLR）的安全功能引起的，这是操作系统使用的一种内存保护技术，通过随机化进程的地址空间来防止缓冲区溢出攻击。

一个解决方案是使用以下命令来减少 ASLR：

```
$ sudo sysctl vm.mmap_rnd_bits=30
```

如果错误仍然存在，可以进一步减少传递给 `vm.mmap_rnd_bits` 的值（上述命令中为 30）。要检查该值是否设置正确，只需运行以下命令：

```
$ sudo sysctl vm.mmap_rnd_bits
vm.mmap_rnd_bits = 30
```

注意，此更改不是永久性的。当机器重新启动时，其值将设置为默认值。要保留此更改，请将 `m.mmap_rnd_bits=30` 添加到 `/etc/sysctl.conf`。

但这会降低系统的安全性，最好使用以下命令暂时禁用特定程序的 ASLR：

```
$ setarch `uname -m` -R ./test
```

运行上述命令将显示与之前使用 Clang 编译时显示的类似的输出。

让我们再看另一个示例，其中 `std::map` 对象在没有互斥锁的情况下访问。即使访问 map 时使用了不同的键值，由于写入 `std::map` 会使其迭代器失效，也可能会导致数据争用：

```
#include <map>
#include <thread>

std::map<int,int> m;

void Thread1() {
    m[123] = 1;
}

void Thread2() {
    m[345] = 0;
}

int main() {
    std::jthread t1(Thread1);
    std::jthread t2(Thread1);
    return 0;
}
```

编译并运行生成的二进制文件会产生大量输出，其中包含三个警告。这里，仅显示第一个警告中最相关的几行（其他警告类似）：

```
WARNING: ThreadSanitizer: data race (pid=8907)
    Read of size 4 at 0x720c00000020 by thread T2:
    Previous write of size 8 at 0x720c00000020 by thread T1:
    Location is heap block of size 40 at 0x720c00000000 allocated by
thread T1:
    Thread T2 (tid=8910, running) created by main thread at:
```

```
    Thread T1 (tid=8909, finished) created by main thread at:
SUMMARY: ThreadSanitizer: data race test.cpp:11:3 in Thread2()
```

当 t1 和 t2 线程都写入映射 m 时，会标记 TSan 警告。

在下一个示例中，只有一个辅助线程通过指针访问映射，但此线程正在与主线程竞争访问和使用映射。t 线程访问映射 m 以更改 foo 键的值；同时，主线程将其值输出到控制台：

```cpp
#include <iostream>
#include <thread>
#include <map>
#include <string>

typedef std::map<std::string, std::string> map_t;

void *func(void *p) {
    map_t& m = *static_cast<map_t*>(p);
    m["foo"] = "bar";
    return 0;
}

int main() {
    map_t m;
    std::thread t(func, &m);
    std::cout << "foo = " << m["foo"] << '\n';
    t.join();
    return 0;
}
```

编译并运行此示例会生成大量输出，其中包含七个 TSan 警告。这里，仅显示第一个警告。可以通过编译并运行 GitHub 库中的示例来查看完整报告：

```
WARNING: ThreadSanitizer: data race (pid=10505)
    Read of size 8 at 0x721800003028 by main thread:
        #8 main test.cpp:17:28 (test+0xe1d75) (BuildId:
8eef80df1b5c81ce996f7ef2c44a6c8a11a9304f)
    Previous write of size 8 at 0x721800003028 by thread T1:
        #0 operator new(unsigned long) <null> (test+0xe0c3b) (BuildId:
8eef80df1b5c81ce996f7ef2c44a6c8a11a9304f)
        #9 func(void*) test.cpp:10:3 (test+0xe1bb7) (BuildId:
8eef80df1b5c81ce996f7ef2c44a6c8a11a9304f)
    Location is heap block of size 96 at 0x721800003000 allocated by
thread T1:
        #0 operator new(unsigned long) <null> (test+0xe0c3b) (BuildId:
8eef80df1b5c81ce996f7ef2c44a6c8a11a9304f)
        #9 func(void*) test.cpp:10:3 (test+0xe1bb7) (BuildId:
8eef80df1b5c81ce996f7ef2c44a6c8a11a9304f)
    Thread T1 (tid=10507, finished) created by main thread at:
```

```
        #0 pthread_create <null> (test+0x616bf) (BuildId:
8eef80df1b5c81ce996f7ef2c44a6c8a11a9304f)
SUMMARY: ThreadSanitizer: data race test.cpp:17:28 in main
ThreadSanitizer: reported 7 warnings
```

从输出来看，TSan 在访问堆中分配的 std::map 对象时发出数据争用警告。该对象是映射 m 。但 TSan 不仅可以检测由于缺少互斥锁而导致的数据竞争，还可以报告变量何时必须是原子的。

下一个示例展示了该情况。RefCountedObject 类定义的对象可以保留已创建该类对象数量的引用计数。智能指针遵循此想法，当计数器达到值 0 时，在销毁时删除底层分配的内存。

此示例中，仅显示增加和减少引用计数变量 ref_ 的 Ref() 和 Unref() 函数。为了避免在多线程环境中出现问题，ref_ 必须是原子变量。但这里并非如此，t1 和 t2 线程正在修改 ref_，可能会发生数据争用：

```cpp
#include <iostream>
#include <thread>

class RefCountedObject {
public:

    void Ref() {
        ++ref_;
    }

    void Unref() {
        --ref_;
    }

private:
    // ref_ should be atomic to avoid synchronization issues
    int ref_{0};
};

int main() {
    RefCountedObject obj;
    std::jthread t1(&RefCountedObject::Ref, &obj);
    std::jthread t2(&RefCountedObject::Unref, &obj);
    return 0;
}
```

编译并运行此示例显示以下输出：

```
WARNING: ThreadSanitizer: data race (pid=32574)
    Write of size 4 at 0x7fffffffcc04 by thread T2:
        #0 RefCountedObject::Unref() test.cpp:12:9 (test+0xe1dd0)
(BuildId: 448eb3f3d1602e21efa9b653e4760efe46b621e6)
```

```
    Previous write of size 4 at 0x7fffffffcc04 by thread T1:
        #0 RefCountedObject::Ref() test.cpp:8:9 (test+0xe1c00) (BuildId:
448eb3f3d1602e21efa9b653e4760efe46b621e6)
    Location is stack of main thread.
    Location is global '??' at 0x7fffffdd000 ([stack]+0x1fc04)
    Thread T2 (tid=32577, running) created by main thread at:
        #0 pthread_create <null> (test+0x6164f) (BuildId:
448eb3f3d1602e21efa9b653e4760efe46b621e6)
        #2 main test.cpp:23:16 (test+0xe1b94) (BuildId:
448eb3f3d1602e21efa9b653e4760efe46b621e6)
    Thread T1 (tid=32576, finished) created by main thread at:
        #0 pthread_create <null> (test+0x6164f) (BuildId:
448eb3f3d1602e21efa9b653e4760efe46b621e6)
        #2 main test.cpp:22:16 (test+0xe1b56) (BuildId:
448eb3f3d1602e21efa9b653e4760efe46b621e6)
SUMMARY: ThreadSanitizer: data race test.cpp:12:9 in
RefCountedObject::Unref()
ThreadSanitizer: reported 1 warnings
```

TSan 输出显示，在访问先前由 Ref() 函数修改的内存位置时，Unref() 函数中发生了数据竞争情况。

数据竞争还可能发生在从多个线程初始化对象时，这些线程没有任何同步机制。以下示例中，在 init_object() 函数中创建了一个 MyObj 类型的对象，并为全局静态指针 obj 分配了其地址。由于此指针不受互斥锁保护，当 t1 和 t2 线程分别尝试从 func1() 和 func2() 函数创建对象并更新 obj 指针时，就会发生数据竞争：

```cpp
#include <iostream>
#include <thread>

class MyObj {};

static MyObj *obj = nullptr;
void init_object() {
    if (!obj) {
        obj = new MyObj();
    }
}

void func1() {
    init_object();
}

void func2() {
    init_object();
}

```

```cpp
int main() {
    std::thread t1(func1);
    std::thread t2(func2);

    t1.join();
    t2.join();
    return 0;
}
```

这是编译并运行该示例后的输出：

```
WARNING: ThreadSanitizer: data race (pid=32826)
    Read of size 1 at 0x5663912cbae8 by thread T2:
        #0 func2() test.cpp (test+0xe0b68) (BuildId:
12f32c1505033f9839d17802d271fc869b7a3e38)
    Previous write of size 1 at 0x5663912cbae8 by thread T1:
        #0 func1() test.cpp (test+0xe0b3d) (BuildId:
12f32c1505033f9839d17802d271fc869b7a3e38)
    Location is global 'obj (.init)' of size 1 at 0x5663912cbae8
(test+0x150cae8)
    Thread T2 (tid=32829, running) created by main thread at:
        #0 pthread_create <null> (test+0x6062f) (BuildId:
12f32c1505033f9839d17802d271fc869b7a3e38)
    Thread T1 (tid=32828, finished) created by main thread at:
        #0 pthread_create <null> (test+0x6062f) (BuildId:
12f32c1505033f9839d17802d271fc869b7a3e38)
SUMMARY: ThreadSanitizer: data race test.cpp in func2()
ThreadSanitizer: reported 1 warnings
```

输出显示了之前描述的内容，由于 `func1()` 和 `func2()` 访问 `obj` 全局变量而发生数据竞争。

由于 C++11 标准已正式将数据竞争视为未定义行为，现在来看看如何使用 UBSan 来检测程序中的未定义行为问题。

12.2.5. UndefinedBehaviorSanitizer

UBSan 可以检测代码中的未定义行为，例如，当位移位过多、整数溢出或误用空指针时。可以通过指定 `-fsanitize=undefined` 选项来启用，通过设置 `UBSAN_OPTIONS` 变量在运行时修改其行为。

UBSan 能够检测到的许多错误在编译过程中也能用编译器检测到。

来看一个简单的例子：

```cpp
int main() {
    int val = 0x7fffffff;
    val += 1;
```

```
    return 0;
}
```

要编译程序并启用 UBSan，请使用以下命令：

```
$ clang++ -fsanitize=undefined -fno-omit-frame-pointer -g -O2 -o test test.cpp
```

运行结果程序将生成以下输出：

```
test.cpp:3:7: runtime error: signed integer overflow: 2147483647 + 1
cannot be represented in type 'int'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior test.cpp:3:7
```

输出非常简单且不言自明；有一个有符号整数溢出操作。

现在来介绍另一个有用的 C++ 消杀器，用于检测未初始化的内存和其他内存使用问题：MSan。

12.2.6. MemorySanitizer

MSan 可以检测未初始化的内存使用情况，例如，在使用尚未赋值或地址的变量或指针时。还可以跟踪位域中未初始化的位。

要启用 MSan，请使用以下编译器标志：

```
-fsanitize=memory -fPIE -pie -fno-omit-frame-pointer
```

通过指定 -fsanitize-memory-track-origins 选项，还可以跟踪每个未初始化值到其创建位置的内存分配。

GCC 不支持 MSan，因此使用此编译器时 -fsanitize=memory 标志无效。

以下示例中，创建了 arr 整数数组，但仅初始化了其位置 5。将消息打印到控制台时将使用位置 0 处的值，但该值仍未初始化：

```
#include <iostream>

int main() {
    auto arr = new int[10];
    arr[5] = 0;
    std::cout << "Value at position 0 = " << arr[0] << '\n';
    return 0;
}
```

要编译程序并启用 MSan，请使用以下命令：

```
$ clang++ -fsanitize=memory -fno-omit-frame-pointer -g -O2 -o test test.cpp
```

运行结果程序将生成以下输出：

```
==20932==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x5b9fa2bed38f in main test.cpp:6:41
    #3 0x5b9fa2b53324 in _start (test+0x32324) (BuildId:
c0a0d31f01272c3ed59d4ac66b8700e9f457629f)
SUMMARY: MemorySanitizer: use-of-uninitialized-value test.cpp:6:41 in
main
```

再次，输出清楚地显示，当读取 arr 数组中位置 0 的值时，第 6 行使用了未初始化的值。

12.2.7. 其他消杀器

还有其他可用的消杀器在针对某些系统进行开发（例如：内核或实时开发）时很有用：

- 硬件辅助 AddressSanitizer（HWASan）：ASan 的新变体，利用硬件功能忽略指针的顶部字节，从而消耗更少的内存。可以通过指定 -fsanitize=hwaddress 选项来启用。
- RealTimeSanitizer（RTSan）：实时测试工具，用于在具有确定性运行时要求的函数中，调用不安全的方法时检测实时违规。
- FuzzerSanitizer：一种通过向程序输入大量随机数据、检查程序是否崩溃，以及查找内存损坏或其他安全漏洞来检测潜在漏洞的消杀器。
- 与内核相关的清理程序：内核开发人员还可以使用清理程序来跟踪问题，下面列出了其中一些：
  - 内核地址消杀器（Kernel Address Sanitizer，KASAN）
  - 内核并发消杀器（Kernel Concurrency Sanitizer，KCSAN）
  - 内核电子栅栏（Kernel Electric-Fence，KFENCE）
  - 内核内存消杀器（Kernel Memory Sanitizer，KMSAN）
  - 内核线程消杀器（Kernel Thread Sanitizer，KTSAN）

Sanitizer 可以自动发现我们代码中的许多问题。当发现并调试了一些错误，并可以重现导致这些特定错误的场景，就可以方便地设计一些涵盖这些情况的测试，以避免将来更改代码而导致类似的问题或事件。

下一节中，将学习如何测试多线程和异步代码。

12.3. 测试异步代码

最后，探索一些测试异步代码的技术。本节中显示的示例需要 GoogleTest 和 Google Test Mock（gMock）库才能编译。如果不熟悉这些库，请查看官方文档以了解如何安装和使用它们。

单元测试是一种编写小型独立测试的做法，用于验证单个代码单元的功能和行为。单元测试有助于查找和修复错误、重构和提高代码质量、记录和传达底层代码设计，以及促进协作和集成。

本节不会介绍将测试分组为逻辑和描述性套件的最佳方法，也不会介绍何时应使用断言或期望来验证不同变量的值和测试方法结果。本节的目的是提供一些有关如何创建单元测试来测试异步代码的指南，所以最好具备一些有关单元测试或测试驱动开发（TDD）的先前知识。

处理异步代码时的主要困难是，可能在另一个线程中执行，并且通常不知道何时发生或何时完成。

测试异步代码时要遵循的主要方法是尝试将功能与多线程分离，所以希望以同步方式测试异步代码，尝试在一个特定线程中执行它，删除上下文切换，线程创建和销毁，以及其他可能影响测试结果和时间的活动。有时也会使用计时器，等待在超时之前调用回调。

12.3.1. 测试简单的异步函数

从测试异步操作的一个小例子开始。此示例展示了一个 `asyncFunc()` 函数，通过使用 `std::async` 异步运行该函数进行测试：

```
#include <gtest/gtest.h>
#include <chrono>
#include <future>

using namespace std::chrono_literals;

int asyncFunc() {
    std::this_thread::sleep_for(100ms);
    return 42;
}

TEST(AsyncTests, TestHandleAsyncOperation) {
    std::future<int> result = std::async(
                              std::launch::async,
                              asyncFunc);
    EXPECT_EQ(result.get(), 42);
}


int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

`std::async` 返回一个用于检索计算值的未来。本例中，`asyncFunc` 只需等待 `100` 毫秒即可返回值 `42`。如果异步任务正常运行，则测试将通过，因为有一个期望指令检查返回值是否确实是 `42`。

只定义了一个测试，使用 `TEST()` 宏，其中其第一个参数是测试套件名称（在此示例中为 `AsyncTests`），第二个参数是测试名称（`TestHandleAsyncOperation`）。

在 `main()` 函数中，通过调用`::testing::InitGoogleTest()` 初始化 `GoogleTest` 库。此函数解析命令行以获取 `GoogleTest` 识别的标志。调用 `RUN_ALL_TESTS()`，收集并运行所有测试，如果所有测试都成功，则返回 `0`，否则返回 `1`。此函数最初是一个宏，这就是其名称大写的原因。

12.3.2. 使用超时限制测试持续时间

这种方法可能出现的一个问题是，异步任务可能因任何原因而无法安排，完成时间可能比预期的要长，或者由于某些原因而无法完成。为了处理这种情况，可以使用计时器，将其超时时间设置为合理值，以便有足够的时间让测试成功完成。因此，如果计时器超时，测试将失败。以下示例通过使用定时等待 `std::async` 返回的 `Future` 来展示该方法：

```
#include <gtest/gtest.h>
#include <chrono>
#include <future>

using namespace std::chrono;
using namespace std::chrono_literals;

int asyncFunc() {
    std::this_thread::sleep_for(100ms);
    return 42;
}

TEST(AsyncTest, TestTimeOut) {
    auto start = steady_clock::now();
    std::future<int> result = std::async(
                              std::launch::async,
                              asyncFunc);
    if (result.wait_for(200ms) ==
                std::future_status::timeout) {
        FAIL() << "Test timed out!";
    }

    EXPECT_EQ(result.get(), 42);

    auto end = steady_clock::now();
    auto elapsed = duration_cast<milliseconds>(
                                  end - start);
    EXPECT_LT(elapsed.count(), 200);
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

现在，调用 `Future` 对象结果的 `wait_for()` 函数，等待 200 毫秒让异步任务完成。由于任务将在 100 毫秒内完成，因此超时不会过期。如果出于任何原因，使用低于 100 毫秒的值调用 `wait_for()`，它将超时，并且将调用 `FAIL()` 宏，从而使测试失败。

测试继续运行并检查返回值是否为 42（如上例所示），然后检查运行异步任务所花费的时间是否小于使用的超时时间。

12.3.3. 测试回调

测试回调是一项相关任务，尤其是在实现库和应用程序编程接口（API）时。以下示例显示如何测试已调用回调及其结果：

```
#include <gtest/gtest.h>
#include <chrono>
#include <functional>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

void asyncFunc(std::function<void(int)> callback) {
    std::thread([callback]() {
        std::this_thread::sleep_for(1s);
        callback(42);
    }).detach();
}

TEST(AsyncTest, TestCallback) {
    int result = 0;
    bool callback_called = false;

    auto callback = [&](int value) {
        callback_called = true;
        result = value;
    };

    asyncFunc(callback);

    std::this_thread::sleep_for(2s);
    EXPECT_TRUE(callback_called);
    EXPECT_EQ(result, 42);
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

TestCallback 测试只是将回调定义为接受参数的 lambda 函数，此 lambda 函数通过引用捕获存储值参数的结果变量，以及默认情况下为 false ，并在调用回调时设置为 true 的 callback_called 布尔变量。

然后，测试调用 asyncFunc( ) 函数，该函数会生成一个线程，等待一秒钟，然后调用回调并传递值 42。测试等待两秒钟，然后使用 EXPECT_TRUE 宏检查回调是否已调用，并检查 callback_called 的值，以及结果是否具有预期值 42。

12.3.4. 测试事件驱动的软件

第 9 章中了解了如何使用 Boost.Asio 及其事件队列来分派异步任务。在事件驱动编程中，还需要测试回调，如上例所示。可以设置测试来注入回调，并在调用后验证结果。以下示例显示如何在 Boost.Asio 程序中测试异步任务：

```
#include <gtest/gtest.h>
#include <boost/asio.hpp>
#include <chrono>
#include <thread>

using namespace std::chrono_literals;

void asyncFunc(boost::asio::io_context& io_context,
               std::function<void(int)> callback) {
    io_context.post([callback]() {
        std::this_thread::sleep_for(100ms);
        callback(42);
    });
}

TEST(AsyncTest, BoostAsio) {
    boost::asio::io_context io_context;

    int result = 0;
    asyncFunc(io_context, [&result](int value) {
        result = value;
    });

    std::jthread io_thread([&io_context]() {
        io_context.run();
    });

    std::this_thread::sleep_for(150ms);
    EXPECT_EQ(result, 42);
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

BoostAsio 测试首先创建一个 I/O 执行上下文对象 io_context，并将其与实现任务或回调的 lambda 函数一起传递给 asyncFunc() 函数以在后台运行。此回调只是将 lambda 函数捕获的结果变量的值，设置为传递给它的值。

asyncFunc() 函数仅使用 io_context 来发布一个由 lambda 函数组成的任务，该函数在等待 100 毫秒后调用值为 42 的回调。

然后，测试只需等待 150 毫秒让后台任务完成，并检查结果值是否为 42 以将测试标记为通过。

12.3.5. 模拟外部资源

如果异步代码还依赖于外部资源，例如：文件访问、网络服务器、计时器或其他模块，可能需要模拟，并避免由于资源问题而导致测试中出现不必要的故障。模拟是用于测试目的的技术，用于用虚假或简化的对象或函数替换或修改真实对象或函数的行为。这样，可以控制异步代码的输入和输出，并避免副作用或来自其他因素的干扰。

例如，测试的代码依赖于服务器，则服务器可能无法连接或执行其任务，从而导致测试失败。这些情况下，失败是由于资源问题，而不是由于正在测试的异步代码，从而导致错误且通常是暂时的失败。可以使用模仿其接口的自己的模拟类来模拟外部资源。

来看一个例子，了解如何使用模拟类并使用依赖项注入来使用该类进行测试。在此示例中，有一个外部资源 `AsyncTaskScheduler`，其 `runTask()` 方法用于执行异步任务。由于我们只想测试异步任务并消除异步任务调度程序可能产生的任何不良副作用，因此可以使用模拟 `AsyncScheduler` 接口的模拟类。此类是 `MockTaskScheduler`，继承自 `AsyncTaskScheduler` 并实现其 `runTask()` 基类方法，其中任务是同步运行的：

```
#include <gtest/gtest.h>
#include <functional>

class AsyncTaskScheduler {
    public:
    virtual int runTask(std::function<int()> task) = 0;
};

class MockTaskScheduler : public AsyncTaskScheduler {
    public:
    int runTask(std::function<int()> task) override {
        return task();
    }
};

TEST(AsyncTests, TestDependencyInjection) {
    MockTaskScheduler scheduler;

    auto task = []() -> int {
        return 42;
    };

    int result = scheduler.runTask(task);
    EXPECT_EQ(result, 42);
}

```

```
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

TestDependencyInjection 测试只是以 lambda 函数的形式创建一个 MockTaskScheduler 对象和一个任务，并使用模拟对象通过运行 runTask() 函数来执行任务。一旦任务运行，结果将为 42。

除了完整定义模拟类之外，还可以使用 gMock 库并仅模拟所需的方法。此示例展示了 gMock 的实际操作：

```
#include <gmock/gmock.h>
#include <gtest/gtest.h>
#include <functional>

class AsyncTaskScheduler {
    public:
    virtual int runTask(std::function<int()> task) = 0;
};

class MockTaskScheduler : public AsyncTaskScheduler {
    public:
    MOCK_METHOD(int, runTask, (std::function<int()> task), (override));
};

TEST(AsyncTests, TestDependencyInjection) {
    using namespace testing;

    MockTaskScheduler scheduler;

    auto task = []() -> int {
        return 42;
    };

    EXPECT_CALL(scheduler, runTask(_)).WillOnce(
        Invoke(task)
    );

    auto result = scheduler.runTask(task);
    EXPECT_EQ(result, 42);
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

现在，MockTaskScheduler 也继承自 AsyncTaskScheduler，其中定义了接口，但不是重写其方法，而是使用 MOCK_METHOD 宏，其中传递返回类型、模拟方法名称及其参数。

然后，TestMockMethod 测试使用 EXPECT_CALL 宏定义对 MockTaskScheduler 中 runTask() 模拟方法的预期调用，该调用只会发生一次并调用 lambda 函数任务，该任务返回值 42。

该调用恰好发生在调用 scheduler.runTask() 的下一条指令中，将返回值存储在结果中。测试通过检查结果是否为预期值 42 结束。

12.3.6. 测试异常和失败

异步任务并不总是成功并产生有效结果。有时可能会出错（网络故障、超时、异常等），返回错误或抛出异常是让用户了解这种情况的方法。我们应该模拟失败，以确保代码能够妥善处理这些情况。

测试错误或异常可以按照通常的方式进行，即使用 try-catch 块并使用断言或期望来检查是否抛出错误并使测试成功或失败。GoogleTest 还提供了 EXPECT_ANY_THROW() 宏，可简化检查是否发生异常的过程。以下示例显示了这两种方法：

```
#include <gtest/gtest.h>
#include <chrono>
#include <future>
#include <iostream>
#include <stdexcept>

using namespace std::chrono_literals;

int asyncFunc(bool should_fail) {
    std::this_thread::sleep_for(100ms);
    if (should_fail) {
        throw std::runtime_error("Simulated failure");
    }
    return 42;
}

TEST(AsyncTest, TestAsyncFailure1) {
    try {
        std::future<int> result = std::async(
                                  std::launch::async,
                                  asyncFunc, true);
        result.get();
        FAIL() << "No expected exception thrown";
    } catch (const std::exception& e) {
        SUCCEED();
    }
}

```

```
TEST(AsyncTest, TestAsyncFailure2) {
    std::future<int> result = std::async(
                              std::launch::async,
                              asyncFunc, true);
    EXPECT_ANY_THROW(result.get());
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

TestAsyncFailure1 和 TestAsyncFailure2 测试非常相似。两者都异步执行 asyncFunc() 函数，该函数现在接受 should_fail 布尔参数，指示任务是否应成功并返回值 42 或失败并抛出异常。两个测试都使任务失败，不同之处在于，如果没有抛出异常，TestAsyncFailure1 将使用 F AIL() 宏，使测试失败，或者如果 try-catch 块捕获到异常，则使用 SUCCEED()，而 TestAsy ncFailure2 使用 EXPECT_ANY_THROW() 宏来检查在尝试通过调用其 get() 方法从未来结果中检索结果时是否发生异常。

12.3.7. 测试多个线程

在 C++ 中测试涉及多个线程的异步软件时，一种常见且有效的技术是使用条件变量来同步线程。正如在第 4 章中看到的，条件变量允许线程等待某些条件满足后再继续，这使得它们对于管理线程间通信和协调至关重要。

接下来是一个例子，其中多个线程执行一些任务，而主线程等待所有其他线程完成。

首先定义一些必要的全局变量，例如：线程总数（num_threads）、计数器（每次调用异步任务时都会增加的原子变量），以及条件变量 cv 及其关联的互斥锁 mtx，这将有助于在所有异步任务完成后解除主线程的阻塞：

```
#include <gtest/gtest.h>
#include <atomic>
#include <chrono>
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <syncstream>
#include <thread>
#include <vector>

using namespace std::chrono_literals;

#define sync_cout std::osyncstream(std::cout)

std::condition_variable cv;
```

```
std::mutex mtx;

bool ready = false;
std::atomic<unsigned> counter = 0;

const std::size_t num_threads = 5;
```

asyncTask() 函数将执行异步任务（在此示例中只需等待 100 毫秒），增加计数器原子变量并通过 cv 条件变量通知主线程其工作已完成：

```
void asyncTask(int id) {
    sync_cout << "Thread " << id << ": Starting work..."
              << std::endl;
    std::this_thread::sleep_for(100ms);
    sync_cout << "Thread " << id << ": Work finished."
              << std::endl;

    ++counter;
    cv.notify_one();
}
```

TestMultipleThreads 测试将首先生成多个线程，每个线程将异步运行 asyncTask() 任务。然后，使用条件变量等待，该条件变量的计数器值与线程数相同，所以所有后台任务都已完成工作。条件变量使用 wait_for() 函数设置 150 毫秒的超时时间，以限制测试可以运行的时间，但为所有后台任务成功完成留出一些空间：

```
TEST(AsyncTest, TestMultipleThreads) {
    std::vector<std::jthread> threads;

    for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back(asyncTask, i + 1);
    }

    {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait_for(lock, 150ms, [] {
            return counter == num_threads;
        });
        sync_cout << "All threads have finished."
                  << std::endl;
    }
    EXPECT_EQ(counter, num_threads);
}
```

通过检查计数器是否确实具有与 num_threads 相同的值，测试完成了。

最后实现 main() 函数：

```
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

如前所述，程序首先通过调用::testing::InitGoogleTest( ) 初始化 GoogleTest 库，然后调用 RUN_ALL_TESTS( ) 来收集和运行所有测试。

12.3.8. 测试协程

在 C++20 中，协程提供了一种编写和管理异步代码的新方法。可以使用与其他异步代码类似的方法测试基于协程的代码，但有一个细微的差别，即协程可以暂停和恢复。

来看一个简单协程的例子。

在第 8 章中，协程有一些样板代码来定义其 promise 类型和可等待方法。让我们从实现将定义协程的 Task 结构开始。

首先定义任务结构：

```
#include <gtest/gtest.h>
#include <coroutine>
#include <exception>
#include <iostream>

struct Task {
    struct promise_type;
    using handle_type =
                std::coroutine_handle<promise_type>;

    handle_type handle_;

    Task(handle_type h) : handle_(h) {}
    ~Task() {
        if (handle_) handle_.destroy();
    }

    // struct promise_type definition
    // and await methods
};
```

在 Task 内部，定义了 promise_type，描述了如何管理协程。此类型提供了某些预定义方法（钩子），用于控制如何返回值、如何暂停协程以及协程完成后如何管理资源：

```
struct Task {
    // ...
    struct promise_type {
        int result_;
```

```
        std::exception_ptr exception_;

        Task get_return_object() {
            return Task(handle_type::from_promise(*this));
        }

        std::suspend_always initial_suspend() {
            return {};
        }

        std::suspend_always final_suspend() noexcept {
            return {};
        }

        void return_value(int value) {
            result_ = value;
        }

        void unhandled_exception() {
            exception_ = std::current_exception();
        }
    };
    // ....
};
```

然后，实现控制协程的暂停和恢复的方法：

```
struct Task {
    // ...
    bool await_ready() const noexcept {
        return handle_.done();
    }

    void await_suspend(std::coroutine_handle<>
                           awaiting_handle) {
        handle_.resume();
        awaiting_handle.resume();
    }

    int await_resume() {
        if (handle_.promise().exception_) {
            std::rethrow_exception(
            handle_.promise().exception_);
        }
        return handle_.promise().result_;
    }

    int result() {
```

```
        if (handle_.promise().exception_) {
            std::rethrow_exception(
            handle_.promise().exception_);
        }
        return handle_.promise().result_;
    }
    // ....
};
```

有了 Task 结构后，定义两个协同程序，一个用于计算有效值，另一个用于引发异常：

```
Task asyncFunc(int x) {
    co_return 2 * x;
}

Task asyncFuncWithException() {
    throw std::runtime_error("Exception from coroutine");
    co_return 0;
}
```

由于 GoogleTest 中 TEST() 宏内的测试函数不能直接作为协程，它们没有与之关联的 promise_type 结构，所以需要定义一些辅助函数：

```
Task testCoroutineHelper(int value) {
    co_return co_await asyncFunc(value);
}

Task testCoroutineWithExceptionHelper() {
    co_return co_await asyncFuncWithException();
}
```

有了这个，就可以实施测试了：

```
TEST(AsyncTest, TestCoroutine) {
    auto task = testCoroutineHelper(5);
    task.handle_.resume();
    EXPECT_EQ(task.result(), 10);
}

TEST(AsyncTest, TestCoroutineWithException) {
    auto task = testCoroutineWithExceptionHelper();
    EXPECT_THROW({
            task.handle_.resume();
            task.result();
        },
        std::runtime_error);
}
```

```
15
16 int main(int argc, char **argv) {
17     ::testing::InitGoogleTest(&argc, argv);
18     return RUN_ALL_TESTS();
19 }
```

TestCoroutine 测试使用 testCoroutineHelper() 辅助函数并传递值 5 来定义一项任务。当恢复协程时，预计它将返回双倍的值，因此返回值 10，使用 EXPECT_EQ() 进行测试。

TestCoroutineWithException 测试使用类似的方法，现在使用 testCoroutineWithExceptionHelper() 辅助函数，该函数将在协程恢复时抛出异常。这正是在检查异常是否属于 std::runtime_error 类型之前 EXPECT_THROW() 断言宏内部发生的情况。

12.3.9. 压力测试

通过执行压力测试可以实现竞争条件检测器。对于高度并发或多线程异步代码，压力测试至关重要。可以使用多个异步任务模拟高负载，以检查系统在压力下是否正常运行。此外，使用随机延迟、线程交错或压力测试工具来减少确定性条件、增加测试覆盖率也很重要。

下一个示例显示了压力测试的实现，该测试生成 100（total_nums）个线程，这些线程执行异步任务，其中原子变量计数器在随机等待后每次运行都会增加：

```
1  #include <gtest/gtest.h>
2  #include <atomic>
3  #include <chrono>
4  #include <iostream>
5  #include <thread>
6  #include <vector>
7
8  std::atomic<int> counter(0);
9  const std::size_t total_runs = 100;
10
11 void asyncIncrement() {
12     std::this_thread::sleep_for(std::chrono::milliseconds(rand() %
13     100));
14     counter.fetch_add(1);
15 }
16
17 TEST(AsyncTest, StressTest) {
18     std::vector<std::thread> threads;
19
20     for (std::size_t i = 0; i < total_runs; ++i) {
21         threads.emplace_back(asyncIncrement);
22     }
23     for (auto& thread : threads) {
24         thread.join();
```

```
    }
    EXPECT_EQ(counter, total_runs);
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

如果计数器的值与线程总数相同，则测试成功。

12.3.10. 并行测试

为了更快地运行测试套件，可以并行化在不同线程中运行的测试，但测试必须是独立的，每个测试都作为同步单线程解决方案在特定线程中运行。此外，需要设置和拆除必要的对象，而不保留以前测试运行的状态。

当将 CMake 与 GoogleTest 一起使用时，可以通过使用以下命令指定要使用的并发作业数来并行运行所有检测到的测试：

```
$ ctest -j <num_jobs>
```

本节中显示的所有示例只是测试异步代码的一小部分，这些技术能够提供足够的洞察力和知识，以便开发进一步的测试技术来应对可能遇到的特定场景。

12.4. 总结

本章中，介绍了如何清理和测试异步程序。

首先接扫如何使用清理器清理代码，以帮助查找多线程和异步问题，例如：竞争条件、内存泄漏、使用范围后错误，以及许多其他问题。

然后，描述了一些针对异步软件的测试技术，使用 GoogleTest 作为测试库。

使用这些工具和技术有助于检测和预防未定义行为、内存错误和安全漏洞，同时确保并发操作正确执行、时序问题得到妥善处理以及代码在各种条件下按预期执行。这提高了整个程序的可靠性和稳定性。

下一章中，将介绍可用于改善异步程序运行时间和资源使用情况的性能和优化技术。

12.5. 扩展阅读

- Sanitizers: https://github.com/google/sanitizers
- Clang 20.0 ASan: https://clang.llvm.org/docs/AddressSanitizer.html
- Clang 20.0 hardware-assisted ASan: https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html
- Clang 20.0 TSan: https://clang.llvm.org/docs/ThreadSanitizer.html

- Clang 20.0 MSan: https://clang.llvm.org/docs/MemorySanitizer.html
- Clang 20.0 UBSan:http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
- Clang 20.0 DataFlowSanitizer: https://clang.llvm.org/docs/DataFlowSanitizer.html
- Clang 20.0 LSan: https://clang.llvm.org/docs/LeakSanitizer.html
- Clang 20.0 RealtimeSanitizer: https://clang.llvm.org/docs/RealtimeSanitizer.html
- Clang 20.0 SanitizerCoverage: https://clang.llvm.org/docs/SanitizerCoverage.html
- Clang 20.0 SanitizerStats: https://clang.llvm.org/docs/SanitizerStats.html
- GCC: Program Instrumentation Options: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html
- Apple Developer: Diagnosing memory, thread, and crash issues early: https://developer.apple.com/documentation/xcode/diagnosing-memory-thread-andcrash-issues-early
- GCC: Options for Debugging Your Program: https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html
- OpenSSL: Compiler Options Hardening Guide for C and C++: https://best.openssf.org/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guidefor-C-and-C++.html
- Memory error checking in C and C++: Comparing Sanitizers and Valgrind: https://developers.redhat.com/blog/2021/05/05/memory-error-checkingin-c-and-c-comparing-sanitizers-and-valgrind
- The GNU C Library: https://www.gnu.org/software/libc
- Sanitizers: Common flags: https://github.com/google/sanitizers/wiki/SanitizerCommonFlags
- AddressSanitizer flags: https://github.com/google/sanitizers/wiki/AddressSanitizerFlags
- AddressSanitizer: A Fast Address Sanity Checker: https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf
- MemorySanitizer: Fast detector of uninitialized memory use in C++: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43308.pdf
- Linux Kernel Sanitizers: https://github.com/google/kernel-sanitizers
- TSan flags: https://github.com/google/sanitizers/wiki/ThreadSanitizerFlags
- TSan: Popular data races: https://github.com/google/sanitizers/wiki/ThreadSanitizerPopularDataRaces
- TSan report format: https://github.com/google/sanitizers/wiki/ThreadSaniti

zerReportFormat

- TSan algorithm: https://github.com/google/sanitizers/wiki/ThreadSanitizerAlgorithm
- Address space layout randomization: https://en.wikipedia.org/wiki/Address_space_layout_randomization
- GoogleTest User's Guide: https://google.github.io/googletest

第 13 章 提高异步软件性能

本章中，将介绍异步代码的性能方面。代码性能和优化是一个深奥而复杂的主题，无法在一章中涵盖所有内容。我们的目标是通过一些如何衡量性能和优化代码的示例为读者提供有关该主题的介绍。

本章将涵盖以下关键主题：

- 专注于多线程应用程序的性能测量工具
- 什么是伪共享，如何发现它，以及如何修复/改进代码
- 现代 CPU 内存缓存架构简介
- 回顾一下实现的单生产者单消费者（SPSC）无锁队列（第 5 章）

13.1. 技术要求

与前几章一样，需要一个支持 C++20 的现代 C++ 编译器。我们将使用 GCC 13 和 Clang 1 8。还需要一台运行 Linux 的 Intel/AMD 多核 CPU 的 PC。在本章中，我们使用了在 CPU AMD Ryzen Threadripper Pro 5975WX（32 核）的工作站上运行的 Ubuntu 24.04 LTS。具有 8 个内核的 CPU 是理想的，但 4 个内核足以运行示例。

还将使用 Linux perf 工具，将在本书后面解释如何获取和安装这些工具。

本章的示例可以在本书的 GitHub 库中找到：https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP。

13.2. 性能测量工具

要了解应用程序的性能，需要能够对其进行测量。如果本章有一个关键要点，那就是永远不要估计或猜测代码性能。要知道程序是否满足其性能要求（延迟或吞吐量），需要测试、测试，然后再测试。

获得性能测试数据后，就会知道代码中的热点。其可能与内存访问模式或线程争用有关（例如，多个线程必须等待获取锁才能访问资源）。这是第二个最重要的要点发挥作用的地方：在优化应用程序时设定目标。不要以实现最佳性能为目标，因为总会有改进的空间。正确的做法是设定一个明确的规范，例如：事务的最大处理时间或每秒处理的网络数据包数量。

考虑到这两个主要思想，可以用来测量代码性能的不同方法开始。

13.2.1. 码内分析

开始了解代码性能的一个非常简单但有用的方法是代码内分析，包括添加一些代码来测量某些代码段的执行时间。在编写代码时，这种方法很适合用作工具（需要访问源码），这能够发现代码中的一些性能问题。

我们将使用 std::chrono 作为分析代码的初始方法。

下面的代码展示了如何使用 std::chrono 对我们的程序进行一些基本分析：

```
auto start = std::chrono::high_resolution_clock::now();
// processing to profile
auto end = std::chrono::high_resolution_clock::now();

auto duration = std::chrono::duration_
cast<std::chrono::milliseconds>(end - start);
std::cout < duration.count() << " milliseconds\n";
```

这里，获取两个时间点，调用 high_resolution_clock::now( ) 并转换为毫秒的时间段。根据估计处理所需的时间，可以使用微秒或秒等单位。使用这种简单的技术，可以轻松了解处理所需的时间，并且可以轻松比较不同的选项。

这里，std::chrono::high_resolution_clock 是提供最高精度（实现提供的最小滴答周期）的时钟类型。C++ 标准库允许它成为 std::chrono::system_clock 或 std::chrono::steady_clock 的别名。libstdc++ 将其别名为 std::chrono::system_clock，而 libc++ 使用 std::chrono::steady_clock。对于本章中的示例，使用了 GCC 和 libstdc++。时钟分辨率为 1 纳秒：

```
/**
 * @brief Highest-resolution clock
 *
 * This is the clock "with the shortest tick period." Alias to
 * std::system_clock until higher-than-nanosecond definitions
 * become feasible.
 * @ingroup chrono
*/
using high_resolution_clock = system_clock;
```

现在，来看一个完整的示例，该示例分析了两个 C++ 标准库算法来对向量进行排序–std::sort 和 std::stable_sort：

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include <random>
#include <utility>

int uniform_random_number(int min, int max) {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    std::uniform_int_distribution dis(min, max);
    return dis(gen);
}

std::vector<int> random_vector(std::size_t n, int32_t min_val, int32_t
max_val) {
    std::vector<int> rv(n);
```

```
    std::ranges::generate(rv, [&] {
        return uniform_random_number(min_val, max_val);
    });
    return rv;
}

using namespace std::chrono;

int main() {
    constexpr uint32_t elements = 100000000;
    int32_t minval = 1;
    int32_t maxval = 1000000000;

    auto rv1 = random_vector(elements, minval, maxval);
    auto rv2 = rv1;

    auto start = high_resolution_clock::now();
    std::ranges::sort(rv1);
    auto end = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end - start);
    std::cout << "Time to std::sort "
              << elements << " elements with values in ["
              << minval << "," << maxval << "] "
              << duration.count() << " milliseconds\n";

    start = high_resolution_clock::now();
    std::ranges::stable_sort(rv2);
    end = high_resolution_clock::now();
    duration = duration_cast<milliseconds>(end - start);
    std::cout << "Time to std::stable_sort "
              << elements << " elements with values in ["
              << minval << "," << maxval << "] "
              << duration.count() << " milliseconds\n";
    return 0;
}
```

上述代码生成一个正态分布的随机数向量，然后使用 `std::sort()` 和 `std::stable_sort()` 对向量进行排序。这两个函数都对向量进行排序，但 `std::sort()` 使用快速排序和插入排序算法的组合（称为 introsort），而 `std::stable_sort()` 使用合并排序。排序是稳定的，等效键在原始向量和排序后的向量中具有相同的顺序。对于整数向量，这并不重要，但如果向量有三个具有相同值的元素，则对向量进行排序后，数字将处于相同的顺序。

运行代码后，得到以下输出：

```
Time to std::sort 100000000 elements with values in [1,1000000000]
6019 milliseconds
Time to std::stable_sort 100000000 elements with values in
```

```
[1,1000000000] 7342 milliseconds
```

这个例子中，std::stable_sort() 比 std::sort() 慢。

本节中，介绍了一种测量代码部分运行时间的简单方法。这种方法具有侵入性，需要修改代码；主要用于开发应用程序。在下一节中，将介绍另一种测量执行时间的方法，称为微基准测试。

13.2.2. 代码微基准测试

有时，只想单独分析一小段代码。可能需要多次运行它，然后获取平均运行时间，或者使用不同的输入数据运行它。在这些情况下，可以使用基准测试（也称为微基准测试）库来做到这一点 - 在不同条件下执行代码的一小部分。

必须使用微基准测试作为指导。记住，代码是独立运行的，由于代码不同部分之间存在许多复杂的交互，当一起运行所有代码时，这可能会给我们带来截然不同的结果。请谨慎使用，并注意微基准测试可能会产生误导。

可以使用许多库来对代码进行基准测试。这里使用 Google Benchmark，这是一个非常好且著名的库。

首先获取代码并编译库。获取代码，请运行以下命令：

```
git clone https://github.com/google/benchmark.git
cd benchmark
git clone https://github.com/google/googletest.git
```

获得了基准和 Google 测试库（后者需要编译前者）的代码后，就来进行构建。

为构建创建一个目录：

```
mkdir build
cd build
```

这样，在基准目录中创建了构建目录。

接下来，将使用 CMake 来配置构建并创建 make 所需的所有信息：

```
cmake .. -DCMAKE_BUILD_TYPE=Release -DBUILD_SHARED_LIBRARIES=ON
↪ -DCMAKE_INSTALL_PREFIX=/usr/lib/x86_64-linux-gnu/
```

最后，运行 make 来构建并安装库：

```
make -j16
sudo make install
```

还需要将库添加到 CmakeLists.txt 文件中，已经在本书的代码中为您完成了这一操作。

安装 Google Benchmark 后，需要通过一个包含一些基准测试函数的示例来了解，如何使用该库进行一些基本的基准测试。

注意，std::chrono 和 Google Benchmark 都不是用于处理异步/多线程代码的专用工具，而更像是通用工具。

这是使用 Google Benchmark 的第一个示例：

```cpp
#include <benchmark/benchmark.h>
#include <algorithm>
#include <chrono>
#include <iostream>
#include <random>
#include <thread>

void BM_vector_push_back(benchmark::State& state) {
    for (auto _ : state) {
        std::vector<int> vec;
        for (int i = 0; i < state.range(0); i++) {
            vec.push_back(i);
        }
    }
}

void BM_vector_emplace_back(benchmark::State& state) {
    for (auto _ : state) {
        std::vector<int> vec;
        for (int i = 0; i < state.range(0); i++) {
            vec.emplace_back(i);
        }
    }
}

void BM_vector_insert(benchmark::State& state) {
    for (auto _ : state) {
        std::vector<int> vec;
        for (int i = 0; i < state.range(0); i++) {
            vec.insert(vec.begin(), i);
        }
    }
}

BENCHMARK(BM_vector_push_back)->Range(1, 1000);
BENCHMARK(BM_vector_emplace_back)->Range(1, 1000);
BENCHMARK(BM_vector_insert)->Range(1, 1000);

int main(int argc, char** argv) {
    benchmark::Initialize(&argc, argv);
    benchmark::RunSpecifiedBenchmarks();
    return 0;
}
```

需要包含库头文件：

```
#include <benchmark/benchmark.h>
```

所有基准测试函数均具有以下签名：

```
void benchmark_function(benchmark::State& state);
```

这是一个带有一个参数的函数，即 benchmark::State& state，返回 void。benchmark::State 参数具有双重用途：

- 控制迭代循环：benchmark::State 对象用于控制基准测试函数或代码的执行次数。通过重复测试足够多次以尽量减少变化并收集有意义的数据，这有助于准确测量性能。
- 测量时间和统计数据：状态对象跟踪基准代码的运行时间，并提供报告指标（例如：经过时间、迭代和自定义计数器）的机制。

实现了三个函数，以不同的方式对向 std::vector 序列添加元素进行基准测试：第一个函数使用 std::vector::push_back，第二个函数使用 std::vector::emplace_back，第三个函数使用 std::vector::insert。前两个函数在向量的末尾添加元素，而第三个函数在向量的开头添加元素。

当实现了基准测试函数，就需要告诉库必须作为基准运行：

```
BENCHMARK(BM_vector_push_back)->Range(1, 1000);
```

我们使用 BENCHMARK 宏来执行此操作。对于此示例中的基准测试，设置每次迭代中要插入到向量中的元素数量。范围从 1 到 1000，每次迭代将插入前一次迭代的元素数量的八倍，直到达到最大值。在本例中，将插入 1、8、64、512 和 1,000 个元素。

当我们运行第一个基准测试程序时，可以得到以下输出：

```
2024-10-17T05:02:37+01:00
Running ./13x02-benchmark_vector
Run on (64 X 3600 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x32)
  L1 Instruction 32 KiB (x32)
  L2 Unified 512 KiB (x32)
  L3 Unified 32768 KiB (x4)
Load Average: 0.00, 0.02, 0.16
---------------------------------------------------------------------
Benchmark                           Time               CPU    Iterations
---------------------------------------------------------------------
BM_vector_push_back/1          10.5 ns           10.5 ns      63107997
BM_vector_push_back/8          52.0 ns           52.0 ns      13450361
BM_vector_push_back/64          116 ns            116 ns       6021740
BM_vector_push_back/512         385 ns            385 ns       1819732
BM_vector_push_back/1000        641 ns            641 ns       1093474
```

```
BM_vector_emplace_back/1       10.8 ns          10.8 ns      64570848
BM_vector_emplace_back/8       53.3 ns          53.3 ns      13139191
BM_vector_emplace_back/64       108 ns           108 ns       6469997
BM_vector_emplace_back/512      364 ns           364 ns       1924992
BM_vector_emplace_back/1000     616 ns           616 ns       1138392
BM_vector_insert/1             10.6 ns          10.6 ns      65966159
BM_vector_insert/8             58.6 ns          58.6 ns      11933446
BM_vector_insert/64             461 ns           461 ns       1485319
BM_vector_insert/512           7249 ns          7249 ns         96756
BM_vector_insert/1000         23352 ns         23348 ns         29742
```

首先，该程序打印有关基准测试执行的信息：日期和时间、可执行文件名称，以及有关其运行的 CPU 信息。

看一下下面这一行：

```
Load Average: 0.00, 0.02, 0.16
```

这一行给出了 CPU 负载的估计值：从 0.0（完全没有负载或负载很低）到 1.0（满载）。这三个数字分别对应过去 5、10 和 15 分钟的 CPU 负载。

输出 CPU 负载信息后，基准测试会打印每次迭代的结果。以下是示例：

```
BM_vector_push_back/64     116 ns          116 ns      6021740
```

所以在将 64 个元素插入向量时，BM_vector_push_back 调用了 6,021,740 次（迭代次数）。

时间和 CPU 列给出了每次迭代的平均时间：

- 时间：这是每次基准测试执行从开始到结束的实际时间，包括基准测试期间发生的所有事情：CPU 计算、I/O 操作、上下文切换等。
- CPU 时间：这是 CPU 处理基准测试指令所花费的时间，可以小于或等于时间。

在基准测试中，由于操作简单，可以看到时间和 CPU 大致相同。

从结果来看，可以得出以下结论：

- 对于简单对象（例如：32 位整数），push_back 和 emplace_back 所花费的时间相同。
- 对于少量元素，insert 所花的时间与 push_back/emplace_back 相同，但从 64 个元素开始，所花的时间就多得多，因为 insert 必须在每次插入后复制所有元素（将元素插入到向量的开头）。

下面的示例也对 std::vector 序列进行排序，但这次将使用微基准测试来测量执行时间：

```
#include <benchmark/benchmark.h>

#include <algorithm>
#include <chrono>
#include <iostream>
```

```cpp
#include <random>
#include <thread>

std::vector<int> rv1, rv2;

int uniform_random_number(int min, int max) {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    std::uniform_int_distribution dis(min, max);
    return dis(gen);
}

std::vector<int> random_vector(std::size_t n, int32_t min_val, int32_t
max_val) {
    std::vector<int> rv(n);
    std::ranges::generate(rv, [&] {
        return uniform_random_number(min_val, max_val);
    });
    return rv;
}

static void BM_vector_sort(benchmark::State& state, std::vector<int>&
vec) {
    for (auto _ : state) {
        std::ranges::sort(vec);
    }
}

static void BM_vector_stable_sort(benchmark::State& state,
std::vector<int>& vec) {
    for (auto _ : state) {
        std::ranges::stable_sort(vec);
    }
}

BENCHMARK_CAPTURE(BM_vector_sort, vector, rv1)->Iterations(1)-
>Unit(benchmark::kMillisecond);
BENCHMARK_CAPTURE(BM_vector_stable_sort, vector, rv2)->Iterations(1)-
>Unit(benchmark::kMillisecond);

int main(int argc, char** argv) {
    constexpr uint32_t elements = 100000000;
    int32_t minval = 1;
    int32_t maxval = 1000000000;

    rv1 = random_vector(elements, minval, maxval);
    rv2 = rv1;
    benchmark::Initialize(&argc, argv);
```

```
    benchmark::RunSpecifiedBenchmarks();

    return 0;
}
```

上述代码生成一个随机数向量。在这里，运行两个基准测试函数来对向量进行排序：一个使用 `std::sort`，另一个使用 `std::stable_sort`。注意，使用了同一向量的两个副本，因此两个函数的输入相同。

以下代码行使用了 `BENCHMARK_CAPTURE` 宏。此宏允许将参数传递给基准测试函数 - 在本例中为 `std::vector` 的引用（通过引用传递以避免复制向量，并影响基准测试结果）。

将结果指定为以毫秒，而不是纳秒为单位：

```
BENCHMARK_CAPTURE(BM_vector_sort, vector, rv1)->Iterations(1)-
>Unit(benchmark::kMillisecond);
```

基准测试结果如下：

```
-------------------------------------------------------------------
Benchmark                         Time             CPU       Iterations
-------------------------------------------------------------------
BM_vector_sort                5877 ms         5876 ms                1
BM_vector_stable_sort.        7172 ms         7171 ms                1
```

结果与使用 `std::chrono` 测量时间得到的结果一致。

对于最后一个 Google Benchmark 示例，我们将创建一个线程 (`std::thread`)：

```
#include <benchmark/benchmark.h>

#include <algorithm>
#include <chrono>
#include <iostream>
#include <random>
#include <thread>

static void BM_create_terminate_thread(benchmark::State& state) {
    for (auto _ : state) {
        std::thread thread([]{ return -1; });
        thread.join();
    }
}

BENCHMARK(BM_create_terminate_thread)->Iterations(2000);

int main(int argc, char** argv) {
    benchmark::Initialize(&argc, argv);
    benchmark::RunSpecifiedBenchmarks();
```

```
    return 0;
}
```

这个例子很简单：BM_create_terminate_thread 创建一个线程（不执行任何操作，仅返回 0）并等待它结束（thread.join()）。我们运行 2000 次迭代来估计创建线程所需的时间。

结果如下：

```
------------------------------------------------------------
Benchmark                        Time        CPU      Iterations
------------------------------------------------------------
BM_create_terminate_thread. 32424 ns   21216 ns            2000
```

本节中，介绍了如何使用 Google Benchmark 库创建微基准测试来测量某些函数的执行时间。同样，微基准测试只是一种近似值，并且由于基准测试代码的孤立性质，可能会产生误导，请谨慎使用。

13.2.3. Linux perf 工具

在代码中使用 std::chrono 或微基准测试库（例如 Google Benchmark）需要访问要分析的代码，并且能够通过添加额外调用来测量代码段的执行时间，或运行小片段作为微基准测试函数来修改它。

使用 Linux perf 工具，可以分析程序的执行情况，而无需更改其代码。

Linux perf 工具是一款功能强大、灵活且广泛使用的 Linux 系统性能分析和分析实用程序。提供了对内核和用户空间级别的系统性能的详细见解。

看一下 perf 的主要用途。首先，我们有 CPU 分析。perf 工具允许捕获进程的执行配置文件，测量哪些函数消耗了最多的 CPU 时间。这对于帮助识别代码中 CPU 密集型部分和瓶颈非常有用。

以下命令行将在我们编写的小型 13x07-thread_contention 程序上运行 perf，该程序用于说明该工具的基础知识。此应用程序的代码可在本书的 GitHub 库中找到：

```
perf record --call-graph dwarf ./13x07-thread_contention
```

--call-graph 选项将函数调用层次结构的数据记录在名为 perf.data 的文件中，而 dwarf 选项指示 perf 使用 dwarf 文件格式来调试符号（以获取函数名称）。

执行完上一个命令后，运行以下命令：

```
perf script > out.perf
```

这会将记录的数据（包括调用堆栈）转储到名为 out.perf 的文本文件中。

现在，需要将文本文件转换为带有调用图的图片。为此，可以运行以下命令：

```
gprof2dot -f perf out.perf -o callgraph.dot
```

这将生成一个名为 callgraph.dot 的文件，可以使用 Graphviz 进行可视化。

这时，可能需要安装 gprof2dot。为此，需要在 PC 上安装 Python。运行以下命令安装 gprof2dot：

```
pip install gprof2dot
```

也安装 Graphviz。在 Ubuntu 中，可以这样做：

```
sudo apt-get install graphviz
```

最后，通过运行以下命令即可生成 callgraph.png 图片：

```
dot -Tpng callgraph.dot -o callgraph.png
```

可视化程序调用图的另一种常见方法是使用火焰图。

要生成火焰图，请克隆 FlameGraph 库：

```
git clone https://github.com/brendangregg/FlameGraph.git
```

在 FlameGraph 文件夹中，将找到生成火焰图的脚本。

运行以下命令：

```
FlameGraph/stackcollapse-perf.pl out.perf > out.folded
```

此命令将堆栈跟踪折叠为 FlameGraph 工具可以使用的格式。现在，运行以下命令：

```
Flamegraph/flamegraph.pl out.folded > flamegraph.svg
```

可以使用网络浏览器来可视化火焰图：

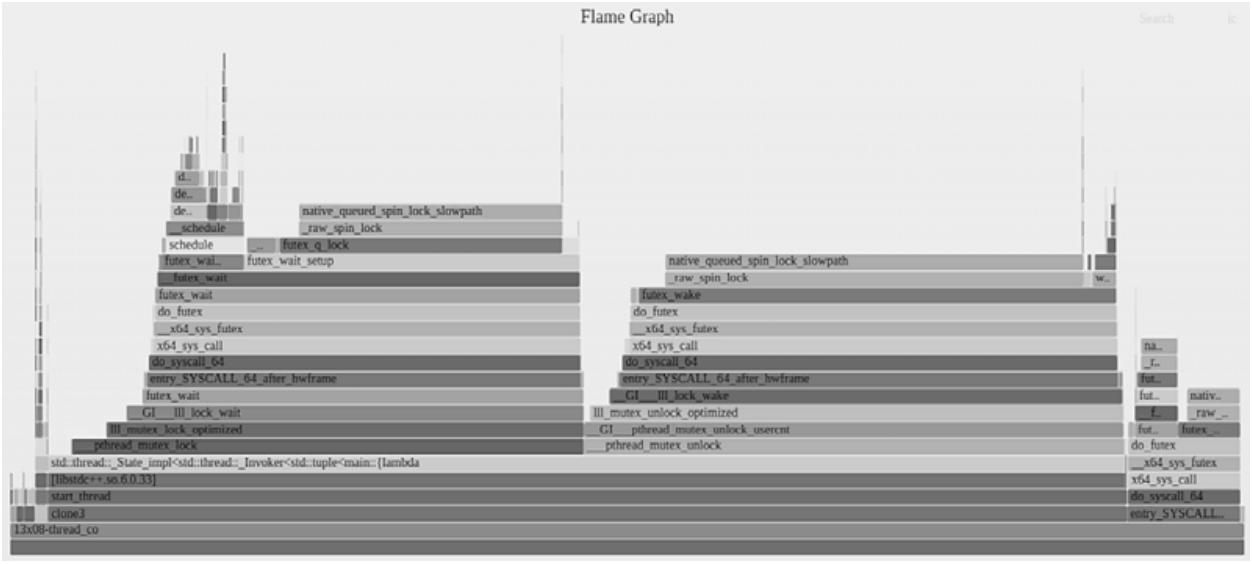

图 13.1：火焰图概览

现在，介绍如何收集程序的性能统计数据。

以下命令将显示在执行 13x05-sort_perf 期间已执行的指令数和使用的 CPU 周期数。每周期指令数是 CPU 在每个时钟周期执行的平均指令数。此指标仅在对代码的短部分进行微基准测试或测量时才有用。对于此示例，可以看到 CPU 每周期执行一条指令，这是现代 CPU 的平均值。在多线程代码中，由于执行的并行性质，可以得到更大的数字，但此指标通常用于测量和优化在单个 CPU 核心中执行的代码。该数字必须解释为保持 CPU 的繁忙程度，其取决于许多因素，例如：内存读取/写入的次数、内存访问模式（线性连续/非线性）、代码中的分支级别等：

```
perf stat -e instructions,cycles ./13x05-sort_perf
```

运行上述命令后，得到以下结果：

```
Performance counter stats for './13x05-sort_perf':
    30,993,024,309        instructions      #       1.03 insn per cycle
    30,197,863,655        cycles
    6.657835162 seconds time elapsed

    6.502372000 seconds user
    0.155008000 seconds sys
```

运行以下命令，可以获取可以使用 perf 分析的所有预定义事件的列表：

```
perf list
```

再做几个：

```
perf stat -e branches ./13x05-sort_perf
```

上述命令测量了已执行的分支指令的数量，得到以下结果：

```
Performance counter stats for './13x05-sort_perf':
    5,246,138,882      branches
     6.712285274 seconds time elapsed
     6.551799000 seconds user
     0.159970000 seconds sys
```

这里，执行的指令中有六分之一是分支指令，这是预料之中的。

如前所述，测量代码中的分支级别非常重要，尤其是对于较短的代码段（以避免可能影响测量结果的交互）。如果没有分支或分支很少，CPU 将以更快的速度运行指令。分支的主要问题是 CPU 可能需要重建管道，而这可能代价高昂，尤其是当分支位于内部/关键循环中时。

以下命令将报告 L1 缓存数据访问的次数（将在下一节中了解 CPU 缓存）：

```
perf stat -e all_data_cache_accesses ./13x05-sort_perf
```

得到以下结果：

```
Performance counter stats for './13x05-sort_perf':
    21,286,061,764      all_data_cache_accesses

       6.718844368 seconds time elapsed

       6.561416000 seconds user
       0.157009000 seconds sys
```

回到锁争用示例并使用 perf 收集一些有用的统计数据。

使用 perf 的另一个好处是 CPU 迁移 - 即线程从一个 CPU 核心移动到另一个 CPU 核心的次数。核心之间的线程迁移可能会降低缓存性能，线程在移动到新核心时会失去缓存数据的好处（下一节将详细介绍缓存）。

运行以下命令：

```
perf stat -e cpu-migrations ./13x07-thread_contention
```

这将有以下输出：

```
Performance counter stats for './13x08-thread_contention':
            45         cpu-migrations
  50.476706194     seconds time elapsed
  57.333880000 seconds user
 262.123060000 seconds sys
```

来看看使用 perf 的另一个优势：上下文切换。计算执行期间上下文切换的次数（线程被换出并调度另一个线程的次数）。高频率上下文切换可能表明有太多线程在争夺 CPU 时间，从而导致性能下降。

运行以下命令：

```
perf stat -e context-switches ./13x07-thread_contention
```

输出结果如下：

```
Performance counter stats for './13x08-thread_contention':
     13,867,866     cs
   47.618283562 seconds time elapsed

   52.931213000 seconds user
  247.033479000 seconds sys
```

本节就到此结束。这里我们介绍了 Linux perf 工具及其一些应用。下一节将研究 CPU 内存缓存和伪共享。

13.3. 伪共享

本节中，将研究多线程应用程序中的一个常见问题，称为伪共享。

多线程应用程序的理想实现是最小化不同线程之间共享的数据。理想情况下，应该只为读取访问而共享数据，在这种情况下不需要同步线程来访问共享数据，因此不需要支付运行时成本并处理死锁和活锁等问题。

现在，考虑一个简单的示例：四个线程并行运行，生成随机数并计算其总和。

每个线程独立工作，生成随机数并计算存储在自己编写的变量中的总和。这是理想的（对于这个例子来说，有点做作）应用程序，线程独立工作，没有共享数据。

以下代码是本节将要分析的示例的完整源代码：

```
#include <chrono>
#include <iostream>
#include <random>
#include <thread>
#include <vector>

struct result_data {
    unsigned long result { 0 };
};

struct alignas(64) aligned_result_data {
    unsigned long result { 0 };
};

void set_affinity(int core) {
    if (core < 0) {
        return;
    }

    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(core, &cpuset);
    if (pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t),
    &cpuset) != 0) {
        perror("pthread_setaffinity_np");
        exit(EXIT_FAILURE);
    }
}

template <typename T>
auto random_sum(T& data, const std::size_t seed, const unsigned long
iterations, const int core) {
    set_affinity(core);
    std::mt19937 gen(seed);
    std::uniform_int_distribution dist(1, 5);
```

```
36      for (unsigned long i = 0; i < iterations; ++i) {
37          data.result += dist(gen);
38      }
39  }
40
41  using namespace std::chrono;
42  void sum_random_unaligned(int num_threads, uint32_t iterations) {
43      auto* data = new(static_cast<std::align_val_t>(64)) result_data[num_threads];
44
45      auto start = high_resolution_clock::now();
46      std::vector<std::thread> threads;
47      for (std::size_t i = 0; i < num_threads; ++i) {
48          set_affinity(i);
49          threads.emplace_back(random_sum<result_data>, std::ref(data[i]), i, iterations, i);
50      }
51      for (auto& thread : threads) {
52          thread.join();
53      }
54      auto end = high_resolution_clock::now();
55      auto duration = std::chrono::duration_cast<milliseconds>(end - start);
56      std::cout << "Non-aligned data: " << duration.count() << " milliseconds" << std::endl;
57
58      operator delete[] (data, static_cast<std::align_val_t>(64));
59  }
60
61  void sum_random_aligned(int num_threads, uint32_t iterations) {
62      auto* aligned_data = new(static_cast<std::align_val_t>(64))
63      aligned_result_data[num_threads];
64      auto start = high_resolution_clock::now();
65      std::vector<std::thread> threads;
66      for (std::size_t i = 0; i < num_threads; ++i) {
67          set_affinity(i);
68          threads.emplace_back(random_sum<aligned_result_data>, std::ref(aligned_data[i]), i,
            ↪  iterations, i);
69      }
70      for (auto& thread : threads) {
71          thread.join();
72      }
73      auto end = high_resolution_clock::now();
74      auto duration = std::chrono::duration_cast<milliseconds>(end - start);
75      std::cout << "Aligned data: " << duration.count() << " milliseconds" << std::endl;
76      operator delete[] (aligned_data, static_cast<std::align_val_t>(64));
77  }
78
79  int main() {
80      constexpr unsigned long iterations{ 100000000 };
81      constexpr unsigned int num_threads = 8;
82
```

```
    sum_random_unaligned(8, iterations);
    sum_random_aligned(8, iterations);

    return 0;
}
```

如果编译并运行上面的代码，将得到类似以下内容的输出：

```
Non-aligned data: 4403 milliseconds
Aligned data: 160 milliseconds
```

该程序仅调用两个函数：sum_random_unaligned 和 sum_random_aligned。这两个函数的作用相同：创建八个线程，每个线程生成随机数并计算其总和，线程之间不共享数据。可以看到这些函数几乎相同，主要区别在于 sum_random_unaligned 使用以下数据结构来存储生成的随机数的总和：

```
struct result_data {
    unsigned long result { 0 };
};
```

sum_random_aligned 函数使用了一个稍微不同的函数：

```
struct alignas(64) aligned_result_data {
    unsigned long result { 0 };
};
```

唯一的区别是使用 alignas(64) 通知编译器数据结构实例必须对齐在 64 字节边界。

由于线程执行的是相同的任务，所以性能差异非常显著。只需将每个线程写入的变量与 64 字节边界对齐，即可大大提高性能。

要了解为什么会发生这种情况，需要考虑现代 CPU 的一个功能——内存缓存。

13.4. CPU 缓存

现代 CPU 的计算速度非常快，当想要实现最大性能时，内存访问是主要瓶颈。内存访问的合理估计时间约为 150 纳秒。在此期间，我们的 3.6 GHz CPU 经历了 540 个时钟周期。粗略估计，如果 CPU 每两个周期执行一条指令，则有 270 条指令。对于普通应用程序，内存访问是一个问题，即使编译器可能会重新排序其生成的指令，并且 CPU 也可能重新排序指令以优化内存访问并尝试运行尽可能多的指令。

因此，为了提高现代 CPU 的性能，我们拥有的 CPU 缓存或内存缓存，即芯片中用于存储数据和指令的内存。这种内存比 RAM 快得多，允许 CPU 更快地检索数据，从而显著提高整体性能。

以实际中的缓存为例，想象一下厨师需要一些食材来为餐厅客户做午餐。现在，他们只在客户来到餐厅点餐时才购买这些食材，这会非常慢。他们也可以去超市购买一整天的食材，则可以在更短的时间内为所有客户做饭，并为他们提供餐点。

CPU 缓存遵循相同的概念：当 CPU 需要访问一个变量（比如一个 4 字节整数）时，会读取 64 个字节（此大小可能不同，具体取决于 CPU，但大多数现代 CPU 都使用该大小）的连续内存，以防需要访问更多连续的数据。

从内存访问的角度来看，线性内存数据结构（如 `std::vector`）的性能会更好，因为缓存可以大大提高性能。对于其他类型的数据结构（如 `std::list`），情况并非如此。

当然，这只是为了优化缓存的使用。

如果 CPU 内置缓存如此好，为什么所有内存不都是这样呢？答案是成本。缓存非常快（比 RAM 快得多），但也非常昂贵。

现代 CPU 采用分层缓存结构，通常由三个级别组成，分别称为 L1、L2 和 L3：

- L1 缓存最小、速度最快。它也是距离 CPU 最近的缓存，也是最昂贵的缓存。通常分为两部分：用于存储指令的指令缓存和用于存储数据的数据缓存。典型大小为 64 KB，分为 32 KB 用于存储指令和 32 KB 用于存储数据。L1 缓存的通常访问时间为 1 到 3 纳秒。
- L2 缓存比 L1 缓存更大，速度也稍慢，但仍比 RAM 快得多。典型的 L2 缓存大小在 128Kb 到 512 Kb 之间（用于运行本章示例的 CPU 每个核芯有 512 Kb 的 L2 缓存）。L2 缓存的通常访问时间约为 3 到 5 纳秒。
- L3 缓存是三者中最大且最慢的。L1 和 L2 缓存是每个核芯的（每个核芯都有自己的 L1 和 L2 缓存），但 L3 由多个核芯共享。我们的 CPU 有 32 Mb 的 L3 缓存，由每组八个核芯共享。通常的访问时间约为 10 到 15 纳秒。

接下来，让我们将注意力转向与内存缓存相关的另一个重要概念。

13.4.1. 缓存一致性

CPU 不直接访问 RAM。这种访问始终通过缓存进行，并且只有当 CPU 在缓存中找不到所需数据时才会访问 RAM。在多核系统中，每个核芯都有自己的缓存，所以一块 RAM 可能同时存在于多个核芯的缓存中。这些副本需要始终保持同步；否则，计算结果可能不正确。

我们已经看到每个核芯都有自己的 L1 缓存。回到我们的例子，思考一下当使用非对齐内存运行该函数时会发生什么。

`result_data` 的每个实例都是 8 个字节。我们创建一个包含 8 个 `result_data` 实例的数组，每个线程一个。占用的总内存将为 64 个字节，所有实例在内存中都是连续的。每次线程更新随机数的总和时，都会更改存储在缓存中的值。记住，CPU 总是会一次读取和写入 64 个字节（称为缓存行 – 可以将其视为最小的内存访问单元）。所有变量都在同一个缓存行中，即使线程不共享它们（每个线程都有自己的变量 – `sum`），CPU 也不知道这一点，需要让所有核芯都看到这些变化。

这里，有 8 个核芯，每个核芯都在运行一个线程。每个核芯都将 RAM 中的 64 字节内存加载到 L1 缓存中。由于线程只读取变量，所以一切正常。但是只要一个线程修改其变量，缓存行的内容就会失效。

现在，由于缓存行在其余 7 个核芯中无效，CPU 需要将更改传播到所有核芯，即使线程不共享变量，CPU 也不可能知道，它会更新所有核芯的所有缓存行以保持值一致。这称为缓存一致性。如果线程共享变量，则不将更改传播到所有核芯是不正确的。

我们的示例中，缓存一致性协议在 CPU 内部产生大量流量，所有线程都共享变量所在的内存区域，尽管从程序的角度来看并非如此。这就是称之为“伪共享”的原因：变量共享是因为缓存和缓存一致性协议的工作方式。

当将数据对齐到 64 字节边界时，每个实例占用 64 个字节。这保证了它们位于自己的缓存行中，并且不需要缓存一致性流量，这种情况下没有数据共享。在第二种情况下，性能要好得多。

使用 perf 来确认这确实发生了。

首先，在执行 sum_random_unaligned 时运行 perf。想要查看程序访问缓存的次数以及缓存未命中的次数。每次缓存需要更新，因为包含的数据也位于另一个核芯的缓存行中，这算作一次缓存未命中：

```
perf stat -e cache-references,cache-misses ./13x07-false_sharing
```

```
Performance counter stats for './13x07-false_sharing':

     251,277,877     cache-references
     242,797,999     cache-misses
                     # 96.63% of all cache refs
```

大多数缓存引用都是缓存未命中。由于伪分享，这是预期的。

现在，如果运行 sum_random_aligned，结果则会大不相同：

```
Performance counter stats for './13x07-false_sharing':

          851,506    cache-references
          231,703    cache-misses
                     # 27.21% of all cache refs
```

缓存引用和缓存未命中的数量都少得多。这是因为无需不断更新所有核芯中的缓存来保持缓存一致性。

本节中，了解了多线程代码最常见的性能问题之一：伪共享。了解了有伪共享和没有伪共享的函数示例，以及伪共享对性能的负面影响。

下一节中，我回到第 5 章中实现的 SPSC 无锁队列并改进其性能。

13.5. SPSC 无锁队列

第 5 章中，实现了一个 SPSC 无锁队列，作为如何在不使用锁的情况下同步两个线程对数据结构的访问的示例。此队列仅由两个线程访问：一个生产者将数据推送到队列，一个消费者从队列中弹出数据。这是最容易同步的队列。

我们使用两个原子变量来表示队列的头（要读取的缓冲区索引）和尾（要写入的缓冲区索引）：

```
1 std::atomic<std::size_t> head_ { 0 };
2 std::atomic<std::size_t> tail_ { 0 };
```

为了避免伪共享，可以将代码改成如下形式：

```
alignas(64) std::atomic<std::size_t> head_ { 0 };
alignas(64) std::atomic<std::size_t> tail_ { 0 };
```

完成此更改后，可以运行我们实现的代码来测量生产者和消费者线程每秒执行的操作数（推送/弹出）。代码可以在本书的 GitHub 库中找到。

现在，运行 perf：

```
perf stat -e cache-references,cache-misses ./13x09-spsc_lock_free_queue
```

将得到以下结果：

```
101559149 ops/sec

    Performance counter stats for <./13x09-spsp_lock_free_queue>:

        532,295,487      cache-references
        219,861,054      cache-misses # 41.30% of all cache refs

        9.848523651 seconds time elapsed
```

这里可以看到队列每秒能够执行大约 1 亿次操作，缓存未命中率约为 41%。

回顾一下队列的工作原理，生产者是唯一写入 tail_ 的线程，而消费者是唯一写入 head_ 的线程。不过，两个线程都需要读取 tail_ 和 head_。我们将两个原子变量声明为 aligned(64)，这样就可以保证它们位于不同的缓存行中，并且不存在错误共享。但存在真正的共享，真正的共享也会产生缓存一致性开销。

真正的共享意味着两个线程都可以共享访问两个变量，即使每个变量仅由一个线程（且始终是同一个线程）写入。这种情况下，为了提高性能，必须减少共享，尽可能避免每个线程对两个变量进行读取访问。我们无法避免数据共享，但可以减少。

再来关注生产者（与消费者的机制相同）：

```
bool push(const T &item) {
    std::size_t tail = tail_.load(std::memory_order_relaxed);
    std::size_t next_tail = (tail + 1) & (capacity_ - 1);
    if (next_tail == cache_head_) {
        cache_head_ = head_.load(std::memory_order_acquire);
        if (next_tail == cache_head_) {
            return false;
        }
    }
    buffer_[tail] = item;
    tail_.store(next_tail, std::memory_order_release);
    return true;
}
```

push( ) 函数仅由生产者调用。
分析一下该函数的作用：

- 原子地读取环形缓冲区中存储中的最后一个索引：

```
std::size_t tail = tail_.load(std::memory_order_relaxed);
```

- 计算该项目在环形缓冲区中存储的索引：

```
std::size_t next_tail = (tail + 1) & (capacity_ - 1);
```

- 检查环形缓冲区是否已满。但没有读取 head_，而是读取缓存的 head 值：

```
if (next_tail == cache_head_) {
```

最初，cache_head_ 和 cache_tail_ 都设置为零。使用这两个变量的目的是尽量减少核芯之间的缓存更新。缓存变量技术的工作原理如下：每次调用 push（或 pop）时，都会以原子方式读取 tail_（由同一线程写入，不需要缓存更新）并生成下一个索引，将在其中存储作为参数传递给 push 函数的项目。现在，不再使用 head_ 来检查队列是否已满，而是使用 cache_head_，仅由一个线程（生产者线程）访问，从而避免任何缓存一致性开销。如果队列已“满”，则通过以原子方式加载 head_ 来更新 cache_head_。更新之后，再次检查。如果第二次检查导致队列已满，则返回 false。
使用这些局部变量（生产者使用 cache_head_，消费者使用 cache_tail_）的优点是它们减少了共享，即访问可能在不同核芯的缓存中更新的变量。当生产者推送多个在消费者尝试获取队列中的项目之前（对于消费者来说也是一样）。假设生产者在队列中插入 10 个项目，而消费者尝试获取一个项目。第一次使用缓存变量检查会告诉队列为空，但在使用实际值更新后，一切正常。消费者只需读取 cache_tail_ 变量检查队列是否为空，即可获取另外 9 个项目。

- 如果环形缓冲区已满，则更新 cache_head_：

```
head_.load(std::memory_order_acquire);
        if (next_tail == cache_head_) {
            return false;
        }
```

- 如果缓冲区已满（不仅仅是 cache_head_ 需要更新），则返回 false。生产者无法将新项目推送到队列。
- 如果缓冲区未满，则将项目添加到环形缓冲区并返回 true：

```
buffer_[tail] = item;
    tail_.store(next_tail, std::memory_order_release);
    return true;
```

我们可能减少了生产者线程访问 tail_ 的次数，从而减少了缓存一致性流量。考虑这种情

况：生产者和消费者使用队列，生产者调用 push()。当 push() 更新 cache_head_ 时，可能比 tail_ 领先一个以上位置，所以不需要读取 tail_。

同样的原则也适用于消费者和 pop()。

修改代码以减少缓存一致性流量后，再次运行 perf：

```
162493489 ops/sec

 Performance counter stats for <./13x09-spsp_lock_free_queue>:

       474,296,947     cache-references
       148,898,301     cache-misses           # 31.39% of all cache refs

       6.156437788 seconds time elapsed

      12.309295000 seconds user
       0.000999000 seconds sys
```

可以看到性能提高了约 60%，并且缓存引用和缓存未命中的数量减少了。

至此，我们了解了如何减少两个线程之间对共享数据的访问可以提高性能。

13.6. 总结

本章中，介绍了三种可用于分析代码的方法：std::chrono、使用 Google Benchmark 库进行微基准测试和 Linux perf 工具。

还了解了如何通过减少/消除伪共享和减少真实共享、减少缓存一致性开销来提高多线程程序的性能。

本章对一些分析技术进行了基本介绍，这些技术对于进一步研究非常有用。正如本章开头所说，性能是一个复杂的主题，需要专门写一本书来进行介绍。

13.7. 扩展阅读

- Fedor G. Pikus, The Art of Writing Efficient Programs, First Edition, Packt Publishing, 2021.
- Ulrich Drepper, What Every Programmer Should Know About Memory, 2007.
- Shivam Kunwar, Optimizing Multithreading Performance (https://www.youtube.com/watch?v=yN7C3SO4Uj8).