

需要整本电子书，联系我QQ：2667271557

MANNING

C++ 并发编程实战

C++
Concurrency
IN ACTION

[美] Anthony Williams 著
周全 梁娟娟 宋真真 许敏 译

需要整本电子书，联系我QQ：2667271557

人民邮电出版社
POSTS & TELECOM PRESS

需要整本电子书，联系我QQ：2667271557

C++ 并发编程实战

C++
Concurrency
IN ACTION

〔美〕Anthony Williams 著

周全 梁娟娟 宋真真 许敏 译

需要整本电子书，联系我QQ：2667271557

人民邮电出版社
北京

需要整本电子书，联系我QQ：2667271557

图书在版编目（CIP）数据

C++并发编程实战 / (美) 威廉姆斯 (Williams, A.)
著；周全等译. — 北京：人民邮电出版社，2015.6
ISBN 978-7-115-38732-5

I. ①C… II. ①威… ②周… III. ①C语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2015)第069669号

版 权 声 明

Original English language edition, entitled C++ Concurrency in Action by Anthony Williams, published by Manning Publications Co., 209 Bruce Park Avenue, Greenwich, CT 06830. Copyright ©2012 by Manning Publications Co.

Simplified Chinese-language edition copyright ©2015 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由 Manning Publications Co.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

-
- ◆ 著 [美]Anthony Williams
 - 译 周 全 梁娟娟 宋真真 许 敏
 - 责任编辑 陈冀康
 - 责任印制 张佳莹 焦志炜
 - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
 - 邮编 100164 电子邮件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 北京鑫正大印刷有限公司印刷
 - ◆ 开本：800×1000 1/16
 - 印张：31.75
 - 字数：686千字 2015年6月第1版
 - 印数：1—3000册 2015年6月北京第1次印刷
 - 著作权合同登记号 图字：01-2009-3542号
-

定价：89.00元

读者服务热线：(010)81055410 印装质量热线：(010)81055316

反盗版热线：(010)81055315

需要整本电子书，联系我QQ：2667271557

内容提要

本书是一本基于 C++11 新标准的并发和多线程编程深度指南。内容包括 `std::thread`、`std::mutex`、`std::future` 和 `std::async` 等基础类的使用，内存模型和原子操作、基于锁和无锁数据结构的构建，以及并行算法和线程管理，最后还介绍了多线程代码的测试。本书的附录部分还对 C++11 新语言特性中与多线程相关的项目进行了简要的介绍，并提供了 C++11 线程库的完整参考。

本书适合于需要深入了解 C++多线程开发的读者，以及使用 C++进行各类软件开发的开发人员、测试人员阅读。使用第三方线程库的读者，也可以从本书后面的章节中了解到相关的指引和技巧。同时，本书还可以作为 C++11 线程库的参考工具书。

序

我是在离开大学后的第一份工作中遇到多线程编程的概念的。我们当时在编写一个数据处理应用程序，它需要用传入的数据记录来填充数据库。虽然数据很多，但每个数据都是独立的，并且在它被插入数据库前需要大量的处理。为了充分利用 10-CPU UltraSPARC 的能力，我们在多线程中运行代码，让每个线程处理它们自己的一组输入数据。在这个过程中，我们用 C++ 编写代码，使用 POSIX 线程，并且犯了相当多的错误。尽管多线程对我们而言都是全新的，但我们最终还是完成了。正是在这个项目中的工作，我第一次了解了 C++ 标准委员会和全新发布的 C++ 标准。

我对多线程和并发有前所未有的兴趣。虽然别人认为它困难、复杂，是问题之源，但我却视它为强大的工具，因为它可以允许你利用现有的硬件让代码运行得更快。随后我了解到它即便是在单核硬件上也能提高应用程序的响应和性能，通过使用多线程来隐藏诸如 I/O 这样耗费时间的操作延迟。我还了解了它怎样在 OS 级别工作以及 Intel CPU 如何处理任务切换。

同时，我对 C++ 的兴趣给我带来了与 ACCU 以及 BSI 的 C++ 标准专家组以及 Boost 接触的机会。我凭兴趣跟进了 Boost 线程库的初期开发，当它被最初的开发者抛弃时，我便趁机介入了。我从此成为了 Boost 线程库最主要的开发者和维护者。

当 C++ 标准委员会的工作从修正现有标准的缺陷转移到编写下一代标准（希望在 2009 年之前完成故命名为 C++0x，现在正式为 C++11，因为最终发布于 2011 年）的提案后，我更多地参与到 BSI 并且开始起草我自己的提案。多线程刚被明确地提上议事日程，我就全力以赴地投入进去，并且撰写或共同撰写了许多与多线程和并发相关的提案，它们组成了新标准的一部分。我感到很荣幸，可以有这个机会用这种方式组合我的计算机相关的两大兴趣——C++ 和多线程。

这本书借鉴了我在 C++ 和多线程上的全部经验，其目标是教会其他 C++ 开发者如何安全并有效地使用 C++11 线程库。我也希望能顺带传授一些我在这方面的热情。

译者简介

周全，软件工程师，毕业于中国科学技术大学信息学院，现就职于中国人民银行合肥中心支行科技处。有着丰富的系统集成和运维经验，对虚拟化也有较深入的研究，曾从事.NET 开发和培训工作。读者可以通过 Email:zhouquan.pbc@foxmail.com 与他联系。

梁娟娟，2010 年毕业于中国科学技术大学信息技术学院，现就职于中国人民银行合肥中心支行。

宋真真，网络工程师，2008 年毕业于合肥工业大学计算机与信息学院，现就职于中国人民银行合肥中心支行科技处，参与软件开发、项目管理等工作，爱好数据库、编程等研究。读者可以通过 Email: hfut_szz@sina.com 与她联系。

许敏，软件工程师，2005 年获得软件测试工程师证书。现就职于中国人民银行合肥中心支行科技处，负责项目管理工作。读者可以通过 Email: xu_min@sina.com 与她联系。

需要整本电子书，联系我QQ：2667271557

致 *Kim*、*Hugh* 和 *Erin*。

需要整本电子书，联系我QQ：2667271557

致谢

我要首先对我的妻子 Kim 说一声“谢谢”，感谢在我写这本书的时候她给我的爱和支持。写书占用了我最近 4 年很多的空闲时间，没有她的耐心、支持和理解，我不可能做到。

其次，我想感谢 Manning 的团队，包括总编 Marjan Bace、副总编 Michael Stephens、开发编辑 Cynthia Kane、编审 Karen Tegtmeier、文字编辑 Linda Recktenwald、校对 Katie Tennant 和制作经理 Mary Piergies。他们使得这本书成功面世，没有他们的努力，你現在就读不到这本书了。

我还想感谢 C++ 标准委员会的其他成员，他们为多线程工具上撰写了文件。正是 Andrei Alexandrescu、Pete Becker、Bob Blainer、Hans Boehm、Beman Dawes、Lawrence Cowl、Peter Dimov、Jeff Garland、Kevlin Henney、Howard Hinnant、Ben Hutchings、Jan Kristofferson、Doug Lea、Paul McKenney、Nick McLaren、Clark Nelson、Bill Pugh、Raul Silvera、Herb Sutter、Detlef Vollmann 和 Michael Wong，以及所有在文件上批注的人们，他们在委员会会议上进行了讨论，还有其他人的帮助才形成了 C++11 中的多线程和并发支持。

最后，我还想感谢下面的人：Dr. Jamie Allsop、Peter Dimov、Howard Hinnant、Rick Molloy、Jonathan Wakely 和 Dr. Russel Winder，他们的建议极大地改进了这本书。另外特别感谢 Russel 的详细审阅，还有技术校对 Jonathan，在编辑出版过程中极其用心地检查了终稿中所有内容中的错误（当然所有的错误都是由我造成的）。此外我想感谢我的审稿专家组：Ryan Stephens、Neil Horlock、John Taylor Jr.、Ezra Jivan、Joshua Heyer、Keith S. Kim、Michele Galli、Mike Tian-Jian Jiang、David Strong、Roger Orr、Wagner Rick、Mike Buksas 和 Bas Vodde。还要谢谢 MEAP 版的读者，他们花费时间来指出错误或是强调了需要阐明的部分。

前言

这本书是对新 C++ 标准中的并发和多线程工具的深度指南，内容包括从 `std::thread`、`std::mutex` 和 `std::async` 的基本用法，到复杂的原子操作与内存模型。

路线图

前 4 章介绍了由类库提供的各种类库工具以及他们如何使用。

第 5 章涵盖了内存模型和原子操作的低阶基础，包括原子操作怎样在其他代码上强制实行排序约束，并标志着导言章节的结束。

第 6 章和第 7 章开始涵盖高阶的论题，包括一些如何使用基础工具来构造更复杂的数据结构的示例——第 6 章中基于锁的数据结构，以及第 7 章中无锁的数据结构。

第 8 章继续高阶论题，包括如何设计多线程代码的指南，涵盖了影响性能的论点，以及各种并行算法的示范实现。

第 9 章涵盖了线程管理——线程池、工作队列和中断操作。

第 10 章包括了测试和调试——bug 的类型，定位它们的技巧，如何测试它们，等等。

附件包含了对由新标准引入的与多线程相关的一些新语言工具的简要介绍，第 4 章中提到的消息传递库的具体实现，以及 C++11 线程库的完整参考。

谁应该阅读本书

如果你打算用 C++ 编写多线程代码，你就应该阅读本书。如果你正要使用 C++ 标准库中新的多线程工具，这本书是必备的指南。如果你正使用替代的线程库，后面几章中的指引和技巧应该也是有用的。

假设你对 C++ 已经有了很好的了解，但对新的语言特性却不甚熟悉，这些在附录 A 中也能找到答案。假定你之前没有多线程编程的知识和经验，那就更应该阅读本书。

如何使用本书

如果你以前从未写过多线程代码，我建议你按顺序从头到尾阅读本书，可以跳过第 5 章中的细节部分，但第 7 章大量依赖第 5 章中的材料，所以如果你跳过了第 5 章，你一定要阅览第 7 章，除非你曾读过。

如果你之前未曾使用过 C++11 语言工具，在你开始确定准备快速开始书中例子之前最好浏览一下附录 A。新语言工具的使用凸显在文字之中，然而，当你遇到了之前没有见过的东西时，总是可以翻看附录的。

如果你在其他环境中拥有大量编写多线程代码的经验，开始的几章可能让你值得浏览一遍，以便你可以看看你了解的工具怎样映射到新 C++ 标准中。如果你打算用原子变量做一些低阶的工作，第 5 章就是必需的。为了确认你熟悉多线程 C++ 中类似异常安全的东西，值得阅览一下第 8 章。如果你在脑海中有特定的任务，索引和目录可以帮助你快速找到相关的章节。

一旦你打算促进 C++ 线程库的使用，附录 D 应该仍然有用，比如查询每个类和函数调用的细节。你可能会想一次又一次地翻回主章节，来刷新你对某一概念的使用或者看一看示例代码。

代码约定和下载

所有代码清单和正文中的源代码，出现等宽字体 (like this)，是为了便于从常规文本中区分出来。代码注解伴随着很多清单，指出重要的概念。在有些情况下，数字符号往往指向清单后面的注释。

本书中所有的工作示例源代码可以在出版社网址下载，www.manning.com/CPlusPlusConcurrencyinAction。

软件需求

为了直截了当地使用本书中的代码，你需要一个最新的 C++ 编译器，它支持示例中使用的新的 C++11 语言特性（参见附录 A），并且你需要 C++ 标准线程库的副本。

在撰写本书的时候，g++ 是我所知道的唯一带有标准线程库实现的编译器，尽管 Microsoft Visual Studio 2011 预览版也包括了实现。线程库的 g++ 实现最早是在 g++ 4.3 中引入了基本形式，并且在后来的版本中进行了扩展。g++ 4.3 也引入了一些新 C++11 语言特性的初次支持；对新语言特性更多的支持在各个后续版本中。详情参见 g++ C++11 状态页面¹。

Microsoft Visual Studio 2010 提供了一些新 C++11 语言特性，例如右值引用和 lambda

¹ GNU Compiler Collection C++0x/C++11 状态页面，<http://gcc.gnu.org/projects/cxx0x.html>。

函数，但并不带有线程库的实现。

作者的公司 Just Soft Solutions Ltd，出售为 Microsoft Visual Studio 2005、Microsoft Visual Studio 2008、Microsoft Visual Studio 2010 和 g++ 的各种版本¹设计的 C++11 标准线程库的完整实现。这些实现已被用来测试本书中的示例。

Boost 线程库²提供了基于 C++ 标准线程库提案的 API，可以移植到许多平台。本书中的大部分示例可以通过审慎地将 `std::` 替换成 `boost::` 进行修改，并使用适当的 `#include` 指令，来与 Boost 线程库一起工作。在 Boost 线程库中有少数工具不受支持（比如 `std::async`）或者拥有不同的名称（比如 `boost::unique_future`）。

作者在线

购买 C++ Concurrency in Action 包括了免费访问由 Manning 出版社运营的私有网络论坛，在这里你可以对本书做出评论，提问技术问题，并且从作者和其他用户那里得到帮助。如果要访问此论坛并订阅它，可以访问网址 www.manning.com/CPlusPlusConcurrencyinAction。该页面提供了论坛的使用指南以及论坛上的行为规则。

Manning 对我们读者的承诺是提供一个场所，在这里每个读者之间以及读者和作者之间可以进行有意义的对话。就作者而言并没有承诺任何规定的参与量，作者对本书论坛的贡献只是义务的（且无偿的）。我们建议你试着向作者提问一些有挑战性的问题，以免他失去兴趣！

作者在线论坛以及过往讨论的归档，在本书在印期间都可以从出版社网站进行访问。

¹ C++ 标准线程库的 `just::thread` 实现，<http://www.stdthread.co.uk>。

² Boost C++ 库集合，<http://www.boost.org>。

资源

印刷资源

Cargill, Tom, "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.

Hoare, C.A.R., *Communicating Sequential Processes* (Prentice Hall International, 1985), ISBN 0131532898. Also available at <http://www.usingcsp.com/cspbook.pdf>.

Michael, Maged M., "Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes" in *PODC'02: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing* (2002), ISBN 1-58113-485-1.

———. U.S. Patent and Trademark Office application 20040107227, "Method for efficient implementation of dynamic lock-free data structures with safe memory reclamation."

Sutter, Herb, *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions* (Addison Wesley Professional, 1999), ISBN 0-201-61562-2.

———. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," in *Dr. Dobbs's Journal* 30, no. 3 (March 2005). Also available at <http://www.gotw.ca/publications/concurrency-ddj.htm>.

在线资源

Atomic Ptr Plus Project Home, <http://atomic-ptr-plus.sourceforge.net/>.

Boost C++ library collection, <http://www.boost.org>.

C++0x/C++11 Support in GCC, <http://gcc.gnu.org/projects/cxx0x.html>.

C++11—The Recently Approved New ISO C++ Standard, <http://www.research.att.com/~bs/C++0xFAQ.html>.

Erlang Programming Language, <http://www.erlang.org/>.

GNU General Public License, <http://www.gnu.org/licenses/gpl.html>.

Haskell Programming Language, <http://www.haskell.org/>.

IBM Statement of Non-Assertion of Named Patents Against OSS, <http://www.ibm.com/ibm/licensing/patents/pledgedpatents.pdf>.

Intel Building Blocks for Open Source, <http://threadingbuildingblocks.org/>.

The just::thread Implementation of the C++ Standard Thread Library, <http://www.stdthread.co.uk>.

Message Passing Interface Forum, <http://www.mpi-forum.org/>.

Multithreading API for C++0X—A Layered Approach, C++ Standards Committee Paper N2094, <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2094.html>.

OpenMP, <http://www.openmp.org/>.

SETI@Home, <http://setiathome.ssl.berkeley.edu/>.

简要目录

第 1 章 你好，C++并发世界	1	第 9 章 高级线程管理	258
第 2 章 管理线程	13	第 10 章 多线程应用的测试与调试	285
第 3 章 在线程间共享数据	31	附录 A C++11 部分语言特性简明 参考	299
第 4 章 同步并发操作	63	附录 B 并发类库简要对比	324
第 5 章 C++内存模型和原子类型操作	97	附录 C 消息传递框架与完整的 ATM 示例	326
第 6 章 设计基于锁的并发数据结构	140	附录 D C++线程类库参考	344
第 7 章 设计无锁的并发数据结构	170		
第 8 章 设计并发代码	213		

目录

第 1 章 你好，C++并发世界 1

- 1.1 什么是并发 2
 - 1.1.1 计算机系统上的并发 2
 - 1.1.2 并发的途径 3
- 1.2 为什么使用并发 5
 - 1.2.1 为了划分关注点而使用并发 5
 - 1.2.2 为了性能而使用并发 6
 - 1.2.3 什么时候不使用并发 7
- 1.3 在 C++中使用并发和多线程 8
 - 1.3.1 C++多线程历程 8
 - 1.3.2 新标准中的并发支持 9
 - 1.3.3 C++线程库的效率 9
 - 1.3.4 平台相关的工具 10
- 1.4 开始入门 11
- 1.5 小结 12

第 2 章 管理线程 13

- 2.1 基本线程管理 13
 - 2.1.1 启动线程 14
 - 2.1.2 等待线程完成 16
 - 2.1.3 在异常环境下的等待 17
 - 2.1.4 在后台运行线程 19
- 2.2 传递参数给线程函数 20
- 2.3 转移线程的所有权 23
- 2.4 在运行时选择线程数量 26

- 2.5 标识线程 28

- 2.6 小结 29

第 3 章 在线程间共享数据 31

- 3.1 线程之间共享数据的问题 32
 - 3.1.1 竞争条件 33
 - 3.1.2 避免有问题的竞争条件 34
- 3.2 用互斥元保护共享数据 35
 - 3.2.1 使用 C++中的互斥元 35
 - 3.2.2 为保护共享数据精心组织代码 36
 - 3.2.3 发现接口中固有的竞争条件 38
 - 3.2.4 死锁：问题和解决方案 44
 - 3.2.5 避免死锁的进一步指南 46
 - 3.2.6 用 std::unique_lock 灵活锁定 51
 - 3.2.7 在作用域之间转移锁的所有权 52
 - 3.2.8 锁定在恰当的粒度 54
- 3.3 用于共享数据保护的替代工具 56
 - 3.3.1 在初始化时保护共享数据 56
 - 3.3.2 保护很少更新的数据结构 59

3.3.3 递归锁 61

3.4 小结 62

4

第4章 同步并发操作 63

4.1 等待事件或其他条件 63

4.1.1 用条件变量等待条件 65

4.1.2 使用条件变量建立一个
线程安全队列 67

4.2 使用 future 等待一次性
事件 71

4.2.1 从后台任务中返回值 72

4.2.2 将任务与 future 相关联 74

4.2.3 生成(std::)promise 77

4.2.4 为 future 保存异常 79

4.2.5 等待自多个线程 80

4.3 有时间限制的等待 82

4.3.1 时钟 83

4.3.2 时间段 84

4.3.3 时间点 85

4.3.4 接受超时的函数 86

4.4 使用操作同步来简化
代码 88

4.4.1 带有 future 的函数式
编程 88

4.4.2 具有消息传递的同步
操作 92

4.5 小结 96

5

第5章 C++内存模型和原子 类型上操作 97

5.1 内存模型基础 98

5.1.1 对象和内存位置 98

5.1.2 对象、内存位置以及
并发 99

5.1.3 修改顺序 100

5.2 C++中的原子操作及
类型 100

5.2.1 标准原子类型 101

5.2.2 std::atomic_flag 上的
操作 103

5.2.3 基于 std::atomic<bool>的
操作 105

5.2.4 std::atomic<T*>上的操作：
指针算术运算 107

5.2.5 标准原子整型的
操作 108

5.2.6 std::atomic<>初级类

模板 109

5.2.7 原子操作的自由函数 111

5.3 同步操作和强制

顺序 112

5.3.1 synchronizes-with
关系 114

5.3.2 happens-before 关系 114

5.3.3 原子操作的内存
顺序 116

5.3.4 释放序列和
synchronizes-with 133

5.3.5 屏障 135

5.3.6 用原子操作排序非原子
操作 137

5.4 小结 138

6

第6章 设计基于锁的并发 数据结构 140

6.1 为并发设计的含义是
什么 141

6.2 基于锁的并发数据
结构 142

6.2.1 使用锁的线程
安全栈 142

6.2.2 使用锁和条件变量的线程
安全队列 145

6.2.3 使用细粒度锁和条件变量
的线程安全队列 149

6.3 设计更复杂的基于锁的
数据结构 160

6.3.1 编写一个使用锁的线程
安全查找表 160

6.3.2 编写一个使用锁的线程
安全链表 165

6.4 小结 169

7

第7章 设计无锁的并发数据 结构 170

7.1 定义和结果 171

7.1.1 非阻塞数据结构的
类型 171

7.1.2 无锁数据结构 172

7.1.3 无等待的数据结构 172

7.1.4 无锁数据结构的优点与
缺点 172

7.2 无锁数据结构的

例子 173

- 7.2.1 编写不用锁的线程安全栈 174
- 7.2.2 停止恼人的泄漏：在无锁数据结构中管理内存 178
- 7.2.3 用风险指针检测不能被回收的结点 182
- 7.2.4 使用引用计数检测结点 189
- 7.2.5 将内存模型应用至无锁栈 194
- 7.2.6 编写不用锁的线程安全队列 198

7.3 编写无锁数据结构的

准则 209

- 7.3.1 准则：使用 `std::memory_order_seq_cst` 作为原型 210
- 7.3.2 准则：使用无锁内存回收模式 210
- 7.3.3 准则：当心ABA问题 210
- 7.3.4 准则：识别忙于等待的循环以及辅助其他线程 211

7.4 小结 211

第8章 设计并发代码 213

8.1 在线程间划分工作的技术 214

- 8.1.1 处理开始前在线程间划分数据 214
- 8.1.2 递归地划分数据 215
- 8.1.3 以任务类型划分工作 219

8.2 影响并发代码性能的因素 222

- 8.2.1 有多少个处理器 222
- 8.2.2 数据竞争和乒乓缓存 223
- 8.2.3 假共享 225
- 8.2.4 数据应该多紧密 225
- 8.2.5 过度订阅和过多的任务切换 226

8.3 为多线程性能设计数据结构 226

- 8.3.1 为复杂操作划分数组元素 227

8.3.2 其他数据结构中的数据访问方式 228

8.4 为并发设计时的额外考虑 230

- 8.4.1 并行算法中的异常安全 230
- 8.4.2 可扩展性和阿姆达尔定律 237
- 8.4.3 用多线程隐藏延迟 238
- 8.4.4 用并发提高响应性 239

8.5 在实践中设计并发代码 241

- 8.5.1 `std::for_each` 的并行实现 241
- 8.5.2 `std::find` 的并行实现 243
- 8.5.3 `std::partial_sum` 的并行实现 248

8.6 总结 256

第9章 高级线程管理 258

9.1 线程池 259

- 9.1.1 最简单的线程池 259
- 9.1.2 等待提交给线程池的任务 261
- 9.1.3 等待其他任务的任务 265
- 9.1.4 避免工作队列上的竞争 267
- 9.1.5 工作窃取 269

9.2 中断线程 273

- 9.2.1 启动和中断另一个线程 274
- 9.2.2 检测一个线程是否被中断 275
- 9.2.3 中断等待条件变量 276
- 9.2.4 中断在 `std::condition_variable_any` 上的等待 279
- 9.2.5 中断其他阻塞调用 281
- 9.2.6 处理中断 281
- 9.2.7 在应用退出时中断后台任务 282

9.3 总结 284

第10章 多线程应用的测试与调试 285

10.1 并发相关错误的

类型	285
10.1.1 不必要的阻塞	286
10.1.2 竞争条件	286
10.2 定位并发相关的错误的技巧	288
10.2.1 审阅代码以定位潜在的错误	288
10.2.2 通过测试定位并发相关的错误	290
10.2.3 可测试性设计	291
10.2.4 多线程测试技术	292
10.2.5 构建多线程的测试代码	295
10.2.6 测试多线程代码的性能	297
10.3 总结	298

A

附录 A C++11 部分语言特性 简明参考	299
---------------------------	-----

B

附录 B 并发类库简要对比	324
---------------	-----

C

附录 C 消息传递框架与完整的 ATM 示例	326
---------------------------	-----

D

附录 D C++线程类库 参考	344
--------------------	-----

第 1 章 你好，C++并发世界

本章主要内容

- 何谓并发和多线程
- 为什么要在应用程序中使用并发和多线程
- C++并发支持的发展历程
- 一个简单的 C++多线程程序是什么样的

这是令 C++ 用户振奋的时刻。距 1998 年初始的 C++ 标准发布 13 年后，C++ 标准委员会给予程序语言和支持库一次重大的变革。新的 C++ 标准（也被称为 C++11 或 C++0x）于 2011 年发布并带来了许多的改变，使得 C++ 的应用更加容易并富有成效。

在 C++11 标准中一个最重要的新特性就是支持多线程程序。这是 C++ 标准第一次在语言中承认多线程应用的存在，并在库中为编写多线程应用程序提供组件。这将使得在不依赖平台相关扩展下编写多线程 C++ 程序成为可能，从而允许以有保证的行为来编写可移植的多线程代码。这也恰逢程序员寻求更多普遍的并发，特别是多线程程序，来提高应用程序的性能。

这本书讲述的就是 C++ 编程中对多线程并发的使用，以及相关的 C++ 语言特性和库工具。我会以解释并发和多线程的含义以及为什么要在应用程序中使用并发开始。在快速全方位地阐述为什么在应用程序中会不使用并发之后，我会对 C++ 中并发支持进行概述，并以一个简单的 C++ 并发实例结束这一章。具有开发多线程应用程序经验的读者可以跳过前面的小节。在随后几章将会涵盖更多广泛的例子，并且更深入地了解库工具。

本书最后附有多线程与并发全部的 C++ 标准库工具的深入参考。

那么，什么是并发（**concurrency**）和多线程（**multithreading**）？

1.1 什么是并发

在最简单和最基本的层面，并发是指两个或更多独立的活动同时发生。并发在生活中随处可见。我们可以一边走路一边说话，也可以两只手同时做不同的动作，还有我们每个人都相互独立地过我们的生活——我在游泳的时候你可以看球赛，等等。

1.1.1 计算机系统中的并发

当我们提到计算机术语的“并发”，指的是在单个系统里同时执行多个独立的活动，而不是顺序地或是一个接一个地。这并不是一种新的现象，多任务操作系统通过任务切换允许一台计算机在同一时间运行多个应用程序已司空见惯多年，一些高端的多任务处理服务器实现并发控制的历史更久远。真正有新意的是增加计算机真正并行运行多任务的普遍性，而不只是给人这种错觉。

以前，大多数计算机都有一个处理器，具有单个处理单元或核心，至今许多台式机仍是这样。这种计算机在某一时刻只可以真正执行一个任务，但它可以每秒切换任务许多次。通过做一点这个任务然后再做一点别的任务，看起来像是任务在并行发生。这就是任务切换（**task switching**）。我们仍然将这样的系统称为并发（**concurrency**），因为任务切换得太快，以至于无法分辨任务在何时会被暂挂而切换到另一个任务。任务切换给用户和应用程序本身造成了一种并发的假象。由于这只是并发的假象，当应用程序执行在单处理器任务切换环境下，与在真正的并发环境下执行相比，其行为还是有着微妙的不同。特别地，对内存模型不正确的假设（参见第 5 章）在这样的环境中可能不会出现。这将在第 10 章中作深入讨论。

包含多个处理器的计算机用于服务器和高性能计算任务已有多年，现在基于单个芯片上具有多于一个核心的处理器（多核心处理器）的计算机也成为越来越常见的台式机。无论它们拥有多个处理器或一个多核处理器（或两者兼具），这些计算机能够真正的并行运行超过一个任务。我们才称之为硬件并发（**hardware concurrency**）。

图 1.1 显示了一个计算机处理恰好两个任务时的理想情景，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）中，每个任务可以在各自的核心执行。在单核机器上做任务切换时，每个任务的块交织进行。但它们也隔开了一位（图中所示灰色分隔条的厚度大于双核机器的分隔条）。为了实现交替进行，该系统每次从一个任务切换到另一个时都得执行一次上下文切换（**context switch**），而这是需要时间的。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，算出要切换到哪

个任务，并为要切换到的任务重新加载处理器状态。然后 CPU 可能要将新任务的指令和数据的内存载入到缓存中，这可能会阻止 CPU 执行任何指令，造成进一步的延迟。

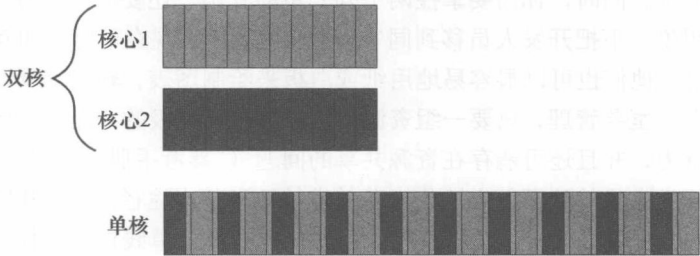


图 1.1 并发的两种方式：双核机器的并行执行对比单核机器的任务切换

尽管硬件并发的可用性在多处理器或多核系统上更显著，有些处理器却可以在一个核心上执行多个线程。要考虑的最重要的因素是硬件线程（**hardware threads**）的数量：即硬件可以真正并发运行多少独立的任务。即便是具有真正硬件并发的系统，也很容易有超过硬件可并行运行的任务要执行，所以在这些情况下任务切换仍将被使用。例如，在一个典型的台式计算机上可能会有几百个的任务在运行，执行后台操作，即使计算机在名义上是空闲的。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（或任何应用的组合）。图 1.2 显示了四个任务在一台双核机器上的任务切换，仍然是将任务整齐地划分为同等大小块的理想情况。实际上，许多因素造成了分割不均和调度不规则。这些因素中的一部分将涵盖在第 8 章中，那时我们再来看一下影响并行代码性能的因素。

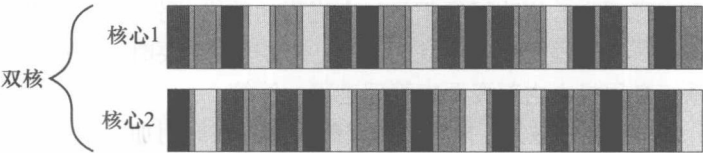


图 1.2 四个任务在两个核心之间的切换

所有的技术、功能和本书所涉及的类都可以使用，无论你的应用程序是在单核处理器还是多核处理器上运行，也不管是任务切换或是真正的硬件并发。但你可以想象，如何在你的应用程序中使用并发很大程度上取决于可用的硬件并发。这将在第 8 章中涵盖，在第 8 章我们具体研究 C++ 代码并行设计问题。

1.1.2 并发的途径

想象一下两个程序员一起做一个软件项目。如果你的开发人员在独立的办公室，它

他们可以各自平静地工作，而不会互相干扰，并且他们各有自己的一套参考手册。然而，沟通起来就不那么直接了；不能转身然后互相交谈，他们必须用电话、电子邮件或走到对方的办公室。同时，你需要掌控两个办公室的开销，还要购买多份参考手册。

现在想象一下把开发人员移到同一间办公室。他们现在可以地相互交谈来讨论应用程序的设计，他们也可以很容易地用纸或白板来绘制图表，辅助阐释设计思路。你现在只有一个办公室要管理，只要一组资源就可以满足。消极的一面是，他们可能会发现难以集中注意力，并且还可能存在资源共享的问题（“参考手册跑哪去了？”）。

组织开发人员的这两种方法代表着并发的两种基本途径。每个开发人员代表一个线程，每个办公室代表一个处理器。第一种途径是有多个单线程的进程，这就类似让每个开发人员在他们自己的办公室，而第二种途径是在单一进程里有多个线程，这就类似在同一个办公室里有两个开发人员。你可以随意进行组合，并且拥有多个进程，其中一些是多线程的，一些是单线程的，但原理是一样的。让我们在一个应用程序中简要地看一看这两种途径。

1. 多进程并发

在一个应用程序中使用并发的第一种方法，是将应用程序分为多个、独立的、单线程的进程，它们运行在同一时刻，就像你可以同时进行网页浏览和文字处理。这些独立的进程可以通过所有常规的进程间通信渠道互相传递信息（信号、套接字、文件、管道等），如图 1.3 所示。有一个缺点是这种进程之间的通信通常设置复杂，或是速度较慢，或两者兼备，因为操作系统通常在进程间提供了大量的保护，以避免一个进程不小心修改了属于另一个进程的数据。另一个缺点是运行多个进程所需的固有的开销：启动进程需要时间，操作系统必须投入内部资源来管理进程，等等。

当然，也并不全是缺点：操作系统在线程间提供的附加保护操作和更高级别的通信机制，意味着可以比线程更容易地编写安全的并发代码。事实上，类似于为 Erlang 编程语言提供的环境，可使用进程作为重大作用并发的基本构造块。

使用独立的进程实现并发还有一个额外的优势——你可以通过网络连接的不同的机器上运行独立的进程。虽然这增加了通信成本，但在一个精心设计的系统上，它可能是一个提高并行可用行和提高性能的低成本方法。

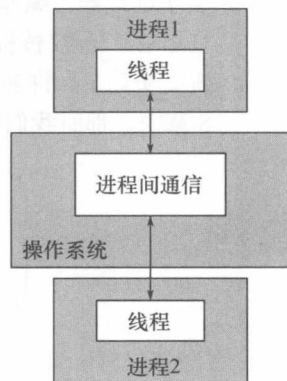


图 1.3 一对并发运行的进程之间的通信

2. 多线程并发

并发的另一个途径是在单个进程中运行多个线程。线程很像轻量级的进程：每个线程相互独立运行，且每个线程可以运行不同的指令序列。但进程中的所有线程都共享相

同的地址空间，并且从所有线程中访问到大部分数据——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。虽然通常可以在进程之间共享内存，但这难以建立并且通常难以管理，因为同一数据的内存地址在不同的进程中也不尽相同。图 1.4 显示了一个进程中的两个线程通过共享内存进行通信。

共享的地址空间，以及缺少线程间的数据保护，使得使用多线程相关的开销远小于使用多进程，因为操作系统有更少的簿记要做。但是，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当每个线程访问时所看到的数据是一致的。线程间数据共享可能会遇到的问题、所使用的工具以及为了避免问题而要遵循的准则在本书中都有涉及，特别是在第 3、4、5 和 8 章中。这些问题并非不能克服，只要在编写代码时适当地注意即可，但这却意味着必须对线程之间的通信作大量的思考。

相比于启动多个单线程进程并在其间进行通信，启动单一进程中的多线程并在其间进行通信的开销更低，这意味着若不考虑共享内存可能会带来的潜在问题，它是包括 C++ 在内的主流语言更青睐的并发途径。此外，C++ 标准没有为进程间通信提供任何原生支持，所以使用多进程的应用程序将不得不依赖平台相关的 API 来实现。因此，本书专门关注使用多线程的并发，并且之后提到并发均是假定通过使用多线程来实现的。

明确了什么是并发后，现在让我们来看看为什么要在应用程序中使用并发。

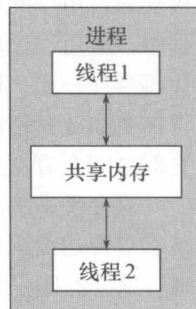


图 1.4 同一进程中的一对并发运行的线程之间的通信

1.2 为什么使用并发

在应用程序中使用并发的原因主要有两个：关注点分离和性能。事实上，我甚至可以说它们差不多是使用并发的唯一原因；当你观察得足够仔细时，一切其他因素都可以归结到这两者之一（或者可能是二者兼有，当然，除了像“我愿意”这样的原因之外）。

1.2.1 为了划分关注点而使用并发

在编写软件时，划分关注点总是个好主意。通过将相关的代码放在一起并将无关的代码分开，这种方法可以使你的程序更容易理解和测试，从而减少出错的可能性。你可以使用并发来分隔不同的功能区域，即使在这些不同功能区域的操作需要在同一时刻发生的情况下。如果不显式地使用并发，你要么被迫编写任务切换框架，要么在操作中主动地调用不相关的一段代码。

考虑一类带有用户界面的密集处理型应用程序，例如为台式计算机提供的 DVD 播放程序。这样一个应用程序基本上具备两套职能：它不仅要从光盘读取数据，解码图像和声音，并把它们及时输出至视频和音频硬件，从而实现 DVD 的无错播放；它还要接受来自用户的输入，例如当用户单击暂停或返回菜单甚至退出按键的情况。在单个线程中，应用程序须在回放期间定期检查用户的输入，于是将 DVD 回放代码和用户界面代码合在一起。通过使用多线程来分隔这些关注点，用户界面代码和 DVD 回放代码不再需要如此紧密地交织在一起。一个线程可以处理用户界面，另一个处理 DVD 回放，它们之间会有交互，例如用户点击暂停，但现在这些交互直接与眼前的任务有关。

这会带来响应性的错觉，因为用户界面线程通常可以立即响应用户的请求，即使在请求被传达给工作的线程，响应为简单地显示正忙的光标或请等待的消息的情况。类似地，独立的线程常被用于运行必须在后台连续运行的任务，例如在桌面搜索程序中监视文件系统的变化。以这种方式使用线程一般会使每个线程的逻辑更加简单，因为它们之间的交互可以被限制为清晰可辨的点，而不是到处散播不同任务的逻辑。

在这种情况下，线程的数量与 CPU 可用内核的数量无关，因为对线程的划分是基于概念上的设计而不是试图增加吞吐量。

1.2.2 为了性能而使用并发

多处理器系统已经存在了几十年，但直到最近，他们几乎只能在超级计算机、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芯片的设计，即在单个芯片上集成 2、4、16 或更多的处理器，从而达到比单核心更好的性能。因此，多核台式计算机，甚至多核嵌入式设备，现在越来越普遍。这些计算机的计算能力的提高不是源自使单一任务运行的更快，而是源自并行运行多个任务。在过去，程序员曾坐等他们的程序随着处理器的更新换代而变得更快，无需他们这边做出任何努力。但是现在，就像 Herb Sutter 所说的，“免费的午餐结束了¹”。如果软件想要利用日益增长的计算能力，它必须设计为并发运行多个任务。程序员因此必须留意，而且那些迄今都忽略并发的人们必须注意它并将其加入他们的工具箱中。

有两种方式为了性能使用并发。首先，也是最明显的，是将一个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，但它可能是一个相当复杂的过程，因为在各个部分之间可能存在很多的依赖。区别可能是在过程方面——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部分——或是在数据方面——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)**。

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>

容易受这种并行影响的算法常被称为易并行（**embarrassingly parallel**）。抛开你可能会尴尬地面对的很容易并行化的代码这一含义，这是一件好事情。我曾遇到过的关于此算法的别的术语是自然并行（**naturally parallel**）和便利并发（**conveniently concurrent**）。易并行算法具有良好的可扩展特性——随着可用硬件线程数量的提升，算法的并行性可以随之增加与之匹配。这样的一个是谚语“人多力量大”的完美体现。对于非易并行算法的那一部分，你可以将算法划分为一个固定（因而不可扩展）数量的并行任务。在线程之间划分任务的技巧涵盖在第 8 章中。

使用并发来提升性能的第二种方法是使用可用的并行方式来解决更大的问题。与此同时处理一个文件，不如酌情处理 2 个或 10 个或 20 个。虽然这实际上只是数据并行的一种应用，通过对多组数据同时执行相同的操作，但还是有不同的重点。处理一个数据块仍然需要同样的时间，但在相同的时间内却可以处理更多的数据。当然，这种方法也存在限制，且并非在所有情况下都是有益的，但是这种方法所带来的吞吐量提升可以让一些新玩意变得可能。例如，如果图片的各部分可以并行处理，就能提高视频处理的分辨率。

1.2.3 什么时候不使用并发

知道何时不使用并发与知道何时要使用它同等重要。基本上，不使用并发的唯一原因就是在收益比不上成本的时候。使用并发的代码在很多情况下难以理解，因此编写和维护的多线程代码就有直接的脑力成本，同时额外的复杂性也可能导致更多的错误。除非潜在的性能增益足够大或关注点分离得足够清晰，能抵消确保其正确所需的额外的开发时间以及与维护多线程代码相关的额外成本，否则不要使用并发。

同样地，性能增益可能不会如预期的那么大。在启动线程时存在固有的开销，因为操作系统必须分配相关的内核资源和堆栈空间，然后将新线程加入调度器中，所有这一切都要占用时间。如果在线程上运行的任务完成得很快，那么任务实际上占据的时间与启动线程的开销时间相比就显得微不足道，可能会导致应用程序的整体性能还不如通过产生线程直接执行该任务。

此外，线程是有限的资源。如果让太多的线程同时运行，则会消耗操作系统资源，并且使得操作系统整体上运行得更缓慢。不仅如此，运行太多的线程会耗尽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限制为 4GB 的扁平架构的 32 位进程来说，这尤其是个问题。如果每个线程都有一个 1MB 的堆栈（对于很多系统来说是典型的），那么 4096 个线程将会用尽所有地址空间，不再为代码、静态数据或者堆数据留有空间。虽然 64 位（或者更大）的系统不存在这种直接的地址空间限制，它们仍然只具备有限的资源：如果你运行太多的线程，最终会导致问题。尽管线程池（参见第 9 章）可以用来限制线程的数量，但这并不是灵丹妙药，它

们也有自己的问题。

如果客户端/服务器应用程序的服务器端为每一个链接启动一个独立的线程，对于少量的链接是可以正常工作的，但当同样的技术用于需要处理大量链接的高需求服务器时，就会因为启动太多线程而迅速耗尽系统资源。在这种场景下，谨慎地使用线程池可以提供优化的性能（参见第9章）。

最后，运行越多的线程，操作系统就需要做越多的上下文切换。每个上下文切换都需要耗费本可以花在有价值工作上的时间，所以在某些时候，增加一个额外的线程实际上会降低而不是提高应用程序的整体性能。为此，如果你试图得到系统的最佳性能，考虑可用的硬件并发（或缺乏之）并调整运行线程的数量是必需的。

为了性能优化而使用并发就像所有其他优化策略一样，它拥有极大提高应用程序性能的潜力，但它也可能使代码复杂化，使其更难理解和更容易出错。因此，只有对应用程序中的那些具有显著增益潜力的性能关键部分才值得这样做。当然，如果性能收益的潜力仅次于设计清晰或关注点分离，可能也值得使用多线程设计。

假设你已经决定确实要在应用程序中使用并发，无论是为了性能、关注点分离，还是因为“多线程星期一”，对于C++程序员来说意味着什么？

1.3 在C++中使用并发和多线程

通过多线程为并发提供标准化的支持对C++来说是新鲜事物。只有在即将到来的C++11标准中，你才能不依赖平台相关的扩展来编写多线程代码。为了理解新版本C++线程库中众多规则背后的基本原理，了解其历史是很重要的。

1.3.1 C++多线程历程

1998C++标准版不承认线程的存在，并且各种语言要素的操作效果都以顺序抽象机的形式编写。不仅如此，内存模型也没有被正式定义，所以对于1998C++标准，你没办法在缺少编译器相关扩展的情况下编写多线程应用程序。

当然，编译器供应商可以自由地向语言添加扩展，并且针对多线程的C API的流行——例如在POSIX C和Microsoft Windows API中的那些——导致很多C++编译器供应商通过各种平台相关的扩展来支持多线程。这种编译器支持普遍地受限於只允许使用该平台相应的C API以及确保该C++运行时库（例如异常处理机制的代码）在多线程存在的情况下运行。尽管极少有编译器供应商提供了一个正式的多线程感知内存模型，但编译器和处理器的实际表现也已经足够好，以至于大量的多线程的C++程序已被编写出来。

由于不满足于使用平台相关的C API来处理多线程，C++程序员曾期望他们的类库

提供面向对象的多线程工具。像 MFC 这样的应用程序框架，以及像 Boost 和 ACE 这样的 C++ 通用类库曾积累了多套 C++ 类，封装了下层的平台相关 API 并提供高级的多线程工具以简化任务。各类库的具体细节，特别是在启动新线程的方面，存在很大差异，但是这些类的总体构造存在很多共通之处。有一个为许多 C++ 类库共有的，同时也是为程序员提供很大便利的特别重要的设计，就是带锁的资源获得即初始化（**RAII, Resource Acquisition Is Initialization**）的习惯用法，来确保当退出相关作用域的时候互斥元被解锁。

许多情况下，现有的 C++ 编译器所提供的多线程支持，例如 Boost 和 ACE，综合了平台相关 API 以及平台无关类库的可用性，为编写多线程 C++ 代码提供一个坚实的基础，也因此大约有数百万行 C++ 代码作为多线程应用程序的一部分而被编写出来。但缺乏标准的支持，意味着存在缺少线程感知内存模型从而导致问题的场合，特别是对于那些试图通过使用处理器硬件能力来获取更高性能，或是编写跨平台代码，但是在不同平台之间编译器的实际表现存在差异。

1.3.2 新标准中的并发支持

所有这些都随着新的 C++11 标准的发布而改变了。不仅有了一个全新的线程感知内存模型，C++ 标准库也被扩展了，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）以及低级原子操作（参见第 5 章）的各个类。

新的 C++ 线程库很大程度上基于之前通过使用上文提到的 C++ 类库而积累的经验。特别地，Boost 线程库被用作新类库所基于的主要模型，很多类与 Boost 中的对应者共享命名和结构。在新标准演进的过程中，这是个双向流动，Boost 线程库也改变了自己，以便在多个方面匹配 C++ 标准，因此从 Boost 迁移过来的用户将会发现自己非常适应。

正如本章开篇提到的那样，对并发的支持仅仅是新 C++ 标准的变化之一，此外还存在很多对于编程语言自身的改善，可以使得程序员们的工作更便捷。这些内容虽然不在本书的论述范围之内，但是其中的一些变化对于线程库本身及其使用方式已经形成了直接的冲击。附录 A 对这些语言特性做了简要的介绍。

C++ 中对原子操作的直接支持，允许程序员编写具有确定语义的高效代码，而无需平台相关的汇编语言。这对于那些试图编写高效的、可移植代码的程序员们来说是一个真正的福利。不仅有编译器可以搞定平台的具体内容，还可以编写优化器来考虑操作的语义，从而让程序作为一个整体得到更好的优化。

1.3.3 C++ 线程库的效率

对于 C++ 整体以及包含低级工具的 C++ 类——特别是在新版 C++ 线程库里的那些，

参与高性能计算的开发者常常关注的一点就是效率。如果你正寻求极致的性能，那么理解与直接使用底层的低级工具相比，使用高级工具所带来的实现成本，是很重要的。这个成本就是抽象惩罚（abstraction penalty）。

C++标准委员会在整体设计C++标准库以及专门设计标准C++线程库的时候，就已经十分注重这一点了。其设计的目标之一就是在提供相同的工具时，通过直接使用低级API就几乎或完全得不到任何好处。因此该类库被设计为在大部分平台上都能高效实现（带有非常低的抽象惩罚）。

C++标准委员会的另一个目标，是确保C++能提供足够的低级工具给那些希望与硬件工作得更紧密的程序员，以获取终极性能。为了达到这个目的，伴随着新的内存模型，出现了一个全面的原子操作库，用于直接控制单个位、字节、线程间同步以及所有变化的可见性。这些原子类型和相应的操作现在可以在很多地方加以使用，而这些地方以前通常被开发者选择下放到平台相关的汇编语言中。使用了新的标准类型和操作的代码因而具有更佳的可移植性，并且更易于维护。

C++标准库也提供了更高级别的抽象和工具，它们使得编写多线程代码更简单和不易出错。有时候运用这些工具确实会带来性能成本，因为必须执行额外的代码。但是这种性能成本并不一定意味着更高的抽象惩罚；总体来看，这种性能成本并不比通过手工编写等效的函数而招致的成本更高，同时编译器可能会很好地内联大部分额外的代码。

在某些情况下，高级工具提供超出特定使用需求的额外功能。在大部分情况下这都不是问题，你没有为你不使用的部分买单。在罕见的情况下，这些未使用的功能会影响其他代码的性能。如果你更看重程序的性能，且代价过高，你可能最好是通过较低级别的工具来手工实现需要的功能。在绝大多数情况下，额外增加的复杂性和出错的几率远大于小小的性能提升所带来的潜在收益。即使有证据确实表明瓶颈出现在C++标准库的工具中，这也可能归咎于低劣的应用程序设计而非低劣的类库实现。例如，如果过多的线程竞争一个互斥元，这将会显著影响性能。与其试图在互斥操作上花掉一点点的时间，还不如重新构造应用程序以减少互斥元上的竞争来得划算。设计应用程序以减少竞争会在第8章中加以阐述。

在非常罕见的情况下，C++标准库不提供所需的性能或行为，这时则有必要运用特定的平台相关的工具。

1.3.4 平台相关的工具

虽然C++线程库为多线程和并发处理提供了颇为全面的工具，但是在所有的平台上，都会有些额外的平台相关工具。为了能方便地访问那些工具而又不用放弃使用标准C++线程库带来的好处，C++线程库中的类型可以提供一个`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。就其本质而言，任何使用

`native_handle()` 执行的操作是完全依赖于平台的，这也超出了本书（同时也是标准 C++ 库本身）的范围。

当然，在考虑使用平台相关的工具之前，明白标准库能够提供什么是很重要的，那么让我们通过一个例子来开始。

1.4 开始入门

好，现在你有一个很棒的与 C++11 兼容的编译器。接下来呢？一个多线程 C++ 程序是什么样子的？它看上去和其他所有 C++ 程序一样，通常是变量、类以及函数的组合。唯一真正的区别在于某些函数可以并发运行，所以你需要确保共享数据的并发访问是安全的，详见第 3 章。当然，为了并发地运行函数，必须使用特定的函数以及对对象来管理各个线程。

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。一个非常简单的在单线程中运行的 Hello, World 程序如下所示，当我们谈到多线程时，它可以作为一个基准。

```
#include <iostream>

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

清单 1.1 这个程序所做的一切就是将“Hello World”写进标准输出流。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信息。

清单 1.1 一个简单的 Hello,Concurrent World 程序

```
#include <iostream>
#include <thread>          ← ❶

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

int main()
{
    std::thread t(hello);   ← ❸
    t.join();               ← ❹
}
```