# 译序

——王高飞

本人长期从事 Linux 下 C++后台开发，主要领域包括搜索引擎、广告系统、向量检索，这些领域天然对并发和高性能有苛刻的要求。在早年，只能手工通过 C++代码调用 Posix 多线程库来打造多线程程序，跨平台可移植性是个问题，虽然也可以使用如 OpenMP，但一路走来的程序员不可能放弃对自己代码的掌控力，不太会在高性能场景使用。

C++11 发布以后，不光在语言层面做了大量改进，同时也加入了多线程的支持，本人在刷新并发编程认知的时候看到了《C++ Cocurrency In Action，Second Edition》这本书，作为一本 C++并发编程的权威著作，在各大网站看了下，第一版的翻译不是很好，吐槽很多，网上其他版本的翻译也有很多问题，大段内容忘了翻译，很多地方理解错误，完全没法匹配原书内容，为了给自己学习做个记录，也为了回馈自己学习过程中得到的各种书籍的滋养，于是就有了翻译第 2 版的冲动，然后这个译本就出现了。

这个译本最大的贡献是行文还算流畅，对内容理解应该说还算到位，同时修正了原书中大量的错误，包括图表，和代码中标注，尤其是在第 7 章最难懂的无锁编程部分。另外在部分本人认为不是很好懂的地方加了一点译注，希望这个版本能让你的学习过程更愉快、更顺畅。本译本没有翻译书后的附录，主要由于附录比较冗长，而且都是网上到处可寻的内容。

本人第一次翻译书籍，纯粹是出于热情，仗着懂一点 C++和多线程编程，对计算机系统也稍微有点了解，就搞起来了，但水平也着实有限，再者经常写代码的人都知道，天天写代码难免有 bug，同样，一个人翻译一整本书可能也难免有犯浑的时候，如有错误或不准确地方，可以在知乎 https://www.zhihu.com/people/wang-fei-63-33，留言指正，我会及时修改。翻译一本书需要大量时间和精力，如果觉得本书不错，欢迎关注、点赞！

好了，闲话少说，可以开始你的 C++并发编程之旅了！

# 第 1 章 你好，C++并发世界!

**本章主要内容**

- 什么是并发和多线程
- 应用程序为什么要用并发和多线程
- C++支持并发的一些历史
- 一个简单的 C++多线程程序长什么样

C++用户激动人心的时刻到了。在 1998 年发布最初 C++标准 13 年后，C++标准委员会给语言以及支持库做了一次重大的变革。新的 C++标准（也称作 C++11 或 C++0x）在 2011 年发布，并带来一系列的变化使得 C++编程更加简单和高效。同时，委员会承诺了一种新的“火车模型”的发布方式，每三年发布一个新的 C++标准。截止目前（译注：这里以本书的写作时间为界，当前 C++20 已经发布），已经有两个版本：2014 年的 C++14 标准和 2017 的 C++17 标准，以及若干个扩展 C++标准的技术规范（Technology Specification）。

C++11 标准中一个最重要的新特性就是对多线程的支持。这是 C++标准第一次在语言层面认可多线程程序的存在，并在库中为编写多线程程序提供了组件。这使得编写 C++多线程程序不再依赖特定平台的扩展，也为编写可移植的多线程代码提供了保证。它也恰逢程序员们为了提高应用的性能，对并发，尤其是多线程编程的关注与日俱增。C++14、C++17 标准在这个基础上为 C++多线程编程提供了进一步的支持，伴随着还有技术规范（Technology Specification）。有一个技术规范（Technology Specification）是关于并发扩展的，还有一个是关于并行的，不过后者已纳入 C++17 标准。

本书介绍了如何用 C++多线程来编写并发程序，以及 C++的语言特性和库设施如何让并发成为可能。我首先解释我是如何理解并发和多线程，以及为什么需要在程序中使用它，在快速阐述为什么可能不需要并发以后，我们将概览 C++对并发的支持，最后我们将实战一个简单的 C++并发示例来结束这一章。资深的多线程开发读者可以跳过前面的一些小节。在随后的章节中，会覆盖更多的例子来加深对库设施的理解。本书最后，将会给出 C++标准库设施中所有关于多线程和并发的深度参考。

那么，什么是并发和多线程?

## 1.1 什么是并发

最简单和最基本的并发,是指两个或更多独立的活动同时发生。并发在生活中无处不在，我们可以边走边聊或者每只手做不同的动作，我们每个人彼此独立地过活——当我在游泳的时候你可以观看足球比赛，等等。

### 1.1.1 计算机系统中的并发

当我们从计算机视角讨论并发，指的是在单个系统里并行执行多个独立的任务，而非顺序地或者一个接一个的执行。这也不是什么新鲜事：多任务操作系统允许一台桌面电脑通过任务切换的方式同时运行多个程序在很多年前就司空见惯，就好像拥有多个处理器能够实现真正并发的高端服务器那样。和以往不同的是可以真正并行地运行多任务而非以前的那种假象方式的计算机日益普及。

以前大多数台式机只有一个处理器，具有单个处理单元(processing unit)或核心(core)，如今很多台式机还是这样。这种机器在某一时刻只能执行一个任务，不过它每秒可以在任务间切换很多次。通过做一点这个任务，再做一点那个任务，循环往复的方式，让这些任务看起来是同时执行的。这叫做任务切换（task switching）。我们仍认为这样的系统是并发的：因为任务切换得太快，以至于无法区分在哪个点由于处理器切换到另一个任务而挂起任务。任务切换会给用户和程序本身造成一种并发的假象。因为只是并发的假象，所以程序在单一处理器任务切换环境下的行为和在真正并发环境下略有不同。特别地，当对内存模型(见第 5 章)做不正确的假设时，在这种环境下可能不会显现出来。我们将在第 10 章深入讨论。

多处理器计算机用于服务器和高性能计算已有多年，同时在单一芯片上有多个核的处理器(多核处理器)的台式机也越来越普遍。无论是多处理器或是多核的单处理器（或联合），这些计算机都能够真正的并行执行多个任务。我们称之为硬件并发(hardware concurrency)”。

图 1.1 显示了一台计算机恰好有两个任务要处理时的理想情况，每个任务被分为 10 个相等大小的块。在一个双核机器(具有两个处理核心)上，每个任务可以在各自的核上执行。在单核机器上做任务切换时，每个任务的块交替进行。但它们中间有一些间隙(在图 1.1 中，分割任务块的灰色条厚度大于双核机器的分隔条)；为了实现交替执行，系统每次从一个任务切换到另一个时都需要执行一次上下文切换(context switch)，这需要时间开销。为了执行上下文切换，操作系统必须为当前运行的任务保存 CPU 的状态和指令指针，然后算出要切换到哪个任务，并为即将切换到的任务重新加载处理器状态。CPU 潜在地需要将将指令和数据从内存加载到缓存中，这会阻止 CPU 执行任何指令，导致进一步的时延。

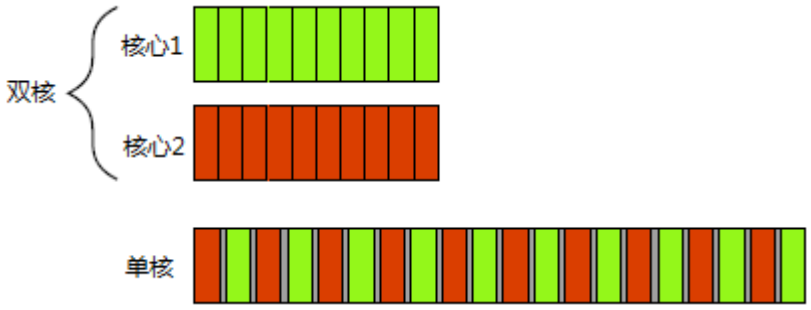


图 1.1 并发的两种方式：双核机器的并发执行 Vs. 单核机器的任务切换

尽管硬件层面的并发可用性在多处理器或多核系统上更加明显，但有些处理器可以在单核上执行多个线程。硬件线程（hardware threads）的数量是最重要的考量因素量，它用于度量多少独立的任务可以被硬件真正并发运行。即便是具备真正硬件并发的系统，也很容易拥有比硬件并发执行数量还要多的任务，所以任务切换仍然适用于这些情况。例如，在一个典型的台式计算机上可能会有数以百计的任务在运行，执行后台操作，即便计算机名义上是空闲的。正是任务切换使得这些后台任务可以运行，并使得你可以同时运行文字处理器、编译器、编辑器和 web 浏览器(或其他应用的组合)。图 1.2 显示了四个任务在双核处理器上的任务切换，仍然是将任务整齐地划分为大小相同块的理想情况。实际

上，许多问题会使得分割不均以及调度不规则。部分问题将在第 8 章中讨论，到时我们再考察影响并发代码性能的因素。

本书涉及的技术，函数，以及类，无论应用程序运行在单核处理器，还是在多核处理器上都可以使用，同时不管并发是通过任务切换还是真正的硬件并发实现的都不受影响。但是正如你所想，如何在程序中使用并发，将很大程度上取决于可用的硬件并发数量。第 8 章将讨论这个议题，同时将讨论了设计 C++并发代码涉及的问题。

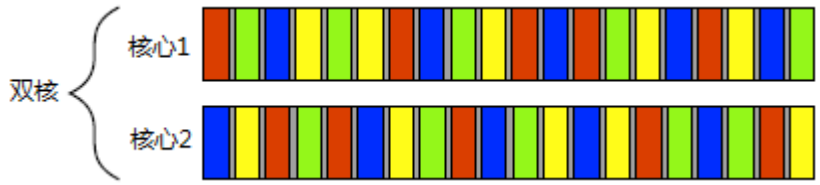


图 1.2 四个任务在两个核之间切换

## 1.1.2 并发的方法

试想在某刻，一对程序员在一起做一个软件项目。如果开发者在独立的办公室，他们可以安静地工作、不会互相干扰，并且他们手里各自有自己的参考手册。但是交流不太方便，比起转过身来互相交谈，他们必须使用电话或者电子邮件或起身到对方的办公室。并且，管理两个办公室和购买多份参考手册需要开支。

现在假设让开发人员集中到一间办公室，他们可以自由的交谈从而对某个应用程序的设计进行讨论，并且可以很轻松地在纸或白板上绘制图表，辅以设计思路或说明以提供帮助。现在只需要管理一个办公室，只要一套资源就够了。遗憾的是，他们可能很难集中注意力，并且还可能存在资源共享的问题（“参考手册哪去了?”）

这两种组织开发人员的方法阐述了并发的两种基本方法。每个开发人员代表一个线程，每个办公室代表一个进程。第一种方法是使用多个单线程的进程，类似于让每个开发人员在自己的办公室，而第二种方法是一个单一进程有多个线程，好比一个办公室里有两个开发人员。你可以以任意的风格来组合它们，对多个进程，有些是多线程的有些是单线程的，但原则是相同的。现在让我们看一下这两种方法在一个程序中是怎么并发的。

**多进程并发**

使用并发的第一种方法，是将应用程序拆分为多个独立的单线程的进程，它们在同一时刻运行，就像同时运行网页浏览器和文字处理器一样。这些独立的进程可以通过常规的进程间通信（信号，套接字，文件，管道等等）通道给彼此传递消息，如图 1.3 所示。不过，这种进程之间的通信通常不是建立起来复杂，就是速度慢，或者兼而有之，这是因为操作系统在进程间提供了许多的保护，以避免一个进程意外修改属于另一个进程的数据。另外还有一个缺点是，运行多个进程有固有的开销：启动进程需要时间，操作系统需要内部资源来管理进程，等等。

多进程并发也并非一无是处：操作系统在进程间附加的保护和更高级别的通信机制，意味着可以更容易编写安全的并发代码。实际上，在类似于 Erlang(www.erlang.org/)编程语言提供的环境，将进程作为并发的基本构造块效果显著。

使用多进程实现并发还有一个额外的优势——你可以在不同的机器上运行独立的进程，通过网络连接起来。虽然这增加了通信成本，但在精心设计的系统上，这可能是一个提高并行可用行和性能的富有成效的方法。

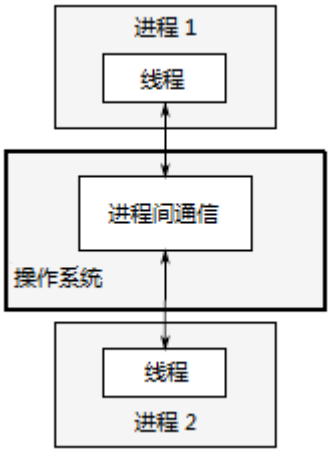


图 1.3 一对并发运行的进程之间的通信

**多线程并发**

并发的另一个方法是在单个进程中运行多个线程。线程很像轻量级的进程：每个线程相互独立，且每个线程可以运行不同的指令序列。但是，进程中的所有线程都共享相同的地址空间，并且大部分数据可以被所有线程直接访问——全局变量仍然是全局的，指针、对象的引用或数据可以在线程之间传递。尽管在进程之间共享内存通常是可能的，但是这种机制比较复杂并且难于管理。因为同一数据的内存地址在不同的进程中可能不相同。图 1.4 展示了一个进程中的两个线程通过共享内存进行通信。

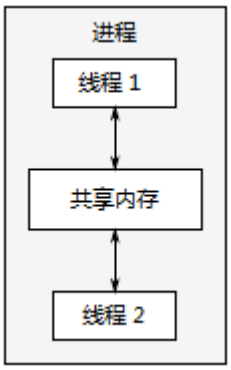


图 1.4 同一进程中一对并发运行线程之间的通信

共享的地址空间，以及缺少线程间数据的保护，使得多线程的开销比多进程开销要小，因为操作系统的簿记工作量减小了。不过，共享内存的灵活性是有代价的：如果数据要被多个线程访问，那么程序员必须确保不管什么时候访问，每个线程所看到的数据视图是一致的。围绕线程间共享数据的问题以及使用工具和遵循指南来避免问题本书都有涉及，尤其在本书第 3、4、5 和 8 章。这些问题也并非不可逾越，只要在编写代码时适当地注意即可，但这也意味着线程间通信需要深思熟虑。

单一进程中的多线程间的启动和通信的开销比多个单线程进程的要小，若不考虑共享内存可能带来的问题，多线程是主流语言包括 C++更青睐的并发方法。此外，C++标准并未对进程间通信提供任何内在支持，所以使用多进程的方式，需要依赖与平台相关的 API。因此本书只关注使用多线程的并发，并且之后提到的并发均假设使用多线程来实现。

在多线程代码中，还有一个词被广泛使用：并行。让我们澄清下区别。

### 1.1.3 并发与并行

对于多线程代码来说，这两个概念有很大部分是重叠的。实际上，很多时候它们的意思没有什么区别。其区别主要在于细微之处，关注点和意图方面。这两个术语都是关于使用硬件同时运行多个任务，不过并行更加面向性能。提到并行，主要关注使用可用硬件来提高批量数据处理的性能，然而谈到并发，关注的重点在于关注点分离（separation of concerns）或响应能力。这种区分并非一刀切，而且在含义上还是有很大的重叠，但了解这个区别有助于澄清讨论。本书中，这两方面的例子都有。

澄清并发和并行后，让来看看为什么要使用并发

## 1.2 为什么使用并发？

主要原因有两个：关注点分离(separation of concerns)和性能。事实上，它们几乎是使用并发的唯一原因；如果你观察得足够仔细，所有其他原因都可以归结为其中的一个或者另一个(或者两者都有)（当然，除了像“我乐意”这样的原因)。

### 1.2.1 关注点分离

编写软件时分离关注点几乎总是个好主意。通过将相关的代码聚合同时将无关的代码分离，可以使程序更易于理解和测试，从而更不容易出 bug。你可以使用并发来分离不同的功能区域，即使在这些不同区域上的操作需要同时发生；如果不显式地使用并发，那要么编写一个任务切换框架，要么在操作中主动地调用一段无关区域的代码。

考虑一个有用户界面的处理密集型应用，比如台式机上的 DVD 播放程序。这样的应用程序，应具备两种基本功能：不光要从磁盘中读出数据，解码图像和声音，然后把它们及时地发送到图形和声音硬件，从而实现 DVD 的无误播放，它还必须接受来自用户的输入，当用户点击“暂停”或“返回菜单”或“退出”按键的时候必须有所反应。当应用程序是

单个线程时，应用需要在播放期间定期检查用户的输入，这需要把用户界面代码合并到DVD播放代码。如果使用多线程来分隔这些关注点，用户界面代码和DVD播放代码就不必交错在一起：一个线程可以处理用户界面，另一个进行DVD播放。它们之间会有交互，比如当用户点击暂停键时，但现在这些交互和手边的任务直接相关。

这会带来响应上的错觉，因为用户界面线程通常可以立即响应用户的请求，尽管当请求被传递给工作中的线程时，响应可能只是显示一个“忙”光标或一条“请等待”的消息。类似地，独立的线程通常用来执行那些必须在后台持续运行的任务，例如：桌面搜索程序中监视文件系统变化的线程。以这种方式使用线程通常会使每个线程的逻辑更简单，因为它们之间的交互都限制在清晰可辨的点，而不是散落在各个任务的逻辑中。

这种情况下，线程数不再依赖可用的CPU核数，因为对线程的划分是基于概念上的设计，而不是尝试增加吞吐量。

## 1.2.2 性能：任务和数据的并行

多处理器系统已经存在了几十年，但直到最近，它们也只在超级计算机、大型机和大型服务器系统中才能看到。然而，芯片制造商越来越倾向于多核设计，即在单个芯片上集成2、4、16或更多的处理器，从而获取更好的性能。因此，多核台式机，甚至多核嵌入式设备，现在越来越普遍。它们算力的提高不是因为单一任务运行的更快，而是并行地运行多个任务。在过去，程序员坐等他们的程序随着处理器的更新换代而变得更快。但是现在，正如Herb Sutter所说的，“没有免费的午餐了。”[1] 如果软件想要利用日益增长的计算能力，那就必须设计成并发地运行多任务。因此程序员必须注意，尤其是那些迄今都忽略并发的程序员，现在很有必要将其加入工具箱中。

有两种利用并发来提高性能的方式：第一种也是最明显的，将单个任务分成几部分，且各自并行运行，从而降低总运行时间。这就是任务并行（task parallelism）。虽然这听起来很直观，但它处理起来相当复杂，因为在各个部分之间可能存在大量的依赖。拆分要么是根据处理来进行——一个线程执行算法的一部分，而另一个线程执行算法的另一个部分——要么是基于数据来进行——每个线程在不同的数据部分上执行相同的操作。后一种方法被称为数据并行（data parallelism）。

容易被并行影响的算法常被称为是“易并行”(embarrassingly parallel，这个术语字面难以理解，意指任务之间没有依赖或者依赖很少，可以完美并行)。尽管这意味着你可能会对代码能如此简单的并行而感到尴尬，但这是好事。我遇到过的其他术语，还有“自然并行”(naturally parallel)和“方便并发”(conveniently concurrent)。易并行算法具有良好的可扩展特性——当可用硬件线程的数量增加时，算法的并行性也会配套增加。这种算法完美体现了“人多力量大”。如果算法中有不易并行的部分，你可以把算法划分成固定(因而不可扩展)数量的并行任务。在线程之间划分任务的技术将在第8章和第10章讨论。

用并发提升性能的第二种方式是利用可用的并行来解决更大的问题：与其每次只处理一个文件，不如酌情处理2个、10个或20个。虽然这是数据并行的一种应用，即在多数据集上同时执行相同的操作，但侧重点不同。处理一个数据块仍然需要同样的时间，但在

相同的时间内处理了更多的数据。显然，这种方法也有限制，且并非在所有情况下都能受益，但这种方法所带来的吞吐量提升，可以使某些事情成为可能——如果图片的不同区域能被并行地处理，就能以更高的分辨率来处理视频。

### 1.2.3 何时不使用并发

知道何时不使用并发与知道何时使用它一样重要。基本上，不使用并发的唯一原因就是收益比不上成本。使用并发的代码在很多情况下较难理解，因此编写和维护多线程代码会直接产生脑力成本，而额外的复杂性也可能会引发更多的 bug。除非潜在的性能增益足够大或关注点分离的足够清晰，能抵消为确保代码逻辑正确所需的额外开发时间以及维护多线程代码相关的额外成本；否则，不要用并发。

同样地，性能增益可能会小于预期；启动线程时存在固定开销，因为操作系统需要分配相关的内核资源和堆栈空间，然后把新线程添加给调度器，这都需要时间。如果在线程上的任务完成得很快，那么实际执行任务的时间要比启动线程的时间更短，这就会导致应用程序的整体性能还不如直接使用主线程（spawning thread）直接执行。

此外，线程是有限的资源。如果太多的线程同时运行，则会消耗很多操作系统资源，从而使得操作系统整体上运行得更慢。不仅如此，运行太多的线程也会耗光进程的可用内存或地址空间，因为每个线程都需要一个独立的堆栈空间。对于一个可用地址空间为 4GB 的 32 位处理器来说，这的确是个问题：如果每个线程都有一个 1MB 的堆栈(很多系统这么干)，那么 4096 个线程将会用尽所有地址空间，不会给代码、静态数据或者堆数据留有任何空间。即便 64 位(或者更大)的系统不存在这种直接的地址空间限制，但资源总是有限的：如果你运行了太多的线程，最终也是出会问题的。尽管线程池(参见第 9 章)可以用来限制线程的数量，但也不是万能的，它们也有自己的问题。

当客户端/服务器(C/S)应用在服务器端为每一个连接启动一个独立的线程，对于少量的连接是可以正常工作的，但当同样的技术用于需要处理大量连接的高需求服务器时，也会因为线程太多而耗光系统资源。在这种场景下，小心使用线程池可以优化性能（参见第 9 章）。

最后，运行越多的线程，操作系统就需要越多的上下文切换，每次上下文切换都需要耗费本可以花在有用工作上的时间。所以在某些时候，增加一个额外的线程实际上会降低，而非提高应用程序的整体性能。为此，如果你尝试获得系统的最佳性能，调整运行线程的数量需要考虑可用的硬件并发。

为了提升性能而使用并发就像所有其他优化策略一样：它拥有大幅度提高应用性能的潜力，但它也可能让代码更加复杂，更难理解，并且更容易出 bug。因此，应用中只有性能悠关并且有潜在重大收益的部分，才值得进行并发化。当然，如果潜在性能收益仅次于设计清晰或关注点分离，可能也值得使用多线程。

假定你已经决定想在应用程序中使用并发，无论是为了性能、关注点分离，亦或是因为一时兴起想要研究多线程(multithreading Monday)，对于 C++程序员来说，这意味着什么呢？

[1] “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software,” Herb Sutter, Dr. Dobb’s Journal, 30(3), March 2005. http://www.gotw.ca/publications/concurrency-ddj.htm.

# 1.3 C++中的并发和多线程

通过多线程为并发提供标准化支持，对C++来讲是新事物。只有C++11标准之后，才能编写不依赖平台扩展的多线程代码。为了理解标准C++线程库（Standard C++ Thread Library）众多决策背后的基本原理，首先要了解其发展的历史。

## 1.3.1 C++多线程历史

C++98(1998)标准不承认线程的存在，并且各种语言要素的操作效果，都是根据顺序执行的抽象机器（sequential abstract machine）编写的。不仅如此，内存模型也没有正式定义，所以在C++98标准下，在缺少编译器相关扩展的情况下，没办法编写多线程应用程序。

编译器供应商可以自由地向语言添加扩展，随着多线程C API的普及——比如POSIX C标准和Microsoft Windows API中的那些——这就使得很多C++编译器供应商通过各种平台相关扩展来支持多线程。这种编译器支持通常受限于只能使用平台相关的C语言API，并且确保C++运行库(例如,异常处理机制的代码)能在多线程情况下正常工作。尽管鲜有编译器供应商提供正式的多线程感知内存模型，但编译器和处理器表现的足够好，以致于大量的多线程C++程序被开发出来。

由于不满足于使用平台相关的C语言API来处理多线程，C++程序员们希望类库能提供面向对象的多线程设施。像MFC这样的应用框架，以及Boost和ACE这样的已积累了各种类的通用C++库，它们封装了底层平台相关的API，提供简化任务的高级多线程设施。尽管各种类和库在细节方面差异很大，尤其在启动新线程方面，但类的总体轮廓却大同小异。一个为C++类库共有的重要设计，同时也为程序员提供很大好处的设计，也就是使用锁的*获取资源即初始化*(RAII, Resource Acquisition Is Initialization)的习惯以确保当退出相关作用域时解锁互斥锁（mutexes）。

大多数情况下，已有的C++编译器为多线程支持组合了平台特定的API以及平台无关的类库，例如：Boost和ACE，为编写多线程C++代码提供了稳固的基础。正因为如此，多线程应用程序中有数以百万计的C++代码。不过，由于缺乏统一标准的支持，意味着有些场合由于缺乏线程感知内存模型会导致一些问题，尤其在利用处理器硬件知识获取更高性能或者编写跨平台代码而编译器的行为又随着不同平台不断变化的时候。

### 1.3.2 C++11 标准对并发的支持

所有的这些随着 C++11 标准的发布而改变，新标准中不仅有线程感知内存模型，C++标准库也扩展了：包含了用于管理线程(参见第 2 章)、保护共享数据(参见第 3 章)、线程间同步操作(参见第 4 章)，以及底层原子操作(参见第 5 章)的各种类。

C++11 线程库很大程度上，是基于上文提到的 C++类库的经验积累。特别是，Boost 线程库作为新类库的主要模型，很多类与 Boost 库中的相关类有着相同名称和结构。随着 C++标准的进化，两者之间变成了一个双向流动，Boost 线程库也配合着 C++标准在许多方面做出改变，因此 Boost 用户在使用标准线程库的时候一点都不陌生。

如本章起始提到的那样，并发支持是 C++标准的变化之一，语言本身也做了很多增强，主要是为了让程序员们的工作更加轻松。尽管这些内容不在本书的论述范围，但是其中的一些变化对于线程库本身及其使用方式产生了直接的影响。附录 A 会对这些语言特性做一个简要介绍。

### 1.3.3 C++14 和 C++17 对并发和并行的更多支持

C++14 中只为并发和并行添加了一个新的互斥锁类型，用于保护共享数据(参见第 3 章)。不过 C++17 加入了相当多内容：一开始就添加了一整套的并行算法(参见第 10 章)。两个标准对语言核心和标准库进行了增强，这也就让我们编写多线程代码更加的简单。

之前我们还提到了一个并发技术规范，它描述了 C++标准对于函数和类的扩展，尤其是围绕线程同步方面(参见第 4 章)。

C++新标准直接支持原子操作，允许程序员通过定义好的语义编写高效的代码，从而无需了解与平台相关的汇编语言。这对于试图编写高效、可移植代码的程序员们来说是个福音；不仅让编译器负责平台细节，还可以让优化器考虑操作的语义，从而让程序整体得到更好的优化。

### 1.3.4 C++线程库的效率

高性能计算的开发者通常关注 C++的性能，毕竟 C++类包装了底层设施，这种情况在新的标准 C++线程库更甚。如果你追求最好的性能，相比直接使用底层设施，了解高层设施的实现开销就很重要。这个开销就是*抽象惩罚*(abstraction penalty)。

C++标准委员会在设计标准库时，特别是标准线程库的时候，就已经注意到了这点；一个设计目标是在实现相同功能的前提下，直接使用底层 API 并不会带来多少收益。因此，类库在大部分主流平台上都比较高效(带有非常低的抽象惩罚)。

C++标准委员会的另一个目标是为了达到终极性能，需要确保 C++能给那些期望与硬件打交道的程序员，提供足够多的的底层设施。为了这个目的，伴随着新的内存模型，形成

了一个全面的原子操作库，可用于直接控制单个位、字节、内部线程间同步，以及对所有变化的可见性。这些原子类型和相应的操作现在可以在很多地方使用，而这些地方以前开发者可能使用的是平台相关的汇编语言。使用了新标准的代码会具有更好的可移植性，并且更易于维护。

C++标准库也提供了更高级别的抽象和设施，使得编写多线程代码更加简单，并且不易出错。有时运用这些设施确实会带来性能开销，因为有额外的代码需要执行。但是，这种性能成本并不一定意味着更高的抽象惩罚；通常这种性能开销并不比手工编写等效功能来的高，而且无论如何编译器可能会内联掉大部分额外代码。

某些情况下，高层设施会提供一些额外的功能，超出特定用途的需求。大部分情况下这都不是问题，因为你没有为你不使用的那部分买单。在一些罕见的场合，这些未使用的功能会影响其他代码的性能。如果你很看重程序的性能，并且高级设施带来的开销过高，你最好是通过较低级别的设施来实现你需要的功能。绝大多数情况下，额外增加的复杂性和出错几率都远大于性能的小幅提升带来的收益。即便分析表明瓶颈在 C++标准库的设施中，那也可能是由于低劣的应用设计，而非类库实现。例如，如果过多的线程竞争一个互斥锁，将会很明显的影响性能。与其在互斥锁的操作上较劲，不如重新设计应用，从而减少互斥锁上的竞争。如何减少应用中的竞争在第 8 章讨论。

罕见的情况下，如果 C++标准库没有提供所需的性能或行为时，就需要使用平台相关的设施。

### 1.3.5 平台相关的设施

虽然 C++线程库为多线程和并发处理提供了较全面的设施，但在任何给定的平台上平台相关的设施比标准库提供的更丰富。为了在不放弃使用标准 C++线程库好处的情况下轻松访问这些设施，在 C++线程库中的类型可能提供一个 `native_handle()`成员函数，允许通过使用平台相关 API 直接操作底层实现。本质上，任何使用 native_handle()执行的操作都是完全依赖于平台的，这超出了本书(包括标准 C++标准库本身)的范围。

考虑使用平台相关的设施之前，了解标准库提供了什么是很重要的，让我们从一个示例开始。

## 1.4 入门

好了，你有一个很赞的支持 C++11/C++14/C++17 标准的编译器。接下来呢？一个 C++多线程程序是什么样？其实，它看上去和其他 C++程序差不多，通常是变量、类以及函数的组合。唯一真正的区别是某些函数可以并发运行，所以需要确保共享数据在并发访问时是安全的，如第 3 章所述。为了并发地运行函数，需要使用特定的函数和对象来管理不同的线程。

### 1.4.1 你好，并发世界

让我们从一个经典的例子开始：一个打印“Hello World.”的程序。一个简单的在单线程中运行的 Hello World 程序如下所示，当我们转向多线程时，它可以作为一个基线。

```
#include <iostream>

int main()
{
  std::cout << "Hello World\n";
}
```

这个程序所做的就是将“Hello World”写入标准输出流。让我们将它与下面清单所示的简单的“Hello, Concurrent World”程序做个比较，它启动了一个独立的线程来显示这个消息。

```
#include <iostream>
#include <thread>  // 1

void hello()  // 2
{
  std::cout << "Hello Concurrent World\n";
}

int main()
{
  std::thread t(hello);  // 3
  t.join();  // 4
}
```

第一个区别是额外的`#include <thread>`，标准 C++库中对多线程支持的声明在新的头文件中：管理线程的函数和类在`<thread>`中声明，而保护共享数据的函数和类在其他头文件中声明。

其次，写消息的代码被移动到了一个独立的函数中。因为每个线程都必须具有一个初始函数(initial function)，新线程的执行从这个函数开始。对于应用程序来说，初始线程是 main()，但是对于其他线程，可以在 `std::thread` 对象的构造函数中指定——本例中，被命名为 t 的 `std::thread` 对象使用新函数 hello()作为其初始函数。

下一个区别：与直接写入标准输出或是从 main()调用 hello()不同，该程序启动了一个新的线程来实现，使线程数量增加到两个——初始线程始于 main()，而新线程始于 hello()。

新的线程启动后，初始线程继续执行。如果它不等待新线程结束，它就将继续运行到main()的结尾，从而结束程序——这有可能发生在新线程运行之前。这就是为什么在这里调用 `join()`的原因——详见第 2 章，这会使得调用线程(在 main()中)等待与 `std::thread` 对象相关联的线程，即这个例子中的 t。

这看起来仅仅为了将一条消息写入标准输出就需要做大量的工作，确实是——正如 1.2.3 节所描述的，一般来说并不值得为了如此简单的任务而使用多线程，尤其是在这期间初始线程无所事事。本书后面将通过实例来展示在哪些情景下使用多线程有明显好处。

# 总结

在本章中，介绍了并发与多线程的含义，以及在你的应用程序中为什么你会选择使用(或不用)它。还介绍了多线程在 C++中的发展历程，从 1998 标准中完全缺乏支持，经历了各种平台相关的扩展，再到 C++11/C++14/C++17 标准和并发技术规范对多线程逐渐支持。这个支持来的正是时候，它使得程序员们可以利用新的 CPU，带来更大的硬件并发。因为芯片制造商不是以增加单核的执行速度，而是选择以多核形式来增加处理能力，使得更多任务可以并发执行。

在 1.4 节中，展示了使用 C++标准库中的类和函数有多简单。在 C++中使用多线程并不复杂，复杂之处在于设计代码使其按预期的方式运行。

了解了 1.4 节的示例后，是时候看看更多实质性的内容了。在第 2 章中，我们将了解一下用于管理线程的类和函数。

# 第 2 章 管理线程

**本章主要内容**

- 启动线程以及各种指定代码让新线程运行的方法
- 等待线程结束对比让它自主运行（连接对比分离）
- 唯一标识线程

好了，看来你已经决定在程序中使用并发，特别是多线程。现在怎么做？怎么启动线程，检查它们是否已经结束，以及如何密切关注它们？C++标准库让大多数线程管理任务变得相对容易，正如你将看到的，所有的管理都是通过和给定线程关联的 std::thread 对象完成。对那些不是那么简单的任务，库提供了灵活性让你从最基本的构建块构建你所需要的东西。

本章将从基础知识开始：启动一个线程，等待它结束，或把它放在后台运行。然后再看看怎么传递额外的参数给启动的线程函数，以及怎么将一个线程的所有权从当前 std::thread 对象转移给另一个。最后，我们将看看怎么选择线程数量，以及标识特定线程。

## 2.1 线程管理基础

每个 C++程序至少有一个线程：由 C++运行时库启动的执行 main()函数的线程。然后你的程序可以以不同的函数作为入口点启动其他线程。这些线程彼此并发的运行。如同 main()函数执行完程序会退出一样，当指定的入口函数返回后，线程也会退出。正如你将看到的，在为一个线程创建了一个 std::thread 对象后，你可以等待这个线程结束；不过，首先你必须启动它，下面看下如何启动线程。

### 2.1.1 启动线程

在第 1 章中已经看到，线程在构造 std::thread 对象时启动，这个对象指定了要运行的任务。最简单的情况下，任务也平淡无奇，通常是返回 void 的无参数函数。这个函数在其所属线程上运行，直到它返回，线程也就结束了。在另一个极端，任务可以是个带有参数的函数对象，在运行的时候，它执行一系列通过消息系统指定的独立的操作，只有通过消息系统明确通知它结束后线程才会停止；线程要做什么，以及在哪启动，都无关紧要，但是使用 C++标准库启动线程，总可以归结为构造 std::thread 对象：

```
void do_some_work();
std::thread my_thread(do_some_work);
```

这是最简形式。当然，为了让编译器看到 std::thread 类的定义，你必须包含<thread>头文件。如同大多数 C++标准库的类一样，std::thread 接受任何可调用类型，所以你可以传递一个重载了调用操作符的类实例给 std::thread 构造函数：

```
class background_task
{
public:
  void operator()() const
  {
    do_something();
    do_something_else();
  }
};

background_task f;
std::thread my_thread(f);
```

在这里，提供的函数对象会拷贝到新创建运行线程的存储当中，调用也在其中进行。因此拷贝后函数对象行为应该和原始对象的相同，否则得到的结果可能不是我们期望的。

有件事需要注意，当把函数对象传入到线程构造函数中时，需要避免“最令人头痛的语法解析”(译注：C++’s most vexing parse,Scott Meyers 在《Effective STL》中创造的术语)。如果你传递了一个临时变量，而非命名变量，编译器会将其解析为函数声明，而不是对象的定义。例如：

```
std::thread my_thread(background_task());
```

这里相当与声明了一个名为 my_thread 的函数，这个函数带有一个参数(这个参数是个函数指针，指向的函数没有参数并返回 background_task 对象)，返回一个 std::thread 对象的函数，而非启动了一个新线程。在前面命名函数对象的基础上，使用额外的括号，或使用新的统一初始化语法，可以避免这个问题。如下所示：

```
std::thread my_thread((background_task()));  // 1
std::thread my_thread{background_task()};    // 2
```

第一个例子中，额外的括号防止解释成函数声明，并允许 my_thread 声明为 std::thread 类型的变量。第二个例子使用新的花括号统一初始化语法，而非圆括号，因而也声明了一个变量。

一种可以避免这个问题的可调用对象类型是 lambda 表达式。lambda 表达式是 C++11 的一个新特性，它允许使用一个可以捕获局部变量的局部函数，这样可以避免传递额外的参数(参见 2.2 节)。想要了解 lambda 表达式的全部细节，可以阅读附录 A 的 A.5 节。之前的例子可以改写为如下 lambda 表达式的类型：

```
std::thread my_thread([]{
  do_something();
  do_something_else();
});
```

一旦启动了线程，你需要明确是要等待线程结束(通过连接它——参见 2.1.2 节)，还是让它自主运行(通过分离它——参见 2.1.3 节)。如果 std::thread 对象销毁之前还没有做出决定，程序就会终止(std::thread 的析构函数会调用 std::terminate())。因此有必要确保线程即使出现异常的情况下也能正确的连接或者分离。2.1.3 节中的一种技术可以处理这种场景。注意，你只需要在 std::thread 对象销毁之前做出决定——早在你连接或者分离它之前，线程本身可能已经结束了，如果你分离它，那么如果线程还在运行，它将继续运行，并且可能在 std::thread 对象销毁很久以后还在继续运行；它只有在线程函数返回后才停止运行。

如果不等待线程结束，你需要确保线程访问的数据直到线程结束之前都是有效的。这不是一个新问题——就算在单线程代码中，访问一个已经销毁的对象，也是未定义的行为——不过，线程的使用增加了这类生命周期问题发生的几率。

一种可能碰到这种问题的情形是当线程函数持有一个指向局部变量的指针或者引用，并且函数退出后线程还没结束。下面的清单中就展示了这样的一个场景。

清单 2.1　函数已经返回，线程仍然访问局部变量

```
struct func
{
  int& i;
  func(int& i_) : i(i_) {}
  void operator() ()
  {
    for (unsigned j=0 ; j<1000000; ++j)
    {
      do_something(i);           // 潜在访问悬垂引用
    }
  }
};

void oops()
{
  int some_local_state=0;
  func my_func(some_local_state);
  std::thread my_thread(my_func);
  my_thread.detach();          // 不等待线程结束
}                              // 新线程可能还在运行
```

这里，my_thread 关联的线程在 oops 退出后仍在运行，因为你明确通过调用 detach()不等待它结束。如果线程还在运行，你就进入到表 2.1 中展示的场景：接下来调用 do_something(i)会访问已经销毁的变量。这和单线程程序很像——让函数退出后继续持有局部变量的指针或引用永远是个坏主意——但是多线程代码更容易犯这种错误，因为当它发生的时候并不一定立刻显现出来。

表2.1 分离线程在局部变量销毁后，仍对该变量进行访问

| 主线程 | 新线程 |
| --- | --- |
| 使用some_local_state的引用构造my_func | |
| 开启新线程my_thread | |
| | 启动 |
| | 调用func::operator() |
| 将my_thread分离 | 运行func::operator();可能带着some_local_state的引用调用do_something |
| 销毁some_local_state | 持续运行 |
| 退出oops函数 | 持续执行func::operator()；可能带着some_local_state的引用调用do_something --> 导致未定义行为 |

处理这种情况的通用做法是让线程函数自包含（译注：也就是函数本身什么都有，不需要依赖外部数据），将数据拷贝到线程中，而非共享数据。如果使用一个可调用的对象作为线程函数，这个对象就会拷贝到线程中，因此原始对象可以被立即销毁。但对于对象中包含的指针和引用还需谨慎，例如清单 2.1 所示。特别地，在一个函数内创建一个线程，并且需要访问函数的局部变量是个坏主意，除非保证线程会在函数退出前结束。

此外，可以通过连接线程来确保线程在函数退出前完成。

## 2.1.2 等待线程完成

如果需要等待线程完成，可以调用 std::thread 实例的 join()函数。清单 2.1 中，将 my_thread.detach()替换为 my_thread.join()，就可以确保局部变量在线程完成后，才被销毁。在这里，因为原始线程在其生命周期中并没有做什么事，使得用一个独立的线程去执行函数变得收益甚微，但在实际编程中，原始线程要么有自己的工作要做；要么会启动多个线程做一些有用的工作，并等待这些线程结束。

join()是简单粗暴的技术——要么等待线程完成，要么不等待。当你需要更细粒度的控制等待线程的过程，比如，检查某个线程是否结束，或者只等待一段时间。想要做到这些，你需要使用其他机制来完成，比如条件变量和期待(futures)，相关的讨论将会在第 4 章进行。调用 join()的动作，还清理了线程相关的存储，这样 std::thread 对象将不再与已经结束的线程有任何关联。这意味着，对一个线程只能调用一次 join()；一旦已经调用过 join()，std::thread 对象就不能再次连接，同时 joinable()将返回 false。

## 2.1.3 异常场景下的等待

如前所述，需要确保销毁 std::thread 对象之前调用 join()或 detach()。如果要分离一个线程，通常可以在线程启动后，立刻调用 detach()进行分离，因此这不算啥事。但是

如果打算等待线程，则需要细心挑选调用 join()代码的位置。这意味着如果在线程启动后调用 join()前有异常抛出，join()调用很容易被跳过。

为了避免应用因为异常抛出而终止，就需要作出一个决定。通常，当倾向于在无异常的情况下使用 join()时，你也需要在异常处理过程中调用 join()，从而避免意外的生命周期的问题。下面的程序清单是一个例子。

清单 2.2　等待线程完成

```
struct func; // 定义在清单 2.1 中
void f()
{
  int some_local_state=0;
  func my_func(some_local_state);
  std::thread t(my_func);
  try
  {
    do_something_in_current_thread();
  }
  catch(...)
  {
    t.join();  // 1
    throw;
  }
  t.join();  // 2
}
```

清单 2.2 中的代码使用了 try/catch 块确保访问本地状态的线程结束后，函数才退出，不管函数是正常退出还是因为异常。try/catch 块比较冗长，而且容易把作用域搞乱，因此这不是个好的场景。如果确保线程在函数退出之前结束很重要——不管是否因为线程函数使用了局部变量的引用，或者其他原因——那么确保覆盖所有可能的退出路径至关重要，无论是正常路径还是异常路径，因此提供一个精简的机制，来做解决这个问题是可取的。

一种方式是使用标准的“资源获取即初始化”(RAII，Resource Acquisition Is Initialization)，并且提供一个类，在析构函数中执行 join()，如同下面清单中的代码。看它如何简化 f()函数。

清单 2.3 使用 RAII 等待线程完成

```
class thread_guard
{
  std::thread& t;
public:
  explicit thread_guard(std::thread& t_):
```

```
    t(t_)
  {}
  ~thread_guard()
  {
    if(t.joinable())
    {
      t.join();
    }
  }
  thread_guard(thread_guard const&)=delete;
  thread_guard& operator=(thread_guard const&)=delete;
};

struct func; // 定义在清单 2.1 中

void f()
{
  int some_local_state=0;
  func my_func(some_local_state);
  std::thread t(my_func);
  thread_guard g(t);
  do_something_in_current_thread();
}
```

当线程执行到 f 的尾部时，局部对象就要按照构造的逆序销毁。因此，thread_guard 对象 g 是第一个被销毁的，这时线程在析构函数中被连接。即使 do_something_in_current_thread 抛出一个异常，这个销毁依旧会发生。

清单 2.3 中，thread_guard 的析构函数在调用 join()之前首先判断线程是否可连接。这很重要，因为 join()只能对给定的对象调用一次，所以连接已经连接过的线程将会导致错误。

拷贝构造函数和拷贝赋值操作被标记为=delete，是为了不让编译器自动生成它们。拷贝或者赋值这类对象是危险的，因为它可能超过它所连接线程的作用域而继续存在。通过删除声明，任何尝试给 thread_guard 对象赋值的操作都会引发一个编译错误。想要了解删除函数的更多知识，请参阅附录 A 的 A.2 节。

如果不需要等待线程结束，可以分离(detaching)线程来避免异常安全的问题。这打破了线程与 std::thread 对象的联系，因此可以确保即使 std::thread 对象被销毁了也不会调用 std::terminate()，哪怕线程仍然在后台运行。

## 2.1.4 在后台运行线程

调用 detach()会让线程在后台运行，这就意味没有直接的方法和他通信。也就没法等待线程结束；如果线程被分离了，那就不可能有 std::thread 对象能引用它，所以也不能被连接。分离的线程的确在后台运行；所有权和控制会传递给 C++运行时库，它会保证和线程相关的资源在线程退出的时候被正确的回收。

分离线程经常叫做*守护线程*(daemon threads)这是参照 UNIX 中守护进程的概念，这种运行在后台的进程没有任何显式的用户接口。这种线程的特点是长时间运行；它们运行在应用的整个生命周期中，可能会在后台监视文件系统，还有可能清理没用的对象缓存，亦或优化数据结构。在另一个极端，分离线程也有用武之地，比如用在有另一种机制标识线程什么时候完成的地方，或者用于*发后即忘*(fire and forget)的任务。

如 2.1.2 节所见，调用 std::thread 成员函数 detach()来分离一个线程。之后，相应的 std::thread 对象就与实际执行的线程无关了，并且这个线程也无法连接：

```
std::thread t(do_background_work);
t.detach();
assert(!t.joinable());
```

为了从 std::thread 对象中分离线程，前提是有可进行分离的线程,不能对没有执行线程的 std::thread 对象使用 detach()，这也是 join()的使用条件，并且可以用同样的方式进行检查——只有当 std::thread 对象 t 执行 t.joinable()返回 true，才可以使用 t.detach()。

考虑一个应用程序比如文字处理器能同时编辑多个文档。在 UI 和内部实现有很多种处理方法。目前普遍的一种方式是用多个独立的顶层窗口，每个文档在编辑的时候对应一个。尽管这些窗口看起来完全独立，每个有它自己的菜单，但他们运行在应用的同一个实例。一种内部处理方式是，让每个文档处理窗口拥有自己的线程；每个线程运行同样的代码，但被编辑的文档有不同的数据，以及配套的窗口属性。打开一个文档就要启动一个新线程。处理请求的线程没有兴趣等待其它线程结束，因为它工作在跟别人无关的文档上。因此，这使得运行分离的线程变成首选。

下面代码简要的展示了这种方法：

清单 2.4 使用分离线程去处理其他文档

```
void edit_document(std::string const& filename)
{
  open_document_and_display_gui(filename);
  while(!done_editing())
  {
    user_command cmd=get_user_input();
    if(cmd.type==open_new_document)
```

```
    {
      std::string const new_name=get_filename_from_user();
      std::thread t(edit_document,new_name);  // 1
      t.detach();  // 2
    }
    else
    {
       process_user_input(cmd);
    }
  }
}
```

如果用户选择打开一个新文档，你提示他们需要打开的文档，启动一个新线程去打开那个文档，然后分离线程。因为新线程与当前线程执行相同的操作，只不过换了个文件而已，所以，你可以复用同样的函数(edit_document)，新选择的文件名作为参数提供给它。

这个例子也展示了传递参数启动线程的方法：不仅可以向 std::thread 构造函数传递函数名，还可以传递文件名参数。尽管其他机制也可以用于这里，比如使用一个带有数据成员的函数对象代替普通的带参数的函数，但 C++标准库给你提供了更简单的方式。

## 2.2 传递参数给线程函数

正如清单 2.4 所示，向可调用对象或函数传递参数很简单，只需要将这些参数作为 std::thread 构造函数的附加参数即可。但需要注意的是，在默认情况下，这些参数会被拷贝至新线程的内部存储，新创建的执行线程可以访问它们，然后像临时值那样以右值引用传递给调用对象或者函数。就算函数的参数期待一个引用时仍然是这样。再来看一个例子：

```
void f(int i, std::string const& s);
std::thread t(f, 3, "hello");
```

这里创建了一个和 t 关联的新的执行线程，这个线程调用 f(3, "hello")。注意，虽然 f 需要一个 std::string 作为第二个参数，但字符串的字面值作为 char const *类型传递，然后只在新线程的上下文转换为一个 std::string 对象。当提供的参数是指向自动变量的指针时，这特别重要，代码如下：

```
void f(int i,std::string const& s);
void oops(int some_param)
{
  char buffer[1024]; // 1
  sprintf(buffer, "%i",some_param);
  std::thread t(f,3,buffer); // 2
  t.detach();
}
```

在这里，buffer 是一个指针变量，指向局部变量，然后传递到新线程中。函数 oops 很有可能会在 buffer 转换成 std::string 对象之前就退出了，从而导致未定义的行为。解决方案就是在传递到 std::thread 构造函数之前就将字面值转化为 std::string 对象：

```
void f(int i,std::string const& s);
void not_oops(int some_param)
{
  char buffer[1024];
  sprintf(buffer,"%i",some_param);
  std::thread t(f,3,std::string(buffer));  // 使用 std::string，避免悬垂指针
  t.detach();
}
```

在这里，问题是因为你依赖一个隐式转换将指向 buffer 的指针转换成函数参数期望的 std::string 对象，但这个转换发生的太晚，因为 std::thread 构造函数原样拷贝提供的值，不会转换成期望的参数类型。

相反的情形不太可能出现：对象被拷贝了，并且你需要的是一个非常量引用，因为这不能编译。你可能会尝试使用线程更新一个引用传递的数据结构；例如：

```
void update_data_for_widget(widget_id w,widget_data& data);
void oops_again(widget_id w)
{
  widget_data data;
  std::thread t(update_data_for_widget,w,data);
  display_status();
  t.join();
  process_widget_data(data);
}
```

尽管 update_data_for_widget 的第二个参数期待传入一个引用，但是 std::thread 的构造函数并不知晓；构造函数无视函数期待的参数类型，并盲目地拷贝已提供的变量。不过，内部代码会将拷贝的参数以右值的方式进行传递，这是为了照顾到那些只能进行移动的类型，而后会尝试以右值为实参调用 update_data_for_widget。这将编译失败，因为你没法传递一个右值到一个期望非常量引用的函数，对于熟悉 std::bind 的开发者来说，解决办法是显而易见的：你需要在 std::ref 将参数包装成引用的形式。这里可以将线程调用改为：

```
std::thread t(update_data_for_widget,w,std::ref(data));
```

从而，update_data_for_widget 就会接收到一个 data 变量的引用，而非 data 的临时拷贝，现在代码就能顺利的通过编译。

如果你熟悉 std::bind，就不会对以上述传参语义感到陌生，因为 std::thread 构造函数和 std::bind 的操作是根据相同的机制定义的。比如，你也可以传递一个成员函数指针作为线程函数，只要你提供一个合适的对象指针作为第一个参数：

```
class X
{
public:
  void do_lengthy_work();
};
X my_x;
std::thread t(&X::do_lengthy_work,&my_x);
```

这段代码中在新线程中调用 my_x.do_lengthy_work()，因为 my_x 的地址作为对象指针提供。你也可以为成员函数提供参数 std::thread 构造函数的第三个参数就是成员函数的第一个参数，以此类推。

另一种有趣的情形是，提供的参数不能*拷贝*，只能*移动*：对象内部持有的数据转移给另一对象，移动后，原始对象就变空了。std::unique_ptr 就是这样一种类型，这种类型为动态分配的对象提供自动内存管理。同一时间内，只允许一个 std::unique_ptr 实例指向一个给定对象，并且当这个实例销毁时，指向的对象也将被删除。*移动构造函数*(move constructor)和*移动赋值操作符*(move assignment operator)允许一个对象的所有权在多个 std::unique_ptr 实例中转移(更多移动相关内容，请参考附录 A 的 A.1.1 节)。这样的转移让原对象留下个空指针。这种值的移动允许这种类型的对象作为函数的参数或者从函数返回。当原对象是一个临时值时，移动自动完成，但当原对象是一个命名变量，那么需要调用 std::move()来请求转移。下面的代码展示了 std::move 的用法，展示了 std::move 是如何转移一个动态对象的所有权到一个线程中：

```
void process_big_object(std::unique_ptr<big_object>);

std::unique_ptr<big_object> p(new big_object);
p->prepare_data(42);
std::thread t(process_big_object,std::move(p));
```

通过在 std::thread 的构造函数中指定 std::move(p)，big_object 对象的所有权首先被转移到新创建线程的的内部存储中，之后再被传递给 process_big_object 函数。

C++ 标准线程库中和 std::unique_ptr 在所属权上有相似语义的类有好几种，std::thread 为其中之一。虽然，std::thread 实例不像 std::unique_ptr 那样能占有一个动态对象的所有权，但它确实拥有一个资源：每个实例都负责管理一个执行线程。这个所有权可以在多个 std::thread 实例中互相转移，因为尽管 std::thread 实例不可拷贝，但他们是*可移动的*。这保证了在任何时候一个 std::thread 实例只能关联一个特定的执行线程，同时允许程序员在对象间转移所有权。

## 2.3 转移线程所有权

假设要写一个函数，创建一个线程在后台运行，但是把新线程的所有权传回到调用函数而不是让函数等待它结束；或与之相反：创建一个线程，并且把所有权传递到某个函数，这个函数等待它结束。不管哪种情况，你都需要把所有权从一个地方转移到另一个地方。

这就是将移动操作引入 std::thread 的原因。就像前面章节描述的那样，C++标准库中有很多*资源占有*(resource-owning)类型，比如 std::ifstream，std::unique_ptr 还有 std::thread 都是可移动，但不可拷贝。这就说明特定执行线程的所有权可以在 std::thread 实例中移动，下面将展示一个例子。例子中，创建了两个执行线程，并且在 std::thread 实例 t1，t2 和 t3 之间转移这些线程的所有权：

```
void some_function();
void some_other_function();
std::thread t1(some_function);
std::thread t2=std::move(t1);
t1=std::thread(some_other_function);
std::thread t3;
t3=std::move(t2);
t1=std::move(t3);                          // 赋值操作将使程序崩溃
```

首先，一个新线程启动并与 t1 相关联。当 t2 构造的时候，所有权通过调用 std::move()显示移动所有权到 t2。在这个时间点，t1 和执行线程已经没有关联了，执行 some_function 的函数线程与 t2 关联。

然后，一个临时 std::thread 对象相关的线程启动了。随后所有权转移到 t1 不需要调用 std::move()来显示移动所有权，因为，所有者是一个临时对象——移动操作将会隐式的调用。

t3 是默认构造的，与任何执行线程都没有关联。显式调用 std::move()将与 t2 关联线程的所有权转移到 t3 中。因为 t2 是一个命名对象。这些移动操作完成后，t1 与执行 some_other_function 的线程相关联，t2 没有关联任何线程，t3 与执行 some_function 的线程相关联。

最后一个移动操作，将 some_function 线程的所有权转移给 t1。不过，t1 已经有了一个关联的线程(运行行 some_other_function 的线程)，所以这里系统直接调用 std::terminate()终止程序运行。这样做是为了保证与 std::thread 的析构函数的行为一致。在 2.1.1 节中已经看到，在线程对象被析构前，需要显式的等待线程完成，或者分离它；这同样适用于赋值操作：不能通过赋予管理它的 std::thread 对象一个新值来简单丢弃这个线程。

std::thread 支持移动，就意味着线程的所有权可以在函数外进行转移，就如下面程序所示。

清单 2.5　函数返回 std::thread 对象

```
std::thread f()
{
  void some_function();
  return std::thread(some_function);
}

std::thread g()
{
  void some_other_function(int);
  std::thread t(some_other_function,42);
  return t;
}
```

同样地，如果所有权需要转到到函数中，可以让这个函数接受一个按值传递的 std::thread 实例作为一个参数，代码如下：

```
void f(std::thread t);
void g()
{
  void some_function();
  f(std::thread(some_function));
  std::thread t(some_function);
  f(std::move(t));
}
```

std::thread 支持移动的一个好处是可以以清单 2.3 为基础构建 thread_guard 类，并让它拥有线程的所有权。这避免了当 thread_guard 的生命周期长于它所引用的线程时引起不愉快的后果，还意味着，一旦所有权转移到这个对象以后，没有谁能连接或者分离这个线程。因为这主要为了确保退出作用域之前线程已经完成，下面的代码里定义了 scoped_thread 类，并伴有一个简单示例：

```
清单 2.6 scoped_thread 的用法

class scoped_thread
{
  std::thread t;
public:
  explicit scoped_thread(std::thread t_):
    t(std::move(t_))
  {
    if(!t.joinable())
      throw std::logic_error(“No thread”);
  }
  ~scoped_thread()
```

```
  {
    t.join();
  }
  scoped_thread(scoped_thread const&)=delete;
  scoped_thread& operator=(scoped_thread const&)=delete;
};

struct func; // 定义在清单 2.1 中

void f()
{
  int some_local_state;
  scoped_thread t(std::thread(func(some_local_state)));
  do_something_in_current_thread();
}
```

与清单 2.3 相似，不过新线程直接传递到 scoped_thread 中，而不是为它创建一个命名变量。当初始线程到达 f 函数末尾时，scoped_thread 对象就会销毁，然后连接构造函数中提供的线程。在清单 2.3 中的 thread_guard 类，需要在析构中检查线程是否可连接。你可以在构造函数中完成这个检查，并且当线程不可连接时，抛出异常。（译注：关于构造函数中抛异常的话题详见《More Effective C++》第 10 条，如果没有智能指针的保护，构造函数抛异常是危险的）。

给 C++17 的一个提案就是添加一个 joining_thread 类，这个类型与 std::thread 类似；不同的是在析构函数中自动连接，类似于 scoped_thread。委员会成员们对此并没有达成统一共识，所以这个类没有添加入 C++17 标准中(译注：已经纳入了 C++20 中，名字是 std::jthread)，不过这个类实现起来相当简单。一种可能的实现展示如下：

清单 2.7 joining_thread 类的实现

```
class joining_thread
{
  std::thread t;

public:
  joining_thread() noexcept=default;
  template<typename Callable,typename ... Args>
  explicit joining_thread(Callable&& func,Args&& ... args):
    t(std::forward<Callable>(func),std::forward<Args>(args)...)
  {}
  explicit joining_thread(std::thread t_) noexcept:
    t(std::move(t_))
  {}
  joining_thread(joining_thread&& other) noexcept:
    t(std::move(other.t))
```

```
{}

joining_thread& operator=(joining_thread&& other) noexcept
{
  if（joinable()）
  {
    join();
  }
  t = std::move(other.t);
  return *this;
}

joining_thread& operator=(std::thread other) noexcept
{
  if(joinable())
    join();
  t=std::move(other);
  return *this;
}

~joining_thread() noexcept
{
  if(joinable())
  join();
}

void swap(joining_thread& other) noexcept
{
  t.swap(other.t);
}

std::thread::id get_id() const noexcept
{
  return t.get_id();
}

bool joinable() const noexcept
{
  return t.joinable();
}

void join()
{
  t.join();
```

```
  }

  void detach()
  {
    t.detach();
  }

  std::thread& as_thread() noexcept
  {
    return t;
  }

  const std::thread& as_thread() const noexcept
  {
    return t;
  }
};
```

std::thread 支持移动，如果容器是移动感知的(比如最新的 std::vector<>)，也允许 std::thread 对象的容器。这意味着你可以写出如下清单中的代码，它生成（spawns）了很多线程，然后等待他们结束。

清单 2.8 创建一些线程并等待它们结束

```
void do_work(unsigned id);

void f()
{
  std::vector<std::thread> threads;
  for (unsigned i = 0; i < 20; ++i)
  {
    threads.emplace_back(do_work,i); // 生成线程
  }
  for (auto& entry : threads) // 对每个线程调用 join()
    entry.join();
}
```

如果线程用于划分一个算法的工作，在返回给调用者之前，所有的线程必须结束。清单 2.8 隐含着线程的工作都是自包含的，它们操作的结果纯粹是共享数据的附带产物。如果 f()有返回值，这个返回值依赖于这些线程执行操作的结果，需要在线程终止后，通过检查共享数据来决定。结果在不同线程中转移的替代方案，我们会在第 4 章中再次讨论。

将 std::thread 放入 std::vector 是向线程自动化管理迈出的第一步：不是为这些线程创建独立的变量，并且直接连接它们，而是可以将它们可以看作一个组。你可以通过在运行时创建动态数量的线程让它更进一步，而非像清单 2.8 那样创建固定数量的线程。

## 2.4 运行时选择线程数量

C++标准库中的 std::thread::hardware_concurrency()可以帮大忙。这个函数会返回一个数量指示，表明执行程序真正可以并发的线程数。在一个多核系统中，返回值可以是 CPU 的核数。返回值也仅仅是个提示，当系统信息无法获取时，函数也可能返回 0，但是，在线程间划分任务的时候它是个非常有用的指南。

清单 2.9 展示了一个并行版的 std::accumulate。在真实的代码中，你可能会使用第 10 章中并行版本的 std::reduce，而不是自己实现一份，但它阐述了最基本的思路。它在线程间拆分工作，每个线程处理最小数量的元素，这样可以避免线程太多造成的开销。注意，尽管异常有可能发生，但这个实现假设没有任何操作会抛出异常；比如 std::thread 构造函数无法启动一个执行线程，就会抛出一个异常。在这样的算法中处理异常超出了作为简单示例的范畴，我们将在第 8 章中再来讨论这个话题。

清单 2.9 初级并行版的 std::accumulate

```
template<typename Iterator,typename T>
struct accumulate_block
{
  void operator()(Iterator first,Iterator last,T& result)
  {
    result=std::accumulate(first,last,result);
  }
};

template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
  unsigned long const length=std::distance(first,last);

  if(!length)
    return init;

  unsigned long const min_per_thread=25;
  unsigned long const max_threads=
      (length+min_per_thread-1)/min_per_thread;

  unsigned long const hardware_threads=
      std::thread::hardware_concurrency();

  unsigned long const num_threads=
      std::min(hardware_threads != 0 ? hardware_threads : 2, max_thread
s);
```

```
  unsigned long const block_size=length/num_threads;

  std::vector<T> results(num_threads);
  std::vector<std::thread> threads(num_threads-1);

  Iterator block_start=first;
  for(unsigned long i=0; i < (num_threads-1); ++i)
  {
    Iterator block_end=block_start;
    std::advance(block_end,block_size);
    threads[i]=std::thread(
        accumulate_block<Iterator,T>(),
        block_start,block_end,std::ref(results[i]));
    block_start=block_end;
  }
  accumulate_block<Iterator,T>()(
      block_start,last,results[num_threads-1]);

  for (auto& entry : threads)
    entry.join();

  return std::accumulate(results.begin(),results.end(),init);
}
```

函数很长，但很好理解。如果输入的范围为空，就会返回 init 参数的值。否则，如果范围内多于一个元素时，需要用元素的数量除以最小块大小，从而确定线程的最大数量，这样可以避免在一台 32 核的机器上创建 32 个线程，而实际上你只有 5 个值。

需要运行的线程数量是你算出的最大线程数和硬件线程数量两者中的最小值。因为上下文频繁的切换会降低线程的性能，所以你肯定不想启动的线程数多于硬件支持的线程数量（这称为超订 oversubscription）。当 std::thread::hardware_concurrency()返回 0，可以替换为你选择的一个数；在本例中，我选择了 2。你也不想在一台单核机器上启动太多的线程，因为这样反而会降低性能，但同样你也不想太少，因为太少意味着放弃了可用的并发。

每个线程中处理的条目数量，是范围中元素的总量除以线程的个数。如果你担心分配的数量不均，那请放心——我们会在后面处理这个问题。

现在你知道有多少个线程了，可以创建一个 std::vector<T>容器存放中间结果，并为线程创建一个 std::vector<std::thread>。需要注意的是，启动的线程数要比 num_threads 少 1 个，因为你已经有了一个线程(即主线程)。

启动线程只是一个简单的循环：让 block_end 迭代器往前指向当前块的末尾，并启动一个新线程累加当前块。下一块的起始是当前块的结尾。

启动所有线程后，主线程会处理最后一个块。这里就是你考虑不均匀拆分的地方：你知道最后一个块的末尾肯定是 last，并且这个块有多少个元素无所谓。

一旦累加完最后一个块以后，你可以想清单 2.8 中那样，用 std::for_each 等待所有你创建的线程，然后使用 std::accumulate 将所有结果进行累加。

结束这个例子之前，值得指出的是：如果类型 T 的加法操作不是可结合的（比如 float 或者 double），那么 parallel_accumulate 得到的结果可能与 std::accumulate 得到的结果不同，原因在于把不同范围聚集成块的方式不一样。同样的，这里对迭代器的要求更加严格：必须都是前向迭代器，然而 std::accumulate 可以支持单路的输入迭代器，并且 T 必须支持默认构造这样才能创建 results 的 vector。对于并行算法，这类需求变更很常见；这些算法并行的本质是不同的，因而这会对结果和需求产生影响。实现并行算法会在第 8 章深入讨论，并在第 10 章中会介绍一些 C++17 中支持的并行算法(其中 std::reduce 操作等价于这里的 parallel_accumulate)。需要注意的是：因为不能直接从一个线程中返回一个值，所以需要传递一个到 results vector 的相关条目的引用。在第 4 章中，另一个解决从线程返回结果的方法是通过使用期望(futures)。

在这里，当启动线程的时候，每个线程需要的所有信息都会传进来，包括存储计算结果的位置。然而，并非总是如此；有时候有必要在处理过程的某个环节能够识别线程。你可以传入一个识别编号，例如清单 2.8 中的 i。不过，当需要这个标识的函数在调用栈的深层，同时可被任何线程调用时，这么搞的话就很不方便。好在设计 C++的标准库时，就预见了这个需求，因此每个线程有一个唯一的标识。

## 2.5 标识线程

线程标识类型为 std::thread::id，并可以通过两种方式进行检索。第一种，可以通过调用 std::thread 对象的成员函数 get_id()来获取。如果 std::thread 对象没有与任何执行线程相关联，get_id()将返回默认构造的 std::thread::id 对象，表示“没有任何线程”（“not any thread”）。第二种，调用 std::this_thread::get_id()也可以获得当前线程的标识，这个函数也定义在<thread>头文件中。

std::thread::id 类型的对象可以随意的拷贝和比较，否则的话作为标识作用就不大。如果两个 std::thread::id 类型的对象相等，那它们代表了同一个线程，或者都持有“没有任何线程”（not any thread）的值。如果不相等，那么就代表了两个不同线程，或者一个代表了某个线程，另一个持有“没有任何线程”的值。

C++线程库不会限制你去检查线程标识是否一样，std::thread::id 类型对象提供相当丰富的比较操作符，为所有不同的值提供了全序关系。这意味着可以将其当做容器的键，或者做排序，或以其他方式做比较，只要你认为合适就行。比较操作符为所有不相等的 std::thread::id 提供了全序关系，所以它们的行为符合直观期待：如果 a<b，b<c 时，那么 a<c，等等。标准库也提供 std::hash<std::thread::id>，所以 std::thread::id 也可以作为无序关联容器的键。

std::thread::id 实例常用作检测线程是否需要执行一些操作。例如，当用线程来分割工作，就像清单 2.9 中那样，初始线程需要启动其他线程，因此可能要做一些与其他线程不同的工作。这种情况下，在启动其他线程前，它存储 std::this_thread::get_id()返回的值，然后在算法的核心部分（对所有线程都一样）可以检查自己的线程 ID 和这个存储的值。

```
std::thread::id master_thread;
void some_core_part_of_algorithm()
{
  if(std::this_thread::get_id()==master_thread)
  {
    do_master_thread_work();
  }
  do_common_work();
}
```

或者，当前线程的 std::thread::id 可以存储到一个数据结构中作为操作的一部分。之后在这个数据结构上的操作可以检查存储的 ID 和执行这个操作的线程 ID，从而决定哪些操作是禁止/需要的。

同样，线程 ID 可以用做关联容器的键，在容器中指定需要关联给某个线程的数据，而其他机制如线程局部存储不太适合。例如，这样的一个容器可以用于控制线程存储在其控制下的每个线程的信息，或者在线程间传递信息。

大多数情况下 std::thread::id 作为一个线程的通用标识是足够的；只有当标识有语义意义和它关联时(比如，作为数组的索引)，就需要其他替代方案了。你甚至可以把一个 std::thread::id 实例写到一个输出流比如 std::cout：

```
std::cout<<std::this_thread::get_id();
```

具体的输出结果是严格依赖于具体实现的，C++标准给的唯一保证就是线程 ID 比较结果相等时，必须有相同的输出，如果不相等，则输出不同。这就让在 debug 和日志的时候非常有用，但这个值本身没有语义意义，所以也没啥太多可说的。

# 总结

本章讨论了 C++标准库中基本的线程管理方式：启动线程，等待它们结束，以及让它们在后台运行。我们还看到了如何在线程启动时，向线程函数中传递参数，如何将管理线程的职责从代码的一部分转移到另一部分，以及如何使用线程组来划分任务。最后，讨论了如何标识线程，以便将数据或者行为与不方便通过其他方式关联的特定线程关联起来。尽管你可以通过独立的线程操作独立的数据，来做很多事情，但有时在线程运行的时候，它们之间确实需要共享数据。第 3 章会围绕在线程间直接共享数据的议题展开讨论，第 4 章会围绕在有/没有共享数据情况下，线程同步操作的更多一般议题。

# 第 3 章 线程间共享数据

**本章主要内容**

- 共享数据面临的问题
- 使用互斥锁保护数据
- 保护共享数据的替代设施

使用线程的一个核心收益是在线程间共享数据简单又直接，现在我们已经了解了启动和管理线程，现在让我们来看一下围绕共享数据的那些议题。

想象一下，你和你的朋友合租一个公寓。公寓中只有一个厨房和一个浴室。除非你们是特别友好，否则你俩不能同时使用浴室，并且，如果你的室友长时间占用浴室，而你又需要用它的时候就会特别沮丧。同样，虽然同时做饭是可能的，但如果厨房里有一个组合式烤箱和烤架，其中一个在烤香肠的时候，另一个在烤蛋糕，那么最后不可能有好结果。此外，我们都知道共享一个空间的挫败感：任务完成一半时，突然发现某人把我们需要的东西借走了，或者改变了我们离开时的样子。

线程也是如此。如果在线程间共享数据，你需要有规则来确定哪些线程可以在何时访问哪个数据位，以及如何将任何更新传达给关心该数据的其他线程。数据可以轻松在单进程的多线程之间共享不只是好处，也可能是个大缺点。错误的共享数据使用是产生并发 bug 的一个主要原因，并且后果远比香肠味的蛋糕更加严重。

本章是关于如何在 C++线程间安全共享数据，怎么避免可能引起的潜在问题，并最大化收益。

## 3.1 共享数据面临的问题

归根结底，线程间共享数据的问题全都归咎于修改数据。如果共享数据是只读的，那就没有问题，读数据互相不会有影响。但是，当一个或多个线程要修改共享数据时，就会产生很多麻烦。这种情况下，就必须小心谨慎，才能确保所有线程都工作良好。

不变量(invariants)广泛用于帮助程序员思考他们的代码——对特定数据结构的表述总是真；比如，“这个变量包含链表中的项数”。不变量通常会在一次更新中被破坏，特别是当数据结构比较复杂或者更新需要修改多个值时。

双链表中每个节点都有一个指针指向链表中下一个节点，还有一个指针指向前一个节点。其中一个不变量就是如果节点 A 通过“next”指向节点 B，那么节点 B 会通过“previous”指针回指 A。为了从链表中删除一个节点，其两边节点的指针都需要更新。当其中一边更新完成时，不变量就被破坏了，直到另一边也完成更新；在两边都完成更新后，不变量就有被支持。

从一个链表中删除一个节点的步骤如图 3.1 所示：

a. 标识要删除的节点 N

b. 更新 N 前面节点的链接指向 N 的下一个节点

c. 更新 N 后面节点的链接指向 N 前面的节点

d. 删除节点 N

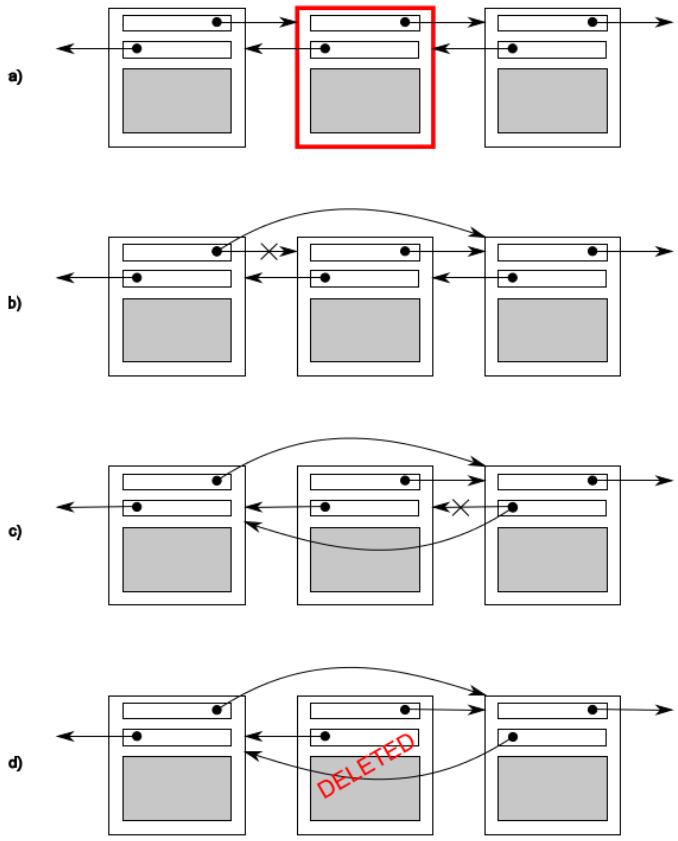


图 3.1 从一个双链表中删除一个节点

从图 3.1 可以看到，在步骤 b 和 c 中间，从一个方向上的链接和反方向的链接不一致，这就破坏了不变量。

修改线程间共享数据最简单的潜在问题是破坏了不变量。假使你不做任何特别的事情来确保，如果一个线程正在读取双向链接的链表同时另一个线程正在删除节点，读线程很有可能看到链表上的一个节点只被部分移除(因为只有一边的链接被改变了，如图 3.1 的步骤 b)，这样不变量就被破坏了。破坏不变量的后果是不确定的，当其他线程按从左往右的顺序来访问链表时，它将跳过被删除的节点。在一方面，如有第二个线程尝试删除图中最右边的节点，那么可能会让数据结构产生永久性的损坏，最终使程序崩溃。无论结果如何，这是并发代码中最常见的引起 bug 的原因：竞争条件(race condition)。

### 3.1.1 竞争条件

假如你正在电影院买电影票。如果去的是大电影院，有很多收银员可以收钱，因此很多人可以在同一时间买电影票。假设某个人在另一个收银台也在买你想看的电影的电影票，有哪些座位供你选择取决于别人先订还是你先订。如果只剩下少量的座位，这个区别可能非常关键：毫不夸张的说，这可能成为一场看谁拿到最后一张票的比赛。这就是竞争条件的例子：你的座位(或者更甚，你是否能买到票)取决于两个购买的相对顺序。

在并发中的竞争条件是指任何结果取决于在两个或多个线程上操作执行的相对顺序；线程竞争去执行它们各自的操作。大多数情况下，这种竞争是良性的，因为任何可能的结果都可接受，哪怕这些结果会随着不同的相对顺序而改变。例如，有两个线程同时向一个队列中添加任务，只要保持住系统不变量，通常谁先谁后都没有什么影响。当竞争条件导致不变量遭到破坏时，才会发生问题，比如双向链表的例子。当谈到并发时，术语“竞争条件”通常指有问题的竞争条件；良性的竞争条件不是很有趣也不会引起 bug。C++标准中也定义了“数据竞争”这个术语意指这个特殊的竞争条件是由于并发的去修改一个单一对象(参见 5.1.2 节)，数据竞争会导致可怕的未定义行为。

有问题的竞争条件典型地发生在完成一个操作需要修改多个不同的数据片段，比如示例对两个连接指针的修改。因为这个操作要访问两个独立的数据片段，它们必须被独立的指令修改，并且当只有其中之一的指令完成的时候，另一个线程可能潜在地访问这个数据结构。因为出现的概率太低，竞争条件难于发现和复现。如果修改是通过连续的 CPU 指令完成的，这个问题出现在任何一次运行中的可能性都很小，即使数据结构被另一个线程并发的访问。随着系统负载增加时，以及操作执行次数的增加，有问题的执行序列出现的机会也在增加。不可避免的是这类问题会在最不适宜的时候出现。竞争条件通常是时序敏感的，所以程序在调试模式运行时，它们经常会完全消失，因为调试模式会影响程序的时序，哪怕只是很轻微地影响。

如果你在写多线程程序，竞争条件就会成为你生命中的祸害；编写并发软件最大的复杂性来自于避免有问题的竞争条件。

### 3.1.2 避免有问题的竞争条件

有一些方法可以解决有问题的竞争条件，最简单的选项就是把数据结构包裹在某种保护机制下，确保只有进行修改的线程才能看到不变量被破坏时的中间状态。从其他访问数据结构的线程的视角来看，修改要么还没开始要么已经完成。C++标准库提供一些这类的机制，我们会在本章描述。

另一个选项是修改数据结构和不变量的设计，以便让修改由一系列不可分割的变化来完成，每一个都维持了不变量，这就是所谓的无锁编程，不过，这种方式很难得到正确的结果。如果在这个层级上工作，内存模型上的细微差异，以及识别哪个线程能看到哪些值集会变的很复杂。内存模型将在第 5 章讨论，无锁编程将在第 7 章讨论。

另一种处理竞争条件的方式是，把更新数据结构当做事务来处理，就像在一个事务中更

新数据库一样。所需的一系列数据修改和读取存储在一个事务日志中，然后再一步提交。如果提交因为数据结构被另一个线程修改了而不能进行，事务就会被重启。这称作为“软件事务内存”(STM，software transactional memory)。写作这本书的时候这是一个很活跃的研究领域。本书不会讨论这个话题，因为在C++中没有直接支持STM （尽管C++有事务性内存扩展的技术规范[1])。但是，私下做一些事情然后一步提交的思想会在后面提及。

C++标准库提供的最基本的保护共享数据的机制是互斥锁(mutex)，所以我们先来看这个。

---

[1] SO/IEC TS 19841:2015—Technical Specification for C++ Extensions for Transactional
Memory http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=66343 .

# 3.2 使用互斥锁保护共享数据

当程序中有共享的数据结构时，比如前面章节的链表，你肯定不想让程序陷入竞争条件，或是出现不变量被破坏的情况。如果你能将所有访问数据结构的代码片段都标记为互斥的（mutually exclusive）岂不美哉？这样的话，任何一个线程在执行其中一段时，其他线程试图访问共享数据的话，就必须进行等待，直到第一个线程操作结束。除非该线程就在执行修改操作，否则任何线程都不可能会看到被破坏的不变量。

嗯，这不是童话里的愿望——如果你使用一个称为互斥锁（mutex，*mut*ual *ex*clusion的组合）的同步原语的话，这正是你所得到的。在访问共享数据前，锁住和数据关联的互斥锁，在访问结束后，再将数据关联的互斥锁解锁。线程库需要保证，一旦一个线程已经锁住了指定的互斥锁，所有其他线程试图锁住这把锁时必须等待，直到成功锁住互斥锁的线程解锁它。这就保证了所有线程都能看到共享数据的自我一致的视图，而不破坏不变量。

互斥锁是C++数据保护最通用的机制，但它不是万能的；组织你的代码来保护正确的数据（见3.2.2节)，并避免接口中固有的竞争条件(见3.2.3节)是非常重要的。不过，互斥锁自身也有问题，会造成死锁(见3.2.4节)，以及保护了太多或太少的数据(见3.2.8节)。让我们从基础开始。

## 3.2.1 C++中使用互斥锁

C++中通过构建一个std::mutex实例创建互斥锁，通过成员函数lock()对互斥锁上锁，unlock()进行解锁。不过，实践中不推荐直接去调用成员函数，因为这意味着，必须记住在每个函数出口都要去调用unlock()，也包括发生异常的情况。C++标准库为互斥锁提供了一个RAII语法的std::lock_guard类模板，在构造时锁住提供的互斥锁，并在析构的时候进行解锁，从而保证了一个锁住的互斥锁能被正确解锁。下面的程序清单中，展示了如何在多线程中，使用std::mutex构造的std::lock_guard实例，对一个链表进行访问保护。std::mutex和std::lock_guard都在<mutex>头文件中声明。

清单 3.1 使用互斥锁保护链表

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> some_list;    // 1
std::mutex some_mutex;    // 2

void add_to_list(int new_value)
{
  std::lock_guard<std::mutex> guard(some_mutex);    // 3
  some_list.push_back(new_value);
}

bool list_contains(int value_to_find)
{
  std::lock_guard<std::mutex> guard(some_mutex);    // 4
  return std::find(some_list.begin(),some_list.end(),value_to_find) !=
some_list.end();
}
```

清单 3.1 中有一个全局变量①，这个全局变量被一个全局的 std::mutex 实例保护②。add_to_list()③和 list_contains()④函数中使用 std::lock_guard<std::mutex>，使得这两个函数中对数据的访问是互斥的：list_contains()永远也看到链表正在被 add_to_list()修改。

C++17 中添加了一个新特性，称为类模板参数推导，这意味着对于像 std::locak_guard 这样的简单类模板，其模板参数列表可以省略。在 C++17 编译器上③和④的代码可以简化成：

```
std::lock_guard guard(some_mutex);
```

3.2.4 节中，会介绍 C++17 中的一种加强版锁保护机制——std::scoped_lock，所以在 C++17 环境中，上面的这行代码也可以写成：

```
std::scoped_lock guard(some_mutex);
```

为了让代码更加清晰，并且兼容老的编译器，我将会继续使用 std::lock_guard，并在代码片段中写明模板参数的类型。

某些情况下使用全局变量没问题，但在大多数情况下，互斥锁通常会与需要保护的数据放在同一类中，而不是使用全局变量。这是面向对象设计准则的标准应用：将其放在一个类中，就可以清晰标记它们为相关的，也可对类的功能进行封装，进而增强保护。这种情况下，函数 add_to_list 和 list_contains 可以作为这个类的成员函数。互斥锁和需要保护的数

据，在类中都定义为 private 成员，这可以更简单的识别哪些代码需要访问这个数据，因而哪些代码需要锁住互斥锁。当所有成员函数在访问任务数据成员前锁住互斥锁，结束时解锁，数据就被很好的保护起来免受所有外来者的干扰。

嗯，这并不完全正确，精明的你一定注意到了：当其中一个成员函数返回的是保护数据的指针或引用时，那么成员函数对互斥锁处理的再好也没有用，因为你在保护罩上炸了个大洞。任何访问指针或者引用的代码现在可以访问（并且潜在的修改）保护的数据而不需要锁住互斥锁。用互斥锁保护数据因而需要谨慎的设计接口，来保证在访问任何保护数据之前，互斥锁是锁住的，并且不留后门。

## 3.2.2 构建代码来保护共享数据

你已经看到，使用互斥锁来保护数据，并不是仅仅在每一个成员函数中都加入一个 std::lock_guard 对象那么简单；一个迷失(stray)指针或引用，也会让这种保护形同虚设。不过，检查迷失指针或引用很容易，只要没有成员函数返回指向保护数据的指针或引用，数据就是安全的。如果你再深入一点，就没这么直观了——没什么永远是。和确保成员函数不会传出指针或引用一样，检查它们不会把这些指针或者引用传给它们调用的函数同样重要，尤其这些函数不受你控制。这些函数可能存储指针或者引用到某个地方，之后在没有互斥锁保护的情况下使用。这方面尤其危险的是：函数是在运行时通过一个函数参数提供的，如同下面清单中所示那样。

清单 3.2 无意中传递了被保护数据的引用

```
class some_data
{
  int a;
  std::string b;
public:
  void do_something();
};

class data_wrapper
{
private:
  some_data data;
  std::mutex m;

public:
  template<typename Function>
  void process_data(Function func)
  {
    std::lock_guard<std::mutex> l(m);
    func(data);    // 1 传递“被保护的”数据给用户函数
  }
```

```
};

some_data* unprotected;

void malicious_function(some_data& protected_data)
{
  unprotected=&protected_data;
}

data_wrapper x;
void foo()
{
  x.process_data(malicious_function);     // 2 传递一个恶意函数
  unprotected->do_something();     // 3 在无保护的情况下访问保护数据
}
```

例子中process_data看起来没有任何问题，std::lock_guard对数据做了很好的保护，但调用用户提供的函数func①，就意味着foo能够绕过保护机制将函数malicious_function传递进去②，然后在没有锁定互斥锁的情况下调用do_something()。

从根本上说，这段代码的问题在于它没有完成你想做的事情：标记所有访问数据结构的代码片段为互斥的。在这里，遗漏了函数foo()中调用unprotected->do_something()的代码。不幸的是，这部分问题C++线程库无法提供任何帮助，只能由开发者使用正确的互斥锁来保护数据。从正面看，还是有章可循的：切勿将指向保护数据的指针或引用传递到互斥锁作用域之外，无论是从函数返回它们，还是把他们存储在外部可见的内存，亦或是把他们以参数形式传递给用户提供的函数。

虽然这是在使用互斥锁保护共享数据时常犯的错误，但远不是唯一陷阱。下一节中，你将会看到，即便是使用了互斥锁对数据进行了保护，竞争条件依旧可能存在。

### 3.2.3 发现接口中固有的竞争条件

仅因为使用了互斥锁或其他机制保护了共享数据，并不意味着你不会受到竞争条件的影响；你仍然必须确保合适的数据受到了保护。考虑双链表的例子，为了能让线程安全地删除一个节点，需要确保防止对这三个节点的并发访问：需要删除的节点及其前后两个节点。如果只对指向每个节点的指针进行单独的访问保护，那比没有使用互斥锁也好不了多少，因为竞争条件仍会发生——不是在单个步骤需要保护单个节点，是整个数据结构和整个删除操作需要保护。这种情况下最简单的解决方案就是使用互斥锁来保护整个链表，如清单 3.1 所示。

尽管链表的单个操作是安全的，但不意味着你就走出了困境；即使在一个很简单的接口中，依旧可能遇到竞争条件。考虑栈的数据结构，像清单3.3中展示的std::stack容器适配器，除了构造函数和swap()以外，只需要对std::stack提供五个操作：push()一个新元素进栈，pop()一个元素出栈，top()查看栈顶元素，empty()判断栈是否为空，size()了解

栈中有多少个元素。如果你更改 top()，使其返回一个拷贝而非引用(即遵循了 3.2.2 节的准则)并使用互斥锁保护内部数据，这个接口本质上仍然受制于竞争条件。这个问题不是基于互斥锁实现所特有的；这是一个接口问题，因此在无锁实现中仍然会出现竞争条件。

清单 3.3 std::stack 容器适配器接口

```
template<typename T,typename Container=std::deque<T> >
class stack
{
public:
  explicit stack(const Container&);
  explicit stack(Container&& = Container());
  template <class Alloc> explicit stack(const Alloc&);
  template <class Alloc> stack(const Container&, const Alloc&);
  template <class Alloc> stack(Container&&, const Alloc&);
  template <class Alloc> stack(stack&&, const Alloc&);

  bool empty() const;
  size_t size() const;
  T& top();
  T const& top() const;
  void push(T const&);
  void push(T&&);
  void pop();
  void swap(stack&&);
  template <class... Args> void emplace(Args&&... args); // C++14 的新特
性
};
```

虽然 empty()和 size()可能在返回时是正确的，但其结果是不可靠的；当它们返回后，其他线程就可以自由地访问栈，并且可能 push()新元素到栈中，也可能 pop()一些已在栈中的元素。这些操作可能发生在调用 empty()或者 size()的线程使用它们的返回值之前。

当栈实例不是共享的，使用 empty()检查非空再调用 top()访问栈顶部的元素是安全的。如下所示：

```
stack<int> s;
if (! s.empty()) // 1
{
  int const value = s.top();    // 2
  s.pop();    // 3
  do_something(value);
}
```

以上，不仅在单线程代码中是安全的，而且在空堆栈上调用 top()是未定义的行为也符

合预期。 对于共享的栈对象，这样的调用顺序就不再安全了，因为在调用 empty()①和调用 top()②之间，可能有来自另一个线程的 pop()调用，它会删除最后一个元素。这是一个经典的竞争条件，使用互斥锁对栈内部数据进行保护，但依旧不能阻止它；这是接口带来的问题。

怎么解决呢？这个问题是接口设计的后果，所以解决的方法就是改变接口。有人会问：怎么改？在这个简单的例子中，当调用 top()时，发现栈已经是空的了，那么就抛出异常。虽然这能直接解决这个问题，但它让编码更加笨重，因为现在你需要有能力捕获异常，即使 empty()返回 false。这使得对 empty()的调用变成了一个优化，它避免了在栈已经为空的情况下抛出一个异常（尽管如果状态在 empty()和 top()之间发生了改变，异常仍然会抛出），而不是设计必要的部分。

当仔细的观察之前的代码段，就会在调用 top()②和 pop()③之间发现另一个潜在的竞争条件。假设两个线程运行着前面的代码，并且都引用同一个栈对象 s。这并非罕见的情况，当为性能而使用线程时，多个线程在不同的数据上执行相同的操作是很平常的，并且共享堆栈对象是在它们之间划分工作的理想方式（虽然更通用的是，使用队列——参见第 6 和 7 章的例子）。假设，一开始栈中有两个元素，在两个中的任何一个线程上你都不需要担心 empty()和 top()之间的竞争，只需要考虑潜在的执行模式。

当栈被一个内部互斥锁所保护时，一次只有一个线程可以调用栈的成员函数，所以调用可以很好地交错执行，但是 do_something()是可以并发运行的。在表 3.1 中，展示一种可能的执行顺序。

表3.1 两个线程在栈上一种可能的执行顺序

| Thread A | Thread B |
| --- | --- |
| if (!s.empty); | |
| | if(!s.empty); |
| int const value = s.top(); | |
| | int const value = s.top(); |
| s.pop(); | |
| do_something(value); | s.pop(); |
| | do_something(value); |

你可以看到，如果只有这两个线程在运行，在对 top()的两次调用之间没有任何东西可以修改栈，所以，两个线程都看到相同的值。不仅是这样，在两次 pop()之间没有调用 top()。结果，栈中两个值中的一个没有读出来就被扔掉了，另一个值又被处理了两次。这是另一个竞争条件，比 empty()/top()未定义行为的竞争更加阴险：看起来没有任何错误，这个 bug 的后果又远离原因，尽管这个后果明显依赖于 do_something()做了什么。

这就要求接口来一个翻天覆地的变化，其中之一就是组合 top()和 pop()在一个互斥锁的保护下。Tom Cargill[1]指出如果对象的拷贝构造函数在栈中会抛出一个异常，这样的处理方式就会有问题。这个问题由 Herb Sutter[2]从异常安全的角度进行了相当全面的处理，不过潜在的竞争条件，为组合带来了一些新东西。

如果还没有意识到问题，考虑 stack<vector<int>>。vector 是一个动态容器，当你拷贝一个 vector，标准库会从堆上分配更多的内存来完成这次拷贝。如果这个系统处在重度负荷，或有严重的资源限制的情况下，这种内存分配就会失败，所以 vector 的拷贝构造函数可能会抛出一个 std::bad_alloc 异常。当 vector 中存有大量元素时，很可能就是这种情况。如果 pop()函数定义成返回弹出值时，当然也就从栈中移除了这个值，那么会有一个潜在的问题：只有在栈已经修改以后，这个弹出值才返回给调用者；但当拷贝数据的返回给调用者的时候可能抛出一个异常。如果这种情况发生了，要弹出的数据就丢失了；它的确从栈上移除了，但是拷贝失败了！std::stack 接口的设计人员将这个操作分为两部分：先获取顶部元素(top())，然后从栈中移除(pop())。这样，如果不能安全的拷贝数据，它仍然呆在栈上。如果问题是堆内存不足，应用可能会释放一些内存，然后重试。

不幸的是，在消除竞争条件时，你试图避免的正是这种分割。幸运的是，我们还有别的选项，但是它们是要付出代价的。

**选项 1： 传入一个引用**

第一个选项是将变量的引用作为参数，传入 pop()函数中获取想要的弹出值：

```
std::vector<int> result;
some_stack.pop(result);
```

大多数情况下，这种方式还不错，但有个明显的劣势：需要调用代码提前构造出一个栈中值类型的实例，用于接收目标值。对于一些类型，这样做是不现实的，因为构造一个实例，从时间和资源的角度上来看，都很昂贵。对于其他的类型，这样也不总能行得通，因为构造函数需要的一些参数，在这个阶段的代码不一定具备。最后，它要求存储的类型必须是可赋值的，这是一个重大限制：很多用户自定义类型可能都不支持赋值操作，即使它们可能支持移动构造，甚至是拷贝构造(从而允许按值返回)。

**选项 2：要求一个无异常抛出的拷贝构造函数或移动构造函数**

对于有返回值的 pop()函数来说，只有异常安全方面的问题，如果值返回可以抛出一个异常的话。很多类型都有不会抛出异常的拷贝构造函数，并且随着新标准中对右值引用的支持(详见附录 A，A.1 节)，更多类型都将会有一个不会抛出异常的移动构造函数，即使他们的拷贝构造函数会抛出异常。一个有效的选项是限制线程安全的栈只用在这些能够安全按值返回并且不会抛异常的类型上。

这虽然是安全的但并不理想。尽管能在编译时可使用 std::is_nothrow_copy_constructible 和 std::is_nothrow_move_constructible 类型特性（type traits），检测出拷贝或移动构造函数不会抛出异常，但是这种方式的局限性太强。在用户自定义的类型中，会在拷贝构造函数中抛异常，并且没有移动构造函数的要比不抛异常的更多（这种情况会随

着人们习惯于 C++11 中的右值引用而有所改变)。如果这些类型不能被存储在你的线程安全的栈中，那将是多么的不幸。

**选项 3：返回指向弹出值的指针**

第三个选项是返回一个指向弹出项的指针，而不是按值返回项。指针的优势是自由拷贝，并且不会产生异常，这样你就能避免 Cargill 提到的异常问题了。缺点就是返回一个指针需要管理分配给对象的内存，对于简单数据类型，比如整型，内存管理的开销要远大于按值返回。对于选择这个方案的接口，使用 std::shared_ptr 作为指针类型是个不错的选择；不仅能避免内存泄露，因为一旦最后一个指针销毁了，对象也会销毁，而且库能够完全控制内存分配机制，也就不需要 new 和 delete 操作。从优化的目的看这是很重要的：需要栈中的每个对象分别用 new 来分配，相比原始的非线程安全版本强加了太多开销。

**选项 4：“选项 1 + 选项 2”或 “选项 1 + 选项 3”**

灵活性永远不应该忽视，尤其对通用代码而言。如果你已经选择了选项 2 或 3 时，再提供选择 1 也是很容易的。这样不费吹灰之力，就为代码的用户提供了选择最适合它们选项的能力。

**线程安全栈的示例定义**

清单 3.4 中是一个接口没有竞争条件的栈类的定义，它实现了选项 1 和选项 3：有两个重载的 pop()，其中一个使用引用，这个引用指向存储值的位置，另一个返回 std::shared_ptr<>。它接口很简单，只有两个函数：push()和 pop()；

清单 3.4 线程安全栈的轮廓

```
#include <exception>
#include <memory>  // For std::shared_ptr<>

struct empty_stack: std::exception
{
  const char* what() const throw();
};

template<typename T>
class threadsafe_stack
{
public:
  threadsafe_stack();
  threadsafe_stack(const threadsafe_stack&);
  threadsafe_stack& operator=(const threadsafe_stack&) = delete; // 1 赋值操作被删除

  void push(T new_value);
```

```
  std::shared_ptr<T> pop();
  void pop(T& value);
  bool empty() const;
};
```

通过削减接口不仅可以获得最大程度的安全，甚至对整个栈的操作也被限制了。栈是不能直接赋值的，因为赋值操作已经删除了①(详见附录 A，A.2 节)，并且这里没有 swap()函数。假设栈中的元素支持拷贝，那么栈也可以支持拷贝。当栈为空时，pop()函数会抛出一个 empty_stack 异常，所以一切运行正常，哪怕调用 empty()之后栈被修改了。如选项 3 描述的那样，使用 std::shared_ptr 可以避免内存分配管理的问题，并避免过度调用 new 和 delete 操作。栈中的五个操作，现在就剩下三个：push()，pop()和 empty()。甚至 empty()都是多余的。简化接口更有利于控制数据，可以保证互斥锁将一个操作完全锁住。下面的代码将展示一个简单的实现——封装了 std::stack<>。

清单 3.5 完整线程安全栈定义

```
#include <exception>
#include <memory>
#include <mutex>
#include <stack>

struct empty_stack: std::exception
{
  const char* what() const throw() {
    return "empty stack!";
  };
};

template<typename T>
class threadsafe_stack
{
private:
  std::stack<T> data;
  mutable std::mutex m;

public:
  threadsafe_stack()
    : data(std::stack<T>()){}

  threadsafe_stack(const threadsafe_stack& other)
  {
    std::lock_guard<std::mutex> lock(other.m);
    data = other.data; // 1 在构造函数体中执行拷贝
  }
```

```
  threadsafe_stack& operator=(const threadsafe_stack&) = delete;

  void push(T new_value)
  {
    std::lock_guard<std::mutex> lock(m);
    data.push(new_value);
  }

  std::shared_ptr<T> pop()
  {
    std::lock_guard<std::mutex> lock(m);
    if(data.empty()) throw empty_stack(); // 在调用 pop 前，检查栈是否为空

    std::shared_ptr<T> const res(std::make_shared<T>(data.top())); // 在
修改栈前，分配出返回值
    data.pop();
    return res;
  }

  void pop(T& value)
  {
    std::lock_guard<std::mutex> lock(m);
    if(data.empty()) throw empty_stack();

    value=data.top();
    data.pop();
  }

  bool empty() const
  {
    std::lock_guard<std::mutex> lock(m);
    return data.empty();
  }
};
```

这个栈可以拷贝——拷贝构造函数在源对象上锁住互斥锁，再拷贝内部栈。在构造函数体中①拷贝而不是成员初始化列表，是因为要确保整个拷贝期间都持有互斥锁。

之前对 top()和 pop()函数的讨论中，显示接口中有问题的竞争条件是因为锁的粒度太小，需要保护的操作并未全部覆盖到。不过，锁的粒度过大同样会有问题。极端情况是用一个全局互斥锁要去保护全部共享数据，在一个系统中存在有大量的共享数据时，这会抵消并发带来的性能收益，因为线程被强制变成一次只能运行一个，哪怕他们访问的是不同的数据位。为多处理器系统设计的第一版 Linux 内核中，就使用了单一的全局内核锁。虽然这能正常工作，但在双核处理系统的上的性能要比两个单核系统的性能差很多，四核系统比四个单核系统就差的更远。内核上有太多竞争，使得运行在额外处理器上的线程没有办法执行有用

的工作。之后的 Linux 内核转向了更细粒度锁机制，因为少了很多竞争，所以四核处理系统的性能就和单核处理的四倍差不多了。

使用细粒度锁机制的一个问题是有时候为了保护一个操作中的所有数据，你需要多个互斥锁上锁。如前所述，有时正确的做法是增大互斥锁覆盖数据的粒度，这样的话只需要锁住一个互斥锁。但是，有时候这不太可取，比如当互斥锁正在保护一个类的不同实例；这种情况下，在下一级上锁意味着要么把上锁的操作交给用户，要么一把锁保护类的所有实例。当然，这两种方式都不是特别可取。

对一个给定操作，如果你最终必须锁住两个或更多互斥锁，另一个潜在的问题将伺机而动：死锁。与竞争条件完全相反——不同于两个线程竞争谁先谁后，每个线程会互相等待，所以导致双方都没有取得什么进展。

### 3.2.4 死锁：问题和解决方案

试想有一个玩具，这个玩具由两部分组成，必须拿到这两个部分，才能玩——例如，一个玩具鼓，和一个鼓锤。现在假设你有两个小孩，他们都很喜欢玩这个玩具。当其中一个孩子拿到了鼓和鼓锤时，那就可以尽情的玩耍，直到厌倦为止。如果另一孩子想要玩，他就得等待。再试想，鼓和鼓锤被分别埋在玩具箱里，并且两个孩子在同一时间都想要玩。之后，他们就跑到玩具箱里搜查。其中一个找到了鼓，并且另外一个找到了鼓锤。现在问题就来了，除非其中一方决定做个好孩子让另一个先玩，否则每个人都抓住自己拥有的，并期望获得玩具的另一部分，这样的话谁也别想玩。

想象一下，现在没有孩子去争抢玩具，但线程争抢着给互斥锁上锁：一对线程中的每一个都需要锁住一对互斥锁来执行某些操作，其中每个线程都有一个互斥锁，并且正在等待另一个。这样没有线程能继续进行，因为他们都在等待对方释放互斥锁。这种情况就是死锁，它是需要给多个锁上锁引发的最大的问题。

避免死锁的一般建议，就是让两个互斥锁总是以相同的顺序上锁：如果总在互斥锁 B 之前锁住互斥锁 A，就永远不会死锁。有时这很简单，因为不同的互斥锁用于不同的目的。不过，事情没那么简单，比如：当有多个互斥锁保护同一个类的不同实例时。考虑，例如，一个操作对同一个类的两个不同实例进行数据的交换操作，为了保证数据被正确的交换，就要避免数据被并发修改，每个实例上的互斥锁都必须锁住。但是，如果选择一个固定的顺序(例如，先是第一个参数对应的互斥锁，然后是第二个参数的互斥锁)，这可能会适得其反：只要交换一下参数的位置，两个线程试图在相同的两个实例间交换数据，就会发生死锁！

幸运的是，C++标准库有医治这个的良方，std::lock——一个函数，它可以一次性锁住两个或更多的互斥锁，并且没有死锁的风险。下面的程序清单中，展示了怎么在一个简单的交换操作中使用 std::lock。

清单 3.6 交换操作中使用 std::lock()和 std::lock_guard

```
class some_big_object;
void swap(some_big_object& lhs,some_big_object& rhs);
```

```
class X
{
private:
  some_big_object some_detail;
  std::mutex m;
public:
  X(some_big_object const& sd):some_detail(sd){}

  friend void swap(X& lhs, X& rhs)
  {
    if(&lhs==&rhs)
      return;
    std::lock(lhs.m,rhs.m); // 1
    std::lock_guard<std::mutex> lock_a(lhs.m,std::adopt_lock); // 2
    std::lock_guard<std::mutex> lock_b(rhs.m,std::adopt_lock); // 3
    swap(lhs.some_detail,rhs.some_detail);
  }
};
```

首先，检查参数是否是不同的实例，因为当你已经持有互斥锁的时候，试图锁住互斥锁是未定义的行为。（标准库中提供的可以在同一线程上多次上锁的互斥锁是std::recursive_mutex。详情见 3.3.3 节）。然后，调用 std::lock()①锁住两个互斥锁，并且创建两个 std:lock_guard 实例②③。额外提供的 std::adopt_lock 参数告诉 std::lock_guard 互斥锁已经上锁了，接过互斥锁的所有权就可以了，不用在 std::lock_guard 的构造函数里上锁。

这确保了受保护的操作可能抛出异常的情况下，在函数退出时正确地解锁互斥锁，它还允许一个简单的返回。还有值得注意的是，当使用 std::lock 去锁住 lhs.m 或 rhs.m 时，可能会抛出异常；这种情况下，异常会传播到 std::lock 之外。当 std::lock 成功的获取了一个互斥锁上的锁，并且当其尝试从另一个互斥锁上再获取锁时，如果有异常抛出，第一个锁会自动释放：std::lock 给互斥锁上锁的时候提供了要么全做要么不做的语义(all-or-nonthing)。

C++17 对这种情况提供了支持，std::scoped_lock<>一种新的 RAII 模板，除了它是个可变参模板以外，与 std::lock_guard<>的功能等价，它接受一个互斥锁类型的列表作为模板参数，并且互斥锁列表作为构造函数的参数。提供给构造函数的互斥锁使用与 std::lock 相同的算法上锁，所以当构造函数完成时，它们全部上锁了，同时它们在析构函数中全部解锁。清单 3.6 中 swap()操作可以重写如下：

```
void swap(X& lhs, X& rhs)
{
  if(&lhs==&rhs)
    return;
  std::scoped_lock guard(lhs.m,rhs.m); // 1
  swap(lhs.some_detail,rhs.some_detail);
```

```
}
```

这个例子使用了 C++17 的另一个特性：类模板参数自动推导。如果你手头上有支持 C++17 的编译器(如果你正在使用 std::scoped_lock，那你的编译器肯定就是了，因为它是 C++17 库里的设施)，C++17 隐式参数模板类型推导机制将从传给对象①的构造函数的对象类型中选择正确的互斥锁类型。这行代码等价于下面全部指定的版本：

```
std::scoped_lock<std::mutex,std::mutex> guard(lhs.m,rhs.m);
```

std::scoped_lock 的好处在于，大多数情况下，在 C++17 以前使用的 std::lock 都可以用 std::scoped_lock 重写，这样犯错的概率更小，这是好事。

虽然 std::lock(和 std::scoped_lock<>)可以在这些需要一起获取两个或者更多锁情况下避免死锁，但它没办法帮你分别获取锁。这时，依赖于开发者的纪律性来确保你的程序不会死锁。但这并不简单：死锁是多线程编程中一个令人相当头痛的问题，并且死锁经常是不可预见的，因为在大多数时间里，一切都很正常。不过，有一些相对简单的规则能帮助你写出“无死锁”的代码。

## 3.2.5 避免死锁的更多指南

死锁不仅仅发生在锁上，尽管这是最常见的原因。无锁的情况下，仅需要两个线程 std::thread 对象互相调用 join()，就能产生死锁。这种情况下，没有线程可以继续运行，因为他们正在互相等待对方结束，就像小孩争夺它们的玩具。这种情况很常见，一个线程会等待另一个线程，其他线程同时也会等待第一个线程，它不限于两个线程：三个或更多线程的互相等待也会发生死锁。避免死锁的指南全部可以归结为一个理念：不要等待另一个线程，如果它有可能等待你的话。个别指南提供了识别和消除其他线程正在等待您的可能性的方法。

**避免嵌套锁**

第一个想法往往是最简单的：如果你已经持有一把锁的话不要再去获取一把。如果坚持这条原则，就不可能仅从锁的用法上造成死锁，因为每个线程只持有一把锁。虽然可以从其他途径造成死锁（比如线程的相互等待），但互斥锁是造成死锁的最常见原因。当你需要获取多个锁，使用 std::lock 来避免死锁。

**避免在持有锁时调用用户提供的代码**

这是对前面指南的简单继承。因为代码是用户提供的，你不知道它会做什么；它可能做任何事情，包括获取锁。你在持有锁的情况下，调用用户提供的代码，并且这个代码要获取一个锁，就会违反了避免嵌套锁的准则，因而可能造成死锁。有时，这是无法避免的。如果你正在写一份通用代码，例如 3.2.3 中的栈，每一个参数类型上的操作都是用户提供的代码，这种情况下，就需要其他指南来帮助你。

**使用固定顺序获取锁**

当你绝对必须获取两个或两个以上的锁，并且不能使用 std::lock 来获取它们时，最好的方法是在每个线程上，用固定的顺序获取它们。我在 3.2.4 节中提到了这个方法，用于当需要获取两个互斥锁时避免死锁;这种方法的关键在于以某种方式定义一个线程间一致的顺序。在某些情况下，这相对容易。比如，3.2.3 节中的栈——每个栈实例中都内置有互斥锁，但是对栈上数据项的操作就需要调用用户提供的代码。虽然，可以添加一些约束，对存储在栈中的数据项的任何操作都不应对栈本身执行任何操作。这就给栈的用户带来了负担，但是存储在容器中的数据访问该容器的情况并不常见，而且这种情况在发生时非常明显，因此这也不是一个特别难以承受的负担。

其他情况下，这就没那么简单了，正如看到的 3.2.4 节中的交换操作。至少在这种情况下你可能同时锁住多个互斥锁但又不是总能做的到。回看 3.1 节中那个链表例子时，一种可能的保护链表的方式是让每个节点一个互斥锁。然后，为了访问链表，线程必须获取他们感兴趣节点上的互斥锁。对一个要删除数据项的线程而言，它必须获取三个节点上的互斥锁：将要删除的节点，两边的邻接节点，因为它们也会被修改。同样的，为了遍历链表，当线程获得序列中下一个节点的锁时，它必须保持当前节点上的锁，以确保指向下一个节点的指针不会同时被修改。一旦下一个节点上的锁被获取，那么第一个节点的锁就可以释放了，因为没有持有它的必要了。

这种交叉（hand-over-hand）的锁风格允许多个线程访问链表，假设每个线程访问不同的节点。但是，为了避免死锁，节点必须以相同的顺序上锁：如果两个线程试图用相反的顺序使用交叉锁遍历链表，它们在链表的中间可能死锁。当节点 A 和 B 在链表中相邻，单向运行的线程将尝试持有节点 A 上的锁并试图获取节点 B 上的锁。另一个线程可能已经获取了节点 B 上的锁，并且试图获取节点 A 上的锁——经典的死锁场景，如图 3.2 所示。

当删除 A、C 节点中间的 B 节点时，如果这个线程在获取 A 和 C 的锁之前，获取了 B 节点上的锁，它潜在的会和一个遍历线程造成死锁。这样的线程可能会首先试图锁住 A 节点或 C 节点(依赖于遍历的方向)，但是后面就会发现，它无法获得 B 上的锁，因为线程在执行删除任务的时候，已经获取了 B 上的锁，并且同时试图获取 A 和 C 上的锁。

一种避免死锁的方法是定义遍历的顺序，因此一个线程必须先锁住 A 才能获取 B 的锁，在锁住 B 之后才能获取 C 的锁。这将消除死锁发生的可能性，代价就是不允许反向遍历链表。类似的套路经常也可以用于别的数据结构。

| 线程1 | 线程2 |
| --- | --- |
| 锁住主入口的互斥量 | |
| 读取头结点指针 | |
| 锁住头结点互斥量 | |
| 解锁主入口互斥量 | |
| | 锁住主入口互斥量 |
| 读取head->next指针 | 锁住尾结点互斥量 |
| 锁住next结点的互斥量 | 读取tail->prev指针 |
| 读取next->next指针 | 解锁尾结点的互斥量 |
| ... | ... |
| 锁住A结点的互斥量 | 锁住C结点的互斥量 |
| 读取A->next指针(也就是B结点) | 读取C->prev指针(也就是B结点) |
| | 锁住B结点互斥量 |
| 阻塞，尝试锁住B结点的互斥量 | 解锁C结点互斥量 |
| | 读取B->prev指针(也就是A结点) |
| | 阻塞，尝试锁住A结点的互斥量 |
| 死锁! | |

图 3.2 线程以相反顺序遍历链表造成的死锁

**使用锁的层次结构**

尽管这是一种定义锁顺序的特殊情况，锁层次结构可以提供一种在运行时检查是否遵守约定的方法。其思想是将应用程序划分为多个层，并识别任何给定层中可能锁住的所有互斥锁。当代码试图对一个互斥锁上锁，在该锁已被低层持有时，上锁是不允许的。可以在运行时检查这个问题，方法是为每个互斥对象分配层编号，并记录每个线程锁定了哪些互斥锁。这是一种常见的模式，但是 C++标准库没有提供对它的直接支持，因此你需要编写自定义的 hierarchical_mutex 互斥锁类型，其代码如清单 3.7 所示。

清单 3.7 使用层次锁来避免死锁

```
hierarchical_mutex high_level_mutex(10000); // 1
hierarchical_mutex low_level_mutex(5000);  // 2
hierarchical_mutex other_mutex(6000); // 3

int do_low_level_stuff();

int low_level_func()
{
  std::lock_guard<hierarchical_mutex> lk(low_level_mutex); // 4
  return do_low_level_stuff();
}

void high_level_stuff(int some_param);

void high_level_func()
{
  std::lock_guard<hierarchical_mutex> lk(high_level_mutex); // 6
  high_level_stuff(low_level_func()); // 5
}

void thread_a()  // 7
{
  high_level_func();
}

void do_other_stuff();

void other_stuff()
{
  high_level_func();  // 10
  do_other_stuff();
}

void thread_b() // 8
{
  std::lock_guard<hierarchical_mutex> lk(other_mutex); // 9
  other_stuff();
}
```

这段代码有三个 hierarchical_mutex 实例(①，②和③)，通过以逐步递减的层级编号进行构造。机制是这么定义的：如果你在 hierarchical_mutex 上持有锁，那么你只能获得

另一个具有更低层编号的 hierarchical_mutex 上的锁，这就对代码的功能施加了限制。

假设 do_low_level_stuff 没有上锁任何互斥锁，low_level_func 为层级最低的函数，并且会对 low_level_mutex④上锁。high_level_func 调用 low_level_func⑤的同时，也持有 high_level_mutex⑥上的锁，这也没什么问题，因为 high_level_mutex(①：10000)要比 low_level_mutex(②：5000)更高级。

thread_a()⑦遵守规则，所以它运行的很好。

另一方面，thread_b()⑧无视规则，因此在运行时肯定会失败。

首先，它锁住了 other_mutex⑨，这个互斥锁的层级编号只有 6000③。这就意味着，中层级的数据已被保护。当 other_stuff()调用 high_level_func()⑧时，就违反了层级结构：high_level_func()试图获取 high_level_mutex，这个互斥锁有一个 10000 的值，要比当前层级值 6000 要大。因此 hierarchical_mutex 将会报告一个错误，可能是抛出一个异常，或终止程序。在层级互斥锁上不可能死锁，因为互斥锁本身强制了锁顺序。这意味着，如果两个锁在层次结构中的层级相同，则不能同时持有它们，所以交叉锁的机制要求链中的每个互斥锁都有一个比前一个更低的层次值，这在某些情况下是不切实际的。

例子也展示了另一点，std::lock_guard<>模板与用户定义的互斥锁类型一起使用。虽然 hierarchical_mutex 不是 C++标准的一部分，但是它写起来很容易；一个简单的实现在清单 3.8 中展示。尽管它是一个用户定义类型，它可以用于 std::lock_guard<>模板中，因为它实现了三个满足互斥锁概念的成员函数：lock()，unlock() 和 try_lock()。虽然你还没见过 try_lock()怎么使用，但其实很简单：当互斥锁上的锁被另一个线程持有，它将返回 false，而不是等待到调用线程能获取互斥锁为止。std::lock()也可以在内部使用它，作为死锁避免算法的一部分。

清单 3.8 简单的层级互斥锁实现

```
class hierarchical_mutex
{
  std::mutex internal_mutex;

  unsigned long const hierarchy_value;
  unsigned long previous_hierarchy_value;

  static thread_local unsigned long this_thread_hierarchy_value;  // 1

  void check_for_hierarchy_violation()
  {
    if(this_thread_hierarchy_value <= hierarchy_value)  // 2
    {
      throw std::logic_error("mutex hierarchy violated");
    }
  }
```

```
  void update_hierarchy_value()
  {
    previous_hierarchy_value=this_thread_hierarchy_value;  // 3
    this_thread_hierarchy_value=hierarchy_value;
  }

public:
  explicit hierarchical_mutex(unsigned long value):
      hierarchy_value(value),
      previous_hierarchy_value(0)
  {}

  void lock()
  {
    check_for_hierarchy_violation();
    internal_mutex.lock();  // 4
    update_hierarchy_value();  // 5
  }

  void unlock()
  {
    if(this_thread_hierarchy_value!=hierarchy_value)
      throw std::logic_error("mutex hierarchy violated");  // 9
    this_thread_hierarchy_value=previous_hierarchy_value;  // 6
    internal_mutex.unlock();
  }

  bool try_lock()
  {
    check_for_hierarchy_violation();
    if(!internal_mutex.try_lock())  // 7
      return false;
    update_hierarchy_value();
    return true;
  }
};
thread_local unsigned long
     hierarchical_mutex::this_thread_hierarchy_value(ULONG_MAX);  // 8
```

hierarchical_mutex 的实现使用了 thread_local 的变量来存储当前线程的层级值。这个值可以被所有的互斥实例访问，但每个线程都有一个不同的值。这允许代码分别检查每个线程的行为，并且每个互斥锁的代码可以检查是否允许当前线程锁定该互斥锁。

这里的关键是使用了 thread_local 值来表达当前线程的层级值

this_thread_hierarchy_value①。它被初始化为最大值⑧，所以最初任何互斥锁都可以被锁住。因为其声明中有 thread_local，所以每个线程都有其自己的副本，因此，一个线程中变量的状态完全独立于从另一个线程读取到的变量的状态。参见附录 A，A.8 节，有更多与 thread_local 相关的内容。

所以，第一次线程锁住一个 hierarchical_mutex 时，this_thread_hierarchy_value 的值是 ULONG_MAX。本质上，这个值会大于其他任何值，所以 check_for_hierarchy_vilation()②的检查会通过。检查完成后，lock()委托内部互斥锁进行锁定④。一旦成功锁住，你可以更新层级值了⑤。

如果你现在在持有第一个互斥锁的同时锁住了另一个 hierarchical_mutex，this_thread_hierarchy_value 的值反映了第一个互斥锁的层级值。第二个互斥锁的层级值必须小于已经持有的互斥锁检查②才能通过。

现在，保存当前线程的层级值的前一个值很重要，这样你就可以在 unlock()⑥中恢复它；否则，即使线程不持有任何锁，也永远无法再用更高层次的值锁定互斥锁。因为只有当持有 internal_mutex③的时候，才保存之前的层级值，并且在解锁内部互斥锁⑥之前还原它的层级值，你可以安全的把它存储在 hierarchical_mutex 自身，因为 hierarchical_mutex 被内部互斥锁保护着。为了避免无序解锁造成层次结构混乱，当解锁的互斥锁不是最近上锁的那个互斥锁，就需要抛出异常⑨。其他机制也能做到这点，但这个是最简单的。

try_lock()与 lock()的功能相似，除了在调用 internal_mutex 的 try_lock()⑦失败时，不能拥有对应锁，所以不必更新层级值，并直接返回 false。

虽然是运行时检测，但是它至少不依赖时序——不必去等待导致死锁出现的罕见条件。以这种方式划分应用程序和互斥锁的设计过程可以帮助消除许多可能导致死锁的原因，甚至在它们被编写之前。所以，即使你不需要编写运行时检查，执行这个设计练习也是值得的。

**将这些指南扩展到锁之外**

如我在本节开头提到的那样，死锁不仅仅会发生在锁之间；任何可能导致循环等待的同步构造都可能发生这种情况，因此，将这些指南扩展到这些情况也是值得的，例如，正如你应该尽可能避免获取嵌套锁一样，在持有锁的情况下等待一个线程不是一个好主意，因为该线程可能需要获得锁才能继续执行。类似的，如果你要等待一个线程完成，那么确定一个线程层次结构可能是值得的，这样，一个线程只等待层次结构中更低的线程。一种简单的方法是确保线程在启动它们的同一个函数中进行连接，如同在 3.1.2 节和 3.3 节中描述的那样。

一旦你设计了避免死锁的代码，std::lock()和 std::lock_guard 就可以处理大多数简单锁的情况，但是有时需要更多的灵活性。在这种情况下，标准库提供了 std::unique_lock 模板。与 std::lock_guard 一样，这是一个互斥锁类型参数化的类模板，它也提供了与 std::lock_guard 相同的 RAII 风格的锁管理，但具有更大的灵活性。

### 3.2.6 灵活的 std::unique_lock 锁

通过放松不变量，std::unique_lock 比 std::lock_guard 提供了更多的灵活性；一个 std::unique_lock 实例并不总是拥有与其关联的互斥锁。首先，正如你可以将 std::adopt_lock 作为第二个参数传递给构造函数，让锁对象管理互斥锁一样，你也可以将 std::defer_lock 作为第二个参数传递给构造函数，以表明互斥锁在构造时应该保持解锁状态。然后，可以通过调用 std::unique_lock 对象（不是互斥锁）上的 lock()或者将 std::unique_lock 对象传递给 std::lock()来获取锁。清单 3.6 可以很容易的重写为清单 3.9，使用 std::unique_lock 和 std::defer_lock①，而非 std::lock_guard 和 std::adopt_lock。代码长度相同，几乎等价，唯一不同的就是：std::unique_lock 会占用比较多的空间，并且比 std::lock_guard 稍慢一些。允许 std::unique_lock 实例不拥有互斥锁的灵活性是要付出代价，这个代价就是需要存储这个信息，同时必要的时候需要更新这个信息。

清单 3.9 交换操作中 std::lock()和 std::unique_lock 的使用

```
class some_big_object;
void swap(some_big_object& lhs,some_big_object& rhs);
class X
{
private:
  some_big_object some_detail;
  std::mutex m;
public:
  X(some_big_object const& sd):some_detail(sd){}
  friend void swap(X& lhs, X& rhs)
  {
    if(&lhs==&rhs)
      return;
    std::unique_lock<std::mutex> lock_a(lhs.m,std::defer_lock); // 1
    std::unique_lock<std::mutex> lock_b(rhs.m,std::defer_lock); // 1 st
d::defer_lock 让互斥锁保持未上锁状态
    std::lock(lock_a,lock_b); // 2 互斥锁在这里上锁
    swap(lhs.some_detail,rhs.some_detail);
  }
};
```

在清单 3.9 中，因为 std::unique_lock 提供了 lock()，try_lock()和 unlock()成员函数，所以能将 std::unique_lock 对象传递到 std::lock()②。它们转发给底层互斥锁上同名的成员函数来执行任务，并在 std::unique_lock 实例中更新一个标志，以指示该互斥锁当前是否属于该实例，这个标志是为了确保 unlock()在析构函数中被正确调用。如果实例拥有互斥锁，那么析构函数必须调用 unlock()；但当实例中不拥有互斥锁时，析构函数就不能去调用 unlock()。这个标志可以通过 owns_lock()成员函数进行查询。除非你要转移锁的所有权或者做其他需要 std::unique_lock 的事情，否则如果 C++ 17 可变参数的 std::scoped_lock 对你可用的话，你最好使用它(详见 3.2.4 节)。

如你所料，这个标志必须存储在某个地方。因此，std::unique_lock 对象的体积通常要比 std::lock_guard 对象大，当使用 std::unique_lock 替代 std::lock_guard，因为会对标志进行适当的更新或检查，这会有轻微的性能惩罚。当 std::lock_guard 已经能够满足你的需求时，还是建议你优先使用它。也就是说，在某些情况下，std::unique_lock 会更适合手头的任务，因为你需要利用额外的灵活性。一个例子是你已经见过的延迟上锁；另一种情况是锁的所有权需要从一个作用域转移到另一个作用域。

## 3.2.7 在作用域间转移互斥锁的所有权

因为 std::unique_lock 实例不一定要拥有与之相关的互斥锁，一个互斥锁的所有权可以通过移动操作，在不同的实例中进行转移。某些情况下，这种转移是自动发生的，例如：当函数返回一个实例；另一些情况下，需要显式的调用 std::move()来实现。从本质上来说，需要依赖于源值是否是左值——一个实际的值或是引用——或一个右值——某种临时值。当源值是一个右值，所有权转移是自动的，为了避免意外从一个变量转移所有权，对左值就必须显式完成。std::unique_lock 是可移动但不可复制类型的一个例子。附录 A，A.1.1 节有更多与移动语义相关的信息。

一种可能的用法是允许函数锁住互斥锁并将该锁的所有权转移给调用者，这样调用者就可以在同一锁的保护下执行其他操作。下面的代码片段展示了一个示例；get_lock()函数锁定互斥锁，然后在将锁返回给调用者之前准备数据。

```
std::unique_lock<std::mutex> get_lock()
{
  extern std::mutex some_mutex;
  std::unique_lock<std::mutex> lk(some_mutex);
  prepare_data();
  return lk;  // 1
}
void process_data()
{
  std::unique_lock<std::mutex> lk(get_lock());  // 2
  do_something();
}
```

因为 lk 是在函数中声明的自动变量，所以它可以直接返回①，不需要调用 std::move()；编译器会调用移动构造函数。process_data()函数然后可以直接将所有权转移到它自己的 std::unique_lock 实例中②，对 do_something()的调用可以依赖于正确准备的数据，因为不会有另一个线程同时在修改数据。

通常在这种模式下，要上锁的互斥锁依赖于程序的当前状态，或者依赖于传递给返回 std::unique_lock 对象的函数的参数。其中一种用法是，锁不直接返回，它是网关类的数据成员，用于确保对某些受保护数据能正确上锁。这种情况下，所有的访问都必须通过网关类：当你想要访问数据，需要获取网关类的实例(如同前面的例子，通过调用 get_lock()之类函数)来获取锁。之后你就可以通过网关类的成员函数访问数据。当完成访问，可以销毁

这个网关类对象，它将释放锁，并且允许别的线程访问保护的数据。这样的一个网关类可能是可移动的(所以他可以从一个函数进行返回)，这种情况下锁对象的数据成员也必须可移动。

std::unique_lock 的灵活性同样也允许实例在销毁之前放弃其拥有的锁。可以使用unlock()成员函数来做这件事，就像互斥锁：std::unique_lock 支持与互斥锁相同的用于上锁和解锁的基本成员函数集，因此它可以与 std::lock 这样的泛型函数一起使用。在std::unique_lock 实例被销毁之前释放锁的能力意味着，如果明显不再需要该锁，你可以在特定的代码分支中选择性地释放它。这对于应用程序的性能来说很重要；持有锁的时间超过所需时间可能会导致性能下降，因为等待该锁的其他线程被阻止继续执行的时间长于必要的等待时间。

## 3.2.8 在适当的粒度上锁

在 3.2.3 节中，已经提过锁的粒度：锁粒度是描述单个锁保护的数据量的术语。细粒度锁保护少量数据，而粗粒度锁保护大量数据。不仅选择足够粗的锁粒度以确保所需数据得到保护很重要，而且确保锁只用于需要它的操作也是很重要的。我们都知道，在超市选了一车的杂货并等待结账的时候，正在结账的顾客突然意识到他忘了拿蔓越莓酱，然后离开柜台去找，并让其他的人等待他回来；或者当收银员，准备收钱时，顾客才去翻钱包拿钱，这样的情况都会让人很沮丧。如果每个人都检查好自己要拿的东西，并且准备好合适的支付方式，那么每件事都会进行得很顺利。

同样的道理也适用于线程：如果很多线程正在等待同一个资源(在收银台的收银员)，当有线程持有锁的时间过长，这就会增加等待的时间(别等到到收银台结账的时候，才去找蔓越莓酱)。在可能的情况下，只在访问共享数据时锁定互斥锁；尝试在锁之外对数据进行任何处理。尤其是在持有锁时不要做任何耗时的活动，如文件 I/O。文件 I/O 通常要比从内存中读或写同样量的数据慢成百上千倍，除非锁的目的是保护对文件的访问，否则在持有锁时执行 I/O 将不必要地耽搁其他线程（因为等待获取锁的时候，它们全被阻塞住了），这没有必要(因为文件锁阻塞住了很多操作)，潜在抵消了多线程带来的性能增益。

在这种情况下，std::unique_lock 工作得很好，因为你可以在代码不再需要访问共享数据时调用 unlock()，然后如果之后代码需要获取锁的话，再调用 lock()：

```
void get_and_process_data()
{
  std::unique_lock<std::mutex> my_lock(the_mutex);
  some_class data_to_process=get_next_data_chunk();
  my_lock.unlock();  // 1 不需要跨 process()函数锁住互斥锁
  result_type result=process(data_to_process);
  my_lock.lock(); // 2 为了写入数据，对互斥锁再次上锁
  write_result(data_to_process,result);
}
```

不需要跨 process()函数锁住互斥锁，所以可以在函数调用①前对互斥锁手动解锁，并

且在之后对其再次上锁②。

希望这是显而易见的：如果有一个互斥锁保护整个数据结构，那么不仅可能会有更多的锁争用，而且减少持有锁的时间的可能性也会减小。更多的操作步骤将需要对同一个互斥锁上锁，因此锁的持有时间必然更长。这种成本的双重打击也是尽可能采用更细粒度上锁的双动力。

如本例所示，适当粒度的锁不仅仅涉及锁住的数据量；它还与持有锁的时间有关，以及在持有锁时执行了哪些操作。通常，锁的持有时间应该是执行所需操作花费的最短时间。这也意味着，除非绝对必要，持有锁时不应该执行诸如获取另一个锁(即使你知道它不会死锁)或等待 I/O 完成等耗时的操作。

清单 3.6 和 3.9 中，交换操作需要锁住两个互斥锁，其明确要求并发访问两个对象。假设用来做比较的是一个简单的数据类型(比如：int 类型)，这有什么不同吗？int 的复制成本很低，所以你可以轻松地复制被比较的每个对象的数据，同时只持有该对象对应的锁，然后比较复制的值。这意味着你持有每个互斥锁的时间是最短的，而且你不是在持有一个锁的同时锁住另一个锁。下面的清单显示了这样一个类 Y，以及相等比较运算符的示例实现。

清单 3.10 比较操作符中一次锁住一个互斥锁

```
class Y
{
private:
  int some_detail;
  mutable std::mutex m;
  int get_detail() const
  {
    std::lock_guard<std::mutex> lock_a(m);  // 1
    return some_detail;
  }
public:
  Y(int sd):some_detail(sd){}

  friend bool operator==(Y const& lhs, Y const& rhs)
  {
    if(&lhs==&rhs)
      return true;
    int const lhs_value=lhs.get_detail();  // 2
    int const rhs_value=rhs.get_detail();  // 3
    return lhs_value==rhs_value;  // 4
  }
};
```

例子中，比较操作符首先通过调用 get_detail()成员函数检索要比较的值②③，函数在索引值时被一个锁保护着①。比较操作符会在之后比较索引出来的值④。注意：虽然这样

能减少锁持有的时间，一个锁只持有一次(并且消除了死锁的可能性)，但相比一起持有两个锁，这改变了操作的语义。清单 3.10 中，当操作符返回 true 时，那就意味着在这个时间点上的 lhs.some_detail 与在另一个时间点的 rhs.some_detail 相同。这两个值在读取之后，可能会被任意的方式所修改；在②和③之间这些可以被交换，例如，让比较失去意义。相等比较可能返回 true 来表明值是相等的，即使值从来没有在某一瞬间是相等的。因此，在进行这些更改时，一定要小心，不要以有问题的方式更改操作的语义：如果在操作的整个过程中没有持有所需的锁，就会将自己暴露于竞争条件中。

有时没有一个合适的粒度级别,因为并不是所有对数据结构的访问都需要同一级的保护。这个例子中，就需要寻找一个合适的机制，去替换简单的 std::mutex。

---

[1] Tom Cargill, “Exception Handling: A False Sense of Security,” in C++ Report 6, no. 9 (November－December 1994). Also available at http://www.informit.com/content/images/020163371x/supplements/Exception_Handling_Article.html.

[2] Herb Sutter, Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions (Addison Wesley Pro-fessional, 1999).

# 3.3 保护共享数据的替代设施

尽管互斥锁是最常见的机制，但在保护共享数据方面，互斥锁并不是唯一的策略；有一些替代方案可以在特定场景中提供更适当的保护。

一个特别极端(但非常常见)的情况是,共享数据只在初始化的时候才需要并发访问的保护，但之后就不需要显式同步了。这可能是因为数据一旦创建就是只读的，因此不存在可能的同步问题，也可能是因为必要的保护是作为对数据的操作的一部分隐式执行的。不论何种情况，在数据初始化之后锁住互斥锁，纯粹只为了保护初始化，是不必要的，同时对性能造成不必要的影响。正是由于这个原因，C++标准提供了一种机制，纯粹用于在初始化期间保护共享数据。

## 3.3.1 初始化期间保护共享数据

假设你有一个共享资源，构建代价很昂贵，你只想在确实需要的时候才构建它；它可能会打开一个数据库连接或分配出很多的内存。*延迟初始化*(Lazy initialization)在单线程代码很常见——需要资源的每个操作首先检查它是否已初始化，如果没有，则在使用之前初始化它：

```
std::shared_ptr<some_resource> resource_ptr;
void foo()
{
```

```
  if(!resource_ptr)
  {
    resource_ptr.reset(new some_resource);  // 1
  }
  resource_ptr->do_something();
}
```

如果共享资源本身对并发访问是安全的，转为多线程代码时，只有①处初始化需要保护，但是像下面清单中那样简单的转换可能会导致使用该资源的线程不必要的串行化。这是因为每个线程必须等待互斥锁，以便检查资源是否已经初始化。

清单 3.11 使用互斥锁的线程安全的延迟初始化

```
std::shared_ptr<some_resource> resource_ptr;
std::mutex resource_mutex;

void foo()
{
  std::unique_lock<std::mutex> lk(resource_mutex);  // 所有线程在此串行化
  if(!resource_ptr)
  {
    resource_ptr.reset(new some_resource);  // 只有初始化过程需要保护
  }
  lk.unlock();
  resource_ptr->do_something();
}
```

这段代码很常见，不必要的串行化问题已经够多了，许多人试图想出一个更好的办法来做这件事，包括声名狼藉的“双重检查加锁”模式：第一次读取指针不需要获取锁（①在下面的代码中），并且只有在指针为 NULL 时才获取锁。然后，当获取锁之后，指针会被再次检查一遍（②这就是双重检查的部分），以防在第一次检查和这个线程获得锁之间，另一个线程已经完成了初始化。

```
void undefined_behaviour_with_double_checked_locking()
{
  if(!resource_ptr)  // 1
  {
    std::lock_guard<std::mutex> lk(resource_mutex);
    if(!resource_ptr)  // 2
    {
      resource_ptr.reset(new some_resource);  // 3
    }
  }
  resource_ptr->do_something();  // 4
```

```
}
```

不幸的是,这种模式之所以臭名昭著,有一个原因:它可能会出现令人讨厌的竞争条件,因为读取是在锁之外进行的①,没有与另一个线程里被锁保护的写入操作③进行同步,这创建了一个竞争条件,它不仅包含指针本身,还包含指向的对象;即使一个线程看到了另一个线程写的指针,它也可能看不到新创建的 some_resource 实例,从而导致调用 do_something()④操作不正确的值。这是竞争条件类型的一个示例,C++标准将其定义为数据竞争(data race),并且规定为未定义的行为。因此,这绝对是要避免的。有关内存模型的详细讨论,请参阅第 5 章,其中包含了什么构成了数据竞争。

C++标准委员会也认为这是一个重要的场景,因此 C++标准库提供了 std::once_flag 和 std::call_once 来处理这种情况。不同于锁住互斥锁并显式地检查指针,每个线程都可以使用 std::call_once,因为知道当 std::call_once 返回时,指针已经被某个线程(以适当同步的方式)初始化了,所以是安全的。必要的同步数据存储在 std::once_flag 实例中;once_flag 的每个实例对应一个不同的初始化。使用 std::call_once 比显式使用互斥锁消耗的资源更少,特别是当初始化完成后,因此,当它与所需功能相匹配时,应该优先使用它。下面的例子展示了与清单 3.11 中的同样的操作,这里使用了 std::call_once。在这种情况下,初始化通过调用函数完成,这样的操作使用类中的函数操作符来实现同样很简单。与标准库中接受函数或谓词作为参数的大多数函数一样,std::call_once 可以和任何函数或可调用对象一起使用。

```
std::shared_ptr<some_resource> resource_ptr;
std::once_flag resource_flag;  // 1

void init_resource()
{
  resource_ptr.reset(new some_resource);
}

void foo()
{
  std::call_once(resource_flag,init_resource);  // 初始化只会调用一次
  resource_ptr->do_something();
}
```

这个例子中,std::once_flag①和初始化好的数据都是命名空间作用域的对象,但 std::call_once()可以很容易用作延迟初始化的类成员,如同下面的清单:

清单 3.12 使用 std::call_once 作为类成员的线程安全的延迟初始化

```
class X
{
private:
  connection_info connection_details;
  connection_handle connection;
  std::once_flag connection_init_flag;
```

```
  void open_connection()
  {
    connection=connection_manager.open(connection_details);
  }

public:
  X(connection_info const& connection_details_):
      connection_details(connection_details_)
  {}

  void send_data(data_packet const& data)  // 1
  {
    std::call_once(connection_init_flag,&X::open_connection,this);  //
2
    connection.send_data(data);
  }

  data_packet receive_data()  // 3
  {
    std::call_once(connection_init_flag,&X::open_connection,this);  //
2
    return connection.receive_data();
  }
};
```

例子中第一次调用 send_data()①或 receive_data()③的线程完成初始化。使用成员函数 open_connection()去初始化数据，也需要将 this 指针传进去。就像标准库中接受可调用对象的其他函数一样，比如 std::thread 和 std::bind()的构造函数，这是通过向 std::call_once()传递额外的参数来完成的②。

值得注意的是，std::mutex 和 std::once_flag 的实例不能拷贝和移动，因此，如果像这样使用它们作为类成员，则必须显式地定义这些特殊的成员函数，可能你需要它们。

还有一种初始化过程中潜存着条件竞争：其中一个局部变量被声明为 static 类型。该变量的初始化被定义为在控制第一次通过声明时发生；对于调用该函数的多个线程，这意味着谁算是第一次有潜在的竞争条件。很多在 C++11 标准之前的编译器上，在实践过程中，这样的竞争条件是有问题的，因为多个线程可能认为他们是第一次并尝试初始化变量，或者线程可能在另一个线程上的初始化已经开始但在它完成之前尝试使用它。在 C++11 标准中，这些问题都被解决了：初始化只会在一个线程中发生，并且没有其他线程可在初始化完成前继续执行，所以竞争条件是关于哪个线程可以执行初始化，而不是其他问题。在只需要一个全局实例情况下，这里提供一个 std::call_once 的替代方案。

```
class my_class;
my_class& get_my_class_instance()
```

```
{
  static my_class instance;  // 初始化保证是线程安全的
  return instance;
}
```

多线程可以安全的调用 get_my_class_instance()①，不用担心初始化上的竞争条件。

仅在初始化时保护数据是更一般场景的一种特殊情况：很少更新的数据结构。大多数情况下，这种数据结构是只读的，并且多线程可以对其并发的读取，但有时数据结构可能需要更新。这里需要的是一种承认这一事实的保护机制。

## 3.3.2 保护很少更新的数据结构

考虑一个用于存储 DNS 条目缓存的表，用于将域名解析为其对应的 IP 地址。通常，给定的 DNS 条目将在很长一段时间内保持不变——在许多情况下，DNS 条目多年保持不变。虽然随着用户访问不同的网站，可能会不时地向表中添加新的条目，但这些数据在其整个生命周期中基本保持不变。定期检查缓存条目的有效性是很重要的，但这仍然需要在细节发生改变时进行更新。

虽然更新频度很低，但更新也有可能发生，并且当这个缓存可被多个线程访问时，它需要在更新期间得到适当的保护，以确保没有读取缓存的线程看到损坏的数据结构。

缺乏一种专用数据结构来匹配类似场景，这种数据结构是为并发更新和读取特别设计的(如第 6 和第 7 章中)，这样的更新要求线程独占数据结构的访问权，直到其完成操作。当更新完成，数据结构对于并发多线程访问又会是安全的。使用 std::mutex 来保护数据结构，显得用力过猛，因为在没有发生修改时，它将削减并发读取数据的可能性。这里需要另一种不同的互斥锁，这种互斥锁常被称为“读者-写者”锁（读写锁），因为它允许两种不同的用法：一个“写者”线程独占访问，让多个“读者”线程并发访问。

C++ 17 标准库提供了两个现成的互斥锁 std::shared_mutex 和 std::shared_timed_mutex。C++14 只提供了 std::shared_timed_mutex，但是在 C++11 中什么都没有。如果你还在用支持 C++14 标准之前的编译器，那你可以使用 Boost 库中实现的互斥锁，它是基于最初的提案。std::shared_mutex 和 std::shared_timed_mutex 的不同点在于，std::shared_timed_mutex 支持更多的操作方式(参考 4.3 节)，所以如果你不需要额外操作的话，std::shared_mutex 可能提供更高的性能优势。

你将在第 8 章中看到，这种锁也不是灵丹妙药，并且性能取决于所涉及的处理器数量以及读线程和更新线程的相对工作负载。因此，分析目标系统上代码的性能是很重要的，以确保额外的复杂性能带来收益。

比起使用 std::mutex 实例进行同步，不如使用 std::shared_mutex。对于更新操作，可以使用 std::lock_guard<std::shared_mutex>和 std::unique_lock<std::shared_mutex>上锁。这些保证了独占访问，就像 std::mutex 一样。那些不需要更新数据结构的线程可以使用 std::shared_lock<std::shared_mutex>来

获得共享访问。这个 RAII 类模板是在 C++14 中添加的，使用方法与 std::unique_lock 相同，只是多个线程可以在同一个 std::shared_mutex 上同时拥有一个共享锁。唯一的限制是，如果任何线程持有一个共享锁，试图获取独占锁的线程将阻塞,直到所有其他线程让出它们的锁，同样，如果任何线程拥有互斥锁，没有其他线程可以获得一个共享或互斥锁直到第一个线程让出它的锁。

下面的清单展示了一个简单的 DNS 缓存，使用 std::map 持有缓存的数据，使用 std::shared_mutex 进行保护。

清单 3.13 使用 std::shared_mutex 保护数据结构

```
#include <map>
#include <string>
#include <mutex>
#include <shared_mutex>

class dns_entry;

class dns_cache
{
  std::map<std::string,dns_entry> entries;
  mutable std::shared_mutex entry_mutex;
public:
  dns_entry find_entry(std::string const& domain) const
  {
    std::shared_lock<std::shared_mutex> lk(entry_mutex);  // 1
    std::map<std::string,dns_entry>::const_iterator const it=
       entries.find(domain);
    return (it==entries.end())?dns_entry():it->second;
  }
  void update_or_add_entry(std::string const& domain,
                           dns_entry const& dns_details)
  {
    std::lock_guard<std::shared_mutex> lk(entry_mutex);  // 2
    entries[domain]=dns_details;
  }
};
```

在清单 3.13 中，find_entry()使用 std::shared_lock<>的一个实例来保护自己，以便进行共享的只读访问①；这就使得多线程可以同时调用 find_entry()，且不会出错。另一方面，当表需要更新时②，update_or_add_entry()使用了 std::lock_guard<>实例，为其提供独占访问；在 update_or_add_entry()调用中，不仅其他线程被阻止进行更新，而且调用 find_entry()的线程也会被阻塞。

### 3.3.3 递归锁

对于 std::mutex，一个线程试图锁住它已经拥有的互斥锁是个错误，并且尝试这样做会导致未定义的行为。但在某些情况下，一个线程在没有首先释放互斥锁的情况下多次获得它是可取的。为此，C++标准库提供了 std::recursive_mutex。它的工作原理类似于 std::mutex，不同之处在于你可以从同一个线程获取单个实例上的多个锁。在互斥锁被另一个线程锁定之前，你必须释放所有的锁，所以如果你调用了三次 lock()，那你也必须调用三次 unlock()。正确使用 std::lock_guard<std::recursive_mutex>和 std::unique_lock<std::recursive_mutex>会为你处理这些。

大多数时候，如果你想要一个递归互斥，你可能需要改变你的设计。递归互斥锁的一种常见用法是，类被设计为可以从多个线程并发地访问，因此它有一个互斥锁来保护成员数据。每个公共成员函数锁住互斥锁，完成相关工作，然后解锁互斥锁。但有时，一个公共成员函数调用另一个成员函数作为其操作的一部分是可取的。在本例中，第二个成员函数也将尝试锁住互斥锁，从而导致未定义的行为。一种应急的解决方案是将互斥锁更改为递归互斥锁。这将允许第二个成员函数中的互斥锁成功上锁，并继续执行。

但不建议这样使用，因为这会导致草率的思考和糟糕的设计。特别地，类不变量通常在持有锁时被破坏，这意味着即使在不变量被破坏时调用第二个成员函数也需要工作。通常更好的方法是从两个成员函数中提取一个新的私有成员函数，它不会锁住互斥锁(它期望它已经被锁住)。然后可以仔细想想在什么情况下可以调用这个新函数，以及在这些情况下数据的状态

# 总结

在本章中，我讨论了在线程之间共享数据时，有问题的竞争条件是如何造成灾难性后果，以及如何使用 std::mutex，并精心设计接口来避免它们。你也看到了互斥锁并不是万能的，而且它们本身也有死锁的问题，虽然 C++标准库提供了一个工具，以 std::lock()的形式帮助避免这个问题。然后，你了解了更多避免死锁的技术，之后简要了解了转移锁的所有权以及围绕选择适当的锁粒度的议题。最后，介绍了为特定场景提供的替代数据保护工具，如 std::call_once()和 std::shared_mutex。

还有一个事情没有涉及，那就是等待其他线程的输入。你的线程安全栈，在栈为空的时候，抛出一个异常，所以如果一个线程想要等待另一个线程把一个值推入栈中的话 (这毕竟是线程安全栈的主要用途之一)，它需要不断尝试弹出一个值，当捕获到抛出的异常时，再次进行尝试。这种消耗资源的检查，没有任何意义。实际上，不断的检查会影响系统中其他线程的运行，反而阻碍了进展。我们需要某种方法让一个线程等待其他线程完成任务，而不是在进程中消耗 CPU 时间。第 4 章基于讨论过的保护共享数据的设施，然后介绍了在 C++线程间同步操作的各种机制；第 6 章中会展示，如何使用它们构建更大型的可复用的数据结构。

# 第 4 章 同步并发操作

**本章主要内容**

- 等待一个事件
- 用期望等待一次性事件
- 带时间限制的等待
- 使用操作的同步来简化代码

上一章中，我们看到各种在线程间保护共享数据的方法。但有时，你不仅需要保护数据，还需要同步不同线程上的操作。例如，一个线程可能需要等待另一个线程完成一个任务，然后第一个线程才能完成自己的任务。一般来说，通常希望线程等待特定事件发生或一个条件变为真。尽管可以通过定期检查共享数据中存储的“任务完成”标志或类似的东西来实现这一点，但这远不够理想。像这样需要在线程之间同步操作的场景是如此的常见，以至于 C++标准库提供了*条件变量*(condition variables)和*期望*（futures）形式的设施来处理它。这些设施在并发技术规范(TS，Conncurrency Technical Specification)中得到了扩展，技术规范为*期望*（futures）提供了更多的操作，一起的还有新的同步设施*锁存器*(latches)和*屏障*(barriers)。

本章将讨论如何使用条件变量，期望，锁存器以及屏障来等待事件，以及如何使用它们来简化操作的同步。

## 4.1 等待一个事件或其他条件

假设你乘坐通宵火车旅行。一种确保你在正确的车站下车的方法是整晚保持清醒，并注意火车停在哪里。这样你就不会误站，但是等你到站的时候估计也累够呛。或者，你可以看一下时刻表，看看火车应该什么时候到达，然后把闹钟定得比到站时间稍微早一点，接着就可以去睡觉了。这样就可以了；你也不会误站，但如果火车晚点，你就醒得太早了。当然，闹钟的电池也可能会没电了，于是你就睡过了头，以至于误了站。理想的方式是，你可以去睡觉，不管什么时候，只要火车到站，就有人或其他东西能把你唤醒就好了。

这和线程有什么关系呢？嗯，如果一个线程正在等待另一个线程完成一个任务，它有几个选择。首先，它可以不断检查共享数据中的标志(由互斥锁保护)，并让第二个线程在完成任务时设置该标志。这在两个方面是浪费的：线程不断检查标记会消耗宝贵的处理时间，并且当互斥锁被等待的线程锁住时，其他线程不能锁住它。这两者对等待线程都不利：如果等待线程在运行，这就限制了可用的执行资源去运行被等待的线程，同时为了检查标志，等待线程锁住了互斥锁来保护它，被等待线程就不能在它完成任务后锁住互斥锁来设置标志。这种情况类似于你整晚和列车驾驶员交谈：驾驶员不得不减慢火车的速度，因为你分散了他的注意力，所以火车需要更长的时间才能到站。类似地，正在等待的线程正在消耗系统中其他线程可以使用的资源，最终等待时间可能比必要的时间更长。

第二个选择是让等待线程在检查的间隙用 std::this_thread::sleep_for()函数休眠很短的时间(参见 4.3 节)：

```
bool flag;
std::mutex m;

void wait_for_flag()
{
  std::unique_lock<std::mutex> lk(m);
  while(!flag)
  {
    lk.unlock();  // 1 解锁互斥锁
    std::this_thread::sleep_for(std::chrono::milliseconds(100));  // 2
休眠 100ms
    lk.lock();   // 3 再次锁住互斥锁
  }
}
```

循环体中，在休眠前②，函数对互斥锁进行解锁①，并且在休眠结束后再对互斥锁进行上锁③，因此另外的线程就有机会获取锁并设置标志。

这是一个进步，因为当线程休眠时，线程没有浪费执行时间，但是很难确定正确的休眠时间。太短的休眠仍然会浪费处理时间去做检查；太长的休眠时间，会导致当被等待线程完成时，线程还处于休眠状态，从而导致耽搁。这种睡过头的情况很少会对程序的运行产生直接影响，但它可能意味着在快节奏的游戏中掉帧或在实时应用中超出时间片。

第三个也是首选的方法是使用 C++标准库提供的设施去等待事件本身。等待另一个线程触发事件的最基本机制(例如前面提到的在流水线中存在的额外工作)是*条件变量*（condition variable）。从概念上讲，条件变量与事件或其他*条件*（condition）相关联，一个或多个线程可以等待该条件满足。当一个线程确定满足条件时，它可以通知一个或多个等待条件变量的线程，以唤醒它们并允许它们继续处理。

## 4.1.1 使用条件变量等待条件

C++标准库对条件变量有两套实现：std::condition_variable 和 std::condition_variable_any。这两个实现都包含在<condition_variable>库头文件中。两者都需要与一个互斥锁一起才能工作，因为需要互斥锁提供适当的同步；前者仅限于使用 std::mutex，而后者可以使用任何满足类似于互斥锁的最低标准的对象，因而带有_any 后缀。由于 std::condition_variable_any 更通用，因此在大小、性能或操作系统资源方面有额外的潜在成本，所以除非需要额外的灵活性，否则应该首选 std::condition_variable。

那么，如何使用 std::condition_variable 来处理简介中的示例呢？如何让正在等待工作的线程休眠，直到有数据要处理？下面的清单展示了使用条件变量实现的一种方法。

清单 4.1 使用 std::condition_variable 等待需要处理的数据

```
std::mutex mut;
std::queue<data_chunk> data_queue;  // 1
std::condition_variable data_cond;

void data_preparation_thread()
{
  while(more_data_to_prepare())
  {
    data_chunk const data=prepare_data();
    {
      std::lock_guard<std::mutex> lk(mut);
      data_queue.push(data);  // 2
    }
    data_cond.notify_one();  // 3
  }
}

void data_processing_thread()
{
  while(true)
  {
    std::unique_lock<std::mutex> lk(mut);  // 4
    data_cond.wait(
         lk,[]{return !data_queue.empty();});  // 5
    data_chunk data=data_queue.front();
    data_queue.pop();
    lk.unlock();  // 6
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

首先，有一个用来在两个线程之间传递数据的队列①。当数据准备好时，准备数据的线程使用 std::lock_guard 来保护队列，并把数据推入队列中②。然后它调用 std::condition_variable 实例的 notify_one() 成员函数通知等待线程（如果有的话）③。注意，你把将数据推入队列的代码放在一个较小的作用域，所以你在解锁之后通知条件变量——这是为了，如果等待线程立即醒来，它没必要再被阻塞在等待你解锁互斥锁。

在栅栏的另一侧，有一个正在处理数据的线程，这个线程首先锁住互斥锁，但这次使用 std::unique_lock 而不是 std::lock_guard④——你马上就会知道为什么。然后线程在

std::condition_variable 上调用 wait()成员函数，并传入锁对象和表示等待条件的 lambda 函数⑤。Lambda 函数是 C++11 添加的新特性，它可以让一个匿名函数作为另一个表达式的一部分，并且它们非常适合被指定为 wait()这种标准库函数的谓词。在这个例子中，简单的 Lambda 函数[]{return !data_queue.empty();}会去检查 data_queue 是否非空——也就是说，队列中有数据准备要处理。附录 A 的 A.5 节有 Lambda 函数更多的细节。

wait()会去检查这些条件(通过调用所提供的 lambda 函数)，当条件满足(lambda 函数返回 true)时返回。如果条件不满足(lambda 函数返回 false)，wait()函数将解锁互斥锁，并且将这个线程置于阻塞或等待状态。当准备数据的线程调用 notify_one()通知条件变量时，处理数据的线程从睡眠状态中醒来，获取互斥锁上的锁，并且再次检查条件是否满足。在条件满足的情况下，从 wait()返回并仍然持有锁；当条件不满足时，线程将对互斥锁解锁，并且重新开始等待。这就是为什么用 std::unique_lock 而不使用 std::lock_guard——等待中的线程必须在等待期间解锁互斥锁，并在这之后对互斥锁再次上锁，而 std::lock_guard 没有这么灵活。如果互斥锁在线程休眠期间保持锁住状态，准备数据的线程将无法锁住互斥锁，也就无法添加数据项到队列中，这样等待线程也永远看不到它的条件被满足。

清单 4.1 为等待使用了一个简单的 lambda 函数⑤，它检查队列是否非空，不过任何函数和可调用对象都可以担此责任。如果已经有了检查条件的函数(可能因为它比像这样简单的测试要复杂一些)，那么可以直接传入此函数，不一定非要包在一个 lambda 中。在调用 wait()期间，条件变量可以对提供的条件检查任意次数；但是它总是在锁住互斥锁的情况下才这么做，并且当(且仅当)用于测试条件的函数返回 true 时，它将立即返回。当等待的线程重新获得互斥锁并检查条件时，如果它不是直接响应来自另一个线程的通知，则称为*伪唤醒*(spurious wakeup)。因为根据定义，任何这种伪唤醒的数量和频率都是不确定的，所以不建议使用具有副作用的函数进行条件检查。如果你这样做，你必须为副作用发生多次做好准备。

基本上，std::condition_variable::wait 是对忙-等待的优化。事实上，一个合格(虽然不太理想)的实现技术可以只是一个简单的循环：

```
template<typename Predicate>
void minimal_wait(std::unique_lock<std::mutex>& lk, Predicate pred)
{
  while(!pred()){
    lk.unlock();
    lk.lock();
  }
}
```

你的代码必须准备好不但能使用这种最小的 wait()实现，而且还能使用只有在调用 notify_one()或 notify_all()时才会唤醒的实现。

解锁 std::unique_lock 的灵活性，不仅适用于对 wait()的调用；它还可以用在数据待处理但还未处理的时候⑥。处理数据可能是一个耗时的操作，正如你在第 3 章中看到的，在互斥锁上持有的时间超过必要的时间不是一个好主意。

像清单 4.1 这样，使用队列在多个线程间转移数据是很常见的。如果做得好，同步可以限制在队列本身，这将极大地减少同步问题和竞争条件的可能数量。鉴于此，现在让我们从清单 4.1 中提取一个通用的线程安全队列

## 4.1.2 使用条件变量构建线程安全队列

如果你准备设计一个通用队列，花点时间想想队列需要哪些操作是值得的，就像在 3.2.3 节线程安全栈中做的一样。我们可以从 C++标准库中找灵感，形式为 std::queue<> 的容器适配器如下所示的：

清单 4.2 std::queue 接口

```
template <class T, class Container = std::deque<T> >
class queue {
public:
  explicit queue(const Container&);
  explicit queue(Container&& = Container());
  template <class Alloc> explicit queue(const Alloc&);
  template <class Alloc> queue(const Container&, const Alloc&);
  template <class Alloc> queue(Container&&, const Alloc&);
  template <class Alloc> queue(queue&&, const Alloc&);

  void swap(queue& q);

  bool empty() const;
  size_type size() const;

  T& front();
  const T& front() const;
  T& back();
  const T& back() const;

  void push(const T& x);
  void push(T&& x);
  void pop();
  template <class... Args> void emplace(Args&&... args);
};
```

如果忽略构造、赋值以及交换操作时，就只剩下了三组操作：查询整个队列的状态的操作(empty()和 size())；查询队列中元素的操作(front()和 back())；修改队列的操作(push()，pop()和 emplace())。这和 3.2.3 中的栈一样，因此也会遇到在接口上固有的竞争条件。所以，需要将 front()和 pop()合成一个函数调用，就像之前在栈实现时合并 top()和 pop()一样。清单 4.1 中的代码加入一些细微的变化：当使用队列在线程之间传递数据时，接收线程通常需要等待数据。这里提供 pop()函数的两个变种：try_pop()和

wait_and_pop()。try_pop()，尝试从队列中弹出数据，它总会直接返回(带有失败指示)，即使没有值可检索；wait_and_pop()，将会等到有值可检索的时候才返回。如果你以栈示例为指引，接口可能会是下面这样：

清单 4.3 threadsafe_queue 的接口

```
#include <memory> // 为了使用 std::shared_ptr

template<typename T>
class threadsafe_queue
{
public:
  threadsafe_queue();
  threadsafe_queue(const threadsafe_queue&);
  threadsafe_queue& operator=(
      const threadsafe_queue&) = delete;  // 不允许简单的赋值

  void push(T new_value);

  bool try_pop(T& value);  // 1
  std::shared_ptr<T> try_pop();  // 2

  void wait_and_pop(T& value);
  std::shared_ptr<T> wait_and_pop();

  bool empty() const;
};
```

和栈一样，为了简化代码，减少了构造函数并删除了赋值操作符。和之前一样，也提供了两个版本的 try_pop()和 wait_for_pop()。第一个重载的 try_pop()①把检索的值存储在引用变量中，所以它可以用返回值做状态；当检索到一个值时，它将返回 true，否则返回 false(参见 A.2 节)。第二个重载②就不能这样了，因为它直接返回检索到的值。不过，当没有值可检索时，这个函数可以返回 NULL 指针。

那么，所有这些与清单 4.1 有什么关系呢？嗯，你可以从中抽取代码用于 push()和 wait_and_pop()，如下面的清单所示。

清单 4.4 从清单 4.1 中提取 push()和 wait_and_pop()

```
#include <queue>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue
```

```
{
private:
  std::mutex mut;
  std::queue<T> data_queue;
  std::condition_variable data_cond;
public:
  void push(T new_value)
  {
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(new_value);
    data_cond.notify_one();
  }

  void wait_and_pop(T& value)
  {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    value=data_queue.front();
    data_queue.pop();
  }
};

threadsafe_queue<data_chunk> data_queue;  // 1

void data_preparation_thread()
{
  while(more_data_to_prepare())
  {
    data_chunk const data=prepare_data();
    data_queue.push(data);  // 2
  }
}

void data_processing_thread()
{
  while(true)
  {
    data_chunk data;
    data_queue.wait_and_pop(data);  // 3
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

互斥锁和条件变量现在包含在 threadsafe_queue 实例中，因此不再需要单独的变量①，并且调用 push()也不需要外部同步②。另外，wait_and_pop()负责条件变量的等待③。

另一个重载的 wait_and_pop()现在编写起来很简单，剩下的函数几乎可以逐字从清单 3.5 中的栈示例中拷贝。最终的队列实现展示如下。

清单 4.5 使用条件变量的线程安全队列的完整类定义

```
#include <queue>
#include <memory>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue
{
private:
  mutable std::mutex mut;  // 1 互斥锁必须是可变的
  std::queue<T> data_queue;
  std::condition_variable data_cond;
public:
  threadsafe_queue()
  {}
  threadsafe_queue(threadsafe_queue const& other)
  {
    std::lock_guard<std::mutex> lk(other.mut);
    data_queue=other.data_queue;
  }

  void push(T new_value)
  {
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(new_value);
    data_cond.notify_one();
  }

  void wait_and_pop(T& value)
  {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    value=data_queue.front();
    data_queue.pop();
  }
```

```
  std::shared_ptr<T> wait_and_pop()
  {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
  }

  bool try_pop(T& value)
  {
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
      return false;
    value=data_queue.front();
    data_queue.pop();
    return true;
  }

  std::shared_ptr<T> try_pop()
  {
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
      return std::shared_ptr<T>();
    std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
  }

  bool empty() const
  {
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
  }
};
```

尽管 empty()是一个 const 成员函数，并且拷贝构造函数的 other 参数是一个 const 引用，但是其他线程可能有对该对象的非 const 引用，并且可能正在调用可变的成员函数，因此你仍然需要锁住互斥锁。因为锁住互斥锁是一种可变操作，所以互斥锁对象必须标记为可变的(mutable)①，这样就可以在 empty()和拷贝构造函数中锁住它。

在多个线程等待同一事件时，条件变量也很有用。如果线程用于划分工作负载，因此只有一个线程应该响应通知，那么可以使用与清单 4.1 中所示完全相同的结构，只需运行多个数据处理线程实例。当新数据准备好时，调用 notify_one()将会触发一个正在执行 wait()的线程去检查它的条件并且从 wait()函数返回(因为你刚向 data_queue 中添加一个

数据项)。 不能保证哪个线程会被通知，甚至不能保证是否有线程在等待被通知，因为有可能所有的处理线程仍然在处理数据。

另一种可能是几个线程在等待同一事件，并且它们都需要响应该事件。这可能发生在共享数据初始化的情况下，所有的处理线程可以使用相同的数据，但是需要等待它被初始化(尽管可能有更好的机制，比如 std::call once；关于这个选项的讨论，请参阅第 3 章的 3.3.1 节)，或者线程需要等待共享数据的更新，比如定期的重新初始化。在这些情况下，准备数据的线程可以对条件变量调用 notify_all()成员函数，而不是 notify_one()。顾名思义，这将导致当前执行 wait()的所有线程检查它们正在等待的条件。

如果等待线程只等待一次，因此当条件为真时，它将不再等待该条件变量，那么条件变量可能不是同步机制的最佳选择。如果等待的条件是某一特定数据的可用性，则尤其如此。在这种情况下，*期望*（future）可能更合适。

# 4.2 使用期望等待一次性事件

假设你要乘飞机去国外度假。一旦你到达机场，完成了各种登机手续，你还得等待你的航班准备登机的通知，这可能要等上好几个小时。是的，你也许能找到一些消磨时间的方式，比如看书、上网，或者在机场价格高昂的咖啡馆用餐，但基本上你只是在等待一件事：登机的信号。不仅如此，一个给定的航班只会有一次；下次你去度假时，你将等待不同的航班。

C++标准库将这种一次性事件建模为所谓的*期望*（future）。如果一个线程需要等待一个特定的一次性事件，它会以某种方式获得一个表示该事件的期望。然后，线程可以周期性地等待很短的一段时间，以查看事件是否已经发生(查看出发时刻表)，同时在检查的间隙执行其他任务(在价格高昂的咖啡馆用餐)。或者，它可以执行另一个任务，直到它需要事件在它继续之前发生，然后就等待期望变成*就绪*（ready）。期望可能有与之相关的数据(比如你的航班在哪个登机口登机)，也可能没有。一旦事件发生(因此期望已经变成就绪)，期望就不能被重置。

C++标准库中，有两种期望，实现为两个类模板，声明在<future>库头文件中：*唯一的期望*（unique futures）(std::future<>)和*共享的期望*（shared futures）(std::shared_future<>)。它们仿照了 std::unique_ptr 和 std::shared_ptr。一个 std::future 的实例是唯一一个引用其关联事件的实例，而多个 std::shared_future 实例可能引用同一事件。后一种情况中，所有实例会在同时变为*就绪*状态，然后他们可以访问与事件相关的任何数据。这些关联的数据是这些类成为模板的原因；就像 std::unique_ptr 和 std::shared_ptr 一样，模板参数是关联数据的类型。如果没有相关联的数据，可以使用 std::future<void>与 std::shared_future<void>的特化模板。尽管期望用于线程间通信，但是期望对象本身不提供同步访问。如果多个线程需要访问一个期望对象，它们必须通过互斥锁或其他同步机制来保护访问，如第 3 章所述。但是，正如你将在 4.2.5 节中看到的，多个线程可以访问它们自己的 std::shared_future<>副本，而无需进一步同步，即使它们都引用相同的异步结果。

并发技术规范在 std::experimental 名空间中提供了这些类模板的扩展版本：std::experimental::future<>和 std::experimental:: shared_future<>。这些类的行为与 std 名空间中的对应类相同，但是它们有额外的成员函数来提供额外的功能。需要重点注意的是，名字 std::experimental 并非暗示代码的质量(我希望实现的质量和你的库供应商提供的其他东西是一样的)，但是需要强调的是，这些都是非标准的类和函数，因此，如果它们最终被采用到未来的 C++标准中，它们的语法和语义可能会有变化。如果想要使用这些设施，需要包含<experimental/future>头文件。

最基本的一次性事件是在后台运行的计算的结果。在第 2 章中，你看到 std::thread 并没有提供一种简单方法从这样的任务中返回一个值，并且我承诺过将在第 4 章中用期望来解决——现在是时候看看怎么解决了。

## 4.2.1 从后台任务返回值

假设有一个长时间运行的计算，你希望最终产生一个有用的结果，但当前不需要该值。也许你已经找到了一种方法来确定生命，宇宙和万物的答案——从道格拉斯·亚当斯[1]那取一个例子（译注：作者这里开玩笑，扯远了，可以无视）。你可以启动一个新的线程来执行计算，但这意味着你必须负责把结果传送回来，因为 std::thread 没有提供直接的机制来做这个事情。这就是需要 std::async 函数模板(也声明在<future>头文件中)的地方。

如果你不需要立即得到结果，可以使用 std::async 来启动一个*异步任务*（asynchronous task）。而不是给你一个 std::thread 对象去等待，std::async 会返回一个 std::future 对象，它将最终持有函数的返回值。当你需要该值时，只需在期望上调用 get()，线程就会阻塞，直到期望*就绪*（ready），然后返回该值。下面的清单显示了一个简单的示例。

清单 4.6 使用 std::future 获取异步任务的返回值

```
#include <future>
#include <iostream>

int find_the_answer_to_ltuae();
void do_other_stuff();
int main()
{
  std::future<int> the_answer=std::async(find_the_answer_to_ltuae);
  do_other_stuff();
  std::cout<<"The answer is "<<the_answer.get()<<std::endl;
}
```

std::async 允许你通过向调用中添加更多的参数来传递额外的参数给函数，这与 std::thread 的方法相同。如果第一个参数是指向成员函数的指针，那么第二个参数提供了应用成员函数的对象(要么直接是对象，要么通过指针，亦或包装在 std::ref 中)，其余的参数作为成员函数的参数传递。否则，第二个和随后的参数将作为函数或可调用对象的第一

个参数。就如 std::thread，当参数为右值时，拷贝操作将使用*移动*（moving）的方式转移原始数据。这就允许使用只支持移动的类型作为函数对象和参数。参见下面的清单：

清单 4.7 使用 std::async 向函数传递参数

```
#include <string>
#include <future>

struct X
{
  void foo(int,std::string const&);
  std::string bar(std::string const&);
};
X x;

auto f1=std::async(&X::foo,&x,42,"hello");  // 调用 p->foo(42, "hello"), p 是指向 x 的指针
auto f2=std::async(&X::bar,x,"goodbye");  // 调用 tmpx.bar("goodbye"), tmpx 是 x 的拷贝副本

struct Y
{
  double operator()(double);
};
Y y;

auto f3=std::async(Y(),3.141);  // 调用 tmpy(3.141)，tmpy 从 Y 通过移动构造得到
auto f4=std::async(std::ref(y),2.718);  // 调用 y(2.718)
X baz(X&);
std::async(baz,std::ref(x));  // 调用 baz(x)

class move_only
{
public:
  move_only();
  move_only(move_only&&)
  move_only(move_only const&) = delete;
  move_only& operator=(move_only&&);
  move_only& operator=(move_only const&) = delete;

  void operator()();
};
```

```
auto f5=std::async(move_only());  // 调用 tmp()，tmp 是通过
std::move(move_only())构造得到
```

默认情况下，当等待期望时，std::async 是否启动一个新线程，还是同步执行任务，取决于实现。在大多数情况下，这是你想要的，但是你可以在调用函数之前，通过 std::async 的附加参数指定要使用哪种模式。这个参数的类型是 std::launch，它可以是 std::launch::defered，表明函数调用被推迟到 wait()或 get()函数调用时才执行，或者是 std::launch::async，表明函数必须在它自己的线程上运行，还可以是 std::launch::deferred | std::launch::async 表明让具体实现来选择哪种方式。最后一个选项是默认的。如果函数调用是推迟的，它可能永远也不会运行。例如：

```
auto f6=std::async(std::launch::async,Y(),1.2);  // 在新线程上执行
auto f7=std::async(std::launch::deferred,baz,std::ref(x));  // 在 wait()
或 get()调用时执行
auto f8=std::async(
              std::launch::deferred | std::launch::async,
              baz,std::ref(x));  // 实现选择执行方式
auto f9=std::async(baz,std::ref(x));
f7.wait();  //  调用延迟函数
```

正如你将在本章后面以及第 8 章中看到的，使用 std::async 可以很容易地将算法划分为可以并发运行的任务。然而，这并不是将 std::future 与任务联系起来的唯一方法；你还可以通过将任务包装到 std::packaged_task<>类模板的实例中，或者通过编写代码使用 std::promise<>类模板显式地设置值来实现。std::packaged_task 是一个比 std::promise 更高层次的抽象，所以我将从它开始。

## 4.2.2 将任务与期望关联起来

std::packaged_task<>将期望绑定到函数或可调用对象。当调用 std::packaged_task<>对象时，它调用关联的函数或可调用对象，并让期望*就绪*（ready），返回值存储为关联的数据。它可以用作线程池(参见第 9 章)或其他任务管理方案的构建块，比如让每个任务运行在它自己的线程上，或者在一个特定的后台线程上依次运行它们。如果大型操作可以划分为自包含的子任务，那么每个子任务都可以封装在 std::packaged_task<>实例中，然后把该实例传递给任务调度器或线程池。这就抽象出了任务的细节；调度器只需要应对 std::packaged_task<>实例，而不是单独的函数。

std::packaged_task<>类模板的模板参数是个函数签名，比如 void()表示不带参数也没有返回值的函数，或者 int(std::string&,double*)表示有一个 std::string 类型的非 const 引用和一个指向 double 类型的指针作为参数，并且返回 int 的函数。当你构建一个 std::packaged_task 实例，你必须传递一个函数或者可调用对象，它能接受指定的参数，并且返回类型可以转换成指定的返回类型。类型不必精确匹配；比如你可以从一个带 int 类型参数并且返回 float 类型的函数构建一个 std::packaged_task<double(double)>，因为这些类型之间都可以隐式转换。

指定函数签名的返回类型确定了从 get_future()成员函数返回的 std::future<>的类型，而函数签名的参数列表用于指定打包任务（packaged task）的函数调用操作符的签名。例如，std::packaged_task <std::string(std::vector<char>*，int)>的部分类定义如下所示。

清单 4.8 特化版 std::packaged_task<>的部分类定义

```
template<>
class packaged_task<std::string(std::vector<char>*,int)>
{
public:
  template<typename Callable>
  explicit packaged_task(Callable&& f);
  std::future<std::string> get_future();
  void operator()(std::vector<char>*,int);
};
```

因为 std::packaged_task 对象是一个可调用对象，所以它可以封装在 std::function 对象中，传给 std::thread 作为线程函数，或传给需要可调用对象的另一个函数，或直接进行调用。当将 std::packaged_task 作为函数对象调用时，提供给函数调用操作符的参数被传递给所包含的函数，返回值被存储为异步结果，它可以从 get_future()获得的 std::future 中获取。因此，你可以将任务包装在 std::packaged_task 中，在将 std::packaged_task 对象传递到其他地方（以在适当的时候调用它）前检索期望。当你需要结果时，你可以等待期望变成就绪状态。下面的示例展示了这一点。

**线程间传递任务**

许多图形用户界面（GUI）框架要求从特定的线程更新 GUI，因此，如果另一个线程需要更新 GUI，它必须向正确的线程发送一条消息来进行更新。std::packaged_task 提供了一种实现此目的的方法，而不需要为每个与 GUI 相关的活动提供自定义消息，如下所示。

清单 4.9 使用 std::packaged_task 在 GUI 线程运行代码

```
#include <deque>
#include <mutex>
#include <future>
#include <thread>
#include <utility>

std::mutex m;
std::deque<std::packaged_task<void()> > tasks;

bool gui_shutdown_message_received();
void get_and_process_gui_message();
```

```
void gui_thread()  // 1
{
  while(!gui_shutdown_message_received())  // 2
  {
    get_and_process_gui_message();  // 3
    std::packaged_task<void()> task;
    {
      std::lock_guard<std::mutex> lk(m);
      if(tasks.empty())  // 4
        continue;
      task=std::move(tasks.front());  // 5
      tasks.pop_front();
    }
    task();  // 6
  }
}

std::thread gui_bg_thread(gui_thread);

template<typename Func>
std::future<void> post_task_for_gui_thread(Func f)
{
  std::packaged_task<void()> task(f);  // 7
  std::future<void> res=task.get_future();  // 8
  std::lock_guard<std::mutex> lk(m);
  tasks.push_back(std::move(task));  // 9
  return res; // 10
}
```

这段代码比较简单：GUI 线程①循环直到收到一条关闭 GUI 的信息后退出②，不断轮询 GUI 消息来处理③（例如用户点击消息），以及处理任务队列上的任务。如果队列中没有任务④，它将再次循环；否则，它从队列中提取一个任务⑤，然后释放队列上的锁，并且执行任务⑥。当任务完成的时候，和任务关联的期望会被变为就绪状态。

将一个任务传入队列也同样简单：从提供的函数⑦创建一个新的打包任务，通过调用 get_future()成员函数可以从那个任务上获取期望⑧，并且在期望返回调用者之前⑩，任务被放入到队列（译注：原文是 list，应该是笔误）⑨中。然后，如果需要知道任务是否已经完成，则将消息发布到 GUI 线程的代码可以等待期望；如果不需要知道的话，则可以丢弃期望。

这个例子使用 std::packaged_task<void()>处理任务，它包装了一个不带任何参数并返回 void 的函数或其他可调用对象 (如果它返回任何其他内容，返回值将被丢弃)。这可能是最简单的任务，但是正如前面所看到的，std::packaged_task 也可以用于更复杂的情况——通过指定一个不同的函数签名作为模板参数，你可以改变返回类型（也就是改变了存储在期望关联状态中数据的类型）还有函数调用操作符的参数类型。这个示例可以很容易地

扩展为允许在 GUI 线程上运行的任务接受参数并在 std::future 中返回一个值，而不仅仅是一个任务完成的指示器。

对于那些不能表示为简单函数调用的任务，或者那些结果可能来自多个地方的任务，该怎么办呢？这些情况由创建期望的第三种方式来处理：使用 std::promise 来显式设置值。

### 4.2.3 做出承诺(std::promises)

当你的应用程序需要处理大量的网络连接时，通常倾向于在单独的线程上处理每个连接，因为这可以使网络通信更容易思考和编程。这对于连接数量少(因此线程数量也少)的情况工作的很好。不幸的是，随着连接数量的增加，这种方法就不那么合适了；大量的线程会消耗大量的操作系统资源，并可能导致大量的上下文切换(当线程数量超过可用的硬件并发时)，从而影响性能。在极端情况下，操作系统可能会在网络连接能力耗尽之前耗光运行新线程的资源。因此，在具有大量网络连接的应用程序中，通常使用少量线程(可能只有一个)处理连接，让每个线程同时处理多个连接。

考虑其中一个处理连接的线程。数据包将从不同的连接中以本质上随机的顺序进入，同样，数据包也将以随机的顺序排队等待发送。在许多情况下，应用程序的其他部分将等待数据被成功发送，或者等待通过特定的网络连接，新一批数据被成功接收。

std::promise<T>提供一种方法来设定一个值(类型为 T)，这个值可以通过关联的 std::future<T>对象读取出来。一对 std::promise/std::future 会为这种设施提供一个可行的机制；等待线程可以阻塞在期望上，同时，提供数据的线程可以使用配对中的*承诺*（promise）来设置相关的值，并将期望的状态置为*就绪*（ready）。

通过调用 get_future()成员函数，可以获得与给定的 std::promise 关联的 std::future 对象，就像 std::packaged_task 一样。当承诺的值被设置(使用 set_value()成员函数)时，期望处于*就绪*（ready）状态，然后可以用来检索存储的值。如果在不设置值的情况下销毁 std::promise，取而代之的是存储一个异常。4.2.4 节描述了异常如何在线程之间转移。

清单 4.10 展示了如前所述的处理连接的线程的一些示例代码，在这个例子中，使用一对 std::promise<bool>/std::future<bool>来确定传出数据块的成功传输；与期望相关的值只是一个简单的成功/失败标记。对于进来的包，与期望相关的数据是数据包的有效负载。

清单 4.10 使用承诺处理单线程上的多个连接

```
#include <future>

void process_connections(connection_set& connections)
{
  while(!done(connections))  // 1
  {
```

```
    for(connection_iterator  // 2
            connection=connections.begin(),end=connections.end();
          connection!=end;
          ++connection)
    {
      if(connection->has_incoming_data())  // 3
      {
        data_packet data=connection->incoming();
        std::promise<payload_type>& p=
            connection->get_promise(data.id);  // 4
        p.set_value(data.payload);
      }
      if(connection->has_outgoing_data())  // 5
      {
        outgoing_packet data=
            connection->top_of_outgoing_queue();
        connection->send(data.payload);
        data.promise.set_value(true);  // 6
      }
    }
  }
}
```

函数 process_connections()一直循环，直到 done()返回 true①为止。每一次循环，都会依次检查每一个连接②，检索出进来的数据，如果有的话③，或发送已入队的传出数据⑤。这里假设进来的数据包是具有 ID 和有效负载的。一个 ID 映射到一个 std::promise(可能是通过在关联容器中查找)④，它的值是包的有效负载。对于传出包，包是从传出队列中取出的，并通过连接发送。一旦发送完成，与传出数据相关的承诺将被设置为 true，以表明传输成功⑥。对网络协议来讲，是否能很好的映射取决于协议；这种承诺/期望风格的结构可能在特定的场景下无法工作，尽管它的结构与某些操作系统支持的异步 I/O 类似。

到目前为止，所有的代码都完全忽略了异常。虽然想象一个所有东西都一直工作的世界可能很好，但现实并非如此。有时磁盘满了，有时你要查找的东西并不在那里，有时网络失败，有时数据库宕机。如果在需要结果的线程中执行操作，代码可能只报告一个异常错误，因此仅仅因为要使用 std::packaged_task 或 std::promise 就要求一切正常，这种限制是不必要的。C++标准库因此提供了一种干净的方法来处理这种情况下的异常，并允许它们作为相关结果的一部分保存。

## 4.2.4 为期望保存异常

考虑下面的一小段代码。如果将-1 传递给 square_root()函数，它会抛出一个异常，然后调用者会看到这个异常：

```
double square_root(double x)
{
  if(x<0)
  {
    throw std::out_of_range("x<0");
  }
  return sqrt(x);
}
```

现在假设不是从当前线程调用 square_root()

```
double y=square_root(-1);
```

将调用作为异步调用运行：

```
std::future<double> f=std::async(square_root,-1);
double y=f.get();
```

如果行为完全相同就很理想了；正如 y 在两种情况下都得到函数调用的结果一样，如果调用 f.get()的线程也能看到异常，那就太好了，就像在单线程情况下那样。

事情就是这样：如果作为 std::async 的一部分调用的函数抛出一个异常，那么该异常将被存储在期望中，代替存储的值，这时期望就变为*就绪*（ready）状态，并且对 get()的调用将重新抛出该存储的异常(注意：标准没有指定重新抛出的这个异常是原始的异常对象，还是一个拷贝；不同的编译器和库将会在这方面做出不同的选择)。如果将函数包装在 std::packaged_task 中，也会发生同样的情况——在调用任务时，如果包装的函数抛出异常，那么该异常将存储在期望中，并准备在调用 get()时抛出。

理所当然，std::promise 通过显式的函数调用提供了相同的设施。如果希望存储异常而不是值，可以调用 set_exception()成员函数而不是 set_value()。这通常会在 catch 块中用于作为算法的一部分抛出的异常，以便用该异常填充承诺：

```
extern std::promise<double> some_promise;
try
{
  some_promise.set_value(calculate_value());
}
catch(...)
{
  some_promise.set_exception(std::current_exception());
}
```

它使用 std::current_exception()来检索抛出的异常；这里的替代方法是使用 std::make_exception_ptr()直接存储新的异常，而不抛出：

```
some_promise.set_exception(std::copy_exception(std::logic_error("foo ")
));
```

如果异常的类型是已知的，那么这比使用 try/catch 块要干净得多，并且它应该被优先使用；因为它不仅简化了代码，而且还为编译器提供了更大的机会来优化代码。

在期望存储异常的另一种方法是销毁与期望关联的 std::promise 或 std::packaged_task，而不调用 promise 上的设置函数或调用打包的任务。无论哪种情况，如果期望还不是就绪状态，std::promise 或 std::packaged_task 的析构函数都会存储一个 std::future_error 异常，这个异常有一个值为 std::future_errc::broken_promise 的错误码在关联的状态中；通过创建一个期望，你做出承诺会提供一个值或异常，而如果不提供的话，就破坏了值或异常的源头，你就违背了该承诺。在这种情况下，如果编译器不存储任何东西到期望中，等待的线程可能会永远等下去。

到目前为止，所有的例子都使用了 std::future。然而，std::future 也有其局限性，其中最重要的是只能有一个线程等待结果。如果需要等待来自多个线程的相同事件，则需要使用 std::shared_future。

## 4.2.5 从多个线程等待

虽然 std::future 处理所有从一个线程到另一个线程传输数据所需的同步，但对特定 std::future 实例成员函数的调用彼此间没有同步。如果你从多个线程访问单个 std::future 对象而没有额外的同步，就会出现*数据竞争*（data race）和未定义的行为。这是有意为之：std::future 建模了异步结果的唯一所有权，get()的一次性特性使得这种并发访问毫无意义——无论如何只有一个线程可以检索值，因为在第一次调用 get()之后就没有值可以检索了。

如果你对并发代码的出色设计要求多个线程可以等待同一事件，请不要绝望；std::shared_future 恰恰允许这样做。尽管 std::future 只能移动(因此所有权可以在实例之间转移，但每次只有一个实例引用一个特定的异步结果)，但 std::shared_future 实例是可拷贝的（copyable）(因此可以有多个对象引用同一个关联状态)。

现在，对于 std::shared_future，单个对象上的成员函数仍然是不同步的，因此为了避免在从多个线程访问单个对象时出现数据竞争，必须使用锁保护访问。使用它的首选方法是将 shared_future 对象的副本传递给每个线程，这样每个线程就可以安全地访问自己的本地 shared_future 对象，因为库现在正确地同步了内部对象。如果每个线程通过自己的 std::shared_future 对象访问共享的异步状态，那么从多个线程访问共享异步状态是安全的。参见图 4.1。

std::shared_future 的一个潜在用途是实现类似于复杂电子表格的并行执行；每个单元格都有一个最终值，可由公式在多个其他单元格中使用。计算依赖单元格结果的公式可以使用 std::shared_future 引用第一个单元格。如果每个单元格的所有公式都是并行执行

的，那么那些可以继续完成的任务将会并发执行，而那些依赖于其他任务的任务将会阻塞，直到它们的依赖准备好。这将允许系统最大限度地利用可用的硬件并发。

引用某个异步状态的 std::shared_future 实例是由引用该状态的 std::future 的实例构造的。由于 std::future 对象不与任何其他对象共享异步状态的所有权，因此必须使用 std::move 将该所有权转移到 std::shared_future 中，使 std::future 处于空状态，就像它是默认构造函数构造的一样：

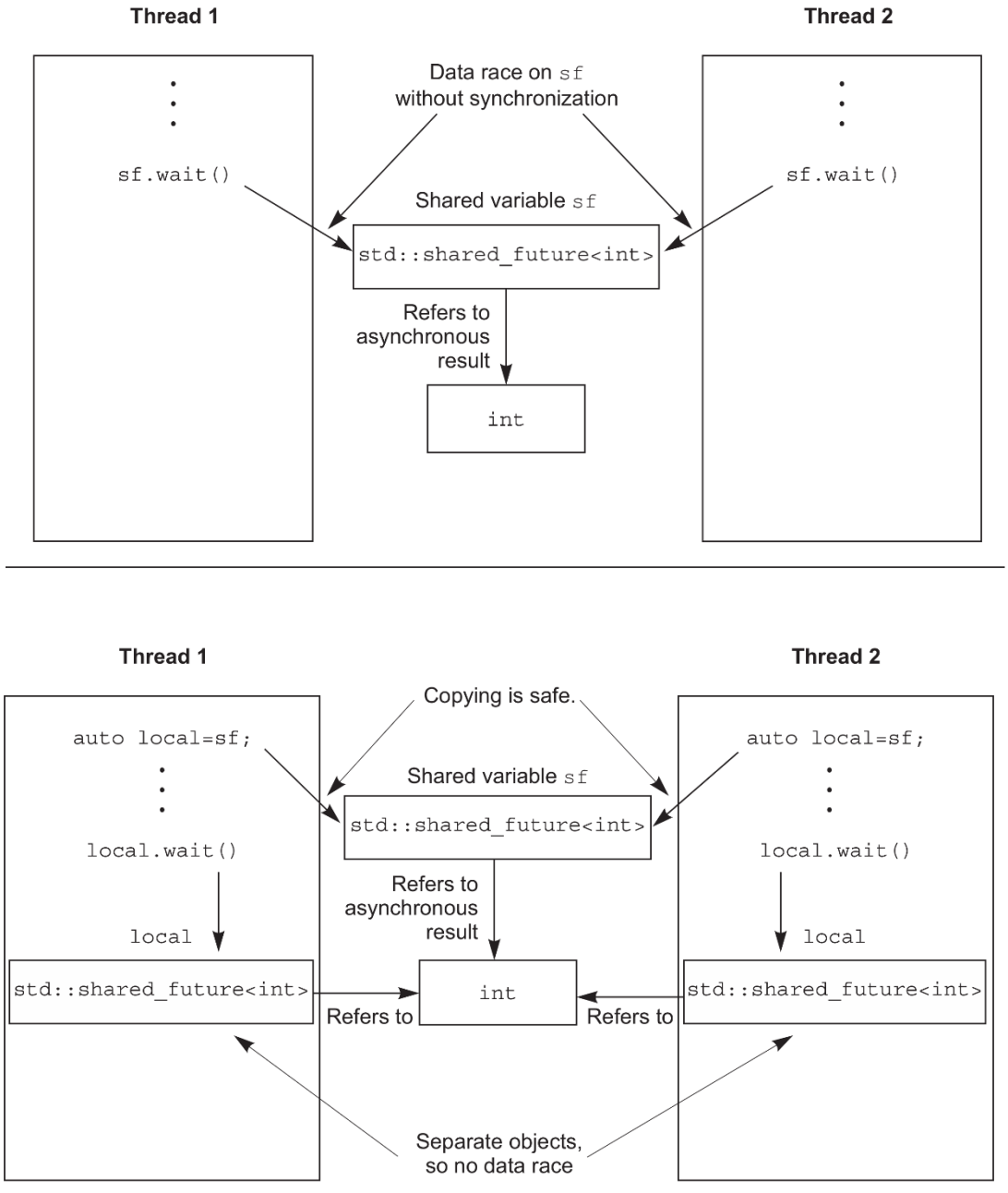


图 4.1 使用多个 std::shared_future 对象来避免数据竞争

```
std::promise<int> p;
std::future<int> f(p.get_future());
```

```
assert(f.valid());  // 1 期望 f 是有效的
std::shared_future<int> sf(std::move(f));
assert(!f.valid());  // 2 期望 f 不在有效
assert(sf.valid());  // 3 sf 现在是有效的
```

这里，期望 f 一开始是有效的①，因为它引用的是承诺 p 的异步状态，但是在转移状态给 sf 后，f 就不再有效了②，而 sf 就变成有效的③。

与其他可移动对象一样，所有权的转移对于右值是隐式的，因此你可以直接从 std::promise 对象的 get_future()成员函数的返回值构建 std::shared_future，例如：

```
std::promise<std::string> p;
std::shared_future<std::string> sf(p.get_future());  // 1 隐式转移所有权
```

这里所有权的转移是隐式的，std::shared_future<>从 std::future<std::string>类型的右值构造而来①。

std::future 还有一个额外的特性使得 std::shared_future 更容易被使用，新设施支持从变量的初始化器中自动推导出变量的类型(参见附录 A, A.6 节)。std::future 有一个 share()成员函数，它创建一个新的 std::shared_future 并直接将所有权转移给它。这可以节省大量的输入，并使代码更容易更改：

```
std::promise< std::map< SomeIndexType, SomeDataType, SomeComparator,
     SomeAllocator>::iterator> p;
auto sf=p.get_future().share();
```

在这里，sf 的类型推导为 std::shared_future<std::map<SomeIndexType, SomeDataType, SomeComparator, SomeAllocator>::iterator>，这个类型一口气很难读完。当比较器或分配器有所改动，你只需要对承诺的类型进行修改即可；期望的类型会自动更新以匹配。

有时你想限制等待一个事件的时间，要么因为你对代码的特定部分所花费的时间有严格的时间限制，要么因为如果事件不是很快就会发生的话，线程有其他有用的工作可以做。为了应对这个需求，许多等待函数都有变体，它们允许指定超时。

---

[1] 《银河系漫游指南》(*The Hitchhiker's Guide to the Galaxy*)中，计算机 Deep Thought 被建造来确定 “生命、宇宙以及万物的答案”，答案是 42。

# 4.3 有时间限制的等待

前面介绍的所有阻塞调用都将阻塞一段不确定的时间，挂起线程，直到等待的事件发生。在许多情况下，这很好，但是在某些情况下，你可能想要限制等待的时间。这就可能

允许你向交互式用户或另一个过程（process）发送某种形式的“我还活着”的消息，或者允许你在用户已放弃等待并单击“取消”时中止等待。

你可能希望指定两种类型的超时：基于时长（duration-based）的超时，即等待指定长度的时间(例如，30 毫秒)；或者绝对超时，等待到指定的时间点(例如，2011 年 11 月 30 日 17:30:15 045987023 UTC)。大多数等待函数都提供了处理这两种超时形式的变体。处理基于时长超时的变体有_for 后缀，处理绝对超时的变体有_until 后缀。

所以，举例来说，std::condition 变量有两个重载的 wait_for()成员函数和两个重载的 wait_until()成员函数，对应到 wait()的两个重载——其中一个重载只是等待，直到被信号通知，或超时过期，或一个伪唤醒发生；另一个重载将在唤醒时检查所提供的谓词，并且只有在所提供的谓词为真(并且条件变量已被信号通知)或超时过期时返回。

在了解使用超时的函数的详细信息之前，让我们先看看 C++中指定时间的方式，首先从时钟（clocks）开始。

## 4.3.1 时钟

就 C++标准库而言，时钟是时间信息的来源。具体来说，时钟是一个类，它提供四种不同的信息：

- *当前时间*（now）
- 值的类型，这个值用于表示从时钟获得的时间
- 时钟计次周期（tick period）
- 时钟是否以相同的速度前进，并以此判断时钟是否*稳定*（steady）

时钟的当前时间可以通过调用时钟类的静态成员函数 now()获取；例如，std::chrono::system_clock::now()将返回系统时钟的当前时间。特定时钟的时间点类型通过 time_point 成员类型定义（typedef）来指定，因此 some_clock::now()的返回类型就是 some_clock::time_point。

时钟的计次周期是以秒为单位的分数，由时钟的 period 成员类型定义（typedef）给出——一秒计 25 次的时钟，它的 period 为 std::ratio<1, 25>，然而，每 2.5 秒计一次的时钟，它的 period 为 std::ratio<5, 2>。如果时钟计次周期（tick period）直到运行时才知道，或者它可能在应用程序的给定运行期间发生变化，则可以将 period 指定为平均计次周期、最小的可能计次周期或库作者认为合适的其他值。不能保证在程序的给定运行中观察到的计次周期与该时钟指定的 period 相匹配。

如果时钟以统一的速度计次（ticks）(无论速度是否与 period 匹配)，并且不可调整，这种时钟就被称为稳定的时钟。如果时钟的 is_steady 静态数据成员为 true 时，这个时钟就是稳定的，否则，就是不稳定的。通常情况下，std::chrono::system_clock 是不稳定的，因为时钟是可调的，即使这样的调整是自动进行，以考虑当地时钟漂移。这样的调整可能导致对 now()的调用返回的值比先前对 now()的调用返回的值早，这违反了统一计次速度的要求。稍后你将看到，稳定时钟对于超时计算很重要，因此 C++标准库以 std::chrono::steady_clock 的形式提供了一个稳定的时钟。C++标准库提供的其他时钟有

std::chrono::system_clock(前面提到过)，它代表了系统的“实时”时钟，并且提供了函数在时间点和 time_t 之间的转换；std::chrono::high_resolution_clock 时钟，可能是标准库中提供的具有最小计次周期(因此具有最高的分辨率)的时钟。它可能是对其他时钟的一个类型定义（typedef）。这些时钟以及其他时间设施是在<chrono>库头文件中定义的。

我们稍后将讨论时间点的表示，但首先我们看下时长（durations）是如何表示的。

## 4.3.2 时长

时长是时间支持里最简单的部分；它们由 std::chrono::duration<>类模板负责（线程库使用的所有 C++时间处理设施，都在 std::chrono 命名空间中)。第一个模板参数是表示（representation）的类型(比如，int，long 或 double)，第二个模板参数是个分数，它指定了时长表示单位是多少秒（译注：duration 是由一个数值和一个分数组合而成，数值表示计数，就是前面说的“表示”，它的类型可以是 int，long 或 double，分数表示单位，是以秒作为基准，它的类型是 std::ration<>,例如 chrono::duration<int, ration<1,3>> d(2),表示这个时长是 2 个 1/3 秒，也就是 2/3 秒）。例如，几分钟的时间存在 short 类型中可以表示为 std::chrono::duration<short, std::ratio<60, 1>>，因为 1 分钟有 60 秒。另一方面，把毫秒数存在 double 类型中可以表示为 std::chrono::duration<double, std::ratio<1, 1000>>，因为每毫秒是 1/1000 秒。

标准库在 std::chrono 命名空间内，提供一系列预先定义好的时长：nanoseconds, microseconds, milliseconds, seconds, minutes 和 hours。它们为选择的表示使用了足够大的整形，以至于如果你愿意的话，你可以用合适的单位来表示 500 年以上的时间。当然还为所有从 std::atto(10-18)到 std::exa(1018)（如果你的平台有 128 位整形的话，还可以更大）的国际单位制（SI，法语：Système International d'Unités）比例（ration）提供了类型定义，当指定自定义时长，如 std::duration<double, std::centi>，以双精度表示百分之一秒的计数时，就可以使用它们。

为了方便起见，在 C++ 14 中引入的 std::chrono_literals 命名空间中有许多预定义的用于时长的字面值后缀操作符。这可以简化使用硬编码时长值的代码，例如：

```
using namespace std::chrono_literals;
auto one_day=24h;
auto half_an_hour=30min;
auto max_time_between_messages=30ms;
```

当与整型字面值一起使用时，这些后缀等同于使用预定义的时长类型定义，因此 15ns 和 std::chrono::nanoseconds(15)是相同的值。但是，当与浮点字面值一起使用时，这些后缀创建了具有未指定表示类型的适当缩放的浮点时长。因此，2.5min 将是 std::chrono::duration<some-float-point type,std::ratio<60,1>>。如果你关心实现所选择的浮点类型的范围或精度，那么你需要自己构造一个具有合适表示类型的对象，而不是贪图字面值后缀带来的便利。

在不需要截断值的情况下，时长之间的转换是隐式的(因此将小时转换为秒是可以的，但将秒转换为小时则不行)。显式转换可以用 std::chrono::duration_cast<>：

```
std::chrono::milliseconds ms(54802);
std::chrono::seconds s=
      std::chrono::duration_cast<std::chrono::seconds>(ms);
```

结果是被截断了而不是四舍五入，因此在本例中 s 的值为 54。

时长支持算术运算，因此可以通过加减时长来获得新的时长，或者乘以或除以底层表示类型的常数(第一个模板参数)。因此 5*seconds(1)等同于 seconds(5)或 minutes(1)-seconds(55)。可以通过成员函数 count()获得时长内的单位数量的计数。因此 std::chrono::milliseconds(1234)是 1234。

基于时长的等待可由 std::chrono::duration<>来完成。例如：你可以等待 35 毫秒来让期望变成就绪状态：

```
std::future<int> f=std::async(some_task);
if(f.wait_for(std::chrono::milliseconds(35))==std::future_status::ready
)
  do_something_with(f.get());
```

等待函数都返回一个状态，以指示等待是超时了还是等待事件发生了。在这种情况下，假设你正在等待一个期望，如果等待超时的话，函数会返回 std::future_status::timeout，如果期望就绪了，就返回 std::future_status::ready，如果期望对应的任务推迟了，则返回 std::future_status::deferred。库内部基于时长的等待会使用稳定的时钟来度量时间，因此 35 毫秒意味着逝去 35 毫秒的时间，即使等待期间调整了系统时钟（往前或往后调整）。当然，变幻莫测的系统调度和不断变化的操作系统时钟精度意味着线程发出调用和从调用返回之间的时间可能要长于 35 毫秒。

啃下了时长以后，我们现在可以转向时间点了。

## 4.3.3 时间点

时钟的时间点可以用 std::chrono::time_point<>类模板的实例来表示，实例的第一个参数用来指定所要使用的时钟，第二个函数参数用来表示计量单位(一个特化的 std::chrono::duration<>)。一个时间点的值是时钟从称为“纪元”的一个特定时间点以来的时间长度(是已指定时长的倍数)。时钟的纪元是基本属性，但不能直接查询或由 C++标准指定的。典型的纪元包括 1970 年 1 月 1 日 00 时以及运行应用程序的计算机启动的瞬间。时钟可以共享一个纪元或有独立的纪元。如果两个时钟共享一个纪元，一个类中的 time_point 类型定义可以指定另一个类作为与 time_point 关联的时钟类型。虽然你不能确定纪元是什么时候，但是你可以为给定的 time_point 获取 time_since_epoch()。这个成员函数返回一个时长值，指定了从时钟纪元到该特定时间点的时间长度。

例如，你可以指定一个时间点为 std::chrono::time_point<std::chrono::system_clock, std::chrono::minutes>。这将保持相对于系统时钟的时间，但以分钟计算，而不是系统时钟的本机精度(通常是秒或更小的值)。

你可以给 std::chrono::time_point<>实例增/减时长，来获得一个新的时间点，所以 std::chrono::hight_resolution_clock::now() + std::chrono::nanoseconds(500)将得到 500 纳秒后的时间。当你知道一个代码块的最大时长，但是代码块中存在对等待函数的多次调用，或者先于等待函数的非等待函数有多次调用，但这些调用占用了部分时间预算，在这些情况下时间点的增减操作对计算绝对超时就很有用。

你还可以从共享同一时钟的另一个时间点中减去一个时间点。其结果是一个时长，它指定了两个时间点之间的时间长度。这对于测定时间的代码块很有用，例如：

```
auto start=std::chrono::high_resolution_clock::now();
do_something();
auto stop=std::chrono::high_resolution_clock::now();
std::cout<<”do_something() took “
  <<std::chrono::duration<double,std::chrono::seconds>(stop-
start).count()
  <<” seconds”<<std::endl;
```

std::chrono::time_point<>实例的时钟参数不仅指定了纪元。当你把时间点传给一个绝对超时的等待函数，时间点的时钟参数用于测量时间。当时钟发生改变时，这将产生重大影响，因为等待会跟踪时钟变化，等待函数直到时钟的 now()函数返回一个比指定超时更靠后的时间点才会返回。如果时钟向前调整，这可能会减少等待的总长度(这个总长度由一个稳定的时钟测量)，如果它向后调整，这可能会增加等待的总长度。

如你所料，时间点与等待函数的_until 变体一起使用。典型的用例是作为程序中某个固定点上的 *some-clock*::now()的偏移量，尽管关联在系统时钟上的时间点可以使用 std::chrono::system_clock::to_time_point()静态成员函数从 time_t 转换而来，以在用户可见的时间安排操作。例如，如果等待与条件变量关联的事件的时间最长为 500 毫秒，则可以执行如下清单所示的操作：

清单 4.11 等待一个带超时的条件变量

```
#include <condition_variable>
#include <mutex>
#include <chrono>

std::condition_variable cv;
bool done;
std::mutex m;

bool wait_loop()
{
```

```
  auto const timeout= std::chrono::steady_clock::now()+
      std::chrono::milliseconds(500);
  std::unique_lock<std::mutex> lk(m);
  while(!done)
  {
    if(cv.wait_until(lk,timeout)==std::cv_status::timeout)
      break;
  }
  return done;
}
```

如果你没有传递谓词来等待，那么这是等待有时间限制的条件变量的推荐做法。这样，循环的总长度是有界的。正如你在 4.1.1 节中看到的，如果不传递谓词，那么在使用条件变量时需要进行循环，以便处理伪唤醒。如果在循环中使用 wait_for()，那么在一个伪唤醒之前，你可能会等待几乎整个时间长度，并且等待时间的下一次再次重新开始。这可能重复任意次数，使总等待时间不受控制。

掌握了指定超时的基本知识后，让我们看看可以使用超时的函数。

## 4.3.4 接受超时的函数

超时最简单的用法是给特定线程的处理添加延时，这样它就不会在无事可做时占用其他线程的处理时间。你在 4.1 节中看到过这样的示例，在循环中轮询了一个“done”标记。处理这个问题的两个函数是 std::this_thread::sleep_ for()和 std::this_thread::sleep_until()。它们的工作方式类似于一个基本的闹钟：线程休眠指定的时长(使用 sleep_for())或到指定的时间点(使用 sleep_until())。sleep for()对于 4.1 节中这样的例子很有意义，在这些例子中，必须定期执行一些操作，并且流逝的时间是最重要的。另一方面，sleep_until()允许你安排线程在特定的时间点唤醒。这可以用于在午夜触发备份，或在早上 6 点运行工资单打印，或在视频回放时挂起线程，直到下一次帧刷新。

休眠并不是唯一需要超时的设施；你已经看到可以对条件变量和期望使用超时。如果互斥锁支持的话，甚至可以在尝试获取互斥锁时使用超时。普通 std::mutex 和 std::recursive_mutex 不支持上锁超时，但是 std::timed_mutex 支持，std::recursive_timed_mutex 同样也支持。这两种类型都支持 try_lock_for()和 try_lock_until()成员函数，它们试图在指定的时间段内或在指定的时间点之前获得锁。表 4.1 显示了 C++标准库中可以接受超时的函数，它们的参数和它们的返回值。列为 duration 的参数必须是 std::duration<>的实例，列为 time_point 的参数必须是 std::time_point<>的实例。

既然我已经介绍了条件变量、期望、承诺和打包任务的机制，是时候看看更广阔的前景了，以及如何使用它们来简化线程间操作的同步。

表4.1 可接受超时的函数

| 类型/命名空间 | 函数 | 返回值 |
| --- | --- | --- |
| std::this_thread 命名空间 | sleep_for(duration)<br>sleep_until(time_point) | N/A |
| std::condition_variable or<br>std::condition_variable_any | wait_for(lock,duration)<br>wait_until(lock,time_point) | bool类型——当唤醒时返回谓词的返回值 |
| std::cv_status::timeout or<br>std::cv_status::no_timeout | wait_for(lock,duration,predicate)<br>wait_until(lock,time_point,predicate) | |
| std::timed_mutex, std::recursive_timed_mutex or std::shared_timed_mutex | try_lock_for(duration)<br>try_lock_until(time_point) | bool类型——如果获得锁返回true否则false |
| std::shared_timed_mutex | try_lock_shared_for(duration)<br>try_lock_shared_until(time_point) | bool类型——如果获得锁返回true否则false |
| std::unique_lock<TimedLockable> unique_lock(lockable,duration) unique_lock(lockable,time_point) | try_lock_for(duration)<br>try_lock_until(time_point) | 若无关联互斥锁会抛异常 |
| std::shared_lock<SharedTimedLockable> shared_lock(lockable,duration) shared_lock(lockable,time_point) | | bool类型——如果获得锁返回true否则false |
| std::future<ValueType> or std::shared_future<ValueType> | wait_for(duration) wait_until(time_point) | 如果等待超时返回std::future_status::timeout，如果期望就绪返回std::future_status::ready，或者如果要计算结果的函数还未启动则返回std::future_status::deferred |

# 4.4 使用操作的同步来简化代码

使用到目前为止在本章中描述的同步设施作为构建块，可以让你专注于需要同步的操作，而不是机制。这可以帮助简化代码的一种方式是，它提供了一种更函数式(从函数式编程的意义上说)的并发编程方法。不同于在线程间直接共享数据，每个任务都可以得到它需要的数据，并且结果可以通过使用期望将其传播给任何其他需要它的线程。

## 4.4.1 使用期望的函数式编程

术语*函数式编程*(FP，functional programming)指的是一种编程风格，其中函数调用的结果仅依赖于该函数的参数，而不依赖于任何外部状态。这与函数的数学概念有关，这意味着如果你使用相同的参数调用一个函数两次，结果将完全相同。这是C++标准库中许多数学函数的属性，比如sin，cos和sqrt，以及基本类型上的简单操作，比如3+3,6*9，或者1.3/4.7。一个纯函数也不修改任何外部状态；函数的效果完全局限于返回值。

这使得事情思考起来比较简单，特别是在涉及并发时，因为在第3章中讨论的与共享内存相关的许多问题都消失了。如果没有对共享数据进行修改，就不存在竞争条件，因而也不需要使用互斥锁来保护共享数据。这是个极大的简化，像Haskell（http://www.haskell.org/）这种编程语言中所有函数默认都是纯函数，它在并发编程中

越来越受欢迎。因为大多数东西都是纯的，所以实际修改共享状态的非纯函数就更加突出，因此更容易思考它们如何适应应用程序的整体结构。

然而，FP 的好处并不仅限于那些把它作为默认范式的语言。C++是一个多范式的语言，也可以写出 FP 风格的程序。这在 C++11 中比 C++98 更容易，因为 C++11 支持 lambda 函数(详见附录 A，A.6 节)，还合入了 Boost 和 TR1 中的 std::bind，以及引入变量的自动类型推导(详见附录 A，A.7 节)。期望是使 FP 风格的并发在 C++中可行的最后一块拼图；一个期望可以在线程之间传递，允许一个计算的结果依赖于另一个计算的结果，而不需要显式地访问共享数据。

**FP 风格的快速排序**

为了演示如何使用期望来实现 FP 风格的并发，让我们看一下快速排序算法的一个简单实现。该算法的基本思想很简单:给定一个值列表，将一个元素作为主元（pivot element），然后将列表划分为两个集合——小于主元的集合和大于或等于主元的集合。通过对两个集合排序，然后返回小于主元的有序列表，接着是主元，最后是大于等于主元的有序列表，可以获得列表的有序拷贝。图 4.2 展示了一个有 10 个整数的列表是怎么按照这个计划排序的。一个 FP 风格的顺序实现如下清单所示；它按值使用并返回列表，而不是像 std::sort()那样在原地排序。

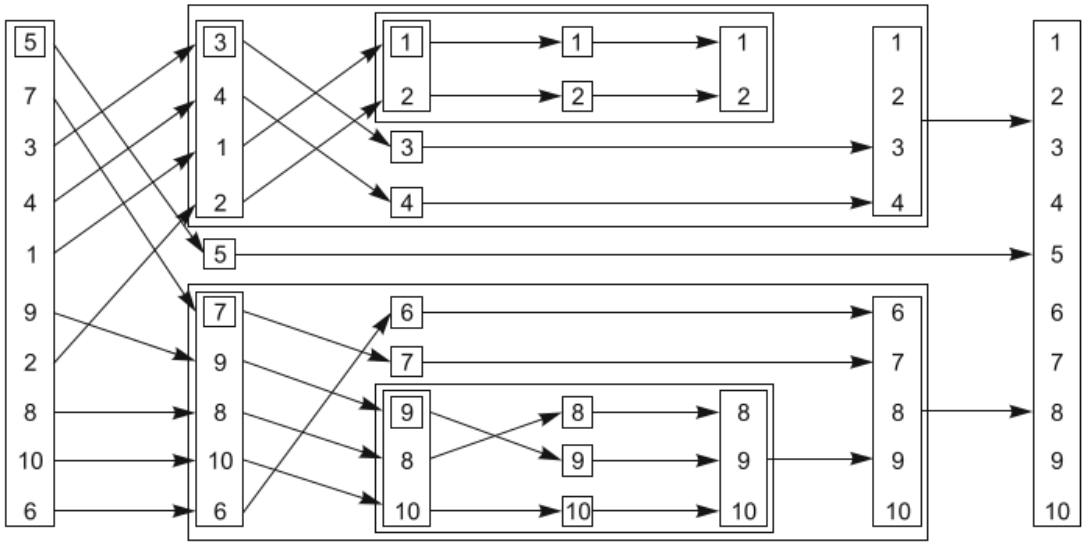


图 4.2 FP 风格的递归排序

清单 4.12 快速排序的顺序实现

```
template<typename T>
std::list<T> sequential_quick_sort(std::list<T> input)
{
  if(input.empty())
  {
```

```
    return input;
  }
  std::list<T> result;
  result.splice(result.begin(),input,input.begin());  // 1
  T const& pivot=*result.begin();  // 2

  auto divide_point=std::partition(input.begin(),input.end(),
            [&](T const& t){return t<pivot;});  // 3

  std::list<T> lower_part;
  lower_part.splice(lower_part.end(),input,input.begin(),
            divide_point);  // 4
  auto new_lower(
            sequential_quick_sort(std::move(lower_part)));  // 5
  auto new_higher(
            sequential_quick_sort(std::move(input)));  // 6

  result.splice(result.end(),new_higher);  // 7
  result.splice(result.begin(),new_lower);  // 8
  return result;
}
```

虽然接口是 FP 风格的，但如果你自始至终使用 FP 风格，会做很多拷贝，所以内部使用“普通”命令式的风格。使用 splice()将第一个元素从链表的前面分割出来，以此作为主元①。尽管这可能会导致次优排序(就比较和交换的数量而言)，但由于遍历链表，使用 std::list 执行任何其他操作都会增加相当多的时间。你知道它会出现在结果中，所以你可以直接将它拼接到你将要使用的链表中。现在，你也会想要使用它来进行比较，所以让我们引用它来避免拷贝②。然后可以使用 std::partition 将序列划分为小于主元的值和不小于主元的值③。指定分区标准的最简单方法是使用 lambda 函数；你可以使用引用捕获来避免拷贝主元值(更多有关 lambda 函数的信息详见附录 A，A.5 节)。

std::partition()原地重排链表并返回一个迭代器，该迭代器标记第一个不小于主元值的元素。迭代器的完整类型可能相当冗长，因此只需使用 auto 类型说明符强制编译器帮你搞定(参见附录 A，A.7 节)。

现在，你选择了一个 FP 风格的接口，因此如果要使用递归对这两个“一半”进行排序，则需要创建两个链表。可以再次用 splice()将 input 链表中小于 divided_point 的值移动到新链表 lower_part④中。这将把其余的值单独留在 input 中。然后可以使用递归调用对这两个链表进行排序⑤⑥。通过使用 std::move()来传递链表，你也可以在这里避免拷贝——结果无论如何都会隐式地移出。最后，你可以再次使用 splice()以正确的顺序将结果拼在一起。new_higher 里的值放在尾部⑦，在主元之后，并且 new_lower 放在前面，在主元之前⑧。

**FP 风格的并行快速排序**

因为已经使用了函数式的风格，所以现在很容易使用期望将其转换为并行版本，如下一个清单所示。操作集和以前相同，只是其中一些操作现在并行运行。这个版本使用了一个使用了期望和函数式风格的快速排序算法的实现。

清单 4.13 使用期望的并行快速排序

```
template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
  if(input.empty())
  {
    return input;
  }
  std::list<T> result;
  result.splice(result.begin(),input,input.begin());
  T const& pivot=*result.begin();

  auto divide_point=std::partition(input.begin(),input.end(),
                [&](T const& t){return t<pivot;});

  std::list<T> lower_part;
  lower_part.splice(lower_part.end(),input,input.begin(),
                divide_point);

  std::future<std::list<T> > new_lower(  // 1
                std::async(&parallel_quick_sort<T>,std::move(lower_part
)));

  auto new_higher(
                parallel_quick_sort(std::move(input)));  // 2

  result.splice(result.end(),new_higher);  // 3
  result.splice(result.begin(),new_lower.get());  // 4
  return result;
}
```

这里最大的变化是，不是在当前线程上排序值较小的部分，而是使用 std::async()在另一个线程上排序①。与前面一样，使用直接递归对链表值较大的部分排序②。通过递归地调用 parallel_quick_sort()，你可以利用可用的硬件并发。如果 std::async()每次启动一个新线程，那么如果你递归三次，你将有八个线程在运行；如果递归 10 次(对于~1000 个元素)，如果硬件能够处理的话，那么将有 1024 个线程在运行。如果库认为生成的任务太多(可能是因为任务的数量超过了可用的硬件并发)，它会切换成同步生成新任务。它们将在调用 get()的线程中运行，而不是在一个新线程上运行，从而避免在无法提高性能时将任务传递给另一个线程的开销。值得注意的是，它完全符合 std::async 的实现：每个任务启动一个新线程（即使面对巨大的超订 oversubscription)除非显式指定了

std::launch::deferred，或者同步运行所有任务，除非显式指定了std::launch::async。如果你依赖库来实现自动伸缩，建议你检查实现的文档，看看它展示了什么行为。

不使用 std::async()的话，你可以编写自己的 spawn_task()函数，作为 std::packaged_task 和 std::thread 的简单包装器，如清单 4.14 所示；你将为函数调用的结果创建一个 std::packaged_task，从它获得期望，在一个线程上运行它，然后返回期望。这也许不会提供太多的优势(实际上可能会导致大量的超订)，但它将为迁移到更复杂的实现铺平道路，该复杂实现将任务添加到由工作线程池运行的队列中。我们将在第 9 章中讨论线程池。只有当你知道自己在做什么，并且希望完全控制线程池的构建和执行任务的方式时，这样做才可能比使用 std::async 更有价值。

清单 4.14 spawn_task 的示例实现

```
template<typename F,typename A>
std::future<std::result_of<F(A&&)>::type>
   spawn_task(F&& f,A&& a)
{
  typedef std::result_of<F(A&&)>::type result_type;
  std::packaged_task<result_type(A&&)>
       task(std::move(f)));
  std::future<result_type> res(task.get_future());
  std::thread t(std::move(task),std::move(a));
  t.detach();
  return res;
}
```

不管怎样，还是回到 parallel_quick_sort。因为你只是使用了直接递归来获得 new_higher，所以可以像前面一样将其拼接到适当的位置③。但是，new_lower 现在是 std::future<std::list<T>>而非是一个链表，所以需要在调用 splice()前调用 get()来检索值④。然后等待后台任务完成，并将结果移动到 splice()调用中；get()返回对所包含结果的一个右值引用，因此可以将其移出(请参阅附录 A，A.1.1 节了解更多关于右值引用和移动语义的信息)。

即使假设 std::async()优化了可用硬件并发的使用，这仍然不是快速排序的理想并行实现。首先，std::partition 做了很多工作，这仍然是一个顺序调用，但目前它已经足够好了。如果你对最快的并行实现感兴趣，请查阅学术文献。或者，你可以使用 C++ 17 标准库中的并行重载(参见第 10 章)。

FP 不是唯一的避免共享可变数据的并发编程范式；另一个范式是通信顺序进程(CSP，Communicating Sequential Processer[2])，这里线程在概念上是完全独立的，没有共享数据，但有通信通道允许消息在它们之间传递。这种范式被 Erlang（http://www.erlang.org/）编程语言所采用，并且在消息传递接口（MPI，Message Passing Interface；http://www.mpi-forum.org/）环境中常用来做 C 和 C++的高性能运

算。我相信现在你得知 C++也可以通过一些规则来支持这一点也不会感到奇怪了；下一节将讨论实现这个的一种方法。

## 4.4.2 使用消息传递来同步操作

CSP 的思想很简单：如果没有共享数据，则可以完全独立地思考每个线程，纯粹基于它对接收到的消息的响应方式。因此，每个线程实际上都是一个状态机：当它接收到一条消息时，它会以某种方式更新自己的状态，可能会向其他线程发送一条或多条消息，执行的处理则取决于初始状态。编写这种线程的一种方法是将其形式化并实现一个有限状态机模型，但这不是唯一的方法；状态机可以隐式地存在于应用程序的结构中。在任何给定的场景中，哪种方法工作得更好取决于具体情况的行为需求和编程团队的专业知识。无论你选择如何实现每个线程，将线程分离为独立的处理都有潜力消除许多共享数据并发的复杂性，从而使编程更容易，降低 bug 率。

真正的通信顺序处理没有共享数据，所有通信都通过消息队列传递，但是由于 C++线程共享一个地址空间，因此不可能强制执行这一要求。这里就是规则的用武之地：作为应用程序或库的作者，我们有责任确保线程之间不共享数据。当然，为了让线程通信，必须共享消息队列，但是细节可以包装在库中。

假设你正在实现 ATM 的代码。该代码需要处理与试图取款的人的交互，和与相关银行的交互，以及控制物理机器来接受取款人的卡、显示相应的信息、处理按键、发钞和退卡。

处理这一切的一种方法是将代码分成三个独立的线程：一个处理物理机，一个处理 ATM 逻辑，一个与银行通信。这些线程可以纯粹通过传递消息而不是共享任何数据进行通信。例如，当一个人在机器上插卡或者按下一个按钮时，处理机器的线程可以发送一条消息给处理逻辑的线程，然后处理逻辑的线程可能发送一条消息给机器线程指示要分配多少钱，等等。

建模 ATM 逻辑的一种方法是将其作为状态机。在每个状态中，线程等待可接受的消息，然后处理该消息。这可能导致转移到一个新的状态，然后循环继续。一个简单实现所涉及的状态如图 4.3 所示。在这个简化的实现中，系统等待卡片被插入。一旦插入了卡，然后它就等待用户输入它们的 PIN，一次一个数字。他们可以删除最后一个输入的数字。一旦输入了足够的数字就校验 PIN。如果 PIN 不符合要求，则服务结束，因此把卡退回给客户，然后继续等待某个人插卡。如果 PIN 没问题，则要么等待他们取消事务或者选择一个取款金额。如果他们取消，服务就结束，然后退卡。如果他们选择一个金额，在发钞之前需要等待银行的确认，然后退卡或者显示一条“余额不足”的消息，再退卡。显然，一个真正的 ATM 要复杂得多，但这足以说明这种思想。

为 ATM 逻辑设计了状态机之后，就可以用一个类来实现它，这个类有成员函数表示每个状态。然后，每个成员函数可以等待特定的传入消息集，并在它们到达时处理它们，可能会触发到另一个状态的切换。每个不同的消息类型由一个独立的结构表示。清单 4.15 显示了这样一个系统中 ATM 逻辑的简单实现的一部分，其中主循环和第一个状态的实现都在等待插卡。

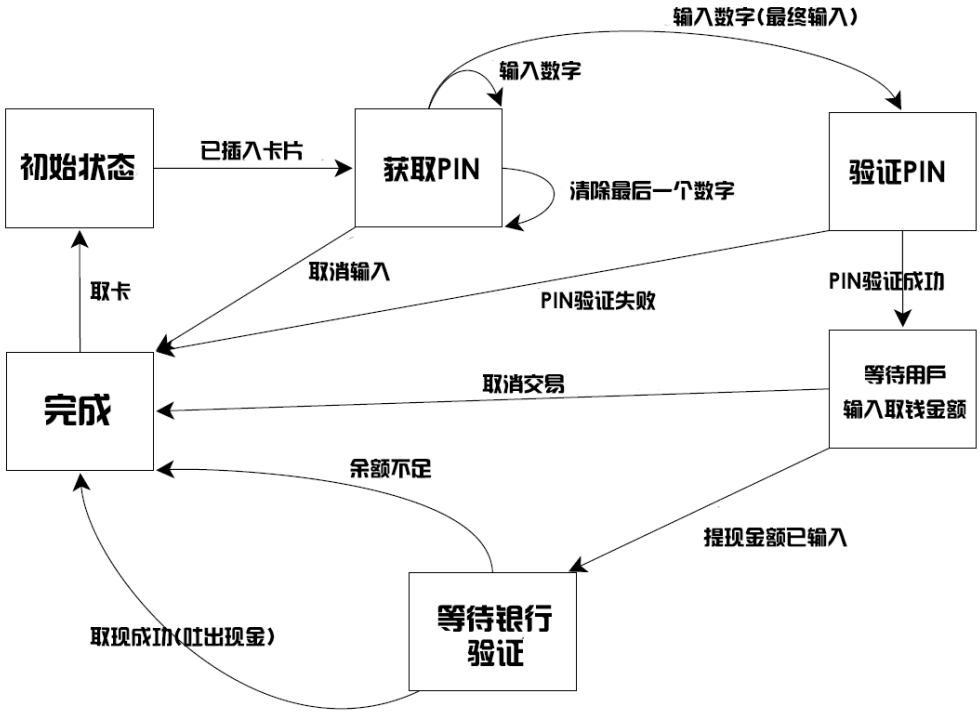


图 4.3 ATM 机的一个简化的状态机模型

如你所见，消息传递所需的所有同步都完全隐藏在消息传递库中(附录 C 给出了该库的基本实现以及本示例的完整代码)。

清单 4.15 ATM 逻辑类的简单实现

```
struct card_inserted
{
  std::string account;
};

class atm
{
  messaging::receiver incoming;
  messaging::sender bank;
  messaging::sender interface_hardware;
  void (atm::*state)();

  std::string account;
  std::string pin;

  void waiting_for_card()  // 1
  {
    interface_hardware.send(display_enter_card());  // 2
```

```
    incoming.wait().  // 3
      handle<card_inserted>(
      [&](card_inserted const& msg)  // 4
      {
       account=msg.account;
       pin="";
       interface_hardware.send(display_enter_pin());
       state=&atm::getting_pin;
      }
    );
  }
  void getting_pin();

public:
  void run()  // 5
  {
    state=&atm::waiting_for_card;  // 6
    try
    {
      for(;;)
      {
        (this->*state)();  // 7
      }
    }
    catch(messaging::close_queue const&)
    {
    }
  }
};
```

正如前面提到的，这里描述的实现是从 ATM 需要的实际逻辑中粗略地简化出来的，但是它确实让你对消息传递编程风格有一些感觉。不需要考虑同步和并发问题，只需要考虑在任何给定点接收哪些消息以及发送哪些消息即可。这种 ATM 逻辑的状态机运行在单个线程上，而系统的其他部分，如银行接口和终端接口运行在单独的线程上。这种风格的程序设计称为角色模型（Actor model）——系统中有几个离散的角色(每个角色运行在单独的线程上)，它们互相发送消息来执行手头的任务，除了直接通过消息传递的状态外，没有共享状态。

执行从 run()成员函数开始⑤，它设置初始状态为 waiting_for_card⑥，然后反复执行代表当前状态(不管它是什么状态)的成员函数⑦。这些状态函数是 atm 类的简单成员函数。wait_for_card 函数①也很简单：它发送一条消息到接口，以此来显示一条“等待插卡”的信息②，之后就等待消息进行处理③。这里唯一可以处理的消息类型是 card_inserted 消息，你可以使用 lambda 函数来处理它④。你可以传递任何函数或函数对象给处理函数，但是对于像这样简单的情况，使用 lambda 是最简单的。注意，handle()函

数调用被链接到 wait()函数上；如果接收到与指定类型不匹配的消息，则将其丢弃，并且线程继续等待，直到接收到匹配的消息。

lambda 函数本身在成员变量中缓存了来自卡上的帐号，清除当前 PIN，向接口硬件发送消息以显示要求用户输入 PIN 的内容，并更改状态为“获取 PIN”状态。一旦消息处理程序完成后，状态函数返回，然后主循环调用新的状态函数⑦。

getting_pin 状态函数稍微复杂一些，因为它可以处理三种不同类型的消息，如图 4.3 所示。下面的清单展示了它的实现。

清单 4.16 简单 ATM 实现中的 getting_pin 状态函数

```
void atm::getting_pin()
{
  incoming.wait()
    .handle<digit_pressed>(  // 1
      [&](digit_pressed const& msg)
      {
        unsigned const pin_length=4;
        pin+=msg.digit;
        if(pin.length()==pin_length)
        {
          bank.send(verify_pin(account,pin,incoming));
          state=&atm::verifying_pin;
        }
      }
      )
    .handle<clear_last_pressed>(  // 2
      [&](clear_last_pressed const& msg)
      {
        if(!pin.empty())
        {
          pin.resize(pin.length()-1);
        }
      }
      )
    .handle<cancel_pressed>(  // 3
      [&](cancel_pressed const& msg)
      {
        state=&atm::done_processing;
      }
      );
}
```

这一次，你可以处理三种消息类型，因此 wait()函数在末尾链接了三个 handle()调用，①②和③。每次对 handle()的调用都将消息类型指定为模板参数，然后传入一个

lambda 函数，该函数将该特定消息类型作为参数。因为调用以这种方式链接在一起，wait()实现知道它在等待一个 digit_ pressed 消息、一个 clear_last_pressed 消息或一个 cancel_pressed 消息。任何其他类型的消息都将被丢弃。

这一次，你不必在收到消息时更改状态。例如，如果你得到一个 digit_pressed 消息，除非它是最后一位数字，否则你只需要将其添加到 pin 中。清单 4.15 中的主循环⑦将会再次调用 getting_pin()去等待下一个数字(或清除数字，或取消交易)。

这与图 4.3 中所示的行为相对应。每个状态框由一个独立的成员函数实现，该函数等待相关消息并适当地更新状态。

如你所见，这种编程风格可以极大地简化设计并发系统的任务，因为可以完全独立地处理每个线程。它是使用多个线程分离关注点的示例，因此需要你明确地决定如何在线程之间划分任务。

在 4.2 节中，我提到了并发技术规范提供了期望的扩展版本。扩展的核心部分是指定*延续*（*continuations*）的能力——当期望就绪时，附带的函数自动运行。让我们借此机会探索这是如何简化我们代码的。

## 4.4.3 延续风格的并发与并发技术规范

并发技术规范在 std::experiment 命名空间中提供了新版本的 std::promise 和 std::packaged_task。与 std 命名空间中类型完全不同，其返回实例类型为 std::experimental::future，而不是 std::future。这使得用户能够使用 std::experimental::future 中的关键新特性——*延续*（*continuations*）。

假设你有一个正在运行的任务，它将产生一个结果，而期望将在该结果可用时持有该结果。然后需要运行一些代码来处理结果。对于 std::future，你必须等待期望准备就绪，或者使用完全阻塞的 wait()成员函数，或者使用 wait_for()或 wait_until()成员函数来允许超时等待。这可能不太方便，并可能使代码复杂化。你需要的是一种“当数据准备好了，就进行处理”的方法。这正是延续带给我们的；不出所料，用于向期望添加延续的成员函数叫做 then()。给定一个期望 fut，可以通过 fut.then(continuation)添加一个延续。

和 std::future 类一样， std::experimental::future 存储的值也只能被检索一次。如果那个值被一个延续消费了，这意味着别的代码不能访问它。因此，当一个延续用 fut.then()被添加时，原来的期望 fut 就失效了。相反，对 fut.then()的调用会返回一个新的期望来保存延续调用的结果。如下代码所示：

```
std::experimental::future<int> find_the_answer;
auto fut=find_the_answer();
auto fut2=fut.then(find_the_question);
assert(!fut.valid());
assert(fut2.valid());
```

find_the_question 延续函数计划在初始期望就绪时在“未指定的线程上”运行。这使得实现可以自由地在线程池或另一个库管理的线程上运行它。就目前而言，这给了实现很大的自由；这是有意为之，目的是当延续被添加到将来的 C++标准中时，实现者将能够利用他们的经验来更好地指定线程的选择，并为用户提供适当的机制来控制线程的选择。

与直接调用 std::async 或 std::thread 不同，不能将参数传递给延续函数，因为参数已经由库定义——传递给延续的是一个就绪的期望，其中包含触发延续的结果。假设你的 find_the_answer 函数返回一个 int，前面例子中引用的 find_the_question 函数必须使用 std::experimental::future<int>作为它的唯一参数；例如：

```
std::string find_the_question(std::experimental::future<int> the_answer
);
```

这样做的原因是，延续链接的期望可能最终持有一个值或一个异常。如果期望被隐式解除引用以致直接将值传递给延续，则库必须决定如何处理异常，而通过将期望传递给延续，延续就可以处理异常。在简单的情况下，这可能通过调用 fut.get()完成并且允许重新抛出的异常传播到延续函数的外面。就像传递给 std::async 的函数一样，从延续逃出的异常将存储在保存延续结果的期望中。

注意，并发技术规范没有指定 std::async 的等价物，尽管实现可能提供一个作为扩展。编写这样一个函数也是相当简单：使用 std::experimental::promise 获得一个期望，然后生成一个新线程运行 lambda，该线程将承诺的值设置为所提供函数的返回值，如下一个清单所示。

清单 4.17 为并发技术规范中的期望而实现的一个简单的 std::async 等价物

```
template<typename Func>
std::experimental::future<decltype(std::declval<Func>()())>
spawn_async(Func&& func){
  std::experimental::promise<
    decltype(std::declval<Func>()())> p;
  auto res=p.get_future();
  std::thread t(
    [p=std::move(p),f=std::decay_t<Func>(func)]()
      mutable{
      try{
        p.set_value_at_thread_exit(f());
      } catch(...){
        p.set_exception_at_thread_exit(std::current_exception());
      }
  });
  t.detach();
  return res;
}
```

这里在期望中存储该函数的结果，或者捕获该函数抛出的异常并将其存储在期望中，就像 std::async 所做的那样。另外，它使用 set_value_at_thread_exit 和

set_exception_at_thread_exit 来确保线程在期望就绪之前，线程局部变量已经被正确地清除了。

then()调用返回的值本身就是一个成熟的期望。这意味着你可以链结延续。

### 4.4.4 链接延续

假设你有一系列耗时的任务要执行，并且希望异步地执行它们，以便为其他任务释放主线程。例如，当用户登录到你的应用程序时，你可能需要将凭据发送到后端进行身份认证；然后，当详细信息经过身份认证后，进一步向后端请求有关用户帐户的信息；最后，当检索到该信息时，用相关信息更新显示。作为顺序代码，你可以编写如下清单所示的内容。

清单 4.18 处理用户登录的简单顺序函数

```
void process_login(std::string const& username, std::string const& pass
word)
{
  try
  {
    user_id const id = backend.authenticate_user(username, password);
    user_data const info_to_display = backend.request_current_info(id);
    update_display(info_to_display);
  }
  catch(std::exception& e)
  {
    display_error(e);
  }
}
```

但是，你不想要顺序代码；你需要异步代码，这样你就不会阻塞用户界面（UI）线程。使用普通的 std::async，你可以像下面的清单一样将其全部推给一个后台线程，但是这仍然会阻塞该线程，在等待任务完成时消耗资源。如果你有许多这样的任务，那么你最终可能会有大量线程，它们除了等待之外什么也不做。

清单 4.19 使用单个异步线程处理用户登录

```
std::future<void> process_login(
  std::string const& username, std::string const& password)
{
  return std::async(std::launch::async,[=](){
    try{
      user_id consst id = backend.authenticate_user(username, password)
;
      user_data const info_to_display =
```

```
        backend.request_current_info(id);
      update_display(info_to_display);
    } catch(std::exception& e){
      display_error(e);
    }
  });
}
```

为了避免所有这些阻塞线程，你需要某种机制来在每个任务完成时链接它们：延续。下面的清单显示了相同的整个流程，但这次分解为一系列任务，每个任务都链接到前一个任务上作为延续。

清单 4.20 使用延续处理用户登录的函数

```
std::experimental::future<void> process_login(
  std::string const& username, std::string const& password)
{
  return spawn_async([=]() {
    return backend.authenticate_user(username, password);
  }).then([](std::experimental::future<user_id> id) {
    return backend.request_current_info(id.get());
  }).then([](std::experimental::future<user_data> info_to_display) {
    try {
      update_display(info_to_display.get());
    } catch(std::exception& e) {
      display_error(e);
    }
  });
}
```

注意，每个延续都采用 std::experimental::future 作为唯一参数，然后使用.get()检索包含的值。这意味着异常可以在整条链上向下传播，所以在最后的延续中的 info_to_display.get()调用将抛出异常，如果链中的任何函数抛出一个异常的话，这里的 catch 块可以处理所有的异常，就像清单 4.18 中的 catch 块所做的那样。

如果到后端的函数调用因为它们在等待消息通过网络或数据库操作完成而内部阻塞了，那你就还没有完成操作。你可能已经将任务分割成单独的部分，但是它们仍然会阻塞调用，所以线程仍然会阻塞。你需要的是后端调用在数据准备好时返回就绪的期望，而不阻塞任何线程。在本例中，backend.async_authenticate_user(username,password)现在将返回 std::experimental::future<user_id>，而不是普通的 user_id。

你可能认为这会使代码复杂化，因为从延续返回一个期望会得到 future<future<some_value>>，否则你必须将 then 调用放入延续中。幸运的是，如果你这么想，那你就错了：延续支持有一个叫做期望展开（future-unwrapping）的漂亮特性。如果传递给 then()调用的延续函数返回一个 future<some_type>，则 then()调用将依次返回

一个 future<some_type>。这意味着你最终的代码类似于下一个清单，并且在异步函数链中没有阻塞。

清单 4.21 全异步操作处理用户登录的函数

```
std::experimental::future<void> process_login(
  std::string const& username, std::string const& password)
{
  return backend.async_authenticate_user(username, password).then(
    [](std::experimental::future<user_id> id){
      return backend.async_request_current_info(id.get());
    }).then([](std::experimental::future<user_data> info_to_display){
      try{
        update_display(info_to_display.get());
      } catch(std::exception& e){
        display_error(e);
      }
    });
}
```

这几乎和清单 4.18 中的顺序代码一样简单，只是在.then 调用和 lambda 声明周围多了点样板。如果你的编译器支持 C++14 泛型 lambda，那么 lambda 参数中的期望类型可以用 auto 替换，这将进一步简化了代码：

```
return backend.async_authenticate_user(username, password).then(
    [](auto id){
      return backend.async_request_current_info(id.get());
    });
```

如果你需要比简单的线性控制流程更复杂的东西，那么你可以通过把逻辑放到其中的一个 lambda 来实现；对于真正复杂的控制流，可能需要写一个单独的函数。

到目前为止，我们主要关注 std::experimental::future 中的延续支持。如你所料，std::experimental::shared_future 也支持延续。这里的区别是 std::experimental::shared_future 对象可以有多个延续，并且延续的参数是 std::experimental::shared_future，而不是 std::experimental::future。这自然源于 std::experimental::shared_future 的共享性质——因为多个对象可以引用相同的共享状态，如果只有一个延续被允许，就会有两个线程之间的竞争条件，它们都试图添加延续到它们自己的 std::experimental::shared_future 对象中。这显然是不希望的，因此允许使用多个延续。一旦允许多个延续，你也可以允许通过相同的 std::experimental::shared_future 实例添加它们，而不是每个对象只允许一个延续。此外，你不能将共享状态打包在一次性的 std::experimental::future 中传递给第一个延续（当你还想将它传递给第二个延续的时候）。因此传递给延续函数的参数也必须是 std::experimental::shared_future：

```
auto fut = spawn_async(some_function).share();
```

```
auto fut2 = fut.then([](std::experimental::shared_future<some_data> dat
a){
      do_stuff(data);
    });
auto fut3 = fut.then([](std::experimental::shared_future<some_data> dat
a){
      return do_other_stuff(data);
    });
```

由于 share()调用，fut 是一个 std::experimental::shared_future，因此延续函数必须采用 std::experimental::shared_future 作为参数。但是，从延续返回的值是一个普通的 std::experimental::future——该值目前还没有共享，直到你做了一些事情来共享它——因此 fut2 和 fut3 都是 std::experimental::future。

延续并不是并发技术规范中对期望的唯一增强，尽管它们可能是最重要的。技术规范也提供了两个重载函数，允许你等待一堆期望中的任何一个变成就绪，或者等待所有的期望准备就绪。

## 4.4.5 等待多个期望

假设你有大量数据要处理，并且每个项都可以独立处理。这是通过生成一组处理数据项的异步任务来利用可用硬件的最佳机会，每个任务通过期望返回处理过的数据。但是，如果你需要等待所有任务完成，然后收集所有结果进行最终处理，这可能会很不方便——你必须依次等待每个期望，然后收集结果。如果你想用另一个异步任务来收集结果，你要么必须提前创建它，这样它就会占用一个线程来等待，要么必须继续轮询期望，并在所有期望就绪时创建新任务。下面的清单中显示了此类代码的示例。

清单 4.22 使用 std::async 从多个期望值中收集结果

```
std::future<FinalResult> process_data(std::vector<MyData>& vec)
{
  size_t const chunk_size = whatever;
  std::vector<std::future<ChunkResult>> results;
  for (auto begin=vec.begin(), end=vec.end(); beg!=end;){
    size_t const remaining_size = end - begin;
    size_t const this_chunk_size = std::min(remaining_size, chunk_size)
;
    results.push_back(
      std::async(process_chunk, begin, begin+this_chunk_size));
    begin += this_chunk_size;
  }
  return std::async([all_results=std::move(results)](){
    std::vector<ChunkResult> v;
    v.reserve(all_results.size());
```

```
    for (auto& f : all_results)
    {
      v.push_back(f.get()); // 1
    }
    return gather_results(v);
  });
}
```

这段代码生成一个新的异步任务来等待结果，然后在结果全部可用时处理它们。但是，由于它分别等待每个任务，所以当每个结果可用时，它将在①重复被调度器唤醒，然后当它发现另一个尚未准备好的结果时再次返回休眠状态。这不仅占用了线程执行等待，而且在每个期望准备就绪时增加了额外的上下文切换，从而增加了额外的开销。

使用 std::experimental::when_all 可以避免这种等待和切换。你将等待的期望集合传递给 when_all，当集合中所有的期望都准备好时，它就会返回一个新的处于就绪状态的期望。然后，当所有的期望都准备好时，这个期望可以与延续一起使用来安排额外的工作。例如，下一个清单。

清单 4.23 使用 std::experimental::when_all 从多个期望值中收集结果

```
std::experimental::future<FinalResult> process_data(
  std::vector<MyData>& vec)
{
  size_t const chunk_size = whatever;
  std::vector<std::experimental::future<ChunkResult>> results;
  for (auto begin = vec.begin(), end = vec.end(); beg != end){
    size_t const remaining_size = end - begin;
    size_t const this_chunk_size = std::min(remaining_size, chunk_size)
;
    results.push_back(
      spawn_async(
        process_chunk, begin, begin+this_chunk_size));
    begin += this_chunk_size;
  }
  return std::experimental::when_all(
    results.begin(), results.end()).then( // 1
      [](std::future<std::vector<std::experimental::future<ChunkResult>
>> ready_results){
        std::vector<std::experimental::future<ChunkResult>> all_results
 = ready_results.get();
        std::vector<ChunkResult> v;
        v.reserve(all_results.size());
        for (auto& f: all_results){
          v.push_back(f.get()); // 2
        }
```

```
        return gather_results(v);
      });
}
```

在本例中,你使用 when_all 等待所有期望变成就绪状态，然后使用.then 而不是 async 来调度函数。尽管 ladmbda 表面上是相同的，它需要结果的 vector 作为一个参数(包装在一个期望中)而不是捕获，并且在期望上调用 get②不会阻塞，因为执行到这的时候所有的值都准备好了。这里只是对代码进行了少许更改就有可能减少系统上的负载。

作为 when_all 的补充，我们还有 when_any。当任何一个提供的期望准备好的时候，它会创建了一个期望。这适用于这样的场景：你已经创建了多个任务以利用可用的并发，但需要在第一个任务就绪时执行某些操作。

## 4.4.6 使用 when_any 等待集合中第一个期望

假设你正在一个大型数据集中搜索满足特定条件的值，但是如果有多个这样的值，那么任何一个都可以。这是并行的主要目标——你可以创建多个线程，每个线程检查数据的一个子集；如果一个给定的线程找到了一个合适的值，那么它会设置一个标记，指示其他线程应该停止它们的搜索，然后设置最终的返回值。这种情况下，当第一个任务完成它的搜索时你想要执行进一步的处理，即使其他任务还没有完成清理。

在这里，你可以使用 std::experimental::when_any 将期望收集在一起，并在原始集合中至少有一个准备好时提供一个新的期望。而 when_all 给了你一个期望，它包装了你传入期望的集合，when_any 又增加了一个层，将集合与一个索引值组合进 std::experimental::when_any_result 类模板的一个实例，那个索引值指示哪个期望触发了组合的期望变成就绪状态。

下一个清单展示了使用 when_any 的示例。

清单 4.24 使用 std::experimental::when_any 处理第一个被找到的值

```
std::experimental::future<FinalResult>
find_and_process_value(std::vector<MyData> &data)
{
  unsigned const concurrency = std::thread::hardware_concurrency();
  unsigned const num_tasks = (concurrency > 0)? concurrency : 2;

  std::vector<std::experimental::future<MyData *>> results;
  auto const chunk_size = (data.size() + num_tasks - 1) / num_tasks;
  auto chunk_begin = data.begin();
  std::shared_ptr<std::atomic<bool>> done_flag =
      std::make_shared<std::atomic<bool>>(false);

  for (unsigned i = 0; i < num_tasks; ++i){ // 1
```

```
    auto chunk_end =
      (i < (num_tasks - 1)? chunk_begin + chunk_size : data.end());
    results.push_back(spawn_async([=]{ // 2
      for (auto entry = chunk_begin;
           !*done_flag && (entry != chunk_end);
           ++entry){
        if (matches_find_criteria(*entry)){
          *done_flag = true;
          return &*entry;
        }
      }
      return (MyData *)nullptr;
    }));
    chunk_begin = chunk_end;
  }

  std::shared_ptr<std::experimental::promise<FinalResult>> final_result
=
    std::make_shared<std::experimental::promise<FinalResult>>();

  struct DoneCheck {
    std::shared_ptr<std::experimental::promise<FinalResult>>
      final_result;

  DoneCheck(
    std::shared_ptr<std::experimental::promise<FinalResult>>
      final_result_)
    : final_result(std::move(final_result_)) {}

  void operator()( // 4
    std::experimental::future<std::experimental::when_any_result<
      std::vector<std::experimental::future<MyData *>>>>
        results_param) {
    auto results = results_param.get();
    MyData *const ready_result =
      results.futures[results.index].get(); // 5

    if (ready_result)
      final_result->set_value( // 6
        process_found_value(*ready_result));
    else {
      results.futures.erase(
        results.futures.begin() + results.index); // 7
      if (!results.futures.empty()) {
```

```
        std::experimental::when_any( // 8
          results.futures.begin(), results.futures.end())
          .then(std::move(*this));
      } else {
        final_result->set_exception(
          std::make_exception_ptr( // 9
            std::runtime_error("Not found")));
      }
    }
  };

  std::experimental::when_any(results.begin(), results.end())
  .then(DoneCheck(final_result)); // 3
  return final_result->get_future(); // 10
}
```

初始循环①创建了 num_tasks 个异步任务，每个任务都会执行②处的 lambda 函数。这个 lambda 按值捕获，所以每个任务都有自己的 chunk_begin 和 chunk_end，这里同样也拷贝了共享指针 done_flag。这就避免了生命周期所带来的问题。

一旦所有任务都被创建，你就需要处理有一个任务返回的情况。这是通过在 when_any 调用上链接一个延续来实现的③。这一次，你将延续编写为一个类，因为你希望递归地重用它。当一个初始任务就绪时，DoneCheck 函数调用操作符被调用④。首先从就绪的期望中抽取值⑤，然后，如果找到了这个值，就对它进行处理并设置最终结果⑥。否则，就需要从集合中丢弃就绪的期望⑦，如果还有更多的期望需要检查，就会向 when_any 发出一个新的调用⑧，当下一个期望准备好了，这将触发它的延续。如果没有期望剩下，那么没有一个期望找到该值，因此代之存储一个异常⑨。函数的返回值是最终结果的期望⑩。还有其他方法可以解决这个问题，但我希望这可以展示如何使用 when_any。

使用 when_all 和 when_any 的这些示例都使用了迭代器范围重载的版本，它使用两个迭代器表示要等待的一组期望的开始和结束。这两个函数都有可变形式，它们接受一定数量的期望直接作为函数的参数。在这种情况下，结果是一个持有元组的期望(或者是一个持有元组的 when_any_result)，而不是一个 vector：

```
std::experimental::future<int> f1=spawn_async(func1);
std::experimental::future<std::string> f2=spawn_async(func2);
std::experimental::future<double> f3=spawn_async(func3);
std::experimental::future<
  std::tuple<
    std::experimental::future<int>,
    std::experimental::future<std::string>,
    std::experimental::future<double>>> result=
  std::experimental::when_all(std::move(f1),std::move(f2),std::move(f3)
);
```

这个示例强调了关于使用 when_any 和 when_all 的所有重要内容——它们总是移动通过容器传入的任何 std::experimental::future，并且按值获取参数，因此你必须显式地将期望移进去，或者是传递临时对象。

有时，你等待的事件是等待一组线程到达代码中的某个特定点，或者在它们之间处理了一定数量的数据项。在这些情况下，使用锁存器或屏障可能比使用期望更好。让我们看看并发技术规范提供的锁存器和屏障。

## 4.4.7 并发技术规范中的锁存器和屏障

首先，让我们考虑一下当我们谈到*锁存器*（latch）或*屏障*（barrier）时是指什么。锁存器是一个同步对象，当它的计数器减到零时，它就准备好了。它的名字来自于它*闩住*（latches）输出——一旦它准备好了，它就会一直保持就绪状态，直到它被销毁。因此，锁存器是等待一系列事件发生的轻量级工具。

另一方面，屏障是一个可重用的同步组件，用于一组线程之间的内部同步。锁存器并不关心哪个线程减少计数器——同一个线程可以减少计数器多次，或者多个线程可以每个线程减少计数器一次，或者两者的某种组合——对于屏障，每个线程每个周期只能到达屏障一次。当线程到达屏障时，它们就会阻塞，直到所有相关的线程都到达屏障，此时它们都被释放。这时屏障可以被重用——然后线程可以再次到达屏障，等待所有线程进入下一个周期。

锁存器本质上要比屏障简单很多，我们就先从 std::experimental::latch 说起。

## 4.4.8 基本的锁存器类型：std::experimental::latch

std::experimental::latch 来自于<experimental/latch>头文件。当你构造一个 std::experimental::latch 时，你将初始计数器值指定为构造函数的唯一参数。然后，当等待的事件发生时，在锁存器对象上调用 count_down，当计数为 0 锁存器就准备好了。如果你需要等待锁存器准备好，那你可以在锁存器上调用 wait；如果你只需要检查它是否准备好了，那你可以调用 is_ready。最后，如果你需要同时递减计数器，并且等待计数器达到零，你可以调用 count_down_and_wait。下面的清单中显示了一个基本示例。

清单 4.25 使用 std::experimental::latch 等待事件

```
void foo(){
  unsigned const thread_count=...;
  latch done(thread_count); // 1
  my_data data[thread_count];
  std::vector<std::future<void> > threads;

  for(unsigned i=0;i<thread_count;++i)
    threads.push_back(std::async(std::launch::async,[&,i]{ // 2
```

```
      data[i]=make_data(i);
      done.count_down(); // 3
      do_more_stuff(); // 4
    }));

  done.wait(); // 5
  process_data(data,thread_count); // 6
} // 7
```

使用需要等待的事件数量构造 done①，然后使用 std::async 产生适量的线程②。在继续进一步的处理前④，当线程生成相应的数据块时，都会对锁存器的计数器进行递减③。在处理生成的数据⑥之前，主线程通过等待锁存器来等待所有数据准备好⑤。⑥处的数据处理可能会与每个线程最终的处理步骤同步进行④——在 std::future 析构函数在函数末尾运行之前，不能保证线程已经全部完成⑦。

需要注意的是，在②传递给 std::async 的 lambda 表达式中，除了 i 是按值捕获外，其他都按引用捕获。这是因为 i 是循环计数器，通过引用捕获它将导致数据竞争和未定义行为，而 data 和 done 是你需要共享访问的内容。此外，在这种情况下，你只需要一个锁存器，因为在数据准备好之后，线程还有额外的处理要做；否则，你可以在处理数据之前等待所有的期望来确保任务完成。

process_data 调用中对 data 的访问是安全的⑥，即便这个值是由其他线程上的任务存储的，这是因为锁存器是一个同步对象，因此，对于调用 count_down 的线程可见的更改，保证对从同一锁存器对象上等待调用返回的线程可见（译注：也就是③之前的修改，对⑤之后的主线程都是可见的，因为主线程是在⑤处从 done 锁存器的 wait 调用返回的）。形式上，count_down 调用与 wait 调用是同步的——当我们在第 5 章中查看低层次的内存顺序和同步约束时，我们将看到这意味着什么。

除了锁存器之外，并发技术规范还为我们提供了屏障——用于同步一组线程的可重用同步对象。接下来让我们看看这些。

## 4.4.9 std::experimental::barrier：一个基本的屏障

并发技术规范在<experimental/barrier>头文件中提供了两种类型的屏障：std::experimental::barrier 和 std::experimental::flex_barrier。前者更基础，因此潜在的开销更低，而后者更灵活，但潜在的开销更大。

假设有一组线程正在操作某些数据。每个线程可以独立于其他线程对数据进行处理，因此在处理期间不需要同步，但是在处理下一个数据项之前，或者在完成后续处理之前，所有线程必须完成它们的处理。std::experimental::barrier 正是针对这个场景。你可以使用同步组中线程数量的计数构造一个屏障。当每个线程完成其处理时，它会到达屏障，并通过在屏障对象上调用 arrive_and_wait 来等待组中剩余的线程。当组中的最后一个线程到达时，所有线程都被释放，屏障被重置。然后，组中的线程可以继续它们的处理，并根据需要处理下一个数据项或进行下一阶段的处理。

锁存器会闩住，所以一旦它们准备好了，它们就会保持就绪状态，屏障就不是这样——屏障会释放等待的线程，然后重新设置，这样它们就可以再次使用。它们也只在一组线程内同步——一个线程不能等待屏障准备就绪，除非它是同步组中的线程之一。通过在屏障上调用 arrive_and_drop，线程可以明确退出组，在这种情况下，那个线程再也不能等待屏障就绪，并且在下一个周期必须到达的线程数量会比当前到达的线程数量少一。

清单 4.26 使用 std::experimental::barrier

```
result_chunk process(data_chunk);
std::vector<data_chunk>
divide_into_chunks(data_block data, unsigned num_threads);

void process_data(data_source &source, data_sink &sink) {
  unsigned const concurrency = std::thread::hardware_concurrency();
  unsigned const num_threads = (concurrency > 0) ? concurrency : 2;

  std::experimental::barrier sync(num_threads);
  std::vector<joining_thread> threads(num_threads);

  std::vector<data_chunk> chunks;
  result_block result;

  for (unsigned i = 0; i < num_threads; ++i) {
    threads[i] = joining_thread([&, i] {
      while (!source.done()) { // 6
        if (!i) { // 1
          data_block current_block =
            source.get_next_data_block();
          chunks = divide_into_chunks(
            current_block, num_threads);
        }
        sync.arrive_and_wait(); // 2
        result.set_chunk(i, num_threads, process(chunks[i])); // 3
        sync.arrive_and_wait(); // 4
        if (!i) { // 5
          sink.write_data(std::move(result));
        }
      }
    });
  }
} // 7
```

清单 4.26 的示例显示了使用屏障同步的一组线程。你有来自 source 的数据，并将输出写入到 sink，但是为了利用系统中可用的并发，你将每个数据块分割为 num_threads 块。这个操作是串行的，所以你有一个初始数据块①只运行在 i 为 0 的线程上。然后，在

到达并行区域之前，所有线程都在屏障上等待该串行代码完成②，而后每个线程都会处理属于自己的数据块，并且再次同步之前④，将结果更新到 result 中③。然后就会到达第二个串行处理区域，这里只有 0 号线程可以将结果输出到 sink 中⑤。然后所有线程继续循环，直到 source 报告所有数据都完成⑥。注意，当每个线程循环时，循环底部的串行部分与顶部的部分可以合起来；因为在这两个部分中只有线程 0 需要做事情，所以这是可以的，并且所有线程在第一次使用屏障时都将同步在一起②。当所有处理完成时，所有线程将退出循环，所有 joining_thread 对象的析构函数将在外部函数结束时等待它们完成⑦（在第 2 章的清单 2.7 中介绍了 joining_thread）。

这里要注意的关键一点是，对 arrive_and_wait 的调用位于代码中最重要的一点，即在所有线程准备就绪之前，不要继续执行任何线程。在第一个同步点，所有线程都在等待线程 0 的到达，但是屏障的使用为你提供了一条清晰的路线。在第二个同步点，情况正好相反：线程 0 等待所有其他线程到达，然后才能将完成的 result 写入 sink。

并发技术规范不止提供了一种屏障类型；除了 std::experimental::barrier 之外，还有 std::experimental::flex_barrier，它更加灵活。它更灵活的一种方式是，在所有线程释放之前，它允许在所有线程都到达屏障时运行最终的一个串行区域。

## 4.4.10 std::experimental::flex_barrier—std::experimental::barrier 的灵活朋友

std::experimental::flex_barrier 与 std::experimental::barrier 的接口只有一点不同：它有一个额外的构造函数，需要传入一个结束函数，以及线程数量。当所有线程都到达屏障时，这个函数将只在其中一个线程上运行。它不仅提供了指定必须串行运行的代码块的方法，还提供了更改下一个周期必须到达屏障的线程数量的方法。线程计数可以更改为任何数字，无论是高于或低于之前的计数；使用这个设施的程序员需要确保下一次到达屏障时，线程数量是正确的。

下面的清单显示了如何重写清单 4.26 以使用 std::experimental::flex_barrier 来管理串行区域。

清单 4.27 使用 std::experimental::flex_barrier 提供串行区域

```
void process_data(data_source &source, data_sink &sink) {
  unsigned const concurrency = std::thread::hardware_concurrency();
  unsigned const num_threads = (concurrency > 0) ? concurrency : 2;

  std::vector<data_chunk> chunks;

  auto split_source = [&] { // 1
    if (!source.done()) {
      data_block current_block = source.get_next_data_block();
      chunks = divide_into_chunks(current_block, num_threads);
```

```
    }
  };

  split_source(); // 2

  result_block result;

  std::experimental::flex_barrier sync(num_threads, [&] { // 3
    sink.write_data(std::move(result));
    split_source(); // 4
    return -1; // 5
  });
  std::vector<joining_thread> threads(num_threads);

  for (unsigned i = 0; i < num_threads; ++i) {
    threads[i] = joining_thread([&, i] {
      while (!source.done()) { // 6
        result.set_chunk(i, num_threads, process(chunks[i]));
        sync.arrive_and_wait(); // 7
      }
    });
  }
}
```

与清单 4.26 的第一个不同在于，这里使用一个 lambda 对数据进行拆分①。它在开始前被调用②，并且封装了在每个迭代开始的时候运行在线程 0 上的代码。

第二个区别是，你的 sync 对象现在是一个 std::experimental::flex_barrier，你正在传递一个完成函数和一个线程计数③。这个函数在每个线程都到达后，在某个线程上运行，所以封装了在每次迭代结束时运行在线程 0 上的代码，然后有一个对新提取的 split_source 的调用，这样的话，在下个迭代开始后的时候，它已经被调用④。返回值-1⑤表示参与的线程数目保持不变；返回值 0 或其他数值则指定的是下一个周期中参与的线程数量。

主循环⑥现在就简单了：它只包含了并行部分的代码，所以只需要一个同步点就够了⑦。因而使用 std::experimental::flex_barrier 简化了代码。

使用完成函数提供串行部分的功能非常强大，更改参与线程数量的能力也是如此。例如，这可以用于流水线风格的代码，在这样的代码中，当所有阶段的流水线运转的时候，初始装填流水线以及最后排干流水线的线程数量比主处理的线程数要少。

---

[2] 《通信顺序进程》(*Communicating Sequential Processes*), C.A.R. Hoare, Prentice Hall, 1985. 免费在线阅读地址 http://www.usingcsp.com/cspbook.pdf.

# 总结

线程之间的同步操作是编写使用并发的应用程序的一个重要部分：如果没有同步，线程本质上是独立的，也可以作为单独的应用程序编写，这些应用程序由于它们的相关活动而作为一个组运行。在这一章，讨论了各种同步操作，从基本的条件变量，到期望，承诺，打包任务，锁存器，以及屏障。还讨论了处理同步问题的方法：函数式编程，其中每个任务产生的结果完全依赖于其输入，而不是外部环境；消息传递，线程间的通信是通过充当中介的消息子系统发送的异步消息进行的；以及延续风格，延续可以指定每个操作的后续任务，然后系统会负责调度。

在讨论了 C++中可用的许多高层次的设施之后，现在是时候来看看使其工作的低层次的设施了：C++内存模型和原子操作。

# 第 5 章 C++内存模型和原子类型操作

## 本章主要内容

- C++内存模型细节
- C++提供的原子类型
- 标准库
- 原子类型上的操作
- 如何使用这些操作来提供线程之间的同步

C++标准中，有一个十分重要的特性，被大多数程序员所忽略。它不是一个新语法特性，也不是新的库设施，而是新的多线程感知内存模型。如果没有内存模型准确地定义基本构建块的工作方式，那么我所介绍的所有设施都无法正常工作。有一个原因大多数程序员都不会注意到：如果你使用互斥锁来保护你的数据，以及条件变量、期望、锁存器或屏障来用信号通知事件，那它们为什么有效的细节并不重要。只有当你开始尝试“接触硬件”（close to the machine）时，内存模型的精确细节才变得重要。

不管怎么说，C++是一个系统级别的编程语言，标准委员会的目标之一就是不需要比C++还要底层的高级语言。在C++中应该为程序员提供足够的灵活性来做他们需要的任何事情，而不受语言的阻碍，当需要的时候允许他们“接触硬件”。原子类型和操作正好允许这样做，它们为低层次同步操作提供的设施通常只需要1~2个CPU指令。

在本章中，我将首先介绍内存模型的基础知识，然后介绍原子类型和操作，最后介绍原子类型操作中各种可用的同步类型。这相当复杂：除非你计划编写使用原子操作进行同步的代码(如第7章中的无锁数据结构)，否则你不需要知道这些细节。

让我们先来看一下有关内存模型的基本知识。

# 5.1 内存模型基础

内存模型包含两个方面：基本*结构*（structural）方面（它与事物在内存中的布局有关）和并发方面。结构方面对于*并发*（conrurrency）非常重要，特别是在考虑低层次原子操作的时候，因此我将从它开始。在C++中，结构方面全部是关于对象和内存位置的内容。

## 5.1.1 对象和内存位置

C++程序中的所有数据都是由*对象*（objects）组成的。这并不是说你可以创建一个从int派生的新类，也不是说基本类型具有成员函数，也不是说在讨论Smalltalk或Ruby这样的语言时，当人们说所有东西都是对象时，通常会暗指的任何其他结果。它是关于C++

中构建数据块的声明。C++标准将一个对象定义为“一个存储区域”（a region of storage），尽管它继续为这些对象分配属性，比如它们的类型和生命周期。

其中一些对象是基本类型，如 int 或 float 的简单值，而其他对象是用户自定义类的实例。有些对象(例如数组、派生类的实例和具有非静态数据成员的类的实例)有子对象，但其他对象没有。

无论对象的类型是什么，它都存储在一个或多个内存位置中。每个内存位置要么是标量类型(如 unsigned short 或 my class*)的对象(或子对象)，要么是相邻位域的序列。如果你使用位域，需要注意的重要一点是尽管相邻的位域是不同的对象，但它们仍然被算作相同的内存位置。图 5.1 显示了一个 struct 如何划分成对象和内存位置。

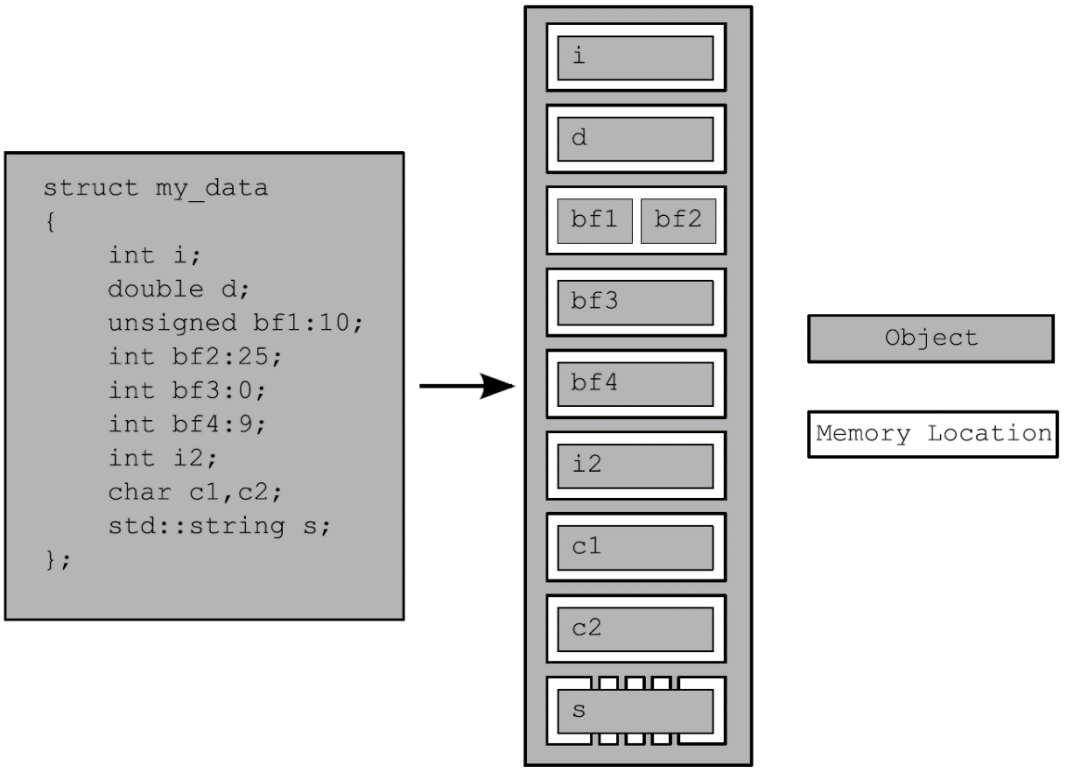


图 5.1 把一个 struct 分割为对象和内存位置

首先，整个 struct 是一个由几个子对象组成的对象，每个子对象对应一个数据成员。bf1 和 bf2 位字段共享一个内存位置，std::string 对象 s 内部由几个内存位置组成，但除此之外，每个成员都有自己的内存位置。注意零长度位字段 bf3(bf3 这个名字是用来注释用的，因为 C++中零长度位字段必须是未命名的)如何将 bf4 分隔到它自己的内存位置，但它本身没有内存位置。(译注：C++中零长度的未命名位域有特殊含义：让下一个位域从分配单元的边界开始。C++中位域的内存布局是与机器相关的，取地址操作符&不能作用于位域)。

从中我们可以了解到四件重要的事情：

1. 每一个变量都是一个对象，包括作为其他对象成员的变量。

2. 每个对象至少占有一个内存位置。
3. 基本类型的变量，如 int 或 char，无论它们的大小如何，即使它们是相邻的或数组的一部分，也只占用一个内存位置。
4. 相邻位域是相同内存位置的一部分。

我敢肯定你想知道这与并发有什么关系，所以让我们来看一下。

## 5.1.2 对象、内存位置和并发

现在，这里是对 C++中的多线程应用程序至关重要的部分：一切都取决于那些内存位置。如果两个线程访问单独的内存位置，没有问题：一切工作正常。另一方面，如果两个线程访问相同的内存位置，那你必须小心。如果两个线程都没有更新内存位置，那就没事；只读数据不需要保护或同步。如果任何一个线程正在修改数据，就可能出现竞争条件，如第 3 章所述。

为了避免竞争条件，两个线程的访问之间必须要有一个强制顺序。这可以是一个固定的顺序，这样一个访问总是在另一个之前，也可以是一个在应用程序的运行之间变化的顺序，但保证有某种已定义的顺序。确保有序的一种方法是使用第 3 章中描述的互斥锁；如果在两次访问之前都锁住了同一个互斥锁，那么每次只有一个线程可以访问该内存位置，所以一个必须在另一个之前发生（尽管，通常你没法提前知道谁先谁后）。另一种方法是在相同的或其他内存位置上使用原子操作的同步属性(关于原子操作的定义，请参阅 5.2 节)来强制两个线程的访问顺序。5.3 节描述了使用原子操作来强制一个顺序。如果有两个以上的线程访问相同的内存位置，那么每对访问都必须有一个定义好的顺序。

如果从不同的线程对单个内存位置的两次访问之间没有强制顺序，那么其中一次或两次访问都不是原子的，并且如果其中一次或两次访问都是写操作，那这就是数据竞争，会导致未定义的行为。

这个声明非常重要：未定义的行为是 C++中最肮脏的角落之一。根据语言标准，一旦应用程序包含任何未定义的行为，将满盘皆输；整个应用程序的行为现在是未定义的，它可以做任何事情。我知道有一个例子，一个未定义的行为导致某人的监视器着火了。虽然这种情况不大可能发生在你身上，但数据竞争绝对是一个严重的错误，应该不惜一切代价加以避免。

该声明还有另一个要点：你还可以通过使用原子操作访问竞争中涉及的内存位置来避免未定义的行为。这并不能防止竞争本身——哪个原子操作首先接触内存位置仍然没有指定——但它确实将程序带回到已定义行为的领域。

在讨论原子操作之前，还有一个关于对象和内存位置的重要概念：修改顺序。

### 5.1.3 修改顺序

C++程序中的每个对象都有一个*修改顺序*（modification order），由程序中所有线程对该对象的所有写操作组成，始于对象初始化。在大多数情况下，这个顺序在运行时是不同的，但是在任何给定的程序执行中，系统中的所有线程都必须对这个顺序达成一致。如果涉及的对象不是 5.2 节中描述的原子类型，那么你要负责确保有足够的同步，以确保线程对每个变量的修改顺序达成一致。如果不同的线程看到一个变量值的不同序列，那么就会出现数据竞争和未定义行为(参见 5.1.2 节)。如果你确实使用了原子操作，则编译器负责确保必要的同步。

这一要求意味着某些类型的推测执行是不允许的，因为一旦一个线程看到了修改顺序中的特定条目，随后那个线程的读操作必须返回之后的值，并且那个线程随后对那个对象的写操作必须在修改顺序中更晚发生。同样，在同一个线程中，在对该对象进行写操作之后对该对象进行读操作，必须要么返回已写的值，要么返回在修改顺序中更晚出现的另一个值。虽然所有线程必须对程序中每个单独对象的修改顺序达成一致，但它们不必对不同对象的相对操作顺序达成一致。关于线程之间的操作顺序，请参阅 5.3.3 节。

那么，什么构成了一个原子操作？如何使用它们来强制顺序呢？

## 5.2 C++中的原子操作和类型

*原子操作*（atomic operation）是不可分割的操作。系统中任何线程都不可能观察到操作只完成一半；要么做了，要么没做。如果读取对象值的加载操作是*原子的*（atomic），并且对该对象的所有修改也是*原子的*（atomic），那么加载将检索到对象的初始值或其中一个修改所存储的值。

另一方面，非原子的操作可能被另一个线程观察到只完成一半。如果非原子操作是由原子操作(例如，赋值给成员是 atomic 的 struct)组成，那么其他线程可能把某个组成原子操作的一个子集视作完整的，但是其他操作还没有开始，所以你可能会观察到，或最后得到一个值，这个值是各种存储值的混乱组合。在任何情况下，对非原子变量的非同步访问都会形成一个简单的有问题的竞争条件，如第 3 章所述，但是在这个层次上，它可能会构成一个*数据竞争*（data race）(参见 5.1 节)并导致未定义的行为。

在 C++中，在大多数情况下，需要使用原子类型来获得原子操作，所以让我们来看看这些原子类型。

### 5.2.1 标准原子类型

标准*原子类型*（atomic types）可以在头文件<atomic>中找到。这些类型的所有操作都是原子的，并且从语言定义讲只有这些类型的操作是原子的，尽管可以使用互斥锁让其他操作看起来是原子的。实际上，标准原子类型它们本身就可能使用这样的仿真：它们(几乎)都有一个 is_lock_free()成员函数，这个函数可以让用户查询给定类型上的操作是直

接用原子指令(x.is_lock_free()返回 true)实现的，还是通过使用编译器和库内部的锁实现的(x.is_lock_free()返回 false)。

在很多情况下了解这一点是很重要的——原子操作的关键用例是替换那些使用互斥锁进行同步的操作；如果原子操作本身使用内部互斥锁，那么期望的性能增益可能不会实现，你最好使用更容易搞定的基于互斥锁的实现。第 7 章中讨论的无锁数据结构就是这样。

事实上，这一点是如此重要，以至于库提供了一组宏，用于在编译时识别各种整型的原子类型是否无锁。自从 C++17 以后，所有的原子类型都有一个静态的 constexpr 成员变量，X::is_always_lock_free，当且仅当原子类型 X 对于当前编译的输出可能运行的所有支持的硬件都是无锁的时候，这个变量的值才为 true。例如，对于给定的目标平台，std::atomic<int>可能总是无锁的，所以 std::atomic<int>::is_always_lock_free 将为真，但是 std::atomic<uintmax_t>可能只有在程序最终运行的硬件支持必要的指令时才无锁。所以这是一个运行时属性，std::atomic<uintmax_t>::is_always_lock_free 在为该平台编译时将为 false。

这些宏是 ATOMIC_BOOL_LOCK_FREE、ATOMIC_CHAR_LOCK_FREE 和 ATOMIC_CHAR16_T_LOCK_FREE、ATOMIC_CHAR32_T_LOCK_FREE ATOMIC_WCHAR_T_LOCK_FREE, ATOMIC_SHORT_LOCK_FREE, ATOMIC_INT_LOCK_FREE, ATOMIC_LONG_LOCK_FREE, ATOMIC_LLONG_LOCK_FREE, ATOMIC_POINTER_LOCK_FREE。它们为指定的内置类型及其无符号相对物对应的原子类型指定了无锁状态(LLONG 指 long long，而 POINTER 指所有的指针类型)。如果原子类型从来都不是无锁的，则值为 0；如果原子类型始终无锁，则值为 2；如果相应的原子类型的无锁状态是前面描述的运行时属性，则值为 1。

唯一不提供 is_lock_free()成员函数的类型是 std::atomic_flag。该类型是一个简单的布尔标志，并且在这种类型上的操作必须是无锁的；一旦你有了一个简单的无锁布尔标志，你就可以使用它来实现一个简单的锁，并使用它作为基础来实现所有其他原子类型。当我说简单时，我的意思是：std::atomic_flag 类型的对象被初始化为清除状态，然后它们可以被查询和设置(使用 test_and_set()成员函数)或者被清除(使用 clear()成员函数)。

其余的原子类型都是通过 std::atomic<>类模板的特化来获得，它们的功能更全面一些，但可能不是无锁的(前面解释过)。在大多数流行的平台上，可以预期所有内置类型的原子变体(例如 std::atomic<int>和 std::atomic::<void*>)的确是无锁的，但这不是必需的。你很快就会看到，每个特化的接口反映了类型的属性；例如，像&=这样的按位操作没有为普通指针定义，所以它们也没有为原子指针定义。

除了直接使用 std::atomic<>类模板之外，还可以使用表 5.1 中所示的一组名称来引用实现提供的原子类型。由于原子类型添加到 C++标准的历史，如果你有一个老的编译器，这些替代类型名称可能引用对应 std::atomic<>的特化类或这个特化类的一个基类，而在一个完全支持 C++17 的编译器上，这些总是对应 std::atomic<>特化类的别名。因此，在同一个程序中混合使用这些代用名和直接命名的 std::atomic<>特化类可能会导致不可移植的代码。

表5.1 标准原子类型的代用名和它对应的std::atomic<>特化类

| 原子类型 | 相关特化类 |
| --- | --- |
| atomic_bool | std::atomic<bool> |
| atomic_char | std::atomic<char> |
| atomic_schar | std::atomic<signed char> |
| atomic_uchar | std::atomic<unsigned char> |
| atomic_int | std::atomic<int> |
| atomic_uint | std::atomic<unsigned> |
| atomic_short | std::atomic<short> |
| atomic_ushort | std::atomic<unsigned short> |
| atomic_long | std::atomic<long> |
| atomic_ulong | std::atomic<unsigned long> |
| atomic_llong | std::atomic<long long> |
| atomic_ullong | std::atomic<unsigned long long> |
| atomic_char16_t | std::atomic<char16_t> |
| atomic_char32_t | std::atomic<char32_t> |
| atomic_wchar_t | std::atomic<wchar_t> |

除了基本的原子类型之外，C++标准库还为原子类型提供了一组 typedef 对应各种非原子的标准库 typedef，如 std::size_t。如表 5.2 所示。

表5.2 标准原子的typedef和它们对应的内置类型typedef

| 原子类型定义 | 标准库中对应类型定义 |
| --- | --- |
| atomic_int_least8_t | int_least8_t |
| atomic_uint_least8_t | uint_least8_t |
| atomic_int_least16_t | int_least16_t |
| atomic_uint_least16_t | uint_least16_t |
| atomic_int_least32_t | int_least32_t |
| atomic_uint_least32_t | uint_least32_t |
| atomic_int_least64_t | int_least64_t |
| atomic_uint_least64_t | uint_least64_t |
| atomic_int_fast8_t | int_fast8_t |
| atomic_uint_fast8_t | uint_fast8_t |
| atomic_int_fast16_t | int_fast16_t |
| atomic_uint_fast16_t | uint_fast16_t |
| atomic_int_fast32_t | int_fast32_t |
| atomic_uint_fast32_t | uint_fast32_t |
| atomic_int_fast64_t | int_fast64_t |
| atomic_uint_fast64_t | uint_fast64_t |
| atomic_intptr_t | intptr_t |
| atomic_uintptr_t | uintptr_t |
| atomic_size_t | size_t |
| atomic_ptrdiff_t | ptrdiff_t |
| atomic_intmax_t | intmax_t |
| atomic_uintmax_t | uintmax_t |

这里有很多类型！有一个相当简单的模式：对于标准的 typedef T，对应的原子类型是相同的名字加上 atomic_前缀：atomic_T。这种情况也适用于内置类型，只是有符号的缩写为 s，无符号的缩写为 u，long long 的缩写为 llong。对于任何 T，比起使用代用名，通常 std::atomic<T>更明了。

标准原子类型在传统意义上是不可复制或赋值的，因为它们没有拷贝构造函数或拷贝赋值操作符。但是，它们确实支持从相应的内置类型赋值以及隐式转换成内置类型，以及直接的 load()和 store()成员函数、exchange()、compare_exchange_weak()和 compare_exchange_strong()。它们还在适当的地方支持复合赋值操作符：+=、-=、*=、|=等等，还有整型，以及用于支持++和—指针的 std::atomic<>特化版本。这些操作符还有相应的有相同功能的命名成员函数：fetch_add()、fetch_or()，等等。赋值操作符和成员函数的返回值要么是存储的值(对于赋值操作符)，要么是操作之前的值(对于命名的函数)。这就避免了潜在的问题，这个问题根源于赋值操作符通常习惯于返回被赋值对象的引用。为了从这些引用中获取存储的值，代码必须执行单独的读操作，从而允许另一个线程在赋值和读之间修改值，并打开竞争条件的大门。

但 std::atomic<>类模板不仅仅是一组特化。它确实有一个主模板，可用于创建用户自定义类型的原子变体。因为它是一个泛型类模板，所以操作被限制为 load()、store()(以及用户自定义类型的赋值和转换)、exchange()、compare_exchange_weak()和 compare_exchange_strong()。

每个原子类型上的操作都有一个可选的内存顺序参数，它是 std::memory_order 枚举类型的某个值，这个参数可以用来指定所需的内存顺序语义。std::memory_order 有六种可能的值：std::memory_order_relaxed，std::memory_order_acquire，std::memory_order_consume，std::memory_order_acq_rel，std::memory_order_release，std::memory_order_seq_cst。

允许使用的内存顺序值依赖于操作的类别，如果你不指定内存顺序，默认使用 std::memory_order_seq_cst，它是最强的内存顺序。5.3 节讨论了内存顺序选项的精确语义。目前，只要知道操作分为三类就足够了：

1. *存储（store）*操作，可以有 memory_order_relaxed，memory_order_release，或 memory_order_seq_cst 的顺序。
2. *加载（load）*操作，可以有 memory_order_relaxed，memory_order_consume，memory_order_acquire，或 memory_order_seq_cst 顺序。
3. *读-改-写（read-modify-write）*操作，可以有 memory_order_relaxed，memory_order_consume，memory_order_acquire，memory_order_release，memory_order_acq_rel，或 memory_order_seq_cst。

现在，让我们来看一下每个标准原子类型上可以执行的操作，就从 std::atomic_flag 开始。

## 5.2.2 std::atomic_flag 上的操作

std::atomic_flag 是最简单的标准原子类型，它表示一个布尔标志。此类型的对象可以处于两种状态之一：设置状态或清除状态。它被刻意设计为基础的，并且仅仅作为一个构建块使用。因此，除了在特殊情况下，我从不期望看到它被使用。尽管如此，它仍将作为讨论其他原子类型的起点，因为它展示了应用于原子类型的一些通用策略。。

std::atomic_flag 类型的对象必须用 ATOMIC_FLAG_INIT 初始化。这将标记初始化为 clear 状态。在这件事上没有选择；这个标志总是从清除状态开始：

```
std::atomic_flag f = ATOMIC_FLAG_INIT;
```

无论对象在哪里声明，它的作用域是什么，这都适用。它是唯一在初始化时需要这种特殊处理的原子类型，但它也是唯一保证无锁的类型。如果 std::atomic_flag 对象具有静态存储持续时间。它保证是静态初始化的，这意味着没有初始化顺序的问题；它总是在对标志进行第一次操作时被初始化。

一旦初始化标志对象后，只有三件事可以做：销毁它、清除它或设置它并查询之前的值。它们分别对应于析构函数、clear()成员函数和 test_and_set()成员函数。clear()和 test_and_set()成员函数两者都可以指定内存顺序。clear()是一个存储操作，因此不能有 memory_order_acquire 或 memory_order_acq_rel 语义，但是 test_and_set()是一个读-改-写操作，因此可以应用任何内存顺序标签。与每个原子操作一样，这两个操作的默认值都是 memory_order_seq_cst。例如：

```
f.clear(std::memory_order_release);  // 1
bool x=f.test_and_set();  // 2
```

调用 clear()①明确要求使用释放语义清除标志，而调用 test_and_set()②时使用默认内存顺序设置标记，并检索旧值。

你不能从第一个对象拷贝构造另一个 std::atomic_flag 对象，并且也不能从一个 std::atomic_flag 赋值给另一个。这不是对 std::atomic_flag 的特殊要求，而是所有原子类型通用的行为。原子类型上的所有操作都被定义为原子的，赋值和拷贝构造涉及两个对象。对两个不同对象的单个操作不可能是原子的。在拷贝构造或拷贝赋值的情况下，必须首先从一个对象读取值，然后将其写入另一个对象。这是两个独立对象上的两个独立操作，并且组合不可能是原子的。因此，不允许这些操作。

有限的特性使得 std::atomic_flag 非常适合于做自旋互斥锁。起初，标志被清除，并且互斥锁处于解锁状态。为了锁住互斥锁，循环运行 test_and_set()直到旧值为 false，这意味着这个线程已经把值设置为 true 了。解锁互斥锁是一件很简单的事情，将标记清除即可。实现如下面的程序清单所示：

```
清单 5.1 使用 std::atomic_flag 的自旋互斥锁实现
```

```
class spinlock_mutex
{
  std::atomic_flag flag;
public:
  spinlock_mutex():
    flag(ATOMIC_FLAG_INIT)
  {}
  void lock()
  {
    while(flag.test_and_set(std::memory_order_acquire));
  }
  void unlock()
  {
    flag.clear(std::memory_order_release);
  }
};
```

这个互斥锁很基础，但是足以用于 std::lock_guard<>(参见第 3 章)。由于它本质上是在 lock()中做了一个忙-等待，所以如果你预期有任何程度的竞争，它是一个糟糕的选择，但它足以确保互斥。当我们研究内存顺序语义时，你将看到和互斥锁一起使用的时候，这如何保证所必需的强制顺序。这样的例子将在 5.3.6 节中介绍。

std::atomic_flag 非常局限，以至于它甚至不能用作一般的布尔标志，因为它没有简单的非修改查询操作。为此，你最好使用 std::atomic<bool>，因此我接下来讨论它。

### 5.2.3 std::atomic<bool>上的操作

最基本的原子整型类型就是 std::atomic<bool>。如你所料，它有着比 std::atomic_flag 更加齐全的布尔标志特性。虽然依旧不能拷贝构造和拷贝赋值，但可以使用非原子的 bool 类型构造它，所以它可以被初始化为 true 或 false，并且可以从非原子 bool 变量赋值给 std::atomic<bool>：

```
std::atomic<bool> b(true);
b=false;
```

需要注意的是，非原子 bool 的赋值操作符不同于通常的惯例：返回被赋值对象的引用，取而代之，它返回要赋给的 bool 值。这是原子类型的另一个常见模式：它们的赋值操作符支持返回值(对应的非原子类型)，而不是引用。如果原子变量的引用被返回了，任何依赖于这个赋值结果的代码都需要显式加载这个值。潜在地可能获得另一个线程修改的结果。通过以非原子值的形式返回赋值结果，可以避免额外的加载操作，并且你得到的就是存储的值。

与使用 std::atomic_flag 受限的 clear()函数不同，写入(true 或者 false)是通过调用 store()来完成的，尽管仍然可以指定内存顺序语义。类似地，test_and_set()已被更

通用的 exchange()成员函数替换，该函数允许你将存储的值替换为你选择的新值，并自动检索原始值。std::atomic<bool>还支持通过隐式转换为简单的 bool 或通过显式调用 load()对值进行非修改查询。如你所料，store()是一个存储操作，而 load()是一个加载操作。exchange()是一个“读-改-写”操作：

```
std::atomic<bool> b;
bool x=b.load(std::memory_order_acquire);
b.store(true);
x=b.exchange(false, std::memory_order_acq_rel);
```

exchange()不是 std::atomic<bool>所支持的唯一的读-改-写操作；它还引入了一个操作，以便在当前值等于预期值时存储一个新值。

**根据当前值存储一个新值(或不存储)**

这个新操作称为比较-交换，它以成员函数 compare_exchange_weak()和 compare_exchange_strong()的形式出现。比较-交换操作是使用原子类型进行编程的基石；它将原子变量的值与提供的预期值进行比较，如果它们相等，则存储提供的期望值。如果值不相等，则预期值会用原子变量的值更新。比较-交换函数的返回类型是 bool 类型，如果执行了存储，则为 true，否则为 false。如果存储完成(因为值相等)，则操作成功，否则操作失败；返回值为 true 表示成功，返回值为 false 表示失败。

对于 compare_exchange_weak()函数，即使原始值与预期值一致时，存储也可能会不成功；在这种情况中变量的值不会发生改变，并且 compare_exchange_weak()的返回值是 false。这最可能发生在缺少“比较并交换”指令的机器上，如果处理器不能保证操作原子地完成——可能是因为执行操作的线程在必要的指令序列中间被切换出去，而操作系统在其位置调度了另一个线程，在这里线程数多于处理器的数量。这被称为*伪失败*(*spurious failure*)，因为失败的原因是时序的作用，而不是变量的值。

因为 compare_exchange_weak()可以伪失败，所以通常必须在一个循环中使用：

```
bool expected=false;
extern atomic<bool> b; // 在其他某个地方设置
while(!b.compare_exchange_weak(expected,true) && !expected);
```

在本例中，只要 expected 仍然为 false，就会继续循环，这表明 compare_exchange_weak()调用伪失败了。

另一方面，只有当值不等于预期值时，compare_exchange_strong()保证返回 false。这可以在像刚才展示的地方消除对循环的需求，在这些地方你想知道是否成功地更改了一个变量，或者另一个线程是否先到达了那里。

如果你想修改变量而不管初始值是什么(可能使用一个依赖于当前值的更新值)，则 expected 的更新将变得很有用。在每次循环中，expected 都会被重新加载，因此如果在此期间没有其他线程修改该值，则 compare_exchange_weak()或 compare_exchange_strong()

调用在下一次循环中应该成功。如果要存储的值计算很简单，那么使用 compare_exchange_weak 可能是有益的,它避免在 compare_exchange_weak()可能伪失败的平台上使用双循环(因此 compare_exchange_strong()包含一个循环)。另一方面，如果要存储的值计算非常耗时，那么使用 compare_exchange_strong()可以避免在 expected 值没有改变时重新计算要存储的值。对于 std::atomic<bool>，这并不是很重要—毕竟只有两个可能的值—但是对于较大的原子类型，这可能会不一样。

比较交换函数的另一个不同寻常之处在于，它们可以使用两个内存顺序参数。这允许在成功和失败的情况下，有不同的内存顺序语义；对于一个成功的调用拥有 memory_order_acq_rel 语义，而失败的调用具有 memory_order_relaxed 语义是可取的。失败的比较交换不会做存储，因此它不能有 memory_order_release 或 memory_order_acq_rel 语义。因此，不允许为失败提供这些值作为顺序。你也不能为失败提供比成功更严格的内存顺序；如果你希望将 memory_order_acquire 或 memory_order_seq_cst 语义用于失败，那么也必须为成功指定这些语义。

如果你没有指定失败的顺序，则假定它与成功的顺序相同，只是顺序的“释放”（release）部分被剥离：memory_order_release 变成 memory_order_relax, memory_order_acq_rel 变成 memory_order_acquire。如果两者都没有指定，则默认为 memory_order_seq_cst，这将提供成功和失败的全序列顺序。下面两个对 compare_exchange_weak()的调用是等效的：

```
std::atomic<bool> b;
bool expected;
b.compare_exchange_weak(expected,true,
  memory_order_acq_rel,memory_order_acquire);
b.compare_exchange_weak(expected,true,memory_order_acq_rel);
```

我将把选择内存顺序的结果留给 5.3 节讨论。

std::atomic<bool>和 std::atomic_flag 之间一个进一步的区别是 std::atomic<bool>可能不是无锁的；实现可能必须在内部获得互斥锁，以确保操作的原子性。在这种情况下，可以使用 is_lock_free()成员函数检查 std::atomic<bool>上的操作是否无锁。这是除了 std::atomic_flag 外所有原子类型的另一个常见特性。

下一个最简单的原子类型是原子指针特化 std::atomic<T*>，所以我们接下来将讨论它们。

### 5.2.4 std::atomic<T*>上的操作： 指针运算

某种类型 T 的指针的原子形式是 std::atomic<T*>，就像 bool 的原子形式是 std::atomic<bool>。接口是相同的，尽管它操作的是对应指针类型的值，而不是 bool 值。与 std::atomic<bool>一样，它既不是可拷贝构造的，也不是可拷贝赋值的，尽管它可以通过适当的指针值构造和赋值。除了必须的 is_lock_free()成员函数外，std::atomic<T*>还有 load()，store()，exchange()，compare_exchange_weak()和 compare_exchange_strong()成员函数，其语义与 std::atomic<bool>成员函数相似，同样获取和返回 T*而不是 bool。

提供给 std::atomic<T*>的新操作为指针运算操作。基本操作由 fetch_add()和 fetch_sub()提供，它们在存储的地址上做原子加法和减法，以及+=，-=，前置后置的++和--提供简易的封装。操作符的工作方式与你在内置类型中预期的一样：如果 x 是指向 Foo 对象数组的第一个条目的 std::atomic<Foo*>，那么 x+=3 将其更改为指向第四个条目，并返回一个普通的 Foo*，该 Foo*也指向第四个条目。fetch_add()和 fetch_sub()稍微有点不同，它们返回初始值（所以 x.ftech_add(3)将更新 x 指向第四个元素，但返回指向数组第一个元素的地址）。这种操作也被称为*交换并相加*（exchange-and-add），并且这是一个原子的读-改-写操作，像 exchange()和 compare_exchange_weak()/compare_exchange_strong()一样。与其他操作一样，返回值是一个普通的 T*值，而不是对 std::atomic<T*>对象的引用，这样调用代码就可以根据之前的值执行操作：

```
class Foo{};
Foo some_array[5];
std::atomic<Foo*> p(some_array);
Foo* x=p.fetch_add(2);  // p 加 2，并返回老的值
assert(x==some_array);
assert(p.load()==&some_array[2]);
x=(p-=1);  // p 减 1，并返回新的值
assert(x==&some_array[1]);
assert(p.load()==&some_array[1]);
```

函数形式也允许内存顺序语义被指定为一个额外的函数调用参数：

```
p.fetch_add(3,std::memory_order_release);
```

因为 fetch_add()和 fetch_sub()都是读-改-写操作，可以拥有任意的内存顺序标签，并且可以加入到一个*释放序列*（release sequence）中。我们没法为操作符形式指定顺序语义，因为没办法提供必要的信息：因此这些形式都具有 memory_order_seq_cst 语义。

其余的基本原子类型都是相同的：它们都是原子整型，彼此具有相同的接口，只是相关的内置类型不同。我们将把他们看成一个整体。

## 5.2.5 标准原子整型类型上的操作

和通常的操作集合一样(load()，store()，exchange()，compare_exchange_weak()，和 compare_exchange_strong())，原子整形类型如 std::atomic<int>和 std::atomic<unsigned long long>有相当全面的一整套可用的操作：fetch_add()，fetch_sub()，fetch_and()，fetch_or()，fetch_xor()，还有复合赋值形式的操作((+=，-=，&=，|=和^=)，以及++和--(++x，x++，--x 和 x--)。它并不是你可以在普通整数类型上执行的完整的复合赋值操作集，但它已经足够接近了：只有除法、乘法和移位操作符缺失了。因为原子整数值通常用作计数器或位掩码，所以这不是一个特别明显的损失；如果需要的话，可以在循环中使用 compare_exchange_weak()轻松完成其他操作。

std::atomic<T>（译注：原文是 std::atomic<T*>，应该是笔误，这里上下文都说的整型类型）的语义与 fetch_add()和 fetch_sub()非常匹配；命名函数自动执行其操作并返回旧值，而复合赋值操作符返回新值。前置和后置的增减操作与往常一样：++x 递增变量并返回新值，而 x++递增变量并返回旧值。正如你所期望的，在这两种情况下，结果都是相关整形的值。

我们已经看过所有基本原子类型；剩下的就只有泛型的 std::atomic<>主类模板，而非其特化。那么接下来让我们来看一下这个。

## 5.2.6 std::atomic<>主类模板

主模板的存在允许用户创建用户自定义类型的原子变体，而不只限于标准原子类型。给定用户自定义的类型 UDT，std::atomic<UDT>提供了与 std::atomic<bool>相同的接口(如 5.2.3 节所述)，除了将 bool 参数和与存储值相关的返回类型(而不是比较交换操作的成功/失败结果)换成 UDT。你不能将任意的用户自定义类型和 std::atomic<>一起使用；类型必须满足特定的条件。为了对某些用户自定义的类型 UDT 使用 std::atomic<UDT>，该类型必须有一个平凡的（trivial）拷贝赋值操作符。这意味着该类型必须没有任何虚函数或虚基类，并且必须使用编译器生成的拷贝赋值操作符。不仅如此，用户自定义类型的每个基类和非静态数据成员还必须具有平凡的拷贝赋值操作符。这允许编译器使用 memcpy()或一个等价的操作用于赋值操作，因为不需要运行用户编写的代码。

最后，值得注意的是，比较交换操作像使用 memcmp 一样进行按位比较，而不是使用任何可能为 UDT 定义的比较运算符。如果该类型提供具有不同语义的比较操作，或者该类型具有不参与普通比较的填充位，那么这可能导致比较-交换操作失败，哪怕比较的值是相等的。

这些限制背后的原因可以追溯到第 3 章中的一条指南：不要将指针和引用作为参数传递给用户提供的函数，从而在锁范围之外传递受保护数据。一般来说，编译器不能为 std::atomic<UDT>生成无锁代码，因此它必须为所有操作使用一个内部锁。如果允许用户提供的拷贝赋值或比较操作符，则需要将对受保护数据的引用作为参数传递给用户提供的函数，这违反了指南。此外，库完全可以自由地对所有需要它的原子操作使用单个锁，允许在持有该锁的同时调用用户提供的函数可能会导致死锁或因为比较操作花费了很长时间从而导致其他线程阻塞。最后，这些限制增加了编译器直接为 std::atomic<UDT>使用原子指令的机会(并使特定的实例化无锁)，因为它可以将用户自定义的类型视为一组原始字节。

注意，尽管你可以使用 std::atomic<float>或 std::atomic<double>，因为内置的浮点类型确实满足与 memcpy 和 memcmp 一起使用的条件，但在 compare_exchange_strong 的情况下，行为可能会令人吃惊(如前所述，compare_exchange_weak 总是会由于任意的内部原因而失败)。如果存储的值具有不同的表示形式，即使旧的存储值与比较数等价，操作也可能失败。注意，没有浮点值上的原子算术操作。如果你使用 std::atomic<>来定义用户自定义的类型，该类型定义了一个相等比较操作符，并且该操作符与使用 memcmp 的比较不同，那么你将在 compare_exchange_strong 中获得类似的行为——操作可能会失败，因为原本相等的值有不同的表示。

如果你的 UDT 与 int 或 void*大小相同(或小于)，那么大多数常见平台都能够对 std::atomic<UDT>使用原子指令。一些平台还能对两倍于 int 或 void*大小的用户自定义类型使用原子指令。这些平台通常支持所谓的*双字比较和交换*(DWCAS，double-word-compare-and-swap)指令，该指令对应于 compare_exchange_xxx 函数。正如你将在第 7 章中看到的，这种支持在编写无锁代码时非常有用。

这些限制意味着你不能，例如，创建 std::atomic<std::vector<int>>（因为它有一个特殊的拷贝构造和拷贝赋值操作符）。但是你可以用包含计数器、标志、指针甚至简单数据元素数组的类实例化 std::atomic<>。这并不是什么大问题；数据结构越复杂，你就越可能希望对其进行操作，而不只是简单的赋值和比较。如果是这种情况，你最好使用 std::mutex，以确保为所需的操作适当地保护数据，如第 3 章所述。

如前所述，当用用户自定义的类型 T 实例化时，std::atomic<T>的接口被限制为 std::atomic<bool>可用的操作集合：load()、store()、exchange()、compare_exchange_weak()、compare_exchange_strong()以及类型 T 实例的赋值和转换。

表 5.3：原子类型上的可用操作

| Operation | atomic_flag | atomic <bool> | atomic <T*> | atomic <integral-type> | atomic <other-type> |
|---|---|---|---|---|---|
| test_and_set | Y | | | | |
| clear | Y | | | | |
| is_lock_free | | Y | Y | Y | Y |
| load | | Y | Y | Y | Y |
| store | | Y | Y | Y | Y |
| exchange | | Y | Y | Y | Y |
| compare_exchange_weak, compare_exchange_strong | | Y | Y | Y | Y |
| fetch_add, += | | | Y | Y | |
| fetch_sub, -= | | | Y | Y | |
| fetch_or, \|= | | | | Y | |
| fetch_and, &= | | | | Y | |
| fetch_xor, ^= | | | | Y | |
| ++, -- | | | Y | Y | |

## 5.2.7 用于原子操作的自由函数

到目前为止，我仅限于描述原子类型上成员函数形式的操作。但是对于各种原子类型的所有操作，也有等价的非成员函数。在大多数情况下，非成员函数以对应的成员函数命

名，但是使用 atomic_前缀(例如，std::atomic_load())。这些函数都会被不同的原子类型所重载。这里有机会指定一个内存顺序标签，他们有两个变种：一个没有标签，一个有_explicit 后缀并且有一个或多个额外的参数，用于表示一个或多个内存顺序的标签(例如，std::atomic_store(&atomic_var,new_value)与 std::atomic_store_explicit(&atomic_var,new_value,std::memory_order_release)。虽然成员函数引用的原子对象是隐式的，但所有自由函数都将原子对象的指针作为第一个参数。

例如，std::atomic_is_lock_free()只有一种形式(尽管每种类型都重载了)，对原子类型的一个对象 a，std::atomic_is_lock_free(&a)和 a.is_lock_free()返回相同的值。同样 std::atomic_load(&a)和 a.load()是一样的，但 a.load(std::memory_order_acquire)相对应的是 std::atomic_load_explicit(&a, std::memory_order_acquire)。

自由函数被设计为与 C 兼容的，因此它们在所有情况下都使用指针而不是引用。例如，compare_exchange_weak()和 compare_exchange_strong()成员函数的第一个参数(预期值)是一个引用，而 std::atomic_compare_exchange_weak()(第一个是对象指针)的第二个参数是一个指针。std::atomic_compare_exchange_weak_explicit()也要求指定成功和失败的内存顺序，而比较交换成员函数有单一的内存顺序形式(默认为 std::memory_order_seq_cst)和一个重载，分别接受成功和失败的内存顺序。

std::atomic_flag 上的操作是“反潮流”的，它们的名字中拼出了 flag：std::atomic_flag_test_and_set()和 std::atomic_flag_clear()。指定内存顺序的额外变种还是有_explicit 后缀：std::atomic_flag_test_and_set_explicit()和 std::atomic_flag_clear_explicit()。

C++标准库也提供了自由函数，用于以原子方式访问 std::shared ptr<>的实例。这打破了原则：只有原子类型支持原子操作，因为 std::shared_ptr<>很肯定不是一个原子类型(从多个线程访问同一个 std::shared_ptr<T>对象，而不使用来自所有线程的原子访问函数，或使用适当的其他外部同步，是一种数据竞争和未定义的行为)。但是 C++标准委员会认为提供这些额外的功能是非常重要的。可用的原子操作有 load、store、exchange 和 compareexchange，它们都是标准原子类型上相同操作的重载，以 std::shared_ptr<>*作为第一个参数：

```
std::shared_ptr<my_data> p;
void process_global_data()
{
  std::shared_ptr<my_data> local=std::atomic_load(&p);
  process_data(local);
}

void update_global_data()
{
  std::shared_ptr<my_data> local(new my_data);
  std::atomic_store(&p,local);
```

```
}
```

与其他类型的原子操作一样，也提供了_explicit 变体来允许你指定所需的内存顺序，并且可以使用 std::atomic_is_lock_free()函数来检查实现是否使用锁来确保原子性。

并行技术规范也提供了一种原子类型 std::experimental::atomic_shared_ptr<T>。要使用它，你必须包含<experimental/atomic>头文件。它提供了和 std::atomic<UDT>一样的操作集合：load，store，exchange，compare-exchange。它是作为一个单独的类型提供的，因为它允许无锁实现，不会对简单的 std::shared_ptr 实例强加额外的成本。但是与 std::atomic 模板一样，你仍然需要检查它在你的平台上是否是无锁的，这可以用 is_lock_ free 成员函数进行测试。即使它不是无锁结构，也推荐使用 std::experimental::atomic_shared_ptr，而不要在普通的 std::shared_ptr 上使用原子自由函数，因为该类型会让代码更加清晰，确保所有的访问都是原子的，并且能避免由于忘记使用原子自由函数，从而导致数据竞争。与原子类型和操作的所有使用一样，如果使用它们是为了提高速度，那么对它们进行分析并与使用其他同步机制进行比较是很重要的。

如简介中所述，标准原子类型不仅避免了与数据竞争相关的未定义行为；它们允许用户强制线程之间的操作顺序。这种强制的顺序是保护数据和同步操作，比如 std::mutex 和 std::future<>的基础。记住这一点，让我们进入本章的实质内容：内存模型并发方面的细节，以及如何使用原子操作来同步数据和强制顺序。

## 5.3 同步操作和强制顺序

假设有两个线程，其中一个线程正在填充数据结构，以便第二个线程读取。为了避免出现有问题的竞争条件，第一个线程设置了一个标志来表示数据已经准备好了，而第二个线程直到设置了该标志才读取数据。

清单 5.2 从不同线程读写数据

```
#include <vector>
#include <atomic>
#include <iostream>

std::vector<int> data;
std::atomic<bool> data_ready(false);

void reader_thread()
{
  while(!data_ready.load())  // 1
  {
    std::this_thread::sleep(std::milliseconds(1));
  }
  std::cout<<"The answer="<<data[0]<<"\n";  // 2
```

```
}

void writer_thread()
{
  data.push_back(42);  // 3
  data_ready=true;  // 4
}
```

先不考虑循环等待数据准备的低效率①，你需要这么做，否则在线程之间共享数据将变得不切实际：每个数据项都必须是原子的。你已经知道，在没有强制顺序的情况下让非原子读②和写③访问相同的数据是未定义的行为，因此要使其工作，必须在某些地方强制顺序。

必需的强制顺序来自 std::atomic<bool>变量 data_ready 上的操作，它们通过内存模型关系“先发生于”（happens-before）和“同步于”（synchronizes-with）提供了必要的顺序。对数据的写操作③发生在对 data_ready 标记的写操作④之前，对标志的读操作①发生在对数据的读操作②之前。当从 data_ready 中读取的值①为真时，写操作与读操作同步，创建了一个“先发生于”关系。因为“先发生于”是可传递的，写入数据③发生在写入标志前④，写入标志又发生在从标志中读出 true 值之前①，读出 true 值又发生在读取数据之前②，然后你有一个强制顺序：数据的写入发生在数据的读取之前，一切正常。图 5.2 显示了这两个线程中重要的“先发生于”关系。我已经从读取线程中添加了几个 while 循环的迭代。

所有这些看起来都很直观：写入值的操作发生在读取值的操作之前。对于默认的原子操作，确实是这样(这就是为什么它是默认的)，但确实需要说清楚：原子操作对于顺序需求还有其他选项，我很快就会讲到。

既然你已经在实战中看到了“先发生于”和“同步于”，现在是时候看看它们意味着什么了。我将从“同步于”开始。

## 5.3.1 “同步于”关系

“同步于”关系只能在原子类型的操作之间获得。如果数据结构包含原子类型，并且数据结构上的操作在内部执行适当的原子操作，那么数据结构上的操作(比如锁住互斥锁)可能提供这种关系，但从根本上说，它只来自于原子类型上的操作。

基本思想是这样的：在变量 x 上，一个适当标记的原子写操作 W 同步于 x 上适当标记的原子读操作，读取的值或者是 W 操作写入的内容，或是和 W 操作同一个线程随后的一个写操作对 x 写入的值；亦或是任意线程对 x 的一系列原子读-改-写操作(例如，fetch_add()或 compare_exchange_weak())写入的值，这里，序列中第一个线程读取的值是 W 写入的值(参见 5.3.4 节)。

先把“适当标记”（suitably-tagged）放一边，因为所有对原子类型的操作，默认都是“适当标记”的。这实际上就是：如果线程 A 存储了一个值，并且线程 B 读取了这个

值，线程 A 的存储操作与线程 B 的加载操作之间有一个“同步于”关系，如同清单 5.2 所示的那样。在图 5.2 中做了说明。

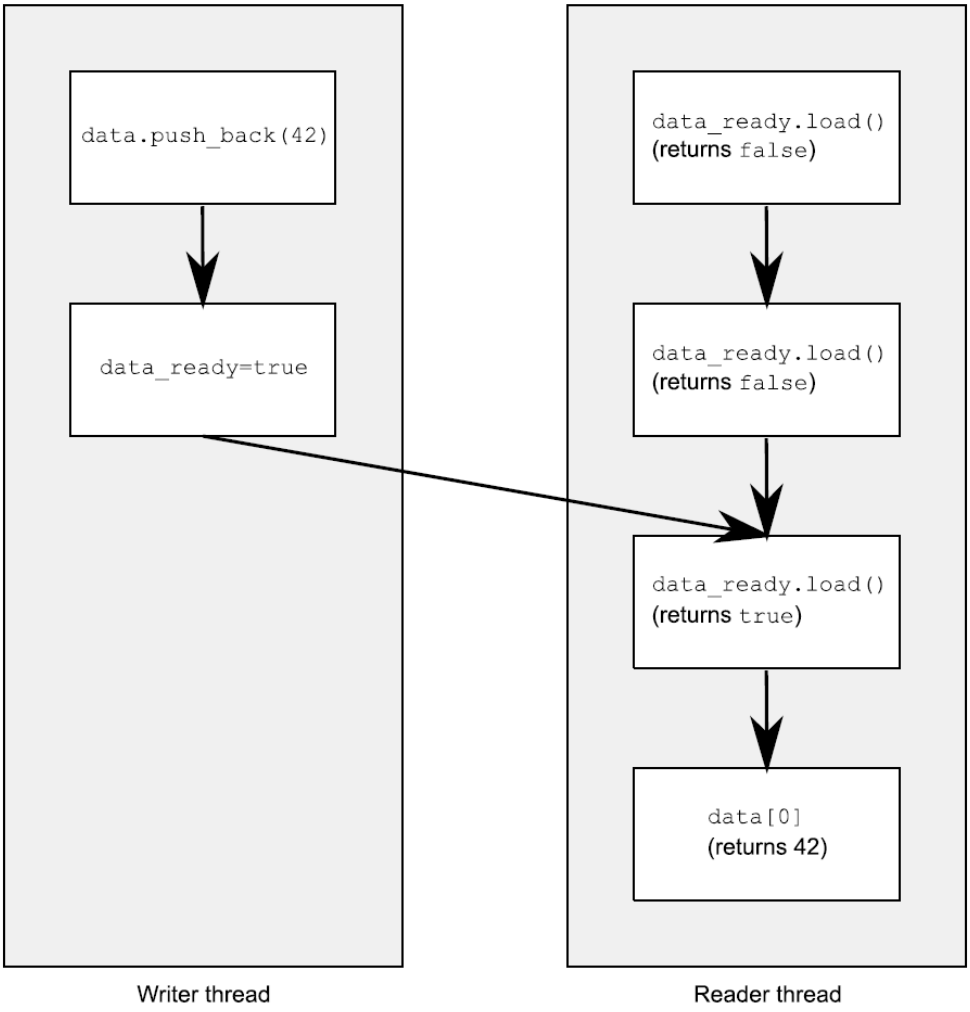


图 5.2 使用原子操作在非原子操作间强制一个顺序

我相信你已经猜到了，细微的差别都在“适当标记”部分。C++内存模型允许对原子类型的操作应用各种顺序约束，这就是我所提到的标记。在 5.3.3 节中介绍了内存顺序的各种选项，以及它们如何与“同步于”关系相关联。首先，让我们回过头来看看“先发生于”关系。

## 5.3.2 “先发生于”关系

“先发生于”和“强先发生于”（strongly-happens-before）关系是一个程序中操作顺序的基础构件块：它指定了哪些操作能看见另一些操作的影响。对于单线程来说，这很大程度上是简单的：当一个操作排在另一个前面，那么它在另一个之前发生，并且“强先发生于”另一个。这意味着，在源代码中，如果一个操作(A)先于另一个操作(B)出现在语句中，那么 A 在 B 之前发生，并且 A“强先发生于”B。在清单 5.2 中可以看到：对 data

的写入操作③在对 data_ready 的写入操作④前发生。如果操作发生在同一个语句，通常情况下它们之间就没有“先发生于”关系了，因为它们是无序的。这是表示顺序未指定的另一种方式。你知道下面的程序会输出“1，2”或“2，1”，但是没有指定是哪一个，因为对 get_num()的两次调用没有指定顺序。

清单 5.3 参数中的求值顺序对函数调用是未指定的

```
#include <iostream>

void foo(int a,int b)
{
  std::cout<<a<<",”<<b<<std::endl;
}

int get_num()
{
  static int i=0;
  return ++i;
}

int main()
{
  foo(get_num(),get_num());  // 无序调用 get_num()
}
```

在某些情况下，单个语句中的操作是有序的，比如使用内置的逗号操作符，或者将一个表达式的结果用作另一个表达式的参数。但通常，单个语句中的操作是无顺序的，它们之间没有“先序于”（sequenced-before）（译注：sequenced-before 特指同一个线程内的求值顺序，一个求值先于另一个求值）(因此也没有“先发生于”)关系。一个语句中的所有操作都发生在下一个语句中的所有操作之前。

这是对你所熟悉的单线程排序规则的重述，那有什么新东西吗？新的内容是线程间的交互：如果在一个线程上的操作 A“线程间先发生于”（inter-thread happens before）另一个线程上的操作 B，那么 A 在 B 之前发生。这并没有多大帮助：你已经添加了一个新的关系(“线程间先发生于”，inter-thread happens-before)，但当你编写多线程的代码时，这是一个重要的关系。

在基本层面上，“线程间先发生于”（inter-thread happens-before）比较简单，并且依赖于 5.3.1 节介绍的“同步于”关系(详见 5.3.1 节)：如果操作 A 在一个线程上，与另一个线程上的操作 B 同步，那么 A“线程间先发生于”B。它同样是一个传递关系：如果 A“线程间先发生于” B，并且 B “线程间先发生于” C，那么 A 就“线程间先发生于”C。在清单 5.2 你也看到了这个。

“线程间先发生于”也可以和“先序于”相结合：如果操作 A “先序于”操作 B，并且操作 B“线程间先发生于”操作 C，那么 A“线程间先发生于”C。类似地，如果 A“同步于”B，并且 B “先序于”C，那么 A “线程间先发生于”C。两者的结合意味着，当你对

数据在一个线程内进行一系列修改时，为了让数据被执行 C 的线程上的后续操作可见，只需要一个“同步于”关系。

“强先发生于”关系会有一些不同，不过在大多数情况下是一样的。上面的两个规则同样适用于“强先发生于”：如果操作 A“同步于”操作 B，或操作 A“先序于”操作 B，那么 A 就是“强先发生于”于 B。也可以应用顺序传递：如果 A“强先发生于”B，并且 B“强先发生于”C，那么 A“强先发生于”C。不同在于，标记为 memory_order_consume（参见 5.3.3 节）的操作参与到"线程间先发生于"关系（因而也是“先发生于”关系），但不参与“强先发生于”关系。由于大多数代码并不适用 memory_order_consume，因此这种区别在实际中可能不会有什么影响。为了简洁起见，我将在本书剩下部分使用“先发生于”。

这些是线程间强制操作顺序至关重要的规则，并使清单 5.2 中的所有内容正常工作。稍后你将看到，数据依赖还有一些额外的细微差别。为了让你理解这一点，我需要介绍用于原子操作的内存顺序标记，以及它们如何与“同步于”关系相关联。

## 5.3.3 原子操作的内存顺序

有六个内存顺序选项可以应用于原子类型上的操作：memory_order_relaxed, memory_order_consume, memory_order_acquire, memory_order_release, memory_order_acq_rel, 以及 memory_order_seq_cst。除非你为特定的操作指定一个其他的选项，否则内存顺序选项对所有原子类型都是 memory_order_seq_cst，这是可用选项中最严格的。虽然有六个顺序选项，但是它们仅代表三种模型：*序列一致*（*sequentially consistent*）顺序，*获得-释放*（acquire-release）顺序(*memory_order_consume, memory_order_acquire, memory_order_release 和 memory_order_acq_rel*)和*宽松*（relaxed）顺序(*memory_order_relaxed*)。

不同的内存顺序模型在不同的 CPU 架构下，成本是不一样的。例如，在基于通过处理器精细控制操作可见性的架构（而不是做出更改的架构）的系统上，“序列一致”顺序相较于“获得-释放”顺序或者“宽松”顺序，以及“获得-释放”顺序相较于“宽松”顺序，都需要额外的同步指令。如果这些系统有很多处理器，这些额外的同步指令可能会花费大量的时间，从而降低系统整体的性能。另一方面，使用 x86 或 x86-64 架构的 CPU(例如，使用 Intel 或 AMD 处理器的台式电脑）除了确保原子性所必需的之外，不需要为“获得-释放”顺序添加额外的指令，甚至“序列一致”顺序对于加载操作也不需要任何特殊的处理，尽管存储操作需要很小的额外成本。

不同内存顺序模型的有效性，允许专家利用更细粒度的顺序关系来提升性能，用于它们有优势的地方，同时允许使用默认的“序列一致”顺序（相较于其他顺序，它比较简单的)用于非关键的情况。

为了选择使用哪个顺序模型或为了了解代码中的顺序关系，知道不同模型是如何影响程序的行为是很重要的。因此，让我们看看每种操作顺序和同步选择的影响。

**“序列一致”顺序**

默认的顺序名为*序列一致*（sequentially consistent），意味着程序的行为与现实世界的简单顺序视图是一致的。如果原子类型实例上的所有操作都是“序列一致”的，那么多线程程序的行为就好像所有这些操作都是由单个线程按照某种特定的序列执行的。这是迄今为止最容易理解的内存顺序，这就是为什么它是默认的：所有线程必须看到相同的操作顺序。这使得思考用原子变量编写的代码的行为变得很容易。你可以写下由不同线程执行的所有可能的操作序列，消除那些不一致的操作，并验证你的代码在其他线程中的行为是否符合预期。这也意味着操作不能被重新排序；如果你的代码在一个线程中有一个操作先于另一个操作，那么该顺序必须被所有其他线程看到。

从同步的角度来看，一个“序列一致”的存储操作“同步于”相同变量的加载操作，这个加载操作读取存储的值。这为两个(或更多)线程的操作提供了一个顺序约束，但“序列一致”比这更强大。对系统中使用“序列一致”原子操作的其他线程，任何那个加载操作之后执行的“序列一致”的原子操作也必须出现在存储操作之后。清单 5.4 中的示例演示了这种顺序约束的作用。这种约束不适用于使用“宽松”内存顺序原子操作的线程；它们仍然可以看到不同顺序的操作，因此必须在所有线程上使用“序列一致”的操作，才能受益。

不过，易于理解是要付出代价的。在有许多处理器的弱顺序机器上，可能会造成明显的性能损耗，因为必须在处理器之间保持整个操作序列的一致性，可能需要在处理器之间进行大量(而且昂贵!)的同步操作。即便如此，一些处理器架构(比如通用的 x86 和 x86-64 架构)就提供了成本相对较低的“序列一致”，如果你关心使用“序列一致”对性能的影响，就需要去查阅你目标处理器的架构文档。

以下清单展示了“序列一致”，对于 x 和 y 的加载和存储都显示标注为 memory_order_seq_cst，尽管这个例子中，标签可以省略，因为它是默认项。

清单 5.4 序列一致意味着全序

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x()
{
  x.store(true,std::memory_order_seq_cst);  // 1
}

void write_y()
{
  y.store(true,std::memory_order_seq_cst);  // 2
}
```

```
void read_x_then_y()
{
  while(!x.load(std::memory_order_seq_cst));
  if(y.load(std::memory_order_seq_cst))  // 3
    ++z;
}

void read_y_then_x()
{
  while(!y.load(std::memory_order_seq_cst));
  if(x.load(std::memory_order_seq_cst))  // 4
    ++z;
}

int main()
{
  x=false;
  y=false;
  z=0;
  std::thread a(write_x);
  std::thread b(write_y);
  std::thread c(read_x_then_y);
  std::thread d(read_y_then_x);
  a.join();
  b.join();
  c.join();
  d.join();
  assert(z.load()!=0);  // 5
}
```

assert⑤语句是永远不会触发的，因为不是存储 x 的操作①发生，就是存储 y 的操作②发生，虽然没有指定哪个。如果在 read_x_then_y 中加载 y③返回 false，是因为存储 x 的操作肯定发生在存储 y 的操作之前，在这种情况下在 read_y_then_x 中加载 x④必定会返回 true，因为 while 循环能保证在某一时刻 y 是 true。因为 memory_order_seq_cst 的语义需要在所有标记为 memory_order_seq_cst 的操作上有一个单一全序，所以在“加载 y 返回 false③”与“存储 y②”的操作之间存在一个隐含的顺序关系。因为是单个全序，当一个线程看到 x==true，随后又看到 y==false，这就意味着在这个全序中存储 x 的操作发生在存储 y 的操作之前。

因为一切都是对称的，所以也可以反过来，加载 x④的操作返回 false，会强制加载 y③的操作返回 true。这两种情况下，z 都等于 1。当两个加载操作都返回 true，z 就等于 2；所以任何情况下，z 都不可能是 0。

针对 read_x_then_y 看见 x 为 true，并且 y 为 false 的情况，相关的操作和“先发生于”关系如图 5.3 所示。从 read_x_then_y 中对 y 的加载操作开始，到达 write_y 中对 y

的存储的虚线展示了为了保持“序列一致”而隐含的必须的顺序关系：在memory_order_seq_cst 操作的全局顺序中，为了获得这里给出的结果，加载操作必须在存储操作之前发生。

“序列一致”是最简单、直观的顺序，但也是最昂贵的内存顺序，因为它需要所有线程之间进行全局同步。在一个多处理器系统上，这可能需要在处理器之间进行大量并且耗时的通信。

为了避免这种同步成本，你需要走出“序列一致”的世界，并且考虑使用其他内存顺序。

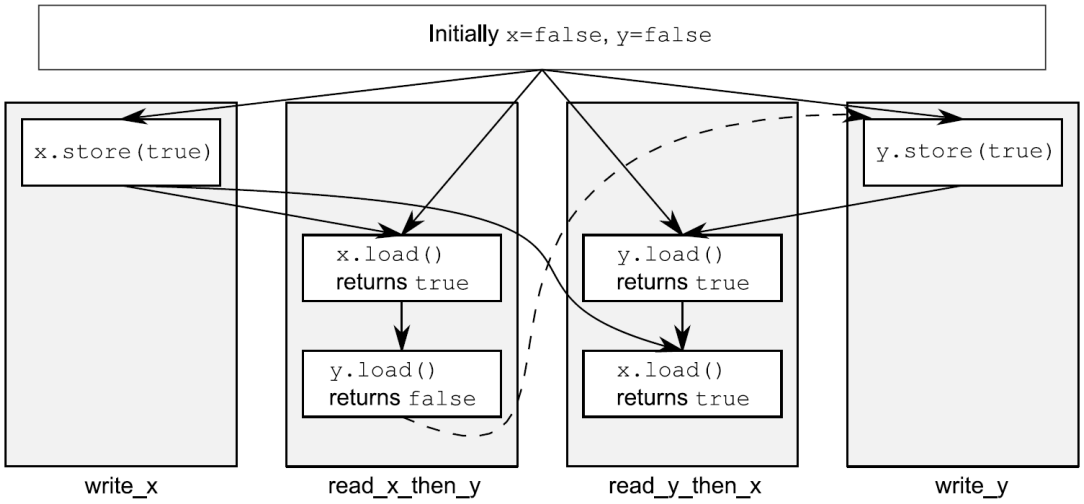


图 5.3 “序列一致”与“先发生于”

**非序列一致内存顺序**

一旦走出“序列一致”的世界，事情就开始变的复杂。需要解决的最大问题可能是这样一个事实：再也没有事件的单一的全局顺序。这意味着对于相同的操作，不同的线程可以看到不同的视图，对于来自不同线程的操作，一个接一个整齐交错的思维模型必须丢弃。你不仅必须考虑真正并发发生的事情，而且线程也不必对事件的顺序达成一致。为了编写(甚至理解)任何使用除了默认的 memory_order_seq_cst 之外的内存顺序的代码，弄清楚这一点绝对重要。不仅编译器能重排指令。即使线程运行相同位的代码，由于其他线程在没有显式顺序约束的情况下进行操作，那么它们也可能在事件的顺序上存在分歧，这是因为不同的 CPU 缓存和内部缓冲区可以为相同的内存保存不同的值。它是如此的重要，我再重申一遍：线程没有必要对事件的顺序达成一致。

你不仅要抛弃基于交错操作的思维模型，还必须抛弃基于编译器或处理器重排指令的思维模型。在没有其他顺序约束的情况下，唯一的要求是所有线程对每个独立变量的修改顺序达成一致。在不同的线程上，对不同变量的操作可以以不同的顺序呈现，只要所看到的值与任何附加的顺序约束一致。

最好的展示是跳出“序列一致”的世界，并对所有操作使用 memory_order_relaxed。一旦你掌握了这一点，你可以回到获得-释放顺序，这允许你有选择地引入操作之间的顺序关系，并恢复部分理智。

**“宽松”顺序**

原子类型上的操作以宽松顺序执行时，不会参与任何“同步于”关系。同一线程中对于同一变量的操作服从“先发生于”的关系，但是相对其他线程几乎对顺序没有任何要求。唯一的要求是从同一个线程访问单个的原子变量不能重排，一旦给定线程看到一个原子变量的特定值时，那个线程随后的读操作就不可能检索到这个变量更早的值。如果没有任何额外的同步，每个变量的修改顺序是使用 memory_order_relaxed 的线程间唯一共享的东西。

为了演示宽松操作有多宽松，你只需要两个线程，如下面的清单所示。

清单 5.5 宽松操作只有很少的顺序要求

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y()
{
  x.store(true,std::memory_order_relaxed);  // 1
  y.store(true,std::memory_order_relaxed);  // 2
}
void read_y_then_x()
{
  while(!y.load(std::memory_order_relaxed));  // 3
  if(x.load(std::memory_order_relaxed))  // 4
    ++z;
}
int main()
{
  x=false;
  y=false;
  z=0;
  std::thread a(write_x_then_y);
  std::thread b(read_y_then_x);
  a.join();
  b.join();
  assert(z.load()!=0);  // 5
```

```
}
```

这次 assert⑤可能会触发，因为加载 x 的操作④可能读取到 false，即使加载 y 的操作③读取到 true，并且存储 x 的操作①“先发生于”存储 y 的操作②。x 和 y 是两个不同的变量，因此，对于每个变量上的操作所产生值的可见性，没有顺序保证。

宽松操作对于不同变量可以自由重排，只要它们遵守约束它们的任何“先发生于”关系即可(比如，在同一线程中)，它们不会引入“同步于”关系。清单 5.5 中的“先发生于”关系如图 5.4 所示，连同一个可能的结果。尽管，在存储与存储，加载与加载操作间有着“先发生于”关系，但存储和加载之间并不存在这种关系，所以载入操作可以看到次序颠倒的存储操作。

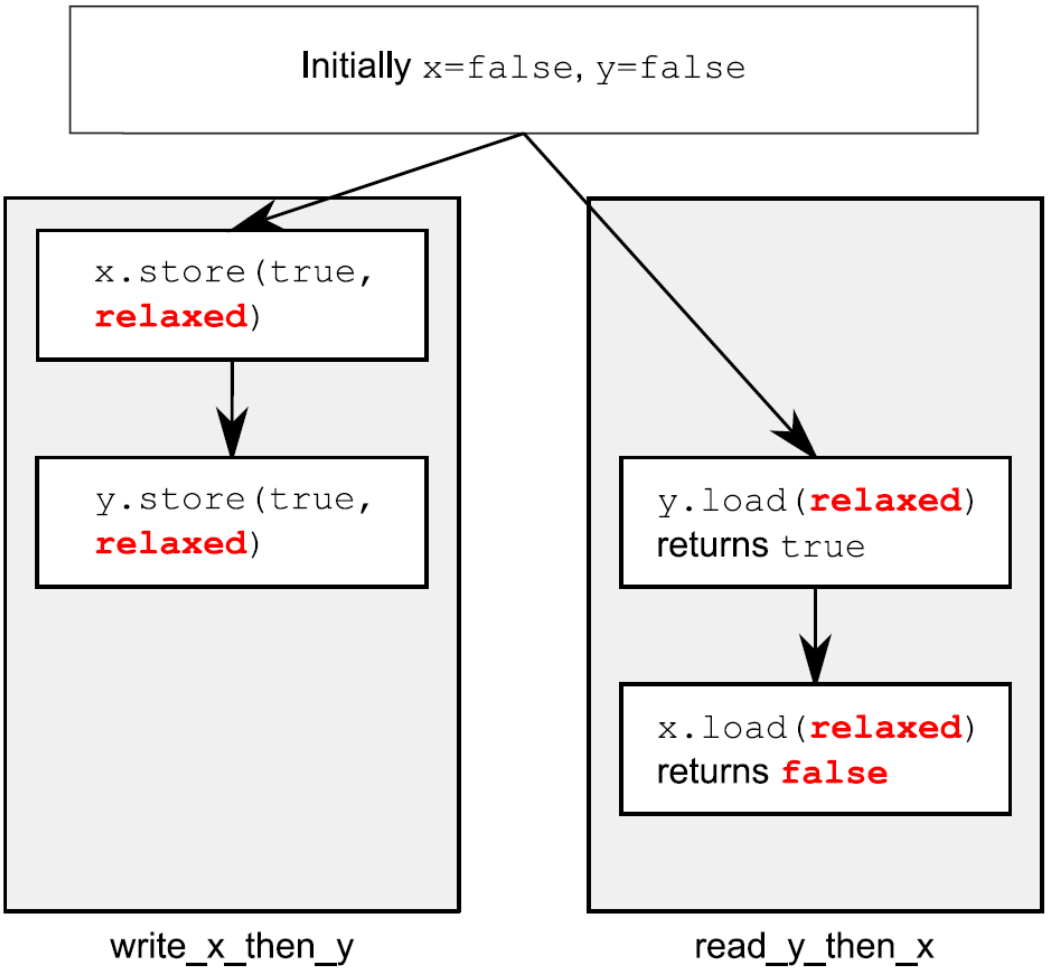


图 5.4 宽松原子操作与“先发生于”

来看一个略微复杂的例子，它有三个变量和五个线程。

清单 5.6 多线程版上的宽松操作

```
#include <thread>
```

```cpp
#include <atomic>
#include <iostream>

std::atomic<int> x(0),y(0),z(0);  // 1
std::atomic<bool> go(false);  // 2

unsigned const loop_count=10;

struct read_values
{
  int x,y,z;
};

read_values values1[loop_count];
read_values values2[loop_count];
read_values values3[loop_count];
read_values values4[loop_count];
read_values values5[loop_count];

void increment(std::atomic<int>* var_to_inc,read_values* values)
{
  while(!go)
    std::this_thread::yield();  // 3 自旋，等待信号
  for(unsigned i=0;i<loop_count;++i)
  {
    values[i].x=x.load(std::memory_order_relaxed);
    values[i].y=y.load(std::memory_order_relaxed);
    values[i].z=z.load(std::memory_order_relaxed);
    var_to_inc->store(i+1,std::memory_order_relaxed);  // 4
    std::this_thread::yield();
  }
}

void read_vals(read_values* values)
{
  while(!go)
    std::this_thread::yield(); // 5 自旋，等待信号
  for(unsigned i=0;i<loop_count;++i)
  {
    values[i].x=x.load(std::memory_order_relaxed);
    values[i].y=y.load(std::memory_order_relaxed);
    values[i].z=z.load(std::memory_order_relaxed);
    std::this_thread::yield();
  }
```

```
}

void print(read_values* v)
{
  for(unsigned i=0;i<loop_count;++i)
  {
    if(i)
      std::cout<<",";
    std::cout<<"("<<v[i].x<<","<<v[i].y<<","<<v[i].z<<")";
  }
  std::cout<<std::endl;
}

int main()
{
  std::thread t1(increment,&x,values1);
  std::thread t2(increment,&y,values2);
  std::thread t3(increment,&z,values3);
  std::thread t4(read_vals,values4);
  std::thread t5(read_vals,values5);

  go=true;  // 6 开始执行主循环的信号

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

  print(values1);  // 7 打印最终结果
  print(values2);
  print(values3);
  print(values4);
  print(values5);
}
```

这是个简单的程序。有三个全局原子变量①和五个线程。每一个线程循环 10 次，使用 memory_order_relaxed 读取三个原子变量的值，并且将它们存储在一个数组上。其中三个线程每次通过循环④来更新其中一个原子变量，而剩下的两个线程负责读取。当线程都被连接（joined）了，就打印出每个线程存到数组上的值。

原子变量 go②用来确保线程尽可能接近同一时间开始循环。启动线程是昂贵的操作，并且没有明确的延迟，第一个线程可能在最后一个线程开始前结束。每个线程在进入主循环之前都在等待 go 变为 true③和⑤，并且一旦 go 设置为 true 所有线程都会开始运行⑥。

程序一种可能的输出如下：

(0,0,0),(1,0,0),(2,0,0),(3,0,0),(4,0,0),(5,7,0),(6,7,8),(7,9,8),(8,9,8),(9,9,10)
(0,0,0),(0,1,0),(0,2,0),(1,3,5),(8,4,5),(8,5,5),(8,6,6),(8,7,9),(10,8,9),(10,9,10)
(0,0,0),(0,0,1),(0,0,2),(0,0,3),(0,0,4),(0,0,5),(0,0,6),(0,0,7),(0,0,8),(0,0,9)
(1,3,0),(2,3,0),(2,4,1),(3,6,4),(3,9,5),(5,10,6),(5,10,8),(5,10,10),(9,10,10),(10,10,10)
(0,0,0),(0,0,0),(0,0,0),(6,3,7),(6,5,7),(7,7,7),(7,8,7),(8,8,7),(8,8,9),(8,8,9)

前三行中线程做了更新，后两行线程只是做读取。每三个值都是一组 x，y 和 z，并按照这个顺序从一个循环通过。在这个输出中有几件事情需要注意：

- 第一组值中 x 在每个三元组中每次增 1，第二组中 y 每次增 1，并且第三组中 z 每次增 1。
- 在给定集和中 x 元素只会增加，y 和 z 也一样，但是增加是不均匀的，并且相对顺序在所有线程中都不同。
- 线程 3 看不到 x 或 y 的任何更新；它只看到它对 z 的更新。这并不妨碍别的线程观察到 z 的更新与对 x 和 y 的更新混在一起。

对于宽松操作来讲，这是个有效的结果，但不是唯一有效的结果。与这三个变量一致的任何一组值，每个变量依次保存 0 到 10 的值，并让递增一个给定变量的线程，为这个变量打印从 0 到 9 的值，都是有效的。

**理解“宽松”顺序**

为了理解这是如何工作的，想象一下，每个变量是一个在小隔间里拿着记事本的人。他的记事本上是一组值的列表。你可以通过打电话的方式让他给你一个值，或你可以告诉他，让他写下一个新值。如果你告诉他写下一个新值，他会将这个新值写在列表的最后。如果你让他给你一个值，他会从列表中读取一个值给你。

当你第一次与这个人交谈时，如果你问他要一个值，他可能会给你那时列表中的任意值。如果之后你再问他要一个值，它可能会再给你同一个值，或将列表更下面的值给你，他不会给你列表更上端的值。如果你让他写下一个数字，并且随后再问他要一个值，他要不就给你你刚告诉他写下的那个数字，要不就是清单上比这个值更靠下的一个值。

想像一下，他的列表上开始有 5，10，23，3，1，2 这几个值。如果你问他要一个值，你可能获取这几个数中的任意一个。如果他给你 10，那么下次再问他要值的时候可能会再给你 10，或者 10 后面的数，但绝对不会是 5。如果你问他要了五次，他就可能回答“10，10，1，2，2”。如果你让他写下 42，他将会把这个值添加在列表的最后。如果你再问他要值，他可能会继续告诉你“42”，直到有另一个值在他的列表上并且他想将那个数告诉你。

现在，想象你的朋友 Carl，他也有那个人的电话。Carl 也可以打电话给他，并让他写下一个值或要一个值，他回应 Carl 的规则和回应你是一样的。他只有一部电话，所以他一次只能处理一个人的请求，所以他记事本上的列表是一个简单的列表。但是，你让他写下一个新值的时候，并不意味着他必须把它告诉 Carl，反之亦然。如果 Carl 从他那里获取一个值“23”，之后因为你告诉他写下 42，但并不意味着下次他会把它告诉 Carl。他可能会告诉 Carl 任意一个值，23，3，1，2，42 亦或是 67，这是 Fred 在你之后告诉他的。他会很高兴的告诉 Carl“23，3，3，1，67”，与他告诉你的值不一致。就像他用一个可移动的便利贴记录他告诉了谁的数字，就像图 5.5 那样。

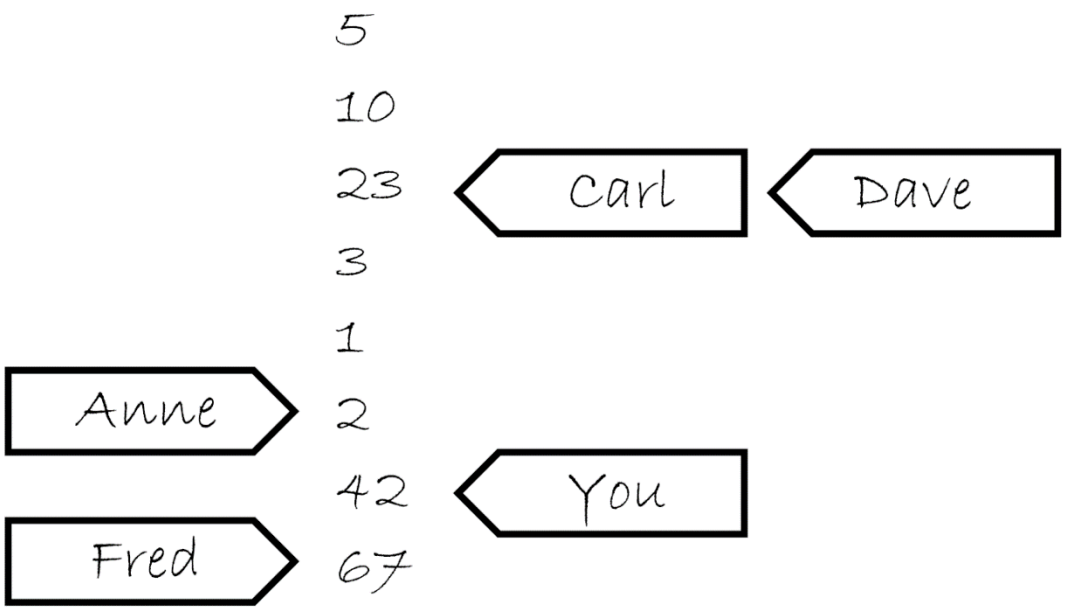


图 5.5 小隔间里的人用的笔记本

现在想象一下，不是只有一个人在一个小隔间里，而是整个小隔间组成的办公室（cubicle farm），里面有很多拿着电话和记事本的人。这些就是我们的原子变量。每一个变量拥有自己的修改顺序(笔记上的值列表)，但是每个原子变量之间没有任何关系。如果每一个打电话的人(你，Carl，Anne，Dave 和 Fred)是一个线程，这就是每个操作使用 memory_order_relaxed 所得到的结果。还有些其他的事情你可以告诉小隔间里的人，例如：“写下这个数字，并且告诉我现在列表中的最后一个数字”(exchange)，或“写下这个数字，当列表的最后一个数字为某值；否则的话，告诉我我猜的那个值是多少”(compare_exchange_strong)，但是这都不影响一般性原则。

如果你思考清单 5.5 的逻辑，那么 write_x_then_y 就像某人打电话给小隔间 x 里的人，并且告诉他写下 true，之后打电话给在小隔间 y 的另一个人，并告诉他写下 true。线程反复运行 read_y_then_x，打电话给小隔间 y 的人问他要值，直到他说 true，然后打电话给小隔间 x 的人，问他要一个值。在小隔间 x 中的人没有义务告诉你他清单上的任何具体值，他完全有权利说 false。

这就让宽松的原子操作变得难以处理。他们必须与原子操作结合使用，这些原子操作必须有较强的顺序语义，从而让这种结合对“线程间同步”更有用。我强烈建议避免宽松

的原子操作，除非它们绝对必要，并且在使用它们的时候需要万分谨慎。在清单 5.5 中仅使用两个线程和两个变量就能给出不直观的结果，不难想象在有更多线程和更多变量时，情况会变的多么复杂。

要想获得额外的同步，而不想要“序列一致”的全面开销，一种方法是使用*获得-释放顺序*(acquire-release ordering)。

**“获得-释放”顺序**

“获得-释放”顺序是*“宽松”顺序的增强*；这里仍然没有操作的全序，但是它确实引入了同步。这种顺序模型中，原子加载是*获得*(acquire)操作(memory_order_acquire)，原子存储是*释放*(memory_order_release)操作，原子读-改-写操作(例如 fetch_add()或 exchange())可以是“获得”，“释放”，中的一个，或者两者都有(memory_order_acq_rel)。同步在执行“释放”的线程和执行“获得”的线程之间是成对的。一个“释放”操作“同步于”一个读取写入值的“获得”操作。这意味着不同的线程仍然可以看到不同的顺序，但是这些顺序是受限制的。下面的清单使用“获得-释放”语义而不是“序列一致”语义对清单 5.4 进行了修改。

清单 5.7 “获得-释放”并不意味着全序

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

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))  // 1
    ++z;
}

void read_y_then_x()
```

```
{
  while(!y.load(std::memory_order_acquire));
  if(x.load(std::memory_order_acquire))  // 2
    ++z;
}

int main()
{
  x=false;
  y=false;
  z=0;
  std::thread a(write_x);
  std::thread b(write_y);
  std::thread c(read_x_then_y);
  std::thread d(read_y_then_x);
  a.join();
  b.join();
  c.join();
  d.join();
  assert(z.load()!=0); // 3
}
```

这里 assert③可能会触发(就如同宽松顺序那样)，因为可能加载 x②和 y①的时候都读取到的是 false。由于 x 和 y 是由不同线程写入的，所以每种情况中从“释放”到“获得”的顺序对其他线程的操作没有作用。

图 5.6 展示了清单 5.7 的“先发生于”关系，连同一个可能的结果，在这里两个读线程每个都有一个不同的视图。这是可能的，如前所述，因为没有“先发生于”关系来强制一个顺序。

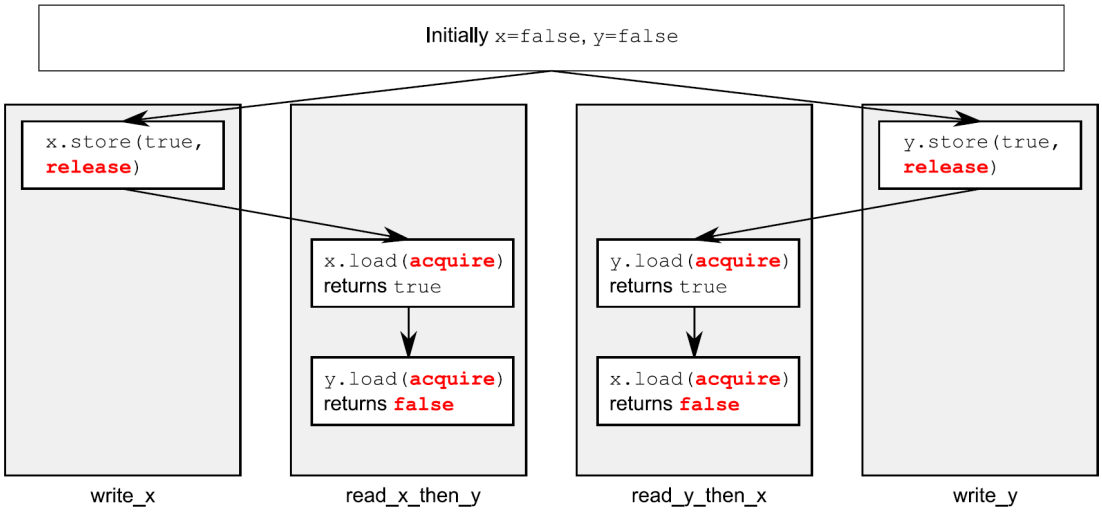


图 5.6 “获得-释放”与“先发生于”

为了了解获得-释放顺序的优点，需要考虑将两次存储由一个线程来完成，就像清单5.5那样。当需要使用memory_order_release改变对y的存储，并且使用memory_order_acquire来加载y中的值，就像下面程序清单所做的那样，那么你就对x上的操作强加了一个顺序。

清单5.8 “获得-释放”操作可以在宽松操作上强加顺序

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y()
{
  x.store(true,std::memory_order_relaxed);  // 1
  y.store(true,std::memory_order_release);  // 2
}

void read_y_then_x()
{
  while(!y.load(std::memory_order_acquire));  // 3 自旋，等待y被设置为
true
  if(x.load(std::memory_order_relaxed))  // 4
    ++z;
}

int main()
{
  x=false;
  y=false;
  z=0;
  std::thread a(write_x_then_y);
  std::thread b(read_y_then_x);
  a.join();
  b.join();
  assert(z.load()!=0);  // 5
}
```

最后，加载y③时会看到true，它是存储操作写入的②。因为存储使用的是memory_order_release，读取使用的是memory_order_acquire，存储就“同步于”读取。因为这两个操作在同一个线程，所以存储x①的操作“先发生于”存储y②的操作。对y的存储“同步于”对y的加载，存储x也就“先发生于”对y的加载，并且引申开来，“先发生于”x的加载。因此，加载x的值必为true，并且断言⑤不会触发。如果来自y的加

载不在 while 循环中，就不一定是这样；y 的加载可能读到 false，在这种情况下，对从 x 读取的值没有要求。为了提供任何同步，获得和释放操作必须成对进行。释放操作所存储的值必须被获得操作看到才能产生效果。如果②的存储或③的加载，有一个是宽松顺序，对 x 的访问就无序了，也就无法保证④处的加载读到的是 true，并且断言可以触发。

你仍然可以从我们的“小隔间里拿着笔记本的人”角度思考“获得-释放”顺序，不过你必须添加更多东西到这个模型中。首先，想象一下，每个已经完成的存储都是某一批更新的一部分，所以当你打电话让一个人写下一个数字时，你也要告诉他这个更新属于哪一批：“请记下 99，作为第 423 批的一部分。”对于一批中最后的存储，你也要告诉他：“请记下 147，这是第 423 批中最后的存储。”然后，小隔间里的人会及时写下这些信息，以及谁给了他这个值。这建模为存储-释放操作。下次你告诉别人写下一个值时，你增加批号：“请记下 41，作为第 424 批的一部分。”

当你请求一个值得时候，就要做出一个选择：要么请求一个值(这就是一次宽松加载），这种情况下，小隔间中的人只会给你一个数字，要么就请求一个值以及它是否是某批次中的最后一个（这就是加载-获得模型)。如果你询问批次信息，且这个值不是批次中的最后一个，小隔间里的人会这样告诉你，“数字是 987，它是一个‘普通’值”，而当这个值是最后一个时，他会告诉你：“数字为 987，这个数字是 956 批次的最后一个，来源于 Anne”。现在，这里是“获得-释放”语义发挥作用的地方：当你请求一个值时，如果你告诉他所有你知道的批次，他会从他的清单上查找你知道的任何批次的最后一个值，然后要么给你那个数字，要么给你列表上更下面的一个。

这个是怎么建模“获得-释放”语义的？让我们看一下例子。首先，线程 a 运行 write_x_then_y，并且告诉在小隔间 x 里的人，“请写下 true 作为来自线程 a 的批次 1 的一部分”，他及时记下了。然后，线程 a 告诉在小隔间 y 里的人，“请写下 true，这是来自线程 a 的批次 1 的最后一个写”，他及时记下了。与此同时，线程 b 正在运行 read_y_then_x。线程 b 持续向小隔间 y 里的人请求一个带批次信息的值，直到他说“true”。可能需要请求很多次，不过最终这个人将说“true”。小隔间 y 里的人不仅仅是说“true”，他还说“这是来自线程 a 的批次 1 的最后一个写”。

现在，线程 b 继续向小隔间 x 里的人请求一个值，但这次他会说“请给我一个值，并且我从线程 a 知道批次 1”。 现在，这个来自小隔间 x 的人不得不查看他的列表，寻找线程 a 最后提到的批次 1。他唯一提到的是值 true，这也是他列表中的最后一个值，所以他必须读出这个值；否则，他将打破游戏规则。

如果你回看 5.3.2 节中对“线程间先发生于”的定义，一个很重要的特性就是它是可传递的：如果 A“线程间先发生于”B，并且 B“线程间先发生于”C，那么 A 就“线程间先发生于”C。这就意味着，获得-释放顺序可以用于在若干线程间同步数据，即使“中间”线程没有接触到数据。

**用“获得-释放”顺序传递同步**

为了思考传递顺序，至少需要三个线程。第一个线程修改一些共享变量，并对其中一个进行“存储-释放”（store-release）。然后，第二个线程使用“加载-获得”（load-acquire）读取“存储-释放”相关的变量，并对第二个共享变量执行“存储-释放”。最

后，第三个线程对第二个共享变量执行“加载-获得”。如果“加载-获得”操作看到“存储-释放”操作写入的值，以确保“同步于”关系，那么第三个线程可以读取第一个线程存储的其他变量的值，即使中间线程没有接触它们中的任何一个。这个场景在下一个清单中说明。

清单 5.9 使用获得和释放顺序传递同步

```
std::atomic<int> data[5];
std::atomic<bool> sync1(false),sync2(false);

void thread_1()
{
  data[0].store(42,std::memory_order_relaxed);
  data[1].store(97,std::memory_order_relaxed);
  data[2].store(17,std::memory_order_relaxed);
  data[3].store(-141,std::memory_order_relaxed);
  data[4].store(2003,std::memory_order_relaxed);
  sync1.store(true,std::memory_order_release);  // 1.设置 sync1
}

void thread_2()
{
  while(!sync1.load(std::memory_order_acquire));  // 2.循环直到 sync1 被设置
  sync2.store(true,std::memory_order_release);  // 3.设置 sync2
}
void thread_3()
{
  while(!sync2.load(std::memory_order_acquire));   // 4.循环直到 sync2 被设置
  assert(data[0].load(std::memory_order_relaxed)==42);
  assert(data[1].load(std::memory_order_relaxed)==97);
  assert(data[2].load(std::memory_order_relaxed)==17);
  assert(data[3].load(std::memory_order_relaxed)==-141);
  assert(data[4].load(std::memory_order_relaxed)==2003);
}
```

尽管 thread_2 只接触到变量 syn1②和 sync2③，不过这对于 thread_1 和 thread_3 的同步就足够了，这能保证断言不会触发。首先，thread_1 将数据存储到 data 中“先发生于”存储 sync1①，因为它们是同一线程内的“先序于”关系。因为加载 sync1②的是一个 while 循环，它最终将看到从 thread_1 存储的值，并形成释放-获得对的后半部分。因此，对于 sync1 的存储“先发生于”最终在 while 循环中对 sync1 的加载。这个加载“先序于”（因而“先发生于”）sync2 的存储③，这和 thread_3 中 while 循环的最终加载④形成了一个“释放-获得”对。sync2 的存储③因此“先发生于”加载④，这个加载“先发生于”对 data 的加载。由于“先发生于”的传递天性，你可以把它们串联起来：data 的

存储“先发生于”sync1 的存储①，sync1 的存储又“先发生于”sync1 的加载②，sync1 的加载又“先发生于”sync2 的存储③，sync2 的存储“先发生于”sync2 的加载④，它又“先发生于”data 的加载。因而 thread_1 中对 data 的存储“先发生于”thread_3 中从 data 的加载，assert 也就不可能触发。

在这种情况下，你可以通过使用 thread_2 中的 memory_order_acq_rel 中的“读-改-写”操作将 sync1 和 sync2 组合为单个变量。一个选项是使用 compare_exchange_strong()来确保值只在看到 thread_1 的存储时才更新：

```
std::atomic<int> sync(0);
void thread_1()
{
  // ...
  sync.store(1,std::memory_order_release);
}

void thread_2()
{
  int expected=1;
  while(!sync.compare_exchange_strong(expected,2,
              std::memory_order_acq_rel))
    expected=1;
}

void thread_3()
{
  while(sync.load(std::memory_order_acquire)<2);
  // ...
}
```

如果使用“读-改-写”操作，选择你想要的语义就很重要了。在本例中，你获得和释放两个语义都需要，因此 memory_order_acq_rel 是合适的，但是你也可以使用其他顺序。具有 memory_order_acquire 语义的 fetch_sub 操作不会与任何东西同步，即使它存储一个值，因为它不是一个释放操作。同样，存储不能与具有 memory_order_release 语义的 fetch_or 同步，因为 fetech_or 的读取部分不是获得操作。具有 memory_order_acq_rel 语义的“读-改-写”操作既可以作为获得操作，也可以作为释放操作，因此先前的存储可以与这样的操作同步，它也可以与随后的加载同步，就像本例中的情况一样。

如果将“获得-释放”操作和“序列一致”操作进行混合，“序列一致”的加载动作，就像使用了“获得”语义的加载操作；并且“序列一致”的存储操作，就如使用了“释放”语义的存储。“序列一致”的“读-改-写”操作行为，就像同时使用了“获取”和“释放”的操作。“宽松”操作依旧那么宽松，但其会被通过使用“获得-释放”引入的额外的“同步于”以及随之而来的“先发生于”关系所束缚。

尽管结果并不那么直观，每个使用锁的人都需要解决同样的顺序问题：锁住互斥锁是一个获得操作，并且解锁这个互斥锁是一个释放操作。对于互斥锁，你已经知道必须确保在读取值时锁住的互斥锁和修改值时锁住的互斥锁是同一个锁，同样的道理适合于这里；“获得和释放”操作必须在同一个变量上，以保证访问顺序。当数据被一个互斥锁保护时，锁的排它性质意味着没法区分上锁和解锁是否是“序列一致”操作。类似地，如果对原子变量使用获得和释放顺序构建一个简单的锁，那么从使用锁的代码看来，行为将呈现“序列一致”，尽管内部操作并非如此。

如果你的原子操作不需要严格的“序列一致”顺序，那么“获得-释放”顺序的成对同步可能会比“序列一致”操作所需的全局顺序的同步成本低得多。这里的权衡是为了确保顺序正确工作，以及确保线程之间的非直观行为没有问题而需要的脑力成本。

**获得-释放顺序和 memory_order_consume 的数据相关性**

在本节介绍的时候，我说过 memory_order_consume 是“获得-释放”顺序模型的一部分，但它显然不在前面的描述中。这是因为 memory_order_consume 很特别：它是关于数据依赖的，并且它引入了数据依赖的细微差别到 5.3.2 节提到的“线程间先发生于”关系。它的特别之处还在于 C++17 标准明确建议你不要使用它。因此，这里只是为了完整性而介绍：你不应该在代码中使用 memory_order_consume!

数据依赖的概念相对简单：如果第二个操作对第一个操作的结果进行操作，那么两个操作之间就存在数据依赖关系。有两种新关系用来处理数据依赖：“*依赖先序于*”(dependency-ordered-before)和“*依赖带入*”(carries-a-dependency-to)。就像“先序于”，“依赖带入”严格应用于一个线程内并在操作之间塑造数据依赖；如果操作 A 的结果作为一个操作数应用于操作 B，那么 A“依赖带入”B。如果 A 操作的结果是一个标量，比如 int，当 A 的结果存储在一个变量中，并且这个变量作为操作数被操作 B 使用，那么关系仍然存在。这个操作也是可以传递的，所以当 A“依赖带入”B，并且 B“依赖带入”C，就可以得出 A“依赖带入”C 的关系。

另一方面，“依赖先序于”关系可以应用于线程间，通过 std::memroy_order_consume 的原子加载操作引入。它是 std::memroy_order_acquire 的特殊情况，限制了同步数据为直接依赖；标记为 memory_order_release，memory_order_acq_rel，memory_order_seq_cst 的存储操作 A“依赖先序于”标记为 memroy_order_consume 的加载操作 B，如果消费操作读取了存储的值。这和你用 memroy_order_acquire 加载获得的“同步于”关系截然相反（译注：把“同步于”和“依赖先序于”看做一堵墙，“同步于”相当于墙左边的操作可以往前延伸，而“依赖先序于”相当于往墙的右边往后延伸，所以说相反，参照后一句话，和下面的例子就好理解了）。如果操作 B“依赖带入”某个操作 C，那么 A 也“依赖先序于”C。

如果它不影响“线程间先发生于”关系时，对于同步来说并未带来任何的好处，但是它确实有影响：当 A“依赖先序于”B，那么 A“线程间先发生于”B。

这种内存顺序的一个很重要使用场景是在原子操作载入指向某个数据的指针时。通过使用 memory_order_consume 作为加载语义，并且 memory_order_release 作为之前的存储

语义，你保证了指针指向的值是正确同步的，没有强加任何在不相关数据上的同步需求。下面的代码就展示了这么一个场景。

清单 5.10 使用 std::memroy_order_consume 同步数据

```
struct X
{
int i;
std::string s;
};

std::atomic<X*> p;
std::atomic<int> a;

void create_x()
{
  X* x=new X;
  x->i=42;
  x->s="hello";
  a.store(99,std::memory_order_relaxed);  // 1
  p.store(x,std::memory_order_release);  // 2
}

void use_x()
{
  X* x;
  while(!(x=p.load(std::memory_order_consume)))  // 3
    std::this_thread::sleep(std::chrono::microseconds(1));
  assert(x->i==42);  // 4
  assert(x->s=="hello");  // 5
  assert(a.load(std::memory_order_relaxed)==99);  // 6
}

int main()
{
  std::thread t1(create_x);
  std::thread t2(use_x);
  t1.join();
  t2.join();
}
```

尽管，对 a 的存储①“先序于”存储 p②，并且存储 p 的操作标记为 memory_order_release，加载 p③的操作标记为 memory_order_consume，这意味着存储 p 仅“先发生于”那些依赖于从 p 加载的值的表达式。这意味着 X 结构体中数据成员所在的断言语句④⑤不会被触发，因为加载 p 的操作“依赖带入”对 x 变量操作的表达式。另一

方面，对于加载变量 a⑥的断言可能会也可能不会触发；这个操作并不依赖于从 p 加载的值，所以这里没法保证读取的值。当然，这个情况也很明显，因为这个操作被标记为 memory_order_relaxed。

有时，不想为携带依赖增加其他开销。想让编译器在寄存器中缓存这些值，以及重排来优化代码，而不是担心依赖关系。在这些场景，可以使用 std::kill_dependecy()来显式打破依赖链。std::kill_dependency()是一个简单的函数模板，会拷贝提供的参数给返回值，但是破坏了依赖链。例如，如果你拥有一个全局的只读数组，当另一个线程对数组索引进行检索时，你使用的是 std::memory_order_consume，那么你可以使用 std::kill_dependency()让编译器知道这里不需要重新读取该数组的内容，就像下面的例子一样：

```
int global_data[]={ … };
std::atomic<int> index;

void f()
{
  int i=index.load(std::memory_order_consume);
  do_something_with(global_data[std::kill_dependency(i)]);
}
```

在实际代码中，你应该总是在可能试图使用 memory_order_consume 的地方使用 memory_order_acquire，而 std::kill_dependency 是不必要的。

既然我已经介绍了内存顺序的基础知识，现在就来看看“同步于”关系中更复杂的部分，它们以*释放序列*（release sequences）的形式表现出来。

## 5.3.4 “释放序列”与“同步于”

回到 5.3.1 节，我提到过你可以在存储一个原子变量和另一个加载这个原子变量的线程之间获得一个“同步于”关系，即使有一系列的“读-改-写”操作在存储和加载之间，只要所有操作都被适当标记就没问题。现在，已经介绍了所有可能使用到的内存顺序“标签”，在这里可以做一个详细说明。当存储操作被标记为 memory_order_release，memory_order_acq_rel 或 memory_order_seq_cst，加载被标记为 memory_order_consum，memory_order_acquire 或 memory_order_sqy_cst，并且操作链上的每个操作加载的值是由前面的操作写入的，那么操作链就构成了一个*释放序列*(*release sequence*)，并且最初的存储“同步于”(对应 memory_order_acquire 或 memory_order_seq_cst)或是“依赖先序于”(对应 memory_order_consume)最终的加载。操作链上的任何原子“读-改-写”操作可以拥有任意的内存顺序(甚至是 memory_order_relaxed)。

为了了解这些操作意味着什么，以及它的重要性，考虑一个 atomic<int>用作对一个共享队列的条目数进行计数，如下面清单所示：

清单 5.11 使用原子操作从队列中读取值

```cpp
#include <atomic>
#include <thread>

std::vector<int> queue_data;
std::atomic<int> count;

void populate_queue()
{
  unsigned const number_of_items=20;
  queue_data.clear();
  for(unsigned i=0;i<number_of_items;++i)
  {
    queue_data.push_back(i);
  }

  count.store(number_of_items,std::memory_order_release);  // 1 初始化存储
}

void consume_queue_items()
{
  while(true)
  {
    int item_index;
    if((item_index=count.fetch_sub(1,std::memory_order_acquire))<=0)  // 2 一个“读-改-写”操作
    {
      wait_for_more_items();  // 3 等待更多元素
      continue;
    }
    process(queue_data[item_index-1]);  // 4 安全读取 queue_data
  }
}

int main()
{
  std::thread a(populate_queue);
  std::thread b(consume_queue_items);
  std::thread c(consume_queue_items);
  a.join();
  b.join();
  c.join();
}
```

一种处理方式是让线程产生数据，并存储到一个共享缓冲区中，然后调用 count.store(number_of_items, memory_order_release)①让其他线程知道数据是可用的。这些线程会消费队列中的项，在读取共享缓冲区④之前可能调用 count.fetch_sub(1, memory_order_acquire)②向队列索取一项。一旦 count 归零，队列中就没有更多项了，然后线程必须等待③。

如果只有一个消费者线程时还好，fetch_sub()是一个带有 memory_order_acquire 的读取操作，并且存储操作是带有 memory_order_release 语义，所以存储“同步于”加载，线程可以从缓冲区中读取项。如果有两个读取线程时，第二个 fetch_sub()操作将看到被第一个线程写入的值，而不是 store 写入的值。如果没有释放序列的规则，第二个线程与第一个线程不会有“先发生于”关系，因此对共享缓冲区中值的读取也是不安全，除非第一个 fetch_sub()也有 memory_order_release 语义，这个语义为两个消费者线程引入了不必要的同步。要没有释放序列的规则，或者 memory_order_release 语义的 fetch_sub 操作，第二个消费者轻易就可以看到 queue_data 的存储操作，你将面临数据竞争问题。好在第一个 fetch_sub()参与到“释放序列”中，所以 store()能同步于第二个 fetch_sub()操作。两个消费者线程间仍然不存在“同步于”关系。图 5.7 的虚线中展示了“释放序列”，实线展示的了“先发生于”关系。

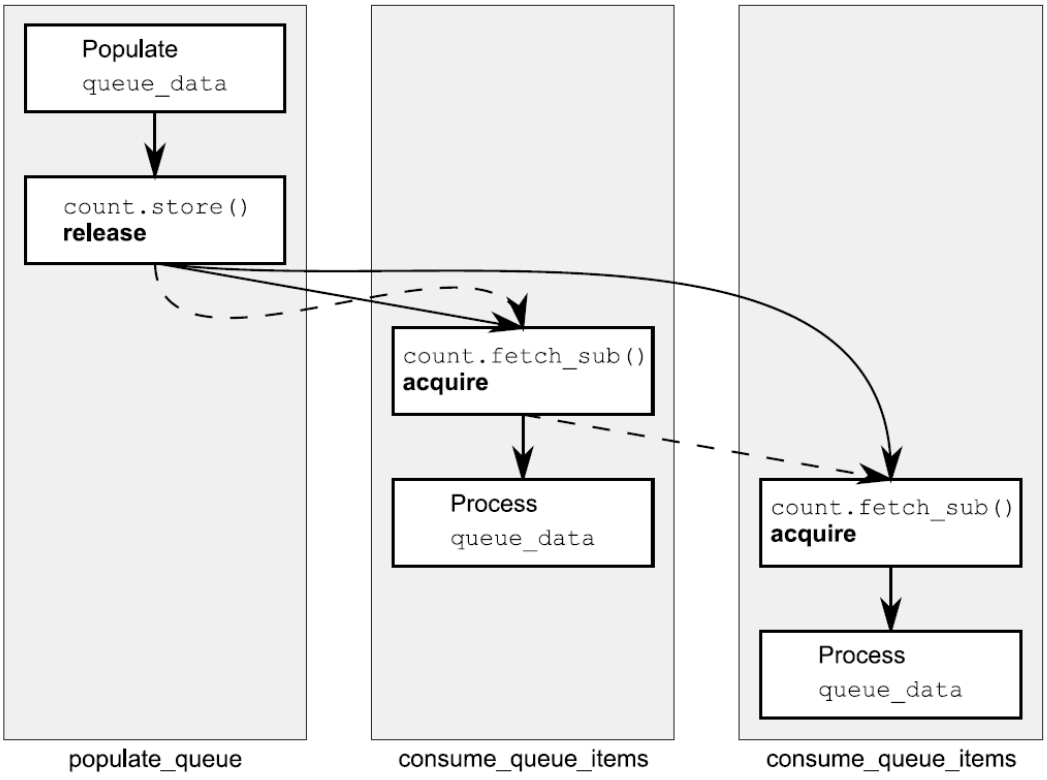


图 5.7 清单 5.11 中对队列操作的“释放序列”

操作链中可以有任意数量的链接，但只要它们都是“读-改-写”操作，比如 fetch_sub()，store()仍然会与标记为 memory_order_acquire 的每个操作同步。例子中所有链接都是一样的，并且都是获得操作，但它们可以是不同内存顺序语义组成的不同操作的混合。

虽然，大多数同步关系，源自于对原子变量上的操作应用的内存顺序语义，但也可以通过使用*栅栏*（fence）引入额外的顺序约束。

### 5.3.5 栅栏

如果没有一组栅栏，原子操作库就不完整。这些操作在不修改任何数据的情况下强制执行内存顺序约束，并且通常与使用 memory_order_relax 顺序约束的原子操作结合使用。栅栏是个全局操作，它会影响执行栅栏的线程中其他原子操作的顺序。栅栏也通常叫做“内存屏障”（memory barriers），它们之所以得名是因为它们在代码中划了一条线，某些操作不能跨越它。你可能回想起 5.3.3 节中不同变量上的宽松操作通常可以被编译器或者硬件自由的重排。栅栏限制了这个自由度并且引入了以前不存在的“先发生于”和“同步于”关系。

让我们从在清单 5.5 中每个线程的两个原子操作之间添加一个栅栏开始，如下面的清单所示。

清单 5.12 宽松操作可以使用栅栏排好顺序

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y()
{
  x.store(true,std::memory_order_relaxed);  // 1
  std::atomic_thread_fence(std::memory_order_release);  // 2
  y.store(true,std::memory_order_relaxed);  // 3
}

void read_y_then_x()
{
  while(!y.load(std::memory_order_relaxed));  // 4
  std::atomic_thread_fence(std::memory_order_acquire);  // 5
  if(x.load(std::memory_order_relaxed))  // 6
    ++z;
}

int main()
{
  x=false;
  y=false;
```

```
  z=0;
  std::thread a(write_x_then_y);
  std::thread b(read_y_then_x);
  a.join();
  b.join();
  assert(z.load()!=0);  // 7
}
```

释放栅栏②“同步于”获得栅栏⑤，这是因为加载 y 的操作④读取③存储的值。这意味着存储 x①“先发生于”加载 x⑥，所以读出来的值肯定是 true，并且断言不会被触发⑦。这与最初的情况形成了对比，在没有栅栏的情况下，对 x 的存储和从 x 的加载没有顺序，因此断言可以触发。注意，两个栅栏都是必要的：你需要在一个线程中释放，在另一个线程中获得，从而获得“同步于”关系。

这个例子中，释放栅栏②的效果就好像对 y 的存储③标记为 memory_order_release 而不是 memory_order_relaxed。同样，获得栅栏⑤就好像加载 y 的操作④标记为 memory_order_acquire。这是使用栅栏的总体思路：当获得操作看到在释放栅栏后存储的结果，那么这个栅栏就“同步于”获得操作；并且，如果在获得栅栏操作前的加载操作，看到一个释放操作的结果，那么这个释放操作“同步于”获得栅栏。你可以在两边都设置栅栏，如这里的示例，在这种情况下，如果在获得栅栏前的加载看到了释放栅栏后面的存储，那么释放栅栏“同步于”获得栅栏。

虽然，栅栏同步依赖于读取/写入的操作发生于栅栏的前/后，但需要重点注意的是同步点就是栅栏本身。如果你把清单 5.12 中的 write_x_then_y 拿出来，并且把对 x 的写移到栅栏后面，就像下面的代码一样。断言中的条件就不保证一定为 true 了，尽管写入 x 的操作在写入 y 的操作之前发生。

```
void write_x_then_y()
{
  std::atomic_thread_fence(std::memory_order_release);
  x.store(true,std::memory_order_relaxed);
  y.store(true,std::memory_order_relaxed);
}
```

这两个操作不再被栅栏隔开，因此不再有顺序。只有当栅栏位于对 x 的存储和对 y 的存储之间才能强加一个顺序。栅栏的存在与否不会影响“先发生于”关系上的强制顺序，这个关系是由于其他原子操作而存在的。

这个例子，以及本章到目前为止的几乎所有其他例子，都完全是由原子类型的变量构建的。但是使用原子操作来强制顺序的真正好处是它们可以强制非原子操作的顺序，并避免未定义的数据竞争，如清单 5.2 所示。

## 5.3.6 使用原子操作对非原子操作排序

如果使用普通的非原子 bool 类型来替换清单 5.12 中的 x(如同下面的代码)，行为可以保证是相同的。

清单 5.13 在非原子操作上强制顺序

```
#include <atomic>
#include <thread>
#include <assert.h>

bool x=false;  // x 现在是一个普通非原子变量
std::atomic<bool> y;
std::atomic<int> z;

void write_x_then_y()
{
  x=true;  // 1 在栅栏前存储 x
  std::atomic_thread_fence(std::memory_order_release);
  y.store(true,std::memory_order_relaxed);  // 2 在栅栏后存储 y
}

void read_y_then_x()
{
  while(!y.load(std::memory_order_relaxed));  // 3 持续等待直到看到 2 的写
  std::atomic_thread_fence(std::memory_order_acquire);
  if(x)  // 4 这将读到 1 中写入的值
    ++z;
}

int main()
{
  x=false;
  y=false;
  z=0;
  std::thread a(write_x_then_y);
  std::thread b(read_y_then_x);
  a.join();
  b.join();
  assert(z.load()!=0);  // 5 断言将不会触发
}
```

栅栏仍然为存储 x①和存储 y②，还有加载 y③和加载 x④提供一个强制顺序，并且这里仍然有一个“先发生于”关系在存储 x 和加载 x 之间，所以断言⑤不会被触发。②中的

存储和③对 y 的加载，仍然必须是原子操作；否则，将会在 y 上产生数据竞争，不过一旦读取线程看到 y 中存储的值，栅栏将会对 x 上的操作强制一个顺序。这个强制顺序意味着，即使它被一个线程修改并且被另外一个线程读取，x 上也不存在数据竞争。

不仅栅栏可以排序非原子操作。你已经看到了在清单 5.10 中 memory_order_release/memory_order_consume 对顺序的影响，它们会对动态分配对象的非原子访问操作进行排序，本章中的很多例子都可以通过用普通的非原子操作替换某些 memory_order_relax 操作来重写。

## 5.3.7 排序非原子操作

通过使用原子操作来对非原子操作进行排序，是凸显“先序于”是“先发生于”一部分的重要性的地方。如果一个非原子操作“先序于”一个原子操作，并且这个原子操作“先发生于”另一个线程中的操作，那么这个非原子操作也会“先发生于”另一个线程中的那个操作。这就是清单 5.13 中对 x 的操作顺序的来源，也是清单 5.2 中的示例可以工作的原因。这也是 C++标准库中更高层次同步设施的基础，例如互斥锁和条件变量。要了解其工作原理，请考虑清单 5.1 中简单的自旋互斥锁。

lock()操作是使用 std::memory_order_acquire 顺序的 flag.test_and_set()上的一个循环，并且 unlock()是使用 std::memory_order_release 顺序 flag.clear()的一个调用。第一个线程调用 lock()时，标志最初是 clear 状态，所以第一次调用 test_and_set()将会设置标志，并且返回 false，表示这个线程持有锁，并且结束循环。然后，线程可以自由的修改由互斥锁保护的数据。这时，任何调用 lock()的线程，将会看到已设置的标志，并且将阻塞在 test_and_set()的循环中。

当持有锁的线程完成对保护数据的修改，它会调用 unlock()，这会调用带有 std::memory_order_release 语义的 flag.clear()。它“同步于”随后对 flag.test_and_set()的调用，这个调用来源于另一个线程调用 lock()，并且这个调用具有 std::memory_order_acquire 语义。因为对保护数据的修改，必须“先序于”unlock()的调用，这个修改“先发生于”unlock()，因而“先发生于”随后第二个线程对 lock()的调用 (因为“同步于”关系是在 unlock()和 lock()之间)，并且一旦第二个线程获得锁以后，也“先发生于”从这个线程上对那个数据的任何访问。

尽管其他互斥锁的实现有不同的内存操作，但基本原理是一样的：lock()是一个内部内存位置上的获得操作，并且 unlock 是同样的内存位置上的一个释放操作。

从“同步于”关系角度看，每个在第 2、3、4 章中描述的同步机制都提供了顺序保障，这使你能够使用它们来同步你的数据，并提供顺序保障。以下是这些设施提供的同步关系：

**std::thread**

- thread 构造函数的完成“同步于”在新线程上调用提供的函数或可调用对象。

- 线程的完成“同步于”成功调用 join 的返回，这个调用是在拥有线程的 std::thread 对象上。

std::mutex, std::timed_mutex, std::recursive_mutex, std::recursibe_timed_mutex

- 对给定互斥锁对象的 lock 和 unlock 的调用，以及对 try_lock，try_lock_for 或 try_lock_until 的成功调用，构成单一全序：互斥锁的锁序。
- 在给定的互斥锁对象上调用 unlock“同步于”随后对 lock 的调用或者随后对 try_lock，try_lock_for 或 try_lock_until 的成功调用，这些调用是在这个对象上以互斥锁的锁序进行。
- 对 try_lock，try_lock_for 或 try_lock_until 失败的调用，不参与任何同步关系。

std::shared_mutex , std::shared_timed_mutex

- 在给定互斥锁对象上的 lock、unlock、lock_shared 和 unlock_shared 的调用，以及对 try_lock , try_lock_for , try_lock_until , try_lock_shared , try_lock_shared_for 或 try_lock_shared_until 的成功调用，构成单一全序：互斥锁的锁序。
- 在给定的互斥锁对象上的 unlock 调用“同步于”，随后对 lock 或 shared_lock 的调用，或是对 try_lock , try_lock_for, try_lock_until, try_lock_shared, try_lock_shared_for 或 try_lock_shared_until 的成功调用，这些调用是在这个对象上以互斥锁的锁序进行。
- 对 try_lock，try_lock_for，try_lock_until，try_lock_shared，try_lock_shared_for 或 try_lock_shared_until l 失败的调用，不参与任何同步关系。

std::promise, std::future 和 std::shared_future

- 在给定 std::promise 对象上对 set_value 或 set_exception 的成功调用“同步于”对 wait 或 get，或 wait_for 或 wait_until 调用的成功返回，它们在期望上返回一个 std::future_status::ready 状态，并且这个期望和承诺共享相同的异步状态。
- 给定 std::promise 对象的析构函数，它存储了一个 std::future_error 异常在承诺相关的共享异步状态中，这个析构函数“同步于”对 wait 或 get，或 wait_for 或 wait_until 调用的成功返回，它们在期望上返回一个 std::future_status::ready 状态，并且这个期望和承诺共享相同的异步状态。

std::packaged_task, std::future 和 std::shared_future

- 给定 std::packaged_task 对象函数调用操作符的成功调用“同步于”对 wait 或 get，或 wait_for 或 wait_until 调用的成功返回，它们在期望上返回一个 std::future_status::ready 状态，并且这个期望和打包任务共享相同的异步状态。

- 给定 std::packaged_task 对象的析构函数，它存储了一个 std::future_error 异常在打包任务相关的共享异步状态中，这个析构函数“同步于”对 wait 或 get，或 wait_for 或 wait_until 调用的成功返回，它们在期望上返回一个 std::future_status::ready 状态，并且这个期望和打包任务共享相同的异步状态。

### std::async, std::future 和 std::shared_future

- 使用 std::launch::async 策略调用 std::async 来完成运行任务的线程“同步于”对 wait 或 get，或 wait_for 或 wait_until 调用的成功返回，它们在期望上返回一个 std::future_status::ready 状态，并且这个期望和生成的任务共享相同的异步状态。
- 使用 std::launch::deferred 策略调用 std::async 完成的任务“同步于”对 wait 或 get，或 wait_for 或 wait_until 调用的成功返回，它们在期望上返回一个 std::future_status::ready 状态，并且这个期望和承诺共享相同的异步状态。

### std::experimental::future, std::experimental::shared_future 和延续

- 使异步共享状态变为就绪的事件“同步于”计划在该共享状态上的延续函数的调用。
- 延续函数的完成“同步于”对 wait 或 get，或 wait_for 或 wait_until 调用的成功返回，它们在期望上返回一个 std::future_status::ready 状态，并且这个期望和对 then 调用返回的期望共享相同的异步状态，这个 then 安排了延续，或者“同步于”安排在那个期望上的任何延续的调用。

### std::experimental::latch

- 在给定 std::experimental::latch 实例上的 count_down 或 count_down_and_wait 调用“同步于”每个在该锁存器上对 wait 或 count_down_and_wait 成功调用的完成。

### std::experimental::barrier

- 在给定 std::experimental::barrier 实例上对 arrive_and_wait 或 arrive_and_drop 的调用“同步于”每个随后在该屏障上对 arrive_and_wait 调用的完成。

### std::experimental::flex_barrier

- 在给定 std::experimental::flex_barrier 实例上对 arrive_and_wait 或 arrive_and_drop 的调用“同步于”每个随后在该屏障上对 arrive_and_wait 成功调用的完成。
- 在给定 std::experimental::flex_barrier 实例上对 arrive_and_wait 或 arrive_and_drop 的调用“同步于”在该屏障上随后调用的完成函数。

- 在给定 std::experimental::flex_barrier 实例上从完成函数的返回“同步于”每个在该屏障上对 arrive_and_wait 调用的完成，当完成函数被调用的时候，arrive_and_wait 阻塞在等待这个屏障。

**std::condition_variable 和 std::condition_variable_any**

- 条件变量不提供任何同步关系。它们是对忙-等待循环的优化，所有同步都是相关互斥锁上的操作提供的。

# 总结

在本章中，我介绍了 C++内存模型的底层细节，以及为线程间同步提供基础的原子操作。这包括基本的由 std::atomic<>类模板特化提供的基本原子类型，还有 std::atomic<>主类模板提供的泛型原子接口，以及 std::experimental::atomic_shared_ptr<>模板，这些类型上的操作，以及各种内存顺序选项的复杂细节。

我们还了解了栅栏，以及如何将它们与原子类型上的操作配对来强制一个顺序。最后，我们回到开头，了解了如何使用原子操作在独立线程上的非原子操作之间强制一个顺序，以及高层级设施提供的同步关系。

在下一章中，我们将研究如何使用高层次同步设施和原子操作来设计高效的并发访问容器，我们还将编写算法并行处理数据。

# 第 6 章 设计基于锁的并发数据结构

**本章主要内容**

- 设计并发数据结构的含义
- 设计指南
- 并发数据结构的示例实现

在上一章中我们了解了底层原子操作和内存模型。本章我们先把底层的细节放一放(尽管在第 7 章我们将需要它们)，探讨一下数据结构。

为编程问题选择数据结构可能是整个解决方案的关键部分，并行程序也不例外。如果一个数据结构需要被多个线程访问，要么它完全不可变，因此数据永远不会变化，也就没有必要同步，要么程序必须设计成确保变化在线程间被正确的同步。一种选择是使用独立的互斥锁以及外部加锁来保护数据，比如使用我们在第 3 和第 4 章讨论的技术，另一种选择是设计自身支持并发访问的数据结构。

当设计并发数据结构时，可以使用前面章节提及的应用于多线程程序的基础构件块，比如：互斥锁和条件变量。实际上，你已经看到了几个示例，这些示例展示了如何组合这些构建块来编写对多线程并发访问安全的数据结构。

在本章中，我们将从为并发设计数据结构的一些通用指南开始。然后，我们将使用锁和条件变量的基础构建块，并在转向更复杂的数据结构之前重新讨论这些基本数据结构的设计。在第 7 章中，我们将看到如何回归基础，并使用第 5 章中描述的原子操作来构建无锁的数据结构。

好吧！言归正传，让我们来看一下设计并发数据结构都需要什么。

## 6.1 并发设计的含义

从基本层面上讲，为并发设计数据结构意味着，多个线程可以并发的访问这个数据结构，不管线程执行的是相同还是不同的操作，并且每一个线程都能看到数据结构前后一致的视图。没有数据丢失或者损坏，所有的不变量都被支持，且没有有问题的竞争条件。这样的数据结构，称之为*线程安全*（thread-safe）的数据结构。通常情况下，一个数据结构只对特殊类型的并发访问是安全的。也许有可能让多个线程并发地对数据结构执行一种类型的操作，而另一种操作则需要由单个线程独占访问。或者，如果多个线程正在执行不同的操作，那它们并发访问数据结构可能是安全的，但多个线程执行相同的操作就会有问题。

然而，真正为并发而设计远不止这些：真正的设计意味着要为线程提供并发访问数据结构的机会。从本质上讲，互斥锁提供了互斥：一次只能有一个线程获得互斥锁。互斥锁保护数据结构是通过显式地阻止对它所保护数据的真正并发访问来实现的。

这称为*串行化*(serialization)：线程轮流访问被互斥锁保护的数据。它们必须串行而非并发的访问它。因此，必须仔细考虑数据结构的设计，使得能够真正的并发访问。虽然有些数据结构比其他数据结构具有更大的并发范围，但在所有情况下，其思想都是相同的：受保护的区域越小，串行化的操作就越少，并发的潜力也就越大。

在进行数据结构的设计之前，让我们快速的浏览一下并发设计的指南。

### 6.1.1 设计并发数据结构的指南

之前提过，当设计并发访问的数据结构时，需要考虑两个方面：确保访问安全以及允许真正的并发访问。在第 3 章中，已经介绍了如何让数据结构是线程安全的基础知识：

- 确保没有线程能够看到数据结构的不变量被另一个线程破坏的状态。
- 通过提供完整操作的函数，而非一个个操作步骤的函数来小心避免接口固有的竞争条件。
- 注意数据结构在有异常时的行为，从而确保不变量不会被破坏。
- 通过限制锁的范围以及避免嵌套锁,将死锁的概率降到最低。

在思考这些细节之前，想想要对数据结构的用户施加什么约束也是很重要的；如果线程通过一个特定的函数对数据结构进行访问，其他线程能安全调用哪些函数？

这是一个需要考虑的关键问题，通常，构造函数和析构函数需要互斥地访问数据结构，但是需要由用户确保它们不会在构造函数完成之前或者析构函数开始以后被访问。如果数据结构支持赋值操作，swap()或拷贝构造，作为数据结构的设计者，你需要决定这些操作与其他操作并发调用是否安全，或者它们是否要求用户确保独占访问，尽管大多数用于操作数据结构的函数可以从多个线程并发地调用而没有任何问题。

第二个需要考虑的方面是允许真正的并发访问。在这个方面我没法提供太多的指南。相反，作为一个数据结构的设计者，需要问自己以下问题：

- 是否可以限制锁的作用范围，以允许操作的某些部分在锁外执行？
- 数据结构不同部分能否被不同的互斥锁保护？
- 所有的操作需要同一级别的保护吗？
- 是否可以对数据结构进行简单的修改，以增加并发访问的机会，并且不影响操作语义？

所有这些问题都基于一个思想：如何最小化必须的串行操作，并且使得真实的并发最大化？就数据结构而言，允许多线程并发的只读访问，而修改线程必须互斥访问的情况很普遍。这是通过使用像 std::shared_mutex 这样的结构来支持的。类似地，你很快就会看到，在串行线程尝试执行相同操作的同时，数据结构支持执行不同操作的线程并发地访问也很普遍。

最简单的线程安全数据结构，通常使用互斥锁来保护数据。尽管这样做存在一些问题，但就像你在第 3 章中看到的，确保一次只有一个线程访问数据结构相对比较简单。为

了让你更容易设计线程安全的数据结构，我们将在本章继续研究这种基于锁的数据结构，并将无锁并发数据结构的设计留到第 7 章讨论。

## 6.2 基于锁的并发数据结构

设计基于锁的并发数据结构，都是为了确保在访问数据时锁住正确的互斥锁，并且持有锁的时间最短。对于只有一个互斥锁的数据结构来说，这很困难。你需要确保数据不能在互斥锁的保护之外被访问，并且接口中没有固有的竞争条件，就如第 3 章中看到的那样。如果使用不同的互斥锁来保护数据结构中不同的部分，问题会进一步恶化，如果操作需要锁住多个互斥锁时，现在也可能产生死锁。所以相比单一互斥锁的设计，使用多个互斥锁的数据结构需要更加小心。

在本节中，你将应用 6.1.1 节中的指南来设计一些简单的数据结构，通过使用互斥锁来保护数据。在每个例子中，都是在确保数据结构保持线程安全的前提下，找出更大并发的机会。

我们先来看看第 3 章中栈的实现，它是最简单的数据结构，且只使用了一个互斥锁。那么它是线程安全的吗？它离真正的并发访问有多远呢？

### 6.2.1 使用锁的线程安全栈

下面的清单复制了第 3 章中线程安全的栈。目的是编写一个类似于 std::stack<>的线程安全的数据结构，它支持将数据项推入栈中并再次弹出它们。

清单 6.1 线程安全栈的类定义

```
#include <exception>

struct empty_stack: std::exception
{
  const char* what() const throw();
};

template<typename T>
class threadsafe_stack
{
private:
  std::stack<T> data;
  mutable std::mutex m;

public:
  threadsafe_stack(){}
  threadsafe_stack(const threadsafe_stack& other)
```

```
  {
    std::lock_guard<std::mutex> lock(other.m);
    data=other.data;
  }

  threadsafe_stack& operator=(const threadsafe_stack&) = delete;

  void push(T new_value)
  {
    std::lock_guard<std::mutex> lock(m);
    data.push(std::move(new_value));  // 1
  }

  std::shared_ptr<T> pop()
  {
    std::lock_guard<std::mutex> lock(m);
    if(data.empty()) throw empty_stack();  // 2
    std::shared_ptr<T> const res(
      std::make_shared<T>(std::move(data.top())));  // 3
    data.pop();  // 4
    return res;
  }

  void pop(T& value)
  {
    std::lock_guard<std::mutex> lock(m);
    if(data.empty()) throw empty_stack();
    value=std::move(data.top());  // 5
    data.pop();  // 6
  }

  bool empty() const
  {
    std::lock_guard<std::mutex> lock(m);
    return data.empty();
  }
};
```

我们依次看下每条指南以及它们是如何应用在这里。

首先，如你所见，基本的线程安全是通过使用互斥锁 m 上的锁保护每个成员函数提供的。这将确保在任何时候只有一个线程在访问数据，因此只要每个成员函数保持不变量，就没有线程能看到被破坏的不变量。

其次，在 empty()和 pop()成员函数之间有潜在的竞争条件，不过代码会在 pop()函数持有锁的时候，显式的查询栈是否为空，所以这里的竞争条件没有问题。通过直接返回弹出的数据项作为调用 pop()的一部分，避免了分离的 top()和 pop()成员函数（std::stack<>类似）之间潜在的竞争条件。

然后，栈中也有一些潜在抛异常的地方。对互斥锁上锁可能会抛出异常，但这种情况不仅极其罕见的(这意味着互斥锁有问题，或者缺乏系统资源)，而且它是每个成员函数的第一个操作。由于没有数据被修改，所以是安全的。解锁互斥锁不会失败，所以总是安全的，并且使用 std::lock_guard<>确保了互斥锁不会一直处于上锁的状态。

对 data.push()①的调用可能会抛出一个异常，只要拷贝/移动数据值抛出一个异常，或者可分配的内存不足。不管是哪种情况，std::stack<>都能保证是安全的，所以也没有问题。

在第一个重载的 pop()中，代码本身可能会抛出一个 empty_stack 的异常②，但由于什么都没有修改，所以是安全的。创建 res③可能会抛出一个异常，有几个方面的原因：对 std::make_shared 的调用，可能因为无法为新对象以及引用计数需要的内部数据分配出足够的内存而抛出异常，或者在拷贝/移动到新分配内存的时候，返回的数据项的拷贝构造或移动构造函数可能抛出异常。两种情况下，C++运行库和标准库会确保没有内存泄露，并且新创建的对象(如果有的话)会被正确的销毁。因为仍然没有对栈进行任何修改，所以也不会有问题。调用 data.pop()④保证不会抛出异常，随后是返回结果，所以这个重载的 pop()函数是异常安全的。

第二个重载的 pop()类似，不过这次是在拷贝赋值或移动赋值时可能抛出异常⑤，而不是在构造新对象和 std::shared_ptr 实例时。再次，直到调用 data.pop()⑥（pop 仍然保证不会抛出异常）前，没有修改数据结构，所以这个函数也是异常安全的。

最后，empty()不会修改任何数据，所以也是异常安全的。

这里有几个可能导致死锁的机会，因为你在持有锁的时候调用了用户代码：数据项上的拷贝构造或移动构造(①，③)和拷贝赋值或移动赋值操作⑤，也可能是用户自定义的 new 操作符。如果这些函数或者调用了栈上的成员函数（而栈正在插入或移除数据项），或者需要任何类型的锁，而在调用栈成员函数时又持有了另一把锁，那么就有可能出现死锁。但明智的做法是要求栈的用户负责确保这一点；你不能期望在不拷贝或不为它分配内存的情况下将数据项添加到栈或从栈中删除。

由于所有成员函数都使用 std::lock_guard<>保护数据，所以不管多少线程调用栈成员函数都是安全的。唯一不安全的成员函数是构造函数和析构函数，但这不是问题；对象只能被构造一次，也只能被销毁一次。不管并发与否，调用一个不完全构造的对象或是部分销毁的对象的成员函数永远都不可取。因此，用户必须确保其他线程直到栈完全构造才能访问它，并且必须确保在栈对象销毁前，所有线程都已经停止访问栈。

尽管多个线程并发调用成员函数是安全的，但由于使用了锁，每次只有一个线程在栈数据结构中做一些工作。线程的串行化会潜在的限制应用程序的性能，因为在栈上有严重的争用：当一个线程在等待锁时，它没有做任何有用的工作。同样，栈也没有提供什么方

法等待添加一个数据项，所以如果线程需要等待时，它必须周期性地调用 empty()或 pop()，并且捕获 empty_stack 异常。如果这种场景是必须的，那这种栈实现就是个糟糕的选择，因为等待线程要么消耗宝贵的资源去检查数据，要么要求用户编写外部等待和通知的代码(例如，使用条件变量)，这就使内部上锁没有必要，因而造成浪费。第 4 章中的队列展示了一种使用数据结构内部的条件变量将这种等待合并到数据结构本身的方法，接下来我们看一下这个。

## 6.2.2 使用锁和条件变量的线程安全队列

清单 6.2 复制了第 4 章中的线程安全队列，就像栈是仿照 std::stack<>一样，这个队列也是仿照了 std::queue<>。再次，接口不同于标准容器适配器，因为实现的数据结构需要支持多线程并发访问。

清单 6.2 使用条件变量实现的线程安全队列

```
template<typename T>
class threadsafe_queue
{
private:
  mutable std::mutex mut;
  std::queue<T> data_queue;
  std::condition_variable data_cond;

public:
  threadsafe_queue()
  {}

  void push(T data)
  {
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(std::move(data));
    data_cond.notify_one();  // 1
  }

  void wait_and_pop(T& value)  // 2
  {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    value=std::move(data_queue.front());
    data_queue.pop();
  }

  std::shared_ptr<T> wait_and_pop()  // 3
  {
```

```
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    std::shared_ptr<T> res(
      std::make_shared<T>(std::move(data_queue.front()))); // 4
    data_queue.pop();
    return res;
  }

  bool try_pop(T& value)
  {
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
      return false;
    value=std::move(data_queue.front());
    data_queue.pop();
    return true;
  }

  std::shared_ptr<T> try_pop()
  {
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
      return std::shared_ptr<T>();  // 5
    std::shared_ptr<T> res(
      std::make_shared<T>(std::move(data_queue.front())));
    data_queue.pop();
    return res;
  }

  bool empty() const
  {
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
  }
};
```

除了在 push()①中调用 data_cond.notify_one()，以及 wait_and_pop()②③外，清单 6.2 中队列的实现与 6.1 清单中的栈类似。两个重载的 try_pop()几乎和清单 6.1 中一样，只是在队列为空时不抛异常，取而代之返回一个 bool 值表示是否检索到值或者一个 NULL 指针（对应返回指针的重载版本）如果没有值可以检索的话。这也是实现栈的一个有效方式。如果排除 wait_and_pop()函数，对栈的分析在这里也同样适用。

新的 wait_and_pop()函数解决了在栈中碰到的等待队列条目的问题；比起持续调用 empty()，等待线程调用 wait_and_pop()函数并且数据结构使用条件变量来处理等待。对 data_cond.wait()的调用，直到队列中至少有一个元素时才会返回，所以不用担心会出现

空队列的情况，并且数据仍然被互斥锁保护。因此，这些函数不会添加任何新的竞争条件或死锁的可能性，并且将支持不变量。

在异常安全性方面有一个细微的变化，当一个条目被推入队列时，如果有多个线程在等待，那么只有一个线程会被 data_cond.notify_one()唤醒。但是，如果这个线程在 wait_and_pop()中抛出一个异常，比如当构造新的 std::shared_ptr<>对象④时，那么没有其他线程被唤醒。如果这种情况不可接受，那么调用可以替换成 data_cond.notify_all()，它将唤醒所有的工作线程，代价就是大多线程发现队列依旧是空时，重新进入休眠状态。第二种替代方案是，有异常抛出的时，让 wait_and_pop()函数调用 notify_one()，从而让另一个线程可以去尝试检索存储的值。第三种替代方案是，将 std::shared_ptr<>的初始化过程移到 push()中，并且存储 std::shared_ptr<>实例，而不是直接使用数据值。将 std::shared_ptr<>从内部 std::queue<>中拷出不会抛出异常，这样 wait_and_pop()又是安全的了。下面的程序清单，就是基于这种思路修改的。

清单 6.3 持有 std::shared_ptr<>实例的线程安全队列

```
template<typename T>
class threadsafe_queue
{
private:
  mutable std::mutex mut;
  std::queue<std::shared_ptr<T> > data_queue;
  std::condition_variable data_cond;

public:
  threadsafe_queue()
  {}

  void wait_and_pop(T& value)
  {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    value=std::move(*data_queue.front());  // 1
    data_queue.pop();
  }

  bool try_pop(T& value)
  {
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
      return false;
    value=std::move(*data_queue.front());  // 2
    data_queue.pop();
    return true;
  }
```

```
  std::shared_ptr<T> wait_and_pop()
  {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    std::shared_ptr<T> res=data_queue.front();  // 3
    data_queue.pop();
    return res;
  }

  std::shared_ptr<T> try_pop()
  {
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
      return std::shared_ptr<T>();
    std::shared_ptr<T> res=data_queue.front();  // 4
    data_queue.pop();
    return res;
  }

  void push(T new_value)
  {
    std::shared_ptr<T> data(
    std::make_shared<T>(std::move(new_value)));  // 5
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }

  bool empty() const
  {
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
  }
};
```

通过 std::shared_ptr<>持有数据的影响比较直接：通过引用变量来接收新值的 pop 函数现在必须对存储的指针解引用①②；并且，在返回给调用者前，返回 std::shared_ptr<>实例的 pop 函数可以从队列中检索它③④。

通过 std::shared_ptr<>持有数据还有个好处：在 push()⑤中分配新实例可以在锁外面完成，而在清单 6.2 中，只能在 pop()持有锁时完成。因为内存分配是个典型的代价高昂的操作，这有利于队列的性能，因为它减少了持有互斥锁的时间，并允许其他线程同时在队列上执行操作。

如同栈示例，使用互斥锁来保护整个数据结构限制了该队列的并发支持；尽管在不同的成员函数中，队列上可能阻塞多个线程，但一次只能有一个线程开展工作。部分限制来自于实现中使用了 std::queue<>；通过使用标准容器，你现在可以决定数据项是否受保护。通过控制数据结构的实现细节，你可以提供更细粒度的锁从而实现更高级别的并发。

## 6.2.3 使用细粒度锁和条件变量的线程安全队列

在清单 6.2 和 6.3 中，有一个受保护的数据项（data_queue）和一个互斥锁。为了使用细粒度锁，需要查看队列内部的组成部分，并将一个互斥锁与每个不同的数据项关联起来。

最简单的队列是单链表，如图 6.1 所示。队列包含一个头指针，指向链表中的第一个项，然后每一项指向下一项。从队列中删除数据项，是用指向下一项的指针替换头指针，然后将之前头指针的数据返回。

数据项从队列的另一端添加到队列。为了做到这点，队列还有一个 tail 指针，它指向链表中的最后一项。新节点的添加是通过改变最后一项的 next 指针，让它指向新的节点，然后更新 tail 指针指向这个新的数据项。当链表为空时，头/尾指针都为 NULL。

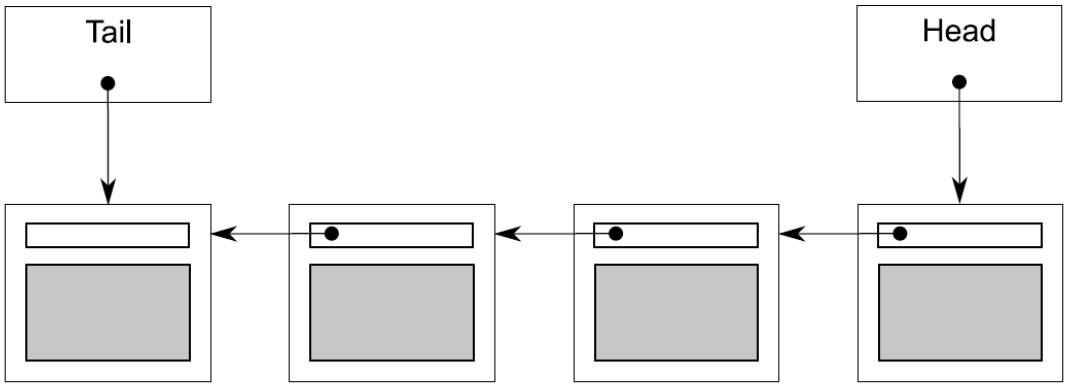


图 6.1 用单链表表示的队列

下面的清单显示了这个队列的简单实现，它基于清单 6.2 中队列接口的简化版本；只有一个 try_pop()函数，没有 wait_and_pop()，因为这个队列只支持单线程使用。

清单 6.4 简单的单线程队列实现

```
template<typename T>
class queue
{
private:
  struct node
  {
```

```
    T data;
    std::unique_ptr<node> next;

    node(T data_):
    data(std::move(data_))
    {}
  };

  std::unique_ptr<node> head;  // 1
  node* tail;  // 2

public:
  queue()
  {}
  queue(const queue& other)=delete;
  queue& operator=(const queue& other)=delete;

  std::shared_ptr<T> try_pop()
  {
    if(!head)
    {
      return std::shared_ptr<T>();
    }
    std::shared_ptr<T> const res(
      std::make_shared<T>(std::move(head->data)));
    std::unique_ptr<node> const old_head=std::move(head);
    head=std::move(old_head->next);  // 3
    return res;
  }

  void push(T new_value)
  {
    std::unique_ptr<node> p(new node(std::move(new_value)));
    node* const new_tail=p.get();
    if(tail)
    {
      tail->next=std::move(p);  // 4
    }
    else
    {
      head=std::move(p);  // 5
    }
    tail=new_tail;  // 6
  }
```

```
};
```

首先，注意清单 6.4 中使用了 std::unique_ptr<node>来管理节点，因为这能保证当不再需要它们的时候，它们（以及它们引用的数据）会自动删除，而不必使用显式的 delete。这个所有权链的管理从 head 开始，tail 是指向最后一个节点的裸指针，因为它需要引用 std::unique_ptr<node>已经拥有的节点。

虽然这个实现在单线程环境工作的很好，但当在多线程下尝试使用细粒度锁时，有几个事情会带来麻烦。因为在给定的实现中有两个数据项(head①和 tail②)；原则上可以使用两个互斥锁来分别保护头和尾指针，但这样做会有几个问题。

最明显的问题就是 push()可能同时修改 head⑤和 tail⑥，所以它必须锁住两个互斥锁。尽管很不幸，但这倒不算是太大的问题，因为锁住两个互斥锁是可能的。关键的问题是 push()和 pop()都能访问 next 指针指向的节点：push()更新 tail->next④，然后 try_pop()读取 head->next③。如果队列中只有一个元素，那么 head==tail，所以 head->next 和 tail->next 是同一个对象，并且这个对象需要保护。由于不同时读取 head 和 tail 的话，没法区分它们是否是同一个对象，你现在必须在 push()和 try_pop()中锁住同一个锁，所以，也没比以前好多少。那有什么办法摆脱这个困境吗？

**通过分离数据使并发成为可能**

你可以通过预先分配一个没有数据的虚拟节点来解决这个问题，以确保队列中至少有一个节点将头部访问的节点与尾部访问的节点分开。对一个空队列，head 和 tail 都指向虚拟节点，而非 NULL 指针。这很好，因为如果队列为空，try_pop()就不会访问 head->next 了。如果添加一个节点到队列中(所以有一个真实节点)，那 head 和 tail 现在会指向不同的节点，所以在 head->next 和 tail->next 上不会有竞争。但缺点是，为了允许虚拟节点的存在，需要增加额外的一层间接性——通过指针来存储数据。下面的清单展示了这个方案。

清单 6.5 带有虚拟节点的简单队列

```
template<typename T>
class queue
{
private:
  struct node
  {
    std::shared_ptr<T> data;  // 1
    std::unique_ptr<node> next;
  };

  std::unique_ptr<node> head;
  node* tail;

public:
```

```
  queue():
    head(new node),tail(head.get())  // 2
  {}
  queue(const queue& other)=delete;
  queue& operator=(const queue& other)=delete;

  std::shared_ptr<T> try_pop()
  {
    if(head.get()==tail)  // 3
    {
      return std::shared_ptr<T>();
    }
    std::shared_ptr<T> const res(head->data);  // 4
    std::unique_ptr<node> old_head=std::move(head);
    head=std::move(old_head->next);  // 5
    return res;  // 6
  }

  void push(T new_value)
  {
    std::shared_ptr<T> new_data(
      std::make_shared<T>(std::move(new_value)));  // 7
    std::unique_ptr<node> p(new node);  //8
    tail->data=new_data;  // 9
    node* const new_tail=p.get();
    tail->next=std::move(p);
    tail=new_tail;
  }
};
```

try_pop()的改动非常小。首先，比较 head 和 tail③，而不是检查 head 是否为 NULL，因为虚拟节点意味着 head 不可能是 NULL。由于 head 是一个 std::unique_ptr<node>对象，所以需要使用 head.get()来做比较。其次，因为 node 现在通过指针存储数据①，你可以直接获取指针④，而不必构造一个 T 类型的新实例。最大的改动在 push()函数：首先，必须先在堆上创建一个 T 类型的实例，并且让一个 std::shared_ptr<>拥有该对象⑦(注意，节点使用 std::make_shared 是为了避免为引用计数增加二次内存分配的开销，译注：make_shared 一次性把需要的内存分配好，而手工做的话会有两次内存分配，一次是显式调用 new，为 T 类型实例分配内存，一次是 share_ptr 内部为引用计数再分配一次内存)。创建的新节点就成为了新的虚拟节点，所以不需要把 new_value 提供给构造函数⑧。相反，把新分配的 new_value 拷贝赋给老的虚拟节点⑨。最后，为了拥有一个虚拟节点，需要在构造函数中创建它②。

到目前为止，我敢肯定你想知道这些更改给你带来了什么，以及它们如何使得队列线程安全。好吧，现在的 push()只需要访问 tail，不用访问 head，这就是个改进。try_pop()需要访问 head 和 tail，但 tail 只需在最初比较的时候访问，所以锁是短暂

的。重大的收益在于虚拟节点的引入，使得 try_pop()和 push()永远不用对同一节点进行操作，所以就不再需要一个总的互斥锁了。你可以给 head 和 tail 分别分配一个互斥锁。那锁放哪呢？

你的目标是为了最大程度的并发，所以上锁的时间越少越好。push()很简单：访问 tail 期间都需要互斥锁被锁住，也就是在新分配节点后⑧，对当前尾节点进行赋值⑨之前上锁。锁需要持有到函数结束。

try_pop()就没那么简单了。首先，需要使用互斥锁锁住 head，一直到不再访问 head。这个互斥锁决定了哪一个线程进行弹出操作，所以你会第一时间锁上它。一旦 head 改变⑤，就可以解锁互斥锁；在返回结果时，互斥锁就不需要上锁了⑥。这使得对 tail 的访问需要锁住 tail 互斥锁。因为只需要访问 tail 一次，且只有在读取时才需要互斥锁。所以最好的做法是把它包装到函数中。实际上，因为需要 head 互斥锁被锁住的代码只是成员的一个子集，把它包装到函数中也会更清晰。最终代码如下。

清单 6.6 细粒度锁的线程安全队列

```
template<typename T>
class threadsafe_queue
{
private:
  struct node
  {
    std::shared_ptr<T> data;
    std::unique_ptr<node> next;
  };

  std::mutex head_mutex;
  std::unique_ptr<node> head;
  std::mutex tail_mutex;
  node* tail;

  node* get_tail()
  {
    std::lock_guard<std::mutex> tail_lock(tail_mutex);
    return tail;
  }

  std::unique_ptr<node> pop_head()
  {
    std::lock_guard<std::mutex> head_lock(head_mutex);
    if(head.get()==get_tail())
    {
      return nullptr;
    }
```

```
    std::unique_ptr<node> old_head=std::move(head);
    head=std::move(old_head->next);
    return old_head;
  }

public:
  threadsafe_queue():
  head(new node),tail(head.get())
  {}
  threadsafe_queue(const threadsafe_queue& other)=delete;
  threadsafe_queue& operator=(const threadsafe_queue& other)=delete;

  std::shared_ptr<T> try_pop()
  {
     std::unique_ptr<node> old_head=pop_head();
     return old_head?old_head->data:std::shared_ptr<T>();
  }

  void push(T new_value)
  {
    std::shared_ptr<T> new_data(
      std::make_shared<T>(std::move(new_value)));
    std::unique_ptr<node> p(new node);
    node* const new_tail=p.get();
    std::lock_guard<std::mutex> tail_lock(tail_mutex);
    tail->data=new_data;
    tail->next=std::move(p);
    tail=new_tail;
  }
};
```

让我们以批判的眼光来考察这段代码，并考虑 6.1.1 节中列出的指南。在你寻找被破坏的不变量前，需要确定有哪些不变量：

- tail->next==nullptr
- tail->data==nullptr
- head==tail 意味着空链表
- 单元素链表有 head->next==tail
- 对链表中的每一个节点 x，x!=tail 的话，x->data 指向一个 T 类型的实例，并且 x->next 指向链表中下一个节点。x->next==tail 意味着 x 就是链表中最后一个节点
- 顺着 head 的 next 节点捋下去，最终会找到 tail

就 push()本身而言是比较简单的：对数据结构的修改都被 tail_mutex 保护，它们支持了不变量，因为新的尾节点是一个空节点，并且旧的尾节点的 data 和 next 都设置好了，这个旧尾节点现在是链表中最后一个真实节点。

有趣的部分在 try_pop()，不仅需要对 tail_mutex 上锁来保护对 tail 的读取；还要保证在从 head 读取数据时，不会产生数据竞争。如果没有这些互斥锁，很可能一个线程调用 try_pop()的同时，另一个线程调用 push()，然后操作就没有一个确定的顺序。即便每一个成员函数都持有一个互斥锁，但他们持有的是不同的互斥锁，并且潜在地访问相同的数据；毕竟队列中的所有数据来源，都是通过调用 push()得到。由于线程可能会乱序地访问相同数据，就会有数据竞争（如你在第 5 章看到的那样），以及未定义的行为。好在 get_tail()中的 tail_mutex 解决了所有的问题。因为调用 get_tail()将会锁住和 push()中相同的锁，这样在两个调用间就有了确定的顺序。要不就是 get_tail()在 push()之前被调用，线程可以看到 tail 的旧值，要不就是在 push()之后调用，线程就能看到 tail 的新值，并且新值会附加到之前的 tail 值上。

对 get_tail()的调用发生在 head_mutex 锁内也是很重要。如果不这样，调用 pop_head()就会卡在 get_tail()和 head_mutex 锁中间，因为其他线程调用 try_pop()(从而调用 pop_head())时，都需要先获取锁，然后阻止你初始线程往前执行：

```
std::unique_ptr<node> pop_head() // 这是个有问题的实现
{
  node* const old_tail=get_tail();  // 1 在 head_mutex 锁范围外获取旧尾节点的值
  std::lock_guard<std::mutex> head_lock(head_mutex);

  if(head.get()==old_tail)  // 2
  {
    return nullptr;
  }
  std::unique_ptr<node> old_head=std::move(head);
  head=std::move(old_head->next);  // 3
  return old_head;
}
```

在这个有问题的场景下，在锁的外面调用 get_tail()①，初始线程能获得 head_mutex 的锁时，也许会发现 head 和 tail 都发生了改变。并且，不光返回的 tail 节点不再是尾节点，甚至可能它都不在链表中了。这意味着比较 head 和 old_tail②会失败，即使 head 是最后一个节点。因此，当更新 head③时，可能会将 head 越过 tail 并离开链表的尾端，这样的话数据结构就遭到了破坏。正确实现中(清单 6.6)，需要保证在 head_mutex 保护的范围内调用 get_tail()。就能保证没有其他线程对 head 进行修改，并且 tail 只会往后移动(当有新节点添加进来时)，这样就很安全了。head 永远无法穿过 get_tail()返回的值，所以不变量被支持。

一旦 pop_head()通过更新 head 把节点从队列中删除时，互斥锁就被解锁，并且 try_pop()可以提取数据，并在有数据的时候删除一个节点(若没有数据，则返回 std::shared_ptr<>的 NULL 实例)，因为只有一个线程可以访问这个节点，所以这个操作是安全的。

接下来，外部接口是清单 6.2 中的一个子集，所以应用同样的分析：接口中没有固有的竞争条件。

异常更有趣。因为你已经改变了数据的分配模式，异常现在可能来自不同的地方。try_pop()唯一会产生异常的操作是锁住互斥锁，并且在获得锁之前数据不会被修改。因此，try_pop()是异常安全的。另一方面，push()在堆上分配一个新的 T 实例，以及一个新的 node 实例，这里都可能会抛出异常。但是，所有新分配的对象都赋给了智能指针，当异常发生时就会被释放掉。一旦获得锁，push()中剩下的操作就不会抛出异常，所以 push()也是异常安全的。

因为没有修改任何接口，所以没有新的外部机会导致死锁。在实现内部也不会有死锁；唯一需要获取两个锁的是 pop_head()，这个函数都是先获取 head_mutex 然后 tail_mutex，所以永远不会死锁。

剩下的问题是关于并发的可能性。与清单 6.2 相比，这个数据结构具有更大的并发范围，因为锁的粒度更细，并且更多操作是在锁外完成的。比如，push()中新节点和新数据项的分配都不需要持有锁。这意味着多个线程可以并发的分配新节点和新数据项。虽然一次只有一个线程能将它的新节点添加到链表中，但完成这个工作的代码只是几个简单的指针赋值，所以相较于基于 std::queue<>的实现整个内存分配操作都会持有锁，这里持有锁的时间很短。

同样，try_pop()持有 tail_mutex 来保护对 tail 读取的时间也很短。因此，try_pop()调用几乎完全可以和 push()调用一起并发发生。同样持有 head_mutex 的时候执行的操作也非常的少；昂贵的 delete（在 node 指针的析构函数中）在锁的外部。这将增加并发调用 try_pop()的次数；虽然一次只有一个线程调用 pop_head()，但是多个线程可以删除它们的旧节点以及安全地返回数据。

**等待数据项弹出**

好了，所以清单 6.6 提供了一个使用细粒度锁的线程安全队列，但它只支持 try_pop()(且只有一个重载)。那清单 6.2 中那个趁手的 wait_and_pop()呢？能通过细粒度锁实现相同的接口吗？

答案是肯定的，关键是怎么做。修改 push()比较简单：只需要在函数末尾添加 data_cond.notify_one()函数的调用即可，像清单 6.2 中那样。事情也没那么简单；之所以使用细粒度锁，是因为你想要尽可能大的并发。如果你让互斥锁在锁住的情况下调用 notify_one()(如清单 6.2 所示)，然后如果被通知的线程在互斥锁解锁前醒来，那它将必须等待互斥锁。另一方面，在调用 notify_one()前解锁互斥锁，那等待线程在它醒来时互斥锁就是可用的(假设没有其他线程率先锁住它)。这是个小小的改进，但在某些情况下却很重要。

wait_and_pop()有点复杂，因为需要确定在哪等待，谓词是什么？以及要锁住哪个互斥锁？需要等待的条件是“队列非空”，也就是 head!=tail。但这么写的话，就需要同时锁住 head_mutex 和 tail_mutex，不过在清单 6.6 中已经明确了，只需要锁住 tail_mutex 来保护对 tail 的读取，比较操作本身不需要锁住 tail_mutex，所以这里也可以应用相同

的逻辑。如果你让谓词变成 head!=get_tail()，你就只需要持有 head_mutex，然后你可以使用 head_mutex 对 data_cond.wait()的调用进行保护。一旦加入了等待逻辑，实现就和 try_pop()一样了。

对于 try_pop()和 wait_and_pop()的重载版本都需要仔细斟酌。如果你把从 old_head 检索到的 std::shared_ptr<>返回值替换为 value 参数的拷贝赋值，将有一个潜在的异常安全问题。此时，数据项已从队列中删除，并且互斥锁已解锁；剩下的就是将数据返回给调用者。但是如果拷贝赋值抛出一个异常(很有可能)，数据项就会丢失，因为它没法返回到队列的相同位置。

如果用于模板参数的实际类型 T 有一个 noexcept 的移动赋值操作或交换操作时，你可以使用它，但你肯定希望有更通用的解决方案适用于任意类型的 T。这种情况下，在节点从链表中删除前，你需要把潜在会抛出异常的操作移到到锁保护的区域内。这意味着你需要一个额外的 pop_head()重载版本，在改动链表前把存储的值检索出来。

相比之下，empty()就简单了：只需要锁住 head_mutex，并且检查 head==get_tail()(详见清单 6.10)就可以了。最终的代码，在清单 6.7，6.8，6.9 和 6.10 中。

清单 6.7 使用锁和等待的线程安全队列：内部构件与接口

```
template<typename T>
class threadsafe_queue
{
private:
  struct node
  {
    std::shared_ptr<T> data;
    std::unique_ptr<node> next;
  };

  std::mutex head_mutex;
  std::unique_ptr<node> head;
  std::mutex tail_mutex;
  node* tail;
  std::condition_variable data_cond;

public:
  threadsafe_queue():
    head(new node),tail(head.get())
  {}
  threadsafe_queue(const threadsafe_queue& other)=delete;
  threadsafe_queue& operator=(const threadsafe_queue& other)=delete;

  std::shared_ptr<T> try_pop();
```

```
  bool try_pop(T& value);
  std::shared_ptr<T> wait_and_pop();
  void wait_and_pop(T& value);
  void push(T new_value);
  bool empty();
};
```

向队列中添加新节点相当简单——实现（在下面的清单展示）与前面展示的差不多。

清单 6.8 使用锁和等待的线程安全队列：推入新节点

```
template<typename T>
void threadsafe_queue<T>::push(T new_value)
{
  std::shared_ptr<T> new_data(
  std::make_shared<T>(std::move(new_value)));
  std::unique_ptr<node> p(new node);
  {
    std::lock_guard<std::mutex> tail_lock(tail_mutex);
    tail->data=new_data;
    node* const new_tail=p.get();
    tail->next=std::move(p);
    tail=new_tail;
  }
  data_cond.notify_one();
}
```

之前已经提到过，所有的复杂性都落在 pop 端，它采用了很多辅助性函数来简化问题。下一个清单中展示了 wait_and_pop()的实现，以及相关的辅助函数。

清单 6.9 使用锁和等待的线程安全队列：wait_and_pop()

```
template<typename T>
class threadsafe_queue
{
private:
  node* get_tail()
  {
    std::lock_guard<std::mutex> tail_lock(tail_mutex);
    return tail;
  }

  std::unique_ptr<node> pop_head()  // 1
  {
    std::unique_ptr<node> old_head=std::move(head);
```

```
    head=std::move(old_head->next);
    return old_head;
  }

  std::unique_lock<std::mutex> wait_for_data()  // 2
  {
    std::unique_lock<std::mutex> head_lock(head_mutex);
    data_cond.wait(head_lock,[&]{return head.get()!=get_tail();});
    return std::move(head_lock);  // 3
  }

  std::unique_ptr<node> wait_pop_head()
  {
    std::unique_lock<std::mutex> head_lock(wait_for_data());  // 4
    return pop_head();
  }

  std::unique_ptr<node> wait_pop_head(T& value)
  {
    std::unique_lock<std::mutex> head_lock(wait_for_data());  // 5
    value=std::move(*head->data);
    return pop_head();
  }
public:
  std::shared_ptr<T> wait_and_pop()
  {
    std::unique_ptr<node> const old_head=wait_pop_head();
    return old_head->data;
  }

  void wait_and_pop(T& value)
  {
    std::unique_ptr<node> const old_head=wait_pop_head(value);
  }
};
```

清单 6.9 所示的 pop 端实现使用了一些辅助函数来简化代码并消除重复，例如 pop_head()①，它修改链表来删除了头部项，以及 wait_for_data()②，它等待队列中有数据可以弹出。wait_for_data()尤其值得注意，因为它不仅使用 lambda 函数作为谓词等待条件变量，而且还会将锁的实例返回给调用者③。这是为了确保当数据被相关的 wait_pop_head()重载函数④⑤修改时持有同一把锁。pop_head()也被 try_pop()代码复用，如下所示。

清单 6.10 使用锁和等待的线程安全队列：try_pop()和 empty()

```cpp
template<typename T>
class threadsafe_queue
{
private:
  std::unique_ptr<node> try_pop_head()
  {
    std::lock_guard<std::mutex> head_lock(head_mutex);
    if(head.get()==get_tail())
    {
      return std::unique_ptr<node>();
    }
    return pop_head();
  }

  std::unique_ptr<node> try_pop_head(T& value)
  {
    std::lock_guard<std::mutex> head_lock(head_mutex);
    if(head.get()==get_tail())
    {
      return std::unique_ptr<node>();
    }
    value=std::move(*head->data);
    return pop_head();
  }

public:
  std::shared_ptr<T> try_pop()
  {
    std::unique_ptr<node> old_head=try_pop_head();
    return old_head?old_head->data:std::shared_ptr<T>();
  }

  bool try_pop(T& value)
  {
    std::unique_ptr<node> const old_head=try_pop_head(value);
    return old_head;
  }

  bool empty()
  {
    std::lock_guard<std::mutex> head_lock(head_mutex);
    return (head.get()==get_tail());
  }
};
```

这个队列的实现将作为第 7 章无锁队列的基础。它是一个无界队列；只要内存够用，即使没有元素被删除，线程也可以不断往队列中添加新的数据项。与之相对的是有界队列，有界队列在创建的时候最大长度就已经固定了。一旦有界队列已满，尝试往队列中添加更多元素要么会失败要么会阻塞到有一个元素从队列中弹出而腾出空间。当在线程间基于任务划分工作的时候，有界队列有助于工作的均匀分布(参见第 8 章)，这防止了填充队列的线程运行的遥遥领先于从队列读取数据项的线程。

这里实现的无界队列可以很容易扩展成在 push()中等待条件变量的限长队列，不同于等待队列中有数据项 (就像 pop()所做的)，你需要等待队列中数据项个数小于最大数。关于有界队列的进一步讨论超出了本书的范围；现在，让我们从队列转移到更复杂的数据结构。

# 6.3 设计更复杂的基于锁的数据结构

栈和队列都很简单：接口非常有限，它们紧盯在一个特定的用途上。但并不是所有的数据结构都那么简单；大多数数据结构支持各种操作。原则上，这可能会带来更大的并发机会，但也会使保护数据的任务变得更加困难，因为需要考虑多种访问模式。在为并发访问设计这些数据结构时，可以执行的各种操作的确切性质非常重要。

为了了解其中涉及的问题，让我们看看查找表的设计。

## 6.3.1 使用锁实现一个线程安全的查找表

一个查找表或字典把一种类型的值(键类型)关联到另一种类型的值 (映射的类型)。一般情况下，这种结构允许代码查询与给定键关联的数据。在 C++标准库，这种设施通过关联容器提供：std::map<>，std::multimap<>，std::unordered_map<>和 std::unordered_multimap<>。

查找表的使用模式不同于栈和队列。尽管栈和队列上几乎每个操作都会以某种方式对它进行修改，无论是添加还是删除一个元素，但查找表很少被修改。清单 3.13 中的简单 DNS 缓存就是这样一个例子，只不过相较于 std::map<>削减了很多接口。正如你在栈和队列中看到的，当要从多个线程并发地访问数据结构时，标准容器的接口并不适合，因为接口设计中存在固有的竞争条件，因此需要删减和修改它们。

从并发视角看，std::map<>接口最大的问题是迭代器。尽管，当其他线程访问(或修改)容器时，是有可能让迭代器提供对容器的安全访问，但这是个棘手的议题。正确处理迭代器需要处理一些问题，比如另一个线程正在删除迭代器所引用的元素，这可能相当复杂。对线程安全查找表的第一次接口削减，会跳过迭代器。考虑到 std::map<>的接口(以及标准库中的其他关联容器)重度基于迭代器，有必要将它们放一边，并从头开始设计接口。

查找表只有几个基本操作：

- 添加一个新的键/值对
- 修改给定键对应的值
- 删除一个键及其关联的值
- 通过给定键，获取对应的值，如果有的话

还有一些容器范围的操作可能很有用，比如检查容器是否为空、完整键列表的快照，或者键/值对的完整集合的快照。

如果坚持简单的线程安全指南，例如：不要返回一个引用，并且用一个简单的互斥锁对每一个成员函数进行上锁，所有这些都是安全的；他们要么在另一个线程的修改前要么在之后。最有可能的竞争条件在当一个新的键/值对正在被添加时；如果两个线程都添加一个新值，只有一个会先执行，然后后面那个就失败了。一种方式是合并添加和修改到一个成员函数，就像清单 3.13 对 DNS 缓存所做的那样。

从接口视角看，唯一有趣的是获取关联数据时的*如果有*(if any)部分。一种选择是在键不存在的时候允许用户提供一个“默认”结果作为返回值。

```
mapped_type get_value(key_type const& key, mapped_type default_value);
```

这种情况下，当 default_value 没有显式给出时，默认构造出的 mapped_type 实例将被使用。也可以扩展成返回一个 std::pair<mapped_type, bool>来代替 mapped_type 实例，其中 bool 代表键是否存在关联的值。另一个选择是返回一个智能指针引用关联的值，当指针的值是 NULL 时，表示没有值返回。

一旦接口已经明确，(假设没有接口竞争条件)那么可以通过使用单一的互斥锁以及在成员函数周围使用一个简单的锁来保护底层数据结构，从而保证线程安全性。但这将浪费读取和修改数据结构的独立函数所提供的并发可能性。一种选择是使用一个支持多个读线程或单个写线程的互斥锁，比如清单 3.13 中使用的 std::shared_mutex。虽然，这会提高并发访问，但是只有一个线程能修改数据结构。理想情况下，你想要做的更好。

**为细粒度上锁设计一个映射数据结构**

与 6.2.3 节中讨论的队列一样，为了允许细粒度上锁，你需要仔细查看数据结构的细节，而不是包装已有的容器，如 std::map<>。有三种常用方法实现像你的查找表这样的关联容器：

- 二叉树，比如：红黑树
- 有序数组
- 哈希表

二叉树并没有为扩展并发机会提供太多空间；每一个查找或者修改操作都需要从访问根节点开始，因此，根节点需要上锁。虽然随着访问线程沿着树向下移动，这个锁可以释放，但相比横跨整个数据结构的单一锁，这并没有好多少。

有序数组更糟，因为你无法提前知道给定的数据值应该放哪，所以你需要一把锁用于整个数组。

现在就剩哈希表了。假设有固定数量的桶，一个键属于哪个桶纯粹由键的属性以及哈希函数决定。这就意味着每个桶都可以有一个独立的锁。当再次使用互斥锁支持多读者或单写者时，就能将并发访问的可能性增加 N 倍，这里 N 是桶的数量。缺点是你需要为键选择一个好的哈希函数。C++标准库提供了 std::hash<>模板，可以用于这个用途。它已经针对基本数据类型比如 int，以及通用库类型 std::string，做了特化，并且用户可以很容易对其他类型进行特化。如果遵循标准的无序容器，并将用于执行哈希的函数对象的类型作为模板参数，用户就可以选择是否用他们的键类型对 std::hash<>进行特化，还是提供一个单独的哈希函数。

让我们来看一些代码。一个线程安全的查找表应该长什么样？下面提供了一种可能实现。

清单 6.11 线程安全的查找表

```
template<typename Key,typename Value,typename Hash=std::hash<Key> >
class threadsafe_lookup_table
{
private:
  class bucket_type
  {
  private:
    typedef std::pair<Key,Value> bucket_value;
    typedef std::list<bucket_value> bucket_data;
    typedef typename bucket_data::iterator bucket_iterator;

    bucket_data data;
    mutable std::shared_mutex mutex;  // 1

    bucket_iterator find_entry_for(Key const& key) const  // 2
    {
      return std::find_if(data.begin(),data.end(),
      [&](bucket_value const& item)
      {return item.first==key;});
    }

  public:
    Value value_for(Key const& key,Value const& default_value) const
    {
      std::shared_lock<std::shared_mutex> lock(mutex);  // 3
      bucket_iterator const found_entry=find_entry_for(key);
      return (found_entry==data.end())?
        default_value:found_entry->second;
```

```
    }

    void add_or_update_mapping(Key const& key,Value const& value)
    {
      std::unique_lock<std::shared_mutex> lock(mutex);  // 4
      bucket_iterator const found_entry=find_entry_for(key);
      if(found_entry==data.end())
      {
        data.push_back(bucket_value(key,value));
      }
      else
      {
        found_entry->second=value;
      }
    }

    void remove_mapping(Key const& key)
    {
      std::unique_lock<std::shared_mutex> lock(mutex);  // 5
      bucket_iterator const found_entry=find_entry_for(key);
      if(found_entry!=data.end())
      {
        data.erase(found_entry);
      }
    }
  };

  std::vector<std::unique_ptr<bucket_type> > buckets;  // 6
  Hash hasher;

  bucket_type& get_bucket(Key const& key) const  // 7
  {
    std::size_t const bucket_index=hasher(key)%buckets.size();
    return *buckets[bucket_index];
  }

public:
  typedef Key key_type;
  typedef Value mapped_type;

  typedef Hash hash_type;
  threadsafe_lookup_table(
    unsigned num_buckets=19,Hash const& hasher_=Hash()):
    buckets(num_buckets),hasher(hasher_)
```

```
  {
    for(unsigned i=0;i<num_buckets;++i)
    {
      buckets[i].reset(new bucket_type);
    }
  }

  threadsafe_lookup_table(threadsafe_lookup_table const& other)=delete;
  threadsafe_lookup_table& operator=(
    threadsafe_lookup_table const& other)=delete;

  Value value_for(Key const& key,
                  Value const& default_value=Value()) const
  {
    return get_bucket(key).value_for(key,default_value);  // 8
  }

  void add_or_update_mapping(Key const& key,Value const& value)
  {
    get_bucket(key).add_or_update_mapping(key,value);  // 9
  }

  void remove_mapping(Key const& key)
  {
    get_bucket(key).remove_mapping(key);  // 10
  }
};
```

这个实现使用了 std::vector<std::unique_ptr<bucket_type>>⑥来持有桶，它允许在构造函数中指定桶的数量。默认是 19 个，它可以是任意一个质数；在桶的数目为质数时（译注：关于使用质数实际上这是个有争议的话题，并没有实质的理论提供支持，详细参考 https://www.partow.net/programming/hashfunctions/），哈希表效果最佳。每一个桶由一个 std::shared_mutex①实例保护，它允许对每个桶进行多个并发的读或单个的写。

由于桶的数量是固定的，所以 get_bucket()⑦⑧⑨⑩可以无锁调用。然后桶的互斥锁就可以上锁了：要么为共享(只读)所有权③，要么为唯一(读/写)所有权④⑤。

这三个函数都使用了 find_entry_for()成员函数②，用来确定条目是否在桶中。每一个桶都包含一个键/值对的 std::list<>，所以添加和删除条目很简单。

我已经讨论了并发角度，并且所有东西都被互斥锁适当地保护，那么异常安全呢？value_for 不修改任何值，所以没什么问题；如果 value_for 抛出异常，也不会影响任何数据结构。remove_mapping 修改链表时，会调用 erase，不过这能保证没有异常抛出，所以也是安全的。那就剩 add_or_update_mapping，它可能会在两个 if 分支上抛出异常：

push_back 是异常安全的，如果有异常抛出，它也会将链表恢复成原来的状态，所以这个分支没有问题；唯一的问题是当你正在替换已存在的值时赋值；如果赋值抛出异常，你就依赖于让原始值不变。不过，总体来说这不会影响数据结构，并且它完全是用户提供类型的属性，因此可以放心的将问题交给用户去处理。

本节开始时，我提到查找表的一个*有的话更好*(nice-to-have)的特性，检索当前状态的快照到，例如：一个 std::map<>中。这要求锁住整个容器，从而保证获取到的状态是一致的拷贝。因为对于查找表的“普通”操作，一次只需要锁住一个桶，这将是唯一一个需要锁住所有桶的操作。因此，只要每次以相同的顺序进行上锁(例如，按桶的索引值递增)，就不会产生死锁。实现如下所示：

清单 6.12 获取 threadsafe_lookup_table 的内容作为一个 std::map<>

```
std::map<Key,Value> threadsafe_lookup_table::get_map() const
{
  std::vector<std::unique_lock<std::shared_mutex> > locks;
  for(unsigned i=0;i<buckets.size();++i)
  {
    locks.push_back(
      std::unique_lock<std::shared_mutex>(buckets[i].mutex));
  }
  std::map<Key,Value> res;
  for(unsigned i=0;i<buckets.size();++i)
  {
    for(bucket_iterator it=buckets[i].data.begin();
        it!=buckets[i].data.end();
        ++it)
    {
      res.insert(*it);
    }
  }
  return res;
}
```

清单 6.11 中的查找表实现增加了整个查找表的并发性，方法是分别锁住每个桶，并使用 std::shared_mutex 来允许读者在每个桶上并发访问。但是，如果你可以通过更细粒度的锁来增加一个桶上的并发会怎么样？在下一节中，你将通过使用支持迭代器的线程安全链表容器来实现这一点。

## 6.3.2 使用锁实现线程安全链表

链表是最基本的数据结构之一，因此它应该很容易实现成线程安全的，不应该吗？嗯，这取决于你想要什么设施，如果你需要一个提供迭代器支持（这是我避免在你的映射中添加的东西，因为它太复杂了）的设施就很难实现。STL 风格迭代器的基本问题是，迭

代器本身会持有某种引用指向容器的内部数据结构，如果容器能被另外一个线程修改，这个引用需要仍然有效，这需要迭代器在部分结构上持有锁。考虑到 STL 风格迭代器的生命周期完全不受容器控制，这不是个好主意。

可选的一种做法是提供迭代函数，例如将 for_each 作为容器本身的一部分。这能让容器很干脆地负责迭代和上锁，不过这将违反第 3 章避免死锁的指南。为了让 for_each 做一些有用的事情，当持有内部锁的时候，必须调用用户提供的代码。不光这样，还需要传递每个数据项的引用到用户代码中，这样用户代码才能对容器中的元素进行操作。你可以传一份拷贝到用户代码中来避免这个问题，不过当数据很大时，代价很高。

所以，现在你将取决于用户来确保他们不会在提供的操作中因获取锁而导致死锁，也不会在锁外面通过引用访问而引起数据竞争。查找表使用链表这个例子是很安全的，因为你知道你不会做什么出格的事情。

这就给你留下了一个问题：为链表提供哪些操作。如果你回头看一下清单 6.11 和 6.12，你可以看到需要的各种操作：

- 添加一个项到链表
- 从链表中删除满足某个条件的项
- 从链表中查找满足某个条件的项
- 更新链表中满足某个条件的项
- 拷贝链表中的每一项到另一个容器中

要使它成为一个优秀的通用链表容器，添加更多的操作，比如按位置插入，将会很有帮助，但这对于查找表是不必要的，因此我将把它留给读者作为练习。

对链表进行细粒度上锁的基本思想是让每个节点都有一个互斥锁。如果链表变大，就会有很多互斥锁！这样做的好处是，对链表中不同部分的操作是真正并发的：每个操作只持有它感兴趣的节点上的锁，并在它移动到下一个节点的时释放锁。下一个清单显示了该链表的实现。

清单 6.13 支持迭代器的线程安全链表

```
template<typename T>
class threadsafe_list
{
  struct node  // 1
  {
    std::mutex m;
    std::shared_ptr<T> data;
    std::unique_ptr<node> next;
    node():  // 2
      next()
    {}
```

```
    node(T const& value):  // 3
      data(std::make_shared<T>(value))
    {}
  };

  node head;

public:
  threadsafe_list()
  {}

  ~threadsafe_list()
  {
    remove_if([](node const&){return true;});
  }

  threadsafe_list(threadsafe_list const& other)=delete;
  threadsafe_list& operator=(threadsafe_list const& other)=delete;

  void push_front(T const& value)
  {
    std::unique_ptr<node> new_node(new node(value));  // 4
    std::lock_guard<std::mutex> lk(head.m);
    new_node->next=std::move(head.next);  // 5
    head.next=std::move(new_node);  // 6
  }

  template<typename Function>
  void for_each(Function f)  // 7
  {
    node* current=&head;
    std::unique_lock<std::mutex> lk(head.m);  // 8
    while(node* const next=current->next.get())  // 9
    {
      std::unique_lock<std::mutex> next_lk(next->m);  // 10
      lk.unlock();  // 11
      f(*next->data);  // 12
      current=next;
      lk=std::move(next_lk);  // 13
    }
  }

  template<typename Predicate>
  std::shared_ptr<T> find_first_if(Predicate p)  // 14
```

```
  {
    node* current=&head;
    std::unique_lock<std::mutex> lk(head.m);
    while(node* const next=current->next.get())
    {
      std::unique_lock<std::mutex> next_lk(next->m);
      lk.unlock();
      if(p(*next->data))  // 15
      {
         return next->data;  // 16
      }
      current=next;
      lk=std::move(next_lk);
    }
    return std::shared_ptr<T>();
  }

  template<typename Predicate>
  void remove_if(Predicate p)  // 17
  {
    node* current=&head;
    std::unique_lock<std::mutex> lk(head.m);
    while(node* const next=current->next.get())
    {
      std::unique_lock<std::mutex> next_lk(next->m);
      if(p(*next->data))  // 18
      {
        std::unique_ptr<node> old_next=std::move(current->next);
        current->next=std::move(next->next);
        next_lk.unlock();
      }  // 20
      else
      {
        lk.unlock();  // 21
        current=next;
        lk=std::move(next_lk);
      }
    }
  }
};
```

清单 6.13 中的 threadsafe_list<>是一个单链表，每个条目是一个 node 结构①。一个默认构造的 node 作为链表的 head，开始时 next 指针②的值是 NULL。新节点通过 push_front()函数添加；首先会构造一个新节点④，它会在堆上分配内存③来存储数据，

同时将 next 指针置为 NULL。然后，为了获取正确的 next 值⑤，需要获取 head 节点的互斥锁，并且通过设置 head.next 指向这个新节点⑥，这样就把节点插在链表的前面。目前一切良好：只需要锁住一个互斥锁，就能将新的数据项添加到链表，所以不存在死锁的风险。另外，较慢的内存分配操作发生在锁的外面，所以锁只用来保护需要更新的几个指针，这些操作也不会失败。现在转向迭代函数。

首先，来看一下 for_each()⑦。这个操作把某种类型的 Function 应用到链表中的每个元素；与大多数标准库算法一样，都会按值传递这个 Function，这样不管是真正的函数或者是重载了调用操作符的函数对象都能正常工作。在这里，函数必须接受类型为 T 的值作为唯一参数。这里有个交叉（hand-over-hand）上锁的过程。起初，需要锁住 head 节点⑧的互斥锁。然后，安全的获取指向下一个节点的指针(使用 get()获取，因为对这个指针没有所有权)。如果那个指针不为 NULL⑨，为了处理数据，就需要上锁那个节点的互斥锁⑩。一旦锁住了那个节点，就可以释放上一个节点的锁⑪，并调用指定的函数⑫。当函数执行完成时，就可以更新 current 指针指向刚处理过的节点，并将所有权从 next_lk 移出到 lk⑬。因为 for_each 直接传递每个数据项到 Function 中，你可以借此更新数据项（如果必要的话）或者把他们拷贝到另外一个容器，或者任何其他处理。如果函数行为良好，这种方式是安全的，因为整个调用期间数据项对应节点的互斥锁已经上锁。

find_first_if()⑭和 for_each()类似；关键的区别在于谓词必须在匹配的时候返回 true，不匹配的时候返回 false⑮。一旦匹配，直接返回找到的数据⑯，而非继续查找。可以使用 for_each()来实现，不过它即使找到了一个匹配项，也会不必要地继续处理链表的剩余部分。

remove_if()⑰稍有不同，因为这个函数需要更新链表；所以，不能使用 for_each()。如果谓词返回 true⑱，通过更新 current->next⑲，将节点从链表移除。做完这些，就可以释放 next 节点对应的互斥锁。当 std::unique_ptr<node>超出作用域⑳时，你移动到这个变量中的节点会被删除。这种情况下，就不需要更新 current，因为你需要检查新的 next 节点。如果谓词返回 false，需要和之前一样往前迭代 。

那么，所有这些互斥锁会有死锁或竞争条件吗？只要提供的谓词和函数行为良好，那肯定没有。迭代通常都是单向的，从 head 节点开始，并且总是在释放当前节点之前，锁住下一个互斥锁，所以不同线程肯定有相同的上锁顺序。唯一可能出现竞争条件的地方就在 remove_if()⑳中删除已移除节点的时候。因为，操作在解锁互斥锁后进行(销毁已上锁的互斥锁是未定义的行为)。但稍加思索就可以确定这无疑是安全的，因为仍然持有前一个节点(current)的互斥锁，所以不会有新的线程尝试去获取正在删除节点的互斥锁。

并发的机会如何？细粒度上锁的全部意义在于提高在单个互斥锁上并发的可能性，你做到了吗？是的，已经做到了：同一时间内，不同线程可以在不同节点上工作，无论是用 for_each()对每一个节点进行处理，还是用 find_first_if()查找，亦或是使用 remove_if()删除一些元素。不过，因为互斥锁必须依次上锁，所以线程不能超越彼此。如果一个线程花费大量的时间处理一个特殊节点，其他线程到达这个特殊节点的时候就必须等待。

# 总结

本章首先讨论了为并发设计数据结构的含义，并为此提供了一些指南。然后，完成了几个通用数据结构(栈，队列，哈希表和单链表)的实现，讨论了如何应用这些指南来实现它们（以一种为并发访问而设计的方式），并使用锁来保护数据以及避免数据竞争。现在，你应该有能力洞察自己数据结构的设计，看看哪里有并发的机会，以及哪里有潜在的竞争条件。

在第 7 章中，我们将看到完全避免锁的方法，通过使用低层次原子操作来提供必要的顺序约束，同时遵循相同的指南。

# 第 7 章 设计无锁并发数据结构

**本章主要内容**

- 无锁并发数据结构的实现
- 无锁数据结构中的内存管理技术
- 帮助实现无锁数据结构的简单指南

上一章讨论了为并发设计数据结构的通用特性，并根据指南思考如何设计以确保安全。然后研究了一些通用数据结构，也讨论了使用互斥锁对共享数据进行保护的示例实现。前几个例子使用单个互斥锁来保护整个数据结构，但之后的例子使用多个锁来保护数据结构不同的部分，从而允许对数据结构进行更高级别的并发访问。

互斥锁是一种强大的机制，可以保证在多线程情况下安全的访问数据结构，而不会有竞争条件或破坏的不变量。使用它们的代码的行为也比较好理解：代码要么在保护数据的互斥锁上持有锁，要么没有。不过，这并非总是一帆风顺；你已经看到第 3 章中由于不正确的使用锁会导致死锁，并且在基于锁的队列和查找表例子中，你也看到了细粒度锁是如何影响真正并发的。如果能写出一个不用锁的并发数据结构，就可能避免这些问题。这样的数据结构称为*无锁*（lock-free）数据结构。

本章我们将讨论第 5 章介绍的原子操作的内存顺序属性是如何用于构建无锁数据结构。掌握第 5 章中所有的内容对理解本章至关重要。设计这样的数据结构时要极其小心，因为这样的数据结构很难正确实现，并且导致其失败的条件很难复现。我们将从无锁数据结构的含义开始；然后介绍使用它们的理由，接着完成一些示例，最后拟定一些通用的指南。

## 7.1 定义和影响

使用互斥锁、条件变量，以及期望来同步数据的算法和数据结构叫做*阻塞*（blocking）数据结构和算法。程序调用库函数将会挂起执行线程，直到另一个线程执行一个操作。这些库调用称为*阻塞*调用，因为在障碍被清除前线程无法跨越这个点。通常，操作系统会完全挂起一个阻塞的线程(并将其时间片分配给其他线程)，直到其被另一个线程排除障碍；不管是解锁一个互斥锁、通知一个条件变量，还是让期望就绪。

不使用阻塞库的数据结构和算法被称为*非阻塞的*（nonblocking）。不过，并非所有的非阻塞数据结构都是无锁的，所以我们来看一下各种类型的非阻塞数据结构。

### 7.1.1 非阻塞数据结构的类型

在第 5 章中，我们使用 std::atomic_flag 实现了一个简单的自旋锁。代码复制如下。

清单 7.1 使用 std::atomic_flag 实现的自旋锁

```
class spinlock_mutex
{
  std::atomic_flag flag;
public:
  spinlock_mutex():
    flag(ATOMIC_FLAG_INIT)
  {}

  void lock()
  {
    while(flag.test_and_set(std::memory_order_acquire));
  }

  void unlock()
  {
    flag.clear(std::memory_order_release);
  }
};
```

这段代码没有调用任何阻塞函数，lock()不断循环，直到 test_and_set()返回 false。这是它为什么叫*自旋锁*（spin lock）的原因——代码在循环中“旋转”。因为没有阻塞调用，所以任意使用这个互斥锁来保护共享数据的代码都是非阻塞的。不过，它不是无锁的。它仍然是个互斥锁，并且一次只能被一个线程锁住。因此，只知道“非阻塞的”的概念在大多数情况下是不够的。相反，你需要知道应用哪个（如果有的话）更明确的术语定义。这些是：

- 无障碍——如果所有其他线程都暂停了，任何给定的线程都将在有限的步骤内完成其操作。
- 无锁——如果多个线程对一个数据结构进行操作，经过有限的步骤后，其中一个线程将完成它的操作。
- 无等待——即使有其他线程也在对该数据结构进行操作，每个线程都将在有限步骤内完成其操作。

大多数情况下无障碍算法不是特别有用——很少有其他线程都暂停的情况，因此它更适合于刻画一个失败的无锁实现。让我们进一步了解这些特性包含哪些内容，我们首先从无锁开始，这样你就可以了解包含哪些类型的数据结构。

## 7.1.2 无锁数据结构

一个可以称为无锁的数据结构，应该支持多个线程并发的访问数据结构。它们没有必要是相同的操作；一个无锁队列可能允许一个线程推入数据，另一个线程弹出数据，当有两个线程同时尝试推入元素时就会被打断。不仅如此，当其中一个正在访问数据的线程被

调度器中途挂起时，其他线程必须仍能能够继续完成自己的工作，而无需等待挂起的线程。

使用“比较/交换”操作的算法，通常都包含一个循环。使用“比较/交换”操作的原因是与此同时另一个线程可能已经修改了数据，这种情况下，在再次尝试“比较/交换”之前，代码需要重做部分操作。如果在其他线程挂起的情况下，“比较/交换”最终会成功，那么这样的代码仍然可以是无锁的。如果不是这样的话，就需要一个自旋锁，而这是“非阻塞的”但不是“无锁的”。

带有循环的无锁算法可能导致一个线程处于“饥饿”状态。如果另一个线程在“错误”时机执行操作，其他线程可能取得进展，但是第一个线程需要不停的重试操作。规避了这个问题的数据结构是“无等待”同时也是“无锁的”。

### 7.1.3 无等待数据结构

“无等待”数据结构是无锁数据结构，并且有额外的属性——每个线程都能在有限的步骤内完成操作，而不管其他线程行为如何。由于与其他线程发生冲突而可能陷入无限次重试的算法并不是无等待的。本章的大多数例子都有这个特性——它们有一个在compare_exchange_weak 或 compare_exchange_strong 操作上的循环，并且循环次数没有上限。操作系统对线程进行进行调度，意味着一个给定的线程可能循环非常多次，但其他线程可能循环次数就很少。因此，这些操作不是“无等待”的。

正确实现一个“无等待”数据结构极其困难。为了保证每个线程都能在有限的步骤内完成操作，你必须确保每次操作都一趟执行，并且当前线程中的操作不会导致另一个线程失败。这就会让各种操作的整体算法变的相当复杂。

考虑到实现无锁或无等待的数据结构非常困难，所以需要充足的理由才值得实现一个。由于需要确保收益要大于成本，因此，让我们检查一下影响平衡的那些点。

### 7.1.4 无锁数据结构的优缺点

归根结底，使用无锁结构的主要原因是将并发最大化。使用基于锁的容器，一个线程总是必须阻塞并等待另一个线程完成其操作，然后它才能继续；通过互斥防止并发是互斥锁的全部目的。在无锁数据结构中，某些线程每步都能取得进展。在无等待数据结构中，无论其他线程在做什么，每一个线程都可以向前运行，不需要等待。但是这种令人神往的属性难以实现。最后很容易变成本质上是自旋锁的实现。

使用无锁数据结构的第二个原因是鲁棒性。如果一个线程在持有锁时死亡，那么数据结构将会被永久性的破坏。但是，如果一个线程在对无锁数据结构执行操作的中途死亡，那么除了线程的数据之外，没有什么会丢失；其他线程可以正常运行。

另一方面，如果不能让线程互斥的访问数据结构，那你必须小心确保不变量被支持或选择可被支持的替代不变量。同样，你必须关注施加在操作上的顺序约束。为了避免和数

据竞争相关的未定义行为，必须在修改时使用原子操作。不过，这还不够；你必须确保修改以正确的顺序对其他线程可见。所有这些意味着为了实现线程安全的数据结构，不用锁的方式要比使用锁难多了。

由于没有任何锁，无锁数据结构不可能发生死锁的问题，尽管可能存在活锁。*活锁*（live lock）的产生是当两个线程同时尝试修改数据结构时，对每个线程而言，对方所做的修改都要求操作重新启动，所以两个线程会循环并重试。试想有两个人要通过狭窄的缝隙，如果他们同时出发，就会陷入僵局，因此他们必须出来然后重来，除非有人先到达(不管是商量好了，还是行动更迅速，亦或是走狗屎运)，否则循环将一直重复。在这种简单示例中，活锁的寿命都很短，因为其依赖于线程的精确调度。因此它们只会降低性能，不会引起长期问题，但它们仍然需要注意。根据定义，无等待的代码不会受到活锁的影响，因为执行操作的总步骤数是有上限的。另一方面，无等待的算法要比等待算法更复杂，且即使没有其他线程访问数据结构，也可能需要更多步骤来完成相应操作。

这给我们带来了“无锁”和“无等待”代码的另一个缺点：虽然提高了操作并发的潜力并且减少了单个线程的等待时间，但是它可能会降低整体的性能。首先，无锁代码的原子操作比非原子操作要慢得多，而且很可能无锁数据结构中原子操作大大多于基于锁的数据结构中互斥锁锁住的代码。不仅如此，访问相同原子变量的硬件必须在线程间同步。第8章将看到“乒乓缓存”（cache ping-pong）与多个线程访问相同的原子变量相关，它会导致严重的性能消耗。与所有情况一样，不管怎样，在提交基于锁的数据结构和无锁数据结构之前，检查相关的性能表现(不管是最坏等待时间、平均等待时间、总体执行时间或其他指标)是很重要的。

现在让我们看几个例子。

## 7.2 无锁数据结构示例

为了演示一些在设计无锁数据结构中用到的技术，我们将研究一系列简单数据结构的无锁实现。不仅在每个例子中描述一个有用数据结构的实现，还将重点突出设计无锁数据结构特殊的地方。

之前提过，无锁结构依赖原子操作和相关的内存顺序保证来确保数据以正确的顺序对其他线程可见。起初，我们所有的原子操作将用默认的 memory_order_seq_cst 内存顺序，因为它最好理解（请记住：所有的 memory_order_seq_cst 操作构成一个全序）。但后面的例子中将会减少一些顺序约束到 memory_order_acquire，memory_order_release，甚至 memory_order_relaxed。虽然例子中没有直接使用互斥锁，但值得谨记的是只有 std::atomic_flag 保证是无锁实现的。在一些平台上，C++标准库实现中看起来像无锁的代码可能内部使用了锁(更多细节请查阅第 5 章)；在这些平台上，简单的基于锁的数据结构可能更合适，但还有更多的事情要做；在选择一种实现前，需要明确需求，并且对满足这些需求的各种选项进行性能分析。

因此，让我们从最简单的数据结构开始：栈。

### 7.2.1 实现一个无锁的线程安全栈

栈相对比较简单：检索节点的顺序和添加节点的顺序想反——后入先出(LIFO)。因此一旦一个值入栈后，你要确保它立刻能被另一个线程安全的检索，并且确保只有一个线程能返回给定值也同样重要。最简单的栈是链表，head 指针指向第一个节点(也就是下一个被检索的节点)，并且每个节点依次指向下一个节点。

在这种方案下，添加一个节点相对比较简单：

1. 创建一个新节点。
2. 设置它的 next 指针指向当前的 head 节点。
3. 设置 head 节点指向新节点。

这在单线程的环境中工作的很好，不过其他线程也需要修改栈时，就不行了。当有两个线程同时添加节点的时候，第 2 步和第 3 步的之间会产生竞争条件：在你的线程执行第 2 步读取 head 的值和第 3 步更新 head 的值之间，第二个线程可能修改 head 的值。这会导致其他线程的修改被丢弃，或者造成更加严重的后果。在解决这个竞争条件之前，还要注意：一旦 head 更新并指向了新节点，另一个线程就能读到这个节点。因此，在 head 设置为指向新节点前，新节点完全准备好就至关重要；因为之后就不能修改这个节点。

那么怎么搞定这个可恶的竞争条件？答案就是在第 3 步的时候使用原子的比较/交换操作，来确保步骤 2 读取到 head 以来，不会对 head 进行修改；如果有修改时，可以循环然后重试。下面的代码就展示了不用锁来实现线程安全的 push()函数。

清单 7.2 不用锁实现 push()

```
template<typename T>
class lock_free_stack
{
private:
  struct node
  {
    T data;
    node* next;

    node(T const& data_):  // 1
     data(data_)
    {}
  };

  std::atomic<node*> head;

public:
  void push(T const& data)
```

```
  {
    node* const new_node=new node(data); // 2
    new_node->next=head.load();  // 3
    while(!head.compare_exchange_weak(new_node->next,new_node));  // 4
  }
};
```

上面代码完美匹配之前所说的三个步骤：创建一个新节点②，设置新节点的 next 指针指向当前 head③，并设置 head 指针指向新节点④。通过 node 的构造函数把数据填充到其自身①，保证了节点在构造完成后随时可以被原子操作使用。之后使用 compare_exchange_weak()来保证在被存储到 new_node->next 的 head 指针和之前的值一样③，如果一样的话就把 new_node 设置给它。这些代码也用到了“比较/交换”的特色功能：如果返回 false 意味着比较失败(例如，head 被另一个线程修改)，第一个参数(new_node->next)的值会被 head 中的内容更新。所以不需要在循环中每次都重新加载 head 指针，因为编译器会帮你做这件事。另外，因为在失败时直接进行循环，所以可以使用 compare_exchange_weak，在某些架构上，它可以产生比 compare_exchange_strong 更优的代码(参见第 5 章)。

虽然现在还没有 pop()操作，但可以先快速检查一下 push()的实现是否遵从指南。唯一一个能抛出异常的地方就是在构造新 node 时①，不过它会自行清理，链表也没有被修改，所以非常安全。由于你把数据存储为为 node 的一部分，并且使用 compare_exchange_weak()来更新 head 指针，所以这里没有有问题的竞争条件。一旦“比较/交换”成功时，节点就在链表中并且随时可以被取走。因为没有锁，所以不存在死锁，所以你的 push()函数非常棒。

现在已经有了往栈中添加数据的方法，你需要一个删除数据的方法。表面看起来也非常简单：

1. 读取 head 的当前值保存到 node。
2. 读取 head->next。
3. 设置 head 为 head->next。
4. 返回 node 中的数据。
5. 删除 node 节点。

但在多线程情况下，就没那么简单了。如果有两个线程要从栈中移除数据时，两个线程可能在步骤 1 中读取到相同的 head 值。其中一个线程处理到步骤 5 时，另一个线程还在处理步骤 2，第二个线程将会解引用一个悬垂指针。这是编写无锁代码时遇到的最大的问题之一，所以现在只好跳过步骤 5，让节点泄漏。

但还是没有解决所有问题。还有另一个问题：如果两个线程读取到相同的 head 值时，它们将返回同一个节点。这违反了栈数据结构的意图，所以需要避免发生这种情况。可以像在 push()函数中解决竞争条件那样来解决这个问题：使用比较/交换操作更新 head。当比较/交换操作失败时，不是一个新节点已被推入，就是另一个线程弹出了节点。无论是哪种情况，都得返回步骤 1(但比较/交换操作会为你重新读取 head)。

一旦比较/交换成功，就可以确定当前线程是从栈上弹出指定节点的唯一线程，之后就可以放心的执行步骤 4 了。这里第一次尝试实现 pop()：

```
template<typename T>
class lock_free_stack
{
public:
  void pop(T& result)
  {
    node* old_head=head.load();
    while(!head.compare_exchange_weak(old_head,old_head->next));
    result=old_head->data;
  }
};
```

尽管这段代码很简洁，但除了节点泄露之外还有几个问题。首先，这段代码在链表为空时没法工作：当 head 是空指针时，尝试访问 next 指针是未定义的行为。这很容易解决：在 while 循环检查是否为 nullptr，并在空栈上抛出一个异常，或者返回一个 bool 值来表明成功与否。

第二个问题是异常安全。当我们在第 3 章中第一次引入线程安全的栈时，已经见识了按值返回对象会导致异常安全问题：在拷贝返回值时如果抛出异常，这个值会丢失。如果那样的话，传入引用是一种可以接受的解决方案，这样就能保证当有异常抛出时，栈不会发生改变。不幸的是，这对你来说是个奢望；一旦你知道你是唯一返回这个节点的线程时，你只能安全的拷贝数据，这意味着这个节点已经从栈上删除了。因此，通过引用获取返回值的方式不再有优势：你可能还是要按值返回。若想要安全的返回，必须使用第 3 章中的其他选项：返回指向数据值的(智能)指针。

如果返回一个智能指针，你可以返回 nullptr 来表明没有值可返回，但这要求数据在堆上分配内存。如果将内存分配做为 pop()的一部分，问题还是没有改善，因为堆分配内存可能会抛出一个异常。相反，你可以在 push()操作中分配内存——反正要为 node 分配内存，在哪不一样呢？返回一个 std::shared_ptr<>不会抛出异常，所以 pop()是安全的。将这些放在一起，就得到如下的清单。

清单 7.3 会泄露节点的无锁栈

```
template<typename T>
class lock_free_stack
{
private:
  struct node
  {
    std::shared_ptr<T> data;  // 1 指针持有数据
    node* next;
```

```
    node(T const& data_):
      data(std::make_shared<T>(data_))  // 2 为新分配的 T 创建
std::shared_ptr
    {}
  };

  std::atomic<node*> head;

public:
  void push(T const& data)
  {
    node* const new_node=new node(data);
    new_node->next=head.load();
    while(!head.compare_exchange_weak(new_node->next,new_node));
  }

  std::shared_ptr<T> pop()
  {
    node* old_head=head.load();
    while(old_head && // 3 解引用前检查 old_head 是否为空指针
      !head.compare_exchange_weak(old_head, old_head->next));
    return old_head ? old_head->data : std::shared_ptr<T>();  // 4
  }
};
```

数据现在由指针持有①，所以 node 的构造函数必须在堆上为数据分配内存②。在 compare_exchage_weak()循环中③，解引用 old_head 之前，你也必须检查指针是否为 nullptr。最后，如果栈中有节点，将会返回节点中的数据；否则返回一个空指针④。注意，尽管结构是无锁的，但并不是无等待的，因为只要 compare_exchange_weak()保持失败，push()和 pop()函数中的 while 循环理论上可以无限循环下去。

如果有一个垃圾收集机制帮助你(就像托管语言一样，比如 C#或 Java)，那到这就完事了；老的节点一旦它不再被任何线程访问，将被收集和循环利用。然而，有垃圾收集机制的 C++编译器并不是很多，所以通常需要自己实现一个。

## 7.2.2 终止泄漏：在无锁数据结构中管理内存

第一次研究 pop()的时候，为了避免当有线程删除一个节点的同时，另一个线程还持有指向该节点的指针并且要解引用，导致竞争条件而选择了泄露节点。对 C++程序而言，内存泄露是不可接受的，所以必须解决这个问题！

基本的问题在于，当要释放一个节点时，需要确认没有其他线程持有这个节点。如果只有一个线程在一个特定的栈实例上调用 pop()，你就大功告成。节点是在 push()中创建，push()并不访问现存节点的内容。所以能够访问给定节点的线程包括把节点添加到栈

中的线程以及调用 pop 的线程。一旦节点入栈后，push()就不会再碰节点，所以只剩下调用 pop()的线程——如果只有一个这样的线程，那么调用 pop()的线程肯定是唯一能访问这个节点的线程，也就可以安全的删除这个节点。

另一方面，如果你要处理在同一个栈实例上多线程调用 pop()的情况，你需要一些方法来追踪节点什么时候能被安全的删除。这意味着你需要为节点写一个专用的垃圾收集器。尽管听起来很棘手，但也没那么糟糕：你只需要检查节点，并且只需要检查那些被 pop()访问的节点。不用担心 push()中的节点，因为这些节点直到入栈以后才能被访问到，然而多线程通过 pop()可能访问相同的节点。

如果没有线程调用 pop()时，就可以安全删除当前等待删除的节点。因此当你提取了数据后，如果你添加节点到“等待删除”链表中，然后，如果没有线程调用 pop()时，就可以安全的删除所有这些节点。那怎么知道没有线程调用 pop()呢？很简单——计数即可。如果你进入时递增计数，退出时递减计数，那么当计数器变为 0 时，从“等待删除”链表中删除节点就是安全的。这个计数器必须是原子的，这样才能安全的被多线程访问。下面的清单展示了修复后的 pop()函数，一些支持函数将在清单 7.5 中给出。

清单 7.4 当没有线程执行 pop()时回收节点

```
template<typename T>
class lock_free_stack
{
private:
  std::atomic<unsigned> threads_in_pop;  // 1 原子变量
  void try_reclaim(node* old_head);

public:
  std::shared_ptr<T> pop()
  {
    ++threads_in_pop;  // 2 首先增加计数
    node* old_head=head.load();
    while(old_head &&
      !head.compare_exchange_weak(old_head,old_head->next));
    std::shared_ptr<T> res;
    if(old_head)
    {
      res.swap(old_head->data);  // 3 从节点中提取数据，而非拷贝指针
    }
    try_reclaim(old_head);  // 4 可以的话，回收删除的节点
    return res;
  }
};
```

原子变量 threads_in_pop①用来记录有多少个线程试图弹出栈中的项。在 pop()②函数的开头，递增计数；调用 try_reclaim()时，递减计数，这个函数在节点被移除后调用

④。由于会延迟删除节点本身，可以通过 swap()函数来移除节点上的数据③，而不是拷贝指针。这样的话，当不再需要这些数据的时候，这些数据会自动删除，否则数据会持续存在，因为还没删除的（not-yet-deleted）节点还有引用指向这个数据（译注：如果不交换出去的话，由于节点会在待删除链表中排队，不知道什么时候会被删除，一直会有一个引用指向数据，但实际上取走数据后，如果业务不要这个数据就可以释放数据的内存，节点本身和数据的生命周期是分开的）。接下来看一下 try_reclaim()是如何实现的。

清单 7.5 采用引用计数的回收机制

```
template<typename T>
class lock_free_stack
{
private:
  std::atomic<node*> to_be_deleted;

  static void delete_nodes(node* nodes)
  {
    while(nodes)
    {
      node* next=nodes->next;
      delete nodes;
      nodes=next;
    }
  }

  void try_reclaim(node* old_head)
  {
    if(threads_in_pop==1)  // 1
    {
      node* nodes_to_delete=to_be_deleted.exchange(nullptr);  // 2 索取
“待删除”链表
      if(!--threads_in_pop)  // 3 当前是否是唯一调用 pop()的线程？
      {
        delete_nodes(nodes_to_delete);  // 4
      }
      else if(nodes_to_delete)  // 5
      {
         chain_pending_nodes(nodes_to_delete);  // 6
      }
      delete old_head;  // 7
    }
    else
    {
      chain_pending_node(old_head);  // 8
```

```
      --threads_in_pop;
    }
  }

  void chain_pending_nodes(node* nodes)
  {
    node* last=nodes;
    while(node* const next=last->next)  // 9 跟着 next 指针遍历到链表末尾
    {
      last=next;
    }
    chain_pending_nodes(nodes,last);
  }

  void chain_pending_nodes(node* first,node* last)
  {
    last->next=to_be_deleted;  // 10
    while(!to_be_deleted.compare_exchange_weak(  // 11 用循环来保证
last->next 的正确性
      last->next,first));
  }

  void chain_pending_node(node* n)
  {
    chain_pending_nodes(n,n);  // 12
  }
};
```

当你尝试回收节点时①，如果 threads_in_pop 的值是 1，那你就是当前唯一在 pop()中的线程，这时就可以安全的删除移除的节点⑦，删除未决（pending）节点可能也是安全的。当数值不是 1 时，删除任何节点都不安全，所以需要把节点添加到未决链表⑧。

假设某一时刻，threads_in_pop 的值为 1。就需要尝试回收未决节点，如果不回收的话，节点会保持未决状态直到整个栈被销毁。为了回收，首先要通过原子 exchange 操作索取②链表，并递减 threads_in_pop 的计数③。如果之后计数的值变为 0，就可以确定没有其他线程访问未决节点链表。虽然可能会有新的未决节点出现，不过现在你不需要为此烦恼，只要安全回收你的链表就好了，而后，可以调用 delete_nodes 对链表进行迭代，并将其删除④。

如果计数在递减后不为 0，回收节点就不安全；因此如果存在未决节点⑤，就需要将其挂在未决删除链表之后⑥，这种情况会发生在多个线程并发访问数据结构的时候。其他线程在第一次测试 threads_in_pop①和“索取”链表②之间调用 pop()，潜在的添加一个新节点到未决链表，而这个节点仍被一个或多个线程访问。图 7.1 中，线程 C 添加节点 Y 到 to_be_deleted 链表，即使线程 B 仍通过 old_head 引用它，并且尝试读它的 next 指针。因此线程 A 不能删除节点，只有这样才不至于导致线程 B 发生未定义的行为。

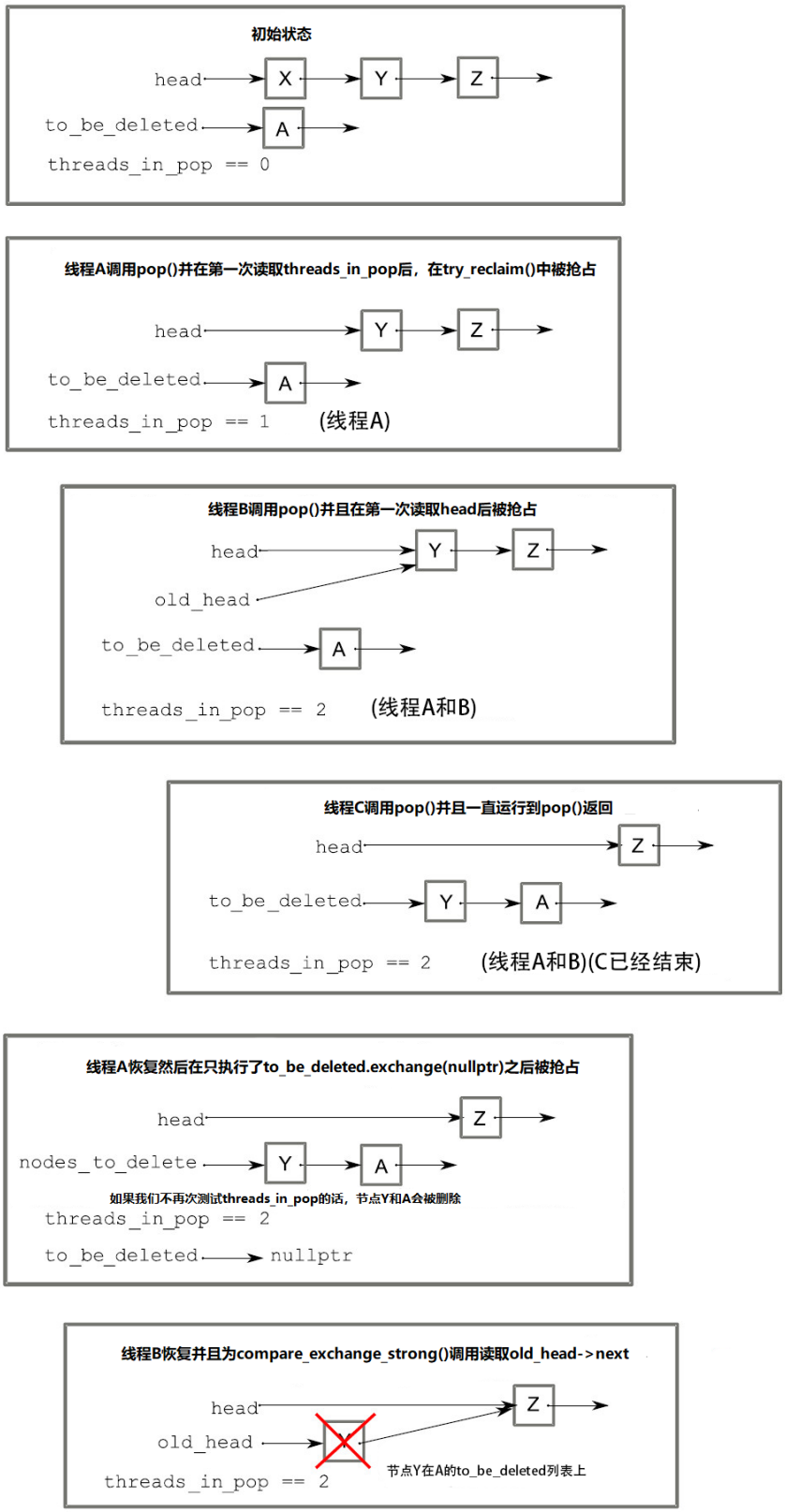


图 7.1 三个线程并发调用 pop()，展示了在 try_reclaim()中为什么需要在回收要删除的节点后对 threads_in_pop 进行检查。

为了把未决删除节点添加到未决链表，需要重用节点的 next 指针将它们链接在一起。在将现有链重新链接回链表的情况下，需要遍历链以找到尾部⑨，将最后一个节点的 next 指针替换为当前 to_be_deleted 指针⑩，并将链中第一个节点存储为新的 to_be_deleted 指针⑪。你必须在循环中使用 compare_exchange_weak 来保证其他线程添加进来的节点不会泄露。这样做有个好处——如果 to_be_deleted 发生改变，可以从链的尾部更新 next 指针。添加单个节点是种特殊情况，因为它既是第一个节点，也是最后一个节点⑫。

在低负载的情况下，这种方式工作的相当好，因为总有合适的静止点没有线程运行 pop()。但这些点稍纵即逝，这也是在回收前，需要测试 threads_in_pop 计数为 0③的原因，也是这个测试发生在删除移除节点⑦之前的原因。删除节点是个耗时的操作，你肯定希望其他线程修改链表的时间窗口越小越好。从第一次发现 threads_in_pop 是 1 到尝试删除节点耗费的时间越长，就越有可能有另一个线程会调用 pop()，因而导致 threads_in_pop 不再等于 1，这样会阻止节点被删除。

在高负载的情况下，可能永远也没有这种静止状态，因为起初在 pop()里面的线程全部离开之前，其他线程已经进入到 pop()，这种情况下 to_be_deleted 链表将会无限增长，这会再次泄漏内存。如果不存在任何静态的时间片，就需要一种替代机制来回收节点。关键是确定什么时候没有更多的线程访问特定节点，这样节点就能被回收。到目前为止，最好理解的机制是使用*风险指针*(hazard pointer)。

## 7.2.3 使用风险指针检测不可回收的节点

“风险指针”这个术语引自 Maged Michael 发现的一项技术[1]。之所以这样叫是因为删除一个可能仍被其他线程引用的节点是危险的。如果其他线程确实持有节点的引用，并继续通过该引用访问节点，就会出现未定义的行为。基本思想是，如果一个线程准备访问另一个线程想要删除的对象，它首先会设置风险指针来引用这个对象，然后通知其他线程使用这个指针是危险的。一旦不再需要这个对象，就可以清除风险指针。如果你看过牛津/剑桥的划船比赛，那你肯定见过类似机制：每个船上的舵手可以举手示意他们还没有准备好。只要有舵手举手，裁判就不能开始比赛。当所有舵手把手放下后，比赛才能开始；不过舵手可以再次举手，只要比赛还未开始并且他们感觉情况有变。

当线程想要删除一个对象时，就必须检查系统中属于其他线程的风险指针。如果没有风险指针引用这个对象，就可以安全的删除对象。否则只能稍后处理。然后周期性的检查留待稍后处理的对象链表以确定他们中是否有现在可以安全删除的对象。

经过一番高层次的描述以后，听起来比较简单，那么在 C++中应该怎么做呢？

首先，你需要个地方用来存储指向访问对象的指针，也就是风险指针本身。这个位置必须对所有线程可见，并且可能访问该数据结构的每个线程都需要一个这样的位置。正确并且高效的分配这个数据结构的确是一个挑战，所以稍后讨论，并且假设有一个 get_hazard_pointer_for_current_thread()函数，它返回风险指针的引用。然后，当你读取一个想解引用的指针时，你需要设置它——在这个例子中是链表中的 head 值：

```
std::shared_ptr<T> pop()
{
  std::atomic<void*>& hp=get_hazard_pointer_for_current_thread();
  node* old_head=head.load();  // 1
  node* temp;
  do
  {
    temp=old_head;
    hp.store(old_head);  // 2
    old_head=head.load();
  } while(old_head!=temp); // 3
  // ...
}
```

必须使用 while 循环来确保 node 不会在读取老的 head 指针①，以及设置风险指针②期间被删除。在这个时间窗口，没有线程知道你正在访问这个特定的节点。幸运的是，老的 head 节点将要删除时，head 本身肯定改变了，所以需要对 head 进行检查并持续循环，直到 head 指针中的值与风险指针中的值相同③。像这样使用风险指针依赖于一个事实：使用一个它引用的对象被删除的指针的值是安全的。如果使用 new 和 delete 的默认实现，这在技术上是未定义的行为，因此需要确保你的实现支持这种用法，或者可以使用允许这种用法的自定义分配器。

既然已经设置了风险指针，就可以继续处理 pop()的其他部分，毫无疑问不会有其他线程从你眼皮底下删除节点。几乎每一次重新加载 old_head 后，在解引用最新读取到的指针值前，都需要更新风险指针。一旦从链表中提取了一个节点，就可以清除风险指针。如果没有其他风险指针引用节点，就可以安全的删除节点；否则，必须将它添加到链表中，之后再进行删除。下面的代码展示了使用这种方案的完整 pop()实现。

清单 7.6 使用风险指针实现 pop()

```
std::shared_ptr<T> pop()
{
  std::atomic<void*>& hp=get_hazard_pointer_for_current_thread();
  node* old_head=head.load();
  do
  {
    node* temp;
    do  // 1 一直循环，直到将风险指针设为 head
    {
      temp=old_head;
      hp.store(old_head);
      old_head=head.load();
    } while(old_head!=temp);
  }
  while(old_head &&
```

```
    !head.compare_exchange_strong(old_head,old_head->next));
  hp.store(nullptr);  // 2 当取出头结点后，清除风险指针
  std::shared_ptr<T> res;
  if(old_head)
  {
    res.swap(old_head->data);
    if(outstanding_hazard_pointers_for(old_head))  // 3 在删除之前检查是否有风险指针引用该节点（因为在上面那个内部循环中可能多个线程都把某个 head 设置为自己线程的风险指针）
    {
      reclaim_later(old_head);  // 4
    }
    else
    {
      delete old_head;  // 5
    }
    delete_nodes_with_no_hazards();  // 6
  }
  return res;
}
```

首先，设置风险指针的循环移到了外部循环里面，这个外部循环会在“比较/交换”操作失败的时候重新加载 old_head①。这里使用了 compare_exchange_strong()，因为在 while 循环内部有工作可做：当 compare_exchange_weak()伪失败后，会导致不必要的重新设置风险指针。这确保了风险指针在解引用 old_head 之前被正确的设置。一旦将一个节点声明为你的，就可以清除风险指针②。如果确实拿到一个节点，就需要检查其他线程上的风险指针，看它们是否引用了该节点③。如果引用了，现在还不能删除它，只能将它放到回收链表中，之后再进行回收④；否则，可以立刻删除节点⑤。最后你调用了一个函数来检查节点，这些节点是由 reclaim_later()收集。如果没有任何风险指针引用这些节点，就可以安全的删除这些节点⑥。如果还有风险指针引用的话，这样的节点就会留给下一个调用 pop()的线程。

还有很多细节隐藏在这些新函数中——get_hazard_pointer_for_current_thread(), reclaim_later(), outstanding_hazard_pointers_for(), 和 delete_nodes_with_no_hazards()——因此让我们揭开神秘的面纱，看看它们是如何工作的。

get_hazard_pointer_for_current_thread()用于分配风险指针实例的确切方案对程序逻辑（尽管会影响效率，你很快就会看到）并不重要。就现在而言，可以使用一个简单的结构：一个固定长度的线程 ID 和指针对的数组。然后 get_hazard_pointer_for_curent_thread()可以通过搜索这个数组来找到第一个空闲槽位，并将当前线程的 ID 设置为这个槽的 ID 条目。线程退出时，槽通过设置 ID 条目为默认构造的 std::thread::id()来释放槽位。如下面清单所示：

清单 7.7 get_hazard_pointer_for_current_thread()的简单实现

```
unsigned const max_hazard_pointers=100;
struct hazard_pointer
{
  std::atomic<std::thread::id> id;
  std::atomic<void*> pointer;
};
hazard_pointer hazard_pointers[max_hazard_pointers];

class hp_owner
{
  hazard_pointer* hp;

public:
  hp_owner(hp_owner const&)=delete;
  hp_owner operator=(hp_owner const&)=delete;
  hp_owner(): // 1
    hp(nullptr)
  {
    for(unsigned i=0;i<max_hazard_pointers;++i)
    {
      std::thread::id old_id;
      if(hazard_pointers[i].id.compare_exchange_strong(  // 2 尝试索取风
险指针的所有权
          old_id,std::this_thread::get_id()))
      {
        hp=&hazard_pointers[i]; // 6
        break;
      }
    }
    if(!hp)  // 7
    {
      throw std::runtime_error("No hazard pointers available");
    }
  }

  std::atomic<void*>& get_pointer()
  {
    return hp->pointer;
  }

  ~hp_owner()
```

```
  {
    hp->pointer.store(nullptr);
    hp->id.store(std::thread::id());
  }
};

std::atomic<void*>& get_hazard_pointer_for_current_thread()  // 3
{
  thread_local static hp_owner hazard;  // 4 每个线程都有自己的风险指针
  return hazard.get_pointer();  // 5
}
```

get_hazard_pointer_for_current_thread()的实现看起来很简单③：它有一个 hp_owner 类型的 thread_local 变量④，用来存储当前线程的风险指针。然后从这个对象返回风险指针⑤。它的工作原理如下：每个线程第一次调用这个函数时，一个新的 hp_owner 实例被创建。这个实例的构造函数①会通过查询“所有者/指针”对的表来寻找一个没有被占用的条目。它使用 compare_exchange_strong()来检查某个条目没有所有者并索取这个条目②。如果 compare_exchange_strong()失败，说明另一个线程拥有这个条目，因此继续考察下一个。当交换成功，当前线程就拥有了这个条目，然后存储这个条目并停止搜索⑥。如果遍历完链表也没有找到空闲条目时⑦，说明有太多线程在使用风险指针，因此抛出一个异常。

对一个给定的线程，一旦创建了 hp_owner 实例，后续访问会很快，因为指针已经缓存起来了，所以不必扫描表。

当每个线程退出时，如果这个线程的 hp_owner 实例被创建的话，将会被销毁。析构函数会在设置拥有者 ID 为 std::thread::id()前，将指针重置为 nullptr，这样就允许另一个线程对这个条目进行复用。

有了 get_hazard_pointer_for_current_thread()的实现后，outstanding_hazard_pointer_for()实现就简单了：扫描风险指针表查找对应条目。

```
bool outstanding_hazard_pointers_for(void* p)
{
  for(unsigned i=0;i<max_hazard_pointers;++i)
  {
    if(hazard_pointers[i].pointer.load()==p)
    {
      return true;
    }
  }
  return false;
}
```

甚至都不需要对记录的所有者进行验证：没有所有者的条目会是一个空指针，所以比较会返回 false，这样简化了代码。

reclaim_later()和 delete_nodes_with_no_hazards()在一个简单的链表上工作；reclaim_later()只是将节点添加到链表中，delete_nodes_with_no_hazards()扫描整个链表，并删除无风险指针的条目。下面的清单展示了相关实现。

清单 7.8 回收函数的简单实现

```
template<typename T>
void do_delete(void* p)
{
  delete static_cast<T*>(p);
}

struct data_to_reclaim
{
  void* data;
  std::function<void(void*)> deleter;
  data_to_reclaim* next;

  template<typename T>
  data_to_reclaim(T* p):  // 1
    data(p),
    deleter(&do_delete<T>),
    next(0)
  {}

  ~data_to_reclaim()
  {
    deleter(data);  // 2
  }
};

std::atomic<data_to_reclaim*> nodes_to_reclaim;

void add_to_reclaim_list(data_to_reclaim* node)  // 3
{
  node->next=nodes_to_reclaim.load();
  while(!nodes_to_reclaim.compare_exchange_weak(node->next,node));
}

template<typename T>
void reclaim_later(T* data)  // 4
{
```

```
  add_to_reclaim_list(new data_to_reclaim(data));  // 5
}

void delete_nodes_with_no_hazards()
{
  data_to_reclaim* current=nodes_to_reclaim.exchange(nullptr);  // 6
  while(current)
  {
    data_to_reclaim* const next=current->next;
    if(!outstanding_hazard_pointers_for(current->data))  // 7
    {
      delete current;  // 8
    }
    else
    {
      add_to_reclaim_list(current);  // 9
    }
    current=next;
  }
}
```

首先，reclaim_later()是一个函数模板而不是个普通函数④。因为风险指针是一个通用方案，你肯定不想绑死在栈节点上。你已经使用 std::atomic<void*>对指针进行存储，因此需要处理任意指针类型。不过不能使用 void*形式，因为当要删除数据项时，delete 需要指针的具体类型。data_to_reclaim 的构造函数很好的处理了这个问题，你马上就能看到；reclaim_later()为你的指针创建了一个新的 data_to_reclaim 实例，并且将实例添加到回收链表中⑤。add_to_reclaim_list()③本身只是一个在链表头的简单 compare_exchange_weak()循环，这跟你以前看到的一样。

回到 data_to_reclaim 的构造函数①：构造函数也是个模板。它把要删除的数据存储为 void*的数据成员，然后存储了一个指针，这个指针指向 do_deltete()一个适当的实例化函数——一个简单的函数，将 void*转换为选择的指针类型，然后删除指向的对象。std::function<>安全的包装了这个函数指针，所以 data_to_reclaim 的析构函数可以通过调用存储的函数对数据进行删除②。

当添加节点到链表时，data_to_reclaim 的析构函数不会被调用；析构函数会在没有未决的风险指针指向那个节点的时候调用，这是 delete_nodes_with_no_hazards()的责任。

delete_nodes_with_no_hazards()使用 exchange()函数获取需要回收的整个链表⑥。这个简单但很关键的步骤确保了只有一个线程回收这个特定集合的节点。其他线程现在就能随意将更多节点添加到回收链表中，甚至尝试对节点进行回收，而不影响那个回收线程。

然后，只要有节点存在于链表中，就需要依次检查每个节点，查看节点是否有未决的风险指针⑦。如果没有的话，就可以安全的将条目删除(并且清除存储的数据)⑧。否则，就只能将这个节点放回链表，后面再回收⑨。

虽然这个简单的实现确实安全的回收了被删除的节点，不过开销增加了很多。扫描风险指针数组需要检查 max_hazard_pointers 个原子变量，并且每次 pop()调用都要做一次。原子操作本来就慢——在台式 CPU 上，原子操作比非原子操作慢 100 倍——这使得 pop()变成昂贵的操作。不仅为删除节点需要扫描风险指针链表，而且等待删除链表上的每个元素也要扫描风险指针链表（译注：其实就是调用 outstanding_hazard_pointers_for 的两个地方，一个在清单 7.6 的③，一个在清单 7.8 的⑦）。显然，这是个糟糕的主意。可能会有 max_hazard_pointers 个节点在链表中，然后你需要和 max_hazard_pointers 个存储的风险指针做比较。天啊，这必须要有更好的办法才行。

**更好的使用风险指针的回收策略**

确实有更好的办法。这里只展示一个简单初级的风险指针实现，来帮助解释这个技术。首先可以用内存换性能。不同于每次调用 pop()都检查回收链表上的每个节点，除非有超过 max_hazard_pointer 数量的节点存在于链表上，否则不会尝试回收任何节点。这样的话可以保证你至少能回收一个节点。如果一直等到链表中的节点数量达到 max_hazard_pointers+1，其实也没好到哪去。一旦有 max_hazard_pointers 个节点，将为大多数 pop()调用尝试回收节点，所以这样也不是很好。不过，如果一直等到 2*max_hazard_pointers 个节点在链表中时，由于最多有 max_hazard_pointers 个节点是活跃的，这样就能保证至少有 max_hazard_pointers 个节点可以被回收，并且再次尝试回收任意节点前，至少会对 pop()有 max_hazard_pointers 次调用。这就好很多。比起每次 pop()调用检查大约 max_hazard_pointers 个节点(不一定能回收到节点)，每 max_hazard_pointers 次 pop()调用，检查 2*max_hazard_pointers 个节点，就会有 max_hazard_pointers 个节点可回收。这实际上就是每次 pop()调用检查两个节点，其中就有一个被回收。

哪怕这个方法有个缺点(不止更大的回收链表增加了内存使用，并且潜在的还有大量可回收节点)：现在需要对回收链表上的节点进行计数，这意味着需要使用原子计数器，并且有多线程争竞争访问回收链表本身。如果内存充裕，可以使用更多内存的来实现更好的回收方案：每个线程通过线程局部变量，持有它自己的回收链表，这样就不需要原子计数和竞争访问了。相反，只需要分配 max_hazard_pointers*max_hazard_pointers 个节点。如果一个线程在它的所有节点被回收前退出，它的本地链表可以像之前一样存储到全局链表中，然后添加到下一个线程的回收链表中，让这个线程做回收操作。

风险指针的另一个缺点是受到 IBM 申请的专利所保护[2]。尽管我相信这个专利已经过期，但如果你在一个专利还在有效期的国家编写软件，你最好咨询专利律师或者确认已获得适当的许可。这在很多无锁内存回收技术中很常见；这是一个活跃的研究领域，所以很多大公司都尽可能申请专利。你可能会问，“为什么花这么大的篇幅来介绍一个大多数人可能都没办法使用的技术呢？”，这是个很好的问题。首先，使用这种技术可能不需要购买一个许可。比如，如果使用 GPL 下的免费软件许可来进行软件开发[3]，你的软件可能被 IBM 的不主张声明（statement of non-assertion）覆盖[4]。其次，也是最重要的，解释这个技术能够展示在编写无锁代码时很多需要重点考虑的东西，比如：原子操作的成本。

最后，有一个提议将风险指针纳入到 C++标准的未来修订版中；因此，知道它们怎么工作是有益处的，即便你有希望在将来能够使用你的编译器供应商的实现。

那么，是否有非专利的内存回收技术可以用于无锁代码？幸好，的确有。引用计数就是这样一种机制。

## 7.2.4 使用引用计数检测节点是否在使用

回到 7.2.2 节，已经看到删除节点的问题是需要检测哪个节点还在被读线程访问。如果能安全并精确的识别节点是否还在被引用，以及什么时候没有线程访问这些节点，你就能删除它们。风险指针通过把使用中的节点存储到链表中来处理这个问题。而引用计数是通过统计每个节点上访问的线程数量来处理这个问题。

这看起来很好很简单，但实际上很难管理：首先，你可能认为 std::shared_ptr<>可以胜任这个任务，毕竟它是有引用计数的指针。不幸的是，虽然 std::shared_ptr<>上有些操作是原子的，但不能保证是无锁的。尽管就其本身而言，这跟原子类型上的操作没有什么不同，但是 std::shared_ptr<>旨在用于多种上下文中，使原子操作无锁可能会给所有类的使用带来开销。如果平台提供的实现对 std::atomic_is_lock_free(&some_shared_ptr)返回 true，那么整个内存回收问题就消失了。因为可以使用 std::shared_ptr<node>用于链表的实现，如清单 7.9 所示。需要注意的是，对弹出节点的 next 指针做清除操作是为了避免当最后一个 std::shared_ptr 引用的给定节点被销毁时，节点深度嵌套销毁。

清单 7.9 使用无锁 std::shared_ptr<>实现的无锁栈

```
template<typename T>
class lock_free_stack
{
private:
  struct node
  {
    std::shared_ptr<T> data;
    std::shared_ptr<node> next;
    node(T const& data_):
      data(std::make_shared<T>(data_))
    {}
  };
  std::shared_ptr<node> head;

public:
  void push(T const& data)
  {
    std::shared_ptr<node> const new_node=std::make_shared<node>(data);
    new_node->next=std::atomic_load(&head);
    while(!std::atomic_compare_exchange_weak(&head,
```

```
        &new_node->next,new_node));
  }

  std::shared_ptr<T> pop()
  {
    std::shared_ptr<node> old_head=std::atomic_load(&head);
    while(old_head && !std::atomic_compare_exchange_weak(&head,
        &old_head,old_head->next));
    if (old_head){
      std::atomic_store(&old_head->next, std::shared_ptr<node>());
      return old_head->data;
    }
    return std::shared_ptr<T>();
  }

  ~lock_free_stack(){
    while(pop());
  }
};
```

不仅对 std::shared_ptr<>使用无锁原子操作的实现很少见，而且记得一致使用原子操作也很难。不过并发技术规范可以帮助你，如果你有一份可用的实现，它在头文件<experimental/atomic>中提供了 std::experimental::atomic_shared_ptr<T>。多数情况下这与 std::atomic<std::shared_ptr<T>>等价，除非 std::atomic<>不能使用 std::shared_ptr<T>，因为它有非平凡的（nontrivial）拷贝语义来确保正确的处理引用计数。std::experimental::atomic_shared_ptr<T>能正确的处理引用计数,并且保证是原子操作。与第 5 章中的其他原子类型一样，在任何给定的实现上它可以是无锁的也可以不是。清单 7.10 是对清单 7.9 的重写。可以看出不需要包含 atomic_load 和 atomic_store 调用使实现变得更简单。

清单 7.10 使用 std::experimental::atomic_shared_ptr<>实现的栈

```
template<typename T>
class lock_free_stack
{
private:
  struct node
  {
    std::shared_ptr<T> data;
    std::experimental::atomic_shared_ptr<node> next;
    node(T const& data_):
      data(std::make_shared<T>(data_))
    {}
  };
  std::experimental::atomic_shared_ptr<node> head;
```

```
public:
  void push(T const& data)
  {
    std::shared_ptr<node> const new_node=std::make_shared<node>(data);
    new_node->next=head.load();
    while(!head.compare_exchange_weak(new_node->next,new_node));
  }

  std::shared_ptr<T> pop()
  {
    std::shared_ptr<node> old_head=head.load();
    while(old_head && !head.compare_exchange_weak(
      old_head,old_head->next.load()));
    if(old_head) {
      old_head->next=std::shared_ptr<node>();
      return old_head->data;
    }
    return std::shared_ptr<T>();
  }

  ~lock_free_stack(){
    while(pop());
  }
};
```

很可能你的 std::shared_ptr<>实现不是无锁的，并且你的实现也没有提供一份无锁的 std::experimental::atomic_shared_ptr<T>，这样的话就需要手动管理引用计数。

一种可能的技术是对每个节点使用两个而不是一个引用计数：一个内部计数和一个外部计数。两个值的和就是对这个节点的引用总数。外部计数与指向节点的指针一起保存，并且每次读取指针的时候外部计数增加。当读线程使用完节点后，递减内部计数。一个简单的读指针操作完成后，外部计数将加 1，内部计数会减 1

当不再需要外部计数/指针对时(该节点不再可以从多个线程访问的位置访问)，给内部计数加上外部计数减一的值，并丢弃外部计数。一旦内部计数等于 0，就没有对该节点的引用，可以将该节点安全的删除。使用原子操作来更新共享数据仍然很重要。现在让我们来看一下使用这种技术实现的无锁栈，它确保节点只在安全的情况下才会回收节点。

下面的清单展示了内部数据结构，以及 push()的实现，非常简单明了。

清单 7.11 使用分离的引用计数把一个节点推入到无锁栈

```
template<typename T>
class lock_free_stack
{
```

```
private:
  struct node;

  struct counted_node_ptr  // 1
  {
    int external_count;
    node* ptr;
  };

  struct node
  {
    std::shared_ptr<T> data;
    std::atomic<int> internal_count;  // 2
    counted_node_ptr next;  // 3

    node(T const& data_):
      data(std::make_shared<T>(data_)),
      internal_count(0)
    {}
  };

  std::atomic<counted_node_ptr> head;  // 4

public:
  ~lock_free_stack()
  {
    while(pop());
  }

  void push(T const& data)  // 5
  {
    counted_node_ptr new_node;
    new_node.ptr=new node(data);
    new_node.external_count=1;
    new_node.ptr->next=head.load();
    while(!head.compare_exchange_weak(new_node.ptr->next,new_node));
  }
};
```

首先，外部计数和节点指针一起包装在 counted_node_ptr 结构中①，然后可以将其用于 node 结构体的 next 指针③，以此同时还有内部计数②。因为 counted_node_ptr 是一个简单的结构体，因此可以和 std::atomic<>模板一起用作链表的 head④。

在支持“双字比较和交换”操作的平台上，由于这个结构体足够小，所以 std::atomic<counted_node_ptr>由于足够小而可以是无锁的。如果平台不支持的话，最好

使用清单 7.9 中的 std::shared_ptr<>版本；因为当类型的体积对平台的原子指令来讲太大的话，std::atomic<>将使用互斥锁来保证其操作的原子性(致使你的“无锁”算法变成“基于锁”的算法)。或者，如果你愿意限制计数器的大小，并且你知道你的平台在指针上有空闲的 bit 位(比如，地址空间只有 48 位，而一个指针占 64 位。译注：intel 平台就是这个情况)，可以将计数存在一个指针的空闲位中，这样就可以塞进一个机器字当中。这样的技巧需要平台特定的知识，因而超出本书讨论的范围。

push()相对简单⑤，你构造了一个 counted_node_ptr 实例，引用新分配出来带有相关数据的 node，并将 node 的 next 指针设置为当前 head。之后你就可以使用 compare_exchange_weak()对 head 的值进行设置，跟之前清单一样。计数就建立起来了，这时 internal_count 为 0，并且 external_count 是 1。因为这是一个新节点所以当前只有一个外部引用指向它(也就是 head 指针本身)。

和往常一样，复杂性来自于 pop()的实现，展示如下。

清单 7.12 使用分离的引用计数从无锁栈中弹出一个节点

```
template<typename T>
class lock_free_stack
{
private:
  void increase_head_count(counted_node_ptr& old_counter)
  {
    counted_node_ptr new_counter;

    do
    {
      new_counter=old_counter;
      ++new_counter.external_count;
    }
    while(!head.compare_exchange_strong(old_counter,new_counter));  //
1

    old_counter.external_count=new_counter.external_count;
  }

public:
  std::shared_ptr<T> pop()
  {
    counted_node_ptr old_head=head.load();
    for(;;)
    {
      increase_head_count(old_head);
      node* const ptr=old_head.ptr;  // 2
```

```
      if(!ptr) // 这个分支没有管前面增加了外部计数，对空链表倒无所谓，因为计数是针对数据的，没有真正指向数据，计数是多少都无所谓，严谨点应该减少计数
      {
        return std::shared_ptr<T>();
      }

      if(head.compare_exchange_strong(old_head,ptr->next))  // 3
      {
        std::shared_ptr<T> res;
        res.swap(ptr->data);  // 4

        int const count_increase=old_head.external_count-2;  // 5

        if(ptr->internal_count.fetch_add(count_increase)==  // 6
           -count_increase)
        {
          delete ptr;
        }

        return res;  // 7
      }
      else if(ptr->internal_count.fetch_sub(1)==1)
      { // 只有上面的 6 发生以后才会出现这个情况，否则内部计数越减越是负数
        delete ptr;  // 8
      }
    }
  }
};
```

一旦加载了 head 的值，首先必须增加对 head 节点的外部引用计数，以表明正在引用这个节点并且确保解引用是安全的。如果在引用计数增加前解引用指针，另一个线程可能在你访问这个节点之前释放它，从而使你持有一个悬垂指针。这就是使用分离的引用计数的主要原因：通过增加外部引用计数，保证了指针在访问期间是有效的。递增操作是通过 compare_exchange_strong()的循环①完成的，这个循环比较和设置整个结构体来确保指针不会在同一时间内被另一个线程修改。

一旦计数增加就能安全的解引用 head 值的 ptr 字段，以便访问指向的节点②。如果指针是空指针，说明到了链表的结尾：链表为空。如果指针非空，就能尝试对 head 调用 compare_exchange_strong()来移除这个节点③。

如果 compare_exchange_strong()成功，你就拥有节点的所有权，可以换出 data 以备返回④。这确保了数据不会仅仅因为其他访问栈的线程碰巧仍然有指向其所在节点的指针而继续存活。然后可以使用原子操作 fetch_add⑥将这个节点的外部计数加到内部计数。如果现在引用计数为 0，那么之前的值(fetch_add 返回的值)是你增加值的负数，这种情况下就可以删除节点。这里需要重点注意的是，增加的值要比外部引用计数少 2⑤；因为当

节点已经从链表中删除时，就要将计数减一，并且不再从这个线程访问节点，所以还要再减一。无论是否删除节点，操作都已经完成，所以可以将数据返回⑦。

如果“比较/交换”③失败，就说明另一个线程在你之前把节点移除了，或者另一个线程添加了一个新的节点到栈中。无论是哪种情况，都需要用“比较/交换”返回的新的 head 值再次启动操作。不过，首先需要递减尝试移除的节点上的引用计数。因为这个线程不会再访问这个节点。如果当前线程是最后一个持有引用(因为另一个线程已经将这个节点从栈上移除了)的线程，那么内部引用计数将会是 1，所以减一的操作将会让计数变为 0。这样，就能在循环⑧之前删除这个节点。

到目前为止，一直使用默认的 std::memory_order_seq_cst 内存顺序用于所有的原子操作。在大多数系统上，这些内存顺序在执行时间和同步开销方面比其他内存顺序更昂贵，并且在某些系统上相当大。现在你已经有了正确的数据结构逻辑，可以考虑放松一些内存顺序的要求；毕竟不希望给栈的用户增加任何不必要的开销。在离开栈，转而设计无锁队列之前，让我们检查一下栈操作，并问自己，我们能否对某些操作使用更宽松的内存顺序，同时仍然获得相同的安全级别?

## 7.2.5 应用内存模型到无锁栈

在修改内存顺序之前，需要检查一下操作并且确定它们之间所需的关系。然后再去确定提供所需关系的最小内存顺序。为了做到这一点，需要在几种不同的场景中从线程的视角查看相关情况。最简单的场景是一个线程将数据项推入栈，然后另一个线程在一段时间过后弹出数据项，因此我们将从这开始。

在这个简单的例子中，涉及到三个重要的数据。第一个是 counted_node_ptr，用于转移数据：head。第二个是 head 引用的 node 结构，第三个是节点所指向的数据项。

执行 push()的线程，会先构造数据项和 node，再设置 head。执行 pop()的线程，会先加载 head 的值，然后在 head 上执行“比较/交换”循环来增加引用计数，再读取 node 结构获取 next 的值。这里可以看到一个必须的关系：next 的值是普通的非原子对象，所以为了安全地读取它，必须在存储(推送线程)和加载(弹出线程)之间有一个“先发生于”（happens-before）关系。因为 push()中唯一的原子操作是 compare_exchange_weak()，并且需要一个释放操作来获得一个线程间的“先发生于”关系，compare_exchange_weak()必须是 std::memory_order_release 或更严格的内存顺序。如果 compare_exchange_weak()调用失败，什么都不会改变，并且继续循环，所以这里使用 std::memory_order_relaxed 就足够了。

```
void push(T const& data)
{
  counted_node_ptr new_node;
  new_node.ptr=new node(data);
  new_node.external_count=1;
  new_node.ptr->next=head.load(std::memory_order_relaxed)
  while(!head.compare_exchange_weak(new_node.ptr->next,new_node,
```

```
    std::memory_order_release,std::memory_order_relaxed));
}
```

pop()的代码呢？为了获得你需要的“先发生于”关系，必须在访问 next 之前使用 std::memory_order_acquire 或更严格的内存顺序的操作。为了访问 next 字段而解引用的指针是 increase_head_count()中使用 compare_exchange_strong()读取的旧值，所以需要把这个内存顺序用在交换成功的时候。和 push()调用一样，当交换失败，循环会继续，所以可以在失败时使用宽松顺序：

```
void increase_head_count(counted_node_ptr& old_counter)
{
  counted_node_ptr new_counter;

  do
  {
    new_counter=old_counter;
    ++new_counter.external_count;
  }
  while(!head.compare_exchange_strong(old_counter,new_counter,
        std::memory_order_acquire,std::memory_order_relaxed));

  old_counter.external_count=new_counter.external_count;
}
```

如果 compare_exchange_strong()调用成功，读取的值将 ptr 字段设置为 old_counter 中 ptr 字段存储的值。由于 push()中的存储操作是个释放操作，并且这个 compare_exchange_strong()操作是个获得操作，所以存储与加载同步，你获得一个“先发生于”关系。因此，push()中存储 ptr 字段的操作在 pop()中对 ptr->next 的访问之前发生，所以操作是安全的。

注意，初始 head.load() 中的内存顺序对分析没有影响，所以可以安全地使用 std::memory_order_relaxed。

接下来考虑 compare_exchange_strong()将 old_head.ptr->next 设置给 head。是否需要这个操作做些什么来保证这个线程的数据完整性吗？当交换成功就会访问 ptr->data，所以需要确保在 push()线程中对 ptr->data 的存储发生在加载之前。但你已经有了这个保证：increase_head_count()中的获得操作确保了在 push()线程中的存储和“比较/交换”间有一个“同步于”关系。因为 push()线程中对数据的存储“先序于”对 head 的存储并且调用 increase_head_count()“先序于”对 ptr->data 的加载，这就有一个“先发生于”关系，即使 pop()中这个“比较/交换”操作使用 std::memory_order_relaxed，还是一切正常。ptr->data 唯一被改变的地方是 swap()调用，并且没有其他线程对同一节点进行操作；这就是“比较/交换”操作的意义所在（译注：因为“比较/交换”操作使得没有别的线程有机会操作这个节点）。

如果 compare_exchange_strong()失败，old_head 的新值直到下一次循环才会触及，并且已经明确了在 increase_head_count()中使用 std::memory_order_acquire 内存顺序就够了，因此这里使用 std::memory_order_relaxed 就好了。

其他线程呢？是否需要更强的内存顺序来保证其他线程的安全？答案是不需要。因为，head 只会因“比较/交换”操作而改变；由于这些是“读-改-写”操作，它们构成了由 push()中“比较/交换”操作领头的释放序列的一部分。因此，即使有很多线程在同一时间对 head 进行修改，push()中的 compare_exchange_weak()也与 increase_head_count()中的 compare_exchange_strong()(它读取存储的值)同步。

现在接近完成：剩下要处理的操作只有用于修改引用计数的 fetch_add()操作。从这个节点返回数据的线程可以继续执行，因为没有其他线程可以修改这个节点的数据。不过，任何没有成功检索到数据的线程知道有另外一个线程修改了节点数据；成功了的线程会使用 swap()来提取引用的数据项。因此为了避免数据竞争，需要保证 swap()发生在 delete 前面。一种简单的做法是让成功返回分支的 fetch_add()使用 std::memory_order_release 内存顺序，并且在再次循环分支的 fetch_add()使用 std::memory_order_qcquire 内存顺序。但还是有点过头了：只有一个线程会 delete(将引用计数设置为 0 的线程)，所以只有这个线程需要获得操作。庆幸的是，因为 fetch_add()是一个“读-改-写”操作，它是释放序列的一部分，所以可以使用一个额外的 load()来实现。如果再次循环分支递减引用计数到 0，它可以使用 std::memory_order_acquire 重新加载引用计数，从而确保必须的“同步于”关系；并且 fetch_add()本身可以使用 std::memory_order_relaxed。使用新版 pop()的最终栈实现如下。

清单 7.13 使用引用计数和宽松原子操作的无锁栈

```
template<typename T>
class lock_free_stack
{
private:
  struct node;
  struct counted_node_ptr
  {
    int external_count;
    node* ptr;
  };

  struct node
  {
    std::shared_ptr<T> data;
    std::atomic<int> internal_count;
    counted_node_ptr next;

    node(T const& data_):
      data(std::make_shared<T>(data_)),
      internal_count(0)
```

```
    {}
  };

  std::atomic<counted_node_ptr> head;

  void increase_head_count(counted_node_ptr& old_counter)
  {
    counted_node_ptr new_counter;

    do
    {
      new_counter=old_counter;
      ++new_counter.external_count;
    }
    while(!head.compare_exchange_strong(old_counter,new_counter,
                                        std::memory_order_acquire,
                                        std::memory_order_relaxed));
    old_counter.external_count=new_counter.external_count;
  }
public:
  ~lock_free_stack()
  {
    while(pop());
  }

  void push(T const& data)
  {
    counted_node_ptr new_node;
    new_node.ptr=new node(data);
    new_node.external_count=1;
    new_node.ptr->next=head.load(std::memory_order_relaxed)
    while(!head.compare_exchange_weak(new_node.ptr->next,new_node,
                                      std::memory_order_release,
                                      std::memory_order_relaxed));
  }
  std::shared_ptr<T> pop()
  {
    counted_node_ptr old_head=
       head.load(std::memory_order_relaxed);
    for(;;)
    {
      increase_head_count(old_head);
      node* const ptr=old_head.ptr;
      if(!ptr)
```

```
      {
        return std::shared_ptr<T>();
      }
      if(head.compare_exchange_strong(old_head,ptr->next,
                                      std::memory_order_relaxed))
      {
        std::shared_ptr<T> res;
        res.swap(ptr->data);

        int const count_increase=old_head.external_count-2;

        if(ptr->internal_count.fetch_add(count_increase,
              std::memory_order_release)==-count_increase)
        {
          delete ptr;
        }

        return res;
      }
      else if(ptr->internal_count.fetch_add(-1,
                  std::memory_order_relaxed)==1)
      {
        ptr->internal_count.load(std::memory_order_acquire);
        delete ptr;
      }
    }
  }
};
```

虽历经磨难，但我们终究到达了终点，实现了一个更好的栈。在深思熟虑后通过使用更多的宽松操作，实现了在不影响正确性的情况下提升了性能。如你所见，pop()的实现现在有 37 行，而等价的栈实现，在清单 6.1 中基于锁的实现只有 8 行，在清单 7.2 中没有内存管理的基本无锁实现更是只有 7 行。当我们继续研究如何编写无锁队列时，将看到一个类似的模式：无锁代码中的许多复杂性来自于管理内存。

## 7.2.6 实现一个无锁的线程安全队列

队列相比栈的挑战有些不同，因为 push()和 pop()在队列中，操作的是数据结构不同的部分，而栈访问的是相同的头结点。因此，对同步的需求不一样。需要确保对一端的修改对另一端的访问是正确可见的。不过清单 6.6 中队列的 try_pop()结构与清单 7.2 中简单无锁栈的 pop()结构相差不大，因此你可以合理地假设无锁代码也不会有太大差异。让我们来看看怎么做。

如果以清单 6.6 作为基础，你需要两个 node 指针：用于表头的 head 和表尾的 tail。由于需要在多线程访问这些指针，因此它们最好是原子的，这样的话就可以不用互斥锁。让我们从一些小改动开始，然后看一下它能给我们带来什么。下面的清单展示了结果。

清单 7.14 一个单生产者，单消费者的无锁队列

```
template<typename T>
class lock_free_queue
{
private:
  struct node
  {
    std::shared_ptr<T> data;
    node* next;

    node():
       next(nullptr)
    {}
  };

  std::atomic<node*> head;
  std::atomic<node*> tail;

  node* pop_head()
  {
    node* const old_head=head.load();
    if(old_head==tail.load())  // 1
    {
      return nullptr;
    }
    head.store(old_head->next);
    return old_head;
  }

public:
  lock_free_queue():
      head(new node),tail(head.load())
  {}

  lock_free_queue(const lock_free_queue& other)=delete;
  lock_free_queue& operator=(const lock_free_queue& other)=delete;

  ~lock_free_queue()
  {
```

```
    while(node* const old_head=head.load())
    {
      head.store(old_head->next);
      delete old_head;
    }
  }

  std::shared_ptr<T> pop()
  {
    node* old_head=pop_head();
    if(!old_head)
    {
      return std::shared_ptr<T>();
    }

    std::shared_ptr<T> const res(old_head->data);  // 2
    delete old_head;
    return res;
  }

  void push(T new_value)
  {
    std::shared_ptr<T> new_data(std::make_shared<T>(new_value));
    node* p=new node;  // 3
    node* const old_tail=tail.load();  // 4
    old_tail->data.swap(new_data);  // 5
    old_tail->next=p;  // 6
    tail.store(p);  // 7
  }
};
```

第一眼瞥过去，这个实现还不赖，如果一次只有一个线程调用 push()，并且只有一个线程调用 pop()的话，这个队列就很完美。在这种情况下，重要的是 push()和 pop()之间的“先发生于”关系，以确保安全地检索数据。对 tail 的存储⑦“同步于”对 tail 的加载①，对前面节点的 data 指针的存储⑤“先序于”存储 tail；并且，加载 tail“先序于”加载 data 指针②，所以对 data 的存储要“先发生于”加载，一切正常。因此，这是一个完美的*单生产者，单消费者*(SPSC, single-producer, single-consume)队列。

但当多个线程并发调用 push()或多个线程并发调用 pop()时，问题就来了。我们先看一下 push()：如果有两个线程并发调用 push()，它们都会分配新节点作为新的虚拟节点③，都会读取到相同的 tail 值④，因此，当设置 data 和 next 指针时，都会更新同一个节点的数据成员⑤⑥。典型的数据竞争！

pop_head()也有类似的问题。如果两个线程并发调用，这两个线程就会读取到相同的 head 值，并且会用相同的 next 指针去覆盖旧值。两个线程现在都认为它们检索到同一个

节点—这会后患无穷。你不仅必须确保只有一个线程在给定的项上使用 pop()，还要保证其他线程可以安全的访问 head 节点中的 next 成员。这正是你在无锁栈 pop()中看到的问题，所以那里的任何方案都可以用在这。

假设 pop()的问题解决了，那么 push()呢？问题在于为了得到 push()和 pop()间的“先发生于”关系，需要在更新 tail 之前设置虚拟节点的数据项。但这意味着对 push()的并发调用会在这些相同的数据项上竞争，因为它们读取了相同的 tail 指针。

**在 push()中处理多线程**

一个选择是在真实节点间添加一个虚拟节点。这样的话，当前 tail 节点唯一需要更新的是 next 指针，因此这个操作可以是原子的。如果一个线程成功地将 next 指针从 nullptr 改成了它的新节点，那么它就成功地添加了指针；否则，就需要重新开始并再次读取 tail。这需要对 pop()做微小的更改，以便丢弃带有空数据指针的节点并再次循环。这里的缺点是每个 pop()调用通常都必须删除两个节点，而且内存分配的数量是原来的两倍。

第二个选择是让 data 指针是原子的，并通过“比较/交换”操作对其进行设置。如果调用成功，这就是你的 tail 节点（译注：说明你是唯一抢到当前 tail 节点的线程），你可以安全的设置它的 next 指针为你的新节点，然后更新 tail。如果“比较/交换”操作失败（因为另一个线程存储了数据），就要接着循环，重新读取 tail 并再次开始。如果 std::shared_ptr<>上的原子操作是无锁的，就大功告成。如果不是的话，就需要一个替代方案。一种可能是让 pop()返回一个 std::unique_ptr<>(毕竟，它是对象的唯一引用)，并将数据作为普通指针存储在队列中。这将允许你把它存储为 std::atomic<T*>，然后它将支持必要的 compare_exchange_strong()调用。如果你正在使用清单 7.12 中的引用计数方案来处理 pop()中的多个线程，那么 push()现在看起来像这样。

清单 7.15 修正 push()的(有问题的)第一次尝试

```
void push(T new_value)
{
  std::unique_ptr<T> new_data(new T(new_value));
  counted_node_ptr new_next;
  new_next.ptr=new node;
  new_next.external_count=1;

  for(;;)
  {
    node* const old_tail=tail.load();  // 1
    T* old_data=nullptr;
    if(old_tail->data.compare_exchange_strong(
      old_data,new_data.get()))  // 2
    {
      old_tail->next=new_next;
      tail.store(new_next.ptr);  // 3
```

```
      new_data.release();
      break;
    }
  }
}
```

使用引用计数方案可以避免这个特殊的竞争，但它不是 push()中唯一的竞争。如果查看清单 7.15 中 push()的修订版，将看到栈中见过的模式：加载一个原子指针①，并且解引用该指针②。同时，另一个线程可能更新指针③，最终导致该节点被释放(在 pop()中)。如果在解引用指针前节点被释放，就会有未定义的行为。一个很有诱惑力的做法是像在 head 中一样给 tail 添加一个外部计数，不过每个节点已经有一个外部计数在队列中前一个节点的 next 指针中。同一个节点有两个外部计数就需要对引用计数方案进行修改，从而避免过早删除节点。你也可以通过计算 node 结构中外部计数器的数量并在每个外部计数器被销毁时减少该数量来解决这个问题（也要把对应的外部计数加到内部计数)。当内部计数是 0，且没有外部计数器时，节点就可以被安全的删除。这是我在 Joe Seigh 的“Atomic Ptr Plus”项目（http://atomic-ptr-plus.sourceforge.net/）中第一次遇到的技术。下面的清单展示了这种方案下 push()的实现。

清单 7.16 使用带有引用计数的 tail，实现无锁队列的 push()

```
template<typename T>
class lock_free_queue
{
private:
  struct node;
  struct counted_node_ptr
  {
    int external_count;
    node* ptr;
  };

  std::atomic<counted_node_ptr> head;
  std::atomic<counted_node_ptr> tail;  // 1

  struct node_counter
  {
    unsigned internal_count:30;
    unsigned external_counters:2;  // 2
  };

  struct node
  {
    std::atomic<T*> data;
    std::atomic<node_counter> count;  // 3
    counted_node_ptr next;
```

```
    node()
    {
      node_counter new_count;
      new_count.internal_count=0;
      new_count.external_counters=2;  // 4
      count.store(new_count);

      next.ptr=nullptr;
      next.external_count=0;
     }
  };

public:
  void push(T new_value)
  {
    std::unique_ptr<T> new_data(new T(new_value));
    counted_node_ptr new_next;
    new_next.ptr=new node;
    new_next.external_count=1;
    counted_node_ptr old_tail=tail.load();

    for(;;)
    {
      increase_external_count(tail,old_tail);  // 5

      T* old_data=nullptr;
      if(old_tail.ptr->data.compare_exchange_strong(  // 6
           old_data,new_data.get()))
      {
        old_tail.ptr->next=new_next;
        old_tail=tail.exchange(new_next);
        free_external_counter(old_tail);  // 7
        new_data.release();
        break;
      }
      old_tail.ptr->release_ref(); // 执行和 5 的反操作，不过是减少内部计
数，和无锁栈差不多
    }
  }
};
```

清单 7.16 中，tail 和 head 一样都是 atomic<counted_node_ptr>类型①，并且 node 结构体中用 count 成员替换了之前的 internal_count③。count 成员变量包括了 internal_count 和额外的 external_counters 成员②。注意，external_counters 只需要

2 比特，因为最多就两个计数器（译注：你最多被队列中前面一个节点的 next 成员引用，以及当你在队尾时被 tail 引用）。通过使用位域，并将 internal_count 指定为 30 比特的值，就能保证计数器的总大小是 32 比特。这为大型内部计数值提供了足够的空间，同时确保了整个结构适合放入 32 位和 64 位机器上的一个机器字中。稍后你将看到，为了避免竞争条件，将这些计数作为单个实体一起更新是很重要的。同时将结构放入一个机器字内，可以使原子操作在许多平台上更有可能是无锁的。

node 初始化时，internal_count 设置为 0，external_counter 设置为 2④，因为每个新节点一旦加入到队列中，都会被 tail 和前一个节点的 next 指针所引用。push()本身与清单 7.15 中的实现类似，除了在解引用从 tail 加载的值（为了在节点的 data 成员上调用 compare_exchange_strong()⑥）之前，调用新的 increase_external_count()函数以增加计数⑤（译注：为了安全解引用，这跟无锁栈中增加 head 的引用是一样的方法），然后后面在旧的尾部值上调用 free_external_counter()⑦（译注：一旦你抢到了尾部节点并把数据放好后，这个尾节点就要挂一个新的 new_next， tail 也会更新成 new_next，tail 就不再有引用指向它，所以你需要释放 tail 对前面尾节点的计数器）。

处理完 push()后，再来看一下 pop()。下面展示的代码将清单 7.12 中 pop()的引用计数逻辑与 7.14 中队列弹出逻辑融合在一起。

清单 7.17 使用带有引用计数的 tail，从无锁队列中弹出节点

```
template<typename T>
class lock_free_queue
{
private:
  struct node
  {
    void release_ref();
  };
public:
  std::unique_ptr<T> pop()
  {
    counted_node_ptr old_head=head.load(std::memory_order_relaxed);  //
 1
    for(;;)
    {
      increase_external_count(head,old_head);  // 2
      node* const ptr=old_head.ptr;
      if(ptr==tail.load().ptr)
      {
        ptr->release_ref();  // 3
        return std::unique_ptr<T>();
      }

      if(head.compare_exchange_strong(old_head,ptr->next))  // 4
```

```
      {
        T* const res=ptr->data.exchange(nullptr);
        free_external_counter(old_head);  // 5
        return std::unique_ptr<T>(res);
      }
      ptr->release_ref();  // 6
    }
  }
};
```

通过在进入循环之前①并且增加外部计数之前②，加载 old_head 的值作为开始。如果 head 节点和 tail 节点相同，说明队列中没有数据，因此可以释放引用计数③并且返回一个空指针。如果队列中还有数据，可以尝试使用 compare_exchange_strong()来索取数据④。与 7.12 中的栈一样，将外部计数和指针做为一个整体做比较；如果任何一个变化了，在释放了引用后⑥，需要再次循环。如果交换成功时，说明已经索取到节点中的数据，在为已弹出节点释放外部计数器后⑤（译注：这个节点已经从队列摘除了，所以前面节点的 next 不会再引用你了，所以必须释放这个计数器），就可以把数据返回给调用函数。一旦两个外部引用计数都被释放，且内部计数降为 0 时，节点本身就可以被删除。处理这一切的引用计数函数展示在清单 7.18，7.19 和 7.20 中。

清单 7.18 在无锁队列中释放一个节点引用

```
template<typename T>
class lock_free_queue
{
private:
  struct node
  {
    void release_ref()
    {
      node_counter old_counter=
        count.load(std::memory_order_relaxed);
      node_counter new_counter;
      do
      {
        new_counter=old_counter;
        --new_counter.internal_count;  // 1
      }
      while(!count.compare_exchange_strong(  // 2
            old_counter,new_counter,
            std::memory_order_acquire,std::memory_order_relaxed));
      // 你可能是最后一个持有引用的线程，这里发生的前提是
free_external_counter 已经执行，因为上面的 while 逻辑一直是减少内部技术，不可能是 0,这和无锁栈中情形差不多，不过这里多了一个外部计数器个数，也是在
free_external_counter 中掌控变成 0
```

```
      if(!new_counter.internal_count &&
         !new_counter.external_counters)
      {
        delete this;  // 3
      }
    }
  };
};
```

与清单 7.12 中 lock_free_stack::pop()实现中的等价代码相比，node::release_ref()的实现只做了轻微的改变。不过，清单 7.12 中的代码只需要处理单个外部计数，所以只需要简单的 fetch_sub，但现在即使你只想更新 internal_count 字段，整个 count 结构也必须原子的更新①。因此，需要一个“比较/交换”循环②。一旦递减了 internal_count，如果内部和外部计数都为 0 时，说明这是最后一个引用，于是可以删除这个节点③。

清单 7.19 获取无锁队列中节点的新引用

```
template<typename T>
class lock_free_queue
{
private:
  static void increase_external_count(
    std::atomic<counted_node_ptr>& counter,
    counted_node_ptr& old_counter)
  {
    counted_node_ptr new_counter;
    do
    {
      new_counter=old_counter;
      ++new_counter.external_count;
    }
    while(!counter.compare_exchange_strong(
      old_counter,new_counter,
      std::memory_order_acquire,std::memory_order_relaxed));

    old_counter.external_count=new_counter.external_count;
  }
};
```

清单 7.19 展示的是另一方面。这次不是释放引用，而是获得一个新的引用，并增加外部计数。increase_external_count()和清单 7.13 中的 increase_head_count()很相似，不同的是 increase_external_count()这里作为静态成员函数，这个函数以要更新的外部计数器作为第一个参数，而不是操作一个固定的计数器。

清单 7.20 无锁队列中释放节点对应的一个外部计数器

```
template<typename T>
class lock_free_queue
{
private:
  static void free_external_counter(counted_node_ptr &old_node_ptr)
  {
    node* const ptr=old_node_ptr.ptr;
    int const count_increase=old_node_ptr.external_count-2;

    node_counter old_counter=
      ptr->count.load(std::memory_order_relaxed);
    node_counter new_counter;
    do
    {
      new_counter=old_counter;
      --new_counter.external_counters;  // 1
      new_counter.internal_count+=count_increase;  // 2
    }
    while(!ptr->count.compare_exchange_strong(  // 3
          old_counter,new_counter,
          std::memory_order_acquire,std::memory_order_relaxed));

    if(!new_counter.internal_count &&
       !new_counter.external_counters)
    {
      delete ptr;  // 4
    }
  }
};
```

与 increase_external_count()对应的是 free_external_counter()。这和清单 7.12 中 lock_free_stack::pop()的等价代码类似，但是做了修改来处理 external_counters 计数。它在整个 count 结构体上使用单个 compare_exchange_strong()来更新两个计数器③，这跟之前在 release_ref()中递减 internal_count 一样。internal_count 值的更新方式和清单 7.12 中一样②，并且 external_counters 的值会减一①。如果两者的值都为 0，就意味着没有更多的引用指向这个节点，所以节点可以被安全的删除④。这需要在单个动作中完成(因此需要“比较/交换”循环)来避免竞争条件。如果它们被分别更新，两个线程可能都认为它们是最后一个引用节点的线程，并且都会删除节点，最终导致未定义的行为。

尽管队列现在能工作并且无竞争，但仍然有性能问题。一旦有一个线程通过成功完成 old_tail.ptr->data 上的 compare_ exchange_strong()来启动 push()操作（7.16 中的⑥)，没有其他线程能执行 push()操作。任何尝试的线程将看到新值而不是 nullptr，这将

导致 compare_exchange_strong()调用失败，然后线程继续循环。这是一个忙等待，它消耗 CPU 周期，但没有任何进展。因此，这实际上是一个锁。第一个 push()调用会阻塞其他线程，直到它完成为止，因此这段代码不再是无锁的。不仅如此，如果有阻塞的线程，操作系统可以给持有互斥锁的线程优先权，但在本例中做不到，所以阻塞的线程将浪费 CPU 周期，直到第一个线程完成。这就需要无锁锦囊中的下一个技巧：等待线程可以帮助执行 push()的线程。

**通过帮助另一个线程实现队列无锁**

为了恢复代码无锁的属性，即使执行 push()的线程停滞了，你也需要找到一种方法让等待的线程继续前进。一种方法是帮停滞的线程执行它的工作。

这种情况下，你确切的知道应该做什么：尾节点的 next 指针需要设置为一个新的虚拟节点，然后 tail 指针本身也必须更新。因为虚拟节点都是一样的，所以不管你的虚拟节点是哪个线程创建的都不重要。如果节点中的 next 指针是原子的，就可以使用 compare_exchange_strong()来设置 next 指针。一旦设置了 next 指针，就可以使用 compare_exchange_weak()循环设置 tail，同时确保它仍然引用相同的原始节点。否则的话，说明其他人已经更新了它，可以停止尝试并再次循环。为了加载 next 指针，这里也需要对 pop()做微小的改动；如下面的清单所示。

清单 7.21 pop()修改为允许帮助 push()端

```
template<typename T>
class lock_free_queue
{
private:
  struct node
  {
    std::atomic<T*> data;
    std::atomic<node_counter> count;
    std::atomic<counted_node_ptr> next;  // 1
  };

public:
  std::unique_ptr<T> pop()
  {
    counted_node_ptr old_head=head.load(std::memory_order_relaxed);
    for(;;)
    {
      increase_external_count(head,old_head);
      node* const ptr=old_head.ptr;
      if(ptr==tail.load().ptr)
      {
        return std::unique_ptr<T>();
      }
```

```
      counted_node_ptr next=ptr->next.load();  // 2
      if(head.compare_exchange_strong(old_head,next))
      {
        T* const res=ptr->data.exchange(nullptr);
        free_external_counter(old_head);
        return std::unique_ptr<T>(res);
      }
      ptr->release_ref();
    }
  }
};
```

之前提过，这里的修改很简单：next 指针现在是原子的①，所以 load②也是原子的。在这个例子中，使用了默认的 memory_order_seq_cst 内存顺序，所以这里可以省略对 load()的显式调用，并依赖加载隐式转换为 counted_node_ptr，不过放入显式的调用可以用来提醒稍后在哪里需要添加显示的内存顺序。

push()的代码比较复杂，展示如下。

清单 7.22 带有帮助的示例 push()用于无锁队列

```
template<typename T>
class lock_free_queue
{
private:
  void set_new_tail(counted_node_ptr &old_tail,  // 1
                    counted_node_ptr const &new_tail)
  {
    node* const current_tail_ptr=old_tail.ptr;
    while(!tail.compare_exchange_weak(old_tail,new_tail) &&  // 2
          old_tail.ptr==current_tail_ptr); // 15
    if(old_tail.ptr==current_tail_ptr)  // 3
      free_external_counter(old_tail);  // 4
    else
      current_tail_ptr->release_ref();  // 5
  }

public:
  void push(T new_value)
  {
    std::unique_ptr<T> new_data(new T(new_value));
    counted_node_ptr new_next;
    new_next.ptr=new node;
    new_next.external_count=1;
    counted_node_ptr old_tail=tail.load();
```

```
    for(;;)
    {
      increase_external_count(tail,old_tail);

      T* old_data=nullptr;
      if(old_tail.ptr->data.compare_exchange_strong(  // 6
         old_data,new_data.get()))
      {
        counted_node_ptr old_next={0};
        if(!old_tail.ptr->next.compare_exchange_strong(  // 7
           old_next,new_next))
        {
          delete new_next.ptr;  // 8
          new_next=old_next;  // 9
        }
        set_new_tail(old_tail, new_next);
        new_data.release();
        break;
      }
      else  // 10
      {
        counted_node_ptr old_next={0};
        if(old_tail.ptr->next.compare_exchange_strong(  // 11
           old_next,new_next))
        {
          old_next=new_next;  // 12
          new_next.ptr=new node;  // 13
        }
        set_new_tail(old_tail, old_next);  // 14
      }
    }
  }
};
```

这与清单 7.16 中的原始 push()类似，但有几个关键的区别。如果设置 data 指针⑥，就需要处理另一个线程帮助你的情况，并且现在有一个 else 子句用来提供这个帮助⑩。

如果设置了节点的 data 指针⑥，这个新版的 push()使用 compare_exchange_strong()更新 next 指针⑦。使用 compare_exchange_strong()是为了避免循环。如果交换失败，说明另外一个线程已经设置了 next 指针，所以也就不需要在开头分配的新节点，然后可以删除这个节点⑧。你还希望使用其他线程设置的 next 值更新 tail⑨（译注：操作有两步，一步是更新 next，另一步是设置 tail，我在⑦没有抢到更新 next 的权利，没关系，我可以继续利用⑨得到的新的虚拟节点继续去抢设置 tail 的操作）。

对 tail 指针的更新已经抽取到 set_new_tail()①。这里使用一个 compare_exchange_weak()循环②来更新 tail，因为如果其他线程尝试 push()一个新节点，external_count 部分可能已经改变，并且你不想弄丢它。但是还需要注意，如果另一个线程已经成功地修改了该值，你就不能替换它；否则，你可能会在队列中出现循环，这是个相当糟糕的主意（译注：你要是不加那个指针相等判断⑮，无脑用 new_tail 去更新 tail 的话，很可能 tail 已经跑到很远了，然后你非把 tail 倒车回来，设置为指向一个更早的 new_tail，这肯定有问题）。因此，如果“比较/交换”操作失败，需要保证加载值的 ptr 部分是相同的。如果 ptr 在循环退出后是相同的③，那么你肯定已经成功设置 tail，所以需要释放旧的外部计数器④。如果 ptr 值不一样，那么另一个线程将释放计数器，所以只需要对该线程持有的单个引用进行释放即可⑤。

如果调用 push()的线程这次通过循环设置 data 指针失败，它可以帮助成功的线程完成更新。首先，尝试更新 next 指针，让其指向该线程分配出来的新节点⑪。如果更新成功，使用这个分配的新节点作为新的 tail 节点⑫，然后需要分配另一个新节点，以便管理将一个项推入队列⑬（译注：你在这里学雷锋做好事把自己的 new_next 设置为别人，你当然要在下一次循环开始前给自己重新安排一个 new_next，就像你刚进入 push 函数里前面设置 new_next 那样，这里由于引用计数没动不需要重新设置，只需要分配个指针）。然后，在再次循环之前，可以尝试通过调用 set_new_tail 来设置尾部节点⑭。

你可能已经注意到，这么短代码有大量的 new 和 delete 调用，因为新节点是在 push()中分配，并且在 pop()中销毁。因此，内存分配器的效率对这个代码的性能有相当大的影响；一个糟糕的分配器可以完全破坏像这样的无锁容器的可扩展属性。选择和实现这类分配器，超出了本书的范围，但是重要的是要记住，识别分配器好坏的唯一方法就是试用它，并对比前后的性能。优化内存分配的常见技术包括在每个线程上有一个单独的内存分配器，并使用空闲链表回收节点，而不是将它们返回给分配器。

到目前为止已经看到足够多的例子；现在，让我们从示例中提取一些编写无锁数据结构的指南。

---

[1] “Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes,” Maged M.Michael, *in PODC ’02: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing* (2002), ISBN 1-58113-485-1.

[2] Maged M. Michael, U.S. Patent and Trademark Office application number 20040107227, “Method for efficient implementation of dynamic lock-free data structures with safe memory reclamation.”

[3] GNU General Public License http://www.gnu.org/licenses/gpl.html.

[4] IBM Statement of Non-Assertion of Named Patents Against OSS, http://www.ibm.com/ibm/licensing/patents/pledgedpatents.pdf.

## 7.3 实现无锁数据结构的指南

如果你已经完成了本章的所有示例，就能领会正确实现无锁代码所涉及的复杂性。如果打算设计自己的数据结构，提供一些需要关注的指南会很有帮助。第 6 章开始时关于并发数据结构的通用指南仍然适用，但这里需要的更多。我从示例中提取了一些有用的指南，可以在设计自己的无锁数据结构时参考它们。

### 7.3.1 指南：使用 std::memory_order_seq_cst 用于原型

std::memory_order_seq_cst 比起其他内存顺序更容易理解，因为所有操作构成一个全序。本章的所有例子，都是从 std::memory_order_seq_cst 开始，并且只有当基本操作正常工作的时候，才放宽内存顺序的约束。从这个意义上讲，使用其他内存顺序就是优化，因此，你需要避免过早地这样做。通常，只有在看到可以对数据结构的内部进行操作的完整代码集时，才能确定哪些操作可以放松。试图乱来的话会让你更加艰难。这个事情很复杂，因为测试通过的话只能说明代码可能没问题，但不能完全保证。除非你有一个算法检查器，可以系统地测试线程可见性的所有组合，这些组合与指定的顺序保证一致（这些东西确实存在），否则只运行代码是不够的。

### 7.3.2 指南：对无锁内存的回收方案

无锁代码最大的困难之一就是内存管理。当其他线程可能仍然引用对象时，必须避免删除它们，但你仍然希望尽快删除对象，从而避免过多的内存消耗。本章介绍了三种技术来确保内存可以被安全的回收：

- 等待直到没有线程访问数据结构时，删除所有等待删除的对象。
- 使用风险指针来标识一个线程正在访问一个特定对象。
- 引用计数对象，直到没有未决的引用时才删除它们。

在所有情况下，核心思想都是使用某种方法来跟踪有多少线程正在访问某个特定对象，并且只在不再从任何地方引用时删除每个对象。在无锁数据结构中，还有许多其他回收内存的方法。例如，这是使用垃圾回收器的理想场景。如果你知道垃圾回收器将在节点不再使用时(而不是之前)释放它们，那么编写算法就会容易得多。

另一种方法是回收节点，并且只在数据结构销毁时完全释放它们。由于节点被重用，内存永远不会失效，因此避免未定义行为的一些困难就消失了。不利的一面是，另一个问题变得更加普遍。这就是所谓的“ABA 问题”。

### 7.3.3 指南：小心 ABA 问题

任何基于“比较/交换”的算法都要小心 ABA 问题。它是这样的：

1. 线程 1 读取原子变量 x，并且发现其值是 A。

2. 线程 1 基于这个值进行一些操作，比如，解引用(如果它是个指针)，或做查找，或者别的什么操作。
3. 线程 1 被操作系统暂停。
4. 另一个线程对 x 执行一些操作，将它的值改为 B。
5. 然后某个线程更改与值 A 相关联的数据，使线程 1 持有的值不再有效。这可能与释放指向的内存或更改关联值一样激烈。
6. 然后某个线程根据这个新数据将 x 修改回 A。如果这是一个指针，它可能是一个碰巧与旧对象共享相同地址的新对象。
7. 线程 1 继续对 x 执行比较/交换操作，与 A 做比较。比较/交换会成功(因为值确实是 A)，但这是错误的 A 值。因为最初在步骤 2 中读取的数据不再有效，但是线程 1 无从得知，并将破坏数据结构。

这里的算法都不会遇到这个问题，但很容易编写碰到这个问题的无锁算法。避免这个问题最常见的方法是在变量 x 旁边包含一个 ABA 计数器，然后在 x 带上计数器的组合结构（作为一个单元）上执行比较/交换操作。每次值被替换时，计数器就会增加，所以即使 x 有相同的值，如果另一个线程修改了 x，比较/交换将失败。

ABA 问题在使用空闲链表或回收节点而不是将它们返回给分配器的算法中尤为普遍。

### 7.3.4 指南：识别忙-等待循环并帮助其他线程

在最后队列的例子中，你看到了线程在执行 push 操作时，必须等待另一个也在执行 push 的线程完成它的操作才能继续。如果放任不管的话，这将是一个“忙-等待”循环，等待线程在无法继续时会浪费 CPU 时间。如果最终是“忙-等待”循环，你实际上就有一个阻塞操作，这跟使用互斥锁也差不多。通过修改算法，如果在原始线程完成操作之前就计划运行等待线程，那么等待线程可以执行未完成的步骤，这样你就可以去掉“忙-等待”，并且操作也不再阻塞。队列示例中需要将一个数据成员更改为一个原子变量，而不是使用非原子变量，并且使用“比较/交换”操作来设置它；不过在更复杂的数据结构中，需要更多修改。

## 总结

继第 6 章基于锁的数据结构之后，本章描述了各种无锁数据结构的简单实现，和之前一样，从栈和队列开始。你看到了必须注意原子操作上的内存顺序，以确保没有数据竞争，并且每个线程看到的数据结构视图是一致的。你也看到了对于无锁数据结构，内存管理比基于锁的数据结构要困难得多，并研究了几种处理它的机制。你还看到了如何通过帮助你正在等待的线程完成它的操作来避免创建等待循环。

设计无锁数据结构是一项困难的任务，而且很容易出错，但这些数据结构具有的可扩展属性在某些情况下非常重要。希望通过遵循本章中的示例并阅读指南，可以更好地设计自己的无锁数据结构，从研究论文实现一个无锁数据结构，或者找出你的前同事在离开公司之前写的无锁数据结构中的 bug。

线程之间不管在哪共享数据，都需要考虑数所使用的数据结构以及线程之间如何同步数据。通过设计并发数据结构，可以将责任封装在数据结构本身，这样的话，代码的其余部分就可以专注于它试图用数据执行的任务，而不是数据同步。在第 8 章中，当我们从并发数据结构过渡到一般的并发代码时，将看到它的实际应用。此外并行算法使用多线程来提升它们的性能，当算法需要工作线程共享数据时，并发数据结构的选择至关重要。

# 第 8 章 设计并发代码

**本章主要内容**

- 线程间划分数据的技术
- 影响并发代码性能的因素
- 性能因素是如何影响数据结构的设计
- 多线程代码中的异常安全
- 可扩展性
- 几个并行算法的示例实现

前面大部分章节都集中在 C++工具箱中用于编写并发代码的工具上。第 6、7 章中考察了如何使用这些工具来设计基本的数据结构，这些数据结构能够安全地被多线程并发访问。就像一个木匠要做一个橱柜或桌子，需要知道的不只是如何制作铰链或接头，设计并发代码也要比设计和使用基本数据结构知道的更多。你现在需要考虑更广泛的环境，以便构建更大的结构来执行有用的工作。我将使用 C++标准库算法的多线程实现作为例子，不过同样的原则也适用所有规模的应用程序。

和任何编程项目一样，仔细考虑并发代码的设计至关重要。但对于多线程代码，需要考虑的因素比顺序代码更多。不仅要考虑一般性因素，例如封装，耦合和内聚(这些在很多软件设计书籍中有充分描述)，还要考虑共享哪些数据，如何同步访问数据，哪些线程需要等待其他线程完成某种操作，等等。

本章我们将重点讨论这些问题，从高层次(但也是最根本的)的考虑使用多少线程，哪些代码在哪个线程执行，以及这如何影响代码的清晰性，到低层次细节：如何为最优性能构建共享数据。

让我们从如何在线程间划分工作的技术开始。

## 8.1 线程间划分工作的技术

想象一下，你被要求建造一座房子。为了完成工作，需要挖地基、砌墙、安装管道、接入电线等等。理论上，只要经过足够的培训你全都可以自己干，但是这可能花费很长时间，并且需要不断的切换任务。或者，你可以雇佣一些人来帮忙。现在你需要决定雇多少人，以及他们需要什么样的技能。比如，你可以雇几个通用技能的人，这几个人什么都干。现在你还得不断的切换任务，不过因为现在有更多跟你一样的人，因此可以更快的完成。

或者，你可以雇佣一个专业团队：瓦工，木匠，电工和水管工。你的专业人员只做他擅长的，所以当没有管道施工时，水管工会坐在那喝茶或咖啡。事情还是能够比以前完成的更快，因为人多，并且电工接通厨房电源的时候，水管工可以投入厕所的工作，但是当没有更多工作给特定专业人员时，会有更多等待。即使有空闲时间，你可能还是能感觉到专业团队要比雇佣一帮什么都能干的杂工快。因为专业人员不需要频繁更换工具，并且每

个人都比一般人员更快的完成任务。当然事实是否如此取决于特定的情况——必须试一下才知道。

即使雇佣专业人员，你仍然可以选择不同工种的人数。比如，可能砖瓦工的数量需要超过电工。同样，如果需要建造不止一座房子，团队的组成和全局效率可能也不一样。即使对任何要建的房子水管工都没有太多的工作，只要你同时建造许多房子，你依旧能让他总是处于忙碌的状态。同样，当他们没有工作可做时，如果你不用给他们钱，那你可能有能力整体上负担起一个更大的团队，即使在任何时候只让相同数量的人在工作。

好了，建造的例子说的够多了；这一切跟线程有什么关系？好吧，这些问题也会发生在线程上。你需要决定使用多少个线程，并且这些线程应该完成什么任务。还需要决定是使用“通才”线程什么都做，还是使用“专才”线程做好一件事，或将两种方法混合。不管使用并发的动因是什么你需要做这些选择，并且如何做选择会对性能和代码的清晰性产生关键的影响。因此理解这些选项至关重要，这样在设计应用程序的结构的时候就能做出明智的决定。本节中，将考察几个划分任务的技术，先从线程间划分数据开始。

### 8.1.1 处理开始前在线程间划分数据

最容易并行的算法，是诸如 std::for_each 这样的简单算法，它会对一个数据集中每个元素执行一个操作。为了让算法并行，你可以把每个元素指派到其中一个处理线程。如何划分才能获得最佳的性能，取决于数据结构实现的细节，你将在本章后面当我们着手性能问题的时候看到。

最简单的数据分配方法是第一组 N 个元素分配一个线程，下一组 N 个元素再分配一个线程，以此类推，如图 8.1 所示，但也可以使用其他分配模式。不管数据怎么分，每个线程都会对分配给它的元素进行操作，不会和其他线程通信，直到处理完成。

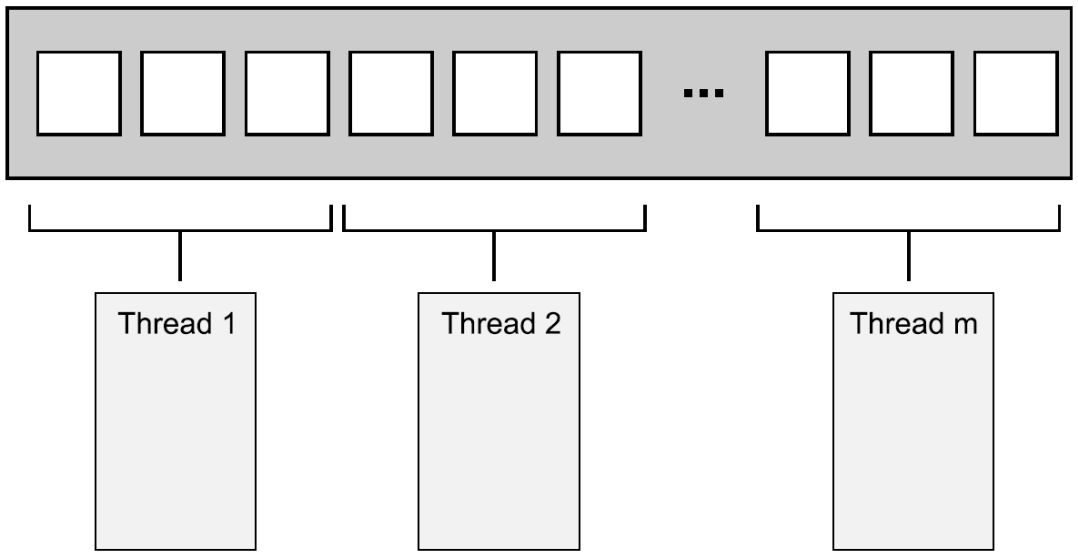


图 8.1 在线程间分发连续的数据块

使用过消息传递接口 (MPI, Message Passing Interface，http://www.mpi-forum.org/)或者 OpenMP(http://www.openmp.org/)框架的人对这个结构一定很熟悉：一个任务被分割成多个并发任务，工作线程独立的运行这些任务，结果会在一个最终的归约（reduction）步骤组合起来。这是在 2.4 节中的 accumulate 例子中使用过的方法；这个例子中，所有并行任务和最终的归约步骤都是累加操作。对于简单的 for_each 来说，最终的步骤是个空操作，因为没有结果需要归约。

把最后一步认定为归约是十分重要的；像清单 2.9 中那种初级的实现，会把归约当成一个最终串行的步骤来执行。但这一步通常也能并行化；accumulate 是一个归约操作，所以清单 2.9 能在当线程数量大于一个线程上最小处理项数的时候修改成递归调用它自己。或者工作线程在完成它自己的任务时执行一些归约步骤而不是每次都生成新的线程。

虽然这个技术很强大，但并不是什么东西都能用。有时候数据不能预先的整齐地划分，因为只有对数据进行处理后，才能进行明确的划分。像快速排序这种递归算法尤其明显；因此它们需要一种不同的方法。

## 8.1.2 递归划分数据

快速排序算法有两个基本的步骤：将数据按照最终排序顺序划分为主元之前或之后的两半，然后递归的对这两半排序。这里不能通过预先对数据划分来并行，因为只有在处理后才知道要分到哪一半。如果要对这种算法进行并行化，你需要利用递归的性质。每一级的递归都会多次调用 quick_sort 函数，因为必须对主元前后的元素都要排序。这些递归调用是完全独立的，因为它访问的是不同的数据集，所以是并发执行的首选。图 8.2 展示了这样的递归划分。

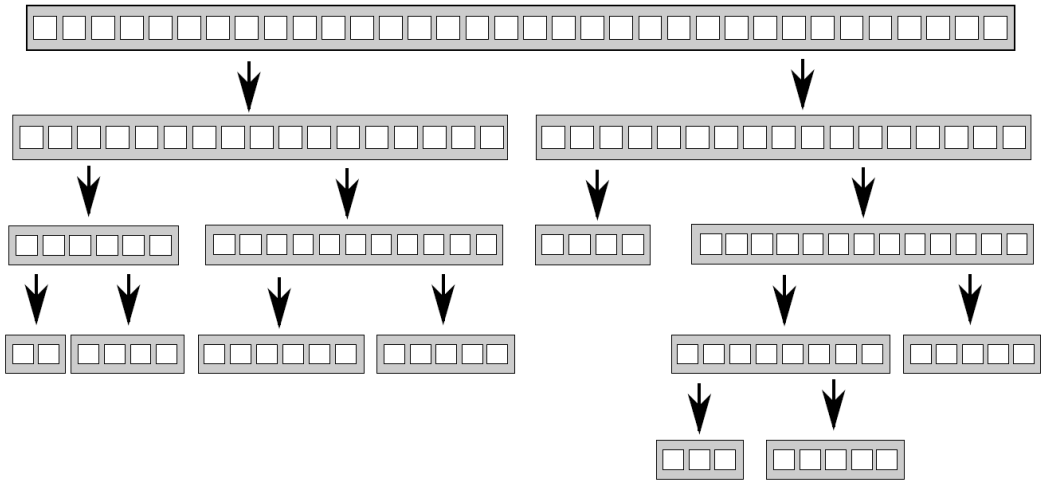

图 8.2 递归划分数据

第 4 章中已经见过这种实现，不是对较大值数据块和较小值数据块执行两个递归的调用，而是使用 std::async()在每个阶段生成较小值数据块的异步任务。通过使用

std::async()，你让 C++线程库决定何时在一个新线程上执行任务，以及何时同步地执行任务。

这很重要：如果你对一个很大的数据集进行排序时，每次递归都产生一个新线程，会导致产生大量的线程。当我们考察性能的时候你将看到，如果有太多的线程存在，可能会拖慢应用程序。如果数据集过于庞大，也可能耗尽线程。像这样用递归的方式划分全局任务是个不错的主意；你只需要看紧线程的数量。std::async()可以处理这种简单的情况，但它不是唯一选择。

另一种选择是使用 std::thread::hardware_concurrency()函数来确定线程的数量，就像清单 2.9 中并行版 accumulate()一样。然后，不同于为递归调用启动一个新线程，可以将需要排序的数据块推到线程安全的栈上，如第 6、7 章中的栈。如果一个线程无所事事，要么是已经完成了对自己数据块的处理，要么是在等待一个需要排序的数据块；它可以从栈上取一个数据块，并对其排序。

下面的清单展示了使用这种技术的示例实现。与大多数示例一样，只是为了演示想法，而不是产品级代码。如果你使用的是 C++17 编译器并且库支持的话，你最好使用标准库提供的并行算法，第 10 章会覆盖这个话题。

清单 8.1 使用待排序数据块栈的并行快速排序

```
template<typename T>
struct sorter  // 1
{
  struct chunk_to_sort
  {
    std::list<T> data;
    std::promise<std::list<T> > promise;
  };

  thread_safe_stack<chunk_to_sort> chunks;  // 2
  std::vector<std::thread> threads;  // 3
  unsigned const max_thread_count;
  std::atomic<bool> end_of_data;

  sorter():
    max_thread_count(std::thread::hardware_concurrency()-1),
    end_of_data(false)
  {}

  ~sorter()  // 4
  {
    end_of_data=true;  // 5

    for(unsigned i=0;i<threads.size();++i)
```

```
  {
    threads[i].join();  // 6
  }
}

void try_sort_chunk()
{
  boost::shared_ptr<chunk_to_sort > chunk=chunks.pop();  // 7
  if(chunk)
  {
    sort_chunk(chunk);  // 8
  }
}

std::list<T> do_sort(std::list<T>& chunk_data)  // 9
{
  if(chunk_data.empty())
  {
    return chunk_data;
  }

  std::list<T> result;
  result.splice(result.begin(),chunk_data,chunk_data.begin());
  T const& partition_val=*result.begin();

  typename std::list<T>::iterator divide_point=  // 10
     std::partition(chunk_data.begin(),chunk_data.end(),
      [&](T const& val){return val<partition_val;});

  chunk_to_sort new_lower_chunk;
  new_lower_chunk.data.splice(new_lower_chunk.data.end(),
     chunk_data,chunk_data.begin(),
     divide_point);

  std::future<std::list<T> > new_lower=
    new_lower_chunk.promise.get_future();
  chunks.push(std::move(new_lower_chunk));  // 11
  if(threads.size()<max_thread_count)  // 12
  {
    threads.push_back(std::thread(&sorter<T>::sort_thread,this));
  }

  std::list<T> new_higher(do_sort(chunk_data));
```

```
    result.splice(result.end(),new_higher);
    while(new_lower.wait_for(std::chrono::seconds(0)) !=
       std::future_status::ready)  // 13
    {
      try_sort_chunk();  // 14
    }

    result.splice(result.begin(),new_lower.get());
    return result;
  }

  void sort_chunk(boost::shared_ptr<chunk_to_sort> const& chunk)
  {
    chunk->promise.set_value(do_sort(chunk->data));  // 15
  }

  void sort_thread()
  {
    while(!end_of_data)  // 16
    {
      try_sort_chunk();  // 17
      std::this_thread::yield();  // 18
    }
  }
};

template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)  // 19
{
  if(input.empty())
  {
    return input;
  }
  sorter<T> s;

  return s.do_sort(input);  // 20
}
```

这里，parallel_quick_sort 函数⑲委托了大部分功能给 sorter 类①，这个类提供了简单的方法来组织栈上没有排序的数据块②，以及线程集③。主要的工作在 do_sort 成员函数完成⑨，它对数据做了分区⑩。不同于对每一个数据块产生一个新的线程，这次会把数据块推入栈上⑪；并在有剩余处理器⑫的时候，产生新的线程。因为较小值数据块可能被另一个线程处理，于是必须等待它就绪⑬。为了让事情顺利完成(当只有一个线程或其他所有线程都在忙碌的时候)，当你等待的时候⑭，你会尝试在当前线程处理栈上的数

据。try_sort_chunk 从栈上弹出一个数据块⑦，并对它排序⑧，存储结果到 promise 中，准备好被投递这个数据块到栈上的线程捡起⑮（译注：也就是在⑬探测到 ready 状态）。

当 end_of_data 没有被设置时⑯，新生成的线程位于一个循环中尝试排序栈上的数据块⑰。在检查间隙，它们会出让出执行权给其他线程⑱，以便给它们机会将更多的工作放入栈中，这里的代码依赖于 sorter 类④的析构函数来清理这些线程。当所有数据都排好序，do_sort 将会返回(即使工作线程仍在运行)，因此主线程将会从 parallel_quick_sort⑳中返回，接着销毁 sorter 对象。sorter 的析构函数会设置 end_of_data 标记⑤，并等待所有线程结束⑥。设置标记将终止线程函数的循环⑯。

有了这个方案，不用为 spawn_task 产生不受限制的线程所困扰，并且不用再依赖 C++线程库来为你选择线程的数量（就像 std::async()所做的那样）。相反通过 std::thread::hardware_concurrency()的值限制线程的数量，这样就能避免过度的任务切换。然而，有另一个潜在的问题：线程管理和线程间通讯给代码增加了很多复杂性。同样，尽管线程处理独立的数据元素，但它们都要访问栈来添加数据块或者弹出数据块。这个严重的竞争会降低性能，即使你使用无锁（因而非阻塞）栈，原因很快会看到。

这种方法是一个特化版本的*线程池*（thread pool）——一系列线程从一个待处理工作的列表获取工作，完成工作，然后回到列表获取更多任务。线程池的一些潜在问题(包括对工作列表的竞争)以及解决方法将在第 9 章介绍。将应用程序扩展到多处理器的问题将在本章后面更详细地讨论(详见 8.2.1)。

在考察划分数据的方法时，不管是处理前划分数据还是递归划分都需要事先保证数据本身是固定的。但情况并非总是如此；如果数据是动态生成，或是来自外部输入，这些方法就没法工作了。这种情况下，基于任务类型的划分方式比基于数据的划分方式更有意义。

## 8.1.3 通过任务类型划分工作

在线程间通过分配不同的数据块给每个线程(不管提前划分还是递归划分)来划分工作仍然停留在一个假设：线程在每个数据块上做相同的工作。另一种划分工作的选择是让线程做专门的工作，也就是每个线程执行不同的任务，就像水管工和电工在建造一所屋子的时候执行不同的任务。线程可能会也可能不会工作在相同的数据上，即便是相同的数据，那也是为了不同的目的。

这种分工起因于在并发中分离关注点；每个线程都有不同的任务，独立于其他线程执行。其他线程偶尔会向某个线程传递数据，或是触发事件让它处理；不过通常，每个线程只专注做好一件事情。就其本身而言，这是良好的设计：每段代码都应该有单一的职责。

**根据任务类型划分工作来分离关注点**

在有多个任务需要持续运行一段时间的地方，或在需要及时处理到来的事件(比如用户按键或进来的网络数据)的地方（甚至当其他任务还在运行的时候），单线程应用程序必须处理与单一职责原则的冲突。在单线程的世界中，你最终需要手工编写代码执行一点 A 任

务，执行一点 B 任务，检查按键，检查进来的网络包，然后再循环回去，执行另一点任务 A。这将会使得任务 A 复杂化，因为需要存储它的状态，并且定期返回控制到主线程。如果你添加太多的任务到循环，程序将会慢很多；并且用户可能会发现需要很长时间才会响应按键。我确定你已经见过以这种极端形式运转的应用程序：你让它完成一些任务，然后发现界面冻结了，直到任务完成。

这就是线程大显身手的地方。如果你让每个任务运行在独立的线程，剩下的操作系统会帮你处理好。在任务 A 的代码中，你可以专注于执行任务，而不用在执行之前担心保存状态以及返回到主循环或者需要花费多长时间。操作系统会自动保存状态，并且在合适的时候切换到任务 B 或 C。如果目标系统有多核或多处理器，任务 A 和任务 B 很可能真正的并发运行。处理按键或网络包的代码现在就能及时运行了。这是个多赢的局面：用户得到了及时的响应；还有你，拥有更简单的代码因为每个线程可以专注于和它职责直接相关的操作，而不用将控制分支和用户交互混杂在一起。

这听起来令人心驰神往。真的是那样吗？一切取决于细节。如果每个任务都是独立的，并且线程间不需要互相通信，那还真就这么简单。不幸的是，现实总是残酷的。后台那些优雅的任务经常做一些用户请求的事情，并且当它们完成的时候，需要以某种方式更新用户界面，从而让用户知晓。或者，用户可能想要取消任务，就需要用户界面以某种方式发送一条消息给后台任务，让它停止运行。这两种情况都需要认真考虑、设计、以及适当的同步，但关注点仍然是分离的。用户界面线程还是处理用户界面，但当其他线程请求它要做什么的时候，用户界面线程需要更新用户界面。同样，运行后台任务的线程仍然关注该任务所需的操作；只是碰巧其中的一个是“允许任务被另一个线程停止”。无论哪种情况，后台线程都不关心请求来自哪里，它们只关注那种本来就是给它们或者和它们责任直接相关的请求。

多线程下分离关注点有两大危险。第一个是分离了错误的关注点。要检查的症状是线程之间共享了大量数据，或者不同的线程最终相互等待，这两种情况都归结为线程间有太多通信。如果发生这种情况，那就值得考虑一下需要这么多通信的原因。如果所有的通信都和同一个问题相关，很可能它应该是单个线程的核心职责，因此需要把它从所有引用它的线程中抽出来。或者如果两个线程相互频繁通信，但跟其他线程又不怎么通信的话，也许它们应该合并成一个线程。

当根据任务类型对线程间的任务进行划分时，没必要限制自己在完全隔离的情况。如果多个输入数据集需要应用相同的操作序列，你可以拆分工作，这样每个线程执行全局序列中的一个阶段。

**线程间划分任务序列**

如果任务会应用相同操作序列到许多独立的数据项，就可以使用*流水线*(pipeline)来利用系统的可用并发。好比一个物理管道：数据流从管道一端进入，进行一系列操作后，从管道另一端出去。

使用这种方式划分工作，可以为流水线中的每一阶段创建一个独立的线程——序列上的每个操作对应一个线程。当操作完成，数据元素会放入一个队列中，以供下一阶段的线

程拾取。这就允许当流水线上的第二个线程正在处理第一个元素时，执行序列中第一个操作的线程开始处理下一个数据元素。

这就是 8.1.1 节描述的线程间划分数据的一种替代方案，这种方式适合于当操作启动后，对输入数据本身知之甚少的情况。例如，数据可能来自网络，或者序列中的第一个操作可能是通过扫描文件系统来确定要处理的文件。

流水线对于序列中每个操作都很耗时的情况也很适合；通过在线程间划分任务而不是数据，你会改变性能表现。假设有 20 个数据项，在 4 个核上处理，并且每一个数据项需要 4 个步骤，每一步需要 3 秒来完成。如果在四个线程间划分数据，那么每个线程上就有五个数据项要处理。假设没有其他的处理会影响计时，在 12 秒后有 4 个数据项处理完成，24 秒后 8 个数据项处理完成，以此类推。所有 20 个数据项将在 1 分钟后完成。如果使用流水线，事情就不一样了，四步中的每一步可以交给一个处理核，现在第一个数据项需要被每一个核处理，所以它仍然会消耗全部的 12 秒。确实，在 12 秒后你只得到一个处理过的数据项，这还不如数据划分。不过，一旦流水线装填好了，事情的进展就有点不同；在第一个核处理了第一个数据项后，数据项就会交给下一个核，所以一旦最后那个核处理了第一个数据项后，它可以对第二个数据项进行处理。现在每 3 秒就可以得到一个已处理的数据项，而不是每隔 12 秒完成 4 个数据项。

现在整批处理的总时间更长了，这是因为最后一个核在开始处理第一个元素时，需要等待 9 秒。但在某些情况下，更平滑、更规律的处理是有益的。例如，考虑一个观看高清数字视频的系统。为了让视频可以观看，至少要保证每秒 25 帧，当然越多越好。另外，这些帧需要均匀分布，才能给观看者留有连续播放的印象；如果要暂停一秒的话，然后播放 100 帧，接着暂停零一秒，再播放另外 100 帧，那么应用程序即使可以每秒解码 100 帧仍然没什么用。另一方面，观看者乐于接受在开始观看视频的时候延迟几秒。这种情况下，并行使用流水线以稳定的速率输出帧更可取。

在了解了各种多线程间划分工作的技术后，让我们来看一下哪些因素会影响多线程系统的性能，以及这些因素是如何影响技术选型的。

## 8.2 影响并发代码性能的因素

如果你正在使用并发在多处理器系统上提升代码的性能，你需要了解一下有哪些因素会影响性能。即使已经使用多线程来分离关注点，还需要确保它不会对性能造成负面影响。如果你的应用程序在新的 16 核机器上跑的比老式的单核机器还慢，用户肯定得骂娘。

你马上就会看到，很多因素会影响多线程代码的性能——甚至像修改每个线程处理的数据元素这样简单的事情(其他条件一样)对性能都有戏剧性的影响。言归正传，让我们来看一下这些因素吧，先从明显的开始：目标系统有多少个处理器？

### 8.2.1 多少个处理器？

处理器个数（和结构）是影响多线程应用性能的第一个重要因素，也是至关重要的一个因素。某些情况下，你对目标硬件了然于胸，设计的时候也会把这些铭记于心，并在目标系统或等同环境上实测。如果是这样的话，你是幸运的；一般来讲，那对你是个奢望。你可能在一个相似的系统上进行开发，但又有显著区别。例如，你可能会在一个双核或四核的系统上做开发，不过你用户的系统可能就只有一个多核处理器(有任意数量的核)，或多个单核处理器，或甚至是多个多核处理器。在这些不同的情况下，并发程序的行为和性能特征会有很大的不同，所以需要仔细考虑有什么影响,可能的话最好做一些测试。

大体上，一个 16 核的处理器和 4 个四核或 16 个单核处理器相同：每种情况下，系统都能并发运行 16 个线程。如果你想充分利用它们，你的应用程序最少要有 16 个线程。如果不足 16 个，会有处理器处于空闲状态(除非系统同时也运行其他应用，不过我们暂时忽略这种可能性)。另一方面，当有多于 16 个线程可以运行的时候(没有阻塞或等待)，应用将会浪费处理器时间用于切换线程，如第 1 章所述。这种情况发生时，这就是所谓的*超订*(oversubscription)。

为了让应用的线程数量能够和硬件支持的并发线程数量保持一致扩展，C++标准线程库提供了 std::thread::hardware_concurrency()。你已经看到过这个函数是如何让线程数量扩展到硬件线程数量。

直接使用 std::thread::hardware_concurrency()需要注意；因为代码不会考虑运行在系统上的其他线程，除非你显式的分享了那个信息。所以在最坏的情况下，如果同一时间多个线程同时调用一个使用 std::thread::hardware_concurrency()的函数来扩展线程数量，这将导致巨大的超订。而 std::async()就能避免这个问题，因为标准库知道所有的调用，会对它们进行适当的安排。小心的使用线程池也可以避免这个问题。

不过，即使已经考虑了应用中运行的所有线程，程序还是要受制于同时运行的其他应用。尽管单用户系统中同时使用多个 CPU 密集型应用程序很少见，但在某些领域这种情况更常见。虽然应对这种场景的系统通常提供相关机制允许应用选择线程数量，但这种机制已经超出 C++标准的范围。一种选择是使用像 std::async()这样的设施，在选择线程数量的时候考虑所有应用程序运行的异步任务的总数。另一个选择是，限制每个应用可以使用的处理核数。尽管没有保证，但我会期望，在这些平台上，这种限制能反映到 std::thread::hardware_concurrency()的返回值上。如果需要处理这种情况，请查阅你的系统文档看是否有什么方法可用。

这种情况的另一个转折是：对一个问题的理想算法可能取决于相对处理单元的数量而言的问题规模。如果你有一个有很多的处理单元的大规模并行系统，总体上执行更多操作的算法可能比执行少量操作的算法更快的结束；因为每个处理器只需要执行很少的操作。

随着处理器数量的增加，另一个问题的可能性和对性能的影响也是如此：多个处理器尝试访问相同的数据。

## 8.2.2 数据竞争与乒乓缓存

当两个线程在不同处理器上并发的执行，并且读取同一数据，这通常不会出现问题；因为数据将会拷贝到每个线程的缓存中，并且两个处理器都可以继续执行。不过，如果有一个线程修改了数据，这个修改就需要传播到其他核的缓存中去，这要耗费一定的时间。取决于两个线程上操作的性质，以及操作使用的内存顺序，这个修改可能会让第二个处理器在它的处理轨迹上停下来，等待这个修改通过内存硬件传播过来。就 CPU 指令而言，尽管确切的时间主要取决于硬件的物理结构，但这可能是一个非常慢的操作，相当于数百条单独的指令。

考虑下面这个简单的代码片段：

```
std::atomic<unsigned long> counter(0);
void processing_loop()
{
  while(counter.fetch_add(1,std::memory_order_relaxed)<100000000)
  {
    do_something();
  }
}
```

counter 是全局变量，所以任何调用 processing_loop()的线程都是修改同一个变量。因此，对每次递增操作，处理器必须确保最新的 counter 拷贝在它的缓存中，修改这个值，然后发布到其他处理器上。即使使用 std::memory_order_relaxed，因而编译器不用同步任何数据，但 fetch_add 是一个“读-改-写”操作，因此需要获取最新的值。如果另一个线程在另一个处理器上运行相同的代码，那么 counter 的数据需要在两个处理器及他们对应缓存之间来回传递，这样的话，当它执行递增操作时，每个处理器都有最新的 counter 值。如果 do_something()足够短，或有很多处理器运行这个代码，这些处理器可能会发现他们在互相等待；一个处理器准备更新这个值，但另一个处理器正在修改这个值，所以它必须等待，直到第二个处理器完成更新并且这个修改已经传播出来。这种情况称为*高竞争*(high contention)。如果处理器很少需要互相等待，你就处于*低竞争*(low contention)。

在这样的循环中，counter 的数据将在缓存间来回传递很多次。这叫做*乒乓缓存*(cache ping-pong)，它将严重影响应用程序的性能。如果一个处理器因为等待缓存转移而哑火，这时它啥也做不了，哪怕有其他线程在等待，可以做有用的工作。所以对整个应用来讲，这是一个坏消息。

你可能会想，这种情况不会发生在你身上；因为你没有这种循环。你确定吗？互斥锁呢？如果需要在循环中获取一个互斥锁，那么从数据访问的角度看，你的代码和之前代码是类似的。为了锁住互斥锁，另一个线程必须将组成互斥锁的数据转移到它所在的处理器上，然后修改它。完事之后，解锁操作将会再次对互斥锁进行修改，同时互斥锁数据将会转移到下一个获取互斥锁的线程上去。这个转移时间是在第二个线程需要等待第一个线程释放互斥锁的时间基础上额外增加的：

```
std::mutex m;
my_data data;
void processing_loop_with_mutex()
{
  while(true)
  {
    std::lock_guard<std::mutex> lk(m);
    if(done_processing(data)) break;
  }
}
```

现在来看下最糟糕的部分：如果数据和互斥锁被多个线程访问，然后你添加更多的核和处理器到系统中，你很可能看到高竞争，以及一个处理器需要等待其他处理器的情况。如果你使用多线程来加速数据处理，线程间会对数据进行竞争，从而也会对同一个互斥锁进行竞争。线程越多，它们在同一时间尝试获取互斥锁的，或者访问原子变量等等的可能性也更大。

互斥锁竞争的影响通常不同于原子操作竞争的影响，这是因为互斥锁是在操作系统级别而不是处理器级别将线程串行化。如果有足够的可运行线程，当某个线程在等待互斥锁时，操作系统会调度另一个线程运行，然而处理器暂停则会阻止任何线程在它上面运行。但它仍会影响这些竞争互斥锁的线程的性能；毕竟，它们一次只能运行一个线程。

回顾第 3 章，你已经见识了一个很少更新的数据结构可以被一个单一写者，多个读者互斥锁(详见 3.3.2 节)保护。如果工作负载不理想，乒乓缓存效应会抵消互斥锁带来的收益，因为所有访问数据(即使是读者线程)的线程仍然需要修改互斥锁本身。随着访问数据的处理器数量增加，对互斥锁的竞争就会增加，这样，包含互斥锁的缓存行必须在核之间进行转移，从而潜在增加获取和释放锁的时间到不可取的水平。有些技术通过将互斥锁底层组件分散开，跨越多个缓存行来改善这个问题，但是除非你实现自己的互斥锁，否则你只能受制于系统提供什么样的实现。

如果乒乓缓存这么不好，那怎么规避它呢？本章后面你将看到，答案和改善并发的通用指南结合的很好：尽可能减少两个线程对同一个内存位置的竞争。

尽管事情没那么简单，事情从来也没简单过。即使特定内存位置总是只被一个线程所访问，可能还是会因为*伪共享*(false sharing)的影响而遇到乒乓缓存。

## 8.2.3 伪共享

处理器缓存通常不会和单个内存位置打交道，取而代之，打交道的对象是称为*缓存行*(cache lines)的内存块。这些内存块的大小通常是 32 或 64 字节，但确切的细节依赖于所使用的特定处理器模型。因为缓存硬件只和缓存行大小的内存块打交道，所以相邻内存位置上的小数据项将属于同一缓存行。有时这样是有益的：当线程要访问的一组数据是在同一缓存行中，对于应用的性能来说就要好于同一组数据分布在多个缓存行中。不过，如果

在同一缓存行存储的是无关数据，并且需要被不同线程访问，这可能成为性能问题的主要原因。

假设你有一个 int 类型的数组，并且有一组线程，它们分别访问数组中属于自己的条目，但访问会不断重复，其中包括更新操作。因为 int 类型的大小通常要小于一个缓存行，同一个缓存行中可以存储多个数据项。因此，即使每个线程都只访问它们自己的数组条目，缓存硬件还是会产生乒乓缓存。每次线程访问 0 号条目需要对其值进行更新时，缓存行的所有权就需要转移给执行该线程的处理器，仅仅因为负责条目 1 的线程需要更新它的数据项，缓存就需要传送到运行这个线程的处理器。缓存行是共享的，尽管没有需要共享的数据也是如此，这就是术语*伪共享（false sharing）*的由来。解决的办法就是构造数据，让同一线程访问的数据项在内存中紧挨着(因而更大概率在同一缓存行中)，对不同线程访问的数据在内存中尽量远离，从而大概率在不同的缓存行。本章接下来的内容中可以看到，这种思路对代码和数据设计的影响。C++17 标准在<new>头文件中定义了 std::hardware_destructive_interference_size，它指定了当前编译目标可能伪共享的最大连续字节数。如果能确保数据间隔不小于这个字节数，就不存在伪共享。

如果多线程访问同一缓存行是糟糕的，那么被单个线程访问的数据的内存布局有什么样的影响呢？

### 8.2.4 数据是否紧凑？

尽管伪共享发生的原因是一个线程所要访问的数据过于靠近另一线程访问的数据，但另一个与数据布局相关的陷阱会直接影响单线程的性能。这个问题是数据邻近度：如果被单线程访问的数据散落在内存中，那么它们很可能位于不同的缓存行上。另一方面，如果单线程访问的数据在内存中紧挨着，它们很可能位于相同的缓存行上。因此，如果数据是分散的，将有更多的缓存行从内存加载到处理器的缓存中，这会增加访问内存的延时，相比数据紧挨着的布局，性能也会降低。

同样的，如果数据是分散的，给定缓存行上不仅包含当前线程需要的数据，同时包含本线程无关数据的机会也在增加。极端情况下，缓存中存储的大部分是你不想要的数据。这会浪费宝贵的缓存空间，增加处理器缓存不命中的情况，即使这个数据项曾在缓存中存在过，还是需要从主存中获取它，因为为了给其他数据项腾出空间，它已经从缓存中移除。

既然这对单线程代码很重要，那我为啥要在这里提及呢？原因就是*任务切换*(task switching)。如果系统中的线程数量要于核的数量，每个核上将会运行多个线程。这就会增加缓存的压力，因为为了避免伪共享，不同线程需要访问不同的缓存行。因此，当处理器切换线程时，如果每个线程使用的数据散落在多个缓存行，它需要重新加载缓存行的可能性就比数据在同一个缓存行中紧挨着的情况要大。C++17 同样在头文件<new>中指定了一个常量 std::hardware_constructive_interference_size，它代表保证在同一缓存行上的最大连续字节数(当然这里假设首地址已经适当对齐)。如果你能将需要的数据放一起并且不多于这个字节数，就能潜在的减少缓存不命中的情况。

如果线程数量多于核或处理器的数量，操作系统可能也会选择在某个时间片将一个线程调度到一个核上，然后在下一个时间片再安排给另一个核。因此要求将线程数据对应的缓存行从第一个核的缓存中转移到第二个核上；越多的缓存行需要转移，也就会消耗越多的时间。尽管操作系统通常会尽可能避免这种情况的发生，但它还是会发生，并且的确会影响性能。

当许多线程准备运行而非等待的时候，任务切换问题相当普遍。这个问题我们之前已经接触过：超订。

### 8.2.5 超订与过度的任务切换

在多线程系统中，除非你运行在*大规模并行*(massively parallel)硬件上，通常线程的数量要多于处理器的数量。但线程经常会花费时间来等待外部 I/O 完成，或被互斥锁阻塞，或等待条件变量等等，因此这不是问题。可以让应用程序使用额外的线程来执行有用的工作，而不是在线程等待的时候，让处理器处于闲置状态。

但这也并非总是好事，如果有太多额外的线程，就可能有多于处理器数量的线程准备运行，然后操作系统就会深陷任务切换中，以确保它们都能获得公平的时间片。如第 1 章所见，这将增加任务切换的开销同时也会混合因为数据邻近度不够导致的缓存问题。当任务无节制的产生新线程，超订就会加剧，就像第 4 章的递归快速排序那样，或者在通过任务类型进行划分的地方，线程数量大于处理器数量，并且任务是计算密集型而非 I/O 密集型。

如果是因为数据划分而生成太多的线程，你可以限定工作线程的数量，正如在 8.1.2 节中看到的那样。如果超订是对工作的自然划分而产生，除了选择不同的划分方式外，没有其他更好的办法来改善这个问题。那种情况下，选择一个恰当的划分需要对目标平台有更多的了解，并且也只有在性能无法接受的情况下才值得这么多，而且可以通过观察改变工作划分是否改进了性能来验证它。

其他因素也会影响多线程代码的性能，例如，乒乓缓存的开销在两个单核处理器和在一个双核处理器上会非常不同，即使他们是相同的 CPU 型号和时钟频率，但这些是产生明显影响的主要因素。现在让我们看一下这些因素如何影响代码与数据结构的设计。

## 8.3 为多线程性能设计数据结构

在 8.1 节中，我们了解了各种划分工作的方法；并且在 8.2 节，了解了影响代码性能的各种因素。在设计数据结构的时候，如何使用这些信息提高多线程代码的性能呢？这个问题与第 6、7 章中解决的问题不同，之前是关于如何设计能够安全并发访问的数据结构。在 8.2 节中已经看到，即使数据并未与其他线程进行共享，单线程中使用的数据布局可能会对性能产生影响。

当为多线程性能而设计数据结构的时候，要记住的关键事情是*竞争*(contention)，*伪共享*(false sharing)和*数据邻近度*(data proximity)。所有这三个因素对于性能都有着重

大的影响，并且通常可以通过改变数据布局，或者更改哪些数据元素分配给哪个线程来改善性能。首先，让我们来看一个简单的场景：线程间划分数组元素。

## 8.3.1 为复杂操作划分数组元素

假设你正在做一些繁重的数学运算，需要将两个大的方阵相乘。要将矩阵相乘，需要将第一个矩阵第一行的每个元素与第二个矩阵第一列的相应元素相乘，然后将乘积相加，得到结果的左上角元素。然后，重复此操作，第二行和第一列，以产生结果的第一列上的第二个元素，第一行和第二列，以产生结果的第二列的第一个元素，以此类推。如图 8.3 所示，高亮展示的是第一个矩阵的第二行与第二个矩阵的第三列产生结果的第二行第三列条目的过程。

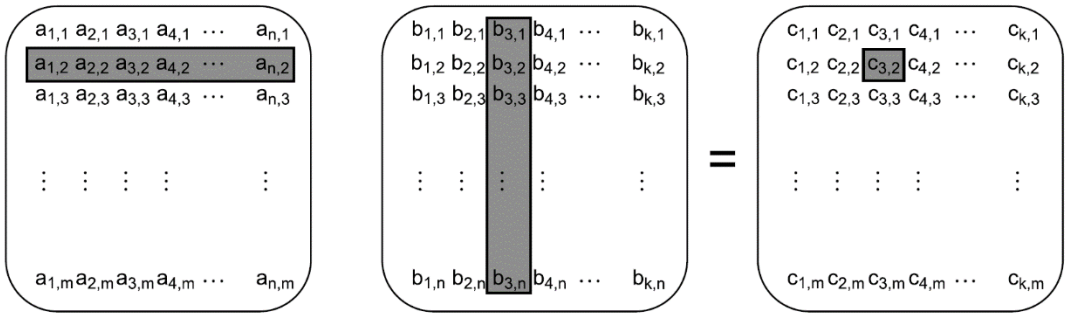


图 8.3 矩阵乘法

现在，让我们假设这些是具有几千行和几千列的大型矩阵，以便值得使用多线程来优化乘法。通常，非稀疏矩阵在内存中可以用一个大数组来表示，第一行的所有元素后跟第二行的所有元素，以此类推。要将矩阵相乘，你需要三个这样的大数组。为了获得最佳性能，你需要仔细注意数据访问模式，特别对第三个数组的写入。

线程间划分工作有很多种方式。假设矩阵的行或列数量大于可用处理器的数量，可以让每个线程计算结果矩阵中若干列的值，或者让每个线程计算若干行的结果，甚至让每个线程计算矩阵的一个矩形子集的结果。

回顾一下 8.2.3 和 8.2.4 节，从一个数组中访问连续的元素要比访问到处散落的值要好，因为这将会减少缓存的使用和伪共享的机会。如果让每个线程计算一组列，它需要从第一个矩阵中读取每个值，并从第二个矩阵中读取相应列的值，但你只需要写入列值。假设矩阵存储在连续的行中，这意味着你将从第一行访问 N 个元素，从第二行访问 N 个元素，以此类推(其中 N 是正在处理的列数)。因为其他线程将访问每一行的其他元素，很明显，你应该访问相邻的列，这样每一行的 N 个元素是相邻的，这样就可以最小化伪共享。如果 N 个元素所占用的空间正好是缓存行的倍数，那么就不会有伪共享，因为线程将在单独的缓存行上工作。（译注：第一次看这段的时候有点懵逼，这里都是从结果矩阵视角说的）

另一方面，如果让每个线程计算一组行，那么它需要从第二个矩阵读取每个值，并从第一个矩阵的相应行读取值，但它只需要写入行值。因为矩阵存储在连续的行中，所以现在访问的是 N 行中的所有元素。如果你再次选择相邻行，这意味着这个线程现在是唯一写这 N 行的线程；它有一个连续的内存块，不会被任何其他线程触碰。与让每个线程计算一组列相比，这可能是一种改进，因为伪共享的唯一可能是一个块的最后几个元素和下一个块的前几个元素，但是值得在目标体系结构上测定时间来确认。

第三个选择怎么样——将矩阵分成矩形块？这可以看作先对列进行划分，再对行进行划分。因此，它潜在的和按列划分有相同的伪共享的问题。如果可以选择块中的列数来避免这种可能性，则从读取端进行矩形划分有一个好处：你不需要读取任何一个源矩阵的全部内容。你只需要读取与目标矩形的行和列对应的值。为了具体地看一下，考虑将两个 1000 行 1000 列的矩阵相乘。那是 1 百万个元素。如果你有 100 个处理器，这样就可以每次处理 10 行的数据，每个都是 10,000 个元素。但是为了计算这 10,000 个元素的结果，他们需要访问第二个矩阵的全部(100 万个元素)加上第一个矩阵中相应行的 10000 个元素，总共有 1010,000 个元素。另一方面，如果每个线程计算 100*100 个元素的块(仍然是 10,000 个元素总数)，它们需要访问第一个矩阵的 100 行(100 x 1,000 = 100,000 个元素)和第二个矩阵的 100 列(另一个 100,000)的值。这仅仅是 200,000 个元素，这将读取的元素数量减少了 5 倍。如果读取更少的元素，那么缓存不命中的几率就越小，潜在性能也越好。

因此，最好将结果矩阵分成小的、正方形的或接近正方形的块，而不是让每个线程计算一小部分行的全部。你可以在运行时调整每个块的大小，这取决于矩阵的大小和可用的处理器数量。和以往一样，如果性能很重要，那么分析目标架构上的各种选项至关重要，同时请查阅与该领域相关的文献——如果你正在做矩阵乘法，我不认为这些是惟一的或最好的选择。

你可能不是在做矩阵乘法，那这对你有什么用呢？同样的原则也适用于需要在线程间划分大数据块的任何情况：仔细查看数据访问模式的所有方面，并确定导致性能下降的潜在原因。在你的问题域中可能存在类似的情况，在这些地方改变工作的划分就可以提高性能，而不需要对基本算法做任何更改。

好了，我们已经了解了数组的访问模式是如何对性能产生影响的。那其他类型的数据结构呢？

## 8.3.2 其他数据结构中的数据访问模式

从根本上讲，当尝试优化其他数据结构的数据访问模式时，需要考虑和优化数组访问相同的问题：

- 尝试调整数据在线程间的分布，这样邻近的数据就可以被同一个线程处理。
- 尽量减少任何给定线程所需的数据。
- 尝试确保由不同线程访问的数据之间的距离足够远，以避免伪共享，可以使用 std::hardware_destructive_interference_size 作为指导。

应用到其他数据结构上并不容易。例如，除了子树之外，二叉树在任何单位上都很难细分，这可能有用也可能没用，这取决于树的平衡程度以及你需要将它划分成多少个部分。同样，树的的属性决定了其节点是动态分配的，因而位于堆的不同位置。

现在，将数据放在堆的不同位置本身并不是一个特别的问题，但它确实意味着处理器必须在缓存中保存更多的东西。但这是有益的。如果多个线程需要遍历树，那么它们都需要访问树节点，但是如果树节点只包含指向该节点上保存的实际数据的指针，那么处理器只需要在需要时从内存中加载数据。如果数据被需要它的线程修改，这可以避免节点数据本身和提供树结构的数据之间伪共享的性能损失。

由互斥锁保护的数据也有类似的问题。假设你有一个简单的类，包含一些数据项和一个用于保护从多线程访问数据的互斥锁。如果互斥锁和数据项在内存紧挨着，对于一个需要获取互斥锁的线程来说比较理想：它需要的数据可能已经在处理器的缓存中，因为它是为了修改互斥锁而加载的。不过，还有一个缺点：当第一个线程还持有锁时，如果其他线程尝试锁住互斥锁，它们需要访问那个内存。互斥锁上锁通常实现为在互斥锁内部内存位置上的“读-改-写”原子操作，以尝试获取互斥锁，如果互斥锁已经锁住，则调用操作系统内核。这个“读-改-写”操作，可能会导致持有互斥锁的线程在缓存中持有的数据失效。就互斥锁而言，这不是个问题；该线程在解锁互斥锁之前不会触碰它。不过，如果互斥锁和线程要使用的数据共享同一缓存行时，持有互斥锁的线程将会有性能损失，因为另一个线程尝试锁住互斥锁。

测试这种伪共享是否是个问题的一种方法是在数据元素之间添加大量填充，不同的线程可以并发地访问这些数据元素。例如，你可以使用：

```
struct protected_data
{
  std::mutex m;
  char padding[65536];  // 如果你的编译器不支持
std::hardware_destructive_interference_size，可以使用类似 65536 字节，这个数字肯定超过一个缓存行
  my_data data_to_protect;
};
```

用来测试互斥锁竞争问题或

```
struct my_data
{
  data_item1 d1;
  data_item2 d2;
  char padding[std::hardware_destructive_interference_size];
};
my_data some_array[256];
```

用来测试数组数据中的伪共享。如果这样能够提高性能，你就知道伪共享确实是个问题，然后你可以保留填充，也可以通过重新安排数据访问，以另一种方式消除伪共享。

当设计并发的时候，需要考虑的不只是数据访问模式，所以让我们看看其他注意事项。

# 8.4 设计并发时的其他注意事项

本章中我们已经了解了很多线程间划分工作的方法，影响性能的因素，以及这些因素是如何影响你选择数据访问模式和数据结构的。但是，为并发设计代码还有更多的工作要做。还需要考虑异常安全性和可扩展性等问题。如果性能随着系统中处理核数的增加而增长（无论是降低了执行时间（译注：原文是执行速度，感觉说不太通），还是增加吞吐量），那这样的代码称为“可扩展”的。理想状态下，性能增长是线性的，因此一个拥有100 个处理器的系统比一个处理器的系统好 100 倍。

尽管代码即使不是可扩展的也能正常工作——例如，一个单线程应用肯定无法扩展——但异常安全是正确性问题。如果代码不是异常安全的，最终会破坏不变量，或是导致竞争条件，或者你的应用可能意外终止，因为某个操作抛出了一个异常。记住这一点后，我们将首先看看异常安全问题。

## 8.4.1 并行算法中的异常安全

异常安全是高质量 C++代码必须具备的，并发代码也不例外。事实上，相比串行算法，并行算法通常需要更小心地处理异常。如果一个操作在串行算法中抛出一个异常，算法只需要确保在它自己之后进行清理，以避免资源泄漏和破坏不变量，它可以欣然允许异常传播给调用者，让它们来处理。相比之下，在并行算法中，许多操作将在单独的线程上运行。这种情况下，不允许传播异常，因为它在错误的调用栈上。如果函数在新创建的线程上带着异常退出，应用程序将终止。

作为一个具体的例子，让我们回顾一下清单 2.9 中的 parallel_accumulate 函数，复制如下：

清单 8.2 std::accumulate 的初级并行版本(源于清单 2.9)

```
template<typename Iterator,typename T>
struct accumulate_block
{
  void operator()(Iterator first,Iterator last,T& result)
  {
    result=std::accumulate(first,last,result);  // 1
  }
};

template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
```

```
  unsigned long const length=std::distance(first,last);  // 2

  if(!length)
    return init;

  unsigned long const min_per_thread=25;
  unsigned long const max_threads=
    (length+min_per_thread-1)/min_per_thread;

  unsigned long const hardware_threads=
    std::thread::hardware_concurrency();

  unsigned long const num_threads=
    std::min(hardware_threads!=0?hardware_threads:2,max_threads);

  unsigned long const block_size=length/num_threads;

  std::vector<T> results(num_threads);  // 3
  std::vector<std::thread> threads(num_threads-1);  // 4

  Iterator block_start=first;  // 5
  for(unsigned long i=0;i<(num_threads-1);++i)
  {
    Iterator block_end=block_start;  // 6
    std::advance(block_end,block_size);
    threads[i]=std::thread(  // 7
      accumulate_block<Iterator,T>(),
      block_start,block_end,std::ref(results[i]));
    block_start=block_end;  // 8
  }
  accumulate_block()(block_start,last,results[num_threads-1]);  // 9

  std::for_each(threads.begin(),threads.end(),
    std::mem_fn(&std::thread::join));

  return std::accumulate(results.begin(),results.end(),init);  // 10
}
```

现在，让我们仔细检查并确定可以抛出异常的位置：任何调用函数的地方都可以抛出异常，或在用户自定义类型上执行某个操作时也可能抛出异常。

首先，调用了 distance②，它会在用户提供的迭代器类型上执行操作。因为还没有做任何工作，并且这是在调用线程上，所以没有问题。接下来，分配了 results③和 threads④的 vector。同样，这些都在调用线程上，并且没有做任何工作或创建任何线程，所以还

是没有问题。如果 threads 的构造函数抛出异常，为 results 分配的内存就必须被清理，不过析构函数会帮你打理好一切。

因为同样是安全的，可以跳过 block_start⑤的初始化，然后来到了生成新线程的循环⑥⑦⑧。一旦在⑦处完成了第一个线程的创建，如果抛出任何异常就会有问题，因为新的 std::thread 对象的析构函数，将调用 std::terminate 来终止程序的运行。这可不是什么好地方。

accumulate_block⑨的调用可能抛出异常，并产生类似的后果：线程对象将会被销毁，并且调用 std::terminate。另一方面，最终调用 std::accumulate⑩可能会抛出异常，不过不会造成什么困难，因为所有的线程在这个点已经被连接了。

主线程分析完了，但还有更多地方：在新线程上调用 accumulate_block 可能会在①抛出异常。由于没有任何 catch 块，所以这个异常不会被处理，并且导致库调用 std::terminater()来终止应用程序。

这里它并不是很明显，但这段代码不是异常安全的。

**添加异常安全**

好了，我们已经确定了所有可能的抛出点，并且分析了异常所带来的恶劣后果。你能做些什么呢？让我们从解决新线程上抛出异常的问题开始。

在第 4 章中遇到过用于这项工作的工具。如果你仔细查看试图用新线程实现的目标，很明显，你正在尝试计算要返回的结果，同时允许代码可能抛出异常。这正是 std::packaged_task 和 std::future 这对组合设计的目的。如果重新安排代码以使用 std::packaged_task，将得到以下代码。

清单 8.3 使用 std::packaged_task 的并行版 std::accumulate

```
template<typename Iterator,typename T>
struct accumulate_block
{
  T operator()(Iterator first,Iterator last)  // 1
  {
    return std::accumulate(first,last,T());  // 2
  }
};

template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
  unsigned long const length=std::distance(first,last);

  if(!length)
    return init;
```

```
  unsigned long const min_per_thread=25;
  unsigned long const max_threads=
    (length+min_per_thread-1)/min_per_thread;

  unsigned long const hardware_threads=
    std::thread::hardware_concurrency();

  unsigned long const num_threads=
    std::min(hardware_threads!=0?hardware_threads:2,max_threads);

  unsigned long const block_size=length/num_threads;

  std::vector<std::future<T> > futures(num_threads-1);  // 3
  std::vector<std::thread> threads(num_threads-1);

  Iterator block_start=first;
  for(unsigned long i=0;i<(num_threads-1);++i)
  {
    Iterator block_end=block_start;
    std::advance(block_end,block_size);
    std::packaged_task<T(Iterator,Iterator)> task(  // 4
      accumulate_block<Iterator,T>());
    futures[i]=task.get_future();  // 5
    threads[i]=std::thread(std::move(task),block_start,block_end);  //
6
    block_start=block_end;
  }
  T last_result=accumulate_block()(block_start,last);  // 7

  std::for_each(threads.begin(),threads.end(),
    std::mem_fn(&std::thread::join));

  T result=init;  // 8
  for(unsigned long i=0;i<(num_threads-1);++i)
  {
    result+=futures[i].get();  // 9
  }
  result += last_result;  // 10
  return result;
}
```

第一个变化是 accumulate_block 的函数调用操作符现在直接返回结果，而非使用引用存储它①。由于你正在使用 std::packaged_task 和 std::future 来实现异常安全，因此也

可以使用它们来传输结果。在 std::accumulate 调用中，要求显式地传递一个默认构造的 T，而不是重用提供的 result 值，不过这是个小改动。

下一个变化是使用一个期望的 vector 来为每个创建的线程存储 std::future<T>③，而不是使用结果的 vector。在创建线程循环中，首先为 accumulate_block 创建一个任务④。std::packaged_task<T(Iterator,Iterator)>声明了一个任务，这个任务需要两个 Iterators 参数并返回 T。然后，从任务中获取期望⑤，再传入要处理数据块的开始和结束⑥迭代器，让新线程去执行这个任务。当任务运行时，结果将会在期望中捕获，任何抛出的异常也会捕获。

因为一直在使用期望，所以没有结果数组，因此必须将来自最后那个块的结果存储在变量中⑦，而不是放入一个数组的槽中。同样，因为必须从期望中获取值，现在使用基本的 for 循环比使用 std::accumulate 更简单，从提供的初始值开始⑧，并且累加每个期望上的结果⑨。如果相关的任务抛出一个异常，这必将被期望捕捉，并且现在被 get()调用再次抛出。最后，在返回全部结果给调用者之前，加上最后一个数据块的结果⑩。

这消除了一个潜在的问题：在工作线程中抛出的异常会在主线程中重新抛出。如果有多个工作线程抛出异常，则只会传播一个，不过这没什么大不了的。如果它很重要，可以使用类似 std::nested_exception 的类来捕获所有异常并抛出它来代替原来的异常。

剩下的问题是，如果在生成第一个线程和连接所有线程之间抛出异常，则会导致线程泄漏。最简单的解决方案是捕获任何异常，连接仍然是 joinable()的线程，并重新抛出异常：

```
try
{
  for(unsigned long i=0;i<(num_threads-1);++i)
  {
    // ... 和之前一样
  }
  T last_result=accumulate_block()(block_start,last);

  std::for_each(threads.begin(),threads.end(),
  std::mem_fn(&std::thread::join));
}
catch(...)
{
  for(unsigned long i=0;i<(num_thread-1);++i)
  {
  if(threads[i].joinable())
    thread[i].join();
  }
  throw;
}
```

现在可以了。无论代码是怎么离开块的，所有的线程都将被连接。但是 try-catch 块很丑陋，并且有重复的代码：你正在连接“正常”控制流和 catch 块中的线程。重复的代码很少是好事，因为它意味着有更多的地方需要更改。相反，我们将它提取到对象的析构函数中；毕竟，这是 C++中清理资源的惯用手法。这是你的类：

```
class join_threads
{
  std::vector<std::thread>& threads;
public:
  explicit join_threads(std::vector<std::thread>& threads_):
    threads(threads_)
  {}
  ~join_threads()
  {
    for(unsigned long i=0;i<threads.size();++i)
    {
      if(threads[i].joinable())
        threads[i].join();
    }
  }
};
```

这和清单 2.3 中的 thread_guard 类很相似，只是它为整个线程 vector 做了扩展。代码可以简化如下：

清单 8.4 异常安全的并发版 std::accumulate

```
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
  unsigned long const length=std::distance(first,last);

  if(!length)
    return init;

  unsigned long const min_per_thread=25;
  unsigned long const max_threads=
    (length+min_per_thread-1)/min_per_thread;

  unsigned long const hardware_threads=
    std::thread::hardware_concurrency();

  unsigned long const num_threads=
    std::min(hardware_threads!=0?hardware_threads:2,max_threads);
```

```
  unsigned long const block_size=length/num_threads;

  std::vector<std::future<T> > futures(num_threads-1);
  std::vector<std::thread> threads(num_threads-1);
  join_threads joiner(threads);  // 1

  Iterator block_start=first;
  for(unsigned long i=0;i<(num_threads-1);++i)
  {
    Iterator block_end=block_start;
    std::advance(block_end,block_size);
    std::packaged_task<T(Iterator,Iterator)> task(
      accumulate_block<Iterator,T>());
    futures[i]=task.get_future();
    threads[i]=std::thread(std::move(task),block_start,block_end);
    block_start=block_end;
  }
  T last_result=accumulate_block()(block_start,last);
  T result=init;
  for(unsigned long i=0;i<(num_threads-1);++i)
  {
    result+=futures[i].get();  // 2
  }
  result += last_result;
  return result;
}
```

一旦创建了线程容器，就对新类创建一个实例①以在退出的时候连接所有线程。然后，可以移除显式的连接循环而无须担心，因为无论函数怎么退出，线程都会被连接。注意对 futures[i].get()②的调用将会阻塞线程，直到结果准备就绪，所以在这个点不需要显式连接线程。这和清单 8.2 中的原始代码不同：原始代码中需要连接线程，以确保 results 的 vector 被正确填充。现在不仅得到异常安全的代码，而且函数也更短，因为将连接代码抽到了新(可复用)类中。

### 使用 std::async()的异常安全

现在已经了解了，当需要显式管理线程的时候，异常安全都需要什么。那现在让我们来看一下使用 std::async()是怎么样实现异常安全的。你已经看到了，在本例中，库负责为你管理线程，当期望就绪时，意味着创建的任何线程都已完成。关于异常安全需要注意的关键一点是，如果你在没有等待它的情况下销毁期望，析构函数将等待线程完成。这巧妙地避免了泄漏线程的问题，这些线程仍然在执行并持有对数据的引用。下一个清单显示了使用 std::async()的异常安全的实现。

清单 8.5 使用 std::async()的异常安全的并行版 std::accumulate

```
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
  unsigned long const length=std::distance(first,last);  // 1
  unsigned long const max_chunk_size=25;
  if(length<=max_chunk_size)
  {
    return std::accumulate(first,last,init);  // 2
  }
  else
  {
    Iterator mid_point=first;
    std::advance(mid_point,length/2);  // 3
    std::future<T> first_half_result=
      std::async(parallel_accumulate<Iterator,T>,  // 4
        first,mid_point,init);
    T second_half_result=parallel_accumulate(mid_point,last,T());  // 5
    return first_half_result.get()+second_half_result;  // 6
  }
}
```

这个版本使用数据的递归划分，而不是预先计算数据块的划分，但它比之前的版本简单了很多，并且仍然是异常安全的。和之前一样，首先找到序列的长度①，如果它小于最大块的大小，直接调用 std::accumulate②。如果元素的数量超出了数据块大小，就需要找到中点③，然后再生成一个异步任务来处理这一半数据④。第二半的数据直接使用递归调用来处理⑤，然后将两个块的结果加和到一起⑥。标准库能确保 std::async 调用能够充分的利用硬件线程，而不会创建大量的线程，一些“异步”调用是在 get()⑥调用中同步执行的。

这里的妙处在于，它不仅可以充分利用硬件并发，而且还可以保证异常安全。如果有异常在递归调用⑤中抛出，从调用 std::async④创建的期望将会在异常传播时销毁。这将依次等待异步任务的完成，因此避免了悬垂线程的出现。另一方面，如果异步调用抛出异常，期望会捕获它，并且对 get()⑥的调用的时候会再次抛出这个异常。

在设计并发代码时，你还需要考虑哪些其他因素？让我们来看一下*可扩展性*（scalability）。如果将代码移动到有更多处理器的系统中，性能有多少提升？

## 8.4.2 可扩展性和 Amdahl 定律

可扩展性其实就是确保你的应用可以利用系统中额外的处理器。一个极端是单线程应用程序完全不可扩展；即使添加了 100 个处理器到你的系统中，应用的性能也不会变化。另一个极端是像 SETI@Home（http://setiathome.ssl.berkeley.edu/）项目一样，它的设计目标是可以利用数千个额外的处理器(以用户添加到网络中的个人电脑的形式)，前提是有这么多处理器。

对任意给定的多线程程序，执行有用工作的线程数量会随着程序的运行而变化。即使每个线程在它存在的整个过程中都在做有用的工作，应用程序最初可能只有一个线程，然后该线程将承担生成所有其他线程的任务。但即使是这样的情况也极不可能发生。更普遍的情况是线程经常花时间等待彼此或等待 I/O 操作完成。

每当一个线程必须等待某个事件(无论这个事件是什么)，除非有另一个线程准备在处理器上取代它，否则就有一个处理器处于空闲状态，而它本来可以做一些有用的工作。

一种简化的方法是将程序分为“串行”部分，其中只有一个线程在工作，以及“并行”部分，所有可用的处理器都可以同时工作。如果在有更多处理的系统上运行应用程序，“并行”部分理论上会完成的更快，因为工作可以在更多的处理器之间划分，而“串行”部分仍然是串行的。在这样一组简化的假设下，你可以评估通过增加处理器数量来获得的潜在性能增益：如果“串行”部分构成程序的一部分 fs，那么使用 N 个处理器的性能增益 P 可以评估为

$$P = \frac{1}{f_S + \frac{1 - f_S}{N}}$$

这就是 Amdahl 定律，当讨论并发代码性能的时候经常被引用。如果每行代码都能并行化，那么串行部分是 0，加速比就是 N。或者，如果串行部分是 1/3，即使有无限数量的处理器，也无法获得超过 3 的加速比。

但这描绘了一幅天真的画面，因为任务很少能被无限地划分成等式所要求的那样，而且也很少说所有的东西都像假设的那样是 CPU 密集型的。正如你所看到的，线程在执行的时候可能会等待很多事件。

Amdahl 定律比较明确的一点是，当你使用并发提高性能时，有必要查看应用程序的整体设计，以最大化并发的潜力，并确保处理器总是有有用的工作要做。如果可以减少“串行”部分的大小或减少线程等待的可能性，在有更多处理器的系统上，就可以提升性能增益的可能性。或者，如果你能提供更多的数据供系统处理，因而可以让并行部分保持工作饱满，这样你就可以减少串行分数并增加性能增益，P。

可扩展性是关于减少执行一个动作所需的时间，或者随着更多处理器的增加，增加在给定时间内可以处理的数据量。有时这些是等价的(如果每个元素处理的更快，就可以处理更多的数据)，但并不总是这样。选择线程间划分工作的技术前，确定可扩展性的哪些方面对你很重要，这点很重要。

在本节开头我提到过，线程并非总有有用的工作可做。有时，它们必须等待其他线程，或者等待 I/O 完成，或是等待其他事件。如果在等待期间，让系统做一些有用的事情，就可以有效的“隐藏”等待。

### 8.4.3 使用多线程隐藏延迟

在大多数关于多线程代码性能的讨论中，我们一直假定线程正在“全速”运行，并且当它们运行在处理器上时总是有有用的工作要做。但事实并非如此；在应用程序代码中，线程在等待时经常会阻塞。例如，他们可能正在等待 I/O 完成，等待获得一个互斥锁，等待另一个线程完成某个操作并通知一个条件变量或填充一个期望，甚至是休眠一段时间。

不论等待的理由是什么，如果只有和系统中物理单元相同数量的线程，那么线程阻塞就意味着浪费 CPU 时间。本应运行阻塞线程的处理器现在什么也不做。因此，如果知道一个线程可能会花费相当多的时间等待，那么可以通过运行一个或多个额外的线程来利用空闲的 CPU 时间。

考虑一个病毒扫描程序，它使用流水线在线程间划分工作。第一个线程在文件系统中搜索要检查的文件，并将它们放入队列中。同时，另一个线程从队列中获取文件名，加载文件，之后对它们进行病毒扫描。你知道在文件系统中搜索要扫描的文件的线程肯定是 I/O 密集型操作，因此可以通过运行一个额外的扫描线程来利用“空闲”CPU 时间。然后，你将有一个文件搜索线程以及和系统中物理核或处理器数量相同的扫描线程。因为扫描线程为了扫描文件，可能还必须从磁盘读取文件的大部分内容，所以可能需要更多的扫描线程。但是在某一时刻会有太多的线程，然后系统会再次变慢，因为它会花费越来越多的时间来切换任务，如 8.2.5 节所述。

和之前一样，这也是个优化，测量线程数量修改前后的性能很重要；线程的最佳数量将高度依赖于正在完成的工作的性质以及线程花费在等待上的时间百分比。

根据应用程序的不同，有可能在不运行额外线程的情况下用完这些空闲的 CPU 时间。例如，如果一个线程因为等待 I/O 操作完成而阻塞，使用异步 I/O 就会比较有意义，然后当 I/O 在后台执行的时候，这个线程可以执行其他有用的工作。在其他情况下，如果一个线程等待其他线程去执行一个操作，不同于阻塞，等待线程自身可以执行那个操作，就像在第 7 章中看到的无锁队列那样。在极端情况下，如果一个线程正在等待一个任务完成，而该任务还没有被任何线程启动，那么等待的线程可能会完整地执行该任务，或者执行另一个未完成的任务。你在清单 8.1 中看到了这样一个示例，其中只要 sort 函数需要的块还没有排序，它就会反复尝试对尚未排序的块进行排序。

不同于添加线程以确保使用所有可用的处理器，有时添加线程以确保及时处理外部事件以提高系统的响应能力是值得的。

### 8.4.4 使用并发提高响应能力

大多数现代图形用户界面框架都是*事件驱动的*(event-driven)，用户通过按键或移动鼠标在用户界面上执行操作，这将产生一系列需要应用处理的事件或消息。系统本身也可能产生消息或事件。为了确保所有的事件和消息都能被正确处理，应用程序通常会有一个事件循环，看起来像这样：

```
while(true)
{
  event_data event=get_event();
  if(event.type==quit)
    break;
  process(event);
}
```

显然，API 中的细节会有所不同，不过结构通常是一样的：等待一个事件，对其做必要的处理，然后等待下一个事件。如果你有一个单线程的应用程序，这会使长时间运行的任务难以编写，就如 8.1.3 节所述。为了确保用户输入被及时处理，不管应用程序正在做什么，get_event()和 process()都必须以合理的频率调用。这就意味着要么任务自身被周期性的挂起，并且返回控制到事件循环中，要么必须在代码中方便的点调用 get_event()/process()代码。这两个选项都会使任务的实现变得复杂。

通过使用并发分离关注点，可以将这个冗长的任务放在一个全新的线程上，并委托一个专用的 GUI 线程来处理事件。然后，线程可以通过简单的机制进行通信，而不必以某种方式将事件处理代码与任务代码混在一起。下面的清单展示了这种分离的一个简单轮廓。

清单 8.6 将 GUI 线程从任务线程中分离

```
std::thread task_thread;
std::atomic<bool> task_cancelled(false);

void gui_thread()
{
  while(true)
  {
    event_data event=get_event();
    if(event.type==quit)
      break;
    process(event);
  }
}

void task()
{
  while(!task_complete() && !task_cancelled)
  {
    do_next_operation();
  }
  if(task_cancelled)
  {
    perform_cleanup();
  }
```

```
  else
  {
    post_gui_event(task_complete);
  }
}

void process(event_data const& event)
{
  switch(event.type)
  {
  case start_task:
    task_cancelled=false;
    task_thread=std::thread(task);
    break;
  case stop_task:
    task_cancelled=true;
    task_thread.join();
    break;
  case task_complete:
    task_thread.join();
    display_results();
    break;
  default:
    //...
  }
}
```

通过这种方式分离关注点，用户线程总能及时响应事件，哪怕任务需要花费很长时间。在使用应用程序时，这种响应能力通常是用户体验的关键；不管什么时候执行某个特定操作（不管是什么操作），如果应用程序完全锁住的话，使用起来就会很不方便。通过提供专门的事件处理线程，GUI 可以处理 GUI 相关的消息(如调整大小或重新绘制窗口)，而不用中断耗时处理的执行，同时仍然传递相关的消息，这些消息也确实会影响长时间运行的任务。

到目前为止，在本章中，你已经彻底了解了设计并发代码时需要考虑的问题。总的来说，这些可能非常难以应对，但是当你习惯使用你的“多线程编程帽”时，它们中的大多数将会成为你的习惯。如果这些注意事项对你来说是新的，那么希望随着你了解了它们是如何影响一些具体的多线程代码示例以后，它们会变得更加清晰。

## 8.5 在实践中设计并发代码

当为一个特殊的任务设计并发代码时，在多大程度上考虑前面描述的每个问题取决于任务。为了演示它们是如何应用的，我们将看看 C++标准库中三个函数的并行版本的实现。这将为你提供一个熟悉的构建基础，同时提供一个研究问题的平台。作为奖励，我们还将提供这些函数的可用实现，这些实现可用于帮助并行处理更大的任务。

选择这些实现主要是为了演示特定的技术而不是做成最先进的实现；更好地利用硬件并发的更高级的实现可以在并行算法的学术文献中找到，或者在专业的多线程库中找到，比如 Intel 的线程构建块(http://threadingbuildingblocks.org/)。

从概念上讲，最简单的并行算法是 std::for_each 的并行版本，因此我们将从它开始。

## 8.5.1 std::for_each 的并行实现

std::for_each 概念上很简单：依次对某个范围中的每个元素调用用户提供的函数。并行和串行调用的最大区别是函数的调用顺序。std::for_each 用范围中的第一个元素调用函数，接着是第二个，以此类推，然而在并行实现中对每个元素的处理顺序并没有保证，并且它们可能(事实上，我们希望如此)被并发处理。

为了实现这个函数的并行版本，需要把范围划分成每个线程上处理的元素集合。由于事先知道元素的数量，所以可以在处理前对数据进行划分(8.1.1 节)。我们假设这是唯一正在运行的并行任务，因此可以使用 std::thread::hardware_concurrency()来确定线程的数量。你还知道元素可以被完全独立地处理，因此可以使用连续块来避免伪共享(8.2.3 节)。

这个算法概念上类似于 8.4.1 节中并行版的 std::accumulate，但不同于计算每个元素的和，这里仅仅需要应用指定的函数。尽管你可能想象这会大大简化代码，因为不需要返回结果，但如果希望传递异常给调用者，就需要使用 std::packaged_task 和 std::future 机制在线程间转移异常。这里展示一个样例实现。

清单 8.7 并行版 std::for_each

```
template<typename Iterator,typename Func>
void parallel_for_each(Iterator first,Iterator last,Func f)
{
  unsigned long const length=std::distance(first,last);

  if(!length)
    return;

  unsigned long const min_per_thread=25;
  unsigned long const max_threads=
    (length+min_per_thread-1)/min_per_thread;

  unsigned long const hardware_threads=
    std::thread::hardware_concurrency();

  unsigned long const num_threads=
    std::min(hardware_threads!=0?hardware_threads:2,max_threads);
```

```
  unsigned long const block_size=length/num_threads;

  std::vector<std::future<void> > futures(num_threads-1);  // 1
  std::vector<std::thread> threads(num_threads-1);
  join_threads joiner(threads);

  Iterator block_start=first;
  for(unsigned long i=0;i<(num_threads-1);++i)
  {
    Iterator block_end=block_start;
    std::advance(block_end,block_size);
    std::packaged_task<void(void)> task(  // 2
      [=]()
      {
        std::for_each(block_start,block_end,f);
      });
    futures[i]=task.get_future();
    threads[i]=std::thread(std::move(task));  // 3
    block_start=block_end;
  }
  std::for_each(block_start,last,f);

  for(unsigned long i=0;i<(num_threads-1);++i)
  {
    futures[i].get();  // 4
  }
}
```

代码的基本结构与清单 8.4 相同，这也并不奇怪。关键的不同在于期望 vector 存储了 std::future<void>①，因为工作线程不会返回值，并且简单的 lambda 函数会对在任务中使用的从 block_start 到 block_end 的范围调用函数 f②。这避免了必须传入范围到线程的构造函数中③。因为工作线程不需要返回值，调用 futures[i].get()④只是提供了一种检索工作线程抛出异常的方法；如果不想传递异常，你可以忽略这个。

就像 std::accumulate 的并行实现使用 std::async 可以简单代码，parallel_for_each 也可以。这个实现如下。

清单 8.8 使用 std::async 的并行版 std::for_each

```
template<typename Iterator,typename Func>
void parallel_for_each(Iterator first,Iterator last,Func f)
{
  unsigned long const length=std::distance(first,last);
```

```
  if(!length)
    return;

  unsigned long const min_per_thread=25;

  if(length<(2*min_per_thread))
  {
    std::for_each(first,last,f);  // 1
  }
  else
  {
    Iterator const mid_point=first+length/2;
    std::future<void> first_half=  // 2
      std::async(&parallel_for_each<Iterator,Func>,
                 first,mid_point,f);
    parallel_for_each(mid_point,last,f);  // 3
    first_half.get();  // 4
  }
}
```

和清单 8.5 中基于 std::async 的 parallel_accumulate 一样，分割数据是运行时递归的进行而不是在执行前，因为不知道要使用多少个线程。和之前一样，在每个阶段把数据一分为二，异步执行一半②，另一半则直接执行③，直到剩下的数据太小而不值得继续划分，这种情况下遵从 std::for_each①的处理方式。同样，使用 std::async 和 std::future 的 get()成员函数④提供了异常传播语义。

让我们从必须对每个元素执行相同操作的算法（其中有几个；std::count 和 std::replace 首先浮现在脑海中)转向一个稍微复杂一点的形如 std::find 的示例。

## 8.5.2 std::find 的并行实现

std::find 是接下来要考虑的一个有用的算法，因为它是几个不需要每个元素都处理就可以完成的算法之一。例如，如果范围中第一个元素就匹配查找标准，就没有必要检查其他元素。你很快会看到，这对性能是个重要属性，并且对设计并行实现有直接的影响。这是数据访问模式是如何影响代码设计的特定示例(8.3.2 节)。该类的其他算法包括 std::equal 和 std::any_of。

假如你和你的搭档在阁楼上的纪念品盒子里寻找一张旧照片，如果你找到了照片，就不会让他们继续查找。相反，你应该让他们知道你找到了照片(也许通过大喊一声：“找到了!”)，这样他们就可以停止搜索，然后去干别的。由于很多算法的性质需要处理每一个元素，所以它们没发大喊一声“找到了！”，对 std::find 这样的算法，能够提早完成的能力是一个重要属性而不是用来挥霍的东西。因此，需要设计代码来利用它——在答案已知时以某种方式中断其他任务，这样代码就不必等待其他工作线程处理剩下的元素。

如果不中断其他线程，那么串行版本可能胜过并行实现，因为一旦找到一个匹配，串行算法就可以停止查找并返回。例如，如果系统能支持四个并发线程，那么每个线程将必须检查范围中 1/4 的元素，并且初级并发实现大概会花费单线程检查每个元素所需时间的 1/4。如果匹配的元素位于第一个 1/4 范围，串行算法将首先返回，因为它不需要检查剩下的元素。

中断其他线程的一种方法是使用原子变量作为标志，并在处理每个元素后检查该标志。如果设置了标志，说明其他线程中的一个已经找到了匹配，因此可以停止处理并返回。通过这种方式中断线程，你保留了不必处理每个值的属性，并且与串行版本相比，在更多的情况下提高了性能。但这样做的缺点是，原子加载可能是缓慢的操作，因此这可能会妨碍每个线程的进度。

现在，对于如何返回值和如何传播异常，有两种选择。你可以使用一个期望数组，std::packaged_task 来传递值和异常，然后在主线程中处理结果；或者你可以使用 std::promise 直接从工作线程设置最终结果。这完全依赖于想怎么处理工作线程上的异常。如果想停止在第一个异常上(即使还没有处理所有元素)，可以使用 std::promise 同时设置值和异常。另一方面，如果希望允许其他工作线程继续查找，可以使用 std::packaged_task，存储所有的异常，然后在没有找到匹配的情况下重新抛出其中一个异常。

这种情况下，我会选择 std::promise，因为它的行为和 std::find 更为接近。这里需要注意的一件事是，要搜索的元素不在提供的范围内。因此，需要等待所有线程完成，然后才能从期望获得结果。如果你阻塞在期望上，要是值不存在的话，你将永远等待下去。结果显示在这里。

清单 8.9 并行 find 算法实现

```
template<typename Iterator,typename MatchType>
Iterator parallel_find(Iterator first,Iterator last,MatchType match)
{
  struct find_element  // 1
  {
    void operator()(Iterator begin,Iterator end,
                    MatchType match,
                    std::promise<Iterator>* result,
                    std::atomic<bool>* done_flag)
    {
      try
      {
        for(;(begin!=end) && !done_flag->load();++begin)  // 2
        {
          if(*begin==match)
          {
            result->set_value(begin);  // 3
            done_flag->store(true);  // 4
```

```
            return;
          }
        }
      }
      catch(...)  // 5
      {
        try
        {
          result->set_exception(std::current_exception());  // 6
          done_flag->store(true);
        }
        catch(...)  // 7
        {}
      }
    }
};

unsigned long const length=std::distance(first,last);
if(!length)
  return last;

unsigned long const min_per_thread=25;
unsigned long const max_threads=
  (length+min_per_thread-1)/min_per_thread;

unsigned long const hardware_threads=
  std::thread::hardware_concurrency();

unsigned long const num_threads=
  std::min(hardware_threads!=0?hardware_threads:2,max_threads);
unsigned long const block_size=length/num_threads;

std::promise<Iterator> result;  // 8
std::atomic<bool> done_flag(false);  // 9
std::vector<std::thread> threads(num_threads-1);
{  // 10
  join_threads joiner(threads);

  Iterator block_start=first;
  for(unsigned long i=0;i<(num_threads-1);++i)
  {
    Iterator block_end=block_start;
    std::advance(block_end,block_size);
    threads[i]=std::thread(find_element(),  // 11
```

```
                              block_start,block_end,match,
                              &result,&done_flag);
      block_start=block_end;
    }
    find_element()(block_start,last,match,&result,&done_flag);  // 12
  }
  if(!done_flag.load())  //13
  {
    return last;
  }
  return result.get_future().get();  // 14
}
```

清单 8.9 中的函数主体与之前的例子相似。这次，工作由局部类 find_element①的函数调用操作符完成。这个循环遍历给定数据块中的元素，每一步都会检查标志②。如果找到了一个匹配，设置最终结果值到承诺③中，然后在返回前设置 done_flag④。

如果抛出一个异常，就会被捕获所有异常的处理器⑤捕获，并在设置 done_flag 前，尝试将异常存储在承诺中⑥。如果承诺已经被设置，在承诺上设置值可能会抛出一个异常，因此，你可以捕获并丢弃这里发生的任何异常⑦。

这意味着，如果调用 find_element 的线程找到了一个匹配或抛出异常，所有其他线程都将看到 done_flag 标志设置并停止。如果多个线程同时找到一个匹配或抛出异常，它们将在承诺中竞争来设置结果。不过这是一个良性竞争条件；无论哪个成功，名义上都是第一个，因此是一个可以接受的结果。

回到 parallel_find 主函数本身，使用承诺⑧和标志⑨来停止搜索，它们都将与要搜索的范围一起传递给新线程⑪。主线程也使用 find_element 来搜索剩下的元素⑫。如前所述，在检查结果之前需要等待所有线程完成，因为可能没有任何匹配的元素。可以通过把线程的启动和连接代码封装在一个块中来实现这一点⑩，因此当你检查标志来查看是否找到一个匹配的时候，所有线程都会被连接⑬。如果找到一个匹配，可以通过调用从承诺获得的 std::future<Iterator>上的 get()来获得结果或抛出存储的异常⑭。

同样，这个实现假设你将使用所有可用的硬件线程，或者你有一些其他机制来确定线程数量，以便在线程之间预先划分工作。与前面一样，你可以使用 std::async 和递归数据划分来简化你的实现，同时使用 C++标准库的自动扩展设施。下面的清单显示了使用 std::async 的 parallel_find 实现。

清单 8.10 使用 std::async 实现的并行 find 算法

```
template<typename Iterator,typename MatchType>  // 1
Iterator parallel_find_impl(Iterator first,Iterator last,MatchType matc
h,
                            std::atomic<bool>& done)
{
```

```
  try {
    unsigned long const length=std::distance(first,last);
    unsigned long const min_per_thread=25;  // 2
    if(length<(2*min_per_thread)) { // 3
      for(;(first!=last) && !done.load();++first) { // 4
        if(*first==match) {
          done=true;  // 5
          return first;
        }
      }
      return last;  // 6
    } else {
      Iterator const mid_point=first+(length/2);  // 7
      std::future<Iterator> async_result=
        std::async(&parallel_find_impl<Iterator,MatchType>,  // 8
                   mid_point,last,match,std::ref(done));
      Iterator const direct_result=
        parallel_find_impl(first,mid_point,match,done);  // 9
      return (direct_result==mid_point)?
        async_result.get():direct_result;  // 10
    }
  } catch(...) {
    done=true;  // 11
    throw;
  }
}

template<typename Iterator,typename MatchType>
Iterator parallel_find(Iterator first,Iterator last,MatchType match)
{
  std::atomic<bool> done(false);
  return parallel_find_impl(first,last,match,done);  // 12
}
```

如果希望在找到匹配的时候提前结束，就意味着需要引入一个在所有线程之间共享的标志，以表明找到了匹配。因此，这需要传递给所有的递归调用。实现这一点最简单的方法是委托给实现函数①，它带一个额外的参数——一个 done 标志的引用，这个引用通过程序的主入口点传入⑫。

核心实现沿着熟悉的路线继续进行。与这里的许多实现一样，你设置了在单个线程上处理的最小项数②；如果数据大小不足最小项数的两倍，那么直接在当前线程运行③。该算法是一个简单的循环，遍历指定的范围，直到到达范围的末尾或 done 标志被设置④。如果找到匹配项，标识 done 就会在返回前进行设置⑤。无论是因为到达列表的末尾，还是因为另一个线程设置了 done 标志，都会停止搜索，否则返回 last 以标示这里没有找到匹配⑥。

如果范围可以划分，首先在使用 st::async 对范围的第二半进行搜索前⑧，找到中点⑦，然后小心使用 std::ref 来传递 done 标志的引用。同时，可以通过对范围的第一半直接执行递归调用来搜索。如果原始范围足够大的话，异步调用和直接的递归都会导致进一步的划分。

如果直接搜索返回的是 mid_point，这就意味着没有找到一个匹配，所以需要从异步搜索中获取结果。如果在这一半中没有找到结果的话，结果将是 last，这是一个正确的返回值表明没有找到指定的值⑩。如果“异步”调用被推迟而不是真正的异步，它会在调用 get()的地方运行；在这些情况下，如果搜索范围的下半部分成功了，就会跳过搜索范围的上半部分。如果异步搜索在另一个线程上运行，async_result 变量的析构函数将会等待该线程完成，所以这里不会有线程泄漏。

和之前一样，使用 std::async 可以给你提供异常安全和异常传播特性。如果直接递归抛出异常，期望的析构函数将确保执行异步调用的线程在函数返回之前已经结束，并且如果异步调用抛出异常，这个异常将会通过对 get()调用进行传播⑩。使用 try/catch 块环绕整个代码只是为了在异常上设置 done 标志，并确保所有线程在抛出异常时迅速终止⑪。没有它，实现仍然是正确的，但会一直搜索元素，直到每个线程都完成。

这个算法的两个实现与你所见过的其他并行算法共享一个关键特性：不再保证项按照你从 std::find 获得的顺序进行处理。如果要将算法并行化，这点很重要。如果顺序很重要，就不能并发地处理元素。如果元素是独立的，这对 parallel_for_each 就无关紧要，但这意味着 parallel_find 可能会返回一个靠近范围末尾的元素，即使有一个靠近开始的匹配，如果你没有意料到的话，这可能会让你感到惊讶。

好了，你已经成功的并行化 std::find 了。正如我在本节开头所述，还有其他类似的算法可以在不处理每个数据元素的情况下完成，这些算法也可以使用相同的技术。我们还将在第 9 章中进一步研究中断线程的问题。

为了完成我们的三个示例，我们将从不同的方向来研究 std::partial_sum。这个算法并没有得到太多的关注，但它是一个有趣的并行算法，并突出了一些额外的设计选择。

## 8.5.3 std::partial_sum 的并行实现

std::partial_sum 会计算一个范围内的连续总数，所以每个元素都被替换为这个元素和原序列中在它之前的所有元素的和。因此序列 1，2，3，4，5，变成 1，(1+2)=3，(1+2+3)=6，(1+2+3+4)=10，(1+2+3+4+5)=15。将它并行化会很有趣，因为你不能仅仅将范围划分为块并独立计算每个块。例如，第一个元素的初始值需要添加给其他每个元素。

确定范围“部分和”的一种方法是计算各个块的部分和，然后将第一个块中最后一个元素的结果值加到下一个块中的元素上，依此类推。如果有元素 1，2，3，4，5，6，7，8，9，然后把它分为三块，在第一个实例中得到{1，3，6}，{4，9，15}，{7，15，24}。然后将 6(第一块中最后一个元素的和)加到第二个块中，那么就得到{1，3，6}，{10，15，21}，{7，15，24}。然后再将第二块的最后一个元素 21 加到第三块中，然后最后一块就得到最终结果：{1，3，6}，{10，15，21}，{28，36，55}。

和原始的块划分一样，前一个块的“部分和”的加法也可以并行化。如果首先更新每个块的最后一个元素，那么块中的其余元素可以由一个线程更新，而第二个线程更新下一块，依此类推。当列表中的元素比处理核多很多时，这种方法可以很好地工作，因为每个核在每个阶段都有合理数量的元素需要处理。

如果有很多的处理核(与元素的数量相同或更多)，这种方式就工作的不是很好。如果将工作分配给多个处理器，那么在第一步你将以成对的元素来完成工作。在这些条件下，结果的正向传播意味着有许多处理器等待，因此需要为它们找一些工作去做。当然你可以采取不同的方法来解决这个问题。与从一个数据块到下一个数据块进行完整的前向传播不同，你进行的是部分传播：首先像前面一样对相邻的元素求和，然后将这些和相加到两个元素之外，然后将下一组结果相加到四个元素之外的结果，以此类推。如果你从相同的初始 9 个元素开始，在第一轮之后你得到 1，3，5，7，9，11，13，15，17，这将给你前两个元素的最终结果，第二轮后得到 1，3，6，10，14，18，22，26，30，这对前四个元素是正确的。第三轮后得到 1，3，6，10，15，21，28，36，44，这对前八个元素是正确的。第四轮后得到 1，3，6，10，15，21，28，36，45，这就是最终的结果。虽然总步骤比第一种方法多，但如果有很多处理器，并行的空间就更大；每个处理器可以每个步骤更新一个条目。

总的来说，第二种方法需要 log2(N)步，大约进行 N 次操作(每个处理器一次)，其中 N 是列表中元素的数量。这与第一个算法相比，在第一个算法中，每个线程必须对分配给它的块的初始“部分和”执行 N/k 个操作，然后进一步 N/k 个操作来进行前向传播，其中 k 是线程数。因此，从总的操作数量来讲，第一种方法的时间复杂度为 O(N)，而第二种方法的时间复杂度为 Q(Nlog(N))。但是如果列表元素的数量和处理器数量一样，第二种方法只需要每个处理器执行 log(N)个操作，而第一种方法由于前向传播，在 k 变大时会串行化操作。对于少量的处理单元，第一种方法将因此完成得更快，而对于大规模并行系统，第二种方法将完成得更快。这是 8.2.1 节中讨论的问题的一个极端例子。

不管怎样，先不考虑效率问题，让我们来看一些代码。下面的清单显示了第一种方法。

清单 8.11 通过划分问题来并行计算部分和

```
template<typename Iterator>
void parallel_partial_sum(Iterator first,Iterator last)
{
  typedef typename Iterator::value_type value_type;

  struct process_chunk  // 1
  {
    void operator()(Iterator begin,Iterator last,
                    std::future<value_type>* previous_end_value,
                    std::promise<value_type>* end_value)
    {
      try
      {
```

```
      Iterator end=last;
      ++end;
      std::partial_sum(begin,end,begin);  // 2
      if(previous_end_value)  // 3
      {
        value_type& addend=previous_end_value->get();  // 4
        *last+=addend;  // 5
        if(end_value)
        {
          end_value->set_value(*last);  // 6
        }
        std::for_each(begin,last,[addend](value_type& item)  // 7
                      {
                        item+=addend;
                      });
      }
      else if(end_value)
      {
        end_value->set_value(*last);  // 8
      }
    }
    catch(...)  // 9
    {
      if(end_value)
      {
        end_value->set_exception(std::current_exception());  // 10
      }
      else
      {
        throw;  // 11
      }
    }
  }
};

unsigned long const length=std::distance(first,last);

if(!length)
  return last;

unsigned long const min_per_thread=25;  // 12
unsigned long const max_threads=
  (length+min_per_thread-1)/min_per_thread;
```

```
  unsigned long const hardware_threads=
    std::thread::hardware_concurrency();

  unsigned long const num_threads=
    std::min(hardware_threads!=0?hardware_threads:2,max_threads);

  unsigned long const block_size=length/num_threads;

  typedef typename Iterator::value_type value_type;

  std::vector<std::thread> threads(num_threads-1);  // 13
  std::vector<std::promise<value_type> >
    end_values(num_threads-1);  // 14
  std::vector<std::future<value_type> >
    previous_end_values;  // 15
  previous_end_values.reserve(num_threads-1);  // 16
  join_threads joiner(threads);

  Iterator block_start=first;
  for(unsigned long i=0;i<(num_threads-1);++i)
  {
    Iterator block_last=block_start;
    std::advance(block_last,block_size-1);  // 17
    threads[i]=std::thread(process_chunk(),  // 18
                           block_start,block_last,
                           (i!=0)?&previous_end_values[i-1]:0,
                           &end_values[i]);
    block_start=block_last;
    ++block_start;  // 19
    previous_end_values.push_back(end_values[i].get_future());  // 20
  }
  Iterator final_element=block_start;
  std::advance(final_element,std::distance(block_start,last)-1);  // 21
  process_chunk()(block_start,final_element,  // 22
                  (num_threads>1)?&previous_end_values.back():0,
                  0);
}
```

在本例中，大体结构与前面的算法相同：将问题划分为块，每个线程有最小的块大小⑫。本例中，除了线程的 vector 外⑬，还有承诺的 vector⑭，用来存储块中的最后一个元素的值，以及一个期望的 vector⑮，用于检索前一个块的最后一个值。可以为期望提前预留空间⑯以避免生成新线程时，再分配内存，因为你明确知道会有多少个期望。

主循环和之前一样，不过这次是让迭代器指向了每个数据块的最后一个元素，而不是通常的最后一个元素后面的一个位置⑰，这样就可以对每个范围内的最后一个元素进行前

向传播。处理是在 process_chunk 函数对象中完成的，我们很快就会看到；该块的开始迭代器和结束迭代器作为参数传入，同时传入前一个范围的结尾值的期望(如果有的话)以及保存该范围的结尾值的承诺⑱。

在生成线程之后，可以更新块的起始迭代器，注意记得将它前进到最后一个元素之后⑲，并将当前块中最后一个值的期望存储到期望的 vector 中，这样在下一次循环中它将被捡起⑳。

处理最后一个数据块前，需要获取最后一个元素的迭代器 ，这样就可以将其作为参数传入 process_chunk 中 。std::partial_sum 不会返回一个值，所以一旦最后一个数据块被处理后，不需要做任何事情。一旦所有线程的操作完成，操作就算完成了。

好了，现在来看一下 process_chunk 函数对象①，它完成了所有的工作。首先为整个块调用 std::partial_sum，包括最后一个元素②，不过接着需要知道当前块是否是第一块③。如果不是第一块，就会有一个来自前面块的 previous_end_value 值，所以需要等待这个值④。为了让算法的并行最大化，首先更新最后一个元素⑤，这样就能将这个值传递给下一个数据块(如果有下一个数据块的话)⑥。一旦完成最后一个元素的处理，就可以使用 std::for_each 和简单的 lambda 函数⑦对范围内剩余的元素进行更新。

如果没有 previous_end_value，那么你就是第一个数据块，所以只需要为下一个数据块更新 end_value⑧(同样，如果有下一个数据块的话——因为你可能是唯一的数据块)。

最后，如果有任意一个操作抛出异常，可以将其捕获⑨，并且存储到承诺中⑩，因此，如果下一个数据块尝试获取前一个数据块的最后一个值时④，异常会传播给下一个数据块。这将传播所有的异常到最后一个数据块，这时异常会重新抛出⑪，因为你知道你运行在主线程上。

因为线程间需要同步，这段代码不适合用 std::async 重写。任务在执行其他任务的过程中等待结果准备好，因此所有任务必须并发运行。

在解决了基于块的前向传播方法之后，让我们看看计算范围“部分和”的第二种方法。

**实现“部分和”的增量成对算法**

第二种通过越来越远相加元素来计算“部分和”的方法在处理器同步执行加法时效果最好。这种情况下，没有进一步同步的必要了，因为所有中间结果都直接传播到下一个需要它们的处理器上。但实践中，除了单个处理器可以同时在少量数据元素上同时执行相同指令的情况外（这就是所谓的*单指令-多数据流*(SIMD)指令），很少有这样的系统供使用。因此，你必须为一般情况设计代码，并在每一步显式地同步线程。

一种方法是使用*屏障*(barrier)——一种同步机制，它会导致线程等待，直到所需的线程数量到达屏障。一旦所有的线程都到达了屏障，它们就会被解除阻塞，可以继续运行。C++ 11 线程库没有直接提供这个设施，所以必须自己设计一个。

想象一下游乐场里的过山车。如果有相当数量的人在等待，游乐场的工作人员会确保在过山车离开站台之前每个座位都有人。屏障的工作原理也是一样的：你预先指定“座位”的数量，线程必须等待，直到所有的“座位”都被填满。一旦有足够的等待线程，它们都可以继续进行；屏障被重置并开始等待下一批线程。通常，这种结构是在循环中使用的，在循环中，相同的线程出现并等待下一次。这样做的目的是让线程保持步调一致，这样，一个线程就不会跑到其他线程的前面，从而打乱节奏。对于像这样的算法，如果出现这种情况将是灾难性的，因为失控的线程可能会修改其他线程仍在使用的数据或使用尚未正确更新的数据。

下面的清单展示了屏障的简单实现。

清单 8.12 简单的屏障类

```
class barrier
{
  unsigned const count;
  std::atomic<unsigned> spaces;
  std::atomic<unsigned> generation;

public:
  explicit barrier(unsigned count_):  // 1
    count(count_),spaces(count),generation(0)
  {}

  void wait()
  {
    unsigned const my_generation=generation;  // 2
    if(!--spaces)  // 3
    {
      spaces=count;  // 4
      ++generation;  // 5
    }
    else
    {
      while(generation==my_generation)  // 6
        std::this_thread::yield();  // 7
    }
  }
};
```

有了这个实现，就可以用“座位”的数量构造一个屏障①，这个数量存储在 count 变量中。起初，屏障中的 spaces 与 count 相等。当每个线程等待时，spaces 的数量递减③。当它变为 0 时，spaces 会重置为 count④，并且 generation 会递增以向其他线程发出信号，它们可以继续⑤。如果 spaces 没有到达 0，线程必须等待。这个实现使用了一个简单的自旋锁⑥，根据在 wait()开头检索到的值检查 generation②。因为 generation 只会

在所有线程都到达栅栏的时候更新⑤，在等待的时候调用 yield()⑦，因此等待线程不会在忙等待中占用 CPU。

当我说这个实现很简单时，我是认真的：它使用了自旋等待，因此对于线程可能等待很长时间的情况并不理想，而且如果在任何时候有超过 count 的线程可以潜在地调用 wait()，它就没法工作。如果需要处理这些场景中的任何一种，则必须使用更鲁棒(但更复杂)的实现来代替。我也一直坚持原子变量上的序列一致操作，因为这使事情更易思考，但你可能会潜在放松一些顺序约束。这种全局同步在大规模并行架构上是昂贵的，因为保持屏障状态的缓存行必须在所有涉及的处理器之间来回穿梭(参阅 8.2.2 节中关于乒乓缓存的讨论)，因此你必须非常小心地确保这是这里的最佳选择。如果你的 C++标准库提供并发规范中的设施，这里就可以使用 std::experimental::barrier，详见第 4 章。

这就是你需要的：有固定数量的线程需要在一个同步循环中运行。嗯，它几乎是固定数量的线程。你可能还记得，列表开头的项经过几个步骤后获得它们的最终值。这意味着，要么必须保持这些线程循环，直到处理完整个范围，要么需要允许屏障处理线程退出并递减 count。我选择后一个选项，因为它避免让线程做不必要的工作：一直循环直到完成最后一步。

这意味着你必须将 count 更改为一个原子变量，这样就可以从多个线程更新它，而不需要外部同步：

```
std::atomic<unsigned> count;
```

初始化保持不变，不过当重置 spaces 的时候，需要显式的对 count 进行 load()操作：

```
spaces=count.load();
```

这些是在 wait()前面需要的所有更改；现在需要一个新的成员函数来递减 count。让我们叫它 done_waiting()，因为线程正在声明它已经完成了等待：

```
void done_waiting()
{
  --count;  // 1
  if(!--spaces)  // 2
  {
    spaces=count.load();  // 3
    ++generation;
  }
}
```

做的第一件事是递减 count①，这样的话，下一次重置 spaces 时，它将反映新的更少数量的等待线程。然后，需要递减 spaces 的值②。如果不这样做，其他线程将永远等待，因为 spaces 被初始化为旧的、更大的值。如果你是这批的最后一个线程，则需要重置计数器并递增 generation③，就像在 wait()里面做的那样。这里的关键区别在于，如果你是批次中的最后一个线程，则不必等待。

现在可以编写“部分和”的第二种实现了。在每个步骤中，每个线程都调用屏障上的 wait()来确保线程一起完成步骤，一旦每个线程完成，它就调用屏障上的 done_waiting()来减少计数。如果和原始范围一起使用第二个缓冲区，屏障将提供所需的所有同步。在每一步中，线程从原始范围或缓冲区中读取值，并将新值写入另一边的相应元素。如果线程在一个步骤中从原始范围中读取数据，那么它们将在下一个步骤中从缓冲区中读取数据，反之亦然。这确保了不同线程的读写操作之间没有竞争条件。线程完成循环后，必须确保将正确的最终值写入原始范围。下面的清单将这一切整合在一起。

清单 8.13 partial_sum 的并行实现：通过成对更新

```
struct barrier
{
  std::atomic<unsigned> count;
  std::atomic<unsigned> spaces;
  std::atomic<unsigned> generation;

  barrier(unsigned count_):
    count(count_),spaces(count_),generation(0)
  {}

  void wait()
  {
    unsigned const gen=generation.load();
    if(!--spaces)
    {
      spaces=count.load();
      ++generation;
    }
    else
    {
      while(generation.load()==gen)
      {
        std::this_thread::yield();
      }
    }
  }

  void done_waiting()
  {
    --count;
    if(!--spaces)
    {
      spaces=count.load();
      ++generation;
```

```
    }
  }
};

template<typename Iterator>
void parallel_partial_sum(Iterator first,Iterator last)
{
  typedef typename Iterator::value_type value_type;

  struct process_element  // 1
  {
    void operator()(Iterator first,Iterator last,
                    std::vector<value_type>& buffer,
                    unsigned i,barrier& b)
    {
      value_type& ith_element=*(first+i);
      bool update_source=false;

      for(unsigned step=0,stride=1;stride<=i;++step,stride*=2) {
        value_type const& source=(step%2)?  // 2
          buffer[i]:ith_element;

        value_type& dest=(step%2)?
          ith_element:buffer[i];

        value_type const& addend=(step%2)?  // 3
          buffer[i-stride]:*(first+i-stride);

        dest=source+addend;  // 4
        update_source=!(step%2);
        b.wait();  // 5
      }
      if(update_source) { // 6
        ith_element=buffer[i];
      }
      b.done_waiting();  // 7
    }
  };

  unsigned long const length=std::distance(first,last);
  if(length<=1)
    return;
  std::vector<value_type> buffer(length);
  barrier b(length);
```

```
  std::vector<std::thread> threads(length-1);  // 8
  join_threads joiner(threads);
  Iterator block_start=first;
  for(unsigned long i=0;i<(length-1);++i) {
    threads[i]=std::thread(process_element(),first,last,  // 9
                           std::ref(buffer),i,std::ref(b));
  }
  process_element()(first,last,buffer,length-1,b);  // 10
}
```

到目前为止，代码的主体结构应该很熟悉了。你有一个带有函数调用操作符(process_element)的类来完成这项工作①，它在一堆线程上运行⑨，这些线程对象存储在vector中⑧，同时你也从主线程调用这个函数对象⑩。这次关键的区别是，线程的数量依赖于列表中项的个数，而不是std::thread::hardware_concurrency。正如我已经说过，除非你是在一个线程很廉价的大型并行机器上，否则这可能是个坏主意，但它展示了整体结构。使用更少的线程是可能的，只要让每个线程处理来自源范围的几个值，但总会有一个点，当线程足够少时，它比正向传播算法的效率要低。

主要的工作都是在process_element的函数调用操作符中完成的。在每一步中，你要么从原始范围中取出第i个元素，要么从缓冲区中取出第i个元素②，并将它加到前面相隔stride距离对应元素的值上③，如果是从原始范围开始，就把它存储在缓冲区中，如果是从缓冲区开始，就存回到原始范围中④。然后，开始下一步之前，在屏障上等待⑤。当stride将你带离范围的起点时，说明你已经完成了工作，在这种情况下，如果最终结果存储在缓冲区中，你需要更新原始范围中的元素⑥。最后，你通过done_waiting()函数告诉屏障完成等待⑦。

注意这个解决方案并不是异常安全的。如果某个工作线程在process_element中抛出一个异常，它会终止应用程序。可以通过使用std::promise存储异常来处理这个问题，就像你对清单8.9中的parallel_find实现所做的那样，或者甚至使用由互斥锁保护的std::exception_ptr。

我们的三个例子到此结束。希望它们有助于明确8.1、8.2、8.3和8.4节中强调的一些设计考量，并演示了如何将这些技术应用到实际代码中。

# 总结

在这一章中，我们已经讨论了相当多的内容。我们首先介绍了在线程之间划分工作的各种技术，例如预先划分数据或使用多个线程组成一条流水线。然后，我们从低层次的角度研究了与多线程代码性能相关的问题，分析了伪共享和数据竞争，然后继续讨论数据访问模式如何影响一段代码的性能。接着了解了设计并发代码时的额外考量，比如：异常安全和可扩展性。最后，我们以一系列并行算法实现的例子收尾，每个例子都突出了在设计多线程代码时可能出现的特定问题。

在本章中多次提到线程池的概念——一组预先配置的线程，它们运行分配给这个线程池的任务。设计一个好的线程池需要大量的思考，因此我们将在下一章中讨论这个问题，以及高级线程管理的其他方面。

# 第 9 章 高级线程管理

**本章主要内容**

- 线程池
- 处理线程池中任务间的依赖
- 为线程池中的线程窃取工作
- 中断线程

之前的章节中，我们通过为每个线程创建 std::thread 对象来显式的管理线程。但在某些地方，你已经看到这种方式不太可取，因为你必须管理线程对象的生命周期，根据当前的硬件和要解决的问题来确定合适的线程数量，等等诸如此类的问题。理想的场景是将代码分成可以并发执行的最小块，之后交给编译器和库，让它们以最优的性能并行执行。我们将在第 10 章中看到，有些情况可以这么做：如果你需要并行的代码可以用标准库算法的调用来表示，那么大多数情况下，你可以让库帮你搞定并发。

另一个在一些例子中反复出现的主题是：你可能使用一些线程来解决问题，但某些条件达成的时候需要提前结束。这可能是因为结果已经确定，或者发生了错误，又或是用户明确的要求终止操作。无论是哪种原因，都需要给线程发送“请停止”请求，让它们放弃给它们的任务，清理相关资源，然后尽快结束。

本章我们将讨论管理线程和任务的机制，让我们从自动管理线程的数量和在线程间划分任务开始。

## 9.1 线程池

在很多公司里，员工大部分时间是在办公室上班，但偶尔也会外出拜访客户或供应商，或是出席贸易展览或会议。尽管这些出行很有必要，并且在任何一天可能有好几个人出行，但对特定员工来说，出行的频率可能是几个月一次甚至几年一次。由于给每个员工都配一辆公车太过昂贵也不切实际，取而代之公司通常提供合伙用车；这样就有有限数量的车来供所有员工使用。当员工需要外出时，他们就可以在合适的时间从共用车辆中预定一辆，并在返回公司的时候把车交还给别人使用。如果某天没有闲置的共用车辆，员工就得延后出行。

线程池就是一种类似的想法，只不过共享的是线程而不是车。在大多数系统中，为每个可以并发的任务分配一个单独的线程是不切实际的，不过你还是想尽可能利用可用的并发。线程池就允许你达成这个目标，你可以把并发执行的任务提交到一个线程池中，也就是把它们放到待处理工作队列。工作线程从队列中取出任务,然后执行，之后又循环到线程池中获取另一个任务。

构建一个线程池时，会遇到几个关键性的设计问题，比如：使用多少线程，最有效地给线程分配任务的方式，以及是否需要等待一个任务完成。本节我们将看看解决这些设计问题的一些线程池实现，让我们从最简单的线程池开始吧。

### 9.1.1 最简单的线程池

简单来说，线程池是固定数量的处理工作的工作线程(通常工作线程数量与std::thread::hardware_concurrency()返回值相同)。当有工作可做时，可以调用函数将它放入待处理工作队列。每个工作线程都会从队列上获取工作，然后运行指定的任务，最后再回到队列获取更多的工作。最简单的线程池中，没有方法等待任务完成。如果需要等待，就需要自己管理同步。

下面的清单展示了一个线程池的示例实现。

清单 9.1 简单的线程池

```
class thread_pool
{
  std::atomic_bool done;
  thread_safe_queue<std::function<void()> > work_queue;  // 1
  std::vector<std::thread> threads;  // 2
  join_threads joiner;  // 3

  void worker_thread()
  {
    while(!done)  { // 4
      std::function<void()> task;
      if(work_queue.try_pop(task)) { // 5
        task();  // 6
      } else {
        std::this_thread::yield();  // 7
      }
    }
  }

public:
  thread_pool():
    done(false),joiner(threads)
  {
    unsigned const thread_count=std::thread::hardware_concurrency();  /
/ 8

    try {
      for(unsigned i=0;i<thread_count;++i) {
        threads.push_back(
          std::thread(&thread_pool::worker_thread,this));  // 9
      }
    } catch(...) {
```

```
      done=true;  // 10
      throw;
    }
  }
  ~thread_pool()
  {
    done=true;  // 11
  }

  template<typename FunctionType>
  void submit(FunctionType f)
  {
    work_queue.push(std::function<void()>(f));  // 12
  }
};
```

这个实现中有一个工作线程的 vector②，并且使用第 6 章中的线程安全队列①来管理工作队列。这里用户没法等待任务，并且它们也没法返回任何值，所以可以使用 std::function<void()>对任务进行封装。submit()会将任何函数或可调用对象包装成一个 std::function<void()>实例，并将其推入到队列中⑫。

线程在构造函数中启动：使用 std::thread::hardware_concurrency()来获取硬件支持多少个并发线程⑧，然后创建这么多数量的线程运行 worker_thread()成员函数⑨。

当有异常抛出时，线程启动就会失败，所以你需要保证任何已启动的线程都能停止，并正确的清理。当有异常抛出时，这是通过 try-catch 块设置 done 标志来实现的⑩，配套还有来自于第 8 章的 join_threads 类实例③来连接所有线程。这也适用于析构函数：析构函数设置 done 标志⑪，并且 join_threads 确保所有线程在线程池销毁前全部执行完成。注意成员声明的顺序很重要：done 标志和 worker_queue 必须在 threads vector 之前声明，而数组必须在 joiner 前声明。这就能确保成员以正确的顺序销毁；例如，所有线程都停止运行时，队列就可以安全的销毁。

worker_thread 函数本身很简单：它一直循环直到 done 标志被设置④,不断从队列上拉取任务⑤，同时执行它们⑥。如果任务队列上没有任务，函数会调用 std::this_thread::yield()让线程短暂停歇⑦，从而给予其他线程向队列上推送任务的机会。

对许多用途而言，这个简单的线程池就足够了，特别当任务完全独立，并且没有返回值或执行任何阻塞操作的时候。不过，还有很多情况下这样简单的线程池无法满足你的需求，还有些地方可能会出现诸如死锁的问题。同样，在简单情况下使用 std::async 是个更好的选择，就像第 8 章中的例子。贯穿本章，我们将研究更加复杂的线程池实现，它们有额外的特性用来满足用户需求，或是减少潜在的问题。首先，从等待已提交的任务开始说起。

### 9.1.2 等待提交到线程池中的任务

在第 8 章的例子中，线程间的工作划分完成后，代码会显式生成新线程，主线程总是等待新线程结束，从而确保在返回调用之前所有任务都完成。使用线程池的话，需要等待提交给线程池的任务完成，而不是工作线程本身完成。这和第 8 章中基于 std::async 的例子等待期望的方法类似，使用清单 9.1 中的简单线程池，你需要手工使用第 4 章中提到的技术：条件变量和期望。这虽然会增加代码的复杂度，但要能直接等待任务会比较好。

通过把复杂性转移到线程池本身，可以直接等待任务完成。可以让 submit()函数返回某种描述的任务句柄，你可以用它来等待任务的完成。任务句柄会包装条件变量或期望的使用，从而简化使用线程池的代码。

需要等待生成任务完成的一种特殊情况是主线程需要任务的计算结果。本书已经出现了多个这样的例子，比如第 2 章中的 parallel_accumulate()。这种情况下，需要用期望来等待并获取最终结果。清单 9.2 展示了对简单线程池的修改，从而允许等待任务完成，然后从任务传递返回值给等待线程，由于 std::packaged_task<>实例是不可拷贝的，只能移动，所以不能把 std::function<>用作队列条目，因为 std::function<>需要存储可拷贝构造的函数对象。取而代之，需要使用一个自定义函数包装器，用来处理只可移动的类型。这是个带有调用操作符的类型擦除类。由于只需要处理没有入参也无返回的函数，所以实现中是个简单的虚函数调用。

清单 9.2 带有可等待任务的线程池

```
class function_wrapper
{
  struct impl_base {
    virtual void call()=0;
    virtual ~impl_base() {}
  };

  std::unique_ptr<impl_base> impl;
  template<typename F>
  struct impl_type: impl_base
  {
    F f;
    impl_type(F&& f_): f(std::move(f_)) {}
    void call() { f(); }
  };
public:
  template<typename F>
  function_wrapper(F&& f):
    impl(new impl_type<F>(std::move(f)))
  {}
```

```
  void operator()() { impl->call(); }

  function_wrapper() = default;

  function_wrapper(function_wrapper&& other):
    impl(std::move(other.impl))
  {}

  function_wrapper& operator=(function_wrapper&& other)
  {
    impl=std::move(other.impl);
    return *this;
  }

  function_wrapper(const function_wrapper&)=delete;
  function_wrapper(function_wrapper&)=delete;
  function_wrapper& operator=(const function_wrapper&)=delete;
};

class thread_pool
{
  thread_safe_queue<function_wrapper> work_queue;  // 使用
function_wrapper，而非 std::function

  void worker_thread()
  {
    while(!done)
    {
      function_wrapper task; // 使用 function_wrapper，而非 std::function
      if(work_queue.try_pop(task))
      {
        task();
      }
      else
      {
        std::this_thread::yield();
      }
    }
  }

public:
  template<typename FunctionType>
  std::future<typename std::result_of<FunctionType()>::type>  // 1
    submit(FunctionType f)
```

```
  {
    typedef typename std::result_of<FunctionType()>::type
      result_type;  // 2

    std::packaged_task<result_type()> task(std::move(f));  // 3
    std::future<result_type> res(task.get_future());  // 4
    work_queue.push(std::move(task));  // 5
    return res;  // 6
  }
  // 剩下的和之前一样
};
```

首先，修改的 submit()函数①返回一个保存任务返回值的 std::future<>，并且允许调用者等待任务完成。因为需要知道函数 f 的返回类型，这就是使用 std::result_of<>的原因：std::result_of<FunctionType()>::type 是调用 FunctionType(如 f)类型实例的返回值的类型，这个类型没有参数。函数中对 result_type typedef②也使用了 std::result_of<>表达式。

然后，将 f 包装到 std::packaged_task<result_type()>③，因为 f 是一个无参数的函数或可调用对象，并且返回 result_type 类型的实例。在把任务推入队列⑤前，你现在可以从 std::packaged_task<>中获取期望④，然后返回期望⑥。注意，要将任务推送到队列中时，只能使用 std::move()，因为 std::packaged_task<>是不可拷贝的。为了处理处理这种情况，队列里面存放的是 function_wrapper 对象，而非 std::function<void()>对象。

这个线程池允许等待任务，并且返回它们的结果。下面的清单展示了 parallel_accumuate 是怎么使用线程池的。

清单 9.3 parallel_accumulate 使用一个带可等待任务的线程池

```
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
  unsigned long const length=std::distance(first,last);

  if(!length)
    return init;

  unsigned long const block_size=25;
  unsigned long const num_blocks=(length+block_size-
1)/block_size;  // 1

  std::vector<std::future<T> > futures(num_blocks-1);
  thread_pool pool;
```

```
  Iterator block_start=first;
  for(unsigned long i=0;i<(num_blocks-1);++i)
  {
    Iterator block_end=block_start;
    std::advance(block_end,block_size);
    futures[i]=pool.submit([=]{
      accumulate_block<Iterator,T>()(block_start,block_end);
    }); // 2
    block_start=block_end;
  }
  T last_result=accumulate_block<Iterator,T>()(block_start,last);

  T result=init;
  for(unsigned long i=0;i<(num_blocks-1);++i)
  {
    result+=futures[i].get();
  }
  result += last_result;
  return result;
}
```

与清单 8.4 相比，需要注意几件事。首先，你的工作是基于有多少块可用(num_blocks)①，而不是线程的数量。为了充分利用线程池的可扩展性，需要将工作块划分为适合并行处理的最小工作块。当线程池中线程不多时，每个线程将会处理多个工作块，不过随着硬件可用线程数量的增长，并发处理的块也会随着增长。

当选择“适合并行处理的最小工作块”时，需要仔细斟酌。向线程池提交任务是有内在开销的；让工作线程运行这个任务，并且通过 std::future<>传输返回值，对于太小的任务，这样的开销不划算。如果任务块太小，使用线程池的运行速度可能比单线程要慢。

假设块的大小合理，那就没必要担心打包任务、获取 future 或存储 std::thread 对象以供后面连接用；使用线程池的时候会妥善处理这些。你需要做的就是调用 submit()来提交任务②。

线程池也是异常安全的。任何抛出的异常都会通过 submit()返回的期望值传递。如果函数因为异常退出，线程池的析构函数会放弃那些没有完成的任务，等待线程池中的线程结束。

像这种简单的例子，这个线程池工作的很好，因为任务都是相互独立的。不过，当任务队列中的任务有依赖关系时，这个线程池就不能胜任了。

### 9.1.3 等待其它任务的任务

快速排序算法是一个贯穿本书的示例。它概念上很简单：需要排序的数据根据主元项，分区为排在前面和后面的两个集合。这两个集合会递归排序，然后缝合在一起组成一个全排序序列。当要将算法并行起来的时候，需要确保递归调用能够使用可用的并发。

回到第 4 章，当我第一次介绍这个例子的时候，我们使用 std::async 来运行每一阶段的递归调用，让库来选择是在新线程上运行，还是当调用相关的 get()时同步运行。这种方式工作的很好，因为每一个任务要么在它自己的线程上运行，要么在需要的时候被调用。

当我们回顾第 8 章中的实现，可以看到使用了一个和硬件并发相关的固定线程数量的结构体。这种情况下，使用栈来处理要排序的数据块。因为每个线程分区它要排序的数据，它会往栈中增加一个新的数据块，然后直接对另一块进行排序。此时直接等待其他线程完成排序会造成死锁，因为这时候你消耗了有限线程中的一个来等待。很容易就会出现一种情况：所有线程都在等某一个数据块被排序，但没有线程在做排序。当线程等待的特定块没有排好序的时候，我们通过让它们拉取栈上数据块并对数据块进行排序，来解决这个问题。

如果用本章截止目前看到的简单线程池替换第 4 章中的 std::async，你会碰到同样的问题。现在只有有限数量的线程，而且，由于没有空闲线程，它们可能最终都在等待未被调度的任务。因此，需要和第 8 章中类似的解决方案：当等待某个数据块完成时，去处理未完成的数据块。如果你正在使用线程池管理任务列表和列表对应的线程——毕竟这是使用线程池的主要原因——你没有权限去访问任务列表了，所以也不必管理它。你需要做的就是修改线程池来自动管理。

最简单的方法就是在 thread_pool 中添加一个新函数来运行队列上的任务，并且自己管理循环，因此我们会采用这种方法。高级的线程池实现可能会在等待函数或额外的等待函数中添加逻辑来处理这种情况，可能让等待的任务按优先级排序。下面清单中的实现，展示了一个新的 run_pending_task()函数，使用它的修改过的快速排序在清单 9.5 中展示。

清单 9.4 run_pending_task()函数实现

```
void thread_pool::run_pending_task()
{
  function_wrapper task;
  if(work_queue.try_pop(task))
  {
    task();
  }
  else
  {
    std::this_thread::yield();
```

```
  }
}
```

run_pending_task()的实现直接取自worker_thread()函数的主循环，现在可以修改它来调用提取出的run_pending_task()。它尝试从队列获取任务，有的话就运行，要是没有的话让出CPU从而允许操作系统重新调度线程。清单9.5中的快速排序算法要比清单8.1中的版本简单很多，因为所有线程管理逻辑都被移入到线程池。

清单9.5 基于线程池的快速排序实现

```
template<typename T>
struct sorter  // 1
{
  thread_pool pool;  // 2

  std::list<T> do_sort(std::list<T>& chunk_data)
  {
    if(chunk_data.empty())
    {
      return chunk_data;
    }

    std::list<T> result;
    result.splice(result.begin(),chunk_data,chunk_data.begin());
    T const& partition_val=*result.begin();

    typename std::list<T>::iterator divide_point=
      std::partition(chunk_data.begin(),chunk_data.end(),
                     [&](T const& val){return val<partition_val;});

    std::list<T> new_lower_chunk;
    new_lower_chunk.splice(new_lower_chunk.end(),
                           chunk_data,chunk_data.begin(),
                           divide_point);
    std::future<std::list<T> > new_lower=  // 3
      pool.submit(std::bind(&sorter::do_sort,this,
                            std::move(new_lower_chunk)));

    std::list<T> new_higher(do_sort(chunk_data));

    result.splice(result.end(),new_higher);
    while(new_lower.wait_for(std::chrono::seconds(0)) !=
      std::future_status::ready)
    {
      pool.run_pending_task();  // 4
```

```
    }

    result.splice(result.begin(),new_lower.get());
    return result;
  }
};

template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
  if(input.empty())
  {
    return input;
  }
  sorter<T> s;

  return s.do_sort(input);
}
```

和清单 8.1 一样，这里将实际工作委托给 sorter 类模板的 do_sort()成员函数①，尽管这里这个类只是包装了一下 thread_pool 实例②。

线程和任务管理现在减少到只需要提交任务到线程池③，以及等待的时候运行待处理的任务④。这比清单 8.1 简单很多，在那里需要显式的管理线程以及栈上要排序的数据块。当要提交任务到线程池中，使用 std::bind()绑定 this 指针到 do_sort()上，并提供要排序的数据块。这种情况下，在 new_lower_chunk 上调用 std::move()，来确保移动数据而非拷贝。

虽然，这解决了因为等待其他任务而引起的死锁问题，但这个线程池还很不理想。首先，每次对 submit()和 run_pending_task()的调用，访问的都是同一个队列。第 8 章中你已经看到让多线程去修改一个单一的数据集，对性能有不利影响，所以需要解决这个问题。

### 9.1.4 避免在工作队列上竞争

线程每次调用线程池的 submit()函数，都会推送一个新的任务到唯一的工作队列中。同样，工作线程为了运行任务会持续从队列中弹出任务。这意味着随着处理器的增加，任务队列上的竞争也会增加，这会让性能下降。即使使用无锁队列，因而没有明显的等待，但乒乓缓存会带来大量时间消耗。

为了避免乒乓缓存，一种方法是每个线程使用独立的工作队列。每个线程会将新任务放在自己的队列上，并且只有当线程上的队列没有工作时，才去全局的工作队列中获取工作。下面的清单展示了使用一个 thread_local 变量来保证每个线程都拥有自己的任务列表，当然还有全局队列。

清单 9.6 使用线程局部工作队列的线程池

```
class thread_pool
{
  thread_safe_queue<function_wrapper> pool_work_queue;

  typedef std::queue<function_wrapper> local_queue_type;  // 1
  static thread_local std::unique_ptr<local_queue_type>
    local_work_queue;  // 2

  void worker_thread()
  {
    local_work_queue.reset(new local_queue_type);  // 3
    while(!done) {
      run_pending_task();
    }
  }

public:
  template<typename FunctionType>
  std::future<typename std::result_of<FunctionType()>::type>
    submit(FunctionType f)
  {
    typedef typename std::result_of<FunctionType()>::type result_type;
    std::packaged_task<result_type()> task(f);
    std::future<result_type> res(task.get_future());
    if(local_work_queue)  { // 4
      local_work_queue->push(std::move(task));
    } else {
      pool_work_queue.push(std::move(task));  // 5
    }
    return res;
  }

  void run_pending_task()
  {
    function_wrapper task;
    if(local_work_queue && !local_work_queue->empty()) { // 6
      task=std::move(local_work_queue->front());
      local_work_queue->pop();
      task();
    } else if(pool_work_queue.try_pop(task))  { // 7
      task();
    } else {
```

```
      std::this_thread::yield();
    }
  }
// 剩下和之前一样
};
```

因为不希望非线程池中的线程也拥有一个任务队列，所以使用 std::unique_ptr<>持有线程局部工作队列②；这个指针在 worker_thread()函数循环处理之前进行初始化③。std:unique_ptr<>的析构函数会保证在线程退出的时候，工作队列被销毁。

然后 submit()会检查当前线程是否拥有一个工作队列④。如果有，它就是线程池中的线程，可以将任务放入线程的局部队列中；否则，就像之前一样将这个任务放在线程池队列中⑤。

run_pending_task()⑥中也有类似检查，只是这次还要检查局部队列是否有任务。如果有的话，就从前面取一个来处理；注意局部任务队列可以是一个普通的 std::queue<>①，因为这个队列只会被一个线程访问。如果局部队列上没有任务，就和之前一样，尝试从线程池队列上获取任务⑦。

这对减少竞争很有用，不过当工作分配不均时，很容易导致某个线程局部队列中有很多任务，同时其他线程无事可做。例如，在快速排序的例子中，只有最顶层的数据块能放入到线程池队列中，因为剩余数据块会放在工作线程的局部队列上进行处理。这就违背使用线程池的目的。

好在这个问题有解决方案：如果局部工作队列和全局工作队列上没有任务时，允许线程可以从彼此的队列中窃取工作。

## 9.1.5 工作窃取

为了让没有工作的线程能从另一个队列满的线程中获取工作，就需要这个队列可以被窃取线程访问，窃取线程会在 run_pending_tasks()中执行窃取动作。这需要每个线程在线程池中注册它的队列，或由线程池提供队列。同样，还需要确保工作队列中的数据有适当的同步和保护，这样队列的不变量就受到保护。

实现一个无锁队列是可能的，让其所属线程在一端推入或弹出工作的同时其他线程从另一端窃取工作，不过，这种队列的实现超出了本书的讨论范围。为了演示这个想法，我们将坚持使用一个互斥锁来保护队列中的数据。我们希望窃取工作是罕见的事件，这样在互斥锁上只有很少的竞争，因此这个简单队列的开销应该很小。一个简单的基于锁的实现如下。

```
清单 9.7 用于工作窃取的基于锁的队列

class work_stealing_queue
{
```

```
private:
  typedef function_wrapper data_type;
  std::deque<data_type> the_queue;  // 1
  mutable std::mutex the_mutex;

public:
  work_stealing_queue()
  {}
  work_stealing_queue(const work_stealing_queue& other)=delete;
  work_stealing_queue& operator=(
    const work_stealing_queue& other)=delete;

  void push(data_type data)  // 2
  {
    std::lock_guard<std::mutex> lock(the_mutex);
    the_queue.push_front(std::move(data));
  }

  bool empty() const
  {
    std::lock_guard<std::mutex> lock(the_mutex);
    return the_queue.empty();
  }

  bool try_pop(data_type& res)  // 3
  {
    std::lock_guard<std::mutex> lock(the_mutex);
    if(the_queue.empty()) {
      return false;
    }
    res=std::move(the_queue.front());
    the_queue.pop_front();
    return true;
  }

  bool try_steal(data_type& res)  // 4
  {
    std::lock_guard<std::mutex> lock(the_mutex);
    if(the_queue.empty()) {
      return false;
    }
    res=std::move(the_queue.back());
    the_queue.pop_back();
    return true;
```

```
  }
};
```

这个队列是对 std::deque<fuction_wrapper>的简单包装①，这样就能使用一个互斥锁对所有访问进行保护。push()②和 try_pop()③都工作在队列的前部，与此同时 try_steal()④工作在队列后部。

这意味着“队列”是它所属线程的一个后进先出栈，最新推入的任务又会第一个弹出。从缓存视角来看，这将有助于提升性能，因为相比于早前推入到队列的线程相关的数据，最近推入的数据很可能仍在缓存中。而且，它很好的映射到譬如快速排序这样的算法。之前的实现中，每次调用 do_sort()都会推送一个任务到栈上，并且等待这个任务结束。通过首先处理最近推入的任务，就可以保证当前调用需要的数据块在其他分支需要的数据块之前被处理，从而可以减少活跃任务的数量以及对栈的使用量。相对于 try_pop()，try_steal()从队列另一端获取任务是为了让竞争最小化；可以使用在第 6、7 章中的讨论的技术来让 try_pop()和 try_steal()并发执行。

好了，现在有了一个很赞的支持窃取的工作队列；怎么在线程池中使用呢？这里是个可能的实现。

清单 9.8 使用任务窃取的线程池

```
class thread_pool
{
  typedef function_wrapper task_type;

  std::atomic_bool done;
  thread_safe_queue<task_type> pool_work_queue;
  std::vector<std::unique_ptr<work_stealing_queue> > queues;  // 1
  std::vector<std::thread> threads;
  join_threads joiner;

  static thread_local work_stealing_queue* local_work_queue;  // 2
  static thread_local unsigned my_index;

  void worker_thread(unsigned my_index_)
  {
    my_index=my_index_;
    local_work_queue=queues[my_index].get();  // 3
    while(!done) {
      run_pending_task();
    }
  }

  bool pop_task_from_local_queue(task_type& task)
  {
```

```
    return local_work_queue && local_work_queue->try_pop(task);
  }

  bool pop_task_from_pool_queue(task_type& task)
  {
    return pool_work_queue.try_pop(task);
  }

  bool pop_task_from_other_thread_queue(task_type& task)  // 4
  {
    for(unsigned i=0;i<queues.size();++i) {
      unsigned const index=(my_index+i+1)%queues.size();  // 5
      if(queues[index]->try_steal(task)) {
        return true;
      }
    }
    return false;
  }

public:
  thread_pool():
    done(false),joiner(threads)
  {
    unsigned const thread_count=std::thread::hardware_concurrency();
    try {
      for(unsigned i=0;i<thread_count;++i) {
        queues.push_back(std::unique_ptr<work_stealing_queue>(  // 6
                         new work_stealing_queue));
        threads.push_back(
          std::thread(&thread_pool::worker_thread,this,i));
      }
    } catch(...) {
      done=true;
      throw;
    }
  }
  ~thread_pool()
  {
    done=true;
  }

  template<typename FunctionType>
  std::future<typename std::result_of<FunctionType()>::type> submit(
    FunctionType f)
```

```
  {
    typedef typename std::result_of<FunctionType()>::type result_type;
    std::packaged_task<result_type()> task(f);
    std::future<result_type> res(task.get_future());
    if(local_work_queue) {
      local_work_queue->push(std::move(task));
    } else {
      pool_work_queue.push(std::move(task));
    }
    return res;
  }

  void run_pending_task()
  {
    task_type task;
    if(pop_task_from_local_queue(task) ||  // 7
       pop_task_from_pool_queue(task) ||  // 8
       pop_task_from_other_thread_queue(task))  { // 9
      task();
    } else {
      std::this_thread::yield();
    }
  }
};
```

这段代码与清单 9.6 类似。第一个不同点是每个线程都有一个 work_stealing_queue，而非普通的 std::queue<>②。当每个线程被创建时，并不分配属于自己的工作队列，而是线程池的构造函数为它分配一个⑥，然后存储到线程池的工作队列列表中①。列表中队列的索引，会传递给线程函数，然后使用索引来检索指向队列的指针③。这就意味着为没有工作可做的线程尝试窃取任务的时候，线程池可以访问这个队列。run_pending_task()现在将尝试从线程自己对应的队列中取出一个任务⑦，从线程池队列中获取一个任务⑧，或从其他线程的队列中获取一个任务⑨。

pop_task_from_other_thread_queue()④会遍历线程池中所有线程的任务队列，然后依次尝试窃取一个任务。为了避免每个线程都尝试从列表中的第一个线程上窃取任务，每一个线程都会从自己对应的序号的下一个偏移开始探测。

现在有了一个可行的线程池，对很多使用场景都有帮助。当然对于特定的用途，还有无数方法可以提升它，不过这就留给读者作为练习吧。有个还没探索的方向是可以动态调整大小的线程池，它用来确保最优的 CPU 使用率，即使当线程阻塞等待 I/O 或互斥锁时也是如此。

下一个“高级”线程管理的技术是中断线程。

# 9.2 中断线程

在很多情况下，希望能发信号给一个长时间运行的线程，告诉它是时候停止了。这种线程可能是因为它是某个线程池的工作线程，并且线程池现在要被销毁，或是用户显式的取消了线程上的工作，或其他各种原因。不管是什么原因，想法都一样：需要从一个线程发信号告诉另一个线程在它自然结束前需要停止运行。这需要一种方法让线程优雅的结束，而非粗暴地中断。

在需要这么做的地方，你可能会给每种情况量身定做一种机制，但这样过犹不及。不仅因为使用通用的机制会让后面的场合编码更容易，而且允许你写的代码可中断，也不用担心代码用在哪。C++11 标准没有提供这样的机制(尽管有个积极的提案希望在以后的 C++ 标准中添加对中断的支持[1])，不过实现这样的机制比较简单。让我们看下如何实现这种机制，我们首先从启动和中断一个线程的接口视角开始，而非被中断线程视角。

## 9.2.1 启动和中断线程

首先看一下外部接口，对可中断线程有什么需求？最起码需要和 std::thread 有相同的接口，以及一个额外的 interrupt()函数：

```
class interruptible_thread
{
public:
  template<typename FunctionType>
  interruptible_thread(FunctionType f);
  void join();
  void detach();
  bool joinable() const;
  void interrupt();
};
```

内部可以使用 std::thread 来管理线程本身，并且使用一些自定义数据结构来处理中断。现在，从线程自身的角度来看呢？起码你要能说“我可以在这里中断”——你需要一个*中断点*（interruption point）。在不向下传递额外数据的前提下，为了使中断点可用，它需要是一个没有参数的简单函数：interruption_point()。这意味着中断相关的数据结构需要通过 thread_local 变量访问，并在线程启动时，对其进行设置，因此当线程调用 interruption_point()函数时，会去检查当前运行线程的数据结构。我们将在后面看到 interruption_point()的实现。

这个 thread_local 标志是不能使用普通的 std::thread 管理线程的主要原因；它需要使用某种方法来分配，使得可被 interruptible_thread 实例访问，还有被新启动的线程访问。你可以在构造函数中，在你把它传递给 std::thread 来启动线程之前，通过包装提供的函数来实现，如下所示。

清单 9.9 interruptible_thread 的基本实现

```
class interrupt_flag
{
public:
  void set();
  bool is_set() const;
};
thread_local interrupt_flag this_thread_interrupt_flag;  // 1

class interruptible_thread
{
  std::thread internal_thread;
  interrupt_flag* flag;

public:
  template<typename FunctionType>
  interruptible_thread(FunctionType f)
  {
    std::promise<interrupt_flag*> p;  // 2
    internal_thread=std::thread([f,&p]{  // 3
      p.set_value(&this_thread_interrupt_flag);
      f();  // 4
    });
    flag=p.get_future().get();  // 5
  }

  void interrupt()
  {
    if(flag)
    {
      flag->set();  // 6
    }
  }
};
```

提供的函数 f 包装在一个 lambda 函数中③，这个 lambda 持有 f 的拷贝和局部承诺 p 的引用②。在调用提供函数的副本拷贝前，lambda 函数为新线程设置 promise 变量的值为 this_thread_interrupt_flag(它声明为 thread_local①)的地址④。调用线程会等待与承诺相关的期望变成就绪状态，并将结果存入到 flag 成员变量中⑤。注意到即使 lambda 函数在新线程上执行，有个悬垂引用指向局部变量 p 是没有问题的，因为在返回之前，interruptible_thread 构造函数会一直等待，直到变量 p 不再被新线程引用。注意，实现没有考虑连接或分离线程，所以需要确保 flag 变量在线程退出或分离时被清理，以避免悬垂指针。

interrupt()函数就相对简单：如果你有一个有效的指针指向一个中断标志，你就有一个线程可以被中断，所以可以设置标志⑥。这时就依赖于被中断的线程怎么处理中断。让我们接下来探索这个问题。

### 9.2.2 检测线程是否被中断

现在你可以设置中断标志，但是如果线程不检查它是否被中断，就没什么卵用。最简单的情况下，你可以使用 interruption_point()函数来检查；在你觉得可以安全中断的地方调用这个函数，如果标记已经设置，就可以抛出一个 thread_interrupted 异常：

```
void interruption_point()
{
  if(this_thread_interrupt_flag.is_set())
  {
    throw thread_interrupted();
  }
}
```

你可以在代码中方便的地方使用这个函数：

```
void foo()
{
  while(!done)
  {
    interruption_point();
    process_next_item();
  }
}
```

尽管这行得通，但并不理想。中断线程最佳的地方是它阻塞等待的时候，这意味着线程不是为了调用 interruption_point()函数而运行的！这里你需要某种方法以可中断的方式等待。

### 9.2.3 中断在条件变量上的等待

好了，通过显式调用 interruption_point()，你可以在代码中仔细选择的位置检测中断，不过线程要阻塞等待时，这种办法就帮不上忙，例如等待条件变量的通知。你需要一个新的函数——interruptible_wait()——然后你可以为各种你希望等待的对象重载它，这样就可以解决如何中断等待。我之前提到，你可能在等待条件变量，所以让我们从它开始：为了能中断一个等待的条件变量，需要做哪些事情呢？最简单的方式是，一旦设置中断标志就通知条件变量，并在等待后面立即安放中断点。但要让它起作用，必须通知所有等待对应条件变量的线程，从而确保你关心的线程能够唤醒。不管怎样，等待线程都要处理伪唤醒，所以其他线程对这个唤醒当做伪唤醒来处理——两者之间没什么区别。

interrupt_flag 结构需要存储一个指向条件变量的指针，这样调用 set()就能被通知到。为条件变量实现的 interruptible_wait()看起来可能像下面的清单。

清单 9.10 为 std::condition_variable 实现的有问题的 interruptible_wait

```
void interruptible_wait(std::condition_variable& cv,
std::unique_lock<std::mutex>& lk)
{
  interruption_point();
  this_thread_interrupt_flag.set_condition_variable(cv);  // 1
  cv.wait(lk);  // 2
  this_thread_interrupt_flag.clear_condition_variable();  // 3
  interruption_point();
}
```

假设设置和清除与中断标志相关连的条件变量的函数已经实现，这个代码很好很简单。它检查中断，为当前线程关联条件变量到 interrupt_flag①，等待条件变量②，清理相关的条件变量③，并再次检查中断。如果线程在等待条件变量期间被中断，中断线程将广播条件变量，并把你从等待中唤醒，这样你就可以检查中断了。不幸的是，代码有问题：有两个问题。如果你想要异常安全，第一个问题就比较明显：std::condition_variable 可以抛出异常，因此你可能退出函数，但没有去掉中断标志关联的条件变量。这个问题很容易通过一个结构在析构函数中移除关联来解决。

第二个问题不太明显，这里存在竞争条件。如果在开始的 interruption_point()被调用后，但在调用 wait()前线程被中断，这时候条件变量和相关中断标志是否关联无关紧要，因为线程不是等待状态，所以不能通过通知条件变量的方式唤醒。你需要确保线程不会在上一次中断检查和调用 wait()间被通知。不探究 std::condition_variable 内部结构的话，你只有一种方法做这个事情：使用 lk 持有的互斥锁对线程进行保护，这就需要将 lk 传递到 set_condition_variable()函数中去。不幸的是，这又会产生它自己的问题：需要传递一个互斥锁(它的生命周期未知)的引用，到另一个线程中去(这个线程做中断操作)，这个互斥锁被该线程用来上锁(在 interrupt()调用中)，当它调用 interrupt()的时候不知道那个线程是否已经锁住了互斥锁。因此这里可能会死锁，并且可能访问到一个已经销毁的互斥锁，所以这种方法行不通。如果不能可靠地中断条件变量等待，那它的限制就太大了——即使没有特殊的 interruptible_wait()你可以做到几乎同样好——那你还有别的选择吗？一个选择就是给等待设置超时，使用 wait_for()而不是 wait()并带有一个比较小的超时(比如 1ms)。在线程看到中断前，给线程设置了一个等待的上限(受限于时钟计次粒度)。如果这样做的话，等待线程将会看到更多因为超时造成的伪唤醒，但又无法轻易避免。下面的清单展示了这个实现，随之一起的还有 interrupt_flag 的实现。

清单 9.11 为 std::condition_variable 在 interruptible_wait 中使用超时

```
class interrupt_flag {
  std::atomic<bool> flag;
  std::condition_variable* thread_cond;
  std::mutex set_clear_mutex;
```

```
public:
  interrupt_flag() : thread_cond(0) {}

  void set()
  {
    flag.store(true,std::memory_order_relaxed);
    std::lock_guard<std::mutex> lk(set_clear_mutex);
    if(thread_cond) {
      thread_cond->notify_all();
    }
  }

  bool is_set() const
  {
    return flag.load(std::memory_order_relaxed);
  }

  void set_condition_variable(std::condition_variable& cv)
  {
    std::lock_guard<std::mutex> lk(set_clear_mutex);
    thread_cond=&cv;
  }

  void clear_condition_variable()
  {
    std::lock_guard<std::mutex> lk(set_clear_mutex);
    thread_cond=0;
  }

  struct clear_cv_on_destruct {
    ~clear_cv_on_destruct()
    {
      this_thread_interrupt_flag.clear_condition_variable();
    }
  };
};

void interruptible_wait(std::condition_variable& cv,
  std::unique_lock<std::mutex>& lk)
{
  interruption_point();
  this_thread_interrupt_flag.set_condition_variable(cv);
  interrupt_flag::clear_cv_on_destruct guard;
```

```
  interruption_point();
  cv.wait_for(lk,std::chrono::milliseconds(1));
  interruption_point();
}
```

如果有等待的谓词，那么 1ms 的超时可以完全隐藏在谓词循环中：

```
template<typename Predicate>
void interruptible_wait(std::condition_variable& cv,
                        std::unique_lock<std::mutex>& lk,
                        Predicate pred)
{
  interruption_point();
  this_thread_interrupt_flag.set_condition_variable(cv);
  interrupt_flag::clear_cv_on_destruct guard;
  while(!this_thread_interrupt_flag.is_set() && !pred())
  {
    cv.wait_for(lk,std::chrono::milliseconds(1));
  }
  interruption_point();
}
```

这将导致比其他情况更频繁的检查谓词，不过可以很容易用它替换普通的 wait()调用。带有超时的变体很容易实现：要么等待指定时间，或者 1ms，以最短时间为准。好了，对于 std::condition_variable 的等待可以处理了；那 std::condition_variable_any 又如何？是跟现在一样处理？还是说能做的更好？

## 9.2.4 中断在 std::condition_variable_any 上的等待

std::condition_variable_any 与 std::condition_variable 的不同在于，std::condition_variable_any 可以使用任意类型的锁，而不只是 std::unique_lock<std::mutex>。这让事情更简单，也使得使用 std::condition_variable_any 可以比 std::condition_variable 做的更好。因为它能与任意类型的锁一起工作，于是你可以构建自己的锁类型，上锁/解锁操作既可以用于 interrupt_flag 的内部互斥锁 set_clear_mutex，也可以用于提供给等待调用的锁，如下所示。

清单 9.12 用于 std::condition_variable_any 的 interruptible_wait

```
class interrupt_flag
{
  std::atomic<bool> flag;
  std::condition_variable* thread_cond;
  std::condition_variable_any* thread_cond_any;
```

```
  std::mutex set_clear_mutex;

public:
  interrupt_flag():
    thread_cond(0),thread_cond_any(0)
  {}

  void set()
  {
    flag.store(true,std::memory_order_relaxed);
    std::lock_guard<std::mutex> lk(set_clear_mutex);
    if(thread_cond)
    {
      thread_cond->notify_all();
    }
    else if(thread_cond_any)
    {
      thread_cond_any->notify_all();
    }
  }

  template<typename Lockable>
  void wait(std::condition_variable_any& cv,Lockable& lk)
  {
    struct custom_lock
    {
      interrupt_flag* self;
      Lockable& lk;

      custom_lock(interrupt_flag* self_,
                  std::condition_variable_any& cond,
                  Lockable& lk_):
        self(self_),lk(lk_)
      {
        self->set_clear_mutex.lock();  // 1
        self->thread_cond_any=&cond;  // 2
      }

      void unlock()  // 3
      {
        lk.unlock();
        self->set_clear_mutex.unlock();
      }
```

```
      void lock()
      {
        std::lock(self->set_clear_mutex,lk);  // 4
      }

      ~custom_lock()
      {
        self->thread_cond_any=0;  // 5
        self->set_clear_mutex.unlock();
      }
    };
    custom_lock cl(this,cv,lk);
    interruption_point();
    cv.wait(cl);
    interruption_point();
  }
  // 剩下的和之前一样
};

template<typename Lockable>
void interruptible_wait(std::condition_variable_any& cv,
                        Lockable& lk)
{
  this_thread_interrupt_flag.wait(cv,lk);
}
```

自定义的锁类型在构造的时候，需要锁住内部 set_clear_mutex①，然后设置 thread_cond_any 指针指向传入构造函数中的 std::condition_variable_any 引用②。保存的 Lockable 引用会在后面用到，它必须已经被锁住。现在可以安心的检查中断，不用担心竞争了。如果中断标志在这时候已经设置，那么标志是在锁住 set_clear_mutex 之前设置的。在 wait()中，当条件变量调用你的 unlock()函数时，解锁 Lockable 对象以及 set_clear_mutex③。一旦进入 wait()调用（而不是在之前），这就允许尝试中断你的线程获得 set_clear_mutex 的锁并检查 thread_cond_any 指针（译注：这里的意思是现在用锁消除了前面提到的第二个关于竞争条件的问题，只有在你的线程进入到 cv.wait 里面了，才会解锁 set_clear_mutex，然后中断你的线程才有机会给它上锁并给你发信号通知）。这正是使用 std::condition_variable 所追求的(但那时没法做到)。一旦 wait()结束等待(要么被通知，要么被伪唤醒)，它将调用你的 lock()函数，并再次获取内部 set_clear_mutex 以及 Lockable 对象上的锁④。在你的 custom_lock 析构函数⑤清理 thread_cond_any 指针，以及解锁 set_clear_mutex 之前，现在可以再次对发生在 wait()调用期间的中断进行检查。

## 9.2.5 中断其他阻塞调用

虽然搞定了中断条件变量的等待，但是其他类型的阻塞等待又如何呢：互斥锁，等待期望，以及诸如此类？通常情况下，你需要使用用于 std::condition_variable 的超时选项，因为不访问互斥锁或者期望的内部构件的话，除非满足了等待的条件，否则无法中断等待。但是对于其他东西，你知道自己在等待什么，因此可以在 interruptible_wait()函数中进行循环。作为一个例子，这里为 std::future<>重载了 interruptible_wait()的实现：

```
template<typename T>
void interruptible_wait(std::future<T>& uf)
{
  while(!this_thread_interrupt_flag.is_set())
  {
    if(uf.wait_for(lk,std::chrono::milliseconds(1))==
       std::future_status::ready)
      break;
  }
  interruption_point();
}
```

等待会在中断标志设置好或者期望是就绪状态时停止，每次在期望上阻塞等待 1ms。这就意味着，假设采用高精度时钟，在确认中断请求前，平均需要大约 0.5ms。通常 wait_for 至少会等待一整个时钟计次，所以如果你的时钟每 15ms 计次一次，那么最终等待大约 15ms，而不是 1ms。是否可接受，要看情况而定。如果有必要(并且时钟支持)，你总是可以减少超时。但减少超时时间的缺点是：将会让线程频繁醒来检查标志，并且增加任务切换的开销。

好了，我们已经了解了如何使用 interruption_point()和 interruptible_wait()函数检测中断。但怎么处理中断呢？

## 9.2.6 处理中断

从被中断线程的角度看，一个中断就是一个 thread_interrupted 异常，因此可以像处理其他异常那样进行处理。特别地，你可以使用标准 catch 块对其进行捕获：

```
try
{
  do_something();
}
catch(thread_interrupted&)
{
  handle_interruption();
```

```
}
```

这意味着你可以捕获中断，用某种方法进行处理，然后继续。如果你这么做，并且另一个线程再次调用 interrupt()时，线程将会在下一次调用中断点时再次被中断。如果线程执行的是一系列独立的任务的话，你可能希望这么做；中断一个任务会导致这个任务被丢弃，然后该线程会继续执行列表中的下一个任务。

因为 thread_interrupted 是一个异常，当调用可中断代码时，所有通常线程安全的防范措施都必须采用，就为了确保资源不会泄露，并且你的数据结构保持一致状态。通常，让中断终止线程是可取的，这样你就可以让异常向上传播。不过，如果你让异常传播到传递给 std::thread 构造函数的线程函数的外面，那么 std::terminate()会被调用，这样整个程序将会终止。为了避免这种情况，必须记住安置一个捕获（thread_interrupted）的处理器在每个你传给 interruptible_thread 的函数中，你可以把 catch 块放入你用来初始化 interrupt_flag 的包装器中。这样就允许中断异常不经过处理传播出去是安全的，因为它随后会终止那个线程。interruptible_thread 构造函数中对线程的初始化现在看起来如下：

```
internal_thread=std::thread([f,&p]{
        p.set_value(&this_thread_interrupt_flag);

        try
        {
          f();
        }
        catch(thread_interrupted const&)
        {}
      });
```

让我们来看个具体的例子，这个例子中中断是很有用的。

## 9.2.7 应用退出时中断后台任务

考虑一个桌面搜索应用。不光要与用户互动，应用还需要监控文件系统的状态，识别任何变化并更新索引。为了避免影响 GUI 的响应能力，通常会将处理线程放在后台运行。在整个应用的生命周期，后台线程需要一直执行；后台线程会作为应用启动的一部分被启动，并且在应用关闭的时候停止运行。通常这样的应用只有在机器关闭时才退出，因为应用需要一直运行从而维护一个最新的索引。在任何情况下，当应用被关闭，需要使用有序的方式将后台线程关闭，其中一种方式就是中断它们。

接下来的清单展示了这个系统线程管理部分的一个示例实现。

```
清单 9.13 在后台监视文件系统

std::mutex config_mutex;
```

```
std::vector<interruptible_thread> background_threads;

void background_thread(int disk_id)
{
  while(true)
  {
    interruption_point();  // 1
    fs_change fsc=get_fs_changes(disk_id);  // 2
    if(fsc.has_changes())
    {
      update_index(fsc);  // 3
    }
  }
}

void start_background_processing()
{
  background_threads.push_back(
    interruptible_thread(background_thread,disk_1));
  background_threads.push_back(
    interruptible_thread(background_thread,disk_2));
}

int main()
{
  start_background_processing();  // 4
  process_gui_until_exit();  // 5
  std::unique_lock<std::mutex> lk(config_mutex);
  for(unsigned i=0;i<background_threads.size();++i)
  {
    background_threads[i].interrupt();  // 6
  }
  for(unsigned i=0;i<background_threads.size();++i)
  {
    background_threads[i].join(); // 7
  }
}
```

启动时，后台线程就被启动④。然后主线程继续处理 GUI⑤。当用户请求应用程序退出时，后台进程将会被中断⑥，主线程在退出前会等待每一个后台线程结束。后台线程运行在一个循环中，检查磁盘的变化②，并且更新索引③。每次在循环过程中通过调用 interruption_point()函数①对中断进行检查。

为什么在等待任何线程前中断所有线程？为什么不先中断一个，然后等待它，之后再处理下一个？答案就是“并发”。线程被中断后，不大可能马上结束，因为它们必须继续

运行到下一个中断点，然后在退出之前运行任何必需的析构函数调用和异常处理代码。通过立即连接每个线程，将导致中断线程等待，即使它仍有有用的工作可做——中断其他线程。只有当你没有更多工作可做（所有线程已经被中断）时才等待。这也允许所有被中断的线程并行处理他们的中断，因而潜在的可以更快的完成。

可以很容易地对这种中断机制进行扩展，以添加更多的可中断调用，或禁用跨特定代码块的中断，不过这个就留给读者作为练习吧。

1. P0660: A Cooperatively Interruptible Joining Thread, Rev 3, Nicolai Josuttis, Herb Sutter, Anthony Williams http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0660r3.pdf.

# 总结

本章中，我们了解了各种“高级”线程管理技术：线程池和中断线程。也看到了使用局部工作队列和工作窃取如何减小同步开销，并潜在的提高线程池的吞吐量，还看到了当等待子任务完成时，如何运行队列中的其他任务能消除死锁。

我们也了解了各种方法，允许一个线程去中断另一个线程的处理，比如使用特定的中断点和函数以可中断的方式执行阻塞等待。

# 第 10 章 并行算法

**本章主要内容**

- 使用 C++17 并行算法

上一章我们介绍了高级线程管理和线程池，在第 8 章中，我们使用一些算法的并行版本作为例子来研究设计并发代码。在本章，我们将看看 C++17 标准提供的并行算法，所以闲话少说，让我们开始吧。

## 10.1 并行化标准库算法

C++17 标准添加了*并行算法*（parallel algorithms）的概念到 C++标准库中。它们是在范围上操作的许多函数的额外重载，如 std::find、std::transform 和 std::reduce。并行版本与“普通”单线程版本具有相同的签名，除了添加了一个新的第一个参数，该参数指定要使用的*执行策略*（execution policy）。例如：

```
std::vector<int> my_data;
std::sort(std::execution::par,my_data.begin(),my_data.end());
```

std::execution::par 的执行策略向标准库表明，它允许以并行算法的形式，使用多个线程执行这个调用。注意！这是一个许可，不是一个要求——如果愿意的话，库仍然可以在单线程上执行代码。还需要重点注意的是，通过指定执行策略，对算法复杂度的要求也发生了变化，并且通常比普通串行版的算法要宽松。这是因为并行算法为了利用系统的并行，通常会做更多的工作——如果将工作划分到 100 个处理器，即便完成的总工作量是原先的的 2 倍，仍然能获得 50 倍的加速。

在讨论算法本身之前，让我们先看看执行策略。

## 10.2 执行策略

标准指定了三种执行策略：

- std::execution::sequenced_policy
- std::execution::parallel_policy
- std::execution::parallel_unsequenced_policy

这些类都定义在<execution>头文件中。这个头文件中也定义了三个对应的策略对象可以传给算法：

- std::execution::seq
- std::execution::par
- std::execution::par_unseq

除了复制这三种类型的对象外，你不能依赖于自己能够从这些策略类构造对象，因为它们可能有一些特殊的初始化要求。实现可能还会定义一些其他的执行策略，这些策略有实现相关的行为。总之你不能定义自己的执行策略。

这些策略对算法行为的影响在 10.2.1 节描述。任何给定的实现也允许提供额外的执行策略，带有它们想要的任何语义。现在，让我们看看使用一种标准执行策略的影响，首先从所有算法重载（这些重载带有一个执行策略）的一般改变开始。

## 10.2.1 指定执行策略的一般影响

如果将执行策略传递给标准算法库中的算法，算法的行为就由执行策略控制。这会影响行为的几个方面：

- 算法的复杂度
- 抛出异常时的行为
- 执行算法步骤的位置、方式和时间

**算法复杂度影响**

如果给算法提供了执行策略，那么算法的复杂度可能会发生变化：除了管理并行执行的调度开销外，很多并行算法会执行更多的算法核心操作(无论是交换，比较，还是所提供函数对象的应用)，目的是在总运行时间方面提供全面的性能改进。

复杂度变化的精确细节会随着每个算法而变化，不过一般的策略是，如果一个算法指定了一些事情恰好发生 some-expression（某个表达式）次，或者至多 some-expression 次，那么执行策略的重载将把这个要求放松到 O(some-expression)。这意味着，相比于不带执行策略的版本，带有执行策略的重载可能会执行数倍于它的操作，而这个倍数将取决于库和平台的内部实现，而不是提供给算法的数据。

**异常行为**

如果在执行带有执行策略的算法期间抛出异常，那么结果由执行策略确定。如果有未被捕获的异常，那么所有标准提供的执行策略都会调用 std::terminate。使用标准执行策略调用标准库算法可能抛出的唯一异常是 std::bad_alloc，如果标准库不能为其内部操作获得足够的内存资源，则会抛出该异常。例如，没有执行策略的情况下，以下对 std::for_each 的调用会将异常进行传播。

```
std::for_each(v.begin(),v.end(),[](auto x){ throw my_exception(); });
```

然而具有执行策略的对应调用，将会终止程序：

```
std::for_each(
  std::execution::seq,v.begin(),v.end(),
  [](auto x){ throw my_exception(); });
```

这就是使用 std::execution::seq 执行策略和不提供执行策略间的关键区别。

**算法步骤执行的位置和时间**

这是执行策略的基本方面，也是标准执行策略之间唯一不同的方面。策略指定了使用哪个执行代理来执行算法步骤，它们是“普通”线程、向量流、GPU 线程，或其他任何东西。执行策略还将指定是否有任何顺序约束用于算法步骤如何运行：他们是否运行在任何特定的顺序，部分单独的算法步骤是否可以相互交错，或并行运行，等等。

每个标准执行策略的详细信息见 10.2.2，10.2.3 和 10.2.4 节，让我们先从最基本的策略 std::execution::sequenced_policy 开始。

## 10.2.2 std::execution::sequenced_policy

顺序策略不是并行策略：使用它强制实现在调用函数的线程上执行所有操作，因此没有并行。但它仍然是一个执行策略，因此对算法的复杂度和异常影响与其他标准执行策略相同。

不仅所有的操作必须在同一个线程上执行，而且它们必须按照确定的顺序执行，所以它们不是交错的。但具体的顺序是未指定的，并且可能在不同的函数调用之间有所不同。特别是，不能保证操作的执行顺序与相应没有执行策略的重载的执行顺序相同。例如：下面对 std::for_each 的调用，以未指定的顺序将 1-1000 填充到 vector 中。这就与没有执行策略的重载形成对比，它将顺序存储数字：

```
std::vector<int> v(1000);
int count=0;
std::for_each(std::execution::seq,v.begin(),v.end(),
  [&](int& x){ x=++count; });
```

数字可能按顺序存储，但你不能依赖它。

这意味着顺序策略对算法使用的迭代器、值和可调用对象没有什么要求：它们可以自由使用同步机制，并且可以依赖在同一线程上调用的所有操作，尽管它们不能依赖这些操作的顺序。

## 10.2.3 std::execution::parallel_policy

并行策略提供了跨多个线程的基本并行执行。操作可以在调用算法的线程上执行，也可以在库创建的线程上执行。在给定线程上执行的操作必须以确定的顺序执行，不能交错执行，但具体的顺序是未指定的，并且在不同的调用之间可能会有所不同。一个给定的操作将在其整个持续时间内运行在一个固定的线程上。

这对顺序策略上算法使用的迭代器、值和可调用对象施加了额外的要求：如果并行调用，它们不能导致数据竞争，并且不能依赖于与其他操作在同一个线程上运行，或者依赖于与其他操作不在同一个线程上运行。

在使用没有执行策略的标准库算法的大多数情况下，你都可以使用并行执行策略。只有在需要的元素之间有特定的顺序，或者对共享数据的非同步访问时，才会有问题。对 vector 中的所有值进行递增操作可以并行完成：

```
std::for_each(std::execution::par,v.begin(),v.end(),[](auto& x){++x;});
```

如果使用并行执行策略，前面填充 vector 的例子就有问题；具体来说，它是未定义的行为：

```
std::for_each(std::execution::par,v.begin(),v.end(),
  [&](int& x){ x=++count; });
```

这里每次调用 lambda 表达式都会对 count 进行修改，因此，如果库跨越多个线程执行 lambda 表达式，这里就会出现数据竞争，从而导致未定义行为。对 std::execution::parallel_policy 的要求抢先得到这个结果：即使库没有为这个调用使用多个线程，进行上述调用也是未定义的行为。是否展示未定义行为是调用的静态属性，而不是依赖于库的实现细节。然而，函数调用之间的同步是允许的，因此可以通过将 count 设置为 std::atomic<int>而不是普通的 int，或者使用互斥锁，再次让它成为已定义的行为。在这种情况下，这可能会破坏使用并行执行策略的意义，因为这将串行化所有调用，但在一般情况下，它将允许对共享状态的同步访问。

## 10.2.4 std::execution::parallel_unsequenced_policy

并行非顺序策略为库提供了最大程度的算法并行，作为交换它对算法使用的迭代器、值和可调用对象也施加了最严格的要求。

使用并行非顺序策略调用的算法，可以在任意线程上执行算法步骤，这些线程彼此间是无序的。这意味着操作现在可以在一个线程上相互交错，这样，在第一个操作完成之前，第二个操作就在同一个线程上启动了，并且可能在线程间迁移，因此一个给定的操作可能会在一个线程启动，在第二个线程继续运行，并在第三个线程上完成。

如果使用并行非顺序策略，在提供给该算法的迭代器、值和可调用对象上调用的操作就一定不能使用任何形式的同步，也不能调用任何与其他函数同步的函数或调用任何与其他代码同步的函数。

这意味着操作只能操作相关的元素，或者基于该元素可以访问的任何数据，并且不能修改线程之间或元素之间共享的任何状态。

稍后我们将用一些例子来充实这些内容。现在，让我们来看看并行算法本身。

# 10.3 C++标准库中的并行算法

<algorithm>和<numeric>头文件中大多数算法都有带一个执行策略的重载。这包括：all_of，any_of，none_of，for_each，for_each_n，find，find_if，find_end，find_first_of，adjacent_find，count，count_if，mismatch，equal，search，search_n，copy，copy_n，copy_if，move，swap_ranges，transform，replace，replace_if，replace_copy，replace_copy_if，fill，fill_n，generate，generate_n，remove，remove_if，remove_copy，remove_copy_if，unique，unique_copy，reverse，reverse_copy，rotate，rotate_copy，is_partitioned，partition，stable_partition，partition_copy，sort，stable_sort，partial_sort，partial_sort_copy，is_sorted，is_sorted_until，nth_element，merge，inplace_merge，includes，set_union，set_intersection，set_difference，set_symmetric_difference，is_heap，is_heap_until，min_element，max_element，minmax_element，lexicographical_compare，reduce，transform_reduce，exclusive_scan，inclusive_scan，transform_exclusive_scan，transform_inclusive_scan 和 adjacent_difference 。

相当长的一个清单；C++标准库中几乎所有可以并行化的算法都在这个列表中。明显的例外是 std::accumulate，它是严格意义上的串行积累（译注：accumulate 有两个版本，一个用‘+’，另一个用二元函数 op，所以翻译成积累），但其广义相对物 std::reduce 确实出现在列表中——标准中有一个适当的警告：如果归约操作既不是可结合的也不是可交换的，那么由于操作未指定顺序，结果可能是不确定的。

对于列表中的每个算法，每个“普通”重载都有一个新的变体，它将执行策略作为第一个参数——“普通”重载的相应参数随后出现在执行策略之后。例如，std::sort 有两个没有执行策略的“普通”重载：

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort(
  RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

因此，它还有两个带执行策略的重载：

```
template<class ExecutionPolicy, class RandomAccessIterator>
void sort(
  ExecutionPolicy&& exec,
  RandomAccessIterator first, RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compa
re>
void sort(
```

```
  ExecutionPolicy&& exec,
  RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

带和不带执行策略参数的签名有一个重要的区别，它只会影响一些算法：如果“普通”算法允许输入迭代器或输出迭代器，那带执行策略的重载则需要前向迭代器。这是因为输入迭代器从根本上是单趟的：只能访问当前元素，而不能存储指向前面元素的迭代器。类似地，输出迭代器只允许写入当前元素：不能通过向前移动迭代器来写入后面的元素，然后向后移动迭代器来写入前面的元素。

**C++标准库中的迭代器类别**

C++标准库定义了五种类型的迭代器：输入迭代器、输出迭代器、前向迭代器、双向迭代器和随机访问迭代器。

输入迭代器是用于检索值的单趟迭代器。通常用于控制台或网络的输入，或生成序列。前移输入迭代器会使该迭代器的任何副本失效。

输出迭代器是用于写入值的单趟迭代器。通常用于输出到文件或向容器添加值。前移输出迭代器会使该迭代器的任何副本失效。

前向迭代器是用于对持久数据进行单向迭代的多趟迭代器。虽然不能让迭代器返回到前面的元素，但可以存储副本并使用它们引用前面的元素。前向迭代器返回对元素的实际引用，因此既可用于读也可用于写(如果目标不是常量的话)。

双向迭代器是像前向迭代器一样的多趟迭代器，但是它也可以后退访问之前的元素。

随机访问迭代器是可以像双向迭代器一样前进和后退的多趟迭代器，但它前进和后退的步伐可以大于一个元素，并且可以使用数组索引运算符，直接访问偏移位置的元素。

因此，给定 std::copy 的“普通”签名：

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(
  InputIterator first, InputIterator last, OutputIterator result);
```

带有执行策略的重载是

```
template<class ExecutionPolicy,
  class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 copy(
  ExecutionPolicy&& policy,
  ForwardIterator1 first, ForwardIterator1 last,
  ForwardIterator2 result);
```

虽然，模板参数的命名从编译器的角度没有带来任何直接影响，但从 C++标准的角度确实有影响：标准库算法的模板参数的名称表示类型的语义约束，并且算法将依赖于那些

约束（具有指定的语义）所隐含的操作。对比输入迭代器和前向迭代器，前者允许解引用迭代器以返回代理类型，该代理类型可转换为迭代器的值类型，而后者要求对迭代器的解引用返回对该值的实际引用，所有相等的迭代器返回对相同值的引用。

这对于并行很重要：这意味着迭代器可以自由地复制，并等价地使用。此外，递增前向迭代器不会使其他副本失效也很重要，因为这意味着不同的线程可以操作各自的迭代器副本，需要的时候递增迭代器，而不必担心使其他线程持有的迭代器失效。如果带有执行策略的重载允许使用输入迭代器，这将迫使任何线程串行访问用于从源序列读取的唯一迭代器，这显然限制了并行的潜力。

让我们看一些具体的例子。

## 10.3.1 使用并行算法的例子

最简单的例子肯定是并行循环：为容器的每个元素做一些事情。这是令人尴尬的并行（embarrassingly parallel，译注：参见第一章，因为太容易并行，让人觉得很意外，很尴尬的意思）场景的经典例子：每个项都是独立的，因此你有最大的并行可能性。如果使用支持 OpenMP 的编译器，你可以这样写：

```
#pragma omp parallel for
for(unsigned i=0;i<v.size();++i){
  do_stuff(v[i]);
}
```

使用 C++标准库算法，可以这样写：

```
std::for_each(std::execution::par,v.begin(),v.end(),do_stuff);
```

这将在库创建的内部线程之间划分范围内的元素，并在范围内的每个元素 x 上调用 do_stuff(x)。但如何在线程间划分元素是一个实现细节。

**执行策略的选择**

除非你的实现提供了更适合你的非标准策略，否则 std::execution::par 是最常用的策略。如果代码适合并行化，那应该与 std::execution::par 一起工作。某些情况下，可以使用 std::execution::par_unseq 代替。这可能一点用也没有(没有一个标准的执行策略会对并行达到的水平做出保证)，但是，通过重新排序和交叉任务，它可能会给库提供额外的范围来提高代码的性能，当然也会换来对代码更严格的要求。这些更严格的要求中最值得注意的是，在访问元素或对元素执行操作时不能使用同步。这意味着不能使用互斥锁或原子变量，或前面章节中描述的任何其他机制，以确保多线程的访问是安全的；相反，必须依赖算法本身不会从多个线程访问相同的元素，并在对并行算法的调用之外使用外部同步来防止其他线程访问数据。

清单 10.1 中的示例显示了一些可以与 std::execution::par 一起使用，但不能与 std::execution::par_unseq 一起使用的代码。使用内部互斥锁进行同步意味着尝试使用 std::execution::par_unseq 将是未定义的行为。

清单 10.1 类（有内部同步）上的并行算法

```
class X{
  mutable std::mutex m;
  int data;
public:
  X():data(0){}
  int get_value() const{
    std::lock_guard guard(m);
    return data;
  }
  void increment(){
    std::lock_guard guard(m);
    ++data;
  }
};
void increment_all(std::vector<X>& v){
  std::for_each(std::execution::par,v.begin(),v.end(),
    [](X& x){
    x.increment();
    });
}
```

下一个清单显示了一种可用于 std::execution::par_unseq 的替代方法。在本例中，内部的每个元素的互斥锁被替换为整个容器的互斥锁。

清单 10.2 类（没有内部同步）上的并行算法

```
class Y{
  int data;
public:
  Y():data(0){}
  int get_value() const{
    return data;
  }
  void increment(){
    ++data;
  }
};
class ProtectedY{
  std::mutex m;
  std::vector<Y> v;
```

```
public:
  void lock(){
    m.lock();
  }

  void unlock(){
    m.unlock();
  }

  std::vector<Y>& get_vec(){
    return v;
  }
};

void increment_all(ProtectedY& data){
  std::lock_guard guard(data);
  auto& v=data.get_vec();
  std::for_each(std::execution::par_unseq,v.begin(),v.end(),
    [](Y& y){
      y.increment();
    });
}
```

清单 10.2 中的元素访问现在没有同步，使用 std::execution::par_unseq 是安全的。但缺点是来自并行算法调用之外的其他线程的并发访问现在必须等待整个操作完成，而不是清单 10.1 中的每个元素粒度。

现在让我们来看一个更现实的例子，看看并行算法是如何使用的：统计网站的访问量。

## 10.3.2 统计访问量

假设你经营一个繁忙的网站，日志有数百万条目，并且你希望对这些日志进行处理以便查看聚合数据：每个页面有多少访问量，这些访问来自哪里，使用哪些浏览器访问的网站，等等。分析日志包含两个部分：处理每一行以提取相关信息，以及将结果聚合在一起。这是使用并行算法的理想场景，因为每一行都可以完全独立的处理，并且只要最终的总数是正确的，结果就可以逐步聚合。

特别地，transform_reduce 就是专门为这类任务设计的。下面的清单显示了如何将它用于此任务。

清单 10.3 使用 transform_reduce 来统计网站页面的访问量

```
#include <vector>
```

```cpp
#include <string>
#include <unordered_map>
#include <numeric>

struct log_info {
  std::string page;
  time_t visit_time;
  std::string browser;
  // any other fields
};

extern log_info parse_log_line(std::string const &line); // 1
using visit_map_type= std::unordered_map<std::string, unsigned long lon
g>;
visit_map_type
count_visits_per_page(std::vector<std::string> const &log_lines) {

  struct combine_visits {
    visit_map_type
    operator()(visit_map_type lhs, visit_map_type rhs) const { // 3
      if(lhs.size() < rhs.size())
        std::swap(lhs, rhs);
      for(auto const &entry : rhs) {
        lhs[entry.first]+= entry.second;
      }
      return lhs;
    }

    visit_map_type operator()(log_info log,visit_map_type map) const{ /
/ 4
      ++map[log.page];
      return map;
    }

    visit_map_type operator()(visit_map_type map,log_info log) const{ /
/ 5
      ++map[log.page];
      return map;
    }

    visit_map_type operator()(log_info log1,log_info log2) const{ // 6
      visit_map_type map;
      ++map[log1.page];
      ++map[log2.page];
```

```
      return map;
    }
  };

  return std::transform_reduce( // 2
    std::execution::par, log_lines.begin(), log_lines.end(),
    visit_map_type(), combine_visits(), parse_log_line);
}
```

假设已经有一个函数 parse_log_line 从日志条目中提取相关信息①，count_visits_per_page 函数是对 std::transform_reduce 调用的简单包装②。复杂度来源于归约（reduction）操作：需要能够组合两个 log_info 结构体来生成一个 map，一个 log_info 结构体和一个 map(无论哪种方式)，或两个 map。因此这意味着 combine_visits 函数对象需要重载 4 个函数调用操作符，③④⑤和⑥，这就排除了用一个简单的 lambda 来实现它，哪怕这四个重载的实现很简单。

因此，std::transform_reduce 的实现将使用可用的硬件并行来执行这个计算(因为你传递了 std::execution::par)。正如我们在前一章中看到的那样，手工编写这个算法并不简单，因此这允许你将实现并行的苦差事委托给标准库的实现者，这样你就可以专注于所需的结果。

# 总结

在本章，我们讨论了 C++标准库中可用的并行算法，以及如何使用它们。讨论了各种执行策略、执行策略的选择对算法行为的影响，以及它对代码施加的限制。然后，我们看了一个示例，演示了如何在实际代码中使用该算法。

# 第 11 章 测试和调试多线程应用程序

**本章主要内容**

- 并发相关的 bug
- 通过测试和代码评审定位 bug
- 设计多线程测试
- 测试多线程代码的性能

到目前为止，我主要关注编写并发代码所涉及的内容——可用的工具、如何使用它们，以及代码的总体设计和结构。但是有一个软件开发的关键部分还没有解决：测试和调试。如果你正在阅读这一章，希望找到一种简单的方法来测试并发代码，那你将会非常失望。因为测试和调试并发代码是非常困难的。而我将呈献给你的也只是让事情变得更简单的一些技术，以及一些值得思考的重要问题。

测试和调试就像硬币的两面——你让你的代码进行测试，以便找到可能存在的 bug，然后你对它进行调试，以消除这些 bug。如果幸运的话，你只需要移除你自己测试发现的 bug，而不是应用程序的最终用户发现的 bug。在我们讨论测试或调试之前，理解可能出现的问题是很重要的，所以让我们来看看这些问题。

## 11.1 并发相关的 bug 类型

你可以在并发代码中碰到任何类型的错误；在这方面它并不特别。但有些类型的 bug 与并发的使用直接相关，因此与本书特别相关。通常，这些与并发相关的 bug 可以分为两类：

- 不必要的阻塞
- 竞争条件

这些都是很宽泛的类别，所以我们把它们稍微划分一下。首先，让我们看看不必要的阻塞。

### 11.1.1 不必要的阻塞

我说的不必要的阻塞是什么意思？当线程因为等待某些东西而无法继续时，它就会被阻塞。等待的东西通常是一个互斥锁、一个条件变量或一个期望，但它也可能是在等待 I/O。这是多线程代码的自然组成部分，但并不总是可取的——因此出现了不必要的阻塞问题。这就引出了下一个问题：为什么这种阻塞是不必要的？通常，这是因为其他一些线程也在等待被阻塞的线程执行某个动作，因此该线程也被阻塞。这一主题有几种变体：

- 死锁——正如你在第 3 章看到的，在死锁的情况下，一个线程会等待另一个，而另一个又会等待第一个。如果你的线程死锁，它们应该做的任务就无法完成。在最明

显的情况下，所涉及的线程之一是负责用户界面的线程，在这种情况下，界面将停止响应。在其他情况下，界面将保持响应，但一些所需的任务不会完成，例如搜索不返回或文档不打印。

- 活锁——活锁与死锁类似，一个线程在等待另一个线程，而另一个线程又在等待第一个线程。这里的关键区别在于，等待不是阻塞等待，而是一个活跃的检查循环，比如自旋锁。在严重的情况下，症状与死锁相同(应用程序不会有任何进展)，只是 CPU 使用率很高，因为线程仍然在运行，但彼此阻塞。在不太严重的情况下，由于随机调度，活锁最终会解决，但是在被卷入活锁的任务中会有很长的延迟，在此延迟期间会有很高的 CPU 使用量。
- 阻塞在 I/O 或外部输入上——如果线程在等待外部输入时被阻塞，它就不能继续执行，即使等待的输入永远不会出现。 因此，让一个线程阻塞在外部输入上，而其他线程又可能等待它执行任务是不可取的。

这里简要介绍了不必要的阻塞。那么竞争条件呢？

## 11.1.2 竞争条件

竞争条件是多线程代码中最常见的问题原因——许多死锁与活锁只在竞争条件下出现。并非所有竞争条件都有问题——当行为依赖于不同线程中操作的相对调度时，竞争条件就会发生。很多竞争条件完全是良性的；例如，哪个工作线程处理任务队列中的下一个任务在很大程度上是不相关的。但是许多并发 bug 是由于竞争条件造成的。特别地，竞争条件经常会导致以下类型的问题：

- 数据竞争——数据竞争是一种特定类型的竞争条件，由于对共享内存位置的非同步并发访问，它会导致未定义的行为。我在第 5 章中讨论 C++内存模型的时候介绍了数据竞争。数据竞争通常是由于不正确地使用原子操作来同步线程或访问共享数据时没有锁住适当的互斥锁而发生的。
- 破坏的不变量——它们可以表现为悬垂指针(因为另一个线程删除了被访问的数据)、随机内存崩溃(由于部分更新导致线程读取了不一致的值)和双重释放内存(例如，当两个线程从队列中弹出相同的值时，它们都会删除某个相关的数据)。被破坏的不变量可以是基于时间的——也可以是基于值的。如果单独线程上的操作需要按照特定的顺序执行，不正确的同步可能会导致竞争条件，在这里有时会违反所需的顺序。
- 生命周期问题——尽管可以将这些问题与破坏的不变量捆绑在一起，但这是一个单独的类别。这类 bug 的基本问题是线程比它所访问的数据寿命长，因此它正在访问已删除或以其他方式销毁的数据，而且存储空间可能会被另一个对象重用。在线程函数完成之前，线程引用了超出作用域的局部变量，这通常会导致生命周期问题，但它们并不仅限于该场景。每当线程的生命周期和它所操作的数据没有以某种方式绑定在一起时，数据就有可能在线程完成之前被销毁，然后线程函数就万劫不复了。如果为了等待线程完成而手动调用 join()，则需要确保在抛出异常时不会跳过对 join()的调用。这是应用于线程的基本异常安全。

有问题的竞争条件是罪魁祸首。使用死锁和活锁时，应用程序看起来会挂起，变得完全没有响应，或者需要很长时间才能完成任务。通常，可以将调试器附到正在运行的进程

上，以确定哪些线程涉及死锁或活锁，以及它们正在争夺哪些同步对象。对于数据竞争，破坏的不变量，以及生命周期问题，问题的明显症状(比如随机崩溃或不正确的输出)可以出现在代码的任何地方——代码可能会覆盖系统另一部分使用的内存，而这些内存直到很久以后才被使用。然后，错误将在与错误代码的位置完全无关的代码中出现，很可能是在程序执行很靠后的时候。这是共享内存系统的真正祸根——无论如何限制哪个线程可以访问哪些数据，以及尝试确保使用了正确的同步，任何线程都可能覆盖应用程序中任何其他线程正在使用的数据。

现在，我们已经简单地确定了要查找的问题种类，让我们看看如何定位代码中的任何实例，以便修复它们。

# 11.2 定位并发相关 bug 的技术

在前一节中，我们了解了可能看到的与并发相关的 bug 类型，以及它们在代码中是如何出现的。有了这些基础以后，你就可以查看代码，了解 bug 可能存在的位置，并且尝试确定特定部分中是否存在 bug。

也许最明显和最直接的方法是查看代码。虽然这看起来显而易见，但却很难彻底做到这一点。因为当你阅读自己编写的代码时，很容易陷入自己的思维逻辑中，而不是客观的审查代码本身。同样地，在审查别人写的代码时，很容易被蜻蜓点水般通读一遍所诱惑，然后仅仅对照编码规范进行检查，并突出任何明显的问题。但我们真正需要做的是花时间仔细梳理代码，思考并发问题——以及非并发问题。(当你审查代码的时候，索性也会审查非并发问题。毕竟，bug 就是 bug，不管是否是并发问题)稍后我们将介绍在评审代码时需要考虑的具体事项。

即使在彻底评审代码之后，仍然可能遗漏一些 bug，并且在任何情况下，你都需要确认它确实可以工作，别的不说，主要为了让你安心。因此，我们将继续从评审代码到一些在测试多线程代码时使用的技术。

## 11.2.1 评审代码以定位潜在 bug

正如我已经提到的，当评审多线程代码以检查与并发相关的 bug 时，彻底评审它很重要。如果可能的话，可以找别人来评审。因为不是他们写的代码，他们将不得不仔细思考它是如何工作的，这将有助于发现任何可能存在的 bug。重要的是，评审者要有足够的时间进行恰当的评审——不是随意的瞄几眼，而是恰当的、经过深思熟虑的评审。大多数并发 bug 不只是快速一瞥就能发现——它们的出现，通常依赖于微妙的时序问题。

如果你让你的同事来评审代码，他们会重新开始。因此，他们会从不同的角度看待事物，可能会发现你看不到的东西。如果你没有同事可以邀请，可以邀请朋友，甚至把代码贴在网上(小心不要惹恼你公司的律师)。如果实在没人帮你评审代码，或者他们没找到问题，不用担心——还有很多事情可以做。对初学者来说，暂时不去管代码可能是值得的——处理应用程序的另一部分、读书或散步。如果你休息一下，你的潜意识可以在你有意识

地专注于其他事情的时候在后台处理这个问题。此外，当你再次回来的时候，对代码可能会相对生疏——你自己可能会设法从不同的角度来看待它。

让别人检查代码的另一种方法是自己检查。一个有用的技巧是试着向别人详细解释它是如何工作的。他们甚至不需要亲自到那里——许多团队都有一只熊或橡皮鸡来做这个，并且我个人发现写详细的笔记会非常有益。在你解释的时候，请思考每一行代码，可能发生什么，它访问哪些数据，等等。问自己一些关于代码的问题，并解释答案。我发现这是一个非常强大的技巧——通过问自己这些问题并仔细思考答案，问题往往会自己显现出来。这些问题对任何代码评审都有帮助，而不仅仅是在评审你自己的代码时。

**当评审多线程代码时需要思考的问题**

正如我已经提到的，对于一个评审者(不管是代码的作者还是其他人)来说，思考与被评审的代码相关的特定问题是很有用的。这些问题可以将评审者的注意力集中在代码的相关细节上，并有助于识别潜在的问题。我喜欢问的问题包括如下，尽管这绝对不是一个详尽的列表。你可能会发现其他问题可以帮助你更好地集中注意力。这里是我的问题列表：

- 需要保护哪些数据不受并发访问的影响？
- 如何确保数据受到了保护？
- 此刻其他线程可能在代码的什么地方？
- 这个线程持有哪些互斥锁？
- 其他线程可能持有哪些互斥锁？
- 这个线程完成的操作和另一个线程完成的操作之间是否有顺序要求？这些要求是如何执行的？
- 这个线程加载的数据仍然有效吗？它可能被其他线程修改过吗？
- 如果你假设另一个线程可能正在修改数据，那么这意味着什么？如何确保这种情况永远不会发生？

最后一个问题是我最喜欢的，因为它让我思考线程之间的关系。通过假设存在与特定代码行相关的 bug，你就可以充当侦探并跟踪原因。为了让自己确信没有 bug，你必须考虑每一个边界情况和可能的顺序。当数据在其生命周期内受到多个互斥锁的保护时，这一点特别有用，例如在第 6 章中的线程安全队列中，队列的头部和尾部有单独的互斥锁：为了确信在持有一个互斥锁的时候，访问是安全的，你必须确保持有另一个互斥锁的线程不能也访问相同的元素。它还清楚地表明，公有数据，或其他代码可以轻易获得指针或引用的数据，必须接受特别的审查。

列表中倒数第二个问题也很重要，因为它解决了一个很容易犯的错误：如果释放一个互斥锁，然后再重新获得它，则必须假定其他线程可能已经修改了共享数据。尽管这是显而易见的，但如果互斥锁不是立即可见的——可能是因为它们位于对象的内部——你可能在不知不觉中就这样做了。在第 6 章中，你看到了在线程安全数据结构上提供的函数粒度太细的情况下，它是如何导致竞争条件和 bug 的。然而对于一个非线程安全的栈，让 top()和 pop()操作分离是有意义的，对一个可能被多个线程同时访问的栈，现在情况不再是这样，因为内部互斥锁在两次调用之间被释放，所以另一个线程可以修改栈。正如你在第 6 章中看到的，解决方案是将这两个操作结合起来，使它们都在同一个互斥锁的保护下执行，从而消除潜在的竞争条件。

好了，你已经评审了代码(或者让其他人评审了代码)。确信没有 bug。常言道，布丁好不好，吃了才知道——如何测试你的代码来证实或反驳你对它没有 bug 的信念呢？

## 11.2.2 通过测试定位并发相关的 bug

在开发单线程应用程序时，测试应用程序是相对简单的，只是比较耗时。原则上，可以确定所有可能的输入数据集(或者至少是所有有趣的用例)，并在应用程序中运行它们。如果应用程序产生了正确的行为和输出，你就知道它在给定的输入集上没有问题。测试错误状态，例如处理磁盘错误，要复杂得多，但思路是一样的：设置初始条件并允许应用程序运行。

测试多线程代码要困难一个数量级，因为线程的精确调度是不确定的，并且可能会随着运行而变化。因此，即使你使用相同的输入数据运行应用程序，它也可能在某些时候正常工作，而在其他时候，如果有竞争条件潜伏在代码中，则会失败。并且具有潜在的竞争条件并不意味着代码总是会失败，只是有时可能会失败。

鉴于再现并发相关 bug 的固有困难，仔细设计测试是划算的。你肯定希望每个测试运行最少量的代码就可以潜在的证明一个问题，这样，如果测试失败，你可以最好地隔离出错的代码——最好直接测试并发队列，以验证并发推送和弹出是否有效，而不是通过使用该队列的整个代码块进行测试。如果你在设计代码时就考虑代码应该如何测试，将会很有帮助——请参阅本章后面关于可测试性设计的一节。

为了验证问题是否与并发相关，从测试中消除并发也是值得的。当所有的东西都在一个线程中运行时，如果你遇到了问题，那这就是一个常见的 bug，而不是并发相关的 bug。当试图追踪“野外”（译注：这里指在测试环境之外的场景，比如在线上服务时）发生的 bug 而不是在测试工具中被检测到的 bug 时，这一点尤为重要。仅仅因为 bug 发生在应用程序的多线程部分并不意味着它就自动是并发相关的。如果你正在使用线程池来管理并发级别，那么通常可以设置一个配置参数来指定工作线程的数量。如果你手动管理线程，则必须修改代码以使用单个线程进行测试。无论采用哪种方式，如果可以将应用程序减少到单个线程，就可以消除并发这一原因。另一方面，如果这个问题在单核系统上消失(即使有多个线程运行)，但在多核系统或多处理器系统上又出现了，那么就存在一个竞争条件，很可能是一个同步或内存顺序问题。

测试并发代码不仅仅是测试代码的结构；测试的结构和测试环境同样重要。如果你继续测试一个并发队列的例子，你必须考虑各种不同的场景：

- 一个线程自己调用 push()或 pop()，以验证队列是否在基本层面上正常工作
- 一个线程在空队列上调用 push()，同时另一个线程调用 pop()
- 多个线程在一个空队列上调用 push()
- 多个线程在一个满队列上调用 push()
- 多个线程在一个空队列上调用 pop()
- 多个线程在一个满队列上调用 pop()
- 多个线程在一个部分满的队列上（但队列中的项数不够所有线程消费）调用 pop()
- 多个线程调用 push()，同时一个线程在一个空队列上调用 pop()

- 多个线程调用 push()，同时一个线程在一个满队列上调用 pop()
- 多个线程调用 push()，同时多个线程在一个空队列上调用 pop()
- 多个线程调用 push()，同时多个线程在一个满队列上调用 pop()

考虑所有这些甚至更多场景后，然后你需要考虑测试环境的其他因素：

- “多线程”在每种情况下是什么意思？(是 3 个，4 个，还是 1024 个？)
- 系统中是否有足够的处理核让每个线程都运行在它自己的核上
- 测试应该运行在哪种处理器架构上
- 如何确保对测试的“同时”部分进行适当的调度

具体到你的特定情况，还有一些额外的因素需要考虑。在这四个环境因素中，第一个和最后一个因素影响测试本身的结构(在 11.2.5 节中讨论)，而另外两个因素则与正在使用的物理测试系统有关。要使用的线程数量与要测试的特定代码相关，但是有各种不同的方法构建测试以获得适当的调度。在研究这些技术之前，让我们先看看如何设计更易于测试的应用程序代码。

## 11.2.3 可测试性设计

测试多线程代码很困难，所以你想要做的是使它变得更容易。你能做的最重要的一件事就是为可测试性设计代码。关于单线程代码可测试性设计的论述有很多，并且其中很多建议仍然适用。通常，如果应用以下因素，代码可能更容易测试：

- 每个函数和类的职责清晰。
- 函数短小精悍。
- 测试可以完全控制被测代码周围的环境。
- 执行被测试的特定操作的代码是紧挨在一起的，而不是分散在整个系统中。
- 在编写代码之前，考虑如何测试代码。

所有这些对多线程代码仍然适用。实际上，我认为关注多线程代码的可测试性比关注单线程代码的可测试性更重要，因为它天生就更难测试。最后一点很重要：即使你没有在编写代码之前编写测试，在编写代码之前考虑如何测试代码也是值得的——需要使用什么输入，哪些情况可能会有问题，如何以可能有问题的方式刺激代码，等等。

设计用于测试的并发代码的最佳方法之一是消除并发。如果可以将代码分解为负责线程间通信路径的部分和操作单个线程中通信数据的部分，那就可以大幅削减问题。应用程序中对仅由该线程访问的数据进行操作的那些部分可以使用普通的单线程技术进行测试。而难以测试的并发代码（它们处理线程之间的通信，并且确保每次只有一个线程访问特定的数据块）现在也小得多，因此测试也更容易处理。

例如，如果应用程序被设计为多线程状态机，那么可以将其分成几个部分。可以使用单线程技术独立地测试每个线程的状态逻辑(它确保转换和操作对于每个可能的输入事件集都是正确的)，并使用测试工具提供可能来自其他线程的输入事件。之后，核心状态机和消

息路由的代码（确保事件以正确的顺序被正确交付到正确的线程）可以被独立测试，但需要使用多个并发线程和专门为测试设计的简单状态逻辑。

另外，如果可以将代码划分为多个读取共享数据/转换数据/更新共享数据的块，那就可以使用所有常用的单线程技术来测试转换数据的部分，因为现在这是单线程代码。所以测试多线程转换的困难问题将简化为测试共享数据的读取和更新，这就简单很多。

需要注意的一点是，库调用可以使用内部变量来存储状态，如果多个线程使用同一组库调用，状态就会成为共享的。这可能是一个问题，因为代码访问共享数据并不是显而易见的。但随着时间的推移，你会了解到这些是哪个库调用，它们特别扎眼。然后，你可以添加适当的保护和同步，或者使用对多个线程并发访问安全的替代函数。

为了可测试性而设计多线程代码，不仅仅是构建代码以尽量减少处理并发相关问题所需的代码量以及注意使用非线程安全的库调用。记住 11.2.1 节中评审代码时要问自己的问题也很有帮助。虽然这些问题并不是直接关于测试和可测试性的，但是如果带着“测试帽”来思考这些问题，并考虑如何测试代码，它将会影响你所做的设计选择，并使测试变得更容易。

现在我们已经研究了设计代码来简化测试，并可能修改代码把“并发”部分(如线程安全的容器或状态机事件逻辑)和“单线程”部分(仍然可以通过并发块与其他线程交互)分开，接下来让我们看一下测试并发感知代码的技术。

## 11.2.4 多线程测试技术

你已经透彻思考了要测试的场景，并编写了少量代码来练习要测试的函数。那如何确保执行了任何可能有问题的调度序列，以清除 bug？

有几种方法可以解决这个问题，我们从暴力测试或压力测试开始。

**暴力测试**

暴力测试背后的想法是给代码加压，看看它是否崩溃。这通常意味着运行代码很多次，可能同时运行很多线程。如果有一个 bug 只在线程以特定方式调度时才显现，那么代码运行次数越多，这个 bug 出现的可能性就越大。如果只运行一次测试，并且测试通过了，那么可能会对代码能够正常工作有一点信心。如果你连续跑十次，每次都通过了，你可能会更有信心。如果你运行这个测试 10 亿次，并且每次都通过了，就会信心爆棚。

对结果的信心取决于每次测试所测试的代码数量。如果测试非常细粒度，就像前面为线程安全队列概述的测试一样，那么这种暴力测试可以让你对代码有高度的信心。另一方面，如果要测试的代码相当大，那么可能的调度排列的数量是如此之大，以至于即使运行 10 亿次测试也可能产生低等级的可信度。

暴力测试的缺点是它可能会给你错误的信心。如果编写测试的方式意味着有问题的情况不会发生，那么你可以任意多次运行测试，测试也不会失败，即使它在稍有不同的情况

中每次都会失败。最糟糕的例子是，由于正在测试的特定系统的运行方式，有问题的情况不会在测试系统上发生。除非代码只在与被测试的系统相同的系统上运行，否则特定的硬件和操作系统组合可能不允许出现导致问题的情况。

这里的经典示例是在单处理器系统上测试多线程应用程序。因为每个线程都必须在相同的处理器上运行，所以所有的东西都是自动串行的，在真正的多处理器系统中可能遇到的许多竞争条件和乒乓缓存问题都消失了。不过，这并不是唯一的变量；不同的处理器架构提供了不同的同步和顺序设施。例如，在 x86 和 x86-64 架构上，原子加载操作总是相同的，无论标记为 memory_order_relaxed 还是 memory_order_seq_cst(参见第 5.3.3 节)。这意味着使用宽松内存顺序编写的代码可以在 x86 架构的系统上工作，而在使用更细粒度内存顺序指令(如 SPARC)的系统上就会失败。

如果需要你的应用程序可以跨一系列目标系统移植，那么在这些系统的代表性实例上测试它是很重要的。这就是为什么我在 11.2.2 节中将用于测试的处理器架构列为要考虑的问题。

避免潜在的错误信心是成功的蛮力测试的关键。这需要仔细考虑测试设计，不仅要考虑到被测试代码单元的选择，还要考虑到测试工具的设计和测试环境的选择。你需要确保尽可能多地测试代码路径和可能的线程交互。不仅如此，还需要知道哪些选项覆盖了，哪些选项未被测试。

尽管暴力测试确实让你对代码有了一定程度的信心，但它不能保证找到所有的问题。如果你有时间将其应用于代码和合适的软件，那么有一种技术可以保证找到问题。我称之为*组合模拟测试*（combination simulation testing）。

**组合模拟测试**

这有点拗口，所以我会解释我的意思。其理念是使用一个特殊的软件来运行代码，该软件模拟代码的真实运行环境。你可能知道一些软件允许在一台物理计算机上运行多个虚拟机，这里虚拟机的特性和它的硬件是通过监测软件来模拟的。这里的思想是类似的，除了不是模拟系统，模拟软件记录来自每个线程的数据访问、锁和原子操作的序列。然后，它使用 C++内存模型的规则对每一个允许的操作组合重复运行，并识别竞争条件和死锁。

虽然这个详尽的组合测试保证找到所有系统设计用于检测的问题，但除了最简单的程序，它将花费大量的时间，因为组合的数量随着线程的数量和每个线程执行的操作的数量呈指数增加。这种技术最好用于对个别代码片段进行细粒度测试，而不是整个应用程序。另一个明显的缺点是，它依赖于模拟软件（可以处理代码中使用的操作）的可用性。

因此，你有一种技术，它涉及到在正常条件下多次运行你的测试，但可能会遗漏问题，并且你有一种技术，涉及到在特殊条件下多次运行你的测试，但更有可能发现存在的任何问题。还有其他选择吗？

第三种选择是使用一个库，在运行测试时检测问题。

**使用专用库监测测试暴露的问题**

尽管这个选项没有提供组合模拟测试的详尽检查，但是可以通过使用库同步原语(如互斥锁、锁和条件变量)的特殊实现来识别许多问题。例如，通常要求对共享数据块的所有访问都要锁住一个特定的互斥锁。如果可以检查访问数据时锁住了哪些互斥锁，就可以验证访问数据时调用线程确实锁住了适当的互斥锁，如果不是这样，就报告失败。通过以某种方式标记共享数据，可以允许库为你检查这一点。

如果一个特定线程同时持有多个互斥锁，这个库实现还可以记录锁的序列。如果另一个线程以不同的顺序锁住了相同的互斥锁，那么即使测试在运行时没有死锁，这也可能被记录为潜在的死锁。

在测试多线程代码时，可以使用另一种类型的专用库，这个库的线程原语(如互斥锁和条件变量)的实现让测试的作者控制当多个线程正在等待时哪个线程获得锁，或者哪个线程被条件变量上的 notify_one()调用通知。这将允许你建立特定的场景，并验证你的代码在这些场景中是否按照预期工作。

其中一些测试设施作为 C++标准库实现的一部分提供，而其他测试设施可以构建在标准库之上，作为你测试工具的一部分。

在了解了执行测试代码的各种方法之后，现在让我们看看构建代码以实现你想要的调度的方法。

## 11.2.5 构建多线程测试代码

回到 11.2.2 节中，我说过你需要找到为测试的“同时”部分提供适当调度的方法。现在是时候看看其中涉及的问题了。

基本问题是，你需要安排一组线程，每个线程在指定的时间执行选定的代码段。在最基本的情况下，你有两个线程，但这可以很容易地扩展到更多线程。在第一步中，你需要确定每个测试的不同部分：

- 必须在任何操作之前执行的通用设置代码
- 必须在每个线程上运行的线程特定的设置代码
- 你希望为每个线程并发运行的代码
- 要在并发执行完成后运行的代码，可能包括对代码状态的断言

为了进一步解释，让我们考虑 11.2.2 节测试列表中的一个具体示例：一个线程在空队列上调用 push()，同时另一个线程调用 pop()。

通用设置代码比较简单：必须创建队列。执行 pop()的线程没有线程特定的设置代码。执行 push()的线程的线程特定的设置代码取决于队列的接口和所存储的对象类型。如果要存储对象的构造成本很高，或者必须进行堆分配，那么你希望将此作为线程特定的设置的一部分，这样就不会影响测试。另一方面，如果队列只是存储普通的 int，那么在设置代码中构造 int 并不能获得任何好处。被测试的代码相对简单——从一个线程调用 push()，从另一个线程调用 pop()——但“完成后”的代码呢？

在本例中，这取决于你希望 pop()做什么。如果它应该在有数据之前阻塞，那么显然你希望看到返回的数据是提供给 push()调用的数据，并且之后队列为空。如果 pop()不是阻塞的，并且即使队列是空的也可以完成，那么你需要测试两种可能：要么 pop()返回提供给 push()的数据项并且队列为空，要么 pop()表示没有数据，并且队列有一个元素。其中一个必须为真；你想要避免的场景是 pop()示意了“没有数据”但是队列是空的，或者 pop()返回了值，队列仍然非空。为了简化测试，假设你有一个阻塞的 pop()。因此，最终的代码是一个断言，即弹出的值是推入的值，且队列为空。

现在，在确定了不同的代码块之后，你需要尽最大努力确保一切都按计划运行。一种方法是当一切准备就绪时，使用一组 std::promise 来指示。每个线程都设置一个承诺来表示它已经准备好了，然后等待从第三个 std::promise 获得的 std::shared_future（拷贝）；主线程等待来自所有线程的所有承诺被设置，然后触发线程开始（go）。这可以确保每个线程都已经启动，并且出现在应该并发运行的代码块之前；任何线程特定的设置都应该在设置那个线程的承诺之前完成。最后，主线程等待线程完成并检查最终状态。你还需要注意异常，并确保当开始（go）信号不会发生的时候，没有任何线程会等待这个信号。下面的清单显示了构建这个测试的一种方法。

清单 11.1 对一个队列并发调用 push()和 pop()的测试用例

```
void test_concurrent_push_and_pop_on_empty_queue()
{
  threadsafe_queue<int> q;  // 1

  std::promise<void> go,push_ready,pop_ready;  // 2
  std::shared_future<void> ready(go.get_future());  // 3

  std::future<void> push_done;  // 4
  std::future<int> pop_done;

  try
  {
    push_done=std::async(std::launch::async,  // 5
                         [&q,ready,&push_ready]()
                         {
                           push_ready.set_value();
                           ready.wait();
                           q.push(42);
                         }
      );

    pop_done=std::async(std::launch::async,  // 6
                        [&q,ready,&pop_ready]()
                        {
                          pop_ready.set_value();
                          ready.wait();
```

```
                        return q.pop();  // 7
                    }
    );

    push_ready.get_future().wait();  // 8
    pop_ready.get_future().wait();
    go.set_value();  // 9

    push_done.get();  // 10
    assert(pop_done.get()==42);  // 11
    assert(q.empty());
  }
  catch(...)
  {
    go.set_value();  // 12
    throw;
  }
}
```

代码结构与前面描述的基本相同。首先，创建空队列作为通用设置的一部分①。然后，为“就绪”信号创建所有承诺②，并且为 go 信号获取一个 std::shared_future③。再后，创建期望用来表示线程已经结束④。这些都必须放在 try 块外面，这样你就可以在异常上设置 go 信号，而不必等待测试线程完成(这将导致死锁——测试代码中的死锁并不理想)。

然后在 try 块中可以启动线程⑤和⑥——使用 std::launch::async 保证每个任务运行在自己的线程上。注意，使用 std::async 使异常安全任务比使用普通的 std::thread 更容易，因为期望的析构函数将会连接线程。lambda 捕获指定了每个任务将引用队列和用于通知就绪的承诺，同时使用一个从 go 承诺中获得的 ready 期望的拷贝。

如前所述，每个任务设置自己的就绪信号，然后在运行测试代码之前等待通用的 ready 信号。主线程则执行相反的操作——在信号通知他们开始真正的测试之前⑨，它等待来自两个线程的信号⑧。

最后，主线程在来自异步调用的期望上调用 get()来等待任务结束⑩⑪，并检查结果。注意 pop 任务通过期望返回检索值⑦，所以你可以使用它来为断言获取结果⑪。

如果有异常抛出，可以设置 go 信号，以避免出现悬垂线程，并重新抛出异常⑫。任务对应的期望④最后声明，所以它们首先被销毁，并且如果它们还没有就绪，它们的析构函数会等待任务完成。

尽管为了测试两个简单的调用，这似乎是相当多的样板，但为了有最好的机会测试你想要测试的内容，使用类似的东西是有必要的。例如，启动一个线程可能是一个非常耗时的过程，所以如果你没有让线程等待 go 信号，那么 push 线程可能在 pop 线程开始之前就已经完成了，这将完全违背测试的目的。以这种方式使用期望可以确保两个线程在同一个

期望上运行和阻塞。解除阻塞的期望允许两个线程运行。一旦你熟悉了这个结构，在相同的模式中创建新的测试应该是相对简单的。对于需要两个以上线程的测试，这个模式很容易扩展到额外的线程。

到目前为止，我们一直在研究多线程代码的正确性。尽管这是最重要的问题，但它不是你进行测试的唯一原因：测试多线程代码的性能也很重要，所以让我们接下来看看这个问题。

### 11.2.6 测试多线程代码的性能

选择在应用程序中使用并发的主要原因之一是利用越来越流行的多核处理器来提高应用程序的性能。因此，正如你对任何其他优化尝试所做的那样，测试你的代码来确定性能确实提高了就很重要。

为了提高性能而使用并发的一个特殊问题是可扩展性——在其他条件相同的情况下，你希望代码在 24 核机器上的运行速度大约是单核机器的 24 倍，或者处理的数据大约是单核机器的 24 倍。你不希望代码在双核机器上运行速度是双核机器的两倍，但在 24 核机器上运行速度更慢。正如在 8.4.2 节中看到的，如果代码的重要部分只在一个线程上运行，这可能会限制潜在的性能增益。因此，在开始测试之前，有必要看一下代码的整体设计，这样你就知道是有希望得到 24 倍的改进，还是代码的串行部分意味着你被限制在 3 倍的最大值。

正如你在前面的章节中已经看到的，处理器之间竞争访问数据结构可能会对性能产生很大的影响。当处理器数量很小时，可以很好地随处理器数量扩展的东西，在当处理器数量很大时，可能由于争用的大量增加表现得很差。

因此，在测试多线程代码的性能时，最好在具有尽可能多不同配置的系统上检查性能，这样你就可以对可扩展性做到心中有数。至少，你应该在一个单处理器系统和一个拥有尽可能多的处理核的系统上进行测试。

## 总结

在本章中，我们讨论了可能会遇到的与并发相关的各种类型的 bug，从死锁和活锁到数据竞争以及其他有问题的竞争条件。随后我们使用了定位 bug 的技术。这些包括在代码评审期间要考虑的问题，编写可测试代码的指南，以及如何为并发代码构建测试结构。最后，我们讨论了一些有助于测试的工具组件。