TURING 图灵程序设计丛书

Seven Concurrency Models in Seven Weeks

When Threads Unravel

七周七并发模型

【美】Paul Butcher 著

黄炎 译

人民邮电出版社

北 京

图书在版编目（CIP）数据

七周七并发模型 / (美) 布彻 (Butcher,P.) 著 ; 黄炎译. -- 北京 : 人民邮电出版社, 2015.4
（图灵程序设计丛书）
ISBN 978-7-115-38606-9

Ⅰ. ①七… Ⅱ. ①布… ②黄… Ⅲ. ①并行程序－程序设计 Ⅳ. ①TP311.11

中国版本图书馆CIP数据核字(2015)第038645号

内 容 提 要

并发编程近年逐渐热起来，Go 等并发语言也对并发编程提供了良好的支持，使得并发这个话题受到越来越多人的关注。本书延续了《七周七语言》的写作风格，通过以下七个精选的模型帮助读者了解并发领域的轮廓：线程与锁，函数式编程，Clojure，actor，通信顺序进程，数据级并行，Lambda 架构。书中每一章都设计成三天的阅读量。每天阅读结束都会有相关练习，巩固并扩展当天的知识。每一章均有复习，用于概括本章模型的优点和缺陷。

本书适合所有想了解并发的程序员。

◆ 著 [美] Paul Butcher
译 黄 炎
责任编辑 朱 巍
执行编辑 陈婷婷
责任印制 杨林杰

◆ 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 印刷

◆ 开本：800×1000 1/16
印张：15.25
字数：366千字 2015年 4 月第 1 版
印数：1－4 000册 2015年 4 月北京第 1 次印刷
著作权合同登记号 图字：01-2014-6027号

定价：49.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

译 者 序

作者认为写作本书非常不易，译者深以为然。翻译本书的主要困难源自于我们与作者在知识积累上的巨大差距，作者在前言中写道“我从 1989 年开始攻读博士学位，在并行计算和分布式计算的领域深造”，那年我只有一岁。我无法精通本书介绍的七个模型中的所有技术细节，因此没有自信确保全无差错，只能竭尽所能保证译文不会有大的偏差。

在此要向提供帮助的人们致以谢意。首先，感谢图灵的编辑老师，他们辛勤的工作完善了本书的每个细节；其次，要感谢我的父亲——大连海事大学的黄映辉教授，他为本书进行了三次审校，对字句进行了细致斟酌，大幅提升了本书的可读性；然后，要感谢我的挚友——工作于 Fox News Digital 的孙培源，他为本书进行了中英文的对照审校，帮助矫正了翻译过程中的很多谬误；还要感谢工作于喜马拉雅的柳飞提供的莫大帮助；最后，要感谢这个时代，让我们有机会能参与到这个伟大的丛书系列中。

本书介绍了七种并发模型，行文通俗易懂，有数量充足且设计精良的样例来帮助读者理解。读完本书，我最大的感受是世界变得更大了，想要学习的有趣的东西变得更多了。希望大家读完后也有类似有趣的体验。

用一个亲身经历的趣事来结束本序。

几年前，我去某软公司应聘，电话面试中与面试官有如下一番对话。

面试官：你了解多线程并发吗？

我：不了解，我之前做业务系统，多线程很大程度上都是委托给容器的……

面试官：我理解了。你不太熟悉并发是吗？

我：是的。

面试官：那我们还是来聊一聊并发吧。

祝大家线程安全。

黄炎

2014 年 12 月 31 日

推　荐　序

本书将讲述一个完整的故事。

将此作为一本书的首要定位似乎有点奇怪，但对我而言这很重要。我们曾回绝数十位申请撰写“七周系列丛书”的作者，他们认为只要将七个分散主题拼凑起来就是一本书，但这有违我们的初衷。

先前的《七周七语言：理解多种编程范型》[①]讲述了一个面向对象编程语言的故事，这是很适应当时的环境的。但在多核架构的驱动下，软件复杂度的增长和并发技术的发展所带来的压力，将函数式编程推到舞台之上，并对今后的编程方式有着深远的影响。Paul Butcher 是《七周七语言》最给力的审校者之一，相识四年后，我开始理解其中原因。

Paul 一直奋斗在将高可扩展的并发技术应用于实际业务系统的第一线。读过《七周七语言》后，对于他所处的日益重要但日趋复杂的问题领域，Paul 觉得可以从编程语言级别获得一些启发。几年后，Paul 表示要写一本自己的书。他解释道：尽管编程语言在整个故事中有着重要的作用，但也只触及了问题的表面。他要为读者讲述一个更完整的故事，为非专业人士介绍现代应用程序用以解决大型并行问题的扩展性良好的重要工具。

一开始我们是持怀疑态度的。这类书是很难写的——比起其他领域的书，这类书需要花费更长的时间，而且失败的几率很高——Paul 显然选择了一块难啃的骨头。作为一个团队，我们不断磨合前进，终于从最初的大纲中研磨出一个优秀的故事。随着书稿逐渐完成，我们更加自信于 Paul 的技术能力和攻关热情。现在，我们已经确信这是一本特别的书，而且恰逢其时。随着阅读的深入，我相信你也会同意这个观点。

当你在开篇阅读到“线程与锁”这种当今最广泛使用的并发解决方案时，可能会不以为然。不过你很快就会看到这种解决方案的不足之处，并开始思考如何解决。Paul 将引领你学习多种非常不同的技术，从一些社交平台使用的 Lambda 架构，到现今世界上许多最大最可靠的电信系统使用的 actor 模型。你会学到职业高手使用的一些语言，从 Java 到 Clojure，再到基于 Erlang 的闪亮新秀 Elixir。旅途中的每一步，Paul 都将从专业的角度为你剖析其中的玄妙和精彩。

在此，我诚意奉上《七周七并发模型》。希望你和我一样乐享其中。

Bruce A. Tate

icanmakeitbetter.com 网站 CTO，七周系列丛书主编

于美国德克萨斯州奥斯汀

---

① 本书中文版电子书在图灵社区有售：http://www.ituring.com.cn/book/829。——编者注

前　言

我从 1989 年开始攻读博士学位，在并行计算和分布式计算的领域深造，当时我便深信并发编程将成为主流。二十年后，我的观点终于得以验证——整个世界都在讨论多核以及如何发挥其优势。

学习并发不仅是为了利用多核来获得更好的性能。若正确使用并发，我们还能在程序的响应性、容错性、效率和简洁程度上获得大幅提升。

关于本书

本书延续了 Pragmatic Bookshelf 的七周系列丛书（《七周七语言》《七周七数据库》[①]《七周七网络框架》）的架构，通过七个精选的模型帮助读者了解并发领域的轮廓。这些模型中，一些已经成为主流，一些很快会成为主流，另一些虽难以成为主流，但在特定领域会威力无穷。当面对一个并发问题时，你可以借助本书准确选择合适的工具，这就是我的期望所在。

本书的每一章都设计成三天的阅读量。每天阅读结束都会有相关练习，巩固并扩展当天的知识。每一章均有复习，用于概括本章模型的优点和缺陷。

尽管有少量具有哲学意味的讨论，但本书还是侧重于实践。我强烈建议你在阅读样例时能亲手实践一下——没什么比代码更有说服力了。

本书未涉及的内容

本书不是语言参考手册。我们会使用一些较新的语言，例如 Elixir 和 Clojure，但本书关注的是并发而不是编程语言，所以不会深入介绍这些语言的具体特性。希望你通过上下文可以初步了解这些语言的主要特性，如果要对其深入探究以期充分理解，就得依靠自身的努力了。阅读本书时，如果手边开着浏览器可随时查阅语言参考手册，就会事半功倍。

本书不是安装配置手册。要运行本书的配套代码，就需要安装和运行相应工具——配套代码的 README 文件会给出一些提示，但还是要依靠你自己。本书所有的样例都采用主流工具编写，如果遇到困难，你可以在网络上找到许多帮助资料。

本书也不是面面俱到——无法囊括所有议题的每个细节。对于某些议题，本书会一笔带过或

① 本书中文版电子书在图灵社区有售：“http://www.ituring.com.cn/book/1369。——编者注

者根本不予讨论。在某些章节中，我会特意使用一些不规范的代码，目的是便于不熟悉该语言的读者来理解代码。如果你有意深入学习本书中的某种技术，建议阅读本书所提及的权威文献。

样例代码

本书讨论的所有样例都可以从本书的网站[①]下载。每个样例都包括源码和构建系统。对于每一种语言，本书都选用最通用的构建系统（Java 使用 Maven，Clojure 使用 Leiningen，Elixir 使用 Mix，Scala 使用 sbt，C 使用 GNU Make）。

大多数情况下，构建系统不仅会编译代码，而且会下载所需的额外依赖。sbt 和 Leiningen 甚至会下载对应版本的 Scala 和 Clojure 的编译器，所以你只需要下载并安装构建系统即可（在网络上可以找到详尽的安装步骤）。

不过第 7 章中使用的 C 代码是个特例，需要根据你的操作系统和显卡类型安装相应的 OpenCL 工具包（除非你使用的是 Mac，因为 Xcode 会搞定一切）。

给 IDE 用户的建议

本书使用的构建系统都在命令行下测试通过。如果你是成熟的 IDE 用户，一定知道如何将构建系统导入到 IDE 中——大多数 IDE 都会兼容 Maven，主流 IDE 也都有兼容 sbt 和 Leiningen 的插件。不过我没有在 IDE 中测试过，所以你与我一样使用命令行也许会容易一些。

给 Windows 用户的建议

所有的样例均在 OS X 和 Linux 上测试通过。理论上，它们也可以在 Windows 上运行良好，但我并没有验证过。

第 7 章中使用的 C 代码是个特例，其使用了 GNU Make 和 GCC。理论上是能被迁移到 Visual C++中，但我没有尝试过这种可能。

在线资源

本书中的样例都可以在本书网站上找到。如果你要提交勘误或者给出建议，也可以在网站上找到勘误表格和交流论坛。

Paul Butcher
Ten Tenths Consulting
paul@tententhsconsulting.com
2014 年 6 月于英国剑桥

① http://pragprog.com/book/pb7con

致　　谢

当我宣告决定写本书时，一个朋友提醒道：“你是不是已经忘记写第一本书时的艰辛了？”我当时一定是太天真，误认为写第二本书会容易一些。现在想来，如果不参与七周系列丛书，而是选择容易一些的题材，我的日子会好过很多。

若没有主编 Bruce Tate 和策划编辑 Jackie Carter 的鼎力相助，本书定然无法完成。感谢两位在本书写作过程中的大力支持，同时感谢 Dave 和 Andy 让我有幸参与到这个伟大的系列丛书中。

感谢许多朋友在成书早期提供的建议和反馈，他们是（排名不分先后）：Simon Hardy-Francis、Sam Halliday、Mike Smith、Neil Eccles、Matthew Rudy Jacobs、Joe Osborne、Dave Strauss、Derek Law、Frederick Cheung、Hugo Tyson、Paul Gregory、Stephen Spencer、Alex Nixon、Ben Coppin、Kit Smithers、Andrew Eacott、Freeland Abbott、James Aley、Matthew Wilson、Simon Dobson、Doug Orr、Jonas Bonér、Stu Halloway、Rich Morin、David Whittaker、Bo Rydberg、Jake Goulding、Ari Gold、Juan Manuel Gimeno Illa、Steve Bassett、Norberto Ortigoza、Luciano Ramalho、Siva Jayaraman、Shaun Parry、Joel VanderWerf。

感谢本书的技术审校者（排名不分先后）：Carlos Sessa、Danny Woods、Venkat Subramaniam、Simon Wood、Páidí Creed、Ian Roughley、Andrew Thomson、Andrew Haley、Sean Ellis、Geoffrey Clements、Loren Sands-Ramshaw、Paul Hudson。

最后，要向我的朋友、同事、家人致以谢忱和歉意。感谢你们的支持和鼓励，也请原谅过去 18 个月中我的种种偏执。

目　录

第 1 章 概　述

并发编程的概念并不新，却直到最近才火起来。一些编程语言，如Erlang、Haskell、Go、Scala、Clojure，也因对并发编程提供了良好的支持，而受到广泛关注。

并发编程复兴的主要驱动力来自于所谓的“多核危机”。正如摩尔定律[①]所预言的那样，芯片性能仍在不断提高，CPU的速度会继续提升，但计算机的发展方向已然转向多核化[②]。

Herb Sutter曾经说过：“免费午餐的时代已然终结。”[③]为了让代码运行得更快，单纯依靠更快的硬件已无法满足要求，我们需要利用多核，也就是发掘并行执行的潜力。

1.1　并发还是并行？

本书的主题是“并发”，那么又为何涉及了“并行”呢？虽然两者有所关联又常被混淆，但*并发*和*并行*的含义却是不同的。

一字之差也是差

*并发*程序含有多个逻辑上的独立执行块[④]，它们可以独立地并行执行，也可以串行执行。

*并行*程序解决问题的速度往往比串行程序快得多，因为其可以同时执行整个任务的多个部分。并行程序可能有多个独立执行块，也可能仅有一个。

我们还可以从另一种角度来看待并发和并行之间的差异：并发是问题域中的概念——程序需

---

① http://en.wikipedia.org/wiki/Moore%27s_law

② 作者在本章不断使用“core”“CPU”“processor”，译者在此尊重原文分别翻译成“核”“CPU”“处理器”。但译者认为此处指的都是广义的处理单元，而不是狭义的硬件。——译者注

③ http://www.gotw.ca/publications/concurrency-ddj.htm

④ 原文是“logical threads of control”，直译为“控制逻辑线程”，但在此语境下“控制”或“线程”指的并不是我们常见的“控制”和“线程”。为便于理解，在此将其译成“独立执行块”，这个概念来自于Google IO 2012的演讲“Go concurrency patterns”中引用的文档“Concurrency is not Parallelism”（http://tinyurl.com/goconcnotpar），其将这个概念称为“independently executing processes”。——译者注

要被设计成能够处理多个同时（或者几乎同时）发生的事件；而并行则是方法域中的概念——通过将问题中的多个部分并行执行，来加速解决问题。

引用Rob Pike的经典描述[①]：

并发是同一时间应对（dealing with）多件事情的能力；

并行是同一时间动手做（doing）多件事情的能力。

那么这本书讲述的是并发还是并行？

小乔爱问：

并发？并行？

我妻子是一位教师。与众多教师一样，她极其善于处理多个任务。她虽然每次只能做一件事，但可以同时处理多个任务。比如，在听一位学生朗读的时候，她可以暂停学生的朗读，以维持课堂秩序，或者回答学生的问题。这是并发，但并不并行（因为仅有她一个人，某一时刻只能进行一件事）。

但如果还有一位助教，则她们中一位可以聆听朗读，而同时另一位可以回答问题。这种方式既是并发，也是并行。

假设班级设计了自己的贺卡并要批量制作。一种方法是让每位学生制作五枚贺卡。这种方法是并行，而（从整体看）不是并发，因为这个过程整体来说只有一个任务。

超越串行编程模型

并发和并行的共同点就是它们比传统的串行编程模型更优秀。 本书将同时涵盖并发和并行（学究可能会给这本书起名为“七周七并发模型和并行模型”， 不过那样的话，封面会变得很难看）。

并发和并行经常被混淆的原因之一是，传统的“线程与锁”模型并没有显式支持并行。如果要用线程与锁模型为多核进行开发，唯一的选择就是写一个并发的程序，并行地运行在多核上。

然而，并发程序的执行通常是不确定的，它会随着事件时序的改变而给出不同的结果。对于真正的并发程序，不确定性是其与生俱来且伴随始终的属性。与之相反，并行程序可能是确定的——例如，要将数组中的每个数都加倍，一种做法是将数组分为两部分并把它们分别交给一个核处理，

① http://concur.rspace.googlecode.com/hg/talk/concur.html

这种做法的运行结果是确定的。用支持并行的编程语言可以写出并行程序，而不引入不确定性。

1.2 并行架构

人们通常认为并行等同于多核，但现代计算机在不同层次上都使用了并行技术。比如说，单核的运行速度现今仍能每年不断提升的原因是：单核包含的晶体管数量，如同摩尔定律预测的那样变得越来越多，而单核在位级和指令级两个层次上都能够并行地使用这些晶体管资源。

位级（bit-level）并行

为什么32位计算机的运行速度比8位计算机更快？因为并行。对于两个32位数的加法，8位计算机必须进行多次8位计算，而32位计算机可以一步完成，即并行地处理32位数的4字节。

计算机的发展经历了8位、16位、32位，现在正处于64位时代。然而由位升级带来的性能改善是存在瓶颈的，这也正是短期内我们无法步入128位时代的原因。

指令级（instruction-level）并行

现代CPU的并行度很高，其中使用的技术包括流水线、乱序执行和猜测执行等。

程序员通常可以不关心处理器内部并行的细节，因为尽管处理器内部的并行度很高，但是经过精心设计，从外部看上去所有处理都像是串行的。

而这种“看上去像串行”的设计逐渐变得不适用。处理器的设计者们为单核提升速度变得越来越困难。进入多核时代，我们必须面对的情况是：无论是表面上还是实质上，指令都不再串行执行了。我们将在2.2节的“内存可见性”部分展开讨论。

数据级（data）并行

数据级并行（也称为“单指令多数据”，SIMD）架构，可以并行地在大量数据上施加同一操作。这并不适合解决所有问题，但在适合的场景却可以大展身手。

图像处理就是一种适合进行数据级并行的场景。比如，为了增加图片亮度就需要增加每一个像素的亮度。现代GPU（图形处理器）也因图像处理的特点而演化成了极其强大的数据并行处理器。

任务级（task-level）并行

终于来到了大家所认为的并行形式——多处理器。从程序员的角度来看，多处理器架构最明显的分类特征是其内存模型（共享内存模型或分布式内存模型）。

对于共享内存的多处理器系统，每个处理器都能访问整个内存，处理器之间的通信主要通过内存进行，如图1-1所示。

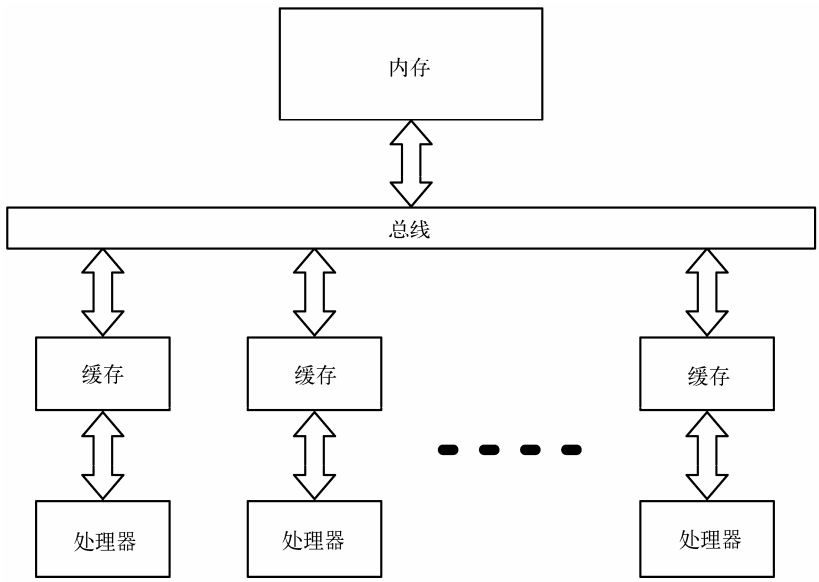

图1-1　共享内存的多处理器系统

对于分布式内存的多处理器系统，每个处理器都有自己的内存，处理器之间的通信主要通过网络进行，如图1-2所示。

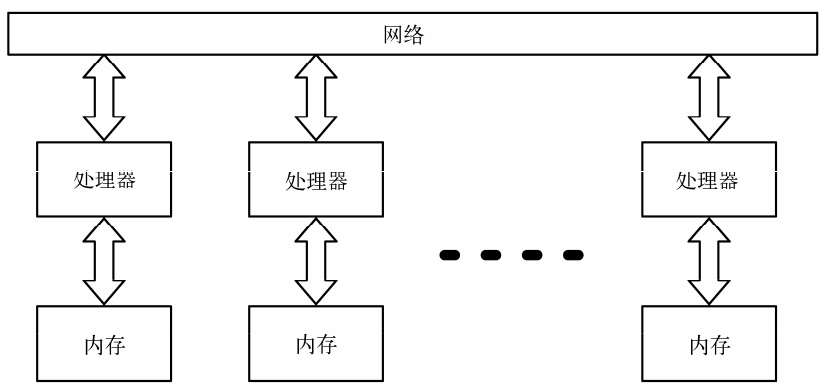

图1-2　分布式内存的多处理器系统

通过内存通信比通过网络通信更简单更快速，所以用共享内存编程往往更容易。然而，当处理器个数逐渐增多，共享内存就会遭遇性能瓶颈——此时不得不转向分布式内存。如果要开发一个容错系统，就要使用多台计算机以规避硬件故障对系统的影响，此时也必须借助于分布式内存。

1.3 并发：不只是多核

使用并发的目的，不仅仅是为了让程序并行运行从而发挥多核的优势。若正确使用并发，程序还将获得以下优点：及时响应、高效、容错、简单。

并发的世界，并发的软件

世界是并发的，为了与其有效地交互，软件也应是并发的。

手机可以同时播放音乐、上网浏览、响应触屏动作。我们在IDE中输入代码时，IDE正在后台悄悄检查代码语法。飞机上的系统也同时兼顾了好几件事情：监控传感器、在仪表盘上显示信息、执行指令、操纵飞行装置调整飞行姿态。

并发是系统及时响应的关键。比如，当文件下载可以在后台进行时，用户就不必一直盯着鼠标沙漏而烦心了。再比如，Web服务器可以并发地处理多个连接请求，一个慢请求不会影响服务器对其他请求的响应。

分布式的世界，分布式的软件

有时，我们要解决地理分布型问题。软件在非同步运行的多台计算机上分布式地运行，其本质是并发。

此外，分布式软件还具有容错性。我们可以将服务器一半部署在欧洲，另一半部署在美国，这样如果一个区域停电就不会造成软件整体不可用。下面就介绍容错性[①]。

不可预测的世界，容错性强的软件

软件有bug，程序会崩溃。即使存在完美的没有bug的程序，运行程序的硬件也可能出现故障。

为了增强软件的容错性，并发代码的关键是*独立性*和*故障检测*。独立性是指一个故障不会影响到故障任务以外的其他任务。故障检测是指当一个任务失败时（原因可能是任务崩溃、失去响应或硬件故障），需要通知负责故障处理的其他任务来处理。

串行程序的容错性远不如并发程序。

复杂的世界，简单的软件

如果曾经花费数小时纠结在一个难以诊断的多线程bug上，那你可能很难接受这个结论，但

---

① 作者在此处用到了两个词：fault-tolerant和resilient，中文都译为“容错性”，但两者略有区别。由于这种微小的区别不会影响对本书的理解，因此之后的译文不再区分两者，统一使用“容错性”以方便读者理解。——译者注

在选对编程语言和工具的情况下，比起串行的等价解决方案，一个并发的解决方案会更简洁清晰。

在处理现实世界的并发问题时，这个结论可以得到印证。用串行方案解决一个并发问题往往需要付出额外的代价，而且解决方案会晦涩难懂。如果解决方案有着与问题类似的并发结构，就会简单许多：我们不需要创建一个复杂的线程来处理问题中的多个任务，只需要用多个简单的线程分别处理不同的任务即可。

1.4　七个模型

本书精心挑选了七个模型来介绍并发与并行。

线程与锁：线程与锁模型有很多众所周知的不足，但仍是其他模型的技术基础，也是很多并发软件开发的首选。

函数式编程：函数式编程日渐重要的原因之一，是其对并发编程和并行编程提供了良好的支持。函数式编程消除了可变状态，所以从根本上是线程安全的，而且易于并行执行。

Clojure之道——分离标识与状态：编程语言Clojure是一种指令式编程和函数式编程的混搭方案，在两种编程方式上取得了微妙的平衡来发挥两者的优势。

actor：actor模型是一种适用性很广的并发编程模型，适用于共享内存模型和分布式内存模型，也适合解决地理分布型问题，能提供强大的容错性。

通信顺序进程（Communicating Sequential Processes，CSP）：表面上看，CSP模型与actor模型很相似，两者都基于消息传递。不过CSP模型侧重于传递信息的通道，而actor模型侧重于通道两端的实体，使用CSP模型的代码会带有明显不同的风格。

数据级并行：每个笔记本电脑里都藏着一台超级计算机——GPU。GPU利用了数据级并行，不仅可以快速进行图像处理，也可以用于更广阔的领域。如果要进行有限元分析、流体力学计算或其他的大量数字计算，GPU的性能将是不二选择。

Lambda架构：大数据时代的到来离不开并行——现在我们只需要增加计算资源，就能具有处理TB级数据的能力。Lambda架构综合了MapReduce和流式处理的特点，是一种可以处理多种大数据问题的架构。

以上每种模型都有各自的甜区[①]。请带着以下的问题来阅读之后的章节。

- 这个模型适用于解决并发问题、并行问题，还是两者皆可？
- 这个模型适用于哪种并行架构？
- 这个模型是否有利于我们写出容错性强的代码，或用于解决分布式问题的代码？

下一章将介绍第一个模型：线程与锁模型。

① 球类运动中球拍上最适合击球的区域。——译者注

第 2 章 线程与锁

线程与锁模型就像一辆福特T型车：驾驶它可以到达目的地，但与新的技术相比，它显得原始且难以驾驭，不可靠还有点儿危险。

抛开那些众所周知的缺点，线程与锁模型仍是开发并发软件的首选技术，它也支撑了本书将要介绍的其他技术。你可能不会直接用到这个模型，但也应该了解它是如何工作的。

2.1 简单粗暴

线程与锁模型其实是对底层硬件运行过程的形式化。这种形式化既是该模型最大的优点，也是它最大的缺点。

线程与锁模型非常简单直接，几乎所有编程语言都以某种形式对其提供了支持，且不对其使用方式加以限制。换句话说，对于不精通该模型的程序员，编程语言没有提供足够的帮助，使得程序容易出错且难以维护。

我们将借助Java语言来学习线程与锁模型，但所述内容也适用于其他语言。第一天，将学习Java的多线程代码、潜在的坑以及一些避免踩坑的原则。第二天，将进一步学习`java.util.concurrent`包提供的工具。第三天，学习一些由标准库提供的并发数据结构，尝试使用其解决一个现实问题。

最佳实践

第一天的学习将从Java提供的底层服务开始。现在的优秀代码很少直接使用底层服务，而是使用将在随后讨论的高层服务。要理解高层服务，我们必须先理解基础的底层服务，但请注意：不应在产品代码上直接使用`Thread`类等底层服务。

2.2　第一天：互斥和内存模型

如果你曾经接触过并发编程，那一定熟悉*互斥*这个概念——用锁保证某一时间仅有一个线程可以访问数据。你也肯定熟悉互斥带来的麻烦，比如说*竞态条件*和*死锁*（如果对此不熟悉也无妨，稍后都会介绍）。

我们会详细讨论实践中使用共享内存带来的一些问题，但首先需要关注更基础更重要的内容——内存模型。如果你认为竞态条件和死锁会导致一系列很奇怪的现象，那就对共享内存的诡异程度拭目以待吧。

想超越自我？让我们从创建一个线程开始。

创建线程

Java中，并发的基本单元是*线程*，可以将线程看作*控制流*（thread of control）。线程之间通过共享内存进行通信。

俗话说：一切编程皆始于“Hello, World!”。我们也不免俗地来个多线程版本：

ThreadsLocks/HelloWorld/src/main/java/com/paulbutcher/HelloWorld.java

```
public class HelloWorld {

  public static void main(String[] args) throws InterruptedException {
    Thread myThread = new Thread() {
        public void run() {
          System.out.println("Hello from new thread");
        }
      };
    myThread.start();
    Thread.yield();
    System.out.println("Hello from main thread");
    myThread.join();
  }
}
```

这段代码创建并启动了一个`Thread`实例。首先从`start()`开始，`myThread.run()`函数与`main()`函数的余下部分一起并发执行。最后`main`线程调用`join()`来等待`myThread`线程结束（即`run()`函数返回）。

运行这段代码的结果可能是

```
Hello from main thread
Hello from new thread
```

也可能是

```
Hello from new thread
Hello from main thread
```

究竟是哪种运行结果完全取决于哪个线程先执行println()（我的测试结果是各占50%）。多线程编程很难的原因之一就是运行结果可能依赖于时序，多次运行的结果并不稳定。

小乔爱问：

Thread.yield的作用？

在多线程版本的“Hello, World!”中我们使用了 Thread.yield()，根据相关的 Java 文档，其作用是：

> 通知调度器：当前线程想要让出对处理器的占用。

如果不调用 Thread.yield()，由于创建新线程要花费一些时间，那么 main 线程几乎肯定会先执行 println()（当然并不保证一定会如此——稍后我们将学到一个规律：并发编程中如果某事可能会发生，那么不论多艰难它一定会发生，而且可能发生在最不利的时刻）。

试将 Thread.yield()注释掉，看看会发生什么。如果换成 Thread.sleep(1)呢？

第一把锁

多个线程同时使用共享内存时，它们往往会“打成一片”。为避免如此，我们可以使用锁达到线程互斥的目的，即某一时间至多有一个线程能持有锁。

先创建两个线程，并使其交互：

ThreadsLocks/Counting/src/main/java/com/paulbutcher/Counting.java

```
public class Counting {
  public static void main(String[] args) throws InterruptedException {
    class Counter {
      private int count = 0;
      public void increment() { ++count; }
      public int getCount() { return count; }
    }
    final Counter counter = new Counter();
    class CountingThread extends Thread {
      public void run() {
        for(int x = 0; x < 10000; ++x)
          counter.increment();
      }
    }
    CountingThread t1 = new CountingThread();
    CountingThread t2 = new CountingThread();
    t1.start(); t2.start();
    t1.join(); t2.join();
    System.out.println(counter.getCount());
  }
}
```

这段代码创建了一个简单的counter类和两个线程，每个线程都调用counter.increment() 10 000次。这段代码看上去很简单但很脆弱。

运行这段代码，每次都将获得不同的结果。最后三次测试的结果是13850、11867和12616。产生这个结果的原因是两个线程使用counter.count对象时发生了竞态条件（即代码行为取决于各操作的时序）。

如果不能理解上面这段话，那让我们来考虑一下Java编译器是如何解释++count的。其字节码是：

```
getfield #2
iconst_1
iadd
putfield #2
```

即使你不熟悉JVM字节码，也可以揣测出这段代码的意图：getfield #2用于获取count的值，iconst_1和iadd将获得的值加1，putfield #2将更新的值写回count中。这就是通称的读–改–写（read-modify-write）模式。

假如两个线程同时调用increment()，线程1执行getfield #2，获得值42。在线程1执行其他动作之前，线程2也执行了getfield #2，获得值42。糟糕的是，现在两个线程都将获得的值加1，将43写回count中。结果count只被递增了一次，而不是两次。

竞态条件的解决方案是对count进行同步（synchronize）访问。一种方法是使用Java对象原生的内置锁（也被称为互斥锁（mutex）、管程（monitor）或临界区（critical section））来同步对increment()的调用：

ThreadsLocks/CountingFixed/src/main/java/com/paulbutcher/Counting.java

```
class Counter {
  private int count = 0;
➤ public synchronized void increment() { ++count; }
  public int getCount() { return count; }
}
```

线程进入increment()函数时，将获取Counter对象级别的锁，函数返回时将释放该锁。某一时间至多有一个线程可以执行函数体，其他线程调用函数时将被阻塞直到锁被释放（稍后我们将了解到：对于这种只涉及一个变量的互斥场景，使用java.util.concurrnet.atomic包是更好的选择）。

毋庸置疑，对于增加了同步功能的代码，每次执行都将得到正确结果20000。

但前路漫漫——代码中仍隐藏了一个bug，我们马上介绍其中的关窍。

诡异的内存

我们用一个小测试来开场，请猜测一下这段代码的输出：

ThreadsLocks/Puzzle/src/main/java/com/paulbutcher/Puzzle.java

```
Line 1 public class Puzzle {
   -   static boolean answerReady = false;
   -   static int answer = 0;
   -   static Thread t1 = new Thread() {
   5       public void run() {
   -         answer = 42;
   -         answerReady = true;
   -       }
   -     };
  10   static Thread t2 = new Thread() {
   -       public void run() {
   -         if (answerReady)
   -           System.out.println("The meaning of life is: " + answer);
   -         else
  15           System.out.println("I don't know the answer");
   -       }
   -     };
   -
   -   public static void main(String[] args) throws InterruptedException {
  20     t1.start(); t2.start();
   -     t1.join(); t2.join();
   -   }
   - }
```

如果你的第一反应是“竞态条件”，那么恭喜你答对了。根据线程执行的时序，这段代码的输出可能是*The meaning of life is XX*或者*I don't know the answer*。但不止于此，还有一种结果可能是：

```
The meaning of life is: 0
```

什么?! 当answerReady为true时answer可能为0吗？这仿佛是第6行和第7行在我们眼皮底下颠倒了执行顺序。

但是乱序执行是完全有可能发生的。以下所述均为事实：

- 编译器的静态优化可以打乱代码的执行顺序；
- JVM的动态优化也会打乱代码的执行顺序；
- 硬件可以通过乱序执行来优化其性能。

比乱序执行更糟糕的是，有时一个线程产生的修改可能对另一个线程不可见。如果将run()写成：

```
public void run() {
  while (!answerReady)
    Thread.sleep(100);
  System.out.println("The meaning of life is: " + answer);
}
```

answerReady可能不会变成true，代码运行后无法退出。

从直觉上来说，编译器、JVM、硬件都不应插手修改原本的代码逻辑。但是，近几年的运行效率提升，尤其是共享内存架构的运行效率提升，都仰仗于此类代码优化。因此我们也无法摆脱此类优化的副作用的影响。

显然，需要有标准来明确告诉我们，可能发生怎样的副作用，这就是Java内存模型。

内存可见性

Java内存模型定义了何时一个线程对内存的修改对另一个线程可见[①]。基本原则是，如果读线程和写线程不进行同步，就不能保证可见性。

我们已经见过一种同步的方法——就是通过获取对象的内置锁。其他的方法包括：开启一个线程并通过`join()`检查线程是否已经终止；使用`java.util.concurrent`包提供的工具。

很容易被忽略的一个重点是：两个线程都需要进行同步。只在其中一个线程进行同步是不够的，这正是之前竞态条件解决方案中潜藏bug的原因：除了`increment()`之外，`getCount()`方法也需要进行同步。否则，调用`getCount()`的线程可能获得一个失效的值（对于前面交互的两个线程，`getCount()`在`join()`之后被调用，因此是线程安全的。但这种设计为其他调用`getCounter()`的代码埋下了隐患）。

至此，我们讨论了竞态条件和内存可见性，这两类问题都可能让多线程程序运行结果出错。下面我们将介绍第三类问题：死锁。

多把锁

综上所述，很容易得出一个结论：让多线程代码安全运行的方法只能是让所有的方法都同步。然而，这也会带来问题。

首先这样做效率低下。如果每个方法都同步，大多数线程会频繁阻塞，使程序失去了并发的意义。问题不止于此，当使用多把锁时（Java中每一个对象都有自己的内置锁[②]），线程之间可能发生死锁。

我们将借助一个学术论文中经常使用的经典模型来诠释死锁——哲学家进餐问题。问题场景是五位哲学家围绕一个圆桌就坐，如图2-1所示，桌上摆着五支（不是五双）筷子。

哲学家的状态可能是“思考”或者“饥饿”。如果饥饿，哲学家将拿起他两边的筷子并进餐一段时间（当然，这些哲学家都是男性，女性在餐桌上会更优雅一些）。进餐结束，哲学家就会放回筷子。

---

① http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4

② 对不同对象的方法进行同步就会用到多把锁。——译者注

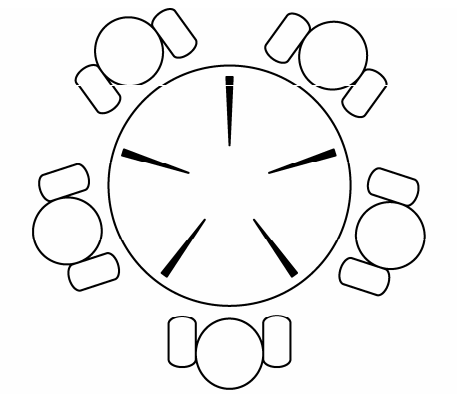

图2-1 哲学家进餐问题

下面的代码实现了哲学家的行为：

ThreadsLocks/DiningPhilosophers/src/main/java/com/paulbutcher/Philosopher.java

```
Line 1 class Philosopher extends Thread {
     -   private Chopstick left, right;
     -   private Random random;
     -
     5   public Philosopher(Chopstick left, Chopstick right) {
     -     this.left = left; this.right = right;
     -     random = new Random();
     -   }
     -
    10   public void run() {
     -     try {
     -       while(true) {
     -         Thread.sleep(random.nextInt(1000));     // 思考一段时间
     -         synchronized(left) {                    // 拿起筷子1
    15           synchronized(right) {                 // 拿起筷子2
     -             Thread.sleep(random.nextInt(1000)); // 进餐一段时间
     -           }
     -         }
     -       }
    20     } catch(InterruptedException e) {}
     -   }
     - }
```

第14、15行使用了另一种方式[①]来获取对象的内置锁：`synchronized(object)`。

在我的计算机上测试过五个哲学家实例，它们可以愉快地运行很久（最长时间是一周），直到某个时刻一切突然都停了下来。

稍加分析就知道发生了什么：如果所有哲学家同时决定进餐，都拿起左手边的筷子，那么就

① 第一种方式是在函数上使用synchronized关键字。——译者注

无法进行下去——所有人都持有一只筷子并等待右手边的人放下筷子。这时死锁就出现了。

一个线程想使用多把锁时，就需要考虑死锁的可能。幸运的是，有一个简单的规则可以避开死锁——总是按照一个全局的固定的顺序获取多把锁。

其中一种实现如下：

ThreadsLocks/DiningPhilosophersFixed/src/main/java/com/paulbutcher/Philosopher.java

```
class Philosopher extends Thread {
  private Chopstick first, second;
  private Random random;

  public Philosopher(Chopstick left, Chopstick right) {
    if(left.getId() < right.getId()) {
      first = left; second = right;
    } else {
      first = right; second = left;
    }
    random = new Random();
  }

  public void run() {
    try {
      while(true) {
        Thread.sleep(random.nextInt(1000));     // 思考一段时间
        synchronized(first) {                   // 拿起筷子1
          synchronized(second) {                // 拿起筷子2
            Thread.sleep(random.nextInt(1000)); // 进餐一段时间
          }
        }
      }
    } catch(InterruptedException e) {}
  }
}
```

我们不再按*左手边*和*右手边*的顺序拿起筷子，而是按照筷子的*编号*获得*编号*1和*编号*2的锁（我们并不关心编号的具体规则，只要保证编号是全局唯一且有序的）。毫无疑问，现在晚宴将一直愉快地进行下去而不会突然卡住。

不难想到，如果获取锁的代码写得比较集中，就有利于维护这个全局顺序。而对于规模较大的程序，使用锁的地方比较零散，各处都遵守这个顺序就变得不太实际。

小乔爱问：

可以用对象的散列值作为锁的全局顺序吗？

有一个常用的技巧是使用对象的散列值作为锁的全局顺序，类似于下面的代码：

```
if(System.identityHashCode(left) < System.identityHashCode(right)) {
  first = left; second = right;
```

```
    } else {
      first = right; second = left;
    }
```

这个技巧的好处是适用于所有 Java 对象，不用为锁对象专门定义并维护一个顺序。但是对象的散列值并不能保证唯一性（虽然几率很小，但对象的散列值确实可能重复）。我的建议是如果不是迫不得已，不要使用这个技巧。

来自外星方法的危害

规模较大的程序常用监听器模式（listener）来解耦模块。在这里，我们构造一个类从一个URL进行下载，并用ProgressListeners监听下载的进度。

ThreadsLocks/HttpDownload/src/main/java/com/paulbutcher/Downloader.java

```
class Downloader extends Thread {
  private InputStream in;
  private OutputStream out;
  private ArrayList<ProgressListener> listeners;

  public Downloader(URL url, String outputFilename) throws IOException {
    in = url.openConnection().getInputStream();
    out = new FileOutputStream(outputFilename);
    listeners = new ArrayList<ProgressListener>();
  }
  public synchronized void addListener(ProgressListener listener) {
    listeners.add(listener);
  }
  public synchronized void removeListener(ProgressListener listener) {
    listeners.remove(listener);
  }
  private synchronized void updateProgress(int n) {
    for (ProgressListener listener: listeners)
      listener.onProgress(n);
  }

  public void run() {
    int n = 0, total = 0;
    byte[] buffer = new byte[1024];

    try {
      while((n = in.read(buffer)) != -1) {
        out.write(buffer, 0, n);
        total += n;
        updateProgress(total);
      }
      out.flush();
    } catch (IOException e) { }
  }
}
```

addListener()、removeListener()和updateProgress()都是同步方法，多线程可以安全地使用这些方法。尽管这段代码仅使用了一把锁，但仍隐藏着一个死锁陷阱。

陷阱在于updateProgress()调用了一个外星方法——但对于这个外星方法一无所知。外星方法可以做任何事情，例如持有另外一把锁。这样一来，我们就在对加锁顺序一无所知的情况下使用了两把锁。就像前面提到的，这就有可能发生死锁。

唯一的解决思路是避免持有锁时调用外星方法。一种方法是在遍历之前对listeners进行保护性复制（defensive copy），再针对这份副本进行遍历：

ThreadsLocks/HttpDownloadFixed/src/main/java/com/paulbutcher/Downloader.java

```
private void updateProgress(int n) {
  ArrayList<ProgressListener> listenersCopy;
  synchronized(this) {
➤   listenersCopy = (ArrayList<ProgressListener>)listeners.clone();
  }
  for (ProgressListener listener: listenersCopy)
    listener.onProgress(n);
}
```

这是个一石多鸟的方法。不仅在调用外星方法时不用加锁，而且大大减少了代码持有锁的时间。长时间地持有锁将影响性能（降低了程序的并发度），也会增加死锁的可能。保护性复制也修复了与并发无关的另一个bug——修复后如果监听器在onProgress()中调用removeListener()，将不会影响到正在进行遍历的副本。

第一天总结

第一天的学习即将结束。我们通过代码学习了Java多线程的基础，在第二天的学习中将介绍标准库提供的更好的实现方式。

第一天我们学到了什么

本节介绍了如何创建线程，并用Java对象的内置锁实现互斥。还介绍了线程与锁模型带来的三个主要危害——竞态条件、死锁和内存可见性，并讨论了一些帮助我们避免危害的准则：

- ❑ 对共享变量的所有访问都需要同步化；
- ❑ 读线程和写线程都需要同步化；
- ❑ 按照约定的全局顺序来获取多把锁；
- ❑ 当持有锁时避免调用外星方法；
- ❑ 持有锁的时间应尽可能短。

第一天自习

查找

- ❑ 阅读William Pugh的网站“Java内存模型”。

- 自学JSR 133（Java内存模型）的FAQ。
- Java内存模型是如何保证对象初始化是线程安全的？是否必须通过加锁才能在线程之间安全地公开对象？
- 了解反模式“双重检查锁模式”（double-checked locking）以及为什么称其为反模式。

实践

- 对于哲学家进餐问题，用最开始有死锁隐患的代码做一些试验。尝试变更哲学家思考状态的时长、进餐状态的时长以及哲学家的人数。这些变量对于出现死锁的时机有什么影响？设想我们正在进行调试，那应该如何增大重现死锁的几率？
- （困难）编写一段程序，在不使用同步的前提下，模拟内存写操作的乱序执行。这个任务之所以有难度，是因为Java内存模型可能不会优化过于简单的例子，故找到这个优化场景比较困难。

2.3 第二天：超越内置锁

第一天我们学习了Java的`Thread`类和Java对象的内置锁。在过去的很长一段时间内，这几乎是Java对并发编程提供的所有支持。Java 5通过引入`java.util.concurrent`包改善了这个状况。今天我们将学习这种增强的锁机制。

内置锁虽然方便但限制很多：

- 一个线程因为等待内置锁而进入阻塞之后，就无法中断该线程了；
- 尝试获取内置锁时，无法设置超时；
- 获得内置锁，必须使用`synchronized`块。

```
synchronized(object) {
  «使用共享资源»
}
```

这种用法的限制是获取锁和释放锁的代码必须严格嵌在同一个方法中。另外，声明`synchronized`的函数其实只是个“语法糖”，其等价于将函数体按以下形式进行包装：

```
synchronized(this) {
  «函数体»
}
```

与`synchronized`不同，`ReentrantLock`提供了显式的`lock`和`unlock`方法，可以突破上述几个限制。

在深入学习之前，先来看一下`ReentrantLock`是如何替代`synchronized`工作的：

```
Lock lock = new ReentrantLock();
lock.lock();
try {
```

```
    «使用共享资源»
  } finally {
    lock.unlock();
  }
```

这段代码中，使用try ... finally是个很好的实践，无论被锁保护的代码发生了什么，都可以确保锁会被释放。

现在我们来看看ReentrantLock是如何突破限制的。

可中断的锁

使用内置锁时，由于阻塞的线程无法被中断，程序不可能从死锁中恢复。我们来看一个小例子，制造一个死锁并尝试中断线程。

ThreadsLocks/Uninterruptible/src/main/java/com/paulbutcher/Uninterruptible.java

```
public class Uninterruptible {

  public static void main(String[] args) throws InterruptedException {

    final Object o1 = new Object(); final Object o2 = new Object();

    Thread t1 = new Thread() {
      public void run() {
        try {
          synchronized(o1) {
            Thread.sleep(1000);
            synchronized(o2) {}
          }
        } catch (InterruptedException e) { System.out.println("t1 interrupted"); }
      }
    };

    Thread t2 = new Thread() {
      public void run() {
        try {
          synchronized(o2) {
            Thread.sleep(1000);
            synchronized(o1) {}
          }
        } catch (InterruptedException e) { System.out.println("t2 interrupted"); }
      }
    };

    t1.start(); t2.start();
    Thread.sleep(2000);
    t1.interrupt(); t2.interrupt();
    t1.join(); t2.join();
  }
}
```

这段程序将永远死锁下去——跳出死锁唯一的方法是终止JVM的运行。

小乔爱问：

真的没办法终止死锁的线程吗？

你可能认为肯定有某种方法来终止一个死锁线程。遗憾的是确实没有。所有这类方法都被证明有缺陷而不推荐使用。[a]

终止线程的最终手段是让run()函数返回（可能是通过抛出InterruptedException）。不过，如果你的线程由于等待内置锁而陷入死锁，且不能中断其等待锁的状态，那么要终止死锁线程就只剩下终止JVM运行这条路了。

a. http://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

不过还是有办法解决这个限制的。我们可以用ReentrantLock替代内置锁，使用它的lockInterruptibly()方法：

ThreadsLocks/Interruptible/src/main/java/com/paulbutcher/Interruptible.java

```
final ReentrantLock l1 = new ReentrantLock();
final ReentrantLock l2 = new ReentrantLock();

Thread t1 = new Thread() {
  public void run() {
    try {
➤     l1.lockInterruptibly();
      Thread.sleep(1000);
➤     l2.lockInterruptibly();
    } catch (InterruptedException e) { System.out.println("t1 interrupted"); }
  }
};
```

这一次Thread.interrupt()可以让线程终止。代码的确比之前稍微复杂一点，这就算是为中断死锁线程付出的一点代价吧。

超时

ReentrantLock突破了内置锁的另一个限制：可以为获取锁的操作设置超时时间。利用这个功能，我们可以通过另一种方法来解决第一天的哲学家进餐问题。

下面是修改后的Philosopher类，拿起两根筷子失败时会超时：

ThreadsLocks/DiningPhilosophersTimeout/src/main/java/com/paulbutcher/Philosopher.java

```
class Philosopher extends Thread {
  private ReentrantLock leftChopstick, rightChopstick;
```